DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master′s Thesis in Informatics

Enhancing Edge Orchestration Flexibility through Addons

Mahmoud Elkodary

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Enhancing Edge Orchestration Flexibility through Addons

Author:	Mahmoud Elkodary
Advisor:	Dr. Ph.D Nitinder Mohan, Giovanni Bartolomeo M.Sc
Supervisor:	Prof. Dr.-Ing. Jörg Ott
Submission Date:	15.09.2024

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.09.2024 Mahmoud Elkodary

Acknowledgments

I want to express my deepest gratitude to my family for their support during my master's program. The encouragement, faith in my abilities, and financial assistance they provided made it possible for me to pursue the master's program at TUM.

I would also like to thank my advisors for their guidance and continuous support in this thesis. Also, I want to thank Prof. Dr. Jörg Ott for the opportunity to conduct my thesis at the Department of Connected Mobility.

Abstract

With the advancements in edge computing, orchestration tools are required to offer both adaptability and high performance at the edge. Traditional orchestration systems, like Kubernetes, were primarily designed for stable, high-bandwidth, centralized cloud environments and often needed customized solutions to meet edge the dynamic and diverse requirements of the edge environment. This thesis focuses on establishing mechanisms for enhancing the flexibility of edge orchestration platforms.

Three significant enhancements were introduced to accomplish this objective: addons system, a hooks system, and custom resources. The addons system facilitates the seamless integration of new features through addons, which enhances extensibility. Moreover, the hooks system enables services to observe and respond to changes in the life-cycle system entities. Finally, "Custom Resources," inspired by Kubernetes, allows custom resource objects to be customized to specific functionality. This work presents several enhancements to create a flexible system.

Contents

1 Introduction

It is estimated that by 2025, the number of generated data will grow to 181 zettabytes [1]. It is only natural that they continue to grow, given the rapid increase in the number of devices connected to the internet. Also, it is reported that Internet of Things (IoT) alone will generate more than 79 zettabytes [2]. The data generated needs to be processed and stored considering the cost incurred, not only the computation or the storage costs but also the network bandwidth and transfer costs. Consider also the situation where applications need to be processed in real-time; as such, another concern is not only cost but also latency delay. This dilemma or challenge led to the rise of edge computing, where computing resources are placed near the devices generating the data [3]. This technique drastically reduces network latency since the distance the data needs to travel is relatively short [4].

1.1 Problem Statement

The advancement of edge computing has increased the demand for efficient and flexible orchestration tools to tackle the challenges presented by edge environments, such as delay sensitivity and heterogeneity of devices [5]. Traditional orchestration platforms often struggle with handling workflows on the edge due to the dynamic and heterogeneous nature of edge environments. Moreover, adding new features to accommodate the requirements of edge environments will probably bloat the system, hindering the ability to deploy workflows and, as a result, impacting the performance of edge applications.

Flexibility in Software Systems

This thesis attempts to enhance the flexibility of edge orchestration tools. However, what does the flexibility of software systems mean? And how can it be measured? In this thesis, we define it as the easiness of a system to grow to handle new sets of requirements, and we measure by how **modular** and **extensible** the system is without impacting its **performance**.

1.2 Motivation

The primary motivation for this thesis is to present mechanisms for extending edge orchestration tools. Given the unique challenges of edge computing, such as limited resources, intermittent connectivity, and diverse hardware platforms [5], orchestration tools must dynamically adapt to evolving requirements without becoming bloated with features.

Software bloat, as defined in [6], is code that is unnecessary when running an application. For instance, there are certain workflows of an application where some piece of code is not being executed, and the system would benefit a lot in terms of performance if this piece of code is removed, specifically when running this type of workload, but re-added when other workloads that need it are running. To illustrate, consider this code similar to a browser extension that can be enabled when needed and disabled when not in use, keeping the application lightweight and improving its performance.

In the context of edge computing, performance is critical. Therefore, avoiding software bloat is vital. Hence, orchestration tools must be designed to integrate new functionalities seamlessly and maintain a lean and efficient codebase in an effort to prevent performance degradation and ensure that the system remains adaptable to the changing demands of edge environments.

1.3 Contribution

This thesis aims to enhance Oakestra's, an orchestration tool designed for edge computing, flexibility by transforming it into an extendable system. The primary contributions of this work are:

- **Addons System**: Developed to enable developers to install new features or replace core components through addons.
- **Hooks System**: Implemented to allow services to listen to changes in system entities, providing dynamic response capabilities.
- **Custom Resources**: Allows for creating custom resources while handling its storage and providing means for manipulating these resources.

2 Background

2.1 Edge Computing

With each task submitted to a cloud computing server, there is typically some acceptable time range for its completion. One of the main challenges traditional cloud computing setups face is high network latency [7]. Network latency is characterized by the Round-Trip Time (RTT), which is the total time it takes for a request to reach some server until it arrives back again at the origin [8]. For a complex task where its execution could span many hours, a few seconds of delay is not critical. However, for applications such as online video games, delay is unacceptable. One of the reasons for the network delay could be that the network is congested [9]. However, the travel distance and the number of hops a request has to make are also factors in the delay. [10] performed an experiment showing the relationship between network load and the number of hops a packet travels and their impact on network latency. Their research explains that when the network load is under 50%, the latency is as low as 250ms regardless of the number of hops a packet has to go through. However, when the network becomes congested, typically above 60%, the impact of the number of hops becomes more pronounced. It is clear that network congestion is the most significant factor influencing latency. However, even a few hops can substantially increase latency if the network is congested.

Edge computing brings compute resources closer to the devices generating the data [3]; it is an extension of cloud computing, and it has emerged to meet the challenges presented by edge environments [4]. The shortcomings of cloud computing become apparent when managing the increasing number of connected devices and handling the data they produce, especially when there are latency and bandwidth limitations [4].

2.1.1 Edge Computing Principles

Edge computing involves placing computing resources near the devices generating the data [3]. Devices generating the data for processing could be sensors, appliances, actuators, or other devices connecting to a network to share data [11]. Such placement aims to reduce latency and bandwidth usage [4, 12]. Unlike traditional cloud computing paradigms, which rely on centralized data centers, edge computing distributes the computational load across a network of devices and edge servers near the data generation points; this approach enhances the efficiency and responsiveness of applications and reduces the burden on cloud infrastructure by offloading tasks to the edge [5]. Consequently, applications requiring low latency, such as online gaming or real-time analytics, could gain from this reduced network latency.

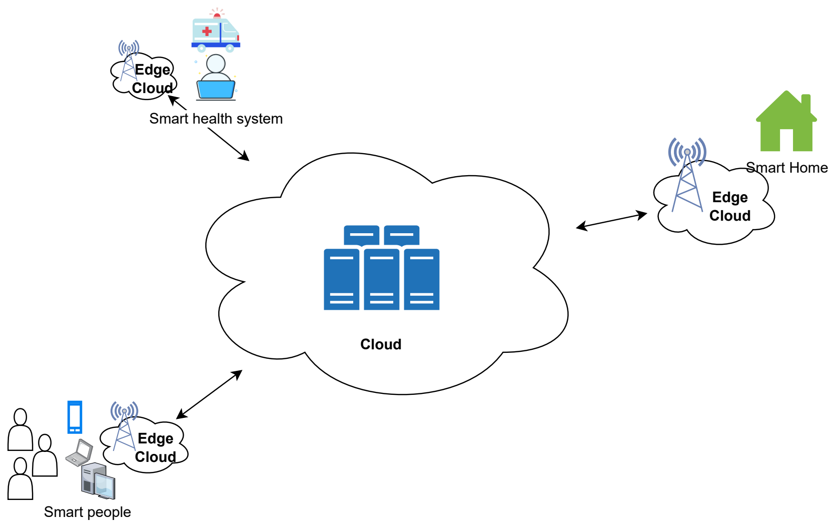

Figure 2.1: Edge computing applications [5].

Offloading

Offloading is a technique designed to offload computationally heavy and time-sensitive tasks to well-equipped edge servers or cloud servers for processing. This method aims at lowering the time it takes to process tasks. Key challenges in offloading involve deciding whether to offload a task, how much of the task needs to be offloaded, and to which server the task should be directed. Typically, offloading involves three types of decisions [13]:

- **Local Execution**: The entire computation task is handled on the device. This approach might be chosen if the computational capabilities of edge servers are not accessible or if a poor wireless connection would lead to significant transmission delays.
- **Full Offloading**: An edge server transfers and processes the complete computation task.
- **Partial Offloading**: A portion of the computation task is processed on the device, while the remaining part is offloaded to an edge server.

2.1.2 Kubernetes on the Edge

Orchestration tools oversee and automate deployment, scaling, and operation across various environments of applications. Platforms like Kubernetes are widely embraced for their ability to handle container applications [14], but they may encounter challenges when orchestrating at the edge [15].

Challenges

The default configuration of Kubernetes presents several limitations when applied to edge computing. [16] explains the following limitations:

- **Centralized control plane**: Kubernetes violates the decentralized nature of edge-computing by having a centralized control plane. In edge environments, computing resources are usually distributed across various geographic locations and may be connected to unreliable networks. However, Kubernetes assumes and relies on an infrastructure with reliable and low-latency network. As a result, computing resources may fail due to delays in the network.
- **Inadequate Scheduling for Edge**: Kubernetes default scheduler only relies on CPU and Memory metrics for scheduling workloads to nodes. However, in edge environments, there are other metrics to consider, such as network latency and bandwidth. In addition, there isn't much support for offloading workloads to the cloud from the edge and vice versa. This leads to sub-optimal scheduling decisions.
- **Topology Awareness**: Given that the physical location of worker nodes in an edge environment is critical, workloads that need real-time data processing or low-latency communication should ideally be placed on nodes geographically close to the data source. For example, nodes that process data from sensors would benefit from low latency and improved responsiveness when placed near the sensor. However, since K8s lacks inherent topology awareness, it may assign workloads to distant nodes. As a result, this leads to increased latency and higher data transfer costs.
- **Resource Constraints**: Computing resources in edge environments are typically limited in their computing power compared to resources found in traditional cloud computing providers. Moreover, K8s worker nodes run **K-proxy** and **Kubelet** that are resource intensive and possibly consume a significant portion of the limited resources of such devices, leading to fewer resources available for the actual workload.

Extending Kubernetes

Custom solutions for Kubernetes were designed in an effort to tackle the challenges faced when utilizing Kubernetes on edge. KubeEdge [17] for instance, provides custom components alongside the core components of K8s to improve Kubernetes performance on the edge. KubeEdge also advertises that edge devices can autonomously continue operation even when disconnected from the cloud. Other solutions replaced the default scheduler found in K8s with a custom one incorporating new metrics, such as network latency and bandwidth. This showed to have improved the efficiency of edge deployments significantly [16].

Although some of the solutions presented improved the reliability of K8s on the edge, they still don't fully handle the requirements of edge computing. KubeEdge lacks support for provisioning different edge computing models. For instance, there isn't much support for

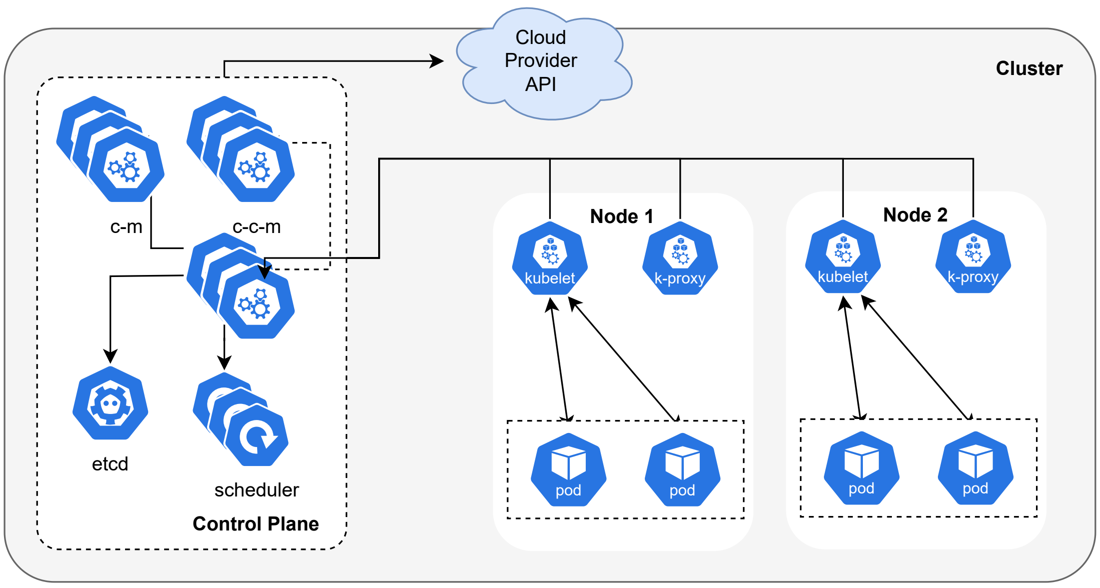

Figure 2.2: Cluster architecture of K8s [18].

offloading to the cloud from the edge or vice versa [16]. Nevertheless, it is still possible to derive from these partial solutions offered by the different custom solutions a unified one.

The default Kubernetes setup wasn't well suited for orchestrating resources at the edge [15, 16]. However, with extensions or custom solutions, K8s efficiency at handling edge orchestration was significantly improved.

2.2 Oakestra Overview

Oakestra is a lightweight orchestration tool specifically designed for the unique needs of edge computing environments [19]. Unlike traditional orchestration systems, Oakestra is built to handle edge devices of a dynamic and heterogeneous nature. Moreover, it facilitates the efficient deployment and management of applications across these decentralized edge nodes, ensuring optimal resource utilization and performance [19].

As seen in Figure 2.3, there are three main systems of Oakestra: **Worker nodes**, **Cluster Orchestrator**, and **Root Orchestrator**. The Root Orchestrator is the centralized control plane of Oakestra [19]. It is responsible for managing clusters. Moreover, the Cluster Orchestrator's responsibility is to manage the worker nodes [19]. Finally, Worker Nodes are the edge resources responsible for running the deployed services [19].

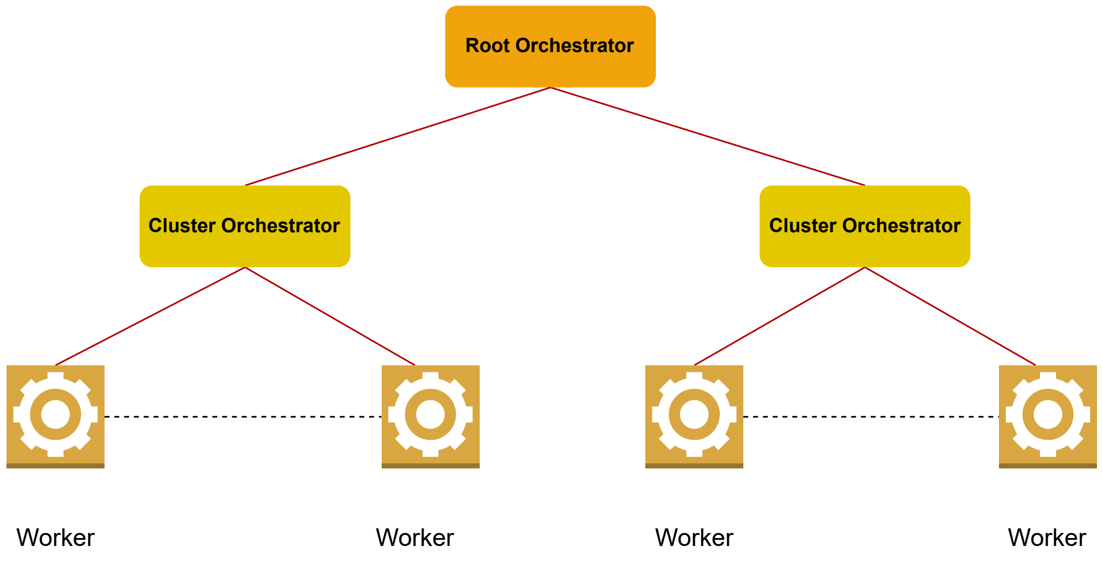

Figure 2.3: Overview of the 3 layers of Oakestra design architecture [19].

2.3 Related Work - Extendability in Software Systems

Extendability in software systems means their capability to grow and support expansion [20]. This involves designing systems in a way that allows for the integration of features and functionalities without requiring significant modifications to the core structure. The key benefits of extendability include prolonging the lifespan of a system as it can evolve alongside the evolving user requirements and technological advancements. It also makes updates simpler for developers.

Extending and adapting systems to new requirements is challenging for many software platforms, especially when trying to minimize software bloat. Extendible software systems should be designed with flexibility in mind, such that developers can add new functionalities and features without risking the performance of a system. This capability is critical in dynamic environments such as edge computing, where the diversity of devices, variability of workloads, and the need for real-time processing necessitate a highly flexible approach.

This section explores various approaches and examples of how different software systems achieve extensibility. By examining these related works, one can identify best practices, common patterns, and innovative techniques that can inform the enhancements made to edge computing platforms, such as Oakestra. Understanding these methods provides mechanisms that help implement effective and robust extensions in orchestration tools.

2.3.1 Plugin Based Systems

One approach to achieving extensibility in software systems is through plugins. [21] defines a method for combating the increasing bloat in software systems by utilizing a plugin architecture. The plugin architecture is designed to support dynamic extensibility. It enables applications to be extended at runtime by dynamically loading modules or classes unknown during the application compilation. Thus, the load time and memory usage are reduced. In addition, it allows for independent development and deployment of additional features.

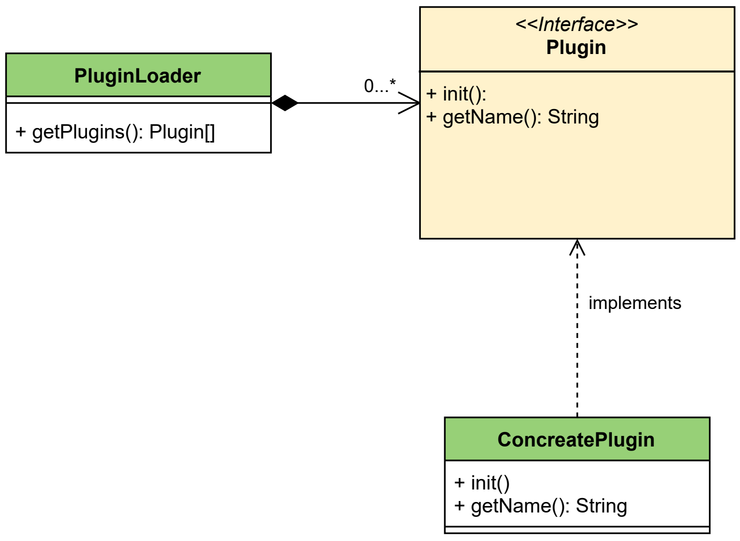

Figure 2.4: Lightweight plugin architecture concept in UML [21].

Figure 2.4 represents a plugin architecture where **PluginLoader** dynamically loads and manages plugins defined by the **PluginInterface**. Furthermore, the **ConcretePlugin** implements the interface by providing a concrete functionality. This design promotes extensibility and allows adding new features without modifying the core system.

Microkernel

Microkernel is a practical approach to extending software systems [22]. The microkernel or plugin architecture is designed to easily add new functionalities without impacting the core system. The microkernel architecture comprises two main components: Core System and Plugin Modules. Figure 2.5 illustrates an example of a microkernel plugin-based application. It shows that for a claims processing application in an insurance company, the complexity of varying jurisdictional rules (e.g., different states in the US) poses some challenges. Traditional rules engines can become convoluted and complex to manage, with changes in one rule potentially impacting others. The microkernel architecture mitigates these issues by maintaining a core system that handles basic claims processing logic, while independent plugin modules manage jurisdiction-specific rules. This separation allows jurisdiction-specific regulations to be added, removed, or modified without affecting the core system or other plugins, thus enhancing flexibility and maintainability.

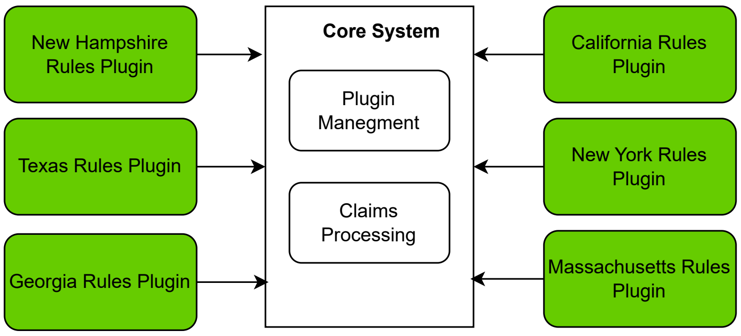

Figure 2.5: An example for a microkernel-based plugin approach [22].

Component Based plugins

[23] presents a similar approach and describes a platform that can be composed entirely of plugins. It provides a comprehensive framework for extending software systems using plugins. This approach emphasizes the need for controlled, restricted, and determined system extensions to maintain system integrity and performance. It explains a notion called "Extension Slot, " specifying how a plugin can contribute to an extension slot. This approach is illustrated in figure 2.6.

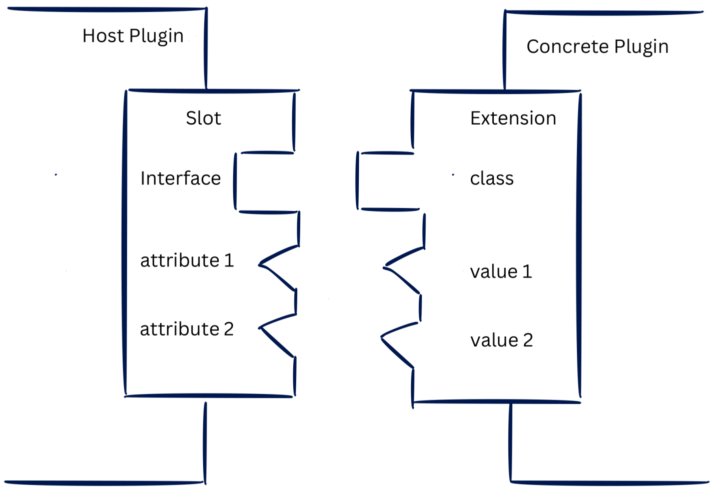

Figure 2.6: Depicting how host plugins and concrete plugins interact [23]

2.3.2 Systems with Extensions

Web Browsers

Modern browsers, such as Google Chrome, have robust support for extensions, which gives developers the capability to create powerful tools that interact with web content and the browser. Extensions can range from ad blockers and grammar checkers to tools for organizing bookmarks and automating repetitive tasks [24]. Unlike previous examples, where the whole application architecture is based on plugins, web browsers typically don't follow this architecture. Instead, they only provide means of adding new functionality via extensions.

Google Chrome is one of the most widely used browsers [25], hosting a massive ecosystem of extensions available through the Chrome Web Store. Chrome extensions are designed to be easy to install and manage to enhance the users' browsing experience. Figure 2.7 depicts the multiple components of a chrome extension [26, 27]:

- **Background Script**: Chrome runs the background script on a separate process alongside the browser.
- **Content Script**: A JavaScript file runs on a web page. Chrome injects the content script on every tab window. The content script typically interacts with the Document Object Model (DOM) of the page. Content scripts can interact with the background script via chrome's Inter-process Communication (IPC).

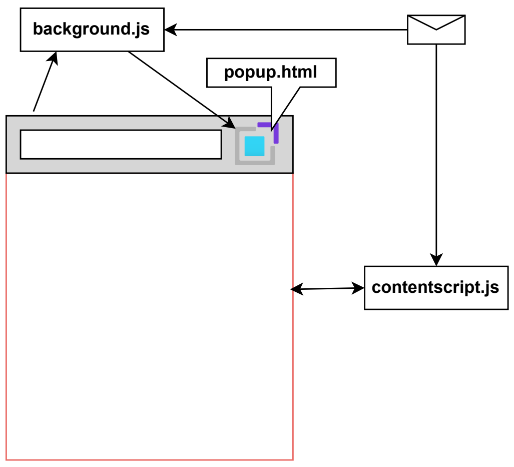

Figure 2.7: Chrome Extension components [28].

Docker Extensions

Docker extensions offer a robust mechanism to extend the capabilities of Docker Desktop. Packaged as Docker images and distributed via Docker Hub, users can install these extensions through the Docker Dashboard or Docker Extensions Command Line Interface (CLI) [29], offering additional functionalities and tools that integrate with Docker Desktop.

Figure 2.8 shows the optional components that compose a docker extension:

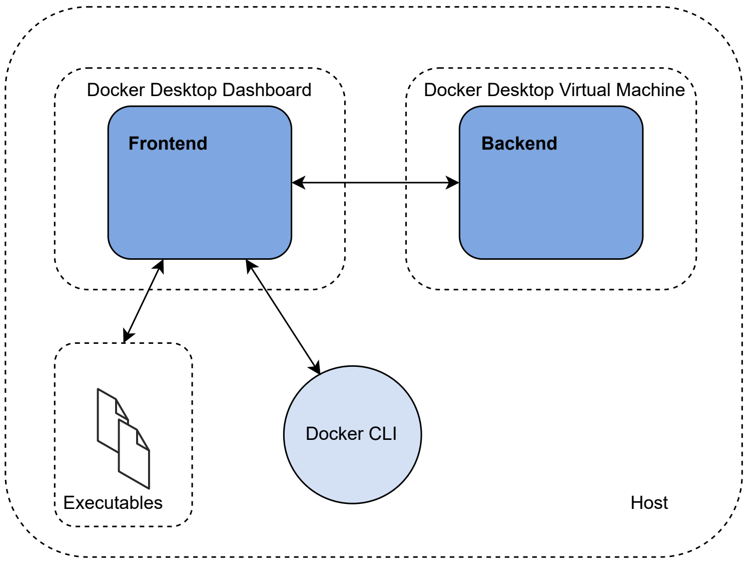

Figure 2.8: Docker extension architecture design [29].

- **Frontend**: Displayed as a tab within the Docker Desktop Dashboard.
- **Backend**: Containerized services running within the Docker Desktop VM.
- **Executables**: Shell scripts or binaries copied to the host when the extension is installed.

Extensions in docker are defined using a 'manifest.json' file that specifies how each component is configured and integrated. Not all extensions necessarily need all the elements; for instance, an extension might not need the frontend component. Consequently, this extension would not need to define the frontend component in the 'manifest.json.'

2.3.3 Other Extendability Mechanisms

Hooks

"Hooks" is a technique for decoupling system components. The main idea is to provide a means for exposing lifecycle events happening in the system. This enables components within a system to collaborate in a decoupled and more modular approach. Lifecycle refers to events that occur when an object is created until its deletion. For example, the Open Container Initiative (OCI) standardized container management by including the definition of lifecycle events for containers. These events describe the timeline where a container is created until it is stopped [30].

Example Docker [31] is a container management tool whose containers follow the OCI standard. When a container is created via docker, it uses hooks to listen to the container creation event, allowing it to perform additional actions such as attaching networks to the container.

Custom Resources

Kubernetes custom resources extend the Kubernetes Application Programming Interface (API), allowing users to add new types of API objects specific to their needs [32]. This flexibility is essential for customizing Kubernetes clusters beyond their default capabilities. Custom resources can be dynamically registered, modified, and managed independently of the core Kubernetes installation, making them highly modular. A custom resource is a user-defined extension to the Kubernetes API. It represents additional types of objects that are not included in a standard Kubernetes setup. For instance, while built-in resources like Pods are always available, custom resources allow users to define and manage their types of objects tailored to their specific use cases.

Custom resources, when used alone, help store and retrieve structured data. However, when paired with custom controllers, they create a powerful declarative API. In Kubernetes, a declarative API allows users to define the desired state of a resource while the Kubernetes controller continuously works to keep this state consistent. Custom controllers, which can be deployed and updated independently of the cluster, monitor custom resources and ensure that the system's current state is the same as the user-defined desired state. This separation of concerns is a crucial feature of the Kubernetes declarative model, contrasting with imperative approaches where users directly instruct the system on the actions to perform [32].

Example One good example of extending Kubernetes using Custom Resource Definitions (CRD) is the development of KubeShare. KubeShare is a framework designed to effectively manage GPUs as shared resources within a Kubernetes cluster [33]. By default, Kubernetes only recognizes CPUs and memory as schedulable resources, which poses challenges when dealing with GPUs, especially in scenarios where GPUs need to be shared among multiple containers. KubeShare addresses these challenges by introducing a new custom resource called **SharePod**. This CRD allows GPUs to be fractionally allocated and shared among containers, which isn't possible with Kubernetes' native capabilities. With KubeShare, GPUs are treated as first-class resources, meaning they can be explicitly identified, allocated, and scheduled based on different workload needs.

The creation of the SharePod CRD has enabled Kubernetes to overcome the issue of resource fragmentation and inefficient GPU utilization that often occurs in containerized environments. By implementing custom controllers alongside this CRD, KubeShare effectively manages the lifecycle of these shared GPU resources, ensuring efficient utilization without causing significant overhead between tasks. Through this extension, KubeShare has significantly improved GPU utilization and overall system throughput in Kubernetes clusters, highlighting the power of adaptability through custom resources.

	Plugin-Based Systems	**Chrome Extensions**	**Docker Extensions**
Language Invariance	No. Plugins language is dependent on the language defined by the platform.	No. Extensions need to be developed in HTML, CSS & JavaScript.	Allows extensions to be developed in any language as long as they are containerized in docker.
Ease of Deployment	High. It supports hot plugging, where plugins can be installed on the fly without disruption.	High. Extensions can be installed on the fly without disruption.	High. Docker extensions are run without the need to restart the docker client.
Maintainability	High. Using the plugin architecture pattern, a program is decoupled into multiple standalone plugins, allowing for isolated changes and faster implementation.	High. Extension development is decoupled from the development of the core app, allowing for isolated changes and faster implementation.	

Table 2.1: Summary of different extensibility architectures.

2.3.4 Comparison of Extensibility Approaches

In this subsection, we compare the different extensibility approaches discussed previously. The following are the key characteristics we would be using to compare the approaches:

- **Language Invariance**: This key criterion assesses the platform's flexibility for installing plugins developed in any programming language.
- **Ease of deployment**: It assesses the ability to add, update, or remove extensions or plugins without restarting or disrupting the core system.
- **Maintainability**: This criterion evaluates how easily the core system can be maintained over time.

3 System Design

This section delves into the requirements and the architectural design of Oakestras' enhancements. It starts with a concise overview of the Oakestra system architecture and then proceeds to explore the enhancements created for Oakestra aimed at improving the system's flexibility.

3.1 Requirements

The requirements for transforming Oakestra into a flexible platform are outlined below. This effort is inspired by the works referenced in section 2.3.

R1 Language Invariance It's essential not to limit Oakestra to a specific programming language so developers can create addons in various programming languages.

R2 Modularity Through enhanced modularity, we can decouple the system and bolster flexibility. Each Oakestra user should be able to tailor the platform so that features or addons can be activated or deactivated as required. This will ensure that only essential components are operational, leading to optimized resource usage and improved system efficiency.

R3 Ease of Deployment It is vital not to disrupt the workflow of the core application when installing addons, for instance, when deploying applications/services on edge nodes while installing addons on the orchestrators, it is critical not to disrupt the process of deployment.

R4 Swap in/out core components We aim to have the ability to integrate various components into the system. For example, we want to be able to plug in a different type of scheduler, thereby replacing the default scheduler that comes with the platform.

R5 Performance Impact The primary objective is to create an addons system that does not impact the performance of the normal system flow. For example, addons should not significantly affect the deployment time of applications.

3.2 Orchestrator Components Analysis

As seen in Figure 2.3, there are three main systems of Oakestra: **Root Orchestrator**, **Cluster Orchestrator**, and **Worker nodes**. This opens possibilities for extending each system independently. In other words, the appropriate extendibility techniques for each system are

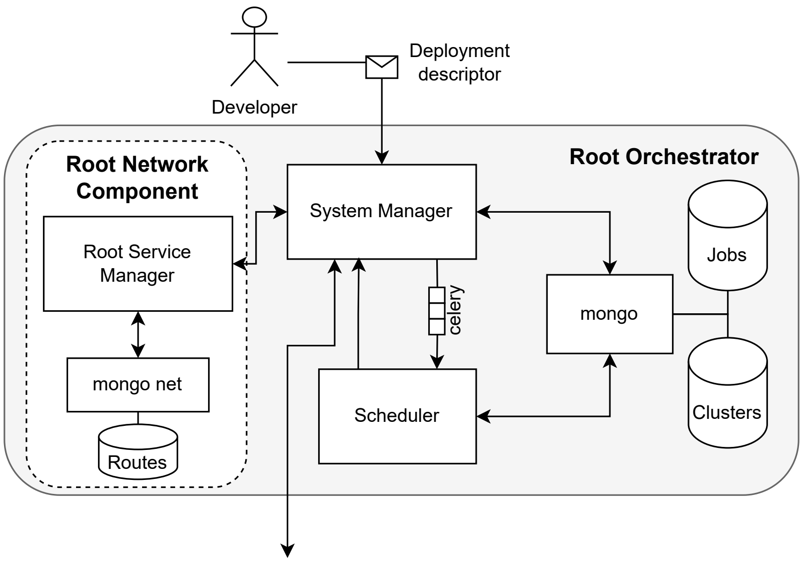

Figure 3.1: Architecture design of the Root Orchestrator [19].

applied. Since the Root Orchestrator and Cluster Orchestrator are nearly the same and are considered logical twins [19], the analysis will focus on the former component. The **Root Orchestrator** system is comprised of docker containers. Currently, the components are mainly written in Python code. However, no strict rules exist for developing these components in a specific language. The only requirement is that the components need to be packaged and containerized. The *System Manager* depicted in figure 3.1 is the entry point to the system - it's where the users' requests are processed or handled.

The analysis of the extendibility starts with the Root Orchestrator. Figure 3.1 gives a closer look into the components of the Root Orchestrator. The components are deployed as docker containers. One idea to achieve extendibility, as seen in section 2.3, is to utilize the Plugin-based pattern or Systems with Extensions.

3.3 Addons

Plugins vs Extensions

Plugins and Extensions are interchangeable concepts that provide a system extension mechanism. In section 2.3, it was noted the difference between "Plugin-Based systems" and "Systems with Extensions." A plugin is where a component is plugged in as part of the system's core component. However, extensions extend the system capabilities and are not considered a core component of the system, as seen with Chrome or docker extensions in section 2.3.2. To avoid confusion and for the sake of simplicity, we denote the keyword "addon" to encompass either plugins or extensions in Oakestra.

Plugin Example

In Oakestra, the idea of replacing one component with another can be utilized. For instance, the current scheduler component can be replaced by another scheduler that uses a different algorithm incorporating environmental factors (e.g., energy consumption or use of green energy) in its decision-making process. This flexibility allows users who need advanced scheduling features to activate this plugin, thus replacing the default one with one better suited for their custom requirements. Meanwhile, users who don't need this feature will continue using the default one, leading to a system that is flexible enough to handle users' needs while keeping the core system as lightweight as possible. This approach ensures that Oakestra can adapt to various user needs without bloating the core system with unnecessary features.

Extension Example

The concept of extensions can be utilized to maintain a modular and lightweight system that adds new features as the user needs. For example, one potential extension in Oakestra could be enabling federated learning on edge computing resources. With federated learning, the learning can be distributed across multiple decentralized devices without sharing raw data, enhancing privacy and security [34]. Implementing this involves adding new components to the "Orchestrator" control plane, facilitating the coordination and aggregation of distributed training processes. Users can activate or deactivate this feature by packaging the federated learning functionality into an extension based on their specific needs. This design ensures the core system remains streamlined, with additional capabilities available on demand, enhancing Oakestra's flexibility.

3.4 Hooks

Addons aren't the only mechanism for enhancing a system's flexibility. This section explores another technique, called "Hooks," to further enhance Oakestra's flexibility. The main idea behind Hooks is to provide a way for services to listen to life-cycle events occurring within the system. For instance, when an application or service is deployed in Oakestra, a corresponding event is triggered, notifying interested parties about the creation of the application. This capability is particularly useful when other services need to react to such events.

Life-cycle events

Life-cycle events refer to various stages in the existence of entities managed by Oakestra, such as applications and services. These events can include creation, deletion, and other states of an entity. By capturing and responding to these events, the system can maintain a dynamic and flexible environment where components are updated and actions are taken automatically based on the current state of the system. To avoid unnecessary complexity of designing a system that allows subscribers to listen to more states of an entity, such as 'deployed', or ‘running‘ state, the Hook feature will provide only reporting the following events happening to an entity: **Creation**, **Update**, and **deletion**. The ’update’ event should encapsulate all other states, and it is up to the client subscribing to pull more information to retrieve a more granular information of the entity.

Benefits of Hooks

Hooks can further decouple the platform, enhancing its modularity. Consider the "Root Network" component depicted in figure 3.1; when a service in an application is being created, the "Root Network" component must be informed about the new service to facilitate communication among deployed services on the "Worker Node." If the "Root Network" is not informed, the service will not be scheduled for deployment to the "Worker Node."

Currently, the "System Manager" component is hard-coded to make an API request to the "Root Network" with details of the service scheduled for deployment. This tight coupling can be alleviated by using Hooks. The "Root Network" component can register to listen for events related to application services. In this setup, the "System Manager" does not need to be aware of the "Root Network" component, thus achieving decoupling.

When a service is scheduled for deployment, the "System Manager" triggers an event that all subscribers, including the "Root Network," can listen to. The "Root Network" must be informed synchronously that a service is scheduled for deployment. If the "Root Network" fails to assign a network address to the service, the "System Manager" will not deploy the application/service.

In other cases, the "System Manager" doesn’t need to wait for a response from the "Root Network" component indicating it has processed the request. In this case, the Hook could be triggered asynchronously so that the core flow of the "System Manager" is not interrupted. For example, when the service is scheduled to be undeployed, the "System Manager" doesn’t need to wait for the subscribers’ response.

This approach prompts designing a Hook system that supports both asynchronous and synchronous mechanisms for listening to system events. By implementing such a Hook system, Oakestra can achieve greater flexibility and modularity, allowing components to interact dynamically without direct dependencies.

3.5 Custom Resources

Custom Resources is another mechanism for extending Oakestra API and enhancing its flexibility. "Custom resources" is mainly inspired by Kubernetes’ implementation of their "Custom Resources" [32].

Kubernetes defines "Custom Resources" as an endpoint that stores a collection of API objects of a specific type [32]. "Custom Resources" are meant to extend the API of a system by dynamically adding new endpoints to a new resource that is originally not part of the default installation of the system and automatically handles their storage.

Controllers

Custom Resources provide the capability to store and retrieve structured data. In addition, a separate component, a controller, can be developed to oversee this data type. This approach effectively separates the responsibilities, with one component managing storage and the other striving to achieve a state the stored object indicates. In Kubernetes, the second component is called a Custom Controller (Kubernetes Custom Resources). In Oakestra, the Custom Controller could be implemented as an Addon.

4 Implementation

4.1 Addons System

To develop addons that satisfy the requirements above, we utilize packaging addons as containers, as well as designing new components to manage the installation of addons to the system. Addons are packaged as containers so that they can be written in any language. Since the core components in "Root Orchestrator" are also containerized, we can fulfill the requirement of swapping in/out core components by shutting down the default container and running the addon/plugin component. Furthermore, the new component should also fulfill the concept of hot-plugging, where addons are installed on the fly without disrupting the system's operations. We have seen previously that this is possible with docker extensions. The system will comprise two main subsystems:

- **Addons Engine**: Consists of Addons Manager, Addons Monitor, and MongoDB Client as shown in figure 4.2
- **Addons Marketplace**: Consists of Marketplace Manager and MongoDB Client as shown in figure 4.1

Addons Marketplace

The "Addons Marketplace" allows developers to publish their add-ons for Oakestra. To create an addon, users must package their code and convert it into an OCI-compliant image that is publicly accessible. Currently, only docker images are supported, but it should be possible to use any OCI-compliant image in the future. The Addons Marketplace subsystem offers API endpoints for developers to publish their addons to Oakestra. Upon receiving a request for registering an Addon, the marketplace asynchronously checks if the image is valid. Once it's verified, the addon's state in the database changes from 'under_review' to 'approved. If invalid, the status becomes 'failed_verification'.

Addons Engine

It is composed of two main components:

- *Addons manager*: handles user requests for installing and uninstalling addons.
- *Addons Monitor*: is responsible for continually monitoring and managing the state of addons. It uses a pull-based mechanism to regularly fetch addon-related tasks from the

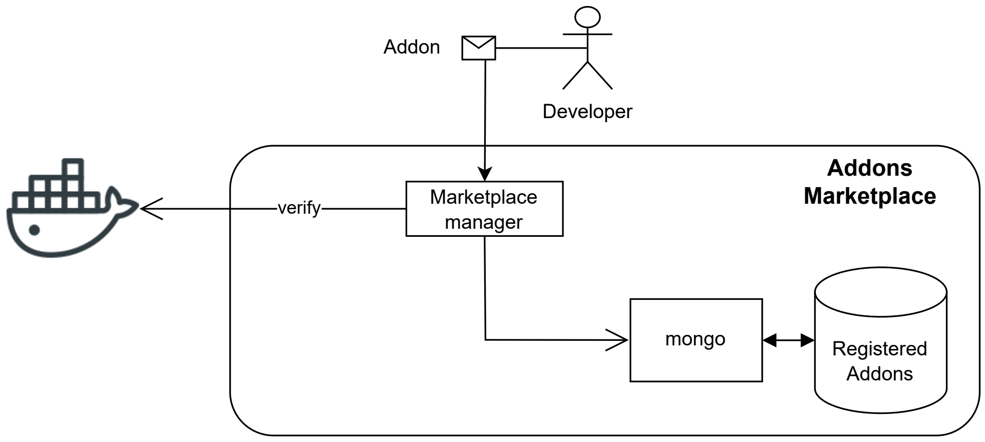

Figure 4.1: Addons marketplace high-level architecture design

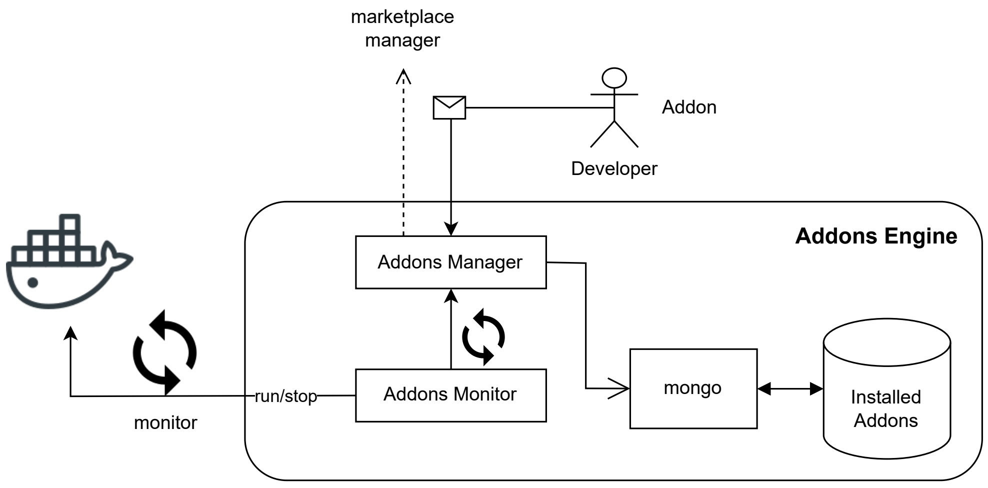

Figure 4.2: Addons engine high-level architecture design.

Addons Manager. It also interacts with the underlying container management engine, such as Docker, to manage the life cycle of addons. This includes handling execution, monitoring, and reporting of addon statuses to ensure that addons are running as expected and reporting any failures to the addons manager. The Addons Monitor performs the following checks:

– **Install check**: Retrieves all addons from the Addons Manager that need installing.

– **Uninstall check**: Retrieves all addons from the Addons Manager that need uninstalling.

– **Cleanup check**: Communicates with underlying containers engines to retrieve all running addons and validates that these addons exist in Addons Manager. The purpose of this is to check if an addon was deleted from the database in the Addons Manager.

Example To add an addon to the marketplace, developers would need to send a [POST] API request to the **Addons Marketplace** containing the Addons Object in the body of the request. The following is a sample example of what the Text-based open standard designed for human-readable data interchange (JSON) body would look like.

```
{
  "name": "mongo_root",
  "networks": [],
  "volumes": [{
      "name": "mongodb_data",
      "driver": "bridge"
    }],
  "services": [{
    "command": "mongod --port 10007",
    "image": "docker.io/library/mongo:3.6",
    "service_name": "mongo_root",
    "networks": [],
    "volumes": ["mongodb_data:/mongodb"],
    "ports": {"10007":"10007"}
    }]
}
```

The above sample is a simple addon with only one service/container to spin up. The image of the service/container is a public one hosted by docker. The addon engine will spin up one container called 'mongo_root'. The addon provides extra configuration, such as creating a volume called 'mongodb_data' and binding the service/container to it. Moreover, the addon requires no extra network configuration. By default, every addon is automatically added to a pre-configured network if no network configuration is provided. For instance, every addon would take the addons engine network by default. This means addons could be added to the

Root Orchestrator or **Cluster Orchestrator**, facilitating the communication between addons and core components.

In order to achieve the requirement, "R3 Swap in/out core components", the *Addons Engine* checks if another service of an addon is similar to another core component found in the Oakestra environment. If this is the case, the core component will be swapped out in favor of the add-on. Check out figure 4.3 for a visual representation of such a process.

Installing an addon Previously, we showed how a developer could publish an addon to the marketplace, and in order to install an addon and have it running, the user would need to send a [POST] API request to the Addons Manager instructing to install the addon. Next, the *Addons Monitor*, which continuously pulls the *Addons Manager* for addons information, installs the new addon. The following is a sample example of what the json body of the request would look like:

```
{
    "marketplace_id": "some_id"
}
```

4.2 Hooks System

In Oakestra, API objects were manipulated by various components, such as the scheduler and system manager, which interacted with the jobs and clusters database. This posed a challenge for implementing the hooks functionality, as it would require modifying every component that interacts with those objects — an inefficient approach. To address this, we introduced a dedicated component to manage these entities, providing a unified interface for manipulation. This allowed other components to continue modifying the objects without directly managing them. The hooks logic was then implemented in this centralized component, ensuring consistency and reducing the need for redundant code changes across multiple components.

Resource Abstractor

The role of "Resource Abstractor" is to centralize the management of all entities within the "Root Orchestrator." This provides a unified interface for creating and managing hooks. Figure 4.4 illustrates the newly designed architecture of the "Root Orchestrator," now incorporating the new component - Resource Abstractor.

Implementation

We start by defining what type of hooks that the system provides:

- **Synchronous Hooks**: Synchronous hooks are executed in a blocking manner, which means the application waits for the response before data is persisted in the database. This hook type is useful when the data needs to be preprocessed before being persisted in the database.

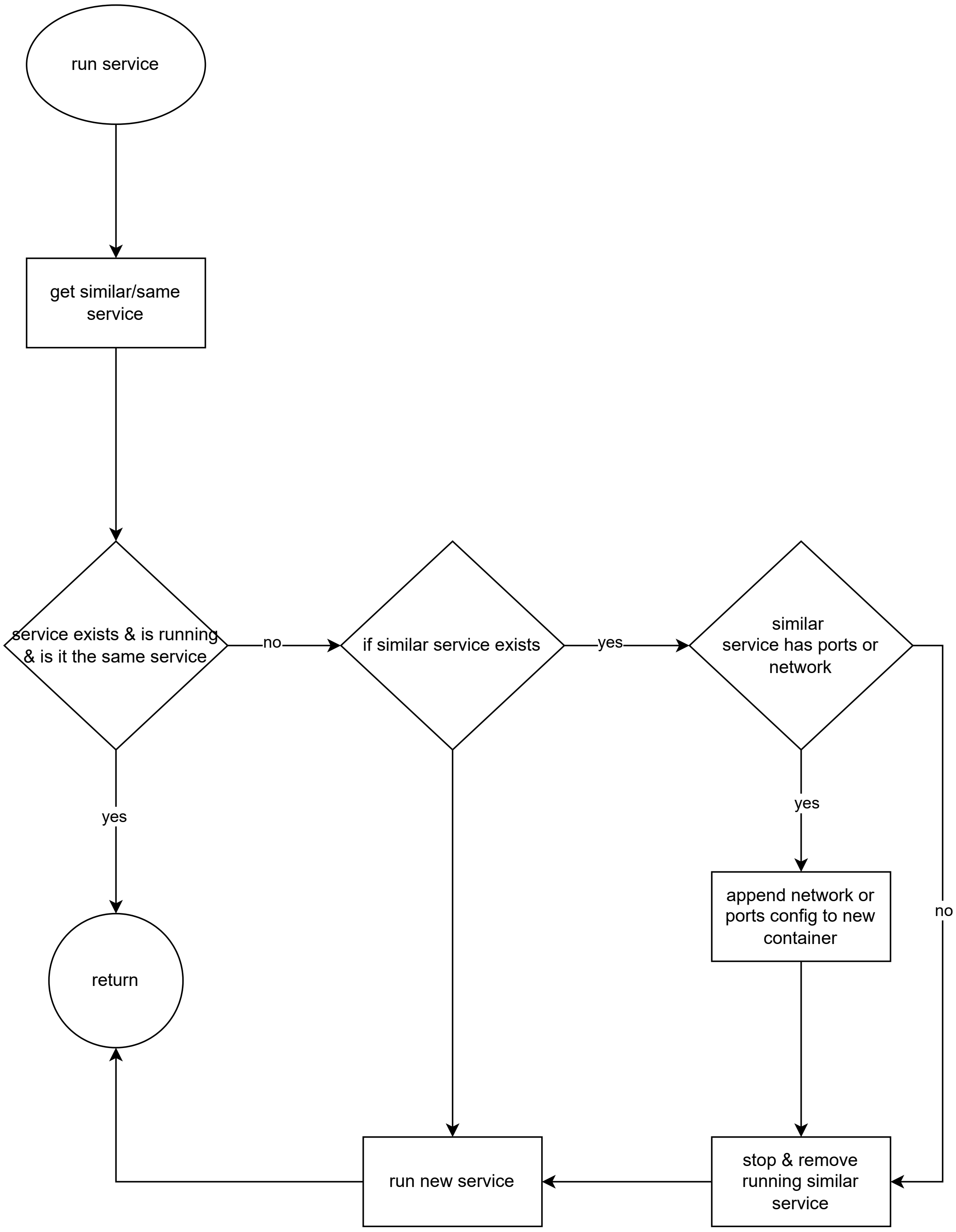
run service
get similar/same service
service exists & is running & is it the same service
no
if similar service exists
yes
similar service has ports or network
yes
yes
no
append network or ports config to new container
return
run new service
stop & remove running similar service

Figure 4.3: Flow chart of the run service process of the Addons Monitor.

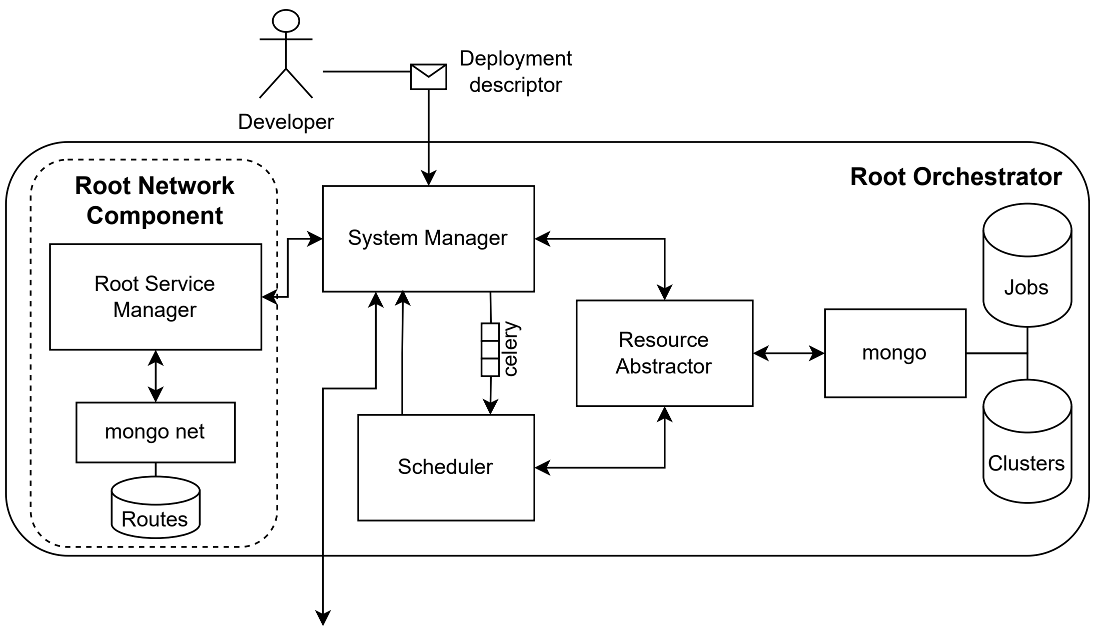

Figure 4.4: Root orchestrator design architecture with the new component - resource abstractor

- **Asynchronous Hooks**: Asynchronous hooks are executed in a non-blocking manner, which means the application does not wait for the response before persisting the data in the database. Instead, it notifies other services that an event of interest has occurred, allowing them to take action if needed.

How to Register a Hook : The "Resource Abstractor" provides several API endpoints where interested services could register for listening on entities for the following events:

- **Creation**
- **Modification**
- **Deletion**

For each event, we provide the synchronous & asynchronous way. To subscribe for synchronous, interested entities would register for 'pre_{event_name}', and for asynchronous, they would register 'post_{event_name}'. For instance, subscribing to 'post_create', entities would be notified of an entity being created and saved to the database. Conversely, when services register for 'pre_create', they would receive in a blocking manner the entity that would be saved, allowing them to modify the object before it being persisted.

Example In order to subscribe to entities, interested parties would send a [**POST**] HyperText Transfer Protocol (HTTP) request with the following as the body of the request:

```
{
  "hook_name": "any name",
  "webhook_url": "URL where for notifying",
  "entity": "application", # name of the entity to be listened to
  "events": ["pre_create", "post_update"]
}
```

In the example above, a user wants to listen synchronously to an entity called "application" when it is being created and asynchronously when an update is being made to an object. As a result, when an entity, like an application, is getting created, the Resource Abstractor will notify the interested party by making another **POST** HTTP to the endpoint indicated in the 'webhook_url'. For the asynchronous event, 'post_update', interested parties would be notified with the 'id' and 'event' of the entity of interest. Here is a response example:

```
  {
    "entity_id": "someid",
    "entity": "application",
    "event": "post_create"
  }
```

For the synchronous event, 'pre_create', interested parties would be notified with the object of the entity being created. In this case, the user would modify the object and return it. Here is a response example:

```
{
  "_id": "someid",
  "application_name": "some name",
  "some_attribute1": "example1",
  "some_attribute2": "example2",
  "some_attribute3": "example3"
}
```

For better illustration, refer to figures 4.5, & 4.6 that depict an interaction diagram when creating an application synchronously and asynchronously.

4.3 Custom Resources

Custom Resources is built inside the "Resource Abstractor" component of the "Root Orchestrator". This allows for utilizing the Hooks System, such that a custom resource's lifecycle events could be listened to.

Example In more detail, the "Resource Abstractor" provides API endpoints for creating and managing a Custom Resource. The details Code 4.1 provides a sample format of the request

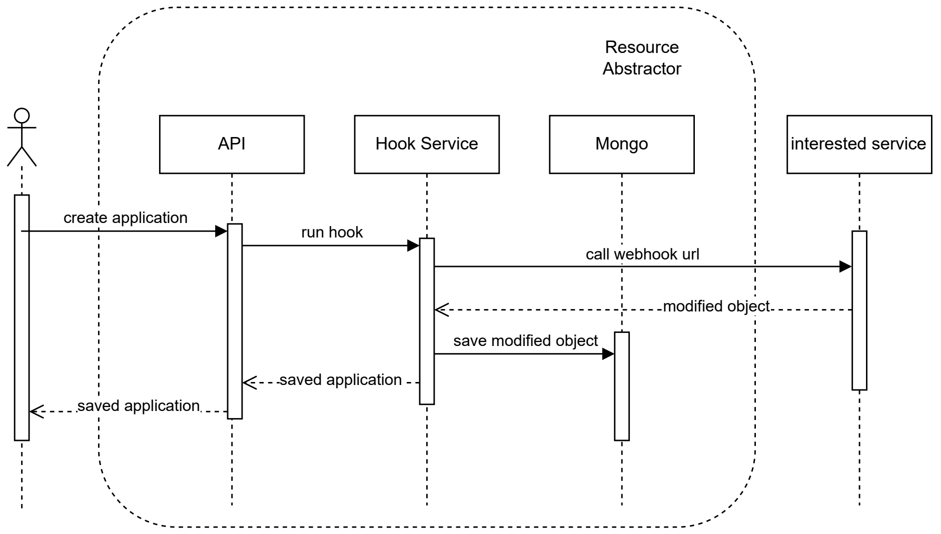

Figure 4.5: Interaction diagram of creating an application with sync hooks setup. It displays how the webhook is first called before returning the saved application to the user.

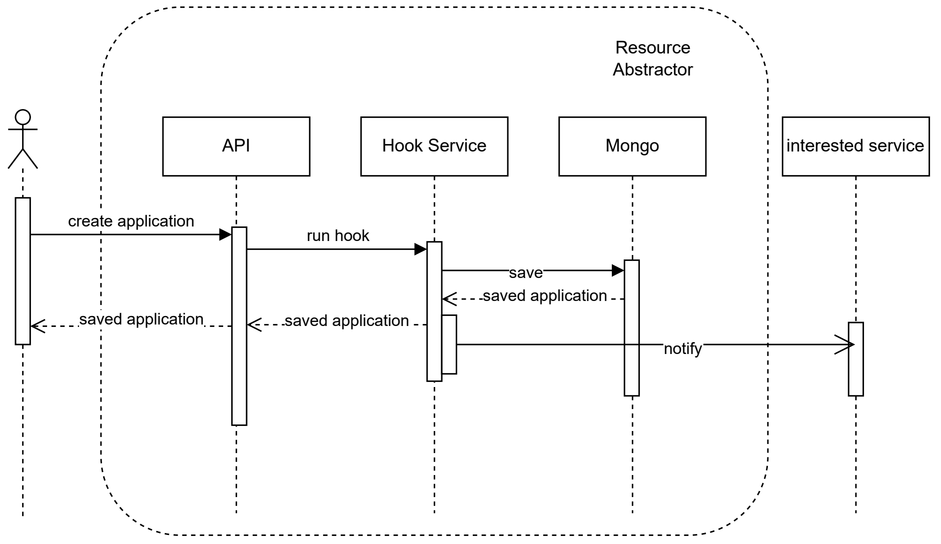

Figure 4.6: Interaction diagram of creating an application with async hooks setup. It displays how the webhook is called concurrently with the user request. Thus, the user would not need to wait for others to be notified of the object being created.

needed to be sent to create a Custom Resource:

Listing 4.1: Schema for creating a custom resource

```
{
    "resource_type": "clusters"
    "schema": {
        "type": "object",
        "properties": {
        "skills": {
          "type": "array",
          "items": {
            "type": "string"
          }
        },
        "languages": {
          "type": "array",
          "items": {
            "type": "string"
          }
        },
        "age": {
          "type": "integer"
        },
        "experience": {
          "type": "integer"
        },
        "address": {
          "type": "string"
        }
      }
    }
}
```

In the example above, we are creating a new custom resource called ‘clusters‘ and a ‘schema‘ that accepts an openapi specification. When creating a custom resource, the "Resource Abstractor" will create a MongoDB collection under that name if not taken, and this custom resource can now be accessed via the following uri ‘/custom-resources/clusters‘.

5 Evaluation

This chapter evaluates the impact of enhancements implemented to Oakestra. 3 key criteria are being considered:

- **Deployment Time**: The impact on deployment time of the normal flow of Oakestra applications when addons are installed and running.
- **Performance**: The performance of Addons on the system, specifically how much resources it consumes to operate. In addition, the performance of custom resources is compared to that of K8s custom resources.
- **Scalability**: The performance of the enhancements is checked when scaled.

Setup

All experiments use two types of machines as depicted by figure 5.1:

- **Worker Node**: The worker node is the edge resource running the deployed services.
- **Root Orchestrator and Cluster Orchestrator**: Both orchestrators are deployed on the same machine - simplifying the testing setup. There is no need to conduct tests across multiple clusters. This is sufficient to test the core functionality and to isolate the impact of enhancements on the platform.

Moreover, both machines are joined through a bridge network to facilitate communication between the node engine and the orchestrators. Also, both machines are identical in terms of specs:

Listing 5.1: Machines Specs

```
{
    "hw_cpu_speed": 3000,
    "hw_cpu_threads": 2,
    "hw_cpu_bits": 64,
    "processor": "AMD EPYC 7302P",
    "hw_cpu_cores": 16,
    "hw_mem_size": 131072
}
```

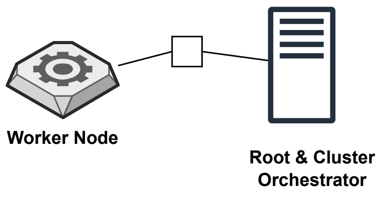

Figure 5.1: Experiments setup

5.1 Deployment Time

It is vital that the enhancements implemented, in particular the **Addons System**, don't negatively impact the normal flow of Oakestra, for example, the deployment of services on Edge compute resources. This section assesses the impact of the implemented **Addons System** on Oakestra deployment time of applications.

Experimental Setup

This experiment evaluates the Addons System by comparing the deployment times of applications in a system with and without addons. To evaluate the impact of the Addons Engine on deployment times, we conducted a series of tests comparing two scenarios:

1. **No addons**: Deployment of applications/services in a baseline system without any addons installed.
2. **With Addons**: Deployment of applications/services in a system with several addons installed and running.

The main metric for this evaluation is the deployment time; a simple service helps in calculating it. The simple service, on startup, calls some endpoint, enabling the calculation of the deployment time. In more detail, deployment time is computed by checking when the request for deployment is made and the time the application on startup requests to some defined endpoint. As a result, the deployment time can be accurately computed. Moreover, the tests were performed under identical conditions, ensuring that the only variable was the presence of the addons. The following are the test Scenarios used for the experiment:

1. *Baseline Test (Without Addons):*
   - Measures the deployment time for 10 applications, each having 6 services in a clean Oakestra environment without any addons.
2. *Addons Test (With Addons):*
   - Measures the deployment time for 10 applications, each having 6 services in a clean Oakestra environment with 20 simple addons getting installed.

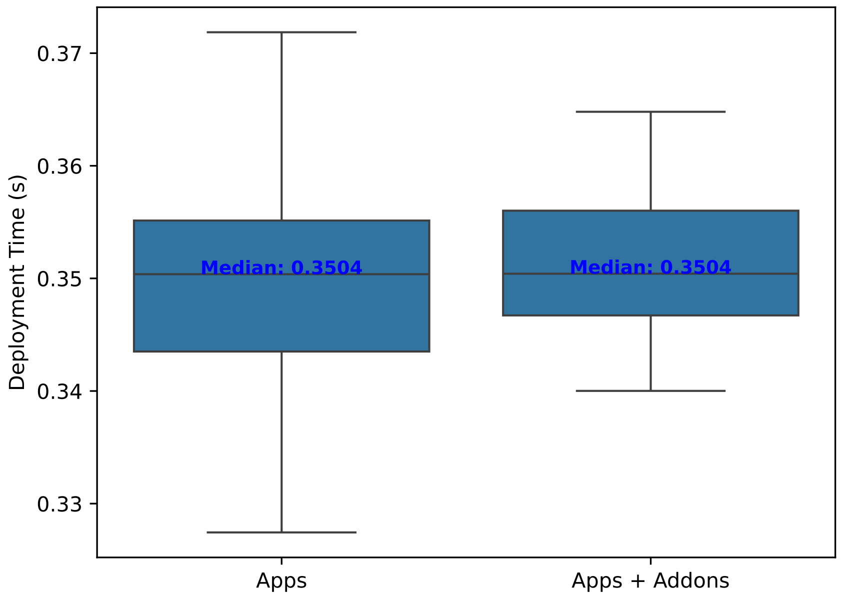

Figure 5.2: Shows a box plot that compares the deployment times of applications in Oakestra under two conditions: no addons and with addons. The x-axis represents the test scenarios, while the y-axis shows the deployment time in seconds.

Results & Analysis

In Figure 5.2, median deployment times suggest that the core deployment process remains largely unaffected by the addons. The slight increase in variability may be attributed to the additional processing required for the addons, but it does not significantly affect the overall performance.

5.2 Performance

5.2.1 Addons

This section examines the performance impact of the Addons System on Oakestra and compares it to several system scenarios. The metrics we focus on are the CPU and Memory Usage.

Experimental setup

Several system configurations are being compared:

1. **System Idle**: In this baseline scenario, no applications are deployed, and only the necessary containers for Oakestra's operation are running. This setup provides a reference for the system's minimal resource consumption.
2. **Stress Test with Services**: Sixty services are deployed simultaneously to evaluate the system's performance under high load conditions without any addons. This scenario helps in understanding the CPU and memory demands of the core orchestration functionality.
3. **Addons Only**: Twenty addons are installed and active, but no services are deployed. This test isolates the resource consumption of the addons system and the installed addons themselves.
4. **Apps + Addons**: Both the services (60) and addons (20) are deployed and running. This scenario simulates a fully operational environment with core services and additional functionalities added by addons, showcasing overall system performance and resource utilization.

Note Image size of addons range from 1.2Mb to 42Mb. The following images are used as addons: busybox, alpine, node:14-alpine, and python:3.9-alpine.

Results

1. **Memory Usage**: Shown in figure 5.3
   - **System Idle**: The memory usage remains stable and low, serving as a baseline.
   - **Apps Only**: Memory usage increases moderately as services are deployed. This reflects the resources needed by the applications to be deployed.
   - **Addons Only**: Memory usage is higher compared to the idle state.
   - **Apps + Addons**: This scenario shows the highest memory usage, combining the demands of both the applications and the addons.
2. **CPU Usage**: Shown in figure 5.4
   - **System Idle**: The CPU usage is minimal and consistent. This provides a baseline measurement.
   - **Apps Only**: CPU usage increases with the deployment of services.
   - **Addons Only**: CPU usage is higher than the idle state but lower than the apps-only scenario.
   - **Apps + Addons Only**: The highest CPU usage is observed in this scenario, which combines the processing requirements of both applications and addons.

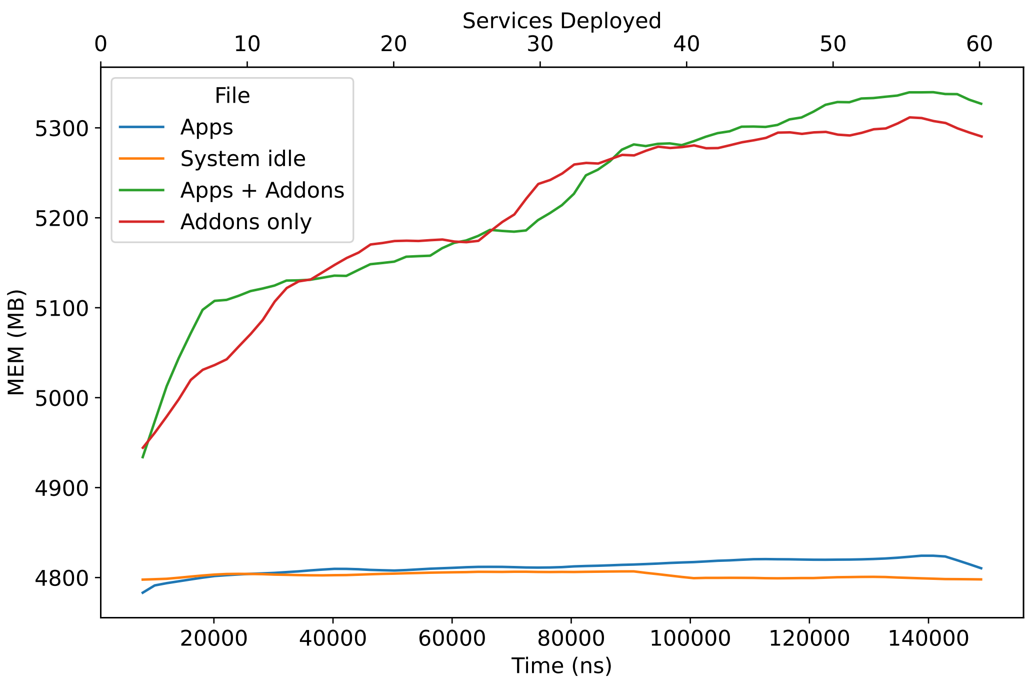

Figure 5.3: MEM levels across the different test scenarios.

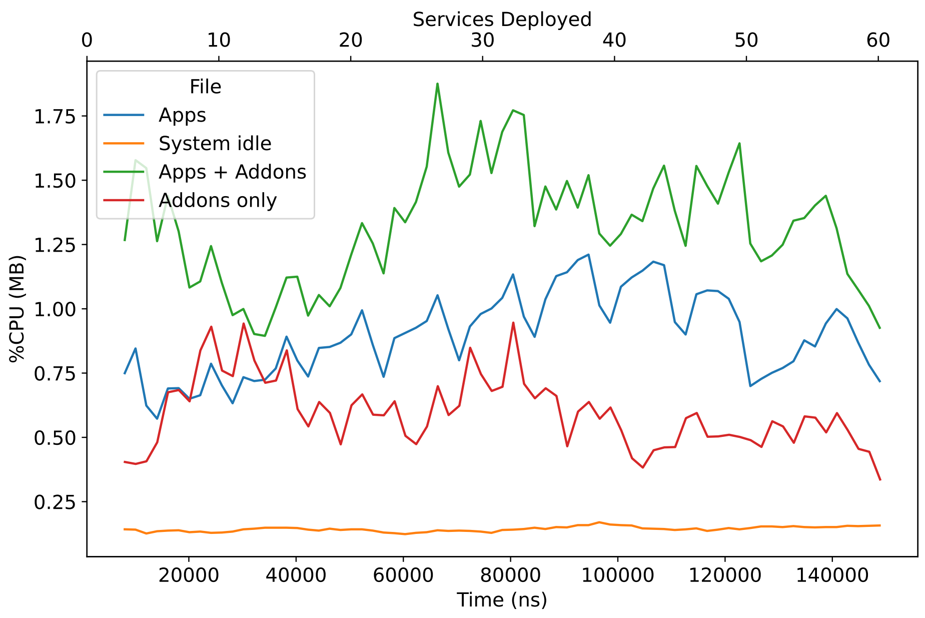

Figure 5.4: CPU levels across the different test scenarios.

Analysis

The results indicate that adding addons introduces additional memory and CPU overhead, which is expected due to increased functionality. The increased memory usage with addons suggests that while they provide enhanced capabilities, they also require additional memory resources. 5.3 indicates a difference of ~450Mb in memory usage between Addons and Apps profile. This result is sensible since the image size of each addon is nearly 22Mb (mean of 1.2Mb, 5Mb, 39Mb, 42Mb) and that 20 addons are being installed, there should be a difference of at least ~440Mb between memory usage of Addons vs Apps profile. Similarly, the CPU usage patterns reflect the processing demands of managing and running the addons alongside the core applications. The highest CPU usage in the combined scenario indicates the additional processing required for the integrated features. These findings demonstrate that the Addons feature enhancement achieves the intended flexibility and extensibility without significantly compromising system performance. The slight increase in resource usage is a reasonable trade-off for Oakestra's added capabilities and improved adaptability.

5.2.2 Custom Resources

An experiment evaluated the performance of custom resources implemented in Oakestra. To draw some baseline performance, the time taken to create custom resources in Kubernetes is compared to that of Oakestra. Note that there was no need to compare the system performance for both platforms' implementation of custom resources since it would have led to a comparison of containers.

Experimental Setup

1000 custom resources were created for each platform, Oakestra & Kubernetes, where the time taken for the creation is recorded.

Results & Analysis

Figure 5.5 compares the time taken (in seconds) to create custom resources between Oakestra (labeled as "OAK") and Kubernetes (labeled as "Kube"). The results indicate that Oakestra is faster in creating custom resources by around 30%. Also, the smaller variance in Oakestra indicates a more consistent performance when creating a custom resource.

In comparison, Kubernetes exhibits higher mean and median times and greater variance. This slower performance may be attributed to the additional processes that Kubernetes might execute while creating custom resources. For example, Kubernetes may set up additional security measures or register the custom resource with various internal controllers and monitoring systems. These additional steps, while enhancing the security and robustness of the system, could introduce overhead that increases the time required to create custom resources.

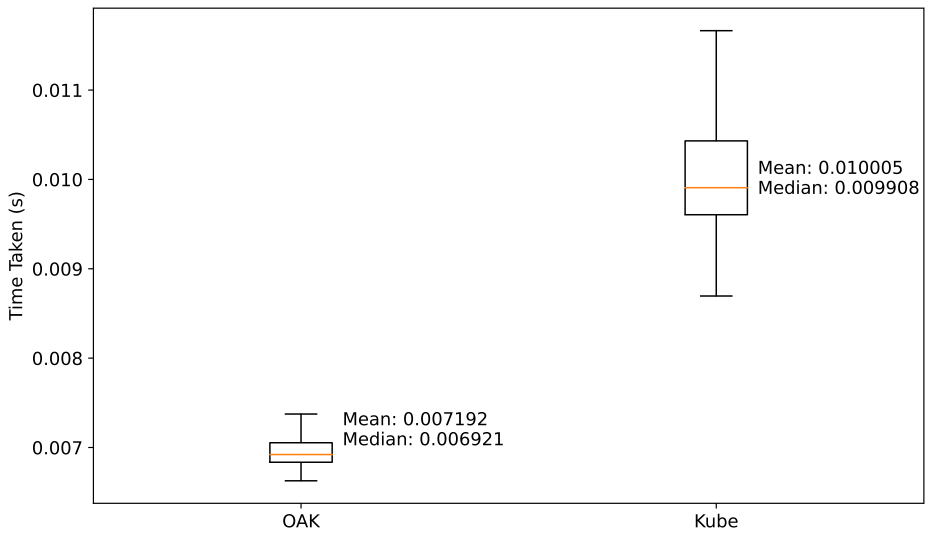

Figure 5.5: Box plot showing the time taken in seconds to create custom resources in Oakestra and Kubernetes.

5.3 Scalability

This section tackles the scalability aspect of the enhancements. First, we investigate how the hooks system scales with the increasing number of entries added to the database. Similarly, we experiment with how the hooks system scales with varying subscribers on an API object.

In section 4.2, we explained how the hook system operates through a subscription model, where interested parties can register their webhooks via an API request. In more detail, the request specifies the entity of interest, such as "applications," and the specific events they wish to listen to, such as "*post_create*" and "*pre_create*." The "*pre_create*" event is handled synchronously, allowing subscribers to manipulate the object before it is persisted to the database. In contrast, the "*post_create*" event is processed asynchronously, notifying subscribers after the object has been successfully saved, along with the object's ID.

By conducting these experiments, we aim to provide a sanity check on the performance impact of the hook system, ensuring that the implementation of hooks does not introduce significant overhead or scalability issues. The results will offer insights into the efficiency of the hook system and its suitability for real-world edge orchestration scenarios.

Experiments Setup

The experiment used many identical Docker containers, where the containers are running a simple server. Upon startup, each server subscribes to listen for the creation of an API object called 'cluster.' The experiment involved two main tests:

- **Synchronous Hooks Test**: Servers subscribed to synchronous hooks, specifically the "*pre_create*" event, which allows subscribers to manipulate the object before it is persisted in the database.
- **Asynchronous Hooks Test**: Servers subscribed to asynchronous hooks, specifically the "*post_create*" event, which notifies subscribers after the object has been successfully saved.

10,000 objects were created for both tests, and the time taken for each creation event was recorded. Each test was conducted with **n** servers subscribed to the creation events, ensuring a consistent number of webhook calls for each event.

Test Flow

1. **Synchronous Hooks Test**:
   - Each server subscribes to the "*pre_create*" event.
   - The server manipulates the object and returns it for persistence.
   - The time taken to create each entity is recorded, including the time spent processing the synchronous hooks.
2. **Asynchronous Hooks Test**:
   - Each server subscribes to the "*post_create*" event.
   - The server receives the ID of the newly created object after it has been persisted.
   - The time taken to create each entity is recorded, focusing on the database persistence time, with webhook notifications handled asynchronously.

Experiment I

The primary goal of this experiment was to measure the time taken to create entities in the presence of the hook system and to observe whether the creation time scales with the number of objects being created. The primary objectives of this experiment are to analyze whether the creation time remains nearly constant regardless of the number of objects being created and to verify that the time taken with the hook system does not scale significantly with more data being created, given the constant number of webhook calls being processed per event.

Results

Figure 5.6 illustrates the average time taken to create each cluster in the presence of synchronous and asynchronous hooks. The average time taken for synchronous hooks (represented in orange) remains relatively constant across the different numbers of clusters created. Also, for the asynchronous test, the average time taken (represented in blue) remains relatively constant across the increasing number of ‘clusters‘ created. The error bars across the two tests,

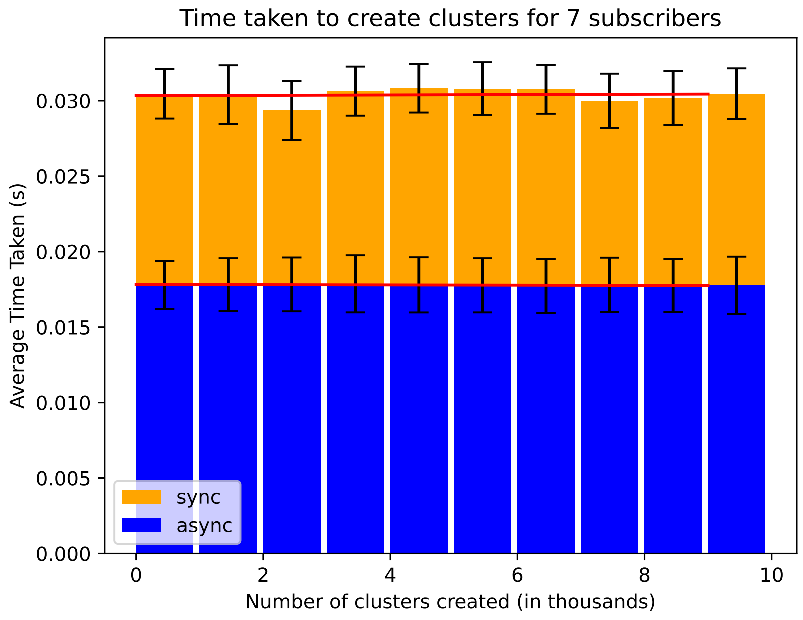

Figure 5.6: Average time taken to create ‘clusters‘ with synchronous and asynchronous hooks. The x-axis represents the number of clusters created (in thousands), while the y-axis shows the average time taken for each creation event. Error bars indicate the variability in the measurements.

synchronous and asynchronous, indicate some variability in the time taken, but it is within a reasonable range. It is clear that the average time taken is slightly higher for the synchronous hooks compared to asynchronous hooks, which is expected due to the additional processing required to manipulate the object before it is persisted into the database.

Analysis

The results confirm the hypothesis that the time taken to create entities should remain nearly constant regardless of the number of objects created, given that each event processes a constant number of webhook calls. The synchronous hooks introduce a slight overhead due to the need for real-time processing and manipulation of the object before persistence. In contrast, the asynchronous hooks exhibit a lower and more consistent average time taken as the processing is decoupled from the creation event.

Experiment II

The second experiment compares the time taken for an API object to be persisted into the database, given a varying number of listeners for this object. The primary object is to determine the impact of varying numbers of listeners on the time taken to persist an API object in the database and to verify.

Results

The results found in figure 5.7 indicate that as the number of servers subscribed to hooks increases, the creation time for entities also increases. However, the impact is more pronounced for synchronous hooks than asynchronous hooks. This is due to the blocking nature of synchronous hooks, where each server must complete its processing before the entity can be created. On the other hand, asynchronous hooks allow the creation process to proceed independently, leading to a more moderate increase in creation time.

Analysis

The results show that increasing listeners' time for both asynchronous and synchronous operations increases linearly, which is an acceptable result.

Conclusion

The findings demonstrate that the hook system, whether synchronous or asynchronous, does not introduce significant scalability issues. The system efficiently handles the webhook calls, maintaining a relatively constant creation time even as the number of entities increases. This validates the effectiveness of the hook system in enhancing Oakestra's flexibility without compromising performance. In conclusion, the evaluation shows that the hook system is a viable solution for extending Oakestra's capabilities.

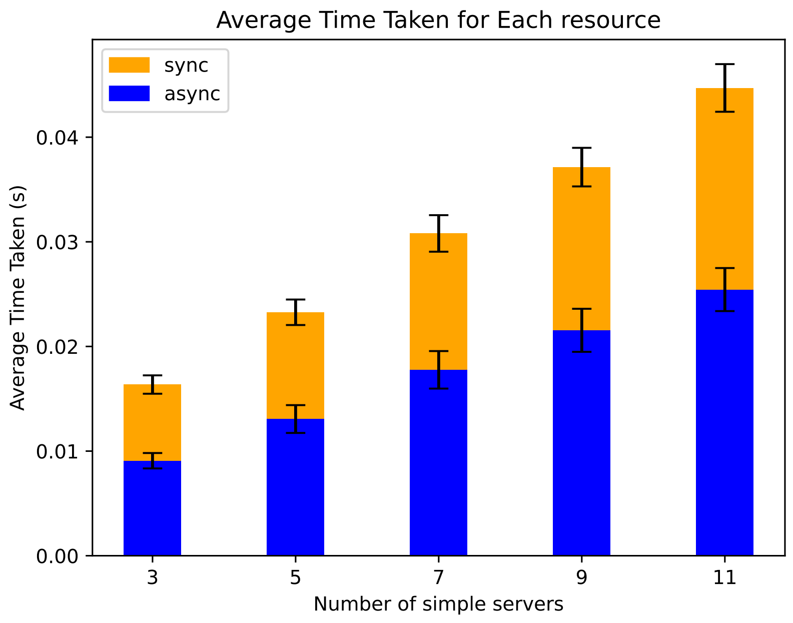

Figure 5.7: Average time taken to create entities with varying numbers of servers subscribed to synchronous and asynchronous hooks. It presents the average time taken to create 'clusters' with varying numbers of servers (3, 5, 7, 9, 11) subscribed. The x-axis represents the number of simple servers, while the y-axis shows the average time taken for each creation event. Error bars indicate the variability in the measurements.

6 Conclusion

Advancements in edge computing have created unique challenges that orchestration systems must tackle. This thesis set out to present solutions or mechanisms to enhance the flexibility of edge computing platforms. Three key enhancements were introduced - Addons, Hooks, and Custom Resources. It is proposed that these enhancements can be applied to any edge computing platform. As a practical approach for applying these enhancements, Oakestra, an orchestration tool designed specifically for the edge, is used. These enhancements transformed Oakestra into a more modular and extendable system that is capable of meeting future growth.

The addons system has introduced a flexible mechanism for integrating new features without altering the core system, allowing users to tailor Oakestra to their specific needs while maintaining system efficiency. In addition, the Addons system supports replacing core components of Oakestra, allowing for custom solutions, such as custom schedulers. Moreover, the hooks system has provided a powerful way to decouple system components. This enables services to listen to life-cycle events and react to them dynamically. Finally, introducing custom resources has allowed the creation of customizable resource objects tailored to the specific needs of edge computing.

Evaluating these enhancements has demonstrated that they effectively increase Oakestra's flexibility without significantly impacting performance. The slight increases in resource usage are a reasonable trade-off for the added capabilities and improved adaptability. In conclusion, this thesis presents the efforts contributed to the ongoing development of orchestration tools for edge computing, providing a foundation for future enhancements that can further improve these systems' flexibility. As edge computing continues to grow, tools like Oakestra will be critical in enabling the deployment and management of complex, distributed applications at the edge.

6.1 Future Work

Addons Enhancements

The Addons System could benefit from further fine-tuning. The list below presents some improvements that can be applied to the Addons System:

- **Addons Manager**: The resource abstractor can handle the management of addons by utilizing custom resources. Hence, it can be utilized to take on the responsibility of the Addons Manager. As a result, removing the Addons manager entirely leads to less code and improved performance.

- **Marketplace Manager**: Instead of pulling the docker image to verify if an image is valid, pull the manifest and validate it.
- **Addons Monitor**: Support different types of container management tools. Currently only docker is supported for container management. Thus addons can only be run by docker, and what is proposed to allow running addons via other management tools, such as "Podman" [35].
- **UI Marketplace**: Design a user-friendly interface for discovering, installing, and publishing addons. Currently, the only client for registering an addon to the marketplace is via REpresentational State Transfer(REST) API calls.

New Ideas

One promising idea is to explore further utilization of the Resource Abstractor component. The core concept is that since there is a notable similarity in the code structures between the Root Orchestrator and the Cluster Orchestrator, there is an opportunity to refactor these components to leverage the Resource Abstractor in a more comprehensive and efficient manner. Potential Benefits of such refactoring are the following:

- **Unified Resource Management**: By extending the Resource Abstractor to handle different types of resources, such as clusters and node resources, across the orchestration layers, it is possible to unify resource management within Oakestra. Such unification could reduce duplication in the codebase and make maintaining and extending the system easier.
- **Code Reusability**: With a more generalized Resource Abstractor, much of the existing logic duplicated between different orchestrators could be abstracted into a common module. This reduces the overall size of the codebase and enhances its clarity and maintainability.

Glossary

openapi It is an open standard for describing your APIs, allowing you to provide an API specification encoded in a JSON or YAML document.. 27

Acronyms

API	Application Programming Interface. 12, 17, 19, 21, 22, 24, 25, 34, 37, 40
CLI	Command Line Interface. 10
CRD	Custom Resource Definitions. 12
DOM	Document Object Model. 10
HTTP	HyperText Transfer Protocol. 25
IoT	Internet of Things. 1
IPC	Inter-process Communication. 10
json	Text-based open standard designed for human-readable data interchange. 21, 22
OCI	Open Container Initiative. 11, 19
REST	REpresentational State Transfer. 40
rtt	Round-Trip Time. 3

Bibliography

[1] *Data growth worldwide 2010-2025 | Statista — statista.com*. https://www.statista.com/statistics/871513/worldwide-data-created/. [Accessed 24-08-2024].

[2] *IoT devices to generate 79.4ZB of data in 2025, says IDC — zdnet.com*. https://www.zdnet.com/article/iot-devices-to-generate-79-4zb-of-data-in-2025-says-idc/. [Accessed 24-08-2024].

[3] *What Is Edge Computing? | IBM — ibm.com*. https://www.ibm.com/topics/edge-computing. [Accessed 24-08-2024].

[4] K. Bilal, O. Khalid, A. Erbad, and S. U. Khan. "Potentials, trends, and prospects in edge technologies: Fog, cloudlet, mobile edge, and micro data centers". In: *Computer Networks* 130 (2018), pp. 94–120.

[5] W. Z. Khan, E. Ahmed, S. Hakak, I. Yaqoob, and A. Ahmed. "Edge computing: A survey". In: *Future Generation Computer Systems* 97 (2019), pp. 219–235. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2019.02.050. URL: https://www.sciencedirect.com/science/article/pii/S0167739X18319903.

[6] C. Soto-Valero, T. Durieux, N. Harrand, and B. Baudry. "Coverage-based debloating for java bytecode". In: *ACM Transactions on Software Engineering and Methodology* 32.2 (2023), pp. 1–34.

[7] S. Shukla, M. F. Hassan, D. C. Tran, R. Akbar, I. V. Paputungan, and M. K. Khan. "Improving latency in Internet-of-Things and cloud computing for real-time data transmission: a systematic literature review (SLR)". In: *Cluster Computing* 26.5 (Apr. 2021), pp. 2657–2680. ISSN: 1573-7543. DOI: 10.1007/s10586-021-03279-3. URL: http://dx.doi.org/10.1007/s10586-021-03279-3.

[8] *What is round-trip time (RTT)? | RTT meaning — cloudflare.com*. https://www.cloudflare.com/en-gb/learning/cdn/glossary/round-trip-time-rtt/. [Accessed 13-08-2024].

[9] S. Srivastava and S. P. Singh. "A survey on latency reduction approaches for performance optimization in cloud computing". In: *2016 Second International conference on computational intelligence & communication technology (CICT)*. IEEE. 2016, pp. 111–115.

[10] Z. Wan. "Cloud Computing infrastructure for latency sensitive applications". In: *2010 IEEE 12th International Conference on Communication Technology*. IEEE. 2010, pp. 1399–1402.

[11] *What is the Internet of Things (IoT)? | IBM — ibm.com*. https://www.ibm.com/topics/internet-of-things. [Accessed 17-08-2024].

[12] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. "Edge Computing: Vision and Challenges". In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646. DOI: `10.1109/JIOT.2016.2579198`.

[13] Y. Zhang. *Mobile edge computing*. Springer Nature, 2022.

[14] *Why Large Organizations Trust Kubernetes — tanzu.vmware.com*. `https://tanzu.vmware.com/content/blog/why-large-organizations-trust-kubernetes`. [Accessed 17-08-2024].

[15] H. H. Andrew Jeffery and R. Mortier. *Rearchitecting Kubernetes for the Edge*. `https://dl.acm.org/doi/10.1145/3434770.3459730`. 2021.

[16] S. Böhm and G. Wirtz. "Cloud-edge orchestration for smart cities: A review of Kubernetes-based orchestration architectures". In: *EAI Endorsed Trans. Smart Cities* 6.18 (May 2022).

[17] *KubeEdge — kubeedge.io*. `https://kubeedge.io/`. [Accessed 18-08-2024].

[18] *Kubernetes Components — kubernetes.io*. `https://kubernetes.io/docs/concepts/overview/components/`. [Accessed 17-08-2024].

[19] G. Bartolomeo, M. Yosofie, S. Bäurle, O. Haluszczynski, N. Mohan, and J. Ott. "Oakestra: A Lightweight Hierarchical Orchestration Framework for Edge Computing". In: *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, July 2023, pp. 215–231. ISBN: 978-1-939133-35-9. URL: `https://www.usenix.org/conference/atc23/presentation/bartolomeo`.

[20] *What is Extensibility in Software? | Definition - Glossary — sanity.io*. `https://www.sanity.io/glossary/extensibility`. [Accessed 22-08-2024].

[21] J. Mayer, I. Melzer, and F. Schweiggert. "Lightweight Plug-In-Based Application Development". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003, pp. 87–102. ISBN: 9783540365570. DOI: `10.1007/3-540-36557-5_9`. URL: `http://dx.doi.org/10.1007/3-540-36557-5_9`.

[22] M. Richards. *Software architecture patterns*. Vol. 4. O'riely, 2022.

[23] R. Wolfinger, D. Dhungana, H. Prähofer, and H. Mössenböck. "A Component Plug-In Architecture for the .NET Platform". In: *Modular Programming Languages*. Ed. by D. E. Lightfoot and C. Szyperski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 287–305. ISBN: 978-3-540-40928-1.

[24] J. Fedewa. *What Is a Browser Extension?* `https://www.howtogeek.com/718676/what-is-a-browser-extension/`.

[25] *Browser Market Share Worldwide*. `https://gs.statcounter.com/browser-market-share/`. [Accessed 15-07-2024].

[26] L. Liu, X. Zhang, G. Yan, and S. Chen. *Chrome Extensions: Threat Analysis and Countermeasures.* 2012.

[27] V. Aravind and M. Sethumadhavan. "A Framework for Analysing the Security of Chrome Extensions". In: *Advanced Computing, Networking and Informatics- Volume 2*. Ed. by M. Kumar Kundu, D. P. Mohapatra, A. Konar, and A. Chakraborty. Cham: Springer International Publishing, 2014, pp. 267–272. ISBN: 978-3-319-07350-7.

[28] *Understanding the Google Chrome Extension Architecture — bluegrid.io*. `https://bluegrid.io/blog/understanding-the-google-chrome-extension-architecture/`.

[29] *Extension architecture*. `https://docs.docker.com/desktop/extensions-sdk/architecture/`. [Accessed 18-07-2024].

[30] *runtime-spec/config.md at main · opencontainers/runtime-spec — github.com*. `https://github.com/opencontainers/runtime-spec/blob/main/config.md`. [Accessed 27-08-2024].

[31] *Home — docker.com*. `https://www.docker.com/`. [Accessed 27-08-2024].

[32] *Custom Resources in Kubernetes*. `https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/`.

[33] T.-A. Yeh, H.-H. Chen, and J. Chou. "KubeShare: A framework to manage GPUs as first-class and shared resources in container cloud". In: *Proceedings of the 29th international symposium on high-performance parallel and distributed computing*. 2020, pp. 173–184.

[34] P. M. Mammen. "Federated learning: Opportunities and challenges". In: *arXiv preprint arXiv:2101.05428* (2021).

[35] *Podman — podman.io*. `https://podman.io/`. [Accessed 02-09-2024].