

**BACHELORTHESIS**  
Micheal Choudhary

# **Exploring WebAssembly-Based Microservices Implementation & Deployment Methods in Kubernetes**

**FAKULTÄT TECHNIK UND INFORMATIK**  
Department Informatik

Faculty of Computer Science and Engineering  
Department Computer Science

Micheal Choudhary

# Exploring WebAssembly-Based Microservices Implementation & Deployment Methods in Kubernetes

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung  
im Studiengang *Bachelor of Science Wirtschaftsinformatik*  
am Department Informatik  
der Fakultät Technik und Informatik  
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt  
Zweitgutachter: Prof. Dr. Olaf Zukunft

Eingereicht am: 28. Februar 2024

**Micheal Choudhary**

## **Thema der Arbeit**

Untersuchung zur Implementierung und Bereitstellung von WebAssembly-basierten Microservices in Kubernetes

## **Stichworte**

containerd, crun, Fermyon Spin, K8s, Kubernetes, Wasm, WasmEdge, WebAssembly

## **Kurzzusammenfassung**

Diese Arbeit stellt Methoden zur Integration von WebAssembly (Wasm) Workloads in Kubernetes (K8s) Umgebungen vor. Die daraus resultierende hybride Architektur nutzt die Vorteile beider Technologien, einschließlich der Skalierbarkeit, Lastverteilung und Hochverfügbarkeit von K8s sowie der Plattformunabhängigkeit, Geschwindigkeit und Sicherheitsvorteile von Wasm. Es werden zwei Methoden vorgestellt: eine, die crun mit WasmEdge nutzt, und die andere, die containerd, einen containerd-shim und Fermyon Spin verwendet. Diese Arbeit demonstriert eine erfolgreiche K8s-Cluster-Implementierung, bei der Worker-Knoten Rust-basierte Wasm-Microservices unter Verwendung dieser Methoden ausführen. Diese Arbeit legt den Grundstein für innovative Implementierungen von Wasm-basierten Microservices innerhalb von K8s.

**Micheal Choudhary**

## **Title of Thesis**

Exploring WebAssembly-Based Microservices Implementation & Deployment in Kubernetes

## **Keywords**

containerd, crun, Fermyon Spin, K8s, Kubernetes, Wasm, WasmEdge, WebAssembly

## **Abstract**

This thesis presents methods for integrating WebAssembly (Wasm) workloads into Kubernetes (K8s) environments. The resulting hybrid architecture leverages the advantages of both technologies, including K8s' scaling, load balancing, and high availability, alongside Wasm's portability, speed, and security benefits. It presents two distinct methods: one leveraging crun with WasmEdge and the other utilizing containerd, containerd-shim, and Fermyon Spin. The thesis demonstrates a successful K8s cluster implementation where worker nodes execute Rust-based Wasm microservices using these methods. This work establishes a foundation for innovative Wasm-based microservice deployment within K8s.

# Table of contents

<b>List of abbreviations .....</b>	<b>6</b>
<b>Table of figures .....</b>	<b>7</b>
<b>List of tables .....</b>	<b>9</b>
<b>1 Introduction .....</b>	<b>10</b>
1.1 Background and Context .....	10
1.2 Thesis Objectives .....	11
1.3 Thesis Overview .....	12
<b>2 Literature Review .....</b>	<b>14</b>
2.1 Containers .....	14
2.1.1 Container Image .....	14
2.1.2 Container Runtime .....	14
2.2 Kubernetes: Architecture and Components .....	19
2.2.1 Pods .....	21
2.2.2 API Server .....	21
2.2.3 Kubernetes Objects and Kubectl .....	21
2.2.4 ETCD .....	22
2.2.5 Scheduler .....	23
2.2.6 Controller Manager .....	23
2.2.7 Kubelet .....	23
2.3 WebAssembly .....	24
2.3.1 WebAssembly System Interface .....	27
2.3.2 WebAssembly Runtime .....	28
2.4 WebAssembly Meets Kubernetes .....	29
2.4.1 Approach 1: crun .....	29
2.4.2 Approach 2: containerd .....	30
<b>3 Materials and Methods .....</b>	<b>32</b>
3.1 Prerequisites .....	33
3.2 Method 1: Wasm Workload Execution using crun Runtime .....	33

3.2.1	Development Environment Setup for Wasm Compilation.....	33
3.2.2	Building, Containerizing, and Publishing Wasm Workload.....	34
3.2.3	Configuring Worker Node 1.....	36
3.2.4	Deployment of Wasm Module in K8s.....	38
3.3	Method 2: Wasm Workload Execution using containerd Runtime .....	39
3.3.1	Development Environment Setup for Wasm Compilation.....	39
3.3.2	Building, Containerizing, and Publishing Wasm Workload.....	40
3.3.3	Configuring Worker Node 2.....	41
3.3.4	Deployment of Wasm Module in K8s.....	43
<b>4</b>	<b>Discussion and Conclusion .....</b>	<b>45</b>
4.1	Challenges Faced in Each Method .....	45
4.1.1	Challenges with crun Runtime .....	45
4.1.2	Challenges with containerd Runtime .....	45
4.2	Future Research Directions.....	45
4.3	Concluding Remarks .....	46
	<b>Bibliography .....</b>	<b>47</b>
	<b>Appendix .....</b>	<b>52</b>
	Appendix A.....	52
	Appendix B.....	65
	Appendix C.....	68
	<b>Eigenständigkeitserklärung .....</b>	<b>72</b>

# List of abbreviations

K8s	Kubernetes
Wasm	WebAssembly
CRI	Container Runtime Interface
OCI	Open Container Initiative
CNCF	Cloud Native Computing Foundation
CLI	Command Line Interface
WASI	WebAssembly System Interface

# Table of figures

Figure 1: A containerized environment architecture consists of containers running on a container runtime. The runtime interfaces with the host operating system, which is layered over the underlying infrastructure. ....	15
Figure 2: CLI interaction with container ecosystems: A user issues commands via a CLI to a high-level runtime, which then relays instructions to the low-level runtime, orchestrating direct container management and resource allocation. The high-level and the low-level runtime together make up the container runtime. ....	15
Figure 3: Interactions between User, CRI-O Daemon, a low-level container runtime, and containers [10]. ....	17
Figure 4: Interactions between User, containerd daemon, a low-level container runtime, and containers [10]. ....	18
Figure 5: Kubernetes architecture: Master node (API Server, etcd, Controller Manager, Scheduler) orchestrates workloads on worker nodes (Kubelet, container runtimes, pods, containers). ....	20
Figure 6: Kubelet communicates with a high-level container runtime over gRPC, which then manages containers using a low-level runtime. ....	24
Figure 7: Compilation of diverse programming languages to Wasm bytecode for platform-agnostic execution. ....	24
Figure 8: Layered architecture of WASI, highlighting libs and system call wrappers utilized by Wasm applications [39]. ....	27
Figure 9: High-level overview of the dependencies among the Operating System, Wasm runtime and Wasm Modules. ....	28
Figure 10: Worker node architecture with Kubelet, a high-level container runtime, crun, and a WebAssembly runtime, enabling the execution of Wasm workloads within a Kubernetes cluster. ....	30
Figure 11: Worker node architecture with Kubelet, containerd, containerd-shim, and a WebAssembly runtime, enabling the execution of Wasm workloads within a Kubernetes cluster. ....	31
Figure 12: The hybrid K8s cluster architecture with one master node and two worker nodes. The worker node 1 with CRI-O, crun, WasmEdge, and worker node 2 with containerd, containerd-shim for Fermion Spin, Fermion Spin, containerd-shim for runc and runc. ....	32
Figure 13: Organizational structure of the source code. ....	34
Figure 14: Successful execution of the <code>sudo kubeadm join</code> command, indicating the node has been added to the K8s cluster. ....	38
Figure 15: Output of the <code>kubectl get deployment</code> command, confirming successful Wasm workload deployment with three ready replicas. ....	39
Figure 16: Output of the <code>curl</code> command, confirming a successful REST request to a WasmEdge workload deployed within a K8s cluster. ....	39

Figure 17: Organizational structure of the source code. ....	40
Figure 18: Output of the <code>kubectl get deployment</code> command, confirming successful Wasm workload deployment with three ready replicas. ....	44
Figure 19: Output of the <code>curl</code> command, confirming a successful REST request to a Fermyon Spin workload deployed within a K8s cluster. ....	44



# List of tables

Table 1: List of required software and versions for compiling and building WasmEdge-compatible Wasm container images. ....	34
Table 2: List of key components and version for worker node 1.....	36
Table 3: List of required software and their versions for compiling and building Wasm images compatible with Fermyon Spin. ....	40
Table 4: List of key components and version for worker node 2.....	42

# 1 Introduction

WebAssembly, commonly known as Wasm, has revolutionised web applications. It allows resource-intensive programs like games, graphic design tools, and simulations – tasks once requiring separate software - to run seamlessly within web browsers. Popular examples of this transformation include Google Earth, AutoCAD, and Figma, all powered by Wasm [1].

However, the benefits of Wasm are not limited to web browsers. Its ability to run applications at near-native speed without excess overhead and vulnerabilities makes it an ideal candidate for server-side usage. With all its benefits, Wasm runtimes make it possible to run Wasm on the server side.

Introduced to the world in 2014 by Google, K8s has revolutionized the field of software deployment and containers [2]. By 2022, nearly half of organizations using containers had adopted K8s to run and manage their containers [3]. Modular in its design, K8s capabilities extend far beyond mere container orchestration, and virtually anything can be orchestrated with it, even Wasm.

By pairing K8s with Wasm, the benefits of both technologies can be harnessed, addressing many challenges that modern server-side applications face. The speed, minimal overhead, and compact bytecode of Wasm enhance K8s scaling and management capabilities. This thesis explores different ways to manage and orchestrate Wasm with K8s.

## 1.1 Background and Context

Server-side development has evolved rapidly in the last decade. The shift from monolithic architectures to microservices, the rise of cloud computing, and the adoption of containerization have changed how server-side applications are developed and deployed. These changes came with their complexities and challenges. These challenges underline the importance of Wasm and K8s in addressing modern server-side application challenges.

### **WebAssembly**

Wasm was developed to overcome the limitations of JavaScript for complex tasks in web browsers. Its binary instruction format offers faster parsing and execution and provides near-native performance. Its secure execution, compact binary format, and low cold boot time make it a perfect alternative to JavaScript for complex web applications.

### **WebAssembly on the Server Side**

Initially developed for the Web, Wasm has gained attention for server-side applications thanks to its low cold boot time and less computational overhead, making it a perfect candidate for serverless computing environments.

Solomon Hykes, the founder of Docker highlighted the potential of Wasm in a tweet:

*"If Wasm+WASI existed in 2008, we wouldn't have needed to created Docker. That's how important it is. Webassembly on the server is the future of computing. A standardized system interface was the missing link. Let's hope WASI is up to the task! "* [4]

His tweet underlines the significance of Wasm's capabilities beyond the browser, especially in server-side computing.

### **Challenges Addressed by WebAssembly**

In serverless computing, one of the significant challenges has been the high cold boot time, which results in high latency. Cloud providers like Amazon offer a Serverless Function as a service known as AWS Lambda, which can take up to one second to cold boot and consume resources in the idle state [5]. On the other hand, Wasm cold boots in less than one millisecond to a few milliseconds and consumes near zero resources in the idle state [6].

### **Kubernetes**

With the release of Docker in 2013, the microservice architecture and containers became popular. With the increase in the use of containers, managing them with Docker became complex, and the need for an orchestration tool arose. K8s comes to the aid of container management and orchestration problems. K8s manages, scales, and automates the deployment of containers. Open-sourced by Google in 2014, it has become the de facto standard for managing and orchestrating these containerized applications in a short time. It offers horizontal scaling, load balancing, and automated rollouts and rollbacks, among many other features.

### **Scaling WebAssembly with Kubernetes:**

K8s is modular in design, and its components can be exchanged with alternative components. Due to its modular design, it can be modified to orchestrate virtually anything, even virtual machines and Wasm workloads. Pairing K8s with Wasm represents a new and innovative approach to building and deploying Wasm-based applications. K8s can manage the orchestration of Wasm-based applications, taking advantage of Wasm's speed, security, and efficiency. Combining K8s with Wasm could offer the information technology industry the best of both worlds.

## **1.2 Thesis Objectives**

This thesis aims to explore, implement, and evaluate two methods of integrating Wasm with K8s, focusing on understanding the advantages and disadvantages of each method. The explorations aim to:

- **Identify Integration Techniques:** Investigate different strategies for integrating Wasm within K8s environments, including modifications to the K8s cluster.

- **Practical Implementation:** Apply these identified methods in practical scenarios, including setting up the K8s environment. Building and deploying the Wasm applications in the K8s cluster.
- **Evaluate Methodologies:** Critically analyse each integration technique, assessing its practicality. This evaluation will focus on how well these methods integrate Wasm modules within K8s.
- **Contribution to Future Research:** Outline areas for future research and potential development in Wasm and K8s integration.

### 1.3 Thesis Overview

In the Literature Review, several critical topics are discussed to understand this thesis's relevant technologies and concepts: It explains container technology, focusing on how container images and runtime function. The architecture of K8s, a system for managing these containers, is also explored, highlighting its role in automating and simplifying application management. Additionally, the review covers Wasm, detailing its use in executing code across various platforms, particularly in server-side applications. Finally, the section investigates how Wasm can be integrated with K8s to enhance application execution, discussing approaches like using `crun` and `containerd` for running Wasm workloads in K8s environments. This overview establishes a foundational understanding for the subsequent practical implementation in the Materials and Methods section.

The Materials and Methods section outlines two methods for setting up and deploying applications using Wasm and K8s. The first method details the execution of a Wasm workload using the `crun` runtime, which involves building, deploying, and running a REST API microservice written in Rust on a K8s worker node. `crun`, a low-level container runtime, is adapted to run containers and Wasm workloads through specific Wasm runtimes. The second method employs the `containerd` runtime for executing a similar Wasm workload. Here, the focus shifts to using `containerd`, `containerd-shim`, and `Fermyon Spin` to build and deploy the microservice on a different worker node. This method leverages `containerd`'s modular architecture and custom `containerd`-shims developed for managing Wasm workloads. Both methods culminate in demonstrating and discussing the functioning of the deployed applications within the K8s environment, showcasing the practical application of the theoretical concepts discussed in the earlier sections.

Finally, the chapter Discussion and Conclusions delve into the advantages, benefits, and potential drawbacks of the two methods used for integrating Wasm with K8s. It critically assesses how effectively the objectives of the thesis were achieved while acknowledging its limitations. Additionally, this section provides thoughtful recommendations for future research directions, building upon the findings and insights gained from this study.

The thesis concludes with a References section, offering a detailed list of all the sources and materials referenced throughout the research, and an Appendix that includes supplementary information and data pertinent to the research.

## 2 Literature Review

This section provides a brief overview of the concepts and topics that form the foundation of this thesis and its subsequent experimental chapters, including Materials and Methods.

### 2.1 Containers

A container is a standard unit of software that encapsulates the application code along with its configuration files, libraries, and binaries. This ensures that the application can run reliably in various computing environments [7].

#### 2.1.1 Container Image

A container image is the blueprint of containers. It contains all the components for an application to run, including the binaries, libraries, system tools, and configuration parameters. During runtime, these container images become containers [7].

Container images are built using a set of instructions, typically defined in a plain text document file known as Dockerfile or Containerfile. Tools for building container images, such as Buildah and Docker, read these instructions from the Dockerfile/Containerfile to build the image.

#### 2.1.2 Container Runtime

Section 2.1.1 discussed the basics of a container image. Now, a question arises: what or who runs and manages the container? This is where container runtimes come into play.

Container runtimes are responsible for the entire lifecycle of a container. They pull images from container repositories, create containers from these images, and manage the starting, stopping, and deletion of containers [8].

Figure 1 illustrates the layered architecture of containerized applications and their operational environment. At the top layer are containers, each containing applications, application configurations, binaries, and libraries used by those applications. Beneath the containers is the container runtime layer. The container runtime is the backbone of container operations and is responsible for various tasks, such as container lifecycle management, networking, storage, and interfacing with the host operating system [9] [10]. Below the container runtime layer is the host operating system, which supports the container runtime by providing the necessary kernel and subsystems.

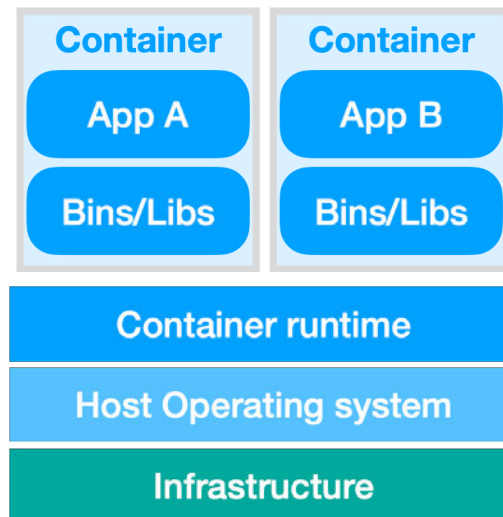


Figure 1: A containerized environment architecture consists of containers running on a container runtime. The runtime interfaces with the host operating system, which is layered over the underlying infrastructure.

Container runtimes can be categorized into high-level container runtimes and low-level container runtimes. These two container runtimes work together to manage and run containers.

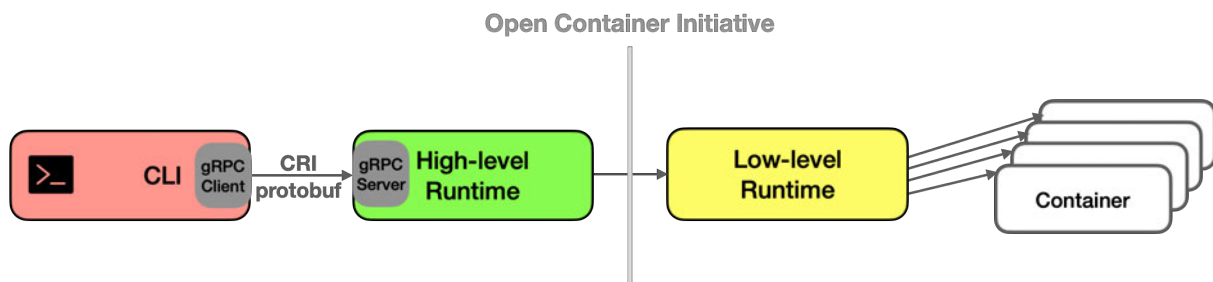


Figure 2: CLI interaction with container ecosystems: A user issues commands via a CLI to a high-level runtime, which then relays instructions to the low-level runtime, orchestrating direct container management and resource allocation. The high-level and the low-level runtime together make up the container runtime.

Figure 2 illustrates the communication between the container runtimes. The figure shows that the CLI communicates with the high-level container runtime. The high-level container runtime then communicates with the low-level container runtime to run the containers [10] [11].

### High-Level Container Runtime and Container Runtime Interface

As can be seen in Figure 2 the User is not communicating directly with the containers through CLI. Instead, the high-level container runtime, such as containerd or CRI-O, bridges the CLI and the low-level container runtime. Typically, high-level container runtime provides a daemon application, a CLI client and an API to interact with it. This API has been standardized and is known as Container Runtime Interface [10].

The Container Runtime Interface, also known as CRI, is a protocol defined and standardised by K8s. It enables K8s to interact with various container runtimes, including containerd and CRI-O, without

recompiling K8s components. The CRI defines the primary gRPC protocol for the communication between the Kubelet and container runtime [11].

### **Low-Level Container Runtime and Open Container Initiative**

Low-level container runtimes, such as runc or crun, focus on running containers, setting up namespaces, and cgroups [12]. These runtimes implement the Open Container Initiative's<sup>1</sup> (OCI) specifications. When a container needs to be created, the high-level container runtime sends a set of instructions in JSON format, which are aligned with OCI specifications, to the low-level container runtime. Low-level container runtime use these instructions to run the container, including setting up namespaces and mounting the container's filesystem.

### **Working Mechanisms of High-level and Low-level Container Runtimes**

Understanding the roles and responsibilities of high-level and low-level container runtimes shows how they work together to offer a complete solution. However, how do these runtimes work together?

This section covers the working mechanisms of two high-level container runtimes: containerd and CRI-O. It also discusses the role of low-level container runtimes. Understanding the mechanism between these components will provide a deeper understanding of the container runtimes and container ecosystem.

Note: This thesis will not go into the details of every low-level container runtime, such as crun and runc. They function very similarly, so this section will refer to them collectively as “low-level container runtimes”.

### **CRI-O: High-Level Container Runtime**

CRI-O is a CNCF-graduated, open source, lightweight, high-level container runtime. It implements the CRI protocol, is compatible with OCI-compatible container runtimes. CRI-O is a community-driven project maintained by an open-source community, along with maintainers and contributors from Red Hat, Intel, Suse, Hyper and IBM [13].

---

<sup>1</sup> The Open Container Initiative (OCI) is a Linux Foundation project started in 2015 to create open standards around container formats and runtimes [62].



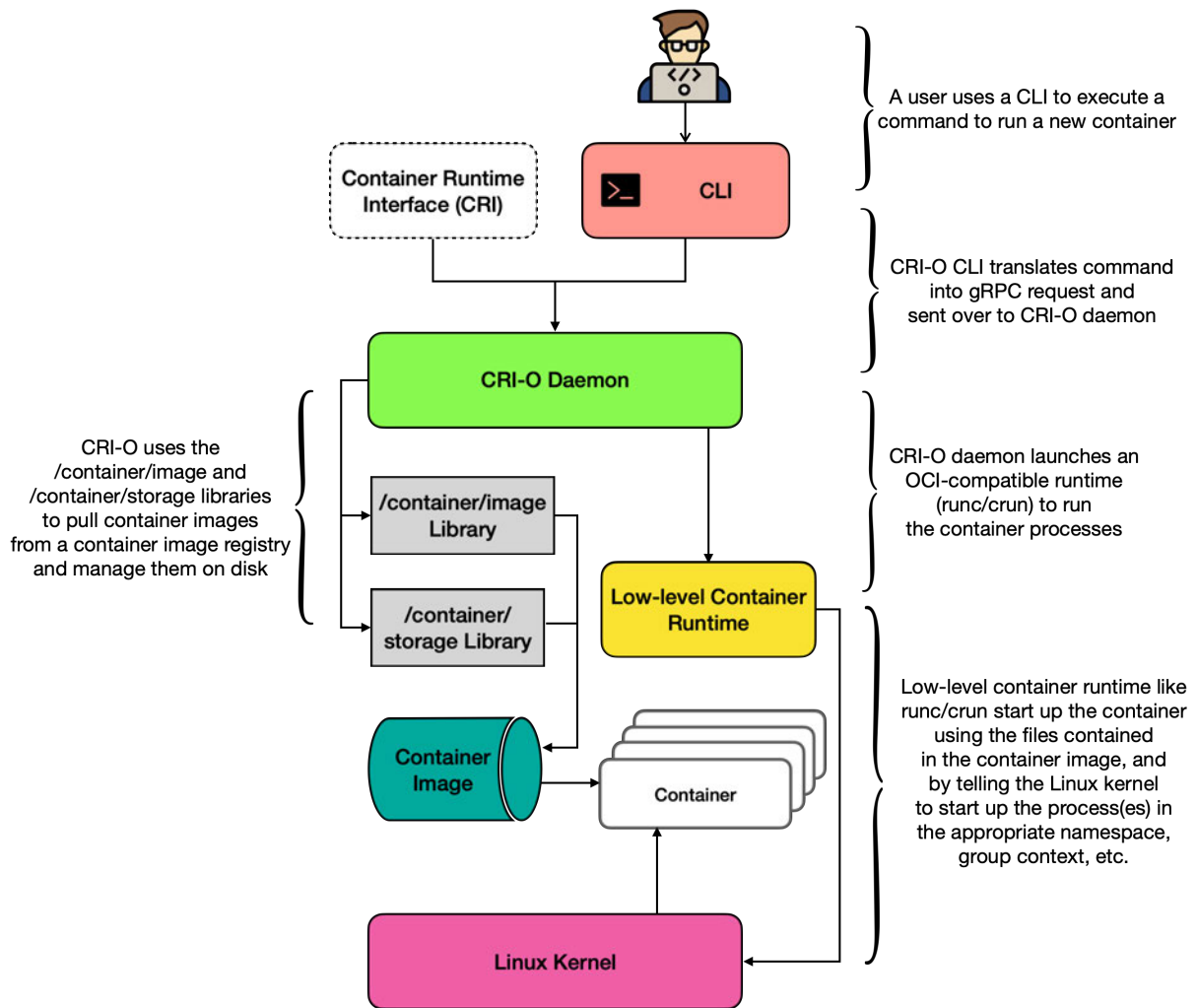


Figure 3: Interactions between User, CRI-O Daemon, a low-level container runtime, and containers [10].

Figure 3 shows the integration of the CRI-O runtime within the container ecosystem, highlighting its interactions with adjacent components.

- The User interacts with CRI-O through CLI to create a container.
- CRI-O checks if the container image already exists in the cache. If not, it pulls the container image from the container image registry [10] [13].
- CRI-O unpacks the container image into the container's root filesystems using the containers/storage library [13].
- Once the container's root file system (rootfs) is set up, CRI-O generates an OCI runtime specification JSON file with all the details to run the container [13].
- CRI-O then launches the OCI-compliant container runtime and passes the OCI-compliant JSON file to the low-level container runtime [14].
- Low-level container runtime starts the container from the container's root file system and requests the Linux kernel to create a namespace, group, context, and other relevant configurations [14] [15].

## containerd:

containerd is a high-level container runtime that manages the container lifecycle. It is an industry standard, CNCF-graduated runtime designed to be part of a more extensive system rather than being directly used by developers or end-users. Furthermore, it serves as the core container runtime for Docker [16].

Unlike CRI-O, containerd has a lightweight intermediary process known as containerd-shim, which acts as a bridge between the low-level container runtime and containerd itself. The containerd-shim manages the lifecycle of the filesystem of the container. It is responsible for mounting the filesystem when the container is created and unmounting it upon removal of a container [17].

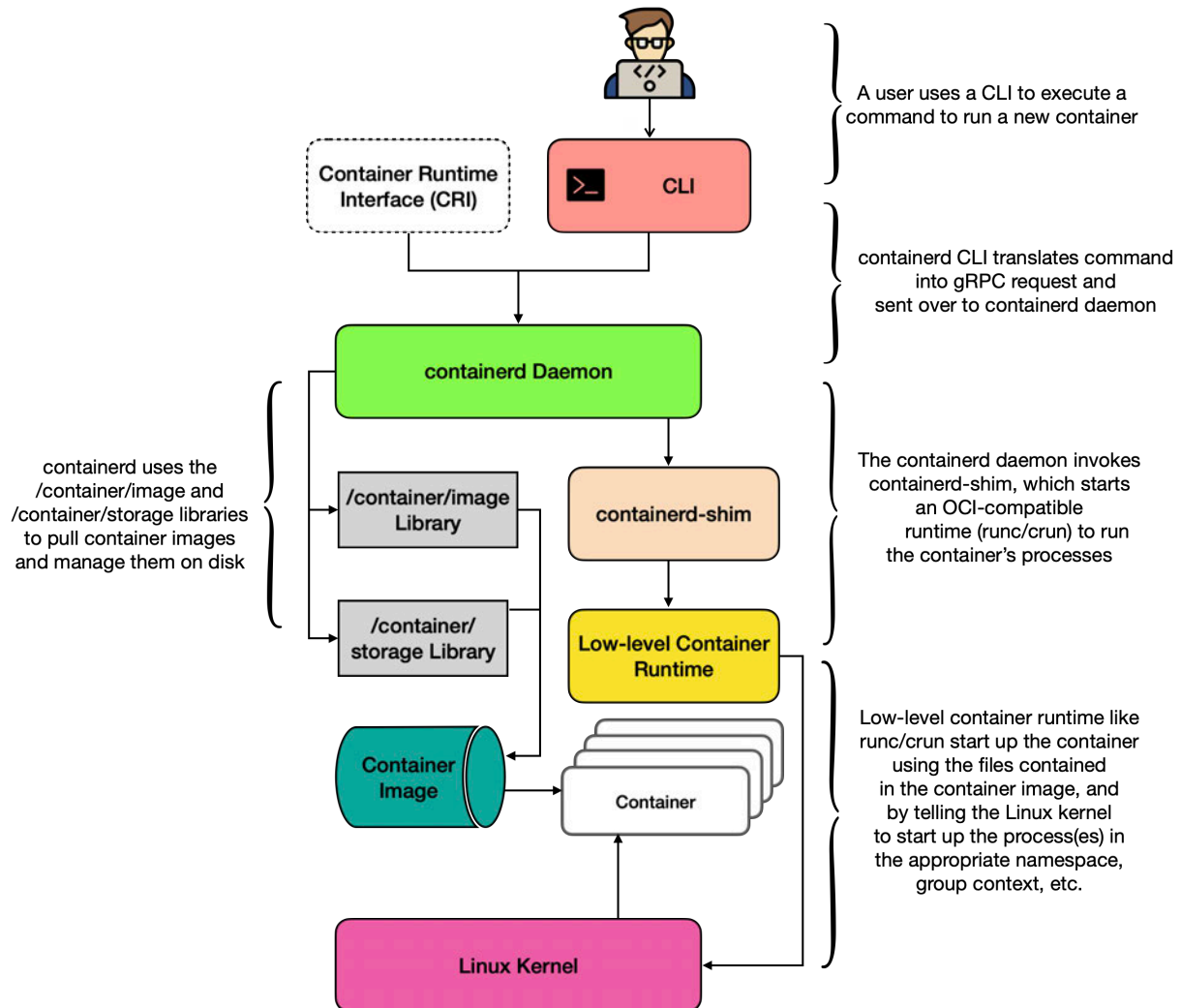


Figure 4: Interactions between User, containerd daemon, a low-level container runtime, and containers [10].

Figure 4 shows the containerd, illustrating how it and containerd-shim interacts with other components.

- The User interacts with containerd through CLI to create a container.
- containerd checks if the container image already exists in the cache. If not, it pulls the container image from a container image registry [10].

- containerd unpacks the container image into the container's root filesystems using the containers/storage library [14].
- Once the container's root file system (rootfs) is set up, containerd generates an OCI runtime specification JSON file with all the details required to run the container [10].
- containerd then uses the containerd-shim to communicate with the OCI-compliant container runtime (runc, by default) and passes the OCI-compliant JSON file to it [14].
- Low-level container runtime starts the container from the container's root file system and requests the Linux kernel to create a namespace, group, context and other relevant resources and configurations [14].

## 2.2 Kubernetes: Architecture and Components

Section 2.1 covers the concepts of containers, container images, container runtimes, and their workings. It is quick to deploy containers. However, as the number of containers increases, deploying and maintaining them manually becomes difficult. This is where container orchestration tools like K8s come into play.

Kubernetes is a portable, extensible, open-source platform for managing containerized applications and services. It simplifies the container management process by using declarative configuration and automation tools. Google developed it, released it as an open-source project in 2014, and later contributed to the CNCF [2].

K8s is based on master-slave<sup>2</sup> architecture and is divided into two main parts: the master node, also known as the Control Plane, and the worker nodes. The master node serves as the central control point of the K8s cluster and is responsible for managing the worker nodes and the state of the entire cluster. Worker nodes run the containers and report to a master node [18].

Understanding K8s requires a thorough knowledge of its architecture. Figure 5 illustrates the architecture of K8s, detailing its components and how they interact to manage containerized applications.

---

<sup>2</sup> This document uses the term master-slave to refer to a widely recognized architectural model in computer science. It is a technical term that describes the relationship between components in a system. This usage is strictly in a technical context and is not intended to be offensive or to trivialize historical and social issues associated with certain terms. The author acknowledges the sensitivity of these terms and their historical connotations.

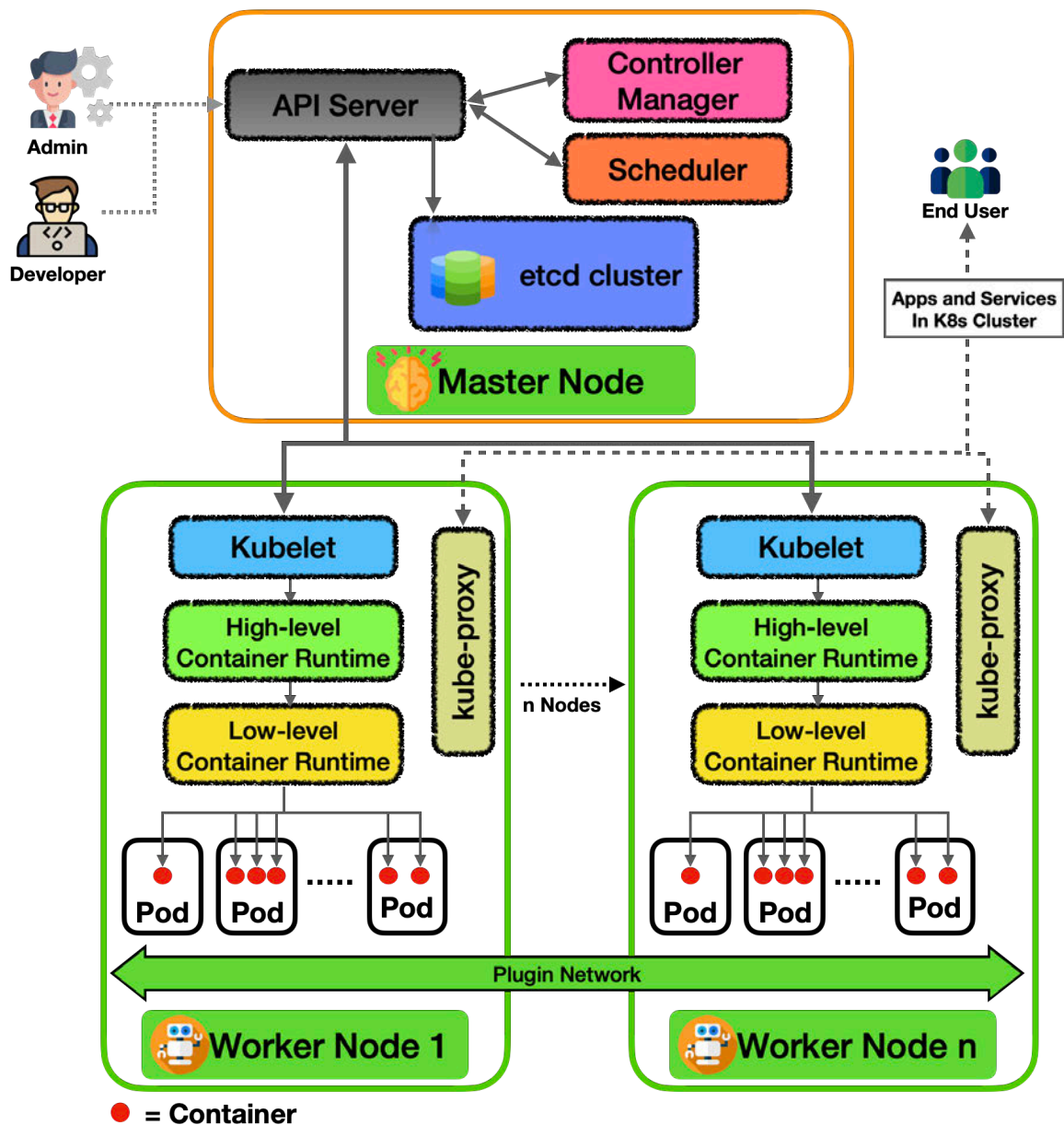


Figure 5: Kubernetes architecture: Master node (API Server, etcd, Controller Manager, Scheduler) orchestrates workloads on worker nodes (Kubelet, container runtimes, pods, containers).

In a high-availability cluster, there are multiple master and worker nodes to increase the system's availability and reliability. In the case of a failure of one or more master nodes in a high-availability cluster, the cluster remains operational.

K8s supports several autoscaling methods to manage application scaling and responding to changes in load [19]:

- **Vertical Pod Autoscaling:** This method involves allocating additional resources, such as CPU or RAM, to manage increased load. This ensures that applications have the necessary resources to run effectively [20].

- **Horizontal Pod Autoscaling:** This method involves deploying additional Pods to manage the increased load. If the load decreases, the number of Pods is reduced to the minimum, as configured by the system administrator [20].
- **Cluster Autoscaler:** This method involves changing the size of the K8s cluster based on the intensity of workload; it automatically adds nodes to the cluster when there are applications that do not have enough resources to run effectively and removes the nodes when they are not needed anymore [21].

### 2.2.1 Pods

Pods are the smallest deployable units of computing that can be created and managed in K8s. Each pod is a collection of one or more containers. As shown in Figure 5, a pod operates on a node, and a node can host multiple pods [22].

Every pod is assigned a unique IP address. All containers within a pod share the same IP address and network ports. Containers within a pod use the same network namespace and can communicate with each other using `localhost` [22].

Pods are intended to function as stateless and disposable entities. They do not survive scheduling failures, node failures, or evictions due to lack of resources. To address these issues, K8s has higher-level controller abstractions, namely `ReplicaSets`, `Deployments`, and `StatefulSets`. These controllers manage the lifecycle of pods, including their initiation and termination. Further information on controller manager is given in 2.2.6.

### 2.2.2 API Server

The K8s API Server, also known as `kube-apiserver`, is a core component of the K8s master node. It acts as the front-end for K8s and exposes an HTTP API that allows end users, different components of the K8s cluster, and external components to communicate with one another [23] [24].

The K8s API Server is stateless and stores its state in an external database like `etcd`. Its statelessness makes it possible to replicate it to handle request loads and enhances fault tolerance. In a highly available K8s cluster, there are multiple master nodes and multiple replications of the API server [24].

### 2.2.3 Kubernetes Objects and Kubectl

K8s objects are the fundamental building blocks of the K8s ecosystem. They represent specific aspects of cluster's desired state. As the Kubernetes documentation states: "*Kubernetes objects are persistent entities in the Kubernetes system.*" [25].

K8s objects can be created, modified, or deleted by calling the K8s API Server over HTTP. A K8s object can be created by providing a specification that describes its desired state and critical details, such as a

name, to the K8s API Server. This specification should then be sent to the K8s API Server in JSON format in the REST API request body. The easiest way to communicate with the K8s API Server is by using the `kubectl` command-line interface and providing the information in a YAML file, known as a manifest file. The `kubectl` CLI reads the manifest file, generates a REST API request with a JSON-formatted specification in the request body, and makes a request on behalf of the user [25].

Listing 1: YAML manifest defining a K8s pod named `nginx-pod` with a single container running the latest version of the Nginx web server. It also includes a label `app: nginx` for identifying and managing the pod.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx:latest
```

The above configuration has the following key aspects:

- `apiVersion`: Determines the version of the K8s API is used to create the object.
- `kind`: Determines the type of K8s object being created, modified, or deleted.
- `metadata`: Consists of the name, labels, and namespace associated with the object.
- `spec`: Consists of the specifications of the K8s object. The above example consists of the specifications for containers that will run inside the pod, including the Container image to be used.

The manifest in Listing 1 can be applied to the K8s cluster by saving the configuration as `nginx-pod.yml` and execute following command:

```
$ kubectl apply -f nginx-pod.yml
```

## 2.2.4 ETCD

In a multi-master node cluster, K8s control plane components have more than one instance. To maintain a consistent cluster state, Kubernetes requires a horizontally scalable database to persist this information. This is where distributed databases, like `etcd`, come into play [26].

etcd is an open-source distributed key-value store. It is fast, secure, can scale horizontally and fully replicate across all nodes and is designed for high availability, ensuring no single point of failure. It serves as the primary key-value store in official K8s [26].

### **2.2.5 Scheduler**

The K8s scheduler, or `kube-scheduler`, is the default scheduler for K8s. It selects a feasible node for a newly created unscheduled pod. It does this by filtering out nodes that do not meet a pod's specific scheduling needs, scoring the feasible nodes, and selecting a node with the high score. It then informs the API server about its decision, and the API server contacts the Kubelet of that specific node to run the container [27].

### **2.2.6 Controller Manager**

The K8s controller manager is a daemon that includes the core control loops shipped with K8s and runs as part of the control plane [28].

K8s controllers are based on the concept of control loops, which monitor and regulate the state of the cluster. Each K8s controller tracks at least one K8s resource type. The controller's purpose is to match the current state of the cluster to the desired state of the cluster. If a discrepancy is found, the controller takes action by issuing specific directives to the API server [29].

A prime example is the deployment controller, which orchestrates the deployment and updating of applications within the cluster. It creates, deletes, and updates Pods based on the specifications defined in the Deployment object's desired state.

### **2.2.7 Kubelet**

Kubelet is one of the key components of K8s, acting as a bridge between the K8s control plane and each worker node. It is responsible for managing and executing containers on worker node [30].

The connection between Kubelet and the API server is bidirectional. This connection is used for pod scheduling, configuration, health checks, and log fetching. Through this connection, Kubelet also sends regular updates to the API server about its node and the Pods it manages, including resource metrics and container states.

When a pod is scheduled to run on a specific node, the API Server provides the Kubelet with Pod Specifications files. Kubelet cannot run the containers by itself. It does it with the help of container runtimes. It runs containers on the node by interacting with the container runtime through CRI.

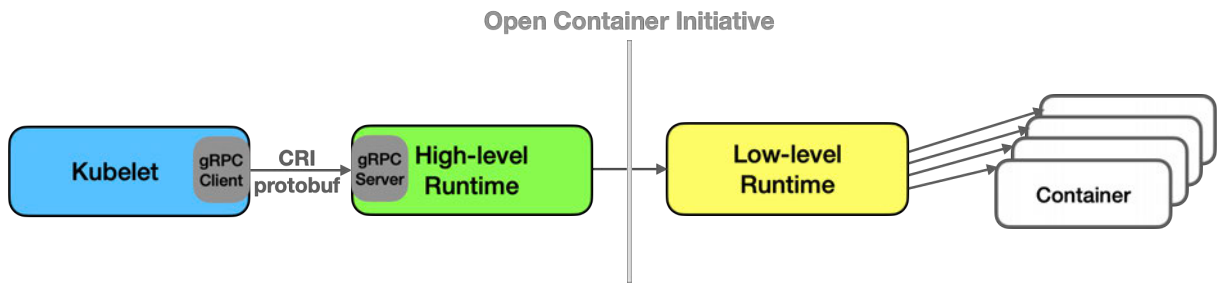


Figure 6: Kubelet communicates with a high-level container runtime over gRPC, which then manages containers using a low-level runtime.

Figure 6 illustrates the communication between the Kubelet and the container runtimes. It shows that Kubelet interacts with high-level container runtime through CRI. High-level container runtime interacts with OCI-compliant low-level container runtime to run the containers [10] [11].

## 2.3 WebAssembly

WebAssembly, also known as Wasm, is a binary format created for a stack-based virtual machine, enabling code execution in multiple programming languages across different platforms. Initially intended for web browser, Wasm has increasingly become famous as a server-side technology, allowing developers to create practical and scalable backend applications.

The adoption of Wasm for server-side purposes stems from the demand for high-performance, resource-saving, secure, portable, and easily manageable applications capable of managing substantial requests and data volumes. Conventional server-side technologies like PHP, Ruby, and Python are commonly used. However, they can frequently be slow and consume significant resources, especially when handling high data processing or machine learning tasks.

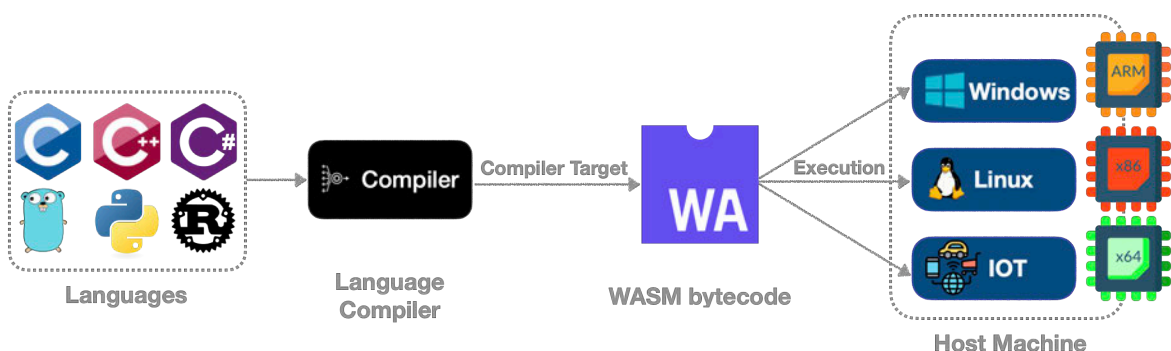


Figure 7: Compilation of diverse programming languages to Wasm bytecode for platform-agnostic execution.

Figure 7 demonstrates the primary benefit of Wasm: the “Write once, run anywhere” philosophy. Different programming languages can be compiled into Wasm bytecode. This Wasm bytecode can run efficiently across various machines and platforms, such as Windows, Linux, and Internet of Things (IoT) devices, regardless of the underlying CPU architecture (ARM, x86, x64).



Wasm offers several benefits for server-side development. The most notable benefits are discussed in the next section.

### **Flexibility**

Wasm's flexibility is significant in its increased usage and recognition among the developer community. Several factors allow developers to create server-side applications that are modular, scalable, and versatile.

- a) **Language Independence:** As shown in Figure 7, Wasm is designed as a compilation target rather than an independent language, enabling it to be language-agnostic and compatible with various programming languages, such as C, C++, Rust, Go, and more [31]. Its Language-agnosticism allows developers to program server-side applications in their preferred language while taking advantage of its performance and efficiency.
- b) **Integration with Server-Side Technologies:** Wasm can be integrated with server-side technologies like Node.js and developers can take advantage of the performance gains offered by Wasm while utilizing established tools and frameworks [32].

### **Portability**

The portability of Wasm is a crucial feature that has contributed to its widespread adoption and popularity as a compilation target for web and server-side applications. There are several factors which make Wasm highly portable across different platforms and environments:

- a) **Platform-Independent Bytecode:** Wasm is designed as a low-level assembly language, and Wasm bytecode can be executed on different operating systems and hardware architecture without recompiling [33].
- b) **Consistent Runtime Environment:** Wasm is a standardized technology which provides a consistent and well-defined set of instructions. This standardization ensures that Wasm runtimes and browsers consistently interpret the instructions, irrespective of the specific hardware or operating system. When the hardware or operating system does not support certain features, the Wasm runtime can simulate them, ensuring a consistent execution environment for the application [34].

### **Performance**

One of the most notable benefits of Wasm in server-side applications is the vastly improved performance compared to traditional methods. Wasm's performance can be attributed to several key factors, which collectively lead to quicker response times and effective request handling, even during heavy workloads. These aspects make Wasm a perfect option for server-side environments, where swift responses and effective request management are crucial. The following aspects illustrate, in more detail, factors that result in the better performance of Wasm:

- a) **Compact Binary Format:** The binary format of Wasm enables rapid parsing, decoding, and implementation, resulting in decreased loading times and enhanced server-side application responsiveness. The compactness of binary facilitates quicker code transmission across networks and contributes to faster start-up times, making server-side applications agile and efficient [35].
- b) **Designed for Modern Hardware:** Wasm is engineered to work closely with current hardware architectures, allowing it to utilize modern processors' abilities fully [33]. Its enhanced optimization could lead to better performance in tasks that demand high computation, like data analysis, machine learning, and real-time analytics, which are frequently needed in server-side applications.
- c) **Efficient Memory Management:** Wasm offers a linear memory model, making memory handling more straightforward and diminishing the usual garbage collection overhead found in other languages, such as JavaScript. This efficient memory management enhances the overall performance of server-side applications by decreasing the chances of memory leaks and problems associated with memory leakages [35].
- d) **Real-World Performance Improvements:** Many real-life examples show the advantages of employing Wasm for server-side environments. For example, the Lucet project by Fastly highlights the application of Wasm in developing high-performance, low-latency server-side applications for edge computing. Shopify has also used Wasm to create a server-side sandbox, enabling users to execute personalized code securely within their e-commerce platform and enjoy the performance improvements provided by Wasm [36] [37].

## **Security**

Wasm security features are essential for its design and execution, making it a popular choice for web-based and server-side applications. Wasm offers multiple elements which contribute to increased security:

- a) **Sandboxed Execution Environment:** Wasm modules run within a sandboxed execution environment that remains isolated from the host system. This isolation protects the host system and other Services running on the host system [38].
- b) **Memory Safety:** Wasm allows direct memory access while maintaining strict boundaries, preventing the Wasm module from accessing memory out of its bounds. This built-in security makes Wasm much more secure than existing technologies [38].

- c) **Control-Flow Integrity:** Wasm modules must declare all functions and their types at load time, even when dynamic linking is used. This prerequisite enables the enforcement of control-flow integrity through structured control-flow, which is created during the compilation process. Since compiled code cannot be modified and observed during runtime, Wasm programs are safe protected by default against control-flow hijacking attacks. This built-in security feature enhances web application’s safety and reliability, ensuring that Wasm does what it is intended to [38].

### 2.3.1 WebAssembly System Interface

Initially, Wasm was designed to execute high-performance applications in web browsers. The Wasm bytecode interacts with the JavaScript engine on the web and utilizes Web APIs to communicate with the browser environment. A defined set of APIs is required to operate outside of the web environments, such as on servers or embedded devices. This is where WebAssembly System Interface (WASI) comes in [39].

WebAssembly System Interface provides a standardized set of APIs that a Wasm module can use to perform system-level operations, such as accessing files or making network requests [39].

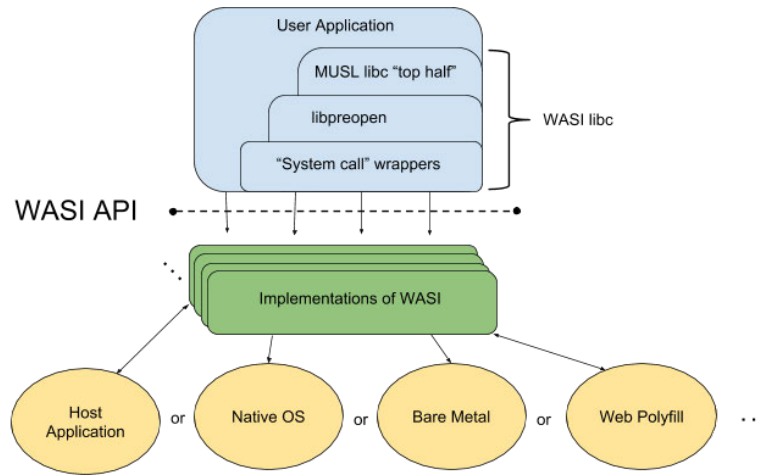


Figure 8: Layered architecture of WASI, highlighting libs and system call wrappers utilized by Wasm applications [39].

Figure 8 shows the software architecture of WASI. At the highest level, there is application, libraries, and more. To facilitate the use of WASI API, there is an implementation called WASI libc<sup>3</sup>. It uses a libpreopen<sup>4</sup> layer and a system call wrapper to communicate with the WASI implementation, interacting with various resources such as host application, native OS, bare metal, or JS runtime [39].

<sup>3</sup> WASI libc is musl-based implementation of the standard C library (libc) that is designed to work with the WASI [59].

<sup>4</sup> libpreopen is a C library designed to help compartmentalize applications, including Wasm applications, by pre-opening file and directory descriptors before they get sandboxed. Wasm applications run in sandbox and libpreopen allow them to perform necessary file operations without requiring broad system permissions [61].

### 2.3.2 WebAssembly Runtime

In previous Section 2.3.1, the WASI was discussed. Now, a crucial question arises: Who is responsible for running the Wasm module? This is where Wasm runtimes come into play.

Wasm runtime interprets or compiles Wasm binary code into machine code. It provides a sandboxed executing environment and a set of WASI APIs with which Wasm modules can interact.

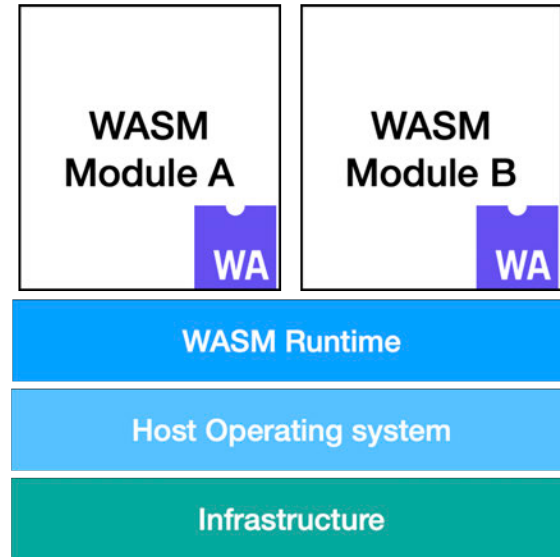


Figure 9: High-level overview of the dependencies among the Operating System, Wasm runtime and Wasm Modules.

Figure 9 provides a comprehensive, high-level depiction of the interdependencies among the operating system, Wasm runtime, and Wasm workloads. The operating system is the fundamental platform on which all other components operate. It provides services like managing hardware resources and system services. Above it is the Wasm runtime, an intermediary between the operating system and Wasm modules. Wasm runtime interprets and translates the bytecode into machine code. Finally, Wasm modules are files which contain Wasm bytecode. All these layers work together to execute Wasm applications efficiently on the host system.

There are numerous Wasm runtimes available for different use cases. These runtimes vary in their performance, security features, and supported APIs. In this thesis, two specific Wasm runtimes will be explored in detail: Fermion Spin and WasmEdge.

#### WasmEdge

WasmEdge is a lightweight, high-performance, and extensible Wasm runtime. It is designed for cloud-native, edge, and decentralized applications, including serverless apps, embedded functions, microservices, smart contracts, and IoT devices. It supports a wide range of programming languages and a wide variety of WASI-like extensions, enabling features such as network sockets, async processing, TensorFlow inference, key-value stores, database connectors, and resource control [40].

## **Fermyon Spin**

Fermyon Spin is a framework used for building Wasm microservices and web applications. It is designed for creating and running event-driven microservice applications with Wasm components. Like WasmEdge, it also supports various programming languages and libraries. It also offers data persistence with Postgres, Redis and file storage [41].

## **2.4 WebAssembly Meets Kubernetes**

In the previous sections, containers, K8s, and Wasm were discussed. In this section, the focus shifts towards combining these technologies for efficient application execution. This thesis explores two methods to run Wasm workload in a K8s environment: the first involving crun and the second with containerd.

### **2.4.1 Approach 1: crun**

crun is an OCI-compatible low-level runtime that natively supports running Wasm workloads through Wasm runtime. It is compatible with three well-known container runtimes: WasmEdge, Wasmer, and Wasmtime.

Figure 10 shows the K8s worker node running Wasm workloads with crun. The architecture is like that of a K8s worker node running containers. The key difference is that instead of creating a container from a container image, crun executes Wasm runtime. The Wasm runtime then creates a sandbox environment and executes the Wasm workloads.

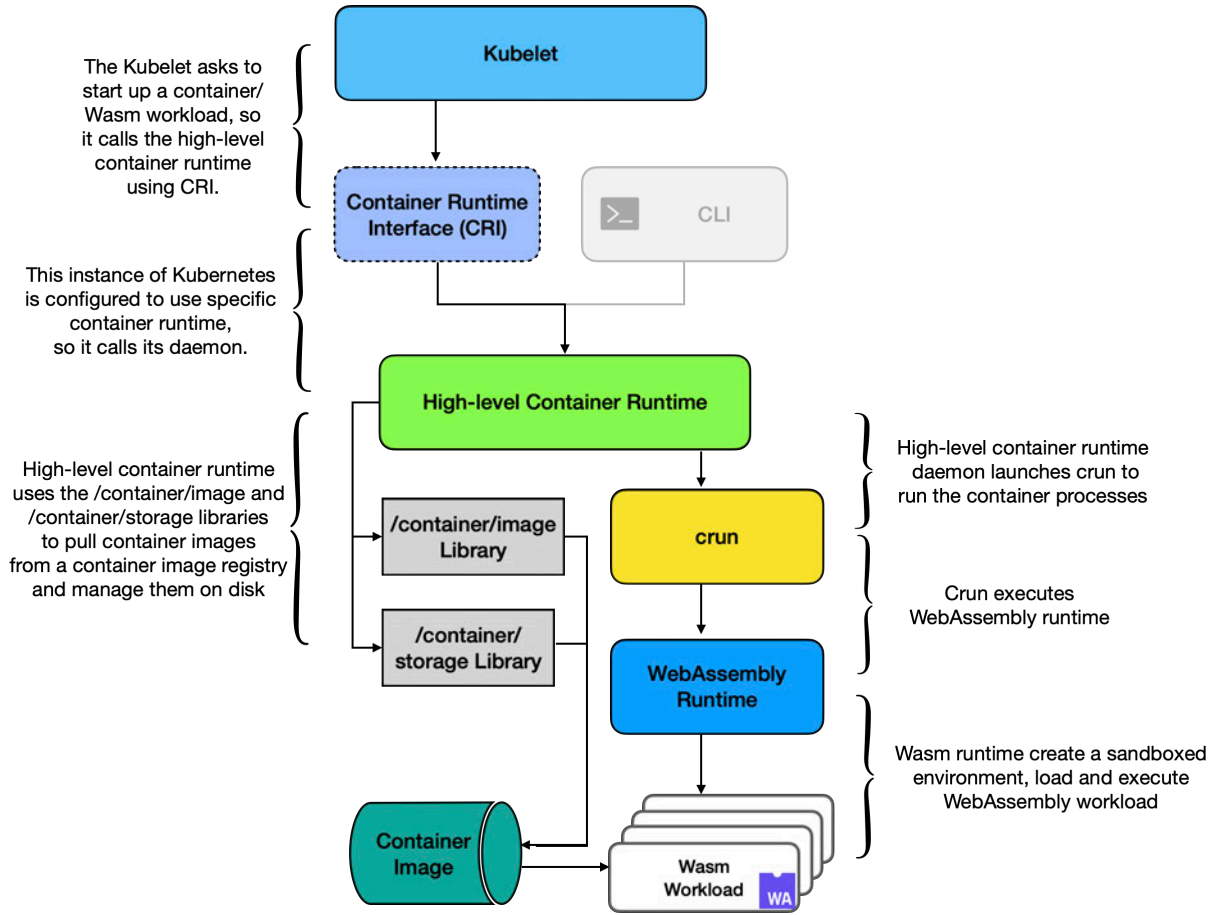


Figure 10: Worker node architecture with Kubelet, a high-level container runtime, crun, and a WebAssembly runtime, enabling the execution of Wasm workloads within a Kubernetes cluster.

In the described architecture of a K8s worker node running Wasm workloads with crun, a critical consideration emerges: How does crun distinguish between a container image and an image containing a Wasm workload? The key lies in the use of annotations. Before creating a container from an image, crun checks the metadata passed by the high-level container runtime and searches for the `run.oci.handler=wasm` annotation in the metadata of the image. This annotation signals to crun that the workload should be run using a Wasm handler rather than the standard handler [42].

## 2.4.2 Approach 2: containerd

containerd is an open-source, industry-standard runtime. It does not support Wasm out of the box. But its modular architecture allows it to be extended with custom containerd-shims.

To enable containerd to run and manage Wasm workloads, a specialized custom containerd-shim is needed. Such a containerd-shim can be developed with the help of the runwasi project. runwasi is intended to be consumed as a library to develop containerd-shim. A custom containerd-shim can be developed to work as a mediator between containerd and Wasm runtime [43].

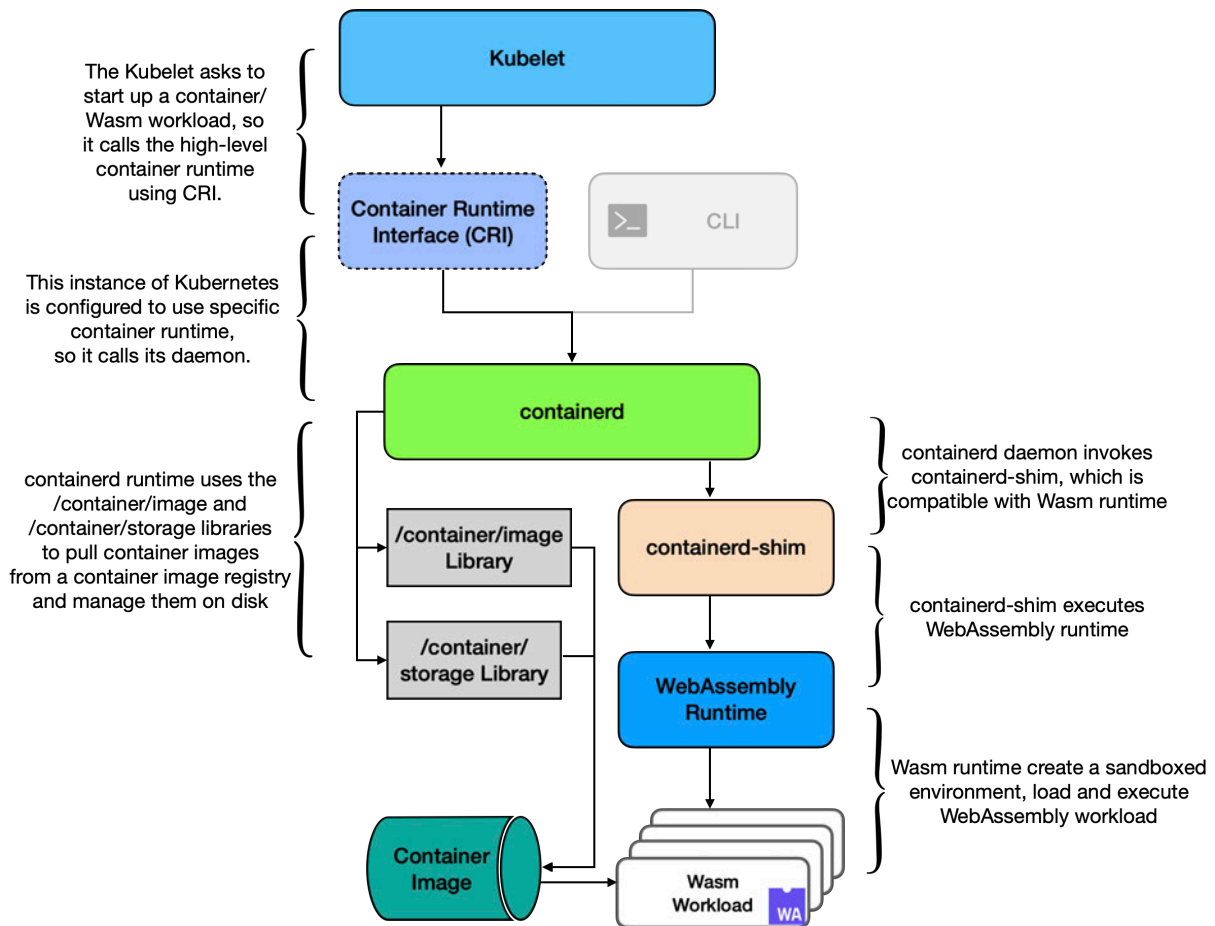


Figure 11: Worker node architecture with Kubelet, containerd, containerd-shim, and a WebAssembly runtime, enabling the execution of Wasm workloads within a Kubernetes cluster.

Figure 11 shows the K8s worker node running Wasm workloads with containerd and a custom containerd-shim, compatible with Wasm runtime. The key difference from a traditional container setup is that instead of calling a standard containerd-shim designed for running containers, a special shim, compatible with Wasm runtime, is invoked. This special containerd-shim interacts with the Wasm runtime, which is responsible for creating a sandboxed environment for running the Wasm Workload [43].

The question arises: How does containerd know which containerd-shim to use to create a container or execute Wasm modules? This is where runtime Handler Configuration and a K8s feature called `RuntimeClass` comes into play. Runtime Handler is specified for each low-level container runtime in the high-level container runtime configuration. A K8s `RuntimeClass` resource object is then created through the K8s API Server, which consists of the name of the `RuntimeClass` resource and the name of the handler specified in the configuration of high-level container runtime. The name of the `RuntimeClass` Resource is then used in the manifest file, which deploys the workload in the K8s cluster [44].

### 3 Materials and Methods

This chapter builds upon the theory discussed in Section 2.4, covering the practical aspects of the methods for integrating Wasm within a K8s environment.

The practical part begins with the compilation and containerization of a Wasm workload, including setting up the development environment. It then provides a guide on modifying K8s nodes for each method to support Wasm execution. Finally, it covers deploying the containerized Wasm workload onto the K8s cluster, which includes modifying the K8s environment, creating manifest files, and managing the deployment process.

This chapter demonstrates two practical methods using a K8s setup with one control plane and two worker nodes. Each node is configured differently to run the Wasm workload: Node 1 uses crun with WasmEdge, and Node 2 uses containerd, containerd-shim, and Fermyon Spin. Figure 12 illustrates the components of a K8s cluster with modification to run the Wasm workload.

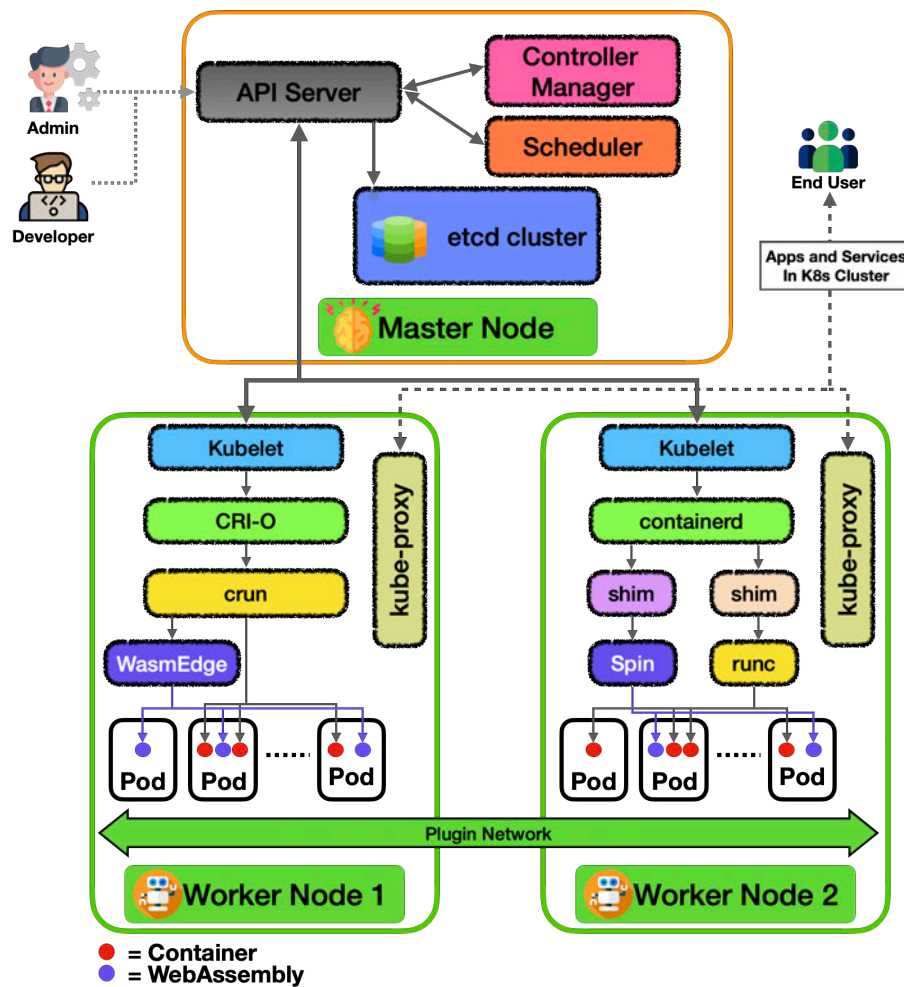


Figure 12: The hybrid K8s cluster architecture with one master node and two worker nodes. The worker node 1 with CRI-O, crun, WasmEdge, and worker node 2 with containerd, containerd-shim for Fermyon Spin, Fermyon Spin, containerd-shim for runc and runc.



### 3.1 Prerequisites

This thesis focuses on installing and configuring necessary components on the two worker nodes to enable Wasm functionality within an existing K8s cluster. Setting up a K8s master node is beyond the scope of this work. The following prerequisites must be met:

- **Worker Nodes:** Two Virtual Machines or physical machines running Ubuntu 20.04. These machines will be configured and joined to the existing K8s cluster.
- **Pre-Configured K8s Master Node:** This thesis assumes a functional K8s master node running K8s version 1.28.2.

If required, a script is provided in Appendix A to automate the installation and configuration of Kubelet, kubectrl, containerd, and runc. This script also initializes the K8s cluster. To use it, save the script as `setup.sh` and run the following command:

```
$ sudo ./setup.sh master
```

### 3.2 Method 1: Wasm Workload Execution using crun Runtime

This method aims to build, deploy, and run a REST API microservice as a Wasm workload on one of the two worker nodes. This process involves using `crun` with WasmEdge support. The theoretical underpinnings of this were detailed in Section 2.4.1.

As discussed in Section 2.2, K8s depend on low-level container runtimes such as `runc` or `crun` to run and manage containers. `crun` can run not only containers but also Wasm workloads. `crun` does this because it natively supports running Wasm workloads using WasmEdge, Wasmer, and Wasmtime [43].

#### 3.2.1 Development Environment Setup for Wasm Compilation

This section outlines the essential software required for building, packaging, and pushing WasmEdge-compatible Wasm workloads to container registries. This enables packaged Wasm workloads to be pulled and executed within a K8s cluster.

Table 1 lists the software and versions needed for the WasmEdge-based Wasm workload development.

Table 1: List of required software and versions for compiling and building WasmEdge-compatible Wasm container images.

<i>Type</i>	<i>Component</i>	<i>Version</i>
Container Image Builder	Buildah	1.32.2
Programming Language	Rust	1.75.0
Wasm runtime	WasmEdge	0.12.1

#### Software Selection and Installation:

- **Buildah:** Buildah is chosen because it is daemon-less and smaller in size than other container image builder tools, such as Docker. For installation, follow the official Buildah GitHub repository instructions [45].
- **Rust:** Rust is chosen for its speed and excellent Wasm integration. Install using the official Rust website’s instructions [46].
- **WasmEdge:** WasmEdge is chosen for its native support by the crun container runtime [47]. Refer to the WasmEdge official website for installation [48].

Please note that this thesis will not detail the step-by-step process for installing the software mentioned above. Each of these tools can be installed directly from their official websites, where comprehensive installation guides are provided.

### 3.2.2 Building, Containerizing, and Publishing Wasm Workload

After setting up the development environment, the next steps involve compiling the Wasm workload, packaging it within a container image, and publishing it to an OCI-compatible image registry.

#### Project structure and source code:

The project folder is illustrated in Figure 13.

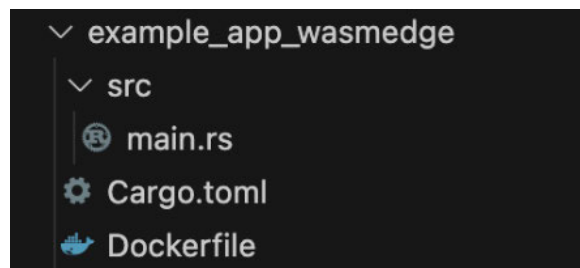


Figure 13: Organizational structure of the source code.

The core components of the Rust application include:

- `main.rs`: This file is the primary entry point of the Rust application.

- `Cargo.toml`: The Rust project manifest file contains configuration details and dependencies.
- `Dockerfile`: The `Dockerfile` provides instructions for compiling and containerizing the application.

Appendix B of this thesis provides the source code for this application.

### Debugging Source Code:

Before containerizing the application, ensure that the project compiles and runs correctly in the local environment. As discussed in Section 2.3, Wasm is designed as a portable compilation target for programming languages, and Wasm support must be added to Rust by executing the following command:

```
$ rustup target add wasm32-wasi
```

After adding Wasm support to Rust, execute the following command to pull the dependencies and build the Rust project for the `wasm32-wasi`<sup>5</sup> target:

```
$ cargo build --target wasm32-wasi --release
```

On successful compilation, the Wasm application will be saved at `./target/wasm32-wasi/release/http_server.wasm`.

To run the Wasm application on the development machine, run the following command:

```
$ wasmedge ./target/wasm32-wasi/release/http_server.wasm
```

### Building a Container Image:

After the successful execution of Wasm workload on the development machine, the next step is to build a container image with Buildah. To build and push the container image, run the following commands:

```
$ buildah bud -t <image_name>:<image_tag> .  
$ buildah push <image_name>:<image_tag>
```

---

<sup>5</sup> `wasm32-wasi` target is compiled for 32-bit Wasm and designed to run outside of web browsers in environments that support the WASI interface.

### 3.2.3 Configuring Worker Node 1

As discussed in Section 2.4, in a standard K8s setup, the Kubelet interacts with a high-level container runtime such as containerd or crun, which manages low-level container runtimes like runc or crun. For the objectives of this method, Node 1 will be configured to use CRI-O as the high-level container runtime, along with crun (with WasmEdge support) as a low-level container runtime and WasmEdge as a Wasm runtime. This section will focus on installing and configuring CRI-O, crun, and WasmEdge, enabling them to execute Wasm workloads within the K8s environment.

The versions specified below were selected to ensure compatibility at the time of writing of this thesis or its subsequent publication:

Table 2: List of key components and version for worker node 1.

<i>Component</i>	<i>Version</i>
WasmEdge	0.12.1
Crun	1.8.4
CRI-O	1.24.6
Kubelet	1.28.2

**Worker Node Setup Automation Script:** To streamline the setup process, a script automating the installation and configuration of the Kubelet, CRI-O, crun with WasmEdge support, and WasmEdge is provided in Appendix A. This script uses the recommended versions but allows specific versions to be set by passing arguments. To execute the script, copy the setup.sh from Appendix A to Node 1 and run the command:

```
$ sudo ./setup.sh method1
```

**Manual Setup:** The following sections detail the installation and configuration of necessary components:

- **Installation of WasmEdge:** The first step involves installing the WasmEdge binary on Node 1. Refer to the WasmEdge official website for installation [48]. This installation ensures that Wasm workloads compatible with WasmEdge can be executed on Node 1.
- **Installation and Configuration of crun with WasmEdge Support:** Once WasmEdge is successfully installed, the next step is to install and configure crun with WasmEdge support. The easiest approach is to build crun binaries with WasmEdge support from the source code and install the crun binary on the worker node 1. Refer to the official WasmEdge documentation [49] for compiling crun with WasmEdge support.

- **Installation and Configuration of CRI-O:** Once crun with WasmEdge support is successfully installed, the next step is to install and configure CRI-O. Install CRI-O by following the official installation instructions found on the CRI-O GitHub repository [50]. By default, CRI-O uses runc runtime as its primary low-level container runtime. It must be configured to use crun with WasmEdge support as the low-level container runtime.

This configuration is achieved by editing the CRI-O configuration file, typically located at `/etc/crio/crio.conf`. If the configuration file doesn't exist, create it, integrate the configuration outlined in Listing 2 into the `[crio.runtime]` section of the file and update the `runtime_path` within the snippet to reflect the actual installation of crun.

Listing 2: This snippet ensures that CRI-O will utilize crun with WasmEdge runtime as its default runtime.

```
# Other CRI-O configuration settings ...

[crio.runtime]
default_runtime = "crun"

[crio.runtime.runtimes.crun]
runtime_path = "/path/to/crun"

# ... Other CRI-O configuration settings
```

After updating the configuration file, restart CRI-O by executing the following command:

```
$ sudo systemctl restart crio
```

- **Installation of Kubelet:** Please refer to the official Kubernetes installation instructions for installing the Kubelet [51].

### Join Worker Node 1 to K8s Master Node

After installing and configuring WasmEdge, crun, CRI-O, and Kubelet on worker node 1, the final step is to join it to the master node.

1. Obtain Join Token: Retrieve the join token and control plane endpoint information that was displayed during the setup of the master node.
2. Initiate Join: On worker node 1, execute the following command, providing the token and control plane endpoint from the last step:

```
$ sudo kubeadm join <control-plane-endpoint:port> --token <token>
--discovery-token-ca-cert-hash sha256:<hash>
```

3. Verify Success: Upon successful execution, the terminal should display output confirming the node has joined the cluster. (See Figure 14 for an example).

```

micheal@worker1:~/thesis$ sudo kubeadm join 10.0.0.182:6443 --token m6ogiu.lhrk2n52ucsvpd4m \
> --discovery-token-ca-cert-hash sha256:5ad611340e28b0a2447293d2583daaabf1d056e8c0ac65fd03a7b548a68bae0c
[preflight] Running pre-flight checks
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -o yaml'
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Starting the kubelet
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...
This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.
Run 'kubectl get nodes' on the control-plane to see this node join the cluster.

```

Figure 14: Successful execution of the `sudo kubeadm join` command, indicating the node has been added to the K8s cluster.

### 3.2.4 Deployment of Wasm Module in K8s

Finally, the WasmEdge runtime compatible Wasm workload is deployed in K8s. As discussed in Section 2.2.3, the deployment and management of the K8s cluster are accomplished through the `kubectl` command-line interface, which was installed on the master node.

To ensure the Wasm workload is scheduled on a node equipped with the WasmEdge runtime, specifically worker node 1, `labels` and `nodeSelector` can be used. Initially, all nodes capable of running a Wasm workload compatible with the WasmEdge runtime are labelled. To label the worker node 1, run the following command, replacing `worker1` with the actual hostname of worker node 1:

```
$ kubectl label nodes worker1 wasmedge=true
```

After the Worker node is labelled, the Wasm workload can be deployed on the node using the manifest from Appendix B. Save the manifest locally `wasmedge-app-manifest.yml`. The image specified in the manifest is maintained by the official WasmEdge community. For deploying the workload built in Section 3.2.2, replace `wasmedge/example-wasi-http:compat-smart` with the image name and tag specified in that section. Finally, apply the manifest to the K8s cluster by executing the following command:

```
$ kubectl apply -f wasmedge-app-manifest.yml
```

After successfully executing the above command, the workload specified in the manifest should be running on the targeted nodes. To verify this, execute the following command:

```
$ kubectl get deployment
```

The output of the above command should resemble Figure 15, verifying that all three replicas of Wasm workload are in a ready state.

```
micheal@master:~$ kubectl get deployment
NAME          READY    UP-TO-DATE    AVAILABLE    AGE
wasmedge-deploy 3/3      3             3            6m29s
```

Figure 15: Output of the `kubectl get deployment` command, confirming successful Wasm workload deployment with three ready replicas.

Assuming the workload built in Section 3.2.2 is deployed using the K8s manifest from Appendix B, it should be accessible on node port 31001. Figure 16 provides confirmation of workload accessibility via a successful `curl` command.

```
~ > curl 10.0.0.177:31001
Hello HAW!! from WasmEdge App%
```

Figure 16: Output of the `curl` command, confirming a successful REST request to a WasmEdge workload deployed within a K8s cluster.

### 3.3 Method 2: Wasm Workload Execution using containerd Runtime

This method aims to build, deploy, and run REST API microservice written in Rust on the second worker node. This method involves using containerd, containerd-shim, and Fermyon Spin. The theoretical underpinnings of this method are detailed in Section 2.4.2.

As discussed in Section 2.4.2, containerd is modular by architecture, and its abilities can be extended with shims. Custom shims can be developed for Wasm runtime, which can be used with containerd to manage the Wasm workload. These custom containerd-shims work as mediators between containerd and Wasm runtime. This method uses Fermyon Spin as Wasm runtime and a compatible containerd-shim from an open-source project named `containerd-wasm-shims`. The members of Microsoft's Deislabs maintain the open-source project and develop multiple containerd-shims, which are compatible with different Wasm runtimes.

#### 3.3.1 Development Environment Setup for Wasm Compilation

This section covers the software needed to build, package, and publish Fermyon Spin compatible Wasm workloads to container registries.

This method primarily utilizes software detailed in the first method's development environment setup, including Buildah and Rust. For their setup, please refer to Section 3.2.1. The only key difference in the use of the Fermyon Spin Framework and Fermyon Spin runtime instead of WasmEdge runtime.

Table 3 lists the software and versions needed for the Fermyon Spin-based Wasm workload development.

Table 3: List of required software and their versions for compiling and building Wasm images compatible with Fermyon Spin.

<i>Type</i>	<i>Component</i>	<i>Version</i>
Container Image Builder	Buildah	V1.32.2
Programming Language	Rust	1.75.0
Wasm runtime	Fermyon Spin	2.0.1

Tool Selections and Installation:

- Fermyon Spin: It is chosen for its focus on simplifying Wasm development for cloud-native environments. Refer to its official website for installation [41].

For installation guidance of other software, please refer to Section 3.2.1.

### 3.3.2 Building, Containerizing, and Publishing Wasm Workload

After setting up the development environment, the next steps involve building the Spin application, packaging it within a container image, and publishing it to a container image registry.

**Project structure and source code:**

The project folder is illustrated in Figure 17.

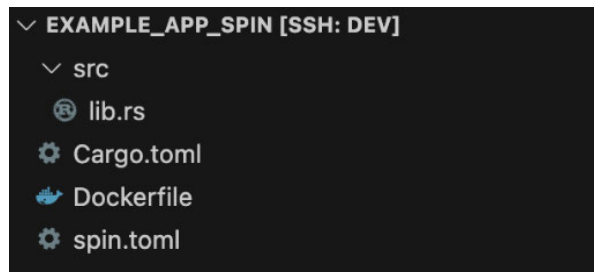


Figure 17: Organizational structure of the source code.

The core components of the Rust application include:

- `lib.rs`: This file contains the main HTTP handling function for Fermyon Spin component.
- `Cargo.toml`: The manifest file contains configuration details and dependencies for the Rust project.
- `spin.toml`: This file provides configuration and metadata for Fermyon Spin application.
- `Dockerfile`: This file defines the steps to build and containerize the application.

The source code for this application is provided in Appendix C.



### Debugging Source Code:

This step covers the process of compiling and running Wasm workload in the local environment to ensure that it compiles and runs correctly. Building source code into Wasm workload and debugging it is similar to the first approach. The only difference is that Spin runtime instead of WasmEdge is needed to run the Wasm workload. First, the Wasm Support must be added to Rust by executing the following command:

```
$ rustup target add wasm32-wasi
```

After adding Wasm Support to Rust, execute the following command to pull the dependencies and build the project:

```
$ cargo build --target wasm32-wasi --release
```

On successful compilation, the Wasm application is located at `./target/wasm32-wasi/release/http_server.wasm`.

To run the Wasm workload with Fermyon Spin execute the following command:

```
$ spin up
```

### Building a Container Image:

After the successful execution of Wasm workload on the development machine, the next step is to build a container image with Buildah. To build and push the container image, run the following commands:

```
$ buildah bud -t <image_name>:<image_tag> .  
$ buildah push <image_name>:<image_tag>
```

### 3.3.3 Configuring Worker Node 2

For the second method, Fermyon Spin is chosen as Wasm runtime and containerd as the high-level container runtime. The versions specified below were selected to ensure compatibility at the time of the writing of this thesis or its subsequent publication:

Table 4: List of key components and version for worker node 2.

<i>Component</i>	<i>Version</i>
Fermyon Spin	2.0.1
containerd-shim	0.10.0
containerd	1.7.12
Kubelet	1.28.2

**Worker Node Setup Automation Script:** To ease the setup process, a script automating the installation and configuration of the Kubelet, containerd, containerd-shim, Fermyon Spin is provided in Appendix A. To execute the script, copy the `setup.sh` from Appendix A to node 2 and run the following command:

```
$ sudo ./setup.sh method2
```

**Manual Setup:** The following sections detail the installation and configuration of necessary components:

- **Installation of Fermyon Spin:** The first step involves the installation of Fermyon Spin binary on the K8s worker node to execute the Wasm workload. Refer to its official website for installation [41].
- **Installation of containerd-shim:** The next step is to install the containerd-shim binaries, which work as a bridge or mediator between the Wasm runtime and containerd. This can be done by downloading the binaries from the GitHub repository and moving them to the `/bin` directory. For binaries and up-to-date documentation, check the official GitHub repository of the containerd-shim project [44].
- **Installation and Configuration of containerd:** Once the containerd-shim compatible with the Fermyon Spin is installed on the system, the next step is to install containerd and configure it to use Fermyon Spin. The easiest way to install containerd on the second worker node is to follow the installation steps of the containerd official documentation [52].

After installing containerd, the last step is to configure containerd. As discussed in Section 2.4.2, a handler for Fermyon Spin is added to the containerd configuration located at `/etc/containerd/config.toml`. In the configuration file, under `[plugins]` section, add a new runtime entry for Fermyon Spin.

This entry should specify the `runtime_type` as `io.containerd.grpc.v1.cri` followed by the name `spin` of handler.

Listing 3: Configuration snippet for integrating Fermyon Spin runtime with containerd by specifying Fermyon Spin as a runtime with the type `io.containerd.spin.v2`.

```
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.spin]
  runtime_type = "io.containerd.spin.v2"
```

- **Installation of Kubelet:** Please refer to the official Kubernetes installation instructions for installing the Kubelet [51].

### Join Worker Node 2 to K8s Master Node

After installing and configuring Fermyon Spin, containerd-shim, containerd and Kubelet on worker node 2, the last step is to join the second worker node to the master node. The process to join the master node is the same as method 1. Refer to Section 3.2.3 for joining the second worker node to K8s cluster.

### 3.3.4 Deployment of Wasm Module in K8s

Finally, a Wasm workload compatible with the Fermyon Spin runtime is deployed to a K8s cluster. As discussed in Section 2.2.3, this is accomplished using the `kubectl` command-line interface.

To ensure the Wasm workload is scheduled on a node equipped with the Fermyon Spin runtime, specifically worker node 2, `labels` and `nodeSelector` can be used. Initially, all nodes capable of running a Wasm workload compatible with the Fermyon Spin runtime are labelled. To label worker node 2, run the following command, replacing `worker2` with the actual hostname of the worker node:

```
$ kubectl label nodes worker2 spin=true
```

After the worker node is labelled, the Wasm workload can be deployed on the node using the manifest from Appendix C. Save the manifest locally as `spin-app-manifest.yml`. The official Fermyon Spin community maintains the image specified in the manifest. For deploying the workload built in Section 3.3.2, replace the value of the image in the `spin-app-manifest.yml` file with the image and tag used to push the image to the container registry. Finally, apply the manifest to the K8s cluster by executing the following command:

```
$ kubectl apply -f spin-app-manifest.yml
```

After executing the above command, the workload specified in the manifest should be running on the targeted nodes. To verify this, execute the command:

```
$ kubectl get deployment
```

The output of the above command should resemble Figure 18.

```
micheal@master:~$ kubectl get deploy
NAME          READY    UP-TO-DATE    AVAILABLE    AGE
wasm-spin-deploy 3/3      3             3            2m40s
```

Figure 18: Output of the `kubectl get deployment` command, confirming successful Wasm workload deployment with three ready replicas.

Assuming the workload built in Section 3.3.2 is deployed using the manifest from Appendix C, it should be accessible on node port 31002. Figure 19 provides confirmation of workload accessibility via a successful `curl` command.

```
~ > curl 10.0.0.150:31002
Hello HAW!! from Fermyon Spin App%
```

Figure 19: Output of the `curl` command, confirming a successful REST request to a Fermyon Spin workload deployed within a K8s cluster.

## 4 Discussion and Conclusion

This thesis explored two primary methods for integrating Wasm into K8s: the crun runtime method and the containerd runtime method. These methods represented unique strategies for deploying Wasm workloads within a K8s cluster.

The first method used WasmEdge as the Wasm runtime with crun to execute Wasm workloads. The second method used Fermyon Spin as the Wasm runtime with containerd, using a containerd-shim, for workload execution.

By utilizing these methods, Wasm workloads were successfully deployed and executed alongside containers within the same K8s cluster.

### 4.1 Challenges Faced in Each Method

#### 4.1.1 Challenges with crun Runtime

The integration of Wasm workloads using crun runtime has several challenges. One of the major challenges is its limited compatibility with specific Wasm runtimes, namely WasmEdge, Wasmer, and Wasmtime. The second challenge encountered with the crun runtime is that crun works only with a single Wasm runtime at a time. Replacing the crun binary with one compatible with the desired Wasm runtime is necessary to switch to a different Wasm runtime.

#### 4.1.2 Challenges with containerd Runtime

Integrating Wasm workloads using containerd, containerd-shim, and Wasm runtime has several challenges. One of the primary challenges is ensuring compatibility between containerd, containerd-shim and Wasm runtime. This involves ensuring the containerd-shim version works seamlessly with containerd and Wasm runtime. The other challenge in using containerd revolves around its capabilities to allow custom containerd-shim. While using custom containerd-shim provides flexibility, it also poses a challenge: the necessity to develop a containerd-shim compatible with a specific Wasm runtime.

### 4.2 Future Research Directions

**Addressing Shim Development Challenges:** Future research could focus on creating more adaptable or universal shims that can work with multiple Wasm runtimes within containerd, reducing the need to develop individual containerd-shim for every Wasm runtime.

**Standardization Efforts:** There's also scope for exploring standardization in containerd-shim development, which could resolve some of the challenges associated with the current need for specific shims for each Wasm runtime.

### 4.3 Concluding Remarks

In conclusion, this thesis offers insights into the integration of Wasm with K8s, contributing to the advancement of server-side application development. Despite the challenges, the successful implementation of both the `crun` and `containerd` methods demonstrates the potential benefits of this integration. This research lays the base for further exploration of the Wasm-based deployment methods in K8s to design more efficient, scalable, and secure computing solutions.

# Bibliography

- [1] “Made with WebAssembly,” [Online]. Available: <https://madewithwebassembly.com/>. [Accessed 26 October 2023].
- [2] “Kubernetes Documentation,” [Online]. Available: <https://kubernetes.io/docs/concepts/overview/>. [Accessed 26 October 2023].
- [3] “9 INSIGHTS ON REAL-WORLD CONTAINER USE,” DataDog, November 2022. [Online]. Available: <https://www.datadoghq.com/container-report/>. [Accessed 26 October 2023].
- [4] S. Hykes, “twitter.com,” 10 November 2023. [Online]. Available: <https://twitter.com/solomonstre/status/1111004913222324225>. [Accessed 12 February 2024].
- [5] J. Beswick, “AWS: Operating Lambda: Performance optimization – Part 1,” Amazon, 26 April 2021. [Online]. Available: <https://aws.amazon.com/blogs/compute/operating-lambda-performance-optimization-part-1/>. [Accessed 22 February 2024].
- [6] R. Matei, “Spin 1.0 — The Developer Tool for Serverless WebAssembly,” 22 March 2023. [Online]. Available: Spin 1.0 — The Developer Tool for Serverless WebAssembly. [Accessed 24 February 2024].
- [7] “Docker: What is a Container?,” [Online]. Available: <https://www.docker.com/resources/what-container/>. [Accessed 1 October 2023].
- [8] “OpenShift: Understanding containers,” [Online]. Available: <https://docs.openshift.com/container-platform/4.12/nodes/containers/nodes-containers-using.html>. [Accessed 30 November 2023].
- [9] “Container Runtimes,” [Online]. Available: <https://kubernetes.io/docs/setup/production-environment/container-runtimes/>. [Accessed 6 May 2023].
- [10] D. Zuev, A. Kropachev and U. Aleksey, “Container Runtime and Container Runtime Interface,” in *Learn OpenShift*, Packt Publishing, 2018, pp. 80-87.
- [11] “Container Runtime Interface (CRI),” [Online]. Available: <https://kubernetes.io/docs/concepts/architecture/cri/>. [Accessed 5 October 2023].

- [12] “Crun: A fast and low-memory footprint OCI Container Runtime fully written in C.,” [Online]. Available: <https://github.com/containers/crun>. [Accessed 3 May 2023].
- [13] “<https://cri-o.io/>,” [Online]. Available: cri-o. [Accessed 10 June 2023].
- [14] S. Grunert, “Demystifying containers – part II: container runtimes,” [Online]. Available: <https://www.cncf.io/blog/2019/07/15/demystifying-containers-part-ii-container-runtimes/>. [Accessed 6 June 2023].
- [15] “containerd-shim/main.go,” 17 June 2016. [Online]. Available: <https://github.com/docker-archive/containerd/blob/a8c73b6959c7b63214bd6ed1e658a165dd61ec0e/containerd-shim/main.go>. [Accessed 13 August 2023].
- [16] “containerd: An open and reliable container runtime,” [Online]. Available: <https://github.com/containerd/containerd>. [Accessed 10 July 2023].
- [17] “containerd Runtime v2,” [Online]. Available: <https://github.com/containerd/containerd/blob/096e99fe7e3febd96df26f743d45d18b8087b6d/runtime/v2/README.md>. [Accessed 12 July 2023].
- [18] “Kubernetes: Kubernetes Components,” [Online]. Available: <https://kubernetes.io/docs/concepts/overview/components/>. [Accessed 29 October 2023].
- [19] “Scaleway: Understanding Kubernetes Autoscaling,” [Online]. Available: <https://www.scaleway.com/en/blog/understanding-kubernetes-autoscaling/>. [Accessed 29 April 2023].
- [20] “Horizontal Pod Autoscaling,” [Online]. Available: <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>. [Accessed 01 May 2023].
- [21] “Kubernetes Cluster Autoscaler,” [Online]. Available: <https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler>. [Accessed 2 May 2023].
- [22] “Pods,” [Online]. Available: <https://kubernetes.io/docs/concepts/workloads/pods/>. [Accessed 03 August 2023].
- [23] “The Kubernetes API,” [Online]. Available: <https://kubernetes.io/docs/concepts/overview/kubernetes-api/>. [Accessed 15 October 2023].
- [24] “The Kubernetes API Server,” oreilly, [Online]. Available: <https://www.oreilly.com/library/view/managing-kubernetes/9781492033905/ch04.html>. [Accessed 15 October 2023].



- [25] “Objects In Kubernetes,” [Online]. Available: <https://kubernetes.io/docs/concepts/overview/working-with-objects/>. [Accessed 14 October 2023].
- [26] “What is etcd?,” IBM, [Online]. Available: <https://www.ibm.com/topics/etcd>. [Accessed 20 October 2023].
- [27] “Kubernetes Scheduler,” [Online]. Available: <https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/>. [Accessed 20 October 2023].
- [28] “kube-controller-manager,” [Online]. Available: <https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/>. [Accessed 04 May 2023].
- [29] “Controller,” [Online]. Available: <https://kubernetes.io/docs/concepts/architecture/controller/>. [Accessed 02 May 2023].
- [30] “kubelet,” [Online]. Available: <https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/>. [Accessed 6 August 2023].
- [31] “WebAssembly Developer Guide,” [Online]. Available: <https://webassembly.org/getting-started/developers-guide/>. [Accessed March 2023].
- [32] “Node.js documentation,” March 2023. [Online]. Available: <https://nodejs.org/api/wasi.html>.
- [33] <https://webassembly.github.io/spec/core/bikeshed/>, “WebAssembly Core Specification,” [Online]. Available: <https://webassembly.github.io/spec/core/bikeshed/>. [Accessed 23 February 2024].
- [34] “WebAssembly.org portability,” [Online]. Available: <https://webassembly.org/docs/portability>. [Accessed 20 March 2023].
- [35] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai and J. Bastien, “Bringing the web up to speed with WebAssembly,” 14 June 2017. [Online]. Available: <https://doi.org/10.1145/3062341.3062363>. [Accessed 10 March 2023].
- [36] P. Hickey, “Lucet Takes WebAssembly Beyond the Browser,” 28 March 2019. [Online]. Available: <https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime>. [Accessed 1 February 2024].
- [37] Surma, “<https://shopify.engineering/javascript-in-webassembly-for-shopify-functions>,” Shopify, 9 Ferbruar 2023. [Online]. Available: Bringing Javascript to WebAssembly for Shopify Functions. [Accessed 20 Ferbruary 2024].

- [38] “Webassembly.org Security,” [Online]. Available: <https://webassembly.org/docs/security/>. [Accessed 25 March 2023].
- [39] D. Gohman , “WASI: WebAssembly System Interface,” [Online]. Available: <https://github.com/bytecodealliance/wasmtime/blob/main/docs/WASI-overview.md>. [Accessed 14 May 2023].
- [40] “WasmEdge Developer Guides,” [Online]. Available: <https://wasmedge.org/docs>. [Accessed 18 August 2023].
- [41] “Fermyon Spin Developer Guide,” [Online]. Available: <https://developer.fermyon.com/spin/index/>. [Accessed 16 May 2023].
- [42] “crun "User Commands",” [Online]. Available: <https://github.com/containers/crun/blob/main/crun.1.md>. [Accessed 23 August 2023].
- [43] “containerd/runwasi Facilitate running Wasm/ WASI workloads managed by containerd,” [Online]. Available: <https://github.com/containerd/runwasi>. [Accessed 29 July 2023].
- [44] “deislabs/containerd-wasm-shims: Containerd shims for running WebAssembly workloads in Kubernetes,” [Online]. Available: <https://github.com/deislabs/containerd-wasm-shims>. [Accessed 30 July 2023].
- [45] “Buildah Installation Guide.,” [Online]. Available: <https://github.com/containers/buildah/blob/main/install.md>. [Accessed 13 February 2024].
- [46] “Rust Installation Guide.,” [Online]. Available: <https://www.rust-lang.org/tools/install>. [Accessed 13 February 2024].
- [47] “Running wasi workload natively on kubernetes using crun,” [Online]. Available: <https://github.com/containers/crun/blob/main/docs/wasm-wasi-on-kubernetes.md>. [Accessed 15 August 2023].
- [48] “WasmEdge Installation Guide.,” [Online]. Available: <https://wasmedge.org/docs/start/install>. [Accessed 13 February 2024].
- [49] “WasmEdge Documentation: Deploy with crun,” [Online]. Available: <https://wasmedge.org/docs/develop/deploy/oci-runtime/crun>. [Accessed 14 February 2024].
- [50] “CRI-O - OCI-based implementation of Kubernetes Container Runtime Interface,” [Online]. Available: <https://github.com/cri-o/cri-o>. [Accessed 14 February 2024].

- [51] “Installing kubeadm,” [Online]. Available: <https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/>. [Accessed 12 December 2023].
- [52] “containerd: Getting started with containerd,” [Online]. Available: <https://github.com/containerd/containerd/blob/main/docs/getting-started.md>. [Accessed 24 February 2024].
- [53] “About the Open Container Initiative,” [Online]. Available: <https://opencontainers.org>. [Accessed 3 May 2023].
- [54] S. Hykes, “Introducing runC: a lightweight universal container runtime,” [Online]. Available: <https://www.docker.com/blog/runc/>. [Accessed 6 August 2023].
- [55] “Redhat: What is a container registry?,” [Online]. Available: <https://www.redhat.com/en/topics/cloud-native-apps/what-is-a-container-registry>. [Accessed 17 July 2023].
- [56] “Rust: The Manifest Format,” [Online]. Available: <https://doc.rust-lang.org/cargo/reference/manifest.html>. [Accessed 17 August 2023].
- [57] D. Uszkay, “How Shopify Uses WebAssembly Outside of the Browser,” 18 December 2020. [Online]. Available: <https://shopify.engineering/shopify-webassembly>. [Accessed 18 January 2024].
- [58] “WASI proposals,” [Online]. Available: <https://github.com/WebAssembly/WASI/blob/main/Proposals.md>. [Accessed 15 May 2023].
- [59] “WebAssembly/wasi-libc: WASI libc implementation for WebAssembly,” [Online]. Available: <https://github.com/WebAssembly/wasi-libc>. [Accessed 26 February 2024].
- [60] M. Yuan, “WasmEdge/wasmedge\_hyper\_demo,” March 2023. [Online]. Available: [https://github.com/WasmEdge/wasmedge\\_hyper\\_demo/blob/cd62f395db79d899e185bf1093fedc3068a843a7/server/src/main.rs](https://github.com/WasmEdge/wasmedge_hyper_demo/blob/cd62f395db79d899e185bf1093fedc3068a843a7/server/src/main.rs). [Accessed 15 September 2023].
- [61] “musec/libpreopen: Library for wrapping libc functions that require ambient authority,” [Online]. Available: <https://github.com/musec/libpreopen>. [Accessed 12 February 2024].
- [62] “Open Container Initiative,” [Online]. Available: <https://opencontainers.org/>. [Accessed 23 November 2023].

# Appendix

## Appendix A

The Bash script in Listing 4 automates the setup of a K8s cluster, including the installation and configuration of container runtimes (containerd, CRI-O, and crun), K8s components, and Wasm runtimes like WasmEdge and Fermyon Spin. It also allows for custom versioning of each component through command-line arguments. The script is developed to configure both master and worker nodes, with specific setups for different types of worker nodes.

To use the script:

1. Save it as `setup.sh`.
2. Make it executable with `chmod +x setup.sh`.
3. Run `./setup.sh master` to configure the master node.
4. Run `./setup.sh method1` to setup the first worker node with CRI-O, crun and WasmEdge runtime.
5. Run `./setup.sh method2` to setup the second worker node with containerd, containerd-shim for Fermyon Spin, and Fermyon Spin runtime.

Listing 4: This Bash script automates the setup of a K8s cluster, including the installation and configuration of container runtimes.

```
#!/bin/bash
set -e

# Ensure the script is run as root
if [ "$(id -u)" -ne 0 ]; then
    echo "This script must be run as root." >&2
    exit 1
fi

# Initialize default versions
OS="${VARIABLE:-xUbuntu_20.04}"
CONTAINERD_VERSION="${VARIABLE:-1.7.12}"
CRIO_VERSION="${VARIABLE:-1.24}"
CRUN_VERSION="${VARIABLE:-1.8.4}"
FERMYON_SPIN_VERSION="${VARIABLE:-2.0.1}"
CONTAINERD_SPIN_SHIM_VERSION="${VARIABLE:-0.10.0}"
KUBERNETES_VERSION="${VARIABLE:-1.28.2-00}"
RUNC_VERSION="${VARIABLE:-1.1.11}"
WASMEDGE_VERSION="${VARIABLE:-0.12.1}"
```

```

# Function for error handling with improved logging and cleanup
error_exit() {
    echo "[$(date +%Y-%m-%dT%H:%M:%S%z)]: Error: $1" >&2
    show_help
    exit 1
}

# Function to avoid redundant downloads
download_file() {
    local url=$1
    local dest=$2

    if [ ! -f "$dest" ]; then
        wget -q "$url" -O "$dest" || error_exit "Failed to download $url"
    else
        echo "File $dest already exists, skipping download."
    fi
}

# Function to add repository and download key.
add_repository_and_key() {
    local add=$1
    local dest=$2
    local key_url=$3
    local key="Release.key"

    echo "$add" >"$dest" || error_exit "Failed to add repository $dest"
    download_file "$key_url" "$key"
    apt-key add "$key" || error_exit "Failed to add key"
    rm -f "$key"
}

# Function to parse arguments and override default versions
parse_args() {
    for opt in "$@"; do
        case $opt in
            --containerd=*) CONTAINERD_VERSION="${opt#*=}" ;;
            --containerd-shim=*) CONTAINERD_SPIN_SHIM_VERSION="${opt#*=}" ;;
            --crun=*) CRUN_VERSION="${opt#*=}" ;;
            --crio=*) CRIO_VERSION="${opt#*=}" ;;
            --kubernetes=*) KUBERNETES_VERSION="${opt#*=}" ;;
            --os=*) OS="${opt#*=}" ;;
            --runc=*) RUNC_VERSION="${opt#*=}" ;;
            --wasmedge=*) WASMEDGE_VERSION="${opt#*=}" ;;
            --spin=*) FERMYON_SPIN_VERSION="${opt#*=}" ;;
            -h | --help) show_help ;;
        esac
    done
}

```

```

        *) ;; # Ignore unrecognized options

    esac

done

}

# Function to show help
show_help() {
    cat <<EOF

Usage: sudo $0 [OPTIONS] COMMAND

Options:
    --containerd[=VERSION]          # Set the containerd version (default:
$CONTAINERD_VERSION)
    --containerd-shim[=VERSION]     # Set the Fermion Spin Shim version (default:
$CONTAINERD_SPIN_SHIM_VERSION)
    --crun[=VERSION]                # Set the crun version (default: $CRUN_VERSION)
    --crio[=VERSION]                # Set the cri-o version (default: $CRIO_VERSION)
    --kubernetes[=VERSION]          # Set the Kubernetes version (default:
$KUBERNETES_VERSION)
    --os[=OS]                       # Set the operating system (default: $OS)
    --runc[=VERSION]                # Set the runc version (default: $RUNC_VERSION)
    --wasmedge[=VERSION]            # Set the WasmEdge version (default:
$WASMEDGE_VERSION)

Commands:
    master                          # Run installation for master node
    method1                         # Run installation for method1 on node
    method2                         # Run installation for method2 on node

Examples:
    sudo $0 master                  # Uses default versions for OS, cri-o, crun,
wasmedge, and Kubernetes
    sudo $0 method1                # Uses default versions for OS, cri-o, crun,
wasmedge, and Kubernetes
    sudo $0 method2                # Uses default versions for OS, containerd,
runc, fermyon spin, containerd-shim, and Kubernetes

    sudo $0 --os=$OS --containerd=$CONTAINERD_VERSION --runc=$RUNC_VERSION --
kubernetes=$KUBERNETES_VERSION master # Set specific versions for OS, containerd,
runc, and Kubernetes for master node
    sudo $0 --os=$OS --crio=$CRIO_VERSION --crun=$CRUN_VERSION --
wasmedge=$WASMEDGE_VERSION --kubernetes=$KUBERNETES_VERSION method1 # Set specific
versions for OS, cri-o, crun, wasmedge, and Kubernetes for method1 on node
    sudo $0 --os=$OS --containerd=$CONTAINERD_VERSION --runc=$RUNC_VERSION --
spin=$FERMYON_SPIN_VERSION --containerd-shim=$CONTAINERD_SPIN_SHIM_VERSION --

```

```

kubernetes=$KUBERNETES_VERSION method2 # Set specific versions for OS, containerd,
runc, fermion spin, containerd-shim, and Kubernetes for method2 on node

EOF
    exit 0
}

# Function to wait for apt lock release
wait_for_apt_lock() {
    local max_attempts=150
    local attempt_counter=1

    while fuser /var/lib/dpkg/lock >/dev/null 2>&1 ||
        fuser /var/lib/dpkg/lock-frontent >/dev/null 2>&1 ||
        fuser /var/lib/apt/lists/lock >/dev/null 2>&1; do
        if ((attempt_counter == max_attempts - 1)); then
            error_exit "Maximum attempts reached while waiting for apt-get lock."
        fi

        echo "apt-get is locked. Attempt: ${attempt_counter}. Waiting..."
        sleep 2
        ((attempt_counter++))
    done
}

# Function to install packages with lock handling
install_packages() {
    wait_for_apt_lock
    apt-get install -y "$@" || error_exit "Failed to install packages: $*"
}

# Function to update packages with lock handling
update_packages() {
    wait_for_apt_lock
    apt-get update --fix-missing -y || error_exit "Failed to update packages"
}

# Function to setup sysctl
# Reference: https://kubernetes.io/docs/setup/production-environment/container-runtimes/
setup_sysctl() {
    tee /etc/modules-load.d/k8s.conf <<EOF
overlay
br_netfilter
EOF

```

```

modprobe overlay
modprobe br_netfilter

tee /etc/sysctl.d/k8s.conf <<EOF
net.bridge.bridge-nf-call-iptables = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward = 1
EOF

# Apply sysctl params without reboot
sysctl --system
}

# Function to disable swap
# Reference: https://kubernetes.io/docs/setup/production-environment/container-runtimes/
disable_swap() {
    swapoff -a
    (
        crontab -l 2>/dev/null
        echo "@reboot /sbin/swapoff -a"
    ) | crontab - || true
}

# Function to install WasmEdge
# Reference: https://wasmedge.org/docs/start/install/
install_wasmedge() {
    update_packages

    echo -e "Installing WasmEdge"
    local install_script="install_wasmedge.sh"
    wget -q
    "https://raw.githubusercontent.com/WasmEdge/WasmEdge/master/utils/install.sh" -O
    "$install_script"
    chmod a+x "$install_script"
    if [[ -z "$WASMEDGE_VERSION" ]]; then
        ./"$install_script" --path="/usr/local"
    else
        ./"$install_script" --path="/usr/local" --version="$WASMEDGE_VERSION"
    fi
    rm -f "$install_script"
}

# Function to install FermyonSpin Runtime
# Reference: https://developer.fermyon.com/spin/v2/install
install_fermyon_spin() {

```



```

echo -e "Installing FermyonSpin Runtime"
local install_script="install_spin.sh"
wget "https://developer.fermyon.com/downloads/install.sh" -O "$install_script"
chmod a+x "$install_script"
if [[ -z "$FermyonSpin_VERSION" ]]; then
    ./"$install_script"
else
    ./"$install_script" --version=v"$FermyonSpin_VERSION"
fi
mv ./spin /usr/local/bin/spin
rm -f "$install_script"
}

# Function to install crun with WasmEdge support
# Reference: https://wasmedge.org/docs/develop/deploy/oci-runtime/crun/
install_crun() {
    echo -e "Building and installing crun"
    install_packages make git gcc build-essential pkgconf libtool libsystemd-dev
    libprotobuf-c-dev libcap-dev libseccomp-dev libyajl-dev go-md2man libtool autoconf
    python3 automake
    local tarball="crun-${CRUN_VERSION}.tar.gz"
    wget
    "https://github.com/containers/crun/releases/download/${CRUN_VERSION}/${tarball}"
    tar --no-overwrite-dir -xzf "$tarball"
    mv "crun-${CRUN_VERSION}" crun
    pushd crun || exit
    ./autogen.sh
    if [[ $USE_WASMEDGE == true ]]; then
        ./configure --with-wasmedge
    else
        ./configure
    fi
    make
    make install
    popd
    rm -rf "$tarball"
    rm -rf "crun"
}

# Function to install runc
# Reference: https://github.com/containerd/containerd/blob/main/docs/getting-
started.md
install_runc() {
    echo -e "Installing runc"

```

```

    local
    runc_url="https://github.com/opencontainers/runc/releases/download/v${RUNC_VERSION}/runc.amd64"
    local runc_file="runc.amd64"
    wget "$runc_url" -O "$runc_file"
    install -m 755 "$runc_file" /usr/local/sbin/runc
    rm -f "$runc_file"
}

# Function to install cri-o
# Reference: https://software.opensuse.org/download/package?package=cri-o&project=devel%3Akubic%3Alibcontainers%3Astable%3Acri-o%3A1.19
install_crio() {
    echo -e "Installing cri-o and its dependencies"
    install_packages libseccomp2

    # Add repositories
    add_repository_and_key "deb
https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stable/$OS/
/" "/etc/apt/sources.list.d/devel:kubic:libcontainers:stable.list"
"https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stable/$OS/Release.key"

    add_repository_and_key "deb
https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stable:/cri-o:/$CRIO_VERSION/$OS/
/"
"/etc/apt/sources.list.d/devel:kubic:libcontainers:stable:cri-o:/$CRIO_VERSION.list"
"https://download.opensuse.org/repositories/devel:kubic:libcontainers:stable:cri-o:/$CRIO_VERSION/$OS/Release.key"

    # Update and install cri-o
    update_packages
    install_packages cri-o

    # Write config for cri-o
    # Path to the CRI-O configuration file
    CRIO_CONFIG="/etc/crio/crio.conf"

    # Get the path of crun
    CRUN_PATH=$(which crun)

    # Backup the original configuration file
    cp $CRIO_CONFIG "${CRIO_CONFIG}.bak"

    # Update the default runtime to crun
    sed -i 's/^#* *default_runtime = .*/default_runtime = "crun"/' $CRIO_CONFIG

```

```

# Add crun runtime configuration
sed -i "/^\#[[:space:]]*\[crio.runtime.runtimes.crun\]/c
[crio.runtime.runtimes.crun]\nruntime_path = \"\$CRUN_PATH\" \"\$CRIO_CONFIG\"

# Enable and start the CRI-O service
systemctl enable crio
# Reload or restart the CRI-O service
if systemctl is-active --quiet crio; then
    echo "Reloading CRI-O service"
    systemctl reload crio
else
    echo "CRI-O service is not active. Trying to start it."
    systemctl start crio
fi

# Check if the service is running
if systemctl is-active --quiet crio; then
    echo "CRI-O service is running with crun as the default runtime."
else
    echo "Failed to start CRI-O service. Please check the service status."
fi
systemctl restart crio
}

# Function to install containerd shim
# Reference: https://github.com/deislabs/containerd-wasm-shims
install_containerd_fermyon_spin_shim() {
    echo -e "Installing containerd shim for Fermyon Spin Runtime"
    wget "https://github.com/deislabs/containerd-wasm-shims/releases/download/v${CONTAINERD_SPIN_SHIM_VERSION}/containerd-wasm-shims-v2-spin-linux-x86_64.tar.gz"
    tar -xzf "containerd-wasm-shims-v2-spin-linux-x86_64.tar.gz" -C /bin
    rm "containerd-wasm-shims-v2-spin-linux-x86_64.tar.gz"
}

# Function to install CNI plugins
# Reference: https://github.com/containerd/containerd/blob/main/docs/getting-started.md
install_cni_plugins() {
    echo -e "Installing CNI plugins"
    wget https://github.com/containerdnetworking/plugins/releases/download/v1.4.0/cni-plugins-linux-amd64-v1.4.0.tgz -O cni-plugins.tgz
    mkdir -p /opt/cni/bin
    tar Cxzf /opt/cni/bin cni-plugins.tgz
    rm cni-plugins.tgz
}

```

```

}

# Function to configure CNI plugins
# Reference: https://github.com/containernetworking/cni
configure_cni_plugins() {
    mkdir -p /etc/cni/net.d
    cat >/etc/cni/net.d/10-mynet.conf <<EOF
{
    "cniVersion": "0.2.0",
    "name": "mynet",
    "type": "bridge",
    "bridge": "cni0",
    "isGateway": true,
    "ipMasq": true,
    "ipam": {
        "type": "host-local",
        "subnet": "10.22.0.0/16",
        "routes": [
            { "dst": "0.0.0.0/0" }
        ]
    }
}
EOF
    cat >/etc/cni/net.d/99-loopback.conf <<EOF
{
    "cniVersion": "0.2.0",
    "name": "lo",
    "type": "loopback"
}
EOF
}

# Function to install containerd
# Reference: https://github.com/containerd/containerd/blob/main/docs/getting-started.md
install_containerd() {
    echo -e "Installing containerd"

    # Download containerd tar file
    wget
    "https://github.com/containerd/containerd/releases/download/v${CONTAINERD_VERSION}/containerd-${CONTAINERD_VERSION}-linux-amd64.tar.gz"

    # Extract containerd tar file
    tar -C /usr/local -xf containerd-${CONTAINERD_VERSION}-linux-amd64.tar.gz

```

```

# Download containerd.service unit file
wget
"https://raw.githubusercontent.com/containerd/containerd/main/containerd.service"
-O /etc/systemd/system/containerd.service

# Reload systemd daemon
systemctl daemon-reload

# Enable and start containerd
systemctl enable --now containerd

# Delete the downloaded tar file
rm containerd-${CONTAINERD_VERSION}-linux-amd64.tar.gz
}

# Function to configure containerd and adding shim to the containerd config
# Reference: https://github.com/deislabs/containerd-wasm-shims
configure_containerd() {
    echo -e "Configuring containerd"
    if [ ! -d "/etc/containerd" ]; then
        mkdir /etc/containerd
    fi
    containerd config default >/etc/containerd/config.toml
    sed -i 's/SystemdCgroup = false/SystemdCgroup = true/'
/etc/containerd/config.toml
    # Map runtime type to the shim binary
    sed -i '/\[plugins\]/a\[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.spin\]
runtime_type = "io.containerd.spin.v2" /etc/containerd/config.toml
    systemctl restart containerd
}

# Function to install kubeadm dependencies
install_k8s_components() {

    # Forwarding IPv4 and letting iptables see bridged traffic
    setup_sysctl

    # Disable swap
    disable_swap

    echo -e "Installing kubeadm dependencies"
    update_packages

    # Install Kubeadm dependencies
    install_packages apt-transport-https ca-certificates curl

```

```

# Add the Kubernetes repository
tee /etc/apt/sources.list.d/kubernetes.list <<EOF
deb [signed-by=/usr/share/keyrings/kubernetes-archive-keyring.gpg]
https://apt.kubernetes.io/ kubernetes-xenial main
EOF

# Download the GPG keys for Kubernetes packages
curl -fsSLo /usr/share/keyrings/kubernetes-archive-keyring.gpg
https://dl.k8s.io/apt/doc/apt-key.gpg

# Update the package list and install Kubeadm, Kubelet, and Kubectl
update_packages
install_packages kubelet="$KUBERNETES_VERSION" kubeadm="$KUBERNETES_VERSION"
apt-mark hold kubelet kubeadm
}

# Function to handle master node setup
# Reference: https://kubernetes.io/docs/setup/production-environment/container-
runtimes/
setup_master() {
    echo -e "Setting up ${NODE_TYPE} node"

    # Install runc
    install_runc

    # Install CNI plugins
    install_cni_plugins

    # Configure CNI plugins
    configure_cni_plugins

    # Install containerd
    install_containerd

    # Install Kubernetes and its dependencies
    install_k8s_components

    # Install kubectl
    install_packages kubectl="$KUBERNETES_VERSION"

    # Initialize kubeadm
    local local_ip=$(hostname -I | awk '{print $1}')
    kubeadm init --apiserver-advertise-address=$local_ip

    # Move kube config to user's home directory

```

```

    local user_home=$(getent passwd "$SUDO_USER" | cut -d: -f6)
    mkdir -p $user_home/.kube
    cp -i /etc/kubernetes/admin.conf $user_home/.kube/config
    chown $(id -u $SUDO_USER):$(id -g $SUDO_USER) $user_home/.kube/config

    # Save join token into a variable for later use
    local join_token=$(kubeadm token create --print-join-command)
    echo -e "Join token: $join_token"
}

# Function to handle installation and configuration of method1 on node
setup_method1() {
    echo -e "Setting up ${NODE_TYPE} node with CRI-O, crun, and WasmEdge Runtime"

    # Install WasmEdge
    install_wasmedge

    # Building and installing crun with WasmEdge support
    USE_WASMEDGE=true install_crun

    # Install CNI plugins
    install_cni_plugins

    # Configure CNI plugins
    configure_cni_plugins

    # Install cri-o
    install_crio

    # Install Kubernetes and its dependencies
    install_k8s_components
}

# Function to handle installation and configuration of method2 on node
setup_method2() {
    echo -e "Setting up ${NODE_TYPE} node with Containerd, containerd-shim, and
    FermyonSpin Runtime"

    # Install FermyonSpin Runtime
    install_fermyon_spin

    # Install runc
    install_runc

    # Install containerd shim binary compatible with Fermyon Spin
    install_containerd_fermyon_spin_shim
}

```

```

# Install CNI plugins
install_cni_plugins

# Configure CNI plugins
configure_cni_plugins

# Install containerd
install_containerd

# Configure containerd to use shim
configure_containerd

# Install Kubernetes and its dependencies
install_k8s_components
}

# Main installation function
main() {
    parse_args "$@"
    if [[ "$1" == "master" || "$HOSTNAME" == "master" ]]; then
        setup_master
    elif [[ "$1" == "method1" ]]; then
        setup_method1
    elif [[ "$1" == "method2" ]]; then
        setup_method2
    else
        show_help
        exit 1
    fi
    echo -e "Finished Installation"
}

# Execute main function with all arguments
main "$@"

```



## Appendix B

This appendix provides the Wasm application source code, its file structure, and K8s manifest for the first method discussed in Section 2.4.1 of this thesis.

Listing 5: This `main.rs` file contains the Rust source code and serves as the entry point for a WasmEdge-based HTTP server application.

```
use httpcodec::{HttpVersion, ReasonPhrase, Response, StatusCode};
use std::io::Write;
use wasmedge_wasi_socket::{Shutdown, TcpListener, TcpStream};

fn handle_client_request(mut stream: TcpStream) -> std::io::Result<()> {
    // Prepare a basic HTTP response with additional headers
    let response = Response::new(
        HttpVersion::V1_1,
        StatusCode::new(200).unwrap(),
        ReasonPhrase::new("").unwrap(),
        format!("Hello HAW!! from WasmEdge App"),
    );
    // Write the response
    stream.write_all(response.to_string().as_bytes())?;
    // Ensure all data is sent before shutdown
    stream.flush()?;
    // Shutdown the writing side of the stream
    stream.shutdown(Shutdown::Both)?;

    Ok(())
}

fn main() -> std::io::Result<()> {
    let listener = TcpListener::bind(format!("0.0.0.0:1234"), false)?;
    println!("Http server bound to {}", listener.port.unwrap());
    loop {
        // Handle the client request
        let stream = listener.accept(false)?.0;
        handle_client_request(stream)?;
    }
}
```

Listing 6: This Cargo.toml file configuration defines the http\_server package, setting its name, version, author, Rust edition, and listing its dependencies for Wasm project.

```
[package]
name = "http_server"           # The name of the package. In this case,
the package is named "http_server".
version = "1.0.0"             # The version of the package. This follows
Semantic Versioning (major.minor.patch).
authors = ["Micheal Choudhary"] # A list of authors.
edition = "2018"              # The Rust edition the package is written
in.

[dependencies]                # Dependencies list.
httpcodec = "0.2.3"
wasmedge_wasi_socket = "0.5.3"
```

Listing 7: This Dockerfile file is designed for building a Wasm workload from Rust source code and packaging it within a scratch image.

```
# Use a Rust base image
FROM rust:1.75.0-slim AS builder

# Add the wasm32-wasi target
RUN rustup target add wasm32-wasi

# Set the working directory
WORKDIR /example_app_wasmedge

# Copy the Rust project files
COPY ./src ./src
COPY ./Cargo.toml ./Cargo.toml

# Build the Rust code into WebAssembly using Wasmer
RUN cargo build --target=wasm32-wasi --release

# Create a new stage with a scratch image
FROM scratch AS final

# Copy the compiled WebAssembly file from the builder stage
COPY --from=builder /example_app_wasmedge/target/wasm32-wasi/release/http_server.wasm /http_server.wasm

# Set the entrypoint to the WebAssembly file
ENTRYPOINT ["/http_server.wasm"]
```

Listing 8: This `wasmedge-app-manifest.yml` manifest YAML file defines a K8s Deployment and Service for a Wasm application compatible with WasmEdge.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: wasmedge-deploy
spec:
  replicas: 3
  selector:
    matchLabels:
      app: app-wasmedge
  template:
    metadata:
      labels:
        app: app-wasmedge
      annotations:
        module.wasm.image/variant: compat-smart
    spec:
      nodeSelector:
        wasmedge: "true"
      containers:
        - name: wasmedge-http-server
          image: wasmedge/example-wasi-http:compat-smart
---
apiVersion: v1
kind: Service
metadata:
  name: wasmedge-service
spec:
  type: NodePort
  selector:
    app: app-wasmedge
  ports:
    - protocol: TCP
      nodePort: 31001
      port: 80
      targetPort: 1234
```

## Appendix C

This appendix provides the Wasm application source code, its file structure, and K8s manifest for the second method discussed in Section 2.4.2 of this thesis.

Listing 9: This `lib.rs` file contains the Rust source code defining the core HTTP server logik for a Fermyon Spin-based application.

```
// Import necessary modules from the spin_sdk crate.
use spin_sdk::http::{IntoResponse, Response};
use spin_sdk::http_component;

// An HTTP component in the Fermyon Spin application.
#[http_component]
// The function takes an HTTP request with an empty body (indicated by
`http::Request<()>`) as its argument.
async fn hello_haw(_request: http::Request<()>) -> anyhow::Result<impl
IntoResponse> {
    // Construct an HTTP response with a status code of 200 (OK) and
response body container the "Hello HAW" Text.
    Ok(Response::new(200, "Hello HAW!! from Fermyon Spin App"))
}
```

Listing 10: This `Cargo.toml` file configuration defines the `http_server` package, setting its name, version, author, Rust edition, and listing its dependencies for Wasm project.

```
[package]
name = "http_server" # Name of the package, used as the identifier.
version = "1.0.0" # Semantic versioning of the package.
authors = ["Micheal Choudhary"] # List of authors, useful for credit and
contact purposes.
edition = "2021" # Specifies which Rust edition to use.

[lib]
crate-type = ["cdylib"] # Indicates that this crate is a dynamically linked
library.

[dependencies]
# Below are the external crates (libraries) that this project depends on.

anyhow = "1.0.79" # Provides idiomatic error handling facilities.
http = "0.2.11" # A foundational crate for HTTP handling in Rust, allows
building HTTP-based services.
```

```
spin-sdk = "2.1.0" # A domain-specific library, ensure compatibility and
check for updates regularly.
wit-bindgen-rust = "0.16.0" # Provides WebAssembly interface types bindings
for Rust. Crucial for WASM-based projects.
```

Listing 11: This `spin.toml` file provides configuration and metadata for a Fermyon Spin application.

```
# The version of the Spin manifest file.
spin_manifest_version = 2

[application]
# The authors of the application, useful for metadata and package
registries.
authors = ["Micheal Choudhary <mc@miche.al>"]
# A short description of the application's purpose or functionality.
description = "A simple Spin app for thesis"
# The name of the application, used for identification.
name = "http_server"
# The version of the application, adhering to semantic versioning.
version = "0.0.1"

# An array of triggers. Each trigger defines how an incoming request is
handled.
[[trigger.http]]
# The route on which the trigger activates. In this case, it activates on
the root path.
route = "/"
# The component to invoke when the trigger is activated.
component = "hello"

# Configuration for the "hello" component.
[component.hello]
# The source file of the component, pointing to the compiled WASM binary.
source = "target/wasm32-wasi/release/http_server.wasm"
# A description of what the "hello" component does.
description = "A simple hello HAW component"

# Build instructions for the "hello" component.
[component.hello.build]
# The command to clean the previous build, and then build the project
targeting wasm32-wasi for release.
command = "rm -rf build && cargo clean && cargo build --target wasm32-wasi
--release"
```

Listing 12: This Dockerfile file is designed for building a Wasm workload from Rust source code and packaging it within a scratch image.

```
# Use a Rust base image
FROM rust:1.75.0-slim AS builder

# Add the wasm32-wasi target
RUN rustup target add wasm32-wasi

# Set the working directory
WORKDIR /example_app_spin

# Copy the Rust project files
COPY ./src ./src
COPY ./Cargo.toml ./Cargo.toml
COPY ./spin.toml ./spin.toml

# Build the Rust code into WebAssembly using Wasmer
RUN cargo build --target=wasm32-wasi --release

# Create a new stage with a scratch image
FROM scratch AS final

# Copy the compiled WebAssembly file from the builder stage
COPY --from=builder /example_app_spin/target/wasm32-wasi/release/http_server.wasm /target/wasm32-wasi/release/http_server.wasm
COPY --from=builder /example_app_spin/spin.toml .

# Set the entrypoint to the WebAssembly file
ENTRYPOINT ["/"]
```

Listing 13: This spin-app-manifest.yml manifest YAML file defines a K8s Deployment and Service for a Wasm application compatible with Fermyon Spin.

```
---
apiVersion: node.k8s.io/v1
kind: RuntimeClass
metadata:
  name: wasmtime-spin-v2
handler: spin
scheduling:
  nodeSelector:
    spin: "true"
---
apiVersion: apps/v1
kind: Deployment
```

```

metadata:
  name: wasm-spin-deploy
spec:
  replicas: 3
  selector:
    matchLabels:
      app: app-spin
  template:
    metadata:
      labels:
        app: app-spin
    spec:
      runtimeClassName: wasmtime-spin-v2
      containers:
        - name: spin-http-server
          image: ghcr.io/deislabs/containerd-wasm-shims/examples/spin-
rust-hello:latest
          command: ["/"]
---
apiVersion: v1
kind: Service
metadata:
  name: spin-service
spec:
  type: NodePort
  selector:
    app: app-spin
  ports:
    - protocol: TCP
      nodePort: 31002
      port: 80
      targetPort: 80

```

# Eigenständigkeitserklärung

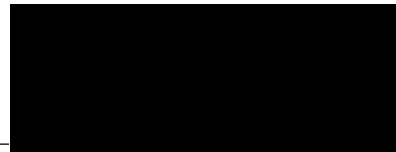
Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit mit dem Titel:

Exploring WebAssembly-Based Microservices Implementation & Deployment Methods in Kubernetes

selbständig und nur mit den angegebenen Hilfsmitteln verfasst habe. Alle Passagen, die ich wörtlich aus der Literatur oder aus anderen Quellen wie z. B. Internetseiten übernommen habe, habe ich deutlich als Zitat mit Angabe der Quelle kenntlich gemacht.

28 Februar 2024

Datum

A solid black rectangular box used to redact the signature of the author.

Unterschrift