Research and Development (Upbring and Optimization) of AERO Cloud Services on the EU Processor v1.0

Deliverable Number:	D.5.1
Due Date:	30.06.2024
Date Of Submission:	19.07.2024
Nature:	OTHER
Dissemination Level:	PU
Work Package:	WP5
Lead Beneficiary	UBI

Grant Agreement No 101092850

Funded by the European Union

Document Control Sheet

Deliverable Title:	**Research and Development (Upbring and Optimization) of AERO Cloud Services on the EU Processor v1.0**
Authors:	**Christos-Alexandros Sarros, Giannis Ledakis, Vasileios Matsoukas (UBI)**
Contributors:	**Polyvios Pratikakis, Anthony Chazapis (FORTH), Christos Katsakioris, Konstantinos Nikas (ICCS)**
Reviewers:	**Konstantinos Nikas (ICCS), Polyvios Pratikakis (FORTH)**
Approved By:	**Christos Kotselidis (UNIMAN), Dionisios Pnevmatikatos (ICCS)**

Document History

Ver.	Date	Status	Description/Comments
0.1	30.05.2024	Draft	Initial version with Table of Contents
0.2	14.06.2024	Draft	Section 2 (Cloud Containerization & Orchestration) added by UBI
0.3	19.06.2024	Draft	Section 4 (Lightweight VMs & Emerging Serverless Frameworks) added by ICCS
0.4	21.06.2024	Draft	Section 3 (Acceleration-aware Cloud Scheduling and Deployment Frameworks) added by FORTH
0.5	25.06.2024	Draft	Section 1 (Introduction) and Section 5 (Summary) added by UBI
0.6	26.06.2024	Draft	Text refinements in all sections (ALL partners)
0.7	27.06.2024	Draft	First full draft, sent for internal review
0.8	01.07.2024	Draft	Document reviewed by ICCS. Refined version based on comments.
0.9	02.07.2024	Draft	Document reviewed by FORH. Refined version based on comments.
1.0	19.07.2024	Final	Final version

DISCLAIMER

AERO has received funding from European Union's Horizon Europe research and innovation programme under Grant Agreement No 101092850. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the granting authority. Neither the European Union nor the granting authority can be held responsible for them.

TABLE OF CONTENTS

Executive Summary

This report provides the documentation regarding the software artifacts developed for the AERO cloud services, during the 1st reporting period of the project (M18). We provide the respective repositories in which the software can be found, along with an overview of the components and instructions on how to build and run the artifacts. The provided software includes the MAESTRO cloud orchestrator, the Knot/Exaflow framework and the various components of the FaaS platform that will be deployed within the AERO project.

List of Abbreviations & Acronyms

Abbreviation/Acronym	Meaning
CNI	Container Network Interface
CRD	Custom Resource Definitions
CRI	Container Runtime Interface
DNS	Domain Name System
FaaS	Function-as-a-Service
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
HTTPS	Hypertext Transfer Protocol Secure
JDK	Java Development Kit
JSON	JavaScript Object Notation
K8s	Kubernetes
OCI	Open Container Initiative
OS	Operating System
VM	Virtual Machine
VMM	Virtual Machine Manager

1 Introduction

Deliverable D5.1 presents the intermediate release of the AERO software artifacts that have been developed and optimsed until M18, under the scope of WP5. The objective of this WP is to bring up and optimse the services selected for cloud management, deployment, monitoring and orchestration in the context of the EU processor ecosystem.

In this document, we describe the work that has been carried out during the reporting period, and present how potential users can reproduce the submitted artifacts for the tested hardware platforms.

This deliverable is structured in three main sections (Sections 2-4). In particular:

- Section 2 “Cloud Containerization and Orchestration” focuses on the MAESTRO cloud orchestration framework, which relies on Kubernetes to deploy containerized applications.
- Section 3 “Acceleration-aware Cloud Scheduling and Deployment Frameworks” focuses on the ExaFlow/Knot frameworks. ExaFlow is used to accelerate complex cloud-based workflows and is based on Knot, a Kubernetes frontend with a focus on facilitating data science activities.
- Section 4 “Lightweight VMs & Emerging Serverless Frameworks”, which focuses on the deployment and optimisation of a Function-as-a-Service (FaaS) platform.

Each main section is also divided into several subsections that:

- Present an overview of the software components
- Document the hardware and software specifications of the tested software
- Provide instructions on how to build the software
- Provide the necessary steps to run the software

.

2 Cloud Containerization and Orchestration

2.1 Overview

With respect to the containerization and orchestration aspects of AERO, we focus heavily on Kubernetes [1](k8s), which acts as the de-facto orchestrator for containerized services. Our final goal is to bring up Kubernetes to the SIPEARL Rhea processor and demonstrate it in the context of the EU processor ecosystem. Kubernetes can use different container runtimes (e.g., containerd, cri-o), as long as the runtime is compatible with the Container Runtime Interface (CRI) API specification[2].

Service provisioning and resource management on top of Kubernetes clusters is performed using UBITECH's Maestro[3] framework. Maestro comprises several components: a GUI, a resource management service, a deployment management service, a k8s-connector, a backend service, an Apache Kafka[4] and a MariaDB[5] database. It should be noted that the Maestro components are typically not deployed in the same servers as the Kubernetes cluster. Instead, they can be deployed anywhere since MAESTRO leverages the Kubernetes API to communicate with clusters and manage the deployments.

Within the AERO project, we are extending MAESTRO's capabilities in the following directions:

- adding the ability to utilize a wider range of devices contained in the EU processor ecosystem including hardware accelerators;
- extending its current application model to include any other additional application and device constraints;
- enabling it to deploy applications that use serverless components (functions) beside microservices, and
- adding the capability to use lightweight VMs to deploy serverless workloads.

In the 1st reporting period, we have successfully managed to:

i) Extend MAESTRO's application model to allow for GPU support and specification of hardware architecture constraints for the deployed cloud services.
ii) Deploy and manage containerized services in ARM and RISC-V devices, using MAESTRO.
iii) Allow for serverless application deployments, by integrating MAESTRO with Knative.

In this deliverable, we document the 1st release of the MAESTRO software that provides the aforementioned features. In this respect, we focus on the extensions that were developed to support serverless applications. The required adaptations and testing of the software component are thoroughly detailed in D6.2.

[1] https://kubernetes.io/
[2] https://kubernetes.io/docs/concepts/architecture/cri/
[3] https://themaestro.ubitech.eu/
[4] https://kafka.apache.org/
[5] https://mariadb.org/

To deploy and manage containerized services in ARM and RISC-V devices, we enhanced MAESTRO's application model the following ways:

When defining a component in MAESTRO, we provide the option to specify the corresponding CPU architecture according to the target deployment device. This enhancement expands beyond the initial support for x86 and amd64 architectures, adding compatibility for ARM64 and RISC-V devices. Consequently, during deployment, the MAESTRO orchestrator is able to fetch the appropriate architecture version of the Docker[6] image.

Additionally, we have introduced a new feature that allows the user to configure service placement prior to deployment using Kubernetes cluster labels. These labels determine the characteristics, such as the architecture of the device, the operating system, the hostname, on which the deployment will occur. The labels are defined in the Kubernetes cluster and retrieved by MAESTRO. This enables the user to specify the target architecture, host, operating system, or any other criteria that differentiate the deployment environment.

We also enhanced MAESTRO so that a user is able to use GPUs for hardware acceleration. Specifically, we allow users to define if they want their component to be GPU-Enabled or not. When enabling GPU option, the Kubernetes deployment specifications are extended appropriately to request GPU resources to run this service.

Moreover, we enhanced MAESTRO's orchestration features for serverless workloads. To achieve this, we rely on Knative[7]. Knative is an open-source platform that extends Kubernetes to manage serverless workloads, providing a powerful framework for building, deploying, and managing modern applications. It simplifies the development of container-based applications by automating many of the complexities associated with scaling, routing, and event-driven processing. The core architecture of Knative comprises two broad components, Serving[8], and Eventing[9] that run over an underlying Kubernetes infrastructure. Knative Serving[10] allows us to deploy containers that can scale automatically as required. It builds on top of Kubernetes and a Network Layer by deploying a set of objects as Custom Resource Definitions[11] (CRDs). Knative Eventing[12] works with custom resources like Source, Broker, Trigger, and Sink. Source is the component that emits events to the Broker. The Broker acts as the hub for the events. These events can then be filtered based on any attribute using a Trigger, and subsequently routed to a Sink[13].

The Knative integration signifies a major upgrade in MAESTRO orchestration capabilities, enabling users to deploy, manage, and scale serverless applications seamlessly. The integration of MAESTRO

[6] https://www.docker.com/
[7] https://knative.dev/
[8] https://knative.dev/docs/serving/
[9] https://knative.dev/docs/eventing/
[10] https://knative.dev/docs/serving/architecture/
[11] https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/
[12] https://knative.dev/docs/eventing/
[13] https://knative.dev/blog/articles/get-started-knative-eventing/

with Knative is accomplished by utilizing the dedicated knative-client[14] library for Java Spring framework.

The addition of Knative support in MAESTRO brings several key advantages. Firstly, it offers event-driven autoscaling, which dynamically adjusts the number of active instances based on real-time demand, thus optimising resource utilization and cost efficiency. Additionally, it enhances the agility and flexibility of application deployment, as users can now deploy services directly within the MAESTRO environment. Finally, this integration provides a unified platform for managing both traditional and serverless applications, enabling a cohesive and streamlined operational workflow. Overall, the integration with Knative empowers our users to harness the full potential of serverless architecture while leveraging MAESTRO's comprehensive orchestration capabilities.

MAESTRO is currently a closed-source project. The implementation of the application model that supports GPU, ARM and RISC-V hardware is part of the closed-source code. However, it is in our plans to open-source it by the end of the project. Moreover, we have released the new Knative Controller service (which provides the integration of MAESTRO with Knative) as an open-source project under an Apache 2.0 license. The project development resources can be found in the AERO Github repository under https://github.com/AERO-Project-EU/maestro-serverless-controller.

2.2 Hardware & Software Specification of Tested Platforms

2.2.1 Hardware Specifications

Table 1. Hardware specifications of tested platform for MAESTRO

Hardware Specifications	
Ampere Altra Mt Jade 2U Server	ARM Neoverse N1 (160 cores), 512GB RAM
4x StarFive VisionFive 2 SBC	RISC-V JH7110 SoC (4+2 cores), 8GB RAM

2.2.2 Software Specifications

To leverage MAESTRO's capabilities for deploying serverless Knative services, we have developed a dedicated Knative Controller. For its development we have used the Quarkus[15] framework, which is being optimised by RHAT for deployment on ARM servers in the context of AERO. The controller itself is a microservice that serves as a REST client to communicate with the MAESTRO deployment controller. Therefore, all the requests for Knative deployments first come through this Quarkus-based controller, which invokes the MAESTRO deployment controller, and then the results are forwarded back to the Quarkus microservice, and reach again the end-user. The Knative Controller is built with Java 17 and Quarkus 3.12.0, featuring a Swagger[16] documentation page where the controller endpoints are documented using the OpenAPI[17] Specification 3.1.0. The project's README.md file includes detailed instructions on how to build and run the controller as a .jar file, a binary, or a containerized image using Docker.

[14] https://mvnrepository.com/artifact/io.fabric8/knative-client
[15] https://quarkus.io/
[16] https://swagger.io/
[17] https://spec.openapis.org/oas/v3.1.0

Table 2. Software specifications of tested platform for MAESTRO

Software Specifications	
JDK	1.8.0_301
Spring Boot	2.0.1
Maven	3.8.6
Docker	26.1.4
Docker compose	2.27
Kubernetes	1.28.0
Knative	1.14.1
OS	Ubuntu 20.04 LTS

Table 3. Software specifications of tested platform for Knative Controller

Software Specifications	
JDK	17
Quarkus	3.12.0
Maven	3.9.6
Docker	26.1.4
Docker compose	2.27
Kubernetes	1.28.0
Knative	1.14.1
OS	Ubuntu 20.04 LTS

2.3 How to build the technology?

In order to build and run MAESTRO, users can follow these steps:

Prerequisites:

- JDK 1.8.0_latest
- Maven 3.x
- Docker 18.03 or higher
- Docker Compose 1.18 or higher

Before moving on, users should verify that they have the required JDK and Maven version using the following shell commands:

```
$ mvn -version
$ java -version
$ javac -version
$ docker --version
$ docker-compose --version (or docker compose version)
```

Clone the repo:

Fetch the repository using: git clone git@gitlab.ubitech.eu:cs3/rnd/aero/aero-maestro.git

Change permissions:

Grafana folder AND subfolders used as volume needs to change its permission from 715 to 717 before spawning the framework:

```
$ sudo mkdir -p /data/maestro/grafana/data
$ sudo mkdir -p /data/maestro/grafana/logs
$ sudo chmod 717 /data/maestro/grafana/data
$ sudo chmod 717 /data/maestro/grafana/logs
```

In order to build the microservices and run the project user must:

1. Move to project home folder:

```
$ cd aero-maestro (project home folder)
```

2. Copy external dependencies in maven folder:

```
$ cp /aero-maestro/ext-deps/maven-libraries/settings.xml pathToMavenFolder/./m2
```

3. Build project microservices:

```
$ mvn clean install
```

4. Build containerized services:

```
$ cd aero-maestro/framework/development/aero
$ docker-compose -f docker-compose-build.yml build
```

5. Create an .env file as in the .env.example and set variables accordingly:

```
$ mv .env.example .env
$ vi .env
```

6. Run containerized services

```
$ docker compose up -d
```

When the docker services are ready, the user can use the frontend UI at http://serverIP:3000, where 'serverIP' is the one you set in the '.env'.

Note that, this docker compose file also includes the knative controller docker service for ease of use.

Steps for building and running the Knative Controller

In order to build and run the Knative Controller, the user can follow these steps:

1. Packaging and running the application:

The application can be packaged using:

```
$ ./mvnw package
```

It produces the 'quarkus-run.jar' file in the 'target/quarkus-app/' directory. Be aware that it's not an **über-jar** as the dependencies are copied into the 'target/quarkus-app/lib/' directory. The application is now runnable using:

```
$ java -jar target/quarkus-app/quarkus-run.jar
```

To build an **über-jar**, execute the following command:

```
$ ./mvnw package -Dquarkus.package.type=uber-jar
```

The application, packaged as an **über-jar**, is now runnable using:

```
$ java -jar target/*-runner.jar
```

The user can create a native executable using:

```
$ ./mvnw package -Pnative
```

Or, if GraalVM is not installed, the user can run the native executable build in a container using:

```
$ ./mvnw package -Pnative -Dquarkus.native.container-build=true
```

The user can then execute the native executable with:

```
$ ./target/knative-serverless-controller-{version}-runner
```

Then user users can directly use the available containerized version:

There are two docker-compose files:

- **docker-compose.yml** which will use the dev profile properties of the project

- **docker-compose.prod.yml** which can be used along with .env file (as the one in the .env.example) to set specifically the parameters of the project.

Then run:

```
$ docker compose up -d
```

2.4 How to run?

In this Section, we provide some information on how a MAESTRO user can leverage the new functionalities implemented in the context of the AERO project. i.e.: i) GPU Hardware acceleration, ii) deployments on ARM and RISC-V hardware and iii) deployment of serverless functions

In order to declare that a service should be run on a GPU, we have to enable the “GPU-Enabled” option for this component through the Components menu and editing the corresponding component.

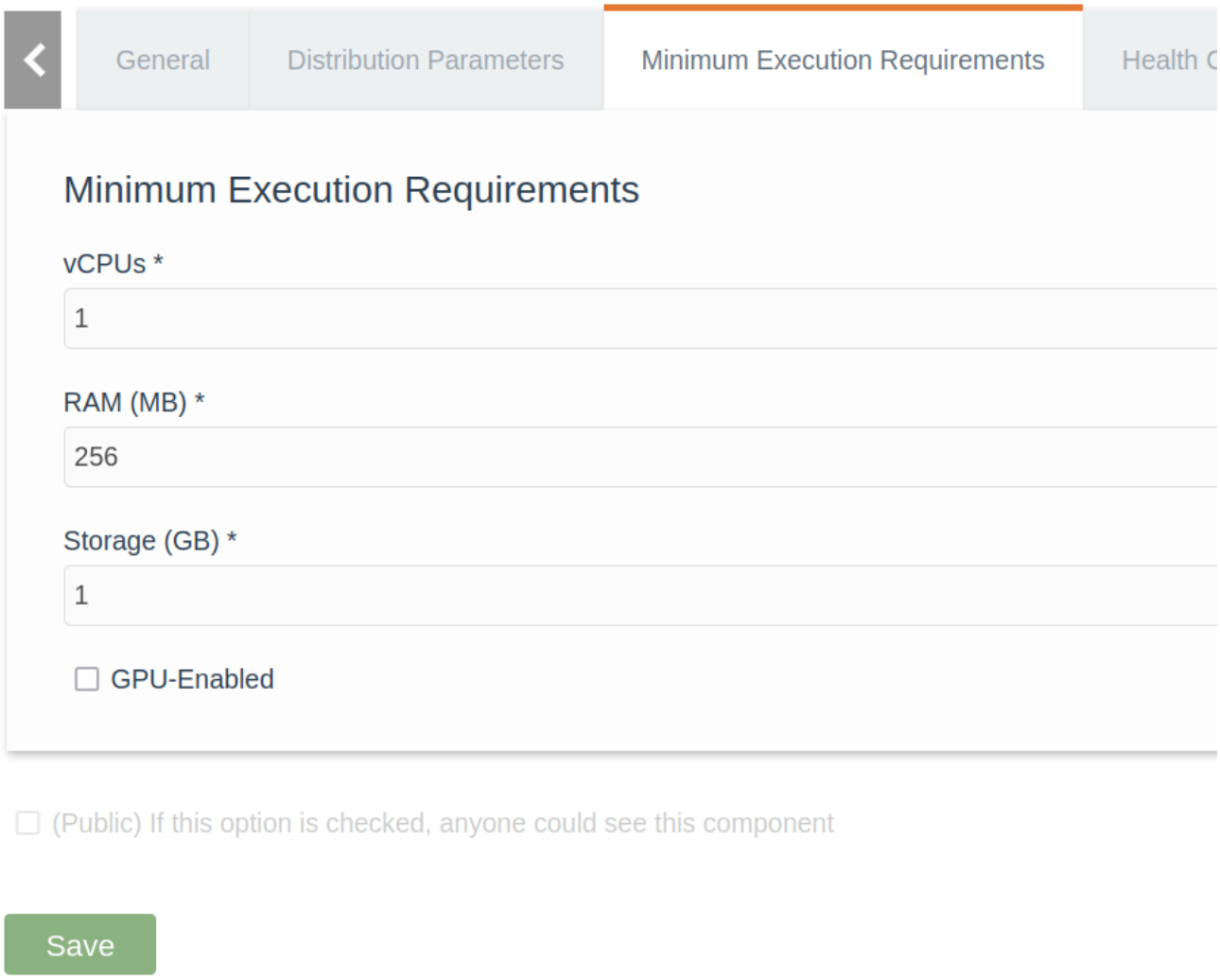

Figure 1. Request GPU placement resources

To select the desired component architecture, you can navigate to the Components menu and select the architecture for the corresponding component:

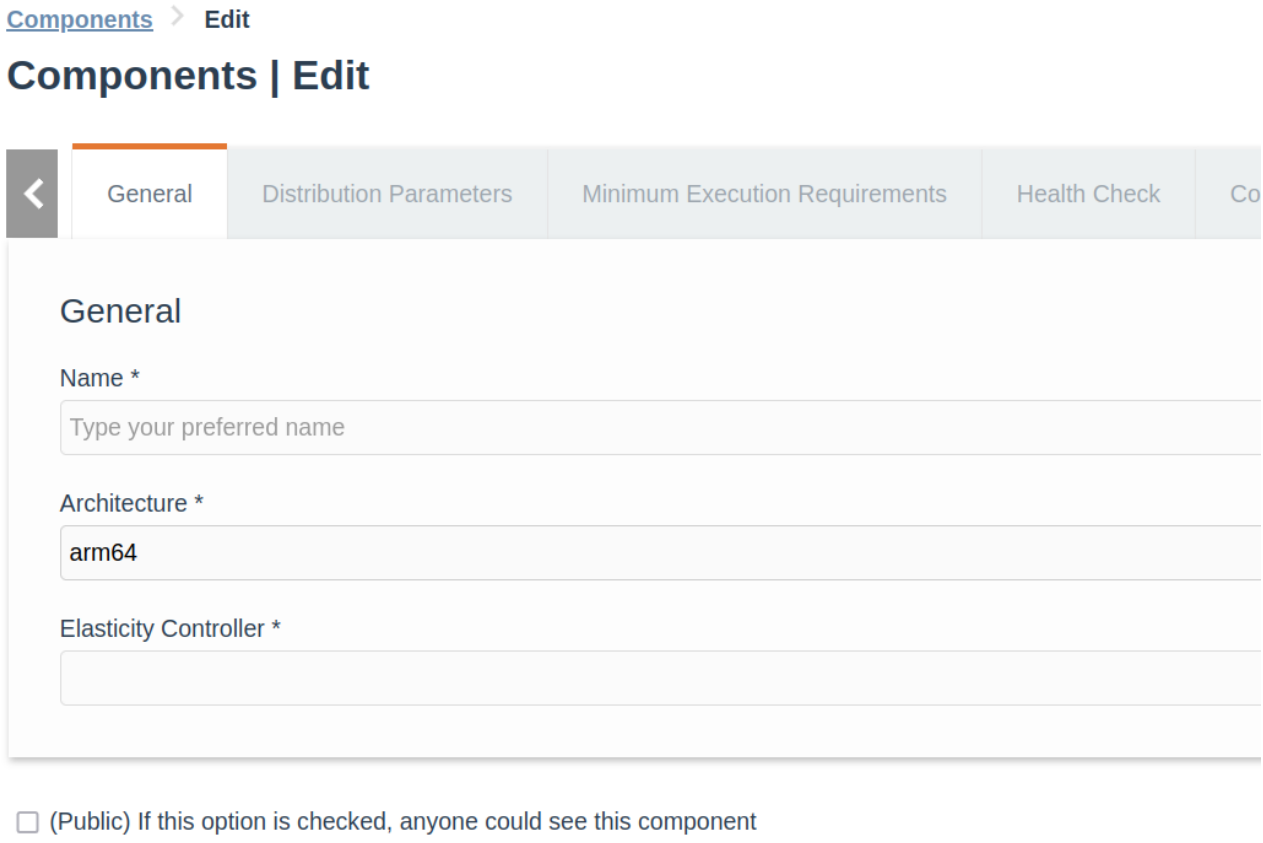

Figure 2. Set component CPU architecture

When configuring the deployment, the user can press on the component symbol and open the configuration menu. From there, the user can navigate to the Node Labels Affinity tab and select the desired node labels for the corresponding component.

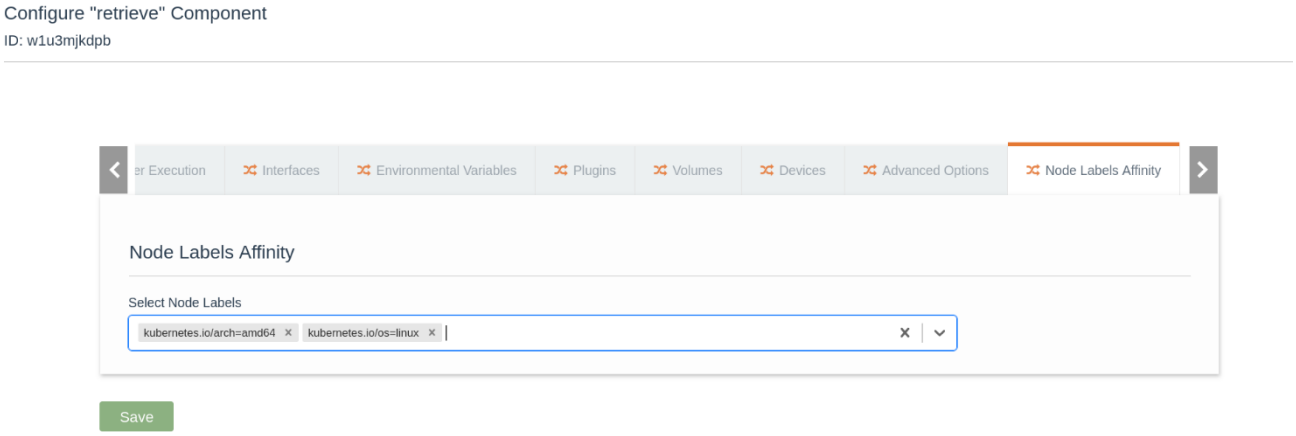

Figure 3. Select Kubernetes node labels for the component deployment

To deploy a serverless application, user can navigate to the MAESTRO-UI, default listening on port 3000. Select an Application to deploy.

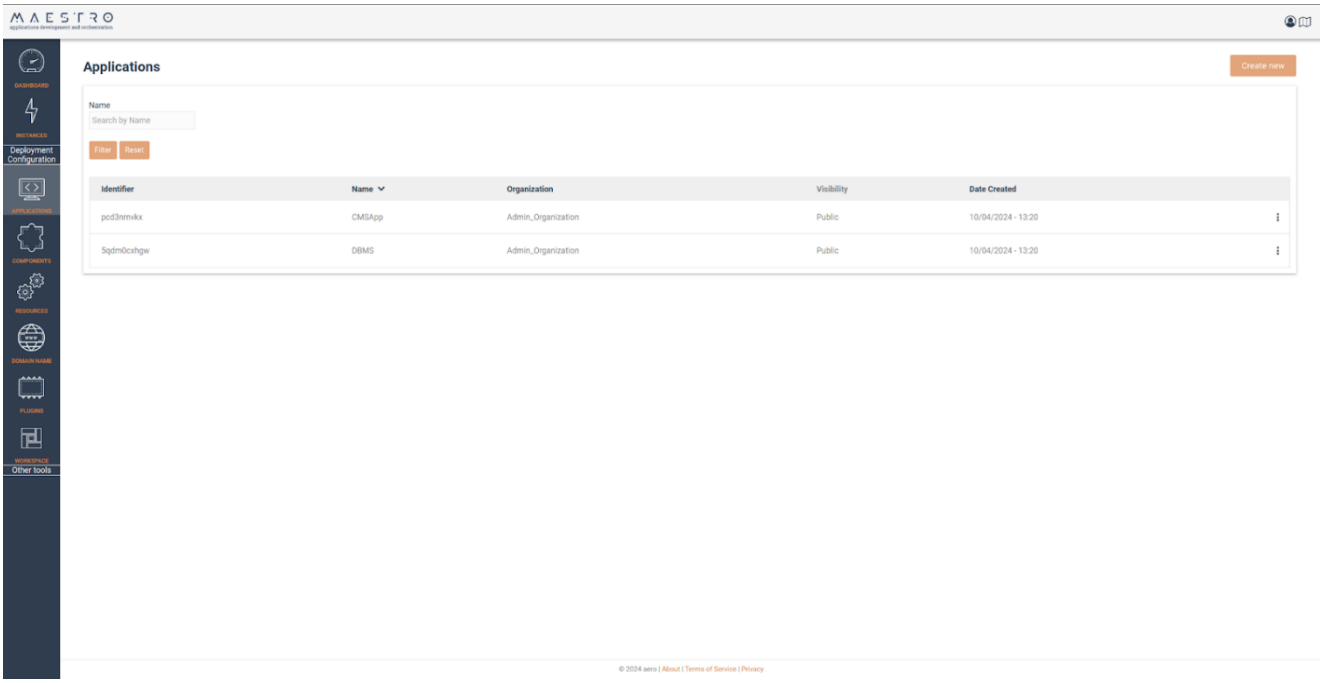

Figure 4. MAESTRO Application view

Then user set the application name and the provider to the Knative Provider and proceed to deployment.

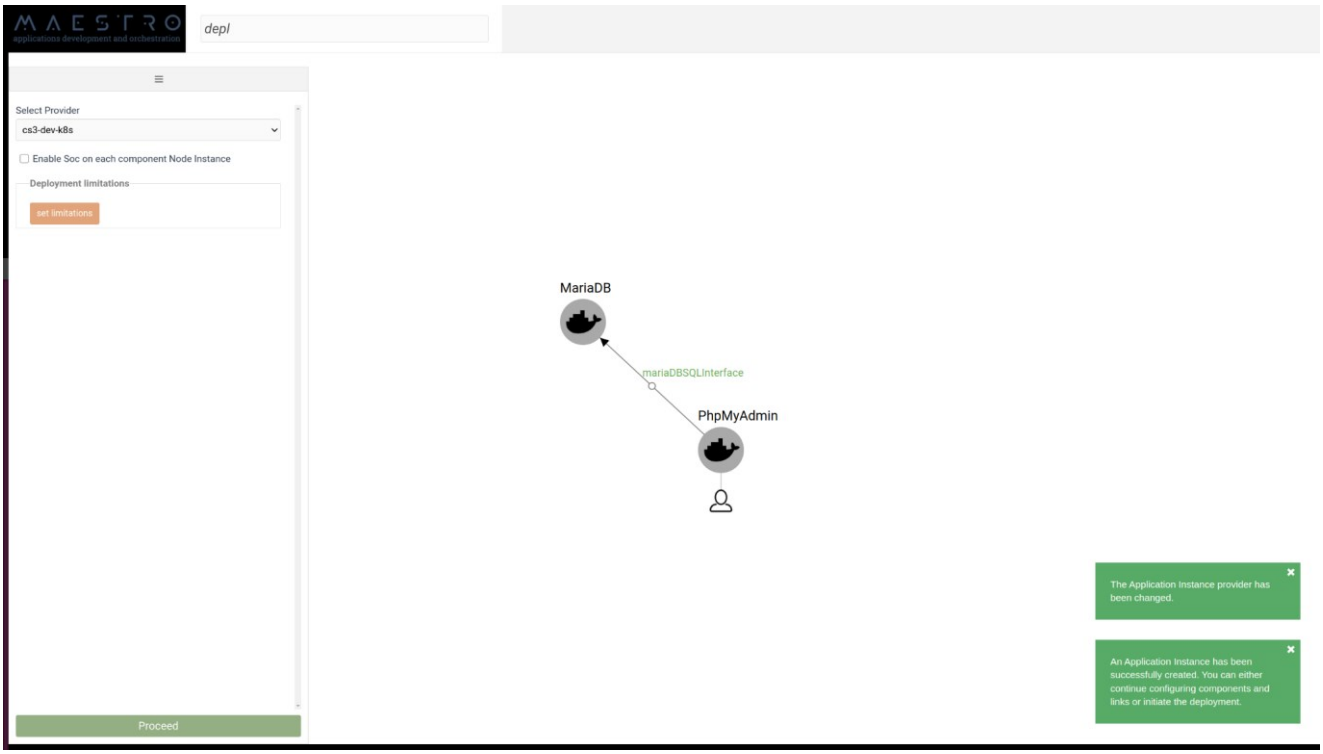

Figure 5. MAESTRO Application Instance configuration before deployment

The procedure has now started and a dedicated namespace has been created in the Kubernetes Cluster. All the resources can be seen under that namespace. When the service deployment is no longer needed, a user can click “Un-deploy” to clear all the resources.

3 Acceleration-aware Cloud Scheduling and Deployment Frameworks (FORTH)

3.1 Overview

This task focuses on exploiting the ExaFlow framework to accelerate complex cloud-based workflows on the EU processor/cloud ecosystem. ExaFlow is based on Knot, a Kubernetes frontend, with a focus on facilitating data science activities. It supplies a web-based landing page for working on a Kubernetes cluster, allowing users to launch services from customizable templates, manage their container images, and launch notebooks for writing portable code. Behind the scenes, Knot manages user accounts, wires up relevant storage to the appropriate paths inside running containers, securely provisions multiple services under a single externally-accessible HTTPS endpoint, while keeping isolated, per-user namespaces at the Kubernetes level, and provides identity services for OAuth 2.0/OIDC-compatible applications. The Knot installation includes JupyterHub[18], Argo Workflows[19], Harbor[20], and Grafana[21], all accessible through the dashboard, taking advantage of the single sign-on feature. ExaFlow extends Knot with support for hardware architectures based on the ARM instruction set (like Rhea), custom services for the AERO project and specialized Kubernetes plug-ins for managing accelerators.

Knot is available at https://github.com/AERO-Project-EU/knot, under an Apache-2.0 open-source license.

3.2 Hardware & Software Specification of Tested Platforms

3.2.1 Hardware Specifications

ExaFlow/Knot has been tested in the following ARM-based hardware.

Table 4. Hardware specifications of tested platorm for ExaFlow/Knot

Hardware Specifications	
Ampere Altra	NeoverseN1 (160 cores), 256 GB RAM
MacBook Pro M1 (2020)	Apple M1 (4+4 cores), 16 GB RAM
QEMU	QEMU 7.2 ARM VM (AArch64) running on MacBook Pro M1 (2020) macOS 13.6.7 (4 CPUs/8 GB RAM)
Raspberry Pi 4	ARM Cortex-A72 (4 cores), 8 GB RAM

We also plan to test on AWS Graviton instances, as well as NVIDIA Grace.

[18] https://jupyter.org/hub

[19] https://argoproj.github.io/workflows/

[20] https://goharbor.io/

[21] https://grafana.com/

3.2.2 Software Specifications

ExaFlow/Knot requires a working Kubernetes installation and optionally CUDA[22] for NVIDIA GPU support.

Table 5. Software specifications of tested platform for ExaFlow/Knot

Software Specifications	
Kubernetes	version >= 1.27.x
CUDA (optional)	version >= 10.2

3.3 How to build the technology?

We provide a comprehensive Makefile to expose all available build actions via simple commands.

To build the Knot container image locally, run:

```
$ make container
```

To test the container in a local Kubernetes environment, run:

```
$ make test-sync
```

Then point a browser to https://<your IP address>.nip.io and login.

To tear down the test environment:

```
$ make test-destroy
```

To build and push the container image, run:

```
$ make container-push
```

To change the version, edit VERSION. Other variables, like the kubectl version and the container registry name are set in the Makefile. For example, the environment variable REGISTRY_NAME points to the Docker namespace that will host the container image (username).

To build and push the container image in the AERO namespace (for example), run:

```
$ REGISTRY_NAME=aero make container-push
```

The Makefile uses buildx to build the Knot container for multiple architectures (linux/amd64 and linux/arm64). Also, a GitHub action automatically builds and releases a new image when a new version tag is pushed. This also triggers publishing the corresponding Knot dashboard Helm chart.

3.4 How to run?

To deploy Knot in a server, users need a typical Kubernetes installation, Helm[23], the Helm diff plugin, and Helmfile installed.

Apply the the latest Knot helmfile.yaml with:

[22] https://developer.nvidia.com/cuda-toolkit

[23] https://helm.sh/

```
$ export KNOT_HOST=example.com
$ helmfile -f git::https://github.com/AERO-Project-EU/knot.git@helmfile.yaml sync
```

The variable KNOT_HOST is necessary. By default, we use cert-manager to self-sign a wildcard certificate for the given host. Users need to make sure that at the DNS level, both the domain name and its wildcard point to their server (i.e., both example.com and *.example.com). If the user already knows their external IP address, they can use a nip.io name (i.e., set KNOT_HOST to <user IP address>.nip.io).

If there already is a certificate, it should be placed in a secret in the ingress-nginx namespace with:

```
$ kubectl create namespace ingress-nginx
$ kubectl create secret tls -n ingress-nginx ssl-certificate --key <key file> --cert <crt file>
```

And then skip the self-signing process at installation by specifying --state-values-set ingress.createSelfsignedCertificate="false" to helmfile, as follows:

```
$ export KNOT_HOST=example.com
$ helmfile -f git::https://github.com/AERO-Project-EU/knot.git@helmfile.yaml --state-values-set ingress.createSelfsignedCertificate="false" sync
```

4 Lightweight VMs & Emerging Serverless Frameworks (ICCS)

4.1 Overview

In the context of AERO, ICCS will deploy and optimise a Function-as-a-Service (FaaS) platform on the upcoming Rhea processor. However, as Rhea is not available yet, this deliverable entails the results of the development on similar processors, namely Ampere Altra Max (ARM Neoverse N1 cores) and NVIDIA's GraceHopper (ARM Neoverse V2 cores).

As defined in deliverable D6.1, the FaaS platform comprises several software components. This deliverable provides contributions on the following levels:

- FaaS software stack: The majority of the software components are leveraged as distributed by their corresponding open-source projects. This deliverable provides the necessary patches and configurations to allow deployment on the selected hardware platforms.
- Serverless workloads: To be able to exercise and demonstrate the FaaS platform, ICCS has ported FunctionBench[24], one of the first publicly available realistic FaaS workload suites.
- Serverless load generator: The serverless ecosystem currently lacks a unified evaluation methodology based on realistic, production-level FaaS workloads. ICCS has developed FaaSRail, a load generator that attempts to fill this gap by combining open source, real-world FaaS workloads with public traces of commercial FaaS platforms to generate representative series of requests suitable for evaluating serverless prototypes.

All these contributions are released as open source (licensed under Apache 2.0.) and are available in the AERO GitHub repository space at the following links:

- FaaS software stack patches: https://github.com/AERO-Project-EU/faas_stack
- Serverless workloads: https://github.com/AERO-Project-EU/aerofb
- Serverless load generator: https://github.com/AERO-Project-EU/faasrail

4.2 Hardware & Software Specification of Tested Platforms

The contributed components have been developed and deployed on two alternative ARM-based platforms, an Ampere Altra server and an NVIDIA GraceHopper system, which have 160 ARMv64 Neoverse N1 cores and 72 ARMv64 Neoverse V2 cores, respectively.

4.2.1 Hardware Specifications

The hardware characteristics of the tested platform are described in Table 6. Hardware specifications of tested platform for the FaaS platform.

[24] https://github.com/ddps-lab/serverless-faas-workbench

Table 6. Hardware specifications of tested platform for the FaaS platform

Hardware Specifications		
System	Ampere Altra Mt Jade	NVIDIA GraceHopper
CPU	2x Ampere Altra (Q80-30), 160x ARMv8.2-A Neoverse N1 Cores	72x ARMv64 Neoverse V2
RAM	4x 64GiB Samsung DDR4-3200	576GB (480GB ECC LPDDR5X, 96GB HBM3)
Disk	Samsung 1TB M.2 PCIe 3.0 x4 NVMe SSD	Intel 1.92TB E1.S PCIe 4.0 x4 NVMe SSD

4.2.2 Software Specifications

Software requirements for this task can be classified as either components of the FaaS stack to be deployed, or utilities that are put into use only for building some of the former.

Note that any dependencies of stack components are extensively documented in their corresponding upstream installation documentation/guides, and hereby omitted. Similarly, Python package dependencies of AERO's FunctionBench port are listed along with their exact revision tag at the organization's code repository in the `requirements.yml` file, while FaaSRail's Rust crates can be found at the associated `Cargo.toml` file.

Table 7. Software specifications for the FaaS platform

Software Specifications	
Deployed FaaS Stack Components	
OS kernel	Linux 5.10.* / 5.15.* / 6.6.*
OCI runtimes	runc: 1.0.0~rc93+ds1 / 1.1.12; against spec 1.0.2-dev
	kata-static 3.3.0
Container Manager / CRI	containerd 1.7.*
Orchestrator	Kubernetes 1.29.* or 1.30.*
CNI	kube-flannel 1.24.*
FaaS Platform	Knative 1.13.* with Kourier 1.13.* or 1.14.*
VMM	Firecracker 1.6.0 (+ICCS's patch for Neoverse V2)
Development-only Utilities	
Local Container Platform	Docker 26.*
Python VM	CPython 3.12.*
Rust	>= 1.79.0
MinIO Server	>= RELEASE.2023-09-20T22-49-55Z

4.3 How to build the technology?

4.3.1 State-of-Practice FaaS Stack

To foster the wider adoption of its bleeding-edge FaaS stack, AERO strives to minimize invasive modifications of the modules in its stack. As a result of this effort, most software components can be deployed as distributed by their corresponding open-source projects, listed in Table 7 and documented in their respective code repositories, without building them from source.

Nevertheless, upbringing the FaaS stack for AArch64 architectures still necessitates certain modifications. In the scope of this deliverable, these modifications entail only a single code patch to

Firecracker VMM's source tree, which can be applied in a single command. The patch file itself can be found in the AERO organization's code repository at https://github.com/AERO-Project-EU/faas_stack/blob/main/firecracker-patches/fc_v1.6.0-aero.patch. To clone the Firecracker git, download ICCS' patch and apply it, the following commands need to be executed:

```
$ git clone --depth 1 --branch v1.6.0 \
>     https://github.com/firecracker-microvm/firecracker.git
$ cd firecracker
$ wget https://raw.githubusercontent.com/AERO-Project-EU/faas_stack/main/firecracker-patches/fc_v1.6.0-aero.patch
$ git apply fc_v1.6.0-aero.patch
```

To subsequently build the patched firecracker binary from the same working directory (i.e., the root of the Firecracker source tree), run:

```
$ tools/devtool build --release
```

The resulting compiled and statically-linked binary can then be found, among others, under `build/cargo_target/aarch64-unknown-linux-musl/release/`, and can be used to replace the original firecracker binary distributed with `kata-static`. The patching procedure is identical in the case of vHive's Firecracker fork, currently used as AERO's research FaaS stack and discussed in Section 4.3.2.

AERO uses the stock kata-static distribution, provided by the open-source Kata Containers project itself. Kata is deployed according to the official documentation, using Firecracker as the VMM ("hypervisor" in terms of Kata Containers' documentation). AERO employs containerd as its FaaS stack's high-level node-local container manager. To provide Firecracker VMs with container root filesystems, containerd has to be configured with a working device-mapper snapshotter, which takes advantage of host kernel's dm-thinpool and dm-snapshot mechanisms to expose these rootfs in the form of block devices. Furthermore, containerd has to be configured with Firecracker-based Kata Containers as one of its available underlying OCI runtimes.

Apart from being the integration point of Kata Containers to the FaaS stack, containerd also acts as the CRI implementation of AERO's orchestrating component (i.e., any Kubernetes distribution, or any Kubernetes API-compatible alternative). In the scope of this deliverable, AERO deploys a standard (vanilla) Kubernetes cluster using kube-flannel as its CNI plugin, according to the official documentation. Moreover, to expose Kata Containers as an OCI runtime option for Kubernetes, we define a new Kubernetes RuntimeClass, which is shown in Figure 6.

Subsequently, AERO deploys Knative on top of Kubernetes, by following the standard installation instructions provided by the upstream project documentation. We use Knative's purpose-built networking layer implementation, Kourier. Furthermore, we configure Knative's control plane to employ sslip.io for setting up Functions' DNS, to make them easily reachable from both inside and outside the Kubernetes cluster. Finally, we set up Knative's associated extension flag to allow specifying the desired underlying Kubernetes RuntimeClass on a per Knative Service basis, as described in the official project's documentation.

```
❯ k describe runtimeclass kata-fc
Name:         kata-fc
Namespace:
Labels:       <none>
Annotations:  <none>
API Version:  node.k8s.io/v1
Handler:      kata-fc
Kind:         RuntimeClass
Metadata:
  Creation Timestamp:  2024-06-14T15:25:17Z
  Resource Version:    1844
  UID:                 efa3e7eb-32c5-4deb-97df-4d9033fe9b7c
Events:                <none>
```

Figure 6. The Kubernetes RuntimeClass that exposes Kata Containers as an OCI runtime option

```
❯ k get -A po -o wide
NAMESPACE         NAME                                      READY   STATUS      RESTARTS       AGE     IP             NODE
knative-serving   activator-579b8bcd94-pt5q6                1/1     Running     3 (16h ago)    3d14h   10.242.0.58    grace1
knative-serving   autoscaler-5cb9794867-p7tcp               1/1     Running     3 (16h ago)    3d14h   10.242.0.56    grace1
knative-serving   controller-78f5d4485f-z5vtp               1/1     Running     3 (16h ago)    3d14h   10.242.0.60    grace1
knative-serving   default-domain-b9kdd                      0/1     Completed   0              3d13h   <none>         grace1
knative-serving   default-domain-hhwcf                      0/1     Error       0              3d13h   <none>         grace1
knative-serving   net-kourier-controller-56c8fcbc4f-5sk5c   1/1     Running     3 (16h ago)    3d13h   10.242.0.62    grace1
knative-serving   webhook-7568cf4f7-86xxw                   1/1     Running     3 (16h ago)    3d14h   10.242.0.63    grace1
kourier-system    3scale-kourier-gateway-6b6749bc9b-2cws7   1/1     Running     3 (16h ago)    3d13h   10.242.0.59    grace1
kube-flannel      kube-flannel-ds-cs9lz                     1/1     Running     22 (41h ago)   4d15h   147.102.4.89   grace1
kube-system       coredns-7db6d8ff4d-8t62g                  1/1     Running     3 (16h ago)    4d15h   10.242.0.57    grace1
kube-system       coredns-7db6d8ff4d-jfsq7                  1/1     Running     3 (16h ago)    4d15h   10.242.0.61    grace1
kube-system       etcd-grace1                               1/1     Running     3 (41h ago)    4d15h   147.102.4.89   grace1
kube-system       kube-apiserver-grace1                     1/1     Running     3 (41h ago)    4d15h   147.102.4.89   grace1
kube-system       kube-controller-manager-grace1            1/1     Running     3 (41h ago)    4d15h   147.102.4.89   grace1
kube-system       kube-proxy-89nnc                          1/1     Running     3 (41h ago)    4d15h   147.102.4.89   grace1
kube-system       kube-scheduler-grace1                     1/1     Running     3 (41h ago)    4d15h   147.102.4.89   grace1
```

Figure 7. Created Kubernetes API Objects and Pods

When everything is configured, several Kubernetes API Objects should be created, and a number of Pods (mostly related to the various control planes in question) should be deployed to the cluster, as depicted in Figure 7. Details about these configurations can be found in the code repository, along with the configuration files themselves.

4.3.2 Research FaaS stack

As discussed in Deliverable D6.1, we intend to use a vHive-based research stack to showcase snapshot-based optimisations. The source code and setup script files distributed by vHive[25] can be used to setup a FaaS stack with Knative on top of Kubernetes that uses not only containerd, but also firecracker-containerd[26], thus enabling the use of Firecracker, a VMM that supports VM snapshotting.

At the time of writing, the provided scripts support the AArch64 architecture only for the stock-only setup, i.e., the setup that uses containerd and not firecrackaer-containerd. Hence, we elaborate below on the steps required to setup vHive with firecracker-containerd on AArch64 systems. More specifically, we setup vHive on the NVIDIA GraceHopper platform listed in Table 6.

Starting with a vanilla Ubuntu 22.04 with Linux kernel v.6.5.0-1022-nvidia-64k, we mainly follow the steps provided in the quickstart guide[27] in the vHive repository.

[25] https://github.com/vhive-serverless/vHive

[26] https://github.com/vhive-serverless/firecracker-containerd

[27] https://github.com/vhive-serverless/vHive/blob/main/docs/quickstart_guide.md

```
$ git clone https://github.com/vhive-serverless/vHive.git
$ cd vHive
```

First, we build the setup tool and run the setup_node command:

```
$ pushd scripts && go build –o setup_tool && popd
$ cd scripts
$ ./setup_tool setup_node firecracker
```

This script performs the basic setup of our node, i.e., installs the required pacakges and binaries, setups basic networking, etc. Unfortunately, the firecracker and firecracker-containerd binaries are only provided for x86_64 systems, and therefore they need to be built from sourse fo AArch64 systems.

```
$ git clone https://github.com/vhive-serverless/firecracker-containerd
$ cd firecracker-containerd
$ make all-in-docker
$ cp ./runtime/containerd-shim-aws-firecracker ./firecracker-
control/cmd/containerd/firecracker-containerd ./firecracker-
control/cmd/containerd/firecracker-ctr /usr/local/bin/
$ git submodule update --init --recursive _submodules/firecracker
$ cd _submodules/firecracker
$ git checkout tags/v1.4.1
```

At this point we need to apply the patch for Firecracker on AArch64 systems, as described in Section 4.3.1. Once this is done, we can proceed with building Firecracker and the rest of the setup.

```
$ cd ../../
$ make firecracker (an error may occur here but we can ignore it)
$ cp _submodules/firecracker/build/cargo_target/aarch64-unknown-linux-
musl/release/firecracker _submodules/firecracker/build/cargo_target/aarch64-unknown-
linux-musl/release/jailer /usr/local/bin/
$ cat << EOF > /etc/containerd/config.toml
version = 2
[plugins]
  [plugins."io.containerd.grpc.v1.cri"]
    [plugins."io.containerd.grpc.v1.cri".cni]
      bin_dir = "/usr/lib/cni"
      conf_dir = "/etc/cni/net.d"
  [plugins."io.containerd.internal.v1.opt"]
    path = "/var/lib/containerd/opt"
  [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc]
    runtime_type = "io.containerd.runc.v2"
    [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc.options]
EOF
```

At this point we need to build the rootfs image and the kernel that will run inside the Firecracker microVM:

```
$ cd firecracker-containerd
$ make image
$ cp tools/image-builder/rootfs.img /var/lib/firecracker-containerd/runtime/default-rootfs-aarch64.img
$ cd _submodules/firecracker
$ cp ../../kernel-configs/microvm-kernel-aarch64-5.10.config .
$ ./tools/devtool -y build_kernel --config ./microvm-kernel-aarch64-5.10.config
$ cd ../../
$ cp _submodules/firecracker/build/kernel/linux-5.10/vmlinux-5.10-aarch64.bin /var/lib/firecracker-containerd/runtime/
```

We then need to create the directory for the CNI K8s plugin:

```
$ mkdir /usr/lib/cni
$ cp /opt/cni/bin/* /usr/lib/cni/
```

Next, we setup and configure containerd's device mapper snapshotter, which is used to create block devices for the microVMs:

```
$ cd vHive/scripts
$ ./setup_tool create_devmapper
```

At this point, we can start containerd, firecracker-containerd and the vHive daemon:

```
$ containerd &> containerd.log &
$ firecracker-containerd –config /etc/firecracker-containerd/config.toml &> firecracker-containerd.log &
$ cd vHive
$ go build
$ ./vhive &> vhive.log &
```

As all daemons are now running successfully, we can create our cluster, e.g., a single node cluster:

```
$ cd vHive/scripts
$ ./setup_tool create_one_node_cluster
```

At this point, as shown in Figure 8, the K8s pods are up and running:

```
$ export KUBECONFIG=/etc/kubernetes/admin.conf
$ kubectl get pods -A
```

```
root@grace1:~# kubectl get pods -A
NAMESPACE          NAME                                      READY   STATUS      RESTARTS   AGE
istio-system       cluster-local-gateway-6f45748884-clsks    1/1     Running     0          47h
istio-system       istio-ingressgateway-7975cbbc47-rxsgv     1/1     Running     0          47h
istio-system       istiod-77d9cd6b46-fw7s6                   1/1     Running     0          47h
knative-eventing   eventing-controller-779445884c-pzp6c      1/1     Running     0          47h
knative-eventing   eventing-webhook-6f499f8479-shd89         1/1     Running     0          47h
knative-eventing   imc-controller-7985996c59-hbtvp           1/1     Running     0          47h
knative-eventing   imc-dispatcher-7cfb975895-hgf8r           1/1     Running     0          47h
knative-eventing   mt-broker-controller-dbd664566-hw7cg      1/1     Running     0          47h
knative-eventing   mt-broker-filter-6bfdb744fc-gkg4d         1/1     Running     0          47h
knative-eventing   mt-broker-ingress-596fcd6964-ftzdz        1/1     Running     0          47h
knative-serving    activator-58db57894b-9lbpj                1/1     Running     0          47h
knative-serving    autoscaler-76f95fff78-stmcl               1/1     Running     0          47h
knative-serving    controller-7dd875844b-6bnm2               1/1     Running     0          47h
knative-serving    default-domain-vdl7p                      0/1     Completed   0          47h
knative-serving    net-istio-controller-57486f879-wx6pz      1/1     Running     0          47h
knative-serving    net-istio-webhook-7ccdbcb557-c4wvt        1/1     Running     0          47h
knative-serving    webhook-d8674645d-whmlr                   1/1     Running     0          47h
kube-system        calico-kube-controllers-787f445f84-snsnr  1/1     Running     0          47h
kube-system        calico-node-gzbvh                         1/1     Running     0          47h
kube-system        coredns-76f75df574-9jnzm                  1/1     Running     0          47h
kube-system        coredns-76f75df574-x4fz6                  1/1     Running     0          47h
kube-system        etcd-grace1                               1/1     Running     0          47h
kube-system        kube-apiserver-grace1                     1/1     Running     0          47h
kube-system        kube-controller-manager-grace1            1/1     Running     0          47h
kube-system        kube-proxy-ls4bf                          1/1     Running     0          47h
kube-system        kube-scheduler-grace1                     1/1     Running     0          47h
metallb-system     controller-5f56cd6f78-lcgx8               1/1     Running     0          47h
metallb-system     speaker-4bslg                             1/1     Running     0          47h
registry           docker-registry-pod-v99x7                 1/1     Running     0          47h
registry           registry-etc-hosts-update-rqrpj           1/1     Running     0          47h
root@grace1:~#
```

Figure 8. Kubernetes pods running for the research FaaS stack

4.3.3 AERO FunctionBench

Functions in the popular FunctionBench FaaS benchmarking suite are originally written in Python. Python is an interpreted language; therefore, the Functions do not need to be built. However, to efficiently distribute them and to use them in a complex FaaS stack, they need to be packaged into an OCI (or Docker) image. To do so, AERO provides a set of Dockerfiles and Makefiles that automate this procedure. To fetch the source code, one may issue:

```
$ git clone https://github.com/AERO-Project-EU/aerofb.git
```

First, since all Function images use a common base, we need to package their base images. To do that, from the root of AERO FunctionBench source tree, we can issue:

```
$ make base-images
```

After creating the OCI images, we can proceed to packaging all AERO FunctionBench Functions, by issuing the following in the same working directory as previously:

```
$ make OWNER=AERO-Project-EU
```

By the time this step finishes, all OCI images have been built, and they are locally stored by Docker daemon. However, this may not suffice; to ease their distribution and make them easily reachable, we can push each created image to an OCI image registry (e.g., Docker Hub, GitHub Container Registry, etc). To do that, we may issue:

```
$ docker push ghcr.io/AERO-Project-EU/aerofb-<BENCHMARK_NAME>:0.0.1
```

Note that most of these arguments are configurable through the top-level Makefile, so that the packaging procedure remains flexible enough for local development.

4.3.4 FaaSRail

While upbringing the serverless ecosystem, AERO identified that a unified evaluation methodology based on realistic, production-level FaaS workloads is currently missing. To this end, ICCS has developed FaaSRail, a load generator that attempts to fill this gap by combining open source, real-world workloads from open-source FaaS benchmarking suites with public traces of commercial FaaS platforms to generate representative series of requests suitable for evaluating serverless prototypes.

To fetch and build FaaSRail one can issue:

```
$ git clone https://github.com/AERO-Project-EU/faasrail.gitmala
$ cd faasrail/faasrail-loadgen
$ cargo build --profile=release
```

4.4 How to run?

4.4.1 AERO FunctionBench on the FaaS stack

The AERO FunctionBench Functions can be deployed, either as standalone containers, or as Knative Services on any Knative-compatible FaaS stack, such the AERO state-of-practice and research stacks.

To be able to deploy FunctionBench Functions' OCI images anywhere, we first have to make them available for distribution through an OCI image registry. AERO has selected to host these images in the GitHub Container Registry, along with the source code repository, built and pushed in a multi-architecture setup via GitHub's CI, GitHub Actions. All AERO FunctionBench hosted OCI images can be viewed at https://github.com/orgs/AERO-Project-EU/packages?repo_name=aerofb.

To locally deploy a FunctionBench function in a standalone fashion, we can use Docker (or even plain containerd, using a command line client like `nerdctl`). For example, to deploy the `video_processing` Function, we can issue:

```
$ docker run --rm -it ghcr.io/aero-project-eu/aerofb-video_processing:0.0.1
```

An example output of this deployment (including the pulling of the OCI image from the GitHub Container Registry repository of AERO) is shown in Figure 9. Note that the command above deploys the Function in a plain container, using runc as its underlying OCI runtime. To deploy it using the kata-fc OCI runtime, the runtime has to be specified as allowed by the command line client in use. For example, Figure 10 shows the equivalent to the above command, but this time using the Kata Containers over Firecracker VMM runtime, along with the respective output, which also proves the successful deployment in a microVM (e.g., notice the difference in Linux kernel versions in the output).

To deploy AERO FunctionsBench Functions on a Knative-based FaaS stack, like AERO's both state-of-practice and research stacks, the Functions have to be represented as Knative Services. AERO distributes such Knative Service definitions for the Functions adopted from FunctionBench, in the form of YAML files that can be found in the associated code repository, at https://github.com/AERO-Project-EU/aerofb/tree/main/tools/knative/svc.

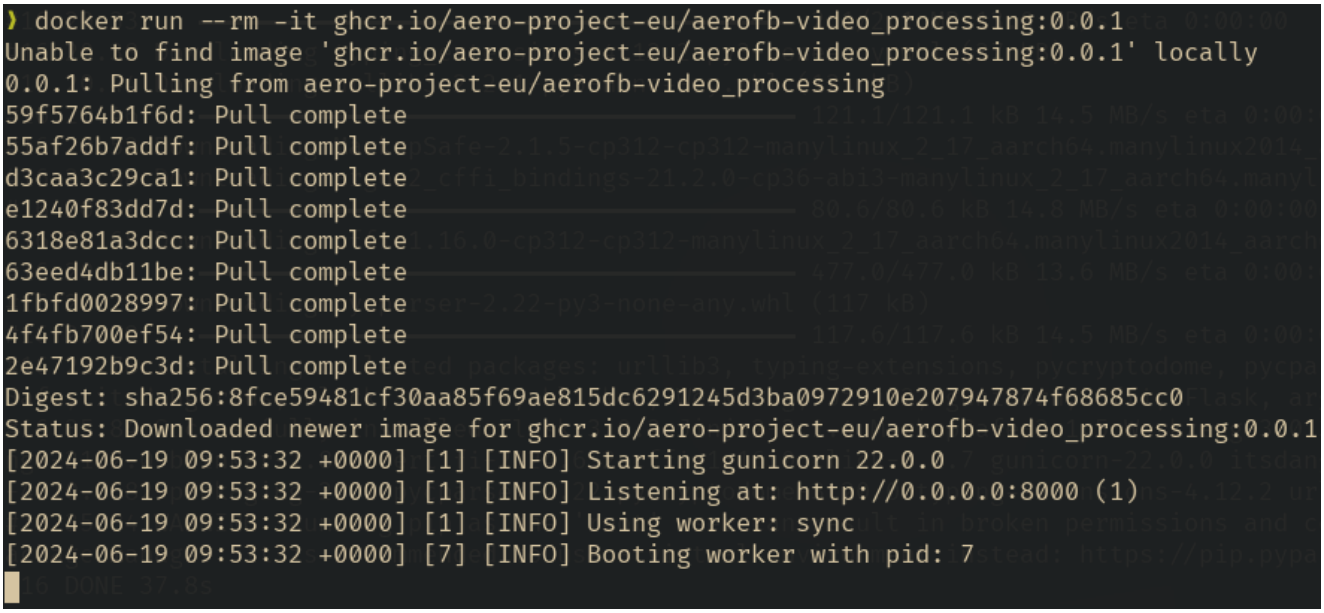

Figure 9: Output of deploying the video_processing Function using runc

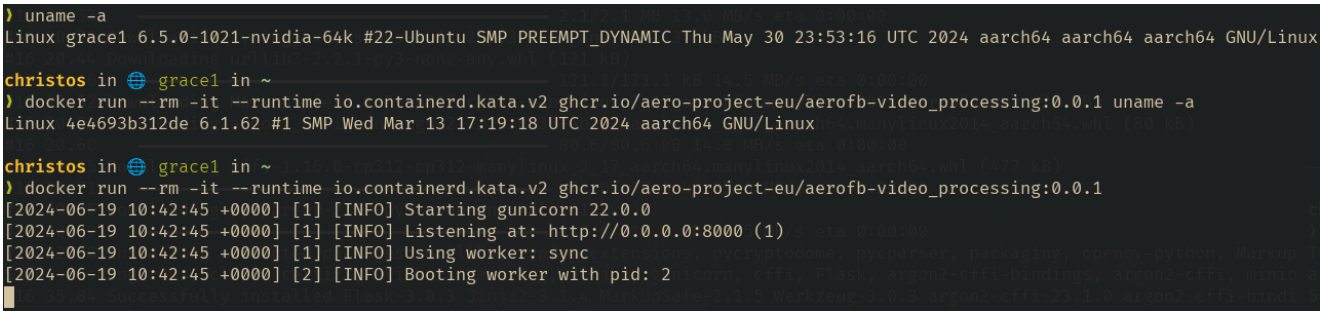

Figure 10: Output of deploying the video_processing Function using the kata-fc OCI runtime

An example of such a file is depicted in Figure 11. We observe that AERO provides two definitions for each Function: one that allows deployment over plain runc (using operating system-level virtualization; i.e., traditional containers), and another that allows deployment over the kata-fc RuntimeClass that was earlier defined in the Kubernetes cluster, after carefully setting up and configuring the Kata Containers project using Firecracker VMM as the hypervisor of choice.

Note that, similar to FunctionBench, some of the AERO FunctionBench Functions attempt to download their input from an AWS S3-compatible object store through a URL provided in their invocation request. For that purpose, AERO employs MinIO, a free and open-source S3-compatible object store:

```
$ minio --version
minio version RELEASE.2023-09-20T22-49-55Z (commit-
id=9788d85ea3a99eeed8073a57a21ccee71035f152)
Runtime: go1.21.1 linux/amd64
License: GNU AGPLv3 <https://www.gnu.org/licenses/agpl-3.0.html>
Copyright: 2015-2023 MinIO, Inc.
```

```
❯ $GOPATH/bin/yq tools/knative/svc/cnn_serving.yml
---
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: aerofb-cnn-serving
spec:
  template:
    spec:
      containers:
        - image: ghcr.io/aero-project-eu/aerofb-cnn_serving:0.0.1
          ports:
            - containerPort: 8000
---
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: aerofb-cnn-serving-fc
spec:
  template:
    spec:
      containers:
        - image: ghcr.io/aero-project-eu/aerofb-cnn_serving:0.0.1
          ports:
            - containerPort: 8000
      runtimeClassName: kata-fc
```

Figure 11. cnn_serving Function YAML file

```
❯ tools/quicktest_client.py -k 147.102.4.89:32563 image_processing
URL: http://aerofb-image-processing-fc.default.147.102.4.89.sslip.io:32563/invoke
INPUT: {"payload": [123, 34, 109, 105, 110, 105, 111, 95, 97, 100, 100, 114, 101, 115, 115, 34, 58, 32, 34, 10
5, 99, 121, 49, 46, 99, 115, 108, 97, 98, 46, 101, 99, 101, 46, 110, 116, 117, 97, 46, 103, 114, 58, 53, 57, 4
8, 48, 48, 34, 44, 32, 34, 98, 117, 99, 107, 101, 116, 95, 110, 97, 109, 101, 34, 58, 32, 34, 115, 110, 97, 11
2, 108, 97, 99, 101, 45, 102, 98, 112, 109, 108, 34, 44, 32, 34, 105, 109, 103, 95, 110, 97, 109, 101, 34, 58,
 32, 34, 105, 109, 103, 50, 51, 48, 107, 46, 106, 112, 101, 103, 34, 44, 32, 34, 111, 112, 115, 34, 58, 32, 34
, 102, 105, 108, 116, 101, 114, 45, 102, 108, 105, 112, 45, 103, 114, 97, 121, 95, 115, 99, 97, 108, 101, 45,
114, 101, 115, 105, 122, 101, 45, 114, 111, 116, 97, 116, 101, 34, 125], "metadata_map": {"header-minio-addres
s": "icy1.cslab.ece.ntua.gr:59000", "header-bucket-name": "snaplace-fbpml", "header-img-name": "img230k.jpeg",
 "header-ops": "filter-flip-gray_scale-resize-rotate"}}

<Response [200]>
{'duration_ns': 230898848, 'output': "{'image_paths': ['/tmp/blur-img230k.jpeg', '/tmp/contour-img230k.jpeg',
'/tmp/sharpen-img230k.jpeg', '/tmp/flip-left-right-img230k.jpeg', '/tmp/flip-top-bottom-img230k.jpeg', '/tmp/g
ray-scale-img230k.jpeg', '/tmp/resized-img230k.jpeg', '/tmp/rotate-90-img230k.jpeg', '/tmp/rotate-180-img230k.
jpeg', '/tmp/rotate-270-img230k.jpeg']}"}
Client-perceived latency: 2983097832 ns
```

Figure 12. Successful invocation of the image_processing Function

Once Functions have been deployed, they are exposed by Knative through a dedicated URL. This is an HTTP/1.1 endpoint served by `gunicorn`, which expects Functions' input data serialized into JSON. To streamline the testing process, we have develop a command line HTTP client for AERO FunctionBench Functions, which is distributed through the associated code repository along with Functions' code, at https://github.com/AERO-Project-EU/aerofb/blob/main/tools/quicktest_client.py. The client reports information about attempted invocation requests (such as the target URL and the serialized input), as well as the target server's response and timing information, as illustrated in Figure 12, which shows the successful invocation of the `image_processing` Function.

Similarly, to deploy the AERO FunctionBench Function on the AERO research FaaS stack a configuration YAML file needs to be created. The following commands create the appropriate configuration for deploying the `video_processing` Function.

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  namespace: default
spec:
  template:
    spec:
      containers:
        - image: ghcr.io/aero-project-eu/aerofb-
video_processing:0.0.1@sha256:d0ce8ae3185c745954a81ac293fce5d5a3c0c9d418cddf5cb8e14bfaa
e72afd8
          ports:
            - name: h2c # For GRPC support
              containerPort: 8000
          env:
            - name: GUEST_ADDR
              value: "127.0.0.1"
            - name: GUEST_PORT
              value: "8000"
            - name: GUEST_IMAGE
              value: "ghcr.io/aero-project-eu/aerofb-
video_processing:0.0.1@sha256:d0ce8ae3185c745954a81ac293fce5d5a3c0c9d418cddf5cb8e14bfaa
e72afd8"
```

Next, to create the Knative service the following command needs to be executed:

```
$ kn service apply video-processing –filename ./video-processing.yaml
```

At the time of writing, a networking problem in vHive limits us to executing only a single function at a time. More specifically, the TUN/TAP devices for the microVMs are not properly set up and when two microVMs execute at the same time, one of them returns an error that the TAP device is in use. We are actively investigating the issue in order to solve it and enable running multiple functions in parallel.

4.4.2 FaaSRail

The first step in using FaaSRail is to run the ShrinkRay component. ShrinkRay receives input related to the configuration of the experiment, produces the experiment specification and formats it as an output CSV file. For instance:

```
$ shrinkray/main.py -w artifacts/icy2-20231011-5.10.189__20231014175133.json \
>         -o azure_spec_rps20_min30.csv trace --trace-dir artifacts/azure-trace \
>         --request-rate 20 --target-duration 30 spec
```

The experiment specification produced by ShrinkRay can then be used as input to FaaSRail's Load Generator. As an example, to run the generator's HTTP-based plugin against a deployed FaaS stack, one can issue the following:

```
$ RUST_LOG='rgv3=debug,reqgen_common=debug,snaplace=info,snaplace_grpc=info' \
>      numactl -N1 -l target/release/rgv3-faascell \
>      --source-address '147.102.4.82:60051' --sink-address '147.102.4.82:60052' \
>      --minio-address 'icy1.cslab.ece.ntua.gr:59000' -o /tmp/tmpfs/sink.json \
>      --inv-log /tmp/tmpfs/inv_log.json --invoc-id 0 \
>      --csv artifacts/azure__rps20__min30.csv --seed 0
```

The output on `stderr` provides details about the invocation requests issued for each Function, along with their timestamp and input information.

5 Summary

This report presented the artifacts developed in the first half of the AERO project, in the context of WP5 “EU Cloud Services”. Due to the delays in the availability of the Rhea platform, some of the planned work has been re-focused on alternative platforms. The software components that are released as part of WP5 are available in the respective AERO repositories, which are provided as part of the report. This document provides an overview of the software components and includes instructions on how to build and run the artifacts.