A Minimal Testbed for Experimenting with Flexible Resource and Application Management in Heterogeneous Edge-Cloud Systems

Alexandros Patras, Foivos Pournaropoulos, Nikolaos Bellas, Christos D. Antonopoulos, Spyros Lalis
University of Thessaly
{patras, spournar, nbellas, cda, lalis}@uth.gr

Maria Goutha, Anastassios Nanos
Nubis
{mgouth, ananos}@nubis-pc.eu

Abstract

The exploration of flexible and intelligent system management in the edge-cloud continuum is a non-trivial task. To start with, a suitable system infrastructure is needed, which is representative of the significant heterogeneity in the computing continuum, especially at the edge. In addition, deployment and orchestration support is necessary to run applications on top of the infrastructure and to exploit acceleration hardware. Furthermore, one needs to be able to monitor application execution and change the current configuration in a flexible way at runtime. Last but not least, it should be possible to plug-in different management policies in a straightforward way. In this paper, we introduce a minimal testbed along with an indicative application, offering several typical features of edge-oriented continuum systems. The testbed comes with a monitoring and control dashboard while providing various hooks and knobs for implementing different policies for automated infrastructure and application management.

Categories and Subject Descriptors

[**Computer systems organization**]: Architectures—*Other architectures, Heterogeneous (hybrid) systems*; [**General and reference**]: Cross-computing tools and techniques—*Experimentation*

Keywords

Heterogeneity, Testbed, Edge-Cloud Continuum, System Management

1 Introduction

Modern computing systems span the continuum, including different and widely heterogeneous nodes at the edge and in datacenters, which may be equipped with hardware accelerators that can be used to perform computations in an energy-efficient way. The deployment and configuration of applications running in such systems is a complex task, not only in terms of the core mechanisms used to execute the application but also in terms of the decisions that need to be taken, often at runtime, in order to respect application performance constraints. It is therefore important to enable testing and evaluating different policies in a straightforward way.

To this end, we introduce a testbed that allows policy developers to flexibly experiment with different policies, without having to deal with the physical setup of nodes or the details of the underlying application execution mechanisms. This is achieved by providing structured interfaces to the core application deployment, node configuration, and telemetry mechanisms. In addition, a graphical dashboard can be used to monitor system operation and to enable manual system configuration and control.

Recent work [20] describes an edge computing testbed for application deployment at the edge, while automatically managing the optimization of the round-trip-time of container interaction. Our work focuses on enabling the configuration and optimization of different aspects of the system via externally provided policies. The Ainur framework [9] is designed as a hardware-agnostic testbed that uses cloud and edge native technologies to automate the configuration of the infrastructure, the deployment of an application workload, and the collection of log data. The main focus of our work is on offering a clearly defined API that an external actor can use to get telemetry data and issue actions at runtime in order to explore different configuration options.

The rest of the paper is structured as follows. Section 2 introduces our testbed. Section 3 discusses the monitoring hooks and various configuration knobs that can be used by system management policies. Section 4 describes an indicative application we have used to validate the testbed, and the respective user interface elements that we used to manually configure the system and observe its operation. Finally, Section 5 concludes the paper.

2 Testbed

2.1 Physical infrastructure

The testbed, illustrated in Figure 1, is designed to provide a minimal but indicative edge-cloud system that can be used to run various test applications. More specifically, it includes 3 heterogeneous physical nodes that represent different layers of the continuum: (i) a Raspberry Pi (RPi) single-board computer, which plays the role of a mid-end edge (and potentially mobile) sensor node with a camera; (ii) a Xilinx ZCU102 development board hosting an ARM MPSoC with an integrated FPGA, which plays the role of a stationary mid-end edge node with acceleration capabilities; (iii) a workstation with an Intel x86 CPU and an NVIDIA GPU, which plays the role of a high-end heterogeneous node in a data center.

In terms of networking, every node is connected to the same LAN using an Ethernet interface. The RPi can also communicate with the Xilinx ZCU102 directly via Wi-Fi. Furthermore, it has a 4G/LTE modem, which can provide mobile Internet connectivity offering yet another path for the RPi to communicate with the Xilinx ZCU102 and the x86 workstation. In order to ensure proper communication of the mobile node with the rest of the cluster nodes, particularly over public 4G/LTE, a VPN gateway is used as a relay.

Part of the testbed are also two virtual machines (VMs) that run on our private cloud cluster, which host (i) the control plane of the application deployment and orchestration mechanism, and (ii) the telemetry storage and dashboard, respectively.

2.2 High-level software organization

Figure 2 shows the conceptual, high-level software organization of the testbed. On the one hand, a deployment and orchestration mechanism is used to install and interconnect the components of a distributed application on the testbed nodes, including support to expose acceleration functionality to the application. On the other hand, a telemetry mechanism is used to monitor the nodes and application execution and to provide status information and performance metrics at runtime. This information can be used by a policy to take management decisions, which are, in turn, applied by invoking the underlying configuration mechanisms.

2.3 Deployment & orchestration

Following the current trend in software packaging and deployment, application components are provided in the form of containers. Notably, for each component, the application developer can provide multiple implementations (container images), which exploit different hardware resources (e.g. CPU only vs. FPGA vs. GPU) and/or offer different performance/accuracy trade-off points.

The testbed currently comes with runtime support for Docker containers [8], while the actual container deployment and orchestration are performed using K3S [5], a lightweight implementation of Kubernetes which is appropriate for resource-constrained devices, such as the RPi. For convenience, besides the components of the application to be tested, the majority of telemetry components also run as containers.

For the flexible and adaptive deployment of the application, we use the Fluidity framework [15]. This is built on top of Kubernetes, providing additional deployment flexibility for distributed applications that employ mobile nodes, while also enabling transparent handling of the application-level data traffic, redirecting it to a potentially more efficient communication channel rather than the default one. The architecture of Fluidity is shown in the lower part of Figure 3. Next, we provide brief descriptions of the main components.

The cluster controller runs in the Management VM and interacts with the Kubernetes control plane via the standard API. It is responsible for invoking Kubernetes to execute the desired deployment of application containers on the nodes of the testbed, as well as for retrieving the status of Kubernetes-related resources. Furthermore, the cluster controller interacts with the node controllers in order to switch among different routing options for application data traffic.

The node controllers, running on each of the testbed nodes, handle the node-level communication interfaces and cooperate with each other to support the components' interactions. Specifically, when instructed by the cluster controller to select a different interface for the interaction between two application components, the node controllers of the respective nodes prepare the link and apply the respective routing rules locally.

Finally, the node controller keeps track of node-level resource utilization and updates custom node-related resources. These updates are retrieved by the cluster controller via the standard Kubernetes API.

2.4 Acceleration

The flexible exploitation of accelerators on the nodes of the testbed is supported using the vAccel [18] framework. Figure 4 provides a high-level overview of the software stack. The core component is the *plugin system*, allowing to attach so-called plugins to pre-defined API operations. For instance, an image classification API operation may have a plugin for a CPU, a GPU, or an FPGA implementation. Multiple plugins may exist for the same operation, providing alternative implementations that can be selected at runtime when an API operation call is issued.

The stock vAccel framework chooses the plugin for a given operation based on a static, framework initialization-time configuration option. In our testbed, we have extended vAccel so that it receive hints from the node controller of the testbed, making it possible to change the plugin selection decision dynamically, at runtime. This entire functionality is wrapped in a frontend library (see Figure 3) making the main application code agnostic both to the vAccel framework and the dynamic plugin selection.

2.5 Telemetry

We have built the telemetry system according to the OpenTelemetry (OTEL) specification [12]. The upper part of Figure 3 shows the telemetry architecture. The main software component is the OTEL collector [13], which can operate in the so-called *agent* or the *gateway* mode.

In every testbed node we run the collector in the agent mode. Different metrics are collected using two open-source tools: (i) the node exporter [10] and (ii) the NVIDIA GPU exporter [3]. The node exporter can query and parse information from every available system source, such as hardware performance counters, using the *perf* tool, and the operating system. The NVIDIA GPU exporter is used to collect statistics from the respective hardware device (if available on the node), relying on the *nvidia-smi* tool to extract the necessary information. The collector agent pulls data from those exporters using the Prometheus receiver plugin.

The collector agent can also gather application-level

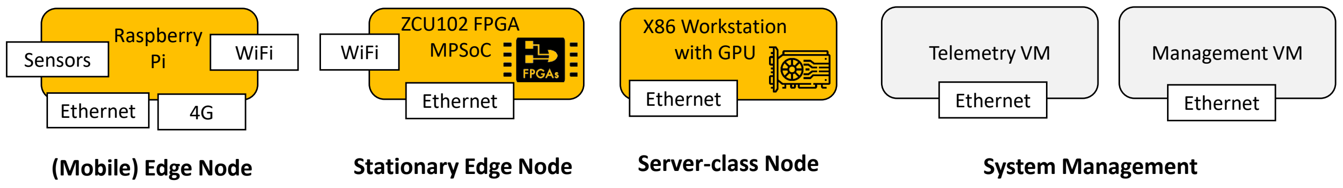

Figure 1: The 3 heterogeneous nodes of the testbed, and the 2 VMs used for telemetry collection and system management.

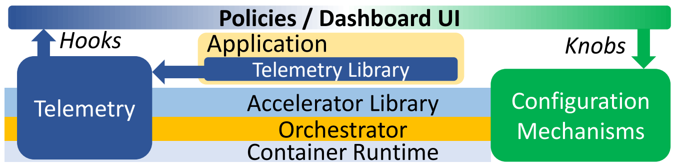

Figure 2: High-level software organization of the testbed.

telemetry. Namely, application components can push arbitrary information to the agent through a simple API, discussed in more detail in Section 3. While the application developer needs to properly "instrument" the component in order to generate the desired metrics, using the API is straightforward and comes with little programming effort. The benefit is that the application can expose its own metrics, so that these can be taken into account by the policy that takes orchestration and configuration decisions.

An agent collector is also placed in the Management VM, to gather cluster-level status information. To this end, suitable receiver plugins are implemented for the interaction with the K3S API Server, while the standard OTEL receiver is used to accept telemetry from the Fluidity controller.

In the Telemetry VM, we run an instance of the OTEL collector in gateway mode, which stores the data received from all collector agents in a Prometheus database [16]. Log messages can also be collected from the agents, and are stored in a Loki database [6]. Finally, Grafana [4] is used to visualize the data stored in both databases.

Notably, collector agents can be configured to process data in batches with low overhead. For instance, certain telemetry data may be produced at a high rate (e.g. every 30ms), whereas the agent can temporarily store the data in memory, and then forward it to the gateway with a lower rate (e.g. every 500ms). The aggregation of telemetry data can be performed in a similar way, to generate and propagate upstream more compact statistics/status digests rather than a voluminous stream of raw data. Our testbed also leverages the internal data pipeline of the OTEL collector, which is composed of three stages: receive, process, and export. The telemetry data stream that enters the collector through different receivers, can be transformed with different methods before being exported to a data sink. In addition, more advanced transformations can be implemented using the OTEL Transformation Language [14]. The telemetry system also emits its own status in the same stream. Finally, we use the routing connector plugin of the OTEL Collector that can create different internal paths for the telemetry data stream, e.g., to export different data to the node-level telemetry interface vs. the data transmitted upstream to the gateway.

Table 1: Application telemetry API.

Method	Arguments	Description
sendMetric	name, value	name & value of the metric
sendLog	name, value	name & contents of log entry

Table 2: Indicative metric attributes.

Attribute	Description
instance	hostname of the node of collector agent
job	hostname of the node of collector gateway
node_name	node related to this reading
value	value of the collected metric

3 Monitoring Hooks & Configuration Knobs

3.1 Telemetry interface

Application telemetry is collected through a simple API, shown in Table 1, which abstracts-out the interaction with the OTEL collector agent running on the node that hosts the application component. The API is simple and generic yet powerful, as the application can send upstream arbitrary performance metrics and/or log messages. However, the container image of the application component must include the corresponding library, which is provided as part of the testbed support toolkit.

The contents of the telemetry database can be retrieved using the Prometheus Query Language (PromQL) [17], a functional query language for querying data in real time and performing different types of analysis, aggregations, and operations. Any container running in the testbed cluster can access the Prometheus database via a REST API provided by a globally accessible endpoint. A node-level telemetry interface is also available, which works as a Prometheus exporter. It can be queried via a simple REST API (without using PromQL) to get raw data in Prometheus metrics format. Note that this API only provides the latest readings from the different local telemetry exporters; it is up to the client to store the data history, if necessary.

By default, a multitude of metrics are collected for each host/node as well as for the individual containers running on it. PromQL can be used to query the collected metrics to extract any piece of information required, e.g., in order to implement a given policy or to display on a dashboard.

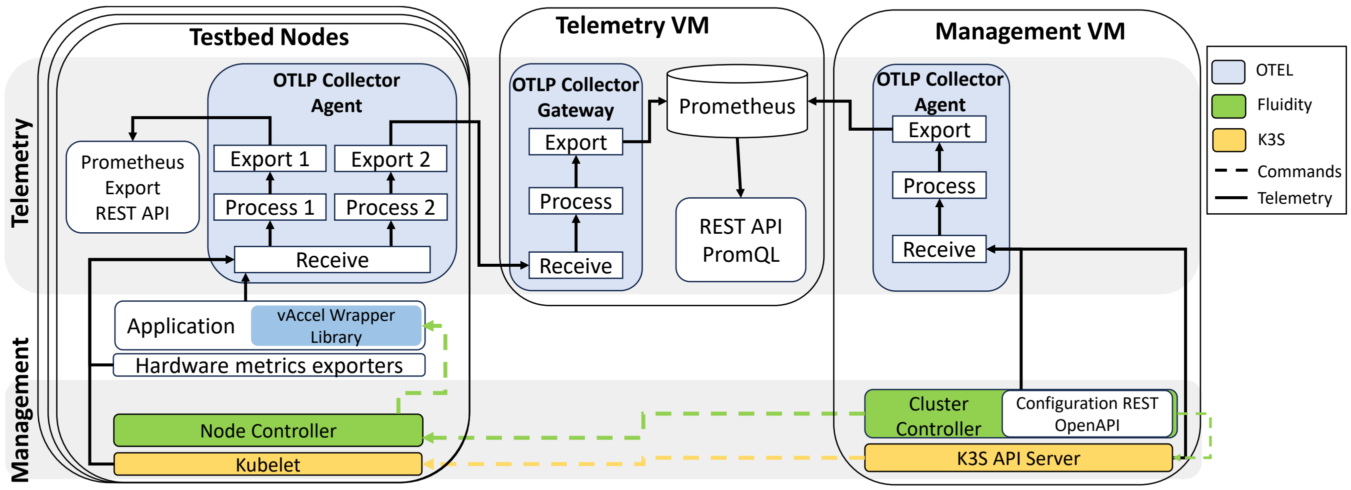

Figure 3: The telemetry and deployment/configuration management architecture of the testbed.

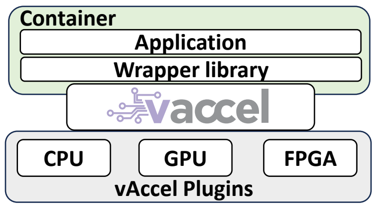

Figure 4: Application running over the vAccel stack.

Table 3: Indicative data samples for node-level metrics.

Metric Name	Raw Data

node_cpu_utilization	instance=otelcol:9464
	job=otel
	node_name=zcu102
	value=50

node_power_consumption	instance=otelcol:9464
	job=otel
	node_name=zcu102
	value=1.4

node_cpu_frequency_profile	instance=otelcol:9464
	job=otel
	node_name=zcu102
	value=4

More specifically, telemetry data carry attributes that can be used to distinguish the origin of the metric (the node or container related with the data). A few indicative attributes are listed in Table 2, while Table 3 provides indicative samples of node-level metrics. There are hundreds of metrics that are exported, related to CPU, Memory, Disk, and Network statistics. Our testbed offers the necessary tools to query, filter and normalize these data, either by configuring the processing within the telemetry pipeline or by querying the telemetry database to get only selected data.

3.2 Configuration options

The main configuration options that are available concern:

1. The placement of the application components on the different nodes of the testbed.
2. The container image to be used for a given application component.
3. The application-level data path used for (individual) component interactions.
4. The use of hardware accelerated function implementations.
5. The operating frequency of the node's CPU / accelerator.

The aforementioned control knobs can be used to exploit different aspects of the heterogeneous nodes based on their unique characteristics and capabilities, leading to different application QoS under different system operation constraints.

For example, an application component can exploit available hardware accelerators, either by deploying a suitable version (container image) that is built to use a GPU/FPGA acceleration module, or by using the vAccel framework to flexibly switch function implementations at runtime without having to change the container image of the component. Moreover, in scenarios where a modular application needs to run one or more components on a mobile node, it is usually desired to replicate or migrate some other application components to the edge and allow them to interact directly with each other over short-range wireless network links, rather than 4G/LTE and the public Internet.

From the system administrator's perspective, some of the supported knobs can be leveraged to reduce the node-level or cluster-level energy consumption and/or cost of operation. More specifically, one may reduce the operating frequency of specific nodes or hardware devices without necessarily harming application QoS, or may place several components on the same node in the spirit of consolidation.

In terms of APIs, the approach is similar to the solution used for the telemetry system. Whenever a configuration decision is made, the corresponding configuration command is

Table 4: Configuration REST API.

Endpoint	Payload	Functionality
/placement	toNode, componentID	Places a component on a node
/image	imageName, componentID	Changes the container image for a component
/network	networkInterface, componentFrom, componentTo	Changes the network interfaces used for a component interaction
/acceleration	plugin, componentID	Change the vAccel plugin used for a component
/frequency	resourceType, node, targetFrequency	Changes the frequency of the hardware resource of a node

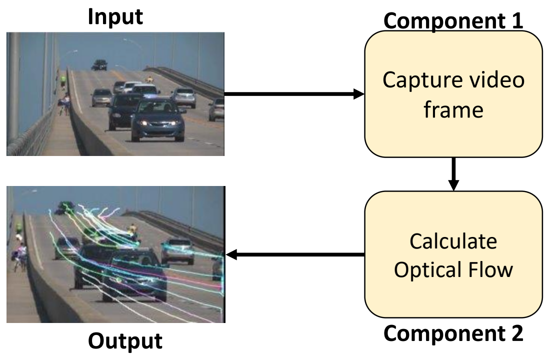

Figure 5: Application Diagram. The snapshot shown here comes from the demo video of OpenCV library [1].

sent to the Fluidity cluster controller via REST to a global endpoint. The Fluidity controller has a dedicated thread running a Flask server [2], which captures and parses the incoming requested action and subsequently executes it (if applicable). The configuration API is described using the OpenAPI specification [11]. Table 4 summarizes the available endpoints and their respective arguments. It is important to note that this API can be used both for manual configuration (e.g., via a control panel) and automated adaptation via a policy (e.g., driven by a suitably trained machine-learning model) designed to achieve autonomic system operation.

4 Application Example

To evaluate the functionality of our testbed, we use an optical flow application, which is characterized by a high degree of resource requirements variability during execution and is modular so it can be split into different components. In a nutshell, the application uses computer vision algorithms to calculate the trajectories of objects in a scene, by recognizing brightness change patterns in the picture.

The application consists of two components, C1 and C2 as seen in Figure 5. C1 takes input from a camera connected to an edge node (in our case, the RPi), pre-processes the frames and sends them to C2. In turn, C2 performs some heavyweight processing on the image to calculate the optical flow. For the implementation of both components, we use the OpenCV library [1], which offers implementations for CPU and GPU (CUDA). For the FPGA implementation of the second component, we use Vitis Vision Library [19] that offers the equivalent hardware implementations for several OpenCV functions. The main test case is to identify an initial features set, and then track those features across frames until they disappear from the field of view of the camera. We use the KITTY Dataset as our main input [7].

We have developed a custom dashboard and control panel to monitor the state of the application deployment and execution, and activate the different configuration options. Figures 6 and Figure 7, depict the dashboard and the control panel respectively.

The dashboard shows the 3 nodes of the testbed, the available computation resources (CPU, GPU, FPGA), the networking interfaces, and the containers (application components) running on the nodes. For each running container the dashboard also shows the image used and the respective application-level metrics. For a node that has no active containers the tool shows which container can potentially run on it, however, the container image is indicated as inactive with a red color. For computational resources we illustrate the current utilization and power consumption and the available frequency configurations. The active configuration is designated with green color. The same applies to the active network configuration and link.

The control panel provides a set of buttons to set the desired configuration options. The buttons for the frequency and the network path will change to the next setting following a round-robin logic. All other buttons perform their assigned action only if it will change the current state. Regarding the placement buttons, if the redeployment is unfeasible e.g. due to resource limitations, the dashboard is suitably updated. This way one can flexibly experiment with different configurations and produce large volumes of telemetry data. In turn, this can be used to design rule-based policies or to train ML models that take configuration decisions.

Note that both the dashboard and the control panel can be tailored to the specific nature of the application at hand, considering its component structure and the monitoring/configuration options that are provided by the person conducting the experiment. In any case, full programmatic access to all telemetry information and all configuration knobs, is provided via the respective APIs. In particular, it is possible to run any configuration policy, using specific rules or an ML model, in tandem with a dashboard to visualize the choices that are taken automatically. At the same time, one can use a control panel to manually introduce dynamic changes in resource availability, which in turn may lead the policy to adapt system configuration.

5 Conclusions

In this paper, we have introduced a prototype testbed designed to represent systems in the heterogeneous edge-cloud continuum. It leverages container technology and the respective tools from the container ecosystem, along with specific enhancements from third-party software, allowing the flexible deployment and transparent usage of heterogeneous

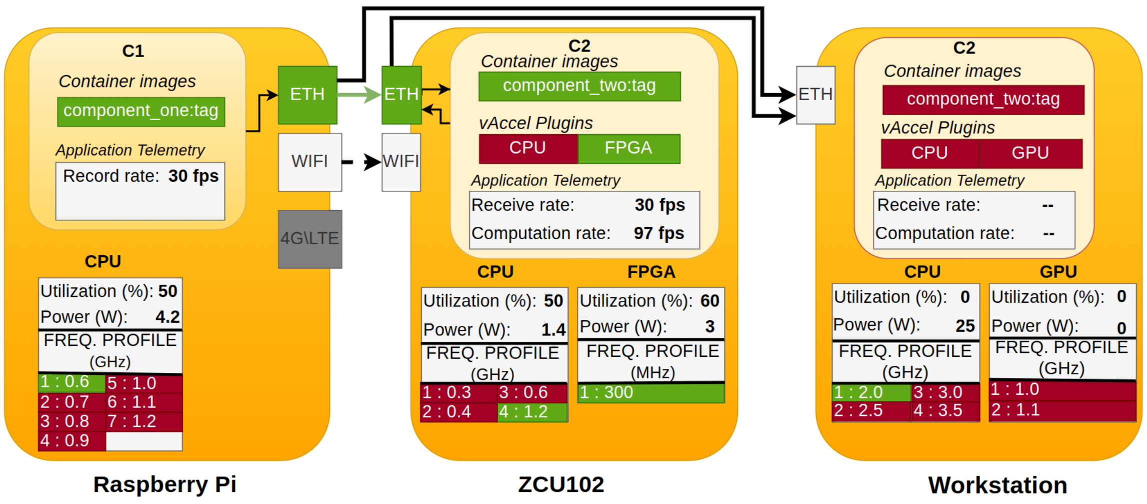

Figure 6: The dashboard for the visual deployment and performance status of the optical flow application.

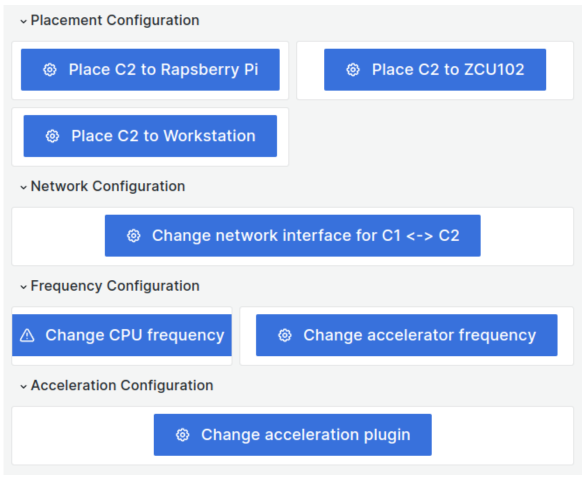

Figure 7: An instance of the control panel used for the manual activation of the different configuration options.

hardware resources. OpenTelemetry components enable a robust and flexible telemetry system, providing rich data that can be used by external policies to make management and configuration decisions. Moreover, a configuration API makes it possible to apply a variety of configuration options related to different aspects of application component placement and execution. A dashboard facilitates the real-time visualization of the status of the system and of telemetry information, while a control panel enables manual control of the system.

In our future work, we intend to extend our testbed to directly support the training and validation of pluggable machine learning models that make the various management/configuration decisions in an automated way.

Acknowledgments

This work has received funding from the Horizon Europe research and innovation programme of the European Union, under grant agreement no 101092912, project MLSysOps.

6 References

[1] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.
[2] Flask. https://palletsprojects.com/p/flask/.
[3] NVIDIA GPU exporter. https://github.com/utkuozdemir/nvidia_gpu_exporter.
[4] Grafana. https://grafana.com/oss/grafana/.
[5] K3S. https://k3s.io/.
[6] Grafana loki. https://grafana.com/oss/loki/.
[7] M. Menze and A. Geiger. Object scene flow for autonomous vehicles. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.
[8] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014:2, 2014.
[9] M. O. J. O. Muñoz, S. S. Mostafavi, V. N. Moothedath, and J. Gross. Ainur: A framework for repeatable end-to-end wireless edge computing testbed research. *ArXiv*, abs/2205.14247, 2022.
[10] Node exporter. https://github.com/prometheus/node_exporter.
[11] Openapi specification. https://swagger.io/specification/v3/.
[12] Opentelemetry. https://opentelemetry.io.
[13] Opentelemetry collector. https://github.com/open-telemetry/opentelemetry-collector.
[14] Opentelemetry transforming telemetry. https://opentelemetry.io/docs/collector/transforming-telemetry.
[15] F. Pournaropoulos, C. D. Antonopoulos, and S. Lalis. Supporting the Adaptive Deployment of Modular Applications in Cloud-Edge-Mobile Systems. In *International Conference on Embedded Wireless Systems and Networks (accepted)*, 2023.
[16] Prometheus. https://prometheus.io.
[17] N. Sabharwal and P. Pandey. *Working with Prometheus Query Language (PromQL)*, pages 141–167. Apress, Berkeley, CA, 2020.
[18] vAccel Runtime. https://github.com/cloudkernels/vaccelrt.
[19] Vitis vision library. https://xilinx.github.io/Vitis_Libraries/vision/2022.1/index.html.
[20] H. Yamanaka, Y. Teranishi, E. Kawai, H. Nagano, and H. Harai. Design of an edge computing testbed to simplify experimental setup. In *2021 24th International Symposium on Wireless Personal Multimedia Communications (WPMC)*, pages 1–6, 2021.