DECICE

DEVICE-EDGE-CLOUD INTELLIGENT COLLABORATION FRAMEWORK

Grant Agreement: 101092582

D4.2 Integration of Monitoring Framework

This project has received funding from the European Union's Horizon Europe Research and Innovation Programme under Grant Agreement No 101092582.

Document Information

Deliverable number:	D4.2
Deliverable title:	Integration of Monitoring Framework
Deliverable version:	1.0
Work Package number:	WP4
Work Package title:	Cloud Management Framework Integration
Responsible partner	USTUTT
Due Date of delivery:	2023-11-30
Actual date of delivery:	2023-11-30
Dissemination level:	PU
Type:	R
Editor(s):	Kamil Tokmakov (USTUTT)
Contributor(s):	Ashley Smith (USTUTT) Oleksandr Shcherbakov (USTUTT)
Reviewer(s):	Felix Stein(UGOE) Mirac Aydin (GWDG)
Project name:	Device-Edge-Cloud Intelligent Collaboration framEwork
Project Acronym:	DECICE
Project starting date:	2022-12-01
Project duration:	36 months
Rights:	DECICE Consortium

Document History

Version	Date	Partner	Description
0.1	2023-11-20	USTUTT	Initial Draft
0.2	2023-11-27	UGOE/GWDG	Internal review
1.0	2023-11-29	USTUTT	Final version

Acknowledgement: This project has received funding from the European Union's Horizon Europe Research and Innovation Programme under Grant Agreement No 10192582.

Disclaimer: The content of this publication is the sole responsibility of the authors, and in no way represents the view of the European Commission or its services.

Executive Summary

This deliverable contains the software components, technical descriptions, and documentation for the monitoring framework on the compute plane. The time-series database Prometheus which functions via a pull-based metric collection model is used for montoring across both cloud and edge. The proof of concept setup for the monitoring framework which was deployed on OpenStack at the University of Göttingen is outlined and additionally, a monitoring API that can be used for easier querying of metrics is introduced.

Contents

1 Purpose and Scope of the Deliverable

This report contains the software components, technical descriptions and documentation for the monitoring framework. Task T4.2 integrates monitoring systems for a range of cloud and edge hardware with the DECICE framework through persistent metrics storage (Task 3.3). Standard interfaces and agents for gathering monitoring and telemetry data across cloud and edge devices have been developed.

2 Abstract / publishable summary

The monitoring part of the DECICE framework is developed. One of the central goals of DECICE is to schedule user jobs efficiently across cloud edge and HPC. Here a proof of concept monitoring setup across cloud and edge is presented. The software components, technical descriptions and documentation for the monitoring framework are outlined. The time-series database Prometheus which functions via a pull-based metric collection model plays a central role. The proof of concept setup was deployed using OpenStack at the University of Göttingen is outlined and additionally, a monitoring API that can be used for easier querying of metrics is introduced.

3 Project objectives

This deliverable contributes directly and indirectly to the achievement of all the macro-objectives and specific goals indicated in section 1.1.1 of the project plan:

Macro-objectives	Contribution of this deliverable
(O1) Develop a solution that allows to leverage a compute continuum ranging from cloud and HPC to edge and IoT.	This deliverable outlines how monitoring can be done across the compute continuum.
(O2) Develop a scheduler supporting dynamic load balancing for energy-efficient compute orchestration, improved use of green energy, and automated deployment.	An interface for the scheduler to access and configure the compute plane is developed.
(O3) Design and implement an API that increases control over network, computing and data resources.	A model for controlling the compute plane resources through interaction with an external scheduler which can later be replaced with the DECICE glue code wrappers is introduced.
(O4) Design and implement a Dynamic Digital Twin of the system with AI-based prediction capabilities as integral part of the solution.	An interface for the scheduler to access and configure the compute plane is developed.
(O5) Demonstrate the usability and benefits of the DECICE solution for real-life use cases.	This deliverable demonstrates how applications can be monitored across a diverse range of platforms such as could and edge.
(O6) Design a solution that enables service deployment with a high level of trustworthiness and compliance with relevant security frameworks.	The monitoring framework has been setup to integrate with the DECICE framework which can apply security rules to the whole system.

4 Changes made and/or difficulties encountered, if any

No significant changes to the project plan were made. No significant challenges were encountered during implementation.

5 Sustainability

The monitoring framework is coupled to multiple WPs such as WP2, WP3. Each partner in each work package should communicate their results regularly for optimal integration of each component into the monitoring framework.

6 Dissemination, Engagement and Uptake of Results

6.1 Target audience

As indicated in the Description of the project, the audience for this deliverable is:

✓	The general public (PU)
	The project partners, including the Commission services (PP)
	A group specified by the consortium, including the Commission services (RE)
	This report is confidential, only for members of the consortium, including the Commission services (CO)

6.2 Record of dissemination/engagement activities linked to this deliverable

See Table 1.

Type of dissemination and communication activities	Details	Date and location of the event	Type of audience activities	Zenodo Link	Estimated number of persons reached
None	N/A	N/A	N/A	N/A	N/A

Table 1: Record of dissemination / engagement activities linked to this deliverable

6.3 Publications in preparation OR submitted

See Table 2.

6.4 Intellectual property rights resulting from this deliverable

None.

In preparation or submitted?	Title	All authors	Title of the periodical or the series	Is/Will open access be provided to this publication?
None	N/A	N/A	N/A	N/A

Table 2: Publications related to this deliverable

7 Detailed report on the deliverable

This deliverable contains the software components, technical descriptions and documentation for the monitoring framework outlined in Task 4.2. Monitoring systems for a range of cloud and edge hardware with the DECICE framework through persistent metrics storage (Task 3.3) are developed. Standard interfaces and agents for gathering monitoring and telemetry data across cloud and edge devices have been developed. Careful thought needs to be put into how to optimally collect and synchronize monitoring and telemetry data from a wide variety of infrastructure types. Firstly, the monitoring tasks need to have much lower overhead in comparison to the compute loads for jobs. Secondly, telemetry data for edge devices needs to be as secure and reliable as possible and account for the poor connectivity that many edge devices can be subject to. In addition to the monitoring system presented here, HWDU will handle the development of metric agents and their integration with the metric storage. Furthermore, MARUN and BIGTRI will contribute to the collection of the metrics and optimize telemetry data collection frequency and synchronization.

This section of the report is structured into three sections. First, we outline the Prometheus concepts needed to understand the monitoring framework. After that, a Proof of Concept (PoC) setup is described for deploying the monitoring framework. Lastly, a detailed list of metrics and the queries needed to obtain them is provided.

7.1 Core Components of the Monitoring Framework

The monitoring system collects and stores data that is used as input to the Digital Twin (DT) with the quality of the data being critical to the performance of the DECICE scheduler which uses the data from the DT. The monitoring framework needs to collect and aggregate metrics about the current system state from a variety of devices such as Internet of Things devices, cloud worker nodes, and high performance computing nodes. This can be achieved by accessing the underlying platforms through their interfaces such as the Kubernetes API on the cloud or KubeAPI on edge. Here we present a PoC setup for the monitoring framework outlining the main tools and technologies employed. In section 7.2 a more detailed description of the architecture and deployment model will be given.

Prometheus [1] is a time-series-based metric collection system that plays a central role in our monitoring framework. It monitors and collects metrics from various targets, such as applications, services, and infrastructure components in a highly scalable and efficient manner. It operates based on a client-server architecture where the Prometheus server periodically retrieves metrics from client components if it knows how to reach them. Figure 3 below introduces how Prometheus works. The following paragraphs introduce the most important components of Prometheus.

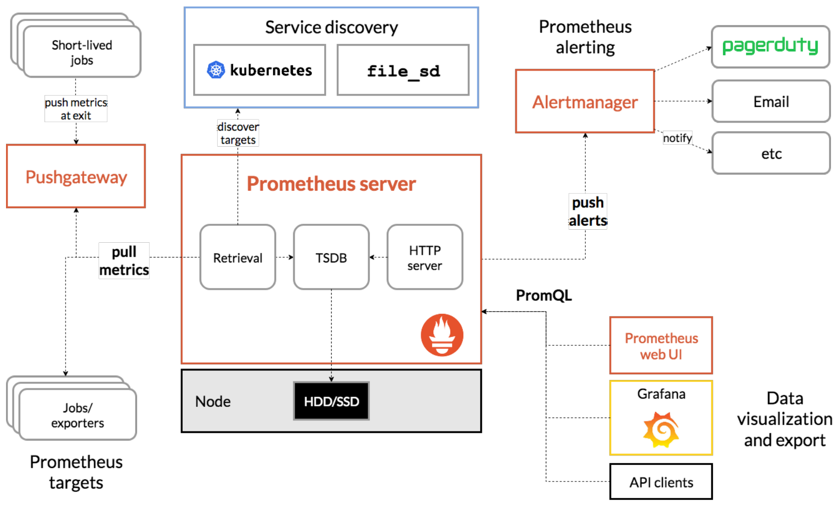

Figure 1: Architecture of Prometheus [1].

Exporters are small processes responsible for exposing specific metrics from applications or systems. They can be thought of as the client in the client-server model (with Prometheus being the server). They can be categorized into standalone exporters e.g. a commonly used exporter is the Node Exporter for various server metrics. And application exporters e.g. an exporter for a database like Elasticseach [3] or an API like FastAPI [9].

Prometheus pull mode. Prometheus uses a pull-based model for data collection, meaning that it periodically scrapes metrics from configured targets over HTTP. This monitoring method contrasts with the push model used in many other monitoring systems. In the push model paradigm application code is commonly modified to send metrics to the server periodically. The advantage of the pull model is that it does not require the systems or applications that are being monitored to be aware of the Prometheus server, so it can be added to a monitoring system without making changes to the application code. Furthermore, the pull model only collects metrics when they are needed, so Prometheus does not waste resources by collecting metrics that are not being used.

Prometheus push mode Pull mode excels when operating within a well-defined network perimeter. Push mode becomes imperative in the presence of network boundaries, firewalls, or NAT configurations. For some systems it is not realistic to continuously provide metrics. For example, jobs on that system may be short-lived, if your scraping interval is set to 30 seconds, but a particular job runs for only 20 seconds, it makes sense for these transient jobs to report their metrics directly to the Pushgateway. A push-based metric collection system e.g. the Prometheus Pushgateway allows the time series to be pushed to an intermediary job which Prometheus can then scrape. This approach may require the jobs to incorporate HTTP requests into their execution or employ sidecar containers to handle the reporting process. It is important to be aware of potential downsides, such as the possibility of overwriting labels and the need for manual label management, which you can find detailed information about in the documentation for e.g. [12].

Prometheus proxy Proxy mode comes into play when a pull approach is required across network boundaries, necessitating access to exporters via a proxy (it is important to note that this proxy might lack authentication or encryption – consult the documentation for PushProx). PushProx represents an alternative method for gathering metrics from network boundaries. This is a client and proxy that allows transversing of NAT and other similar network topologies by Prometheus, while still following the pull model. For a detailed explanation of this architecture, you can refer to the diagram available on GitHub [13]. Within DECICE, PushProx will be needed to incorporate HPC systems. In HPC systems, the nodes do not have direct internet access and rely on a proxy on the frontend nodes to access the internet. This is a deliberate design choice made to reduce the attack surface and potential for malicious usage. To circumvent this problem the master nodes can use the proxy to communicate with the Kublets running on the HPC nodes enabling the pull model to be preserved.

Grafana is an open source data visualization and monitoring platform that supports Prometheus. It allows you to create interactive and customizable dashboards to visualize data from various sources in real time. Grafana supports a wide range of data sources, including popular databases, cloud monitoring services, time series databases, and more.

7.2 Architecture and Deployment Model

7.2.1 Proof of Concept

Here the key elements of the setup e.g. software components to configure will be outlined. More detailed setup and installation instructions are available here [2]. For an overview of the overall high-level architecture please refer to the first figure of D2.1. For our PoC, Kubernetes and KubeEdge clusters were deployed on the cloud infrastructure at GWDG. The monitoring framework was deployed on top of the clusters via the Prometheus Stack Helm charts [11]. The Prometheus Stack includes the following relevant components:

- **Prometheus Operator** [10] provides simplified configuration and management of Prometheus, specifically in Kubernetes.
- **Exporters**: **Node Exporter** [7] and **Kubernetes State Metrics** [6]. Node Exporter is an interface on the cluster nodes to retrieve the system metrics. Kubernetes State Metrics is an exporter to expose the Kubernetes cluster-level metrics.
- **Grafana**: provides a web-UI for displaying Prometheus metrics.

Three VMs were used in our PoC:

- **Cloud** VM master node with Kubernetes, Prometheus Operator, and Clourdcore as part of KubeEdge installed.
- **Edge** VM worker node with KubeEdge. KubeEdge itself consists of CloudCore and EdgeCore. CloudCore is installed on the cloud VM, whereas EdgeCore is on the edge VM.
- VM for a **dummy scheduler** as a placeholder for more advanced schedulers further in the project.

The resources and layout are shown in Figure 2 and Table 3. Figure 3 shows the overall architecture of our PoC setup.

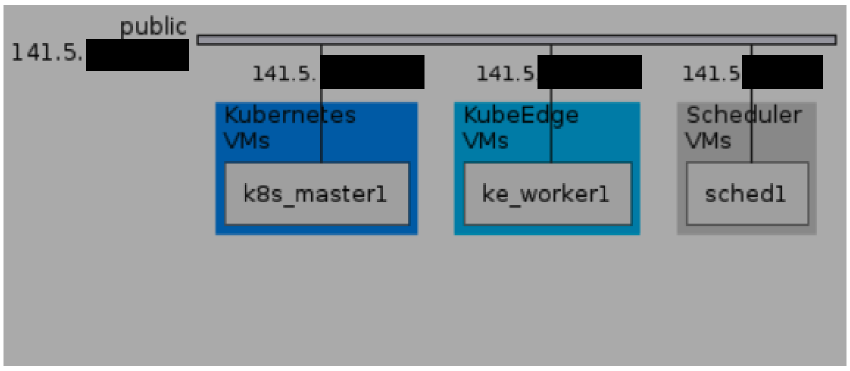

Figure 2: Proof of concept setup for the monitoring framework. Real IP addresses redacted.

Node	VM Name	Image Name	Resources
Kubernetes(k3s v1.24.10+k3s1, Prometheus)	k8s_master1	Ubuntu 22.04.2 Server x86_64 (ssd)	c1.large (8 vCPU; 8 GB RAM; 80 GB Storage)
KubeEdge	ke_worker1	Ubuntu 22.04.2 Server x86_64 (ssd)	m1.medium (2 vCPU; 4 GB RAM; 40 GB Storage)
Scheduler	sched1	Ubuntu 22.04.2 Server x86_64 (ssd)	c1.small (2 vCPU; 2 GB RAM; 20 GB Storage)

Table 3: Description of minimal setup

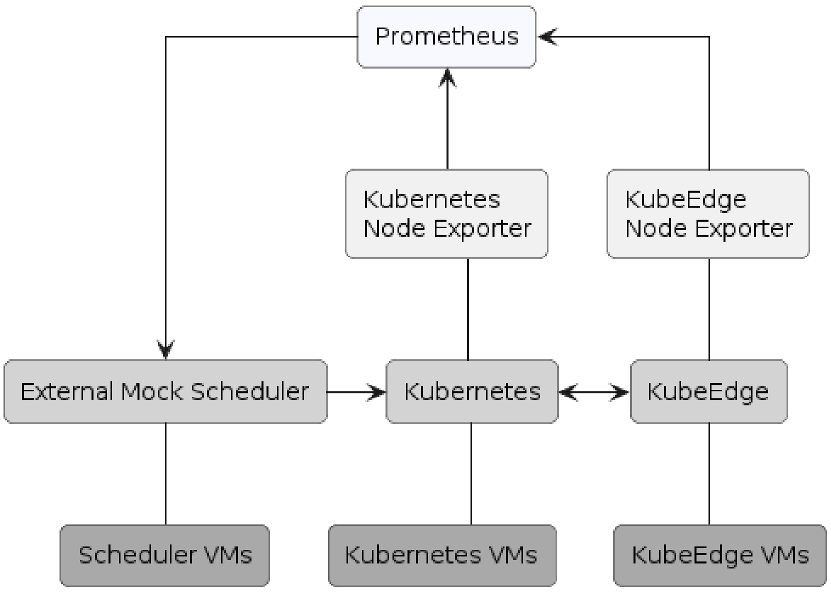

Figure 3: Proof of concept setup for the monitoring framework

The Prometheus Operator [10] follows the Kubernetes Operator pattern where custom resources (i.e. extensions of the Kubernetes API that are not necessarily available in a default Kubernetes installation) can be used to manage applications and their components. Using custom resources allows us to extend the set of available Kubernetes objects, and in this case, allows us to define and manage Prometheus objects (such as Prometheus itself, AlertManager, Prometheus Rules, etc.) in a Kubernetes-native way without prior knowledge of complex Prometheus-specific configurations. Due to the declarative nature of Kubernetes, one can define and later modify the Prometheus configurations, and the changes will be reflected in the Prometheus deployment and runtime. The Prometheus Operator was installed using the helm chart [11] with additional endpoint */metrics/resources* for obtaining resources requests and limits for the pods (*kube_pod_resource_limits* and *kube_pod_resource_requests*) and with the node selector monitoring enabled. Exposing scheduler metrics would not be possible in a public k8s cloud provider as it would reveal internal infos of the cloud resources. In addition for more metrics related to the network we also suggest snmp-exporter and blackbox-exporter could be explored. When GPU based workflows are intergrated into the project we also suggest the use of nvidia-dcgm-exporter, nvidia-smi-exporter and rocm-exporter.

To create an ingress for Prometheus and Grafana the security rules in Table 4 were applied. Both Prometheus and Grafana then provide endpoints that can be queried for metrics. Both the Prometheus and Grafana endpoints support queries via the Prometheus query language promQL. An additional Monitoring API shown in Figure 4 for querying metrics was developed. It acts as an interface to query monitoring metrics, providing a user and developer friendly way to query the metrics instead of writing complicated metrics ourselves.

Security Group	Rules	VMs	Description
kubernetes	TCP/6443	k8s_master1	Kubernetes API interface
kubeedge_cloudcore	TCP/10000, TCP/10002	k8s_master1	For edgecore connection and getting CA certificates
ingress	TCP/443	k8s_master1	Ingress for deployed web applications (HTTPS)

Table 4: Security rules for proof of concept setup

7.2.2 Expanded model

For the PoC, Prometheus was deployed on the Master VM. Looking forward, a central question is how Prometheus should be deployed across clusters. Two popular approaches are: deploying per-domain/cluster and deploying in a centralized/federated way.

Centralized Prometheus can suffer from issues of scalability as lots of configuration and reconfiguration of exporters needs to be done on the Prometheus side. A solution to this is to instead use federated Prometheus or separate Prometheus servers that the DECICE API can access. Network boundaries and security would need to be configured to expose the exporters in certain environ-

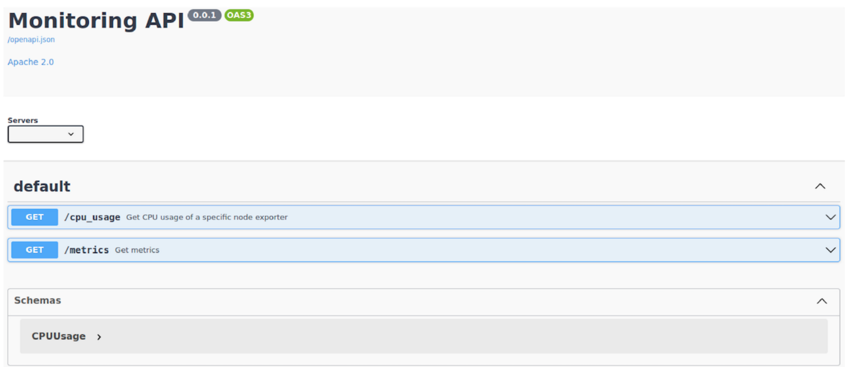
Monitoring API 0.0.1 OAS3
/openapi.json
Apache 2.0
Servers
default
GET /cpu_usage Get CPU usage of a specific node exporter
GET /metrics Get metrics
Schemas
CPUUsage

Figure 4: OpenAPI specification of the monitoring API interface

ments, such as isolated or private clusters. Another approach could be via PushProx, PushGateway although this would also introduce additional constraints such as latency in collecting metrics and the need for storage on small devices as remote Prometheus writes require storage.

An alternative strategy is to deploy a Prometheus server within each cluster and directly access the server instead of reaching out to individual exporters. Utilizing exporters, particularly those within the same clusters can involve additional infrastructure components such as load balancing, ingress controllers, services exposing node ports, or similar solutions. This may result in exporter traffic leaving the local network, potentially leading to bottlenecks, security vulnerabilities, and the risk of data leakage. Managing numerous exporters and configuring network proxies can also pose scalability and management challenges. Conversely, managing separate Prometheus servers requires storage resources, which can be a concern, especially in resource-constrained environments. Therefore, it is advisable to strike a balance between scalability and resource allocation.

Moving forward the next setup we suggest for our test environment is illustrated in Figure 5. At some point in the future, the dummy scheduler will be replaced when the monitoring framework is integrated into the full DECICE architecture.

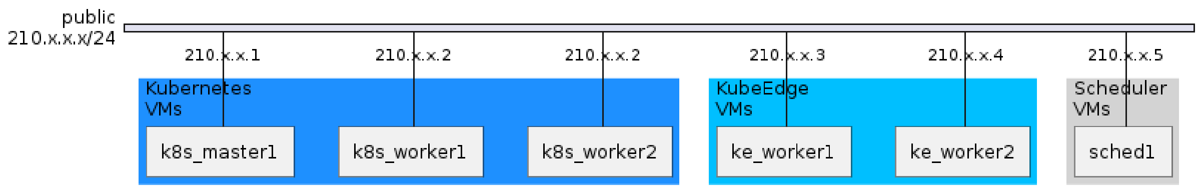

Figure 5: Expanded monitoring setup

7.2.3 HAICGU cluster

A Kubernetes cluster was created on top of a set of nodes in the HAICGU cluster. Specifically, a multi-master setup with two master nodes, four worker nodes and an edge node running KubeEdge was deployed using version v1.24 of Kubernetes and v1.14 of KubeEdge as shown in Figure 6. The *dev* node in Figure 6 is a development machine where the DECICE users can build their code and push it to the Kubernetes cluster to be subsequently deployed.

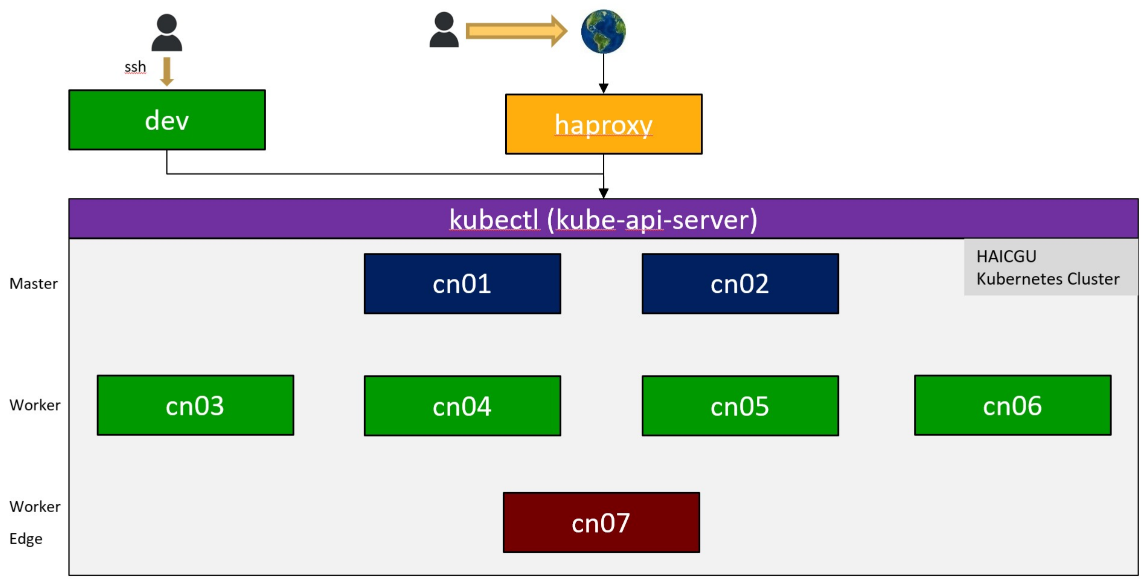

Figure 6: HAICGU Kubernetes Cluster Architecture.

Inside the Kubernetes cluster, multiple components were installed to provide different functionalities. These components are listed below followed by a description of their purpose in the context of the DECICE project and technical details regarding their deployment:

- **Kube-state-metrics**
  - Description. Kube-state-metrics is an add-on agent for Kubernetes that generates metrics about the state of Kubernetes objects, such as deployments, nodes, and pods. It does this by listening to the Kubernetes API server and translating the object status information into Prometheus-compatible metrics. Kube-state-metrics is not focused on the health of individual Kubernetes components, but rather on the health of the various objects inside the cluster. This makes it a valuable tool for monitoring the overall health of a Kubernetes cluster and identifying potential issues.
  - Purpose. Generation of Kubernetes cluster, low-level metrics to be passed to Prometheus.
  - Deployment Strategy. Basic Kubernetes *Deployment* (1 instance).
- **metrics-server**
  - Description. Metrics Server is a crucial component in the Kubernetes ecosystem, serving as a scalable and efficient source of container resource metrics for Kubernetes' built-in autoscaling pipelines. It operates by collecting resource metrics from Kubelets, the agents responsible for managing pods on nodes, and exposes them through the Kubernetes API server via the Metrics API. This data is then utilized by Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA) to dynamically adjust the number of pods in a deployment based on real-time resource usage. Metrics Server also facilitates debugging autoscaling pipelines by enabling users to access resource metrics using the *kubectl* top command.

  - Purpose. Similarly to kube-state-metrics, it collects cluster-level metrics. However, this component is necessary for Metricbeat to work properly.
  - Deployment Strategy. Basic Kubernetes *Deployment* (1 instance).

- **Prometheus**
  - Description. Prometheus is a free and open-source monitoring system and time series database (TSDB) built by SoundCloud. It collects metrics from targeted systems and services and stores them in a time series database for later retrieval and analysis. Prometheus can be used to monitor a wide range of systems, including servers, applications, and infrastructure devices. It is highly customizable and can be extended with a wide range of exporters and dashboards.
  - Purpose. Store time-series metrics and provide an alerting system. This component is crucial for the DECICE scheduler.
  - Deployment Strategy. *Deployment* (1 instance) with *PersistentVolume* mapped on top of NFS hosted inside the HAICGU cluster.

- **Prometheus Node Exporter**
  - Description. Prometheus Node Exporter is a free and open source exporter that collects and exposes a wide range of hardware- and kernel-related metrics from Linux systems. It provides a rich set of metrics that can be used to monitor the health and performance of servers, including CPU usage, memory usage, disk I/O, network traffic, and more. Prometheus Node Exporter is a valuable tool for any organization that wants to proactively monitor its infrastructure and identify potential problems before they cause outages.
  - Purpose. Export metrics from the nodes in the Kubernetes cluster and send them to Prometheus. This is especially useful for edge nodes to transfer data to a remote Prometheus master node.
  - Deployment Strategy. *Daemonset* (1 instance for each node in the Kubernetes cluster).

- **Grafana**
  - Description. Grafana is an open source observability platform that allows you to collect, visualize, alert on, and understand metrics, logs, and traces from a variety of data sources. It provides a wide range of visualization options, including charts, graphs, and dashboards, as well as alerting capabilities to notify you of critical issues. Grafana is highly extensible through a plugin system, which allows you to integrate it with other tools and services. It is a popular choice for monitoring infrastructure, applications, and DevOps pipelines.
  - Purpose. Similar to Kibana, it is used to prepare dashboards and visualize data collected by Prometheus or Elasticsearch.
  - Deployment Strategy. Basic Kubernetes *Deployment* (1 instance).

- **Private Docker Registry**

  – A private Docker registry is a repository for storing and managing Docker images that is not publicly accessible. This means that only authorized users can access and download images from the registry. Private Docker registries are often used by organizations that need to store sensitive or confidential images, or that want to have more control over the distribution of their images.
  – Purpose. Provide a secure, high-performance, and fully controllable registry where to store Docker images created by the DECICE members.
  – Deployment Strategy. Basic Kubernetes *Deployment* (1 instance) mapped to a *PersistentVolume* to safely store pushed Docker images.

Of the above components, only Grafana is accessible from the outside world through username and password authentication. Prometheus will be enabled after appropriate security measures are in place.

Finally, a hands-on tutorial explaining how to access the cluster, deploy a basic application, and additional information about the installed components can be found on GitHub [4]. In brief, the repository contains the following:

- A set of PowerPoint slides with a detailed walkthrough to access the cluster, deploy two MQTT-based applications, and provide information about the role of KubeEdge in the DECICE project.
- Python code for two MQTT examples: *(i)* MQTT Basic Publisher/Subscriber and *(ii)* Data Aggregation and Collection at the Edge.
- Dockerfiles to build the containers with the MQTT components (publisher, subscriber, aggregator, etc.).
- Customizable YAML files to deploy the containers on the Kubernetes cluster.
- BASH scripts to tailor the deployment for each user.

The live hands-on tutorial has been delivered at the DECICE Workshop in Vienna.

7.3 Metrics

This section describes the metrics collection and queries used to collect the metrics in order to provide data for the DECICE digital twin and which will be utilised by the DECICE scheduler. We have catergorised the metrics into Job and Queue, Hardware and Kubernetes specific metrics.

7.3.1 Job and Queue

Job Completion and Status and Queue Metrics (we are interested in scheduling user workloads and therefore exclude "kube-system" namespace as these are the workloads (pods) of Kubernetes itself and should not be rescheduled). The following metrics are described below:

- Specification of the jobs
- Pending jobs

- Running jobs with their mapping to resources
- Priority
- Restrictions
- Statistics about job arrival
- Time of day
- Common burst times
- Jobs scheduled on a node
- Scheduler Performance

Specification of the jobs

Description
Specification and runtime information about pods and containers: container images, namespaces, nodes, pod IP, etc.

Queries
Information about pods: `kube_pod_info{namespace!="kube-system"}`
Units: list of labels with information about pods

Information about containers: `kube_pod_container_info{namespace!="kube-system"}`
Units: list of labels with information about containers

Notes
It is exposed via kube-state-metrics exporter

Pending jobs

Description
A list of pods that are pending for execution

Queries
`kube_pod_status_phase{namespace!="kube-system", phase="Pending"} == 1`
The query should contain equality to 1 (`== 1`) in order to retrieve the pods that are pending at the moment. Otherwise, the pods that *were* pending in the past would also be included
Units: list of objects with information about pods with "Pending" status

Notes
It is exposed via kube-state-metrics exporter

Running jobs with their mapping to resources

Description
Requests and limits of CPU and memory resources for the scheduled pods. It is more precise than container requests and limits (*kube_pod_container_resource_requests*, *kube_pod_container_resource_limits*), as these metrics are obtained directly from kube-scheduler.

Queries

CPU requests of pods:

`kube_pod_resource_request{namespace!="kube-system",resource="cpu"}`

Units: Fraction of CPU cores, e.g. 0.2, 2.7

CPU limits of pods:

`kube_pod_resource_limit{namespace!="kube-system",resource="cpu"}`

Units: Fraction of CPU cores, e.g. 0.2, 2.7

Memory requests of pods:

`kube_pod_resource_request{namespace!="kube-system",resource="memory"}`

Units: Bytes

Memory limits of pods:

`kube_pod_resource_limit{namespace!="kube-system",resource="memory"}`

Units: Bytes

Notes

It is exposed via kube-scheduler exporter

Priority

Description

Priority of pods in the queues of kube-scheduler

Queries

`kube_pod_resource_request{namespace!="kube-system",resource="cpu"}`

The output of the query will contain the "priority" label

Units: integer

Notes

It is exposed via kube-scheduler exporter

Restrictions

Description

Restrictions for the selection of nodes, where pods will be scheduled to. Tolerations and node selectors of the pods are considered in this case

Queries

Tolerations: `kube_pod_tolerations{namespace!="kube-system"}`

Units: list of labels with information about tolerations of pods

Node selectors: `kube_pod_nodeselectors{namespace!="kube-system"}`

Units: list of labels with information about node selectors of pods

Notes

It is exposed via kube-state-metrics exporter. An opt-in argument with the node selector metric must be supplied when starting kube-state-metrics, e.g.

```
--metric-opt-in-list=kube_pod_nodeselectors
```

Statistics about job arrival

Description

Number of requests to create or modify pods. In this case, the number of requests over the last 24 hours at the time of metrics collection is considered, however, this range can be arbitrarily modified. The HTTP POST, PUT and PATCH requests to Kubernetes API server are monitored.

Queries

```
floor(increase(
  apiserver_request_total{
    endpoint="https",resource="pods",verb=~"POST|PUT|PATCH",subresource!="status"
  }[24h]
))
```

The range `[24h]` in the request can be changed to the suitable one. The scope of Kubenetes API metrics is further narrowed: pods, HTTP verbs and non-status changing requests (i.e. requests to change the status of the pods, e.g. from Pending to Running). The *increase()* function calculates the difference between the first and last metrics of the list of metrics (delta over time). The *floor()* function is needed when the timeseries are smaller than the time range specified: it may cause artifacts due to out of range of metrics
Units: integer

Notes

It is exposed via kube-apiserver exporter

Time of day

Description

Day of week, hour and minute in UTC when the pod was created

Queries

Day of week: `day_of_week(kube_pod_start_time{namespace!="kube-system"})`
Units: integer, UTC format

Hour: `hour(kube_pod_start_time{namespace!="kube-system"})`
Units: integer, UTC format

Minute: `minute(kube_pod_start_time{namespace!="kube-system"})`
Units: integer, UTC format

Notes

It is exposed via kube-state-metrics exporter

Common burst times

Description

Bursts of user requests over time. This metric will be addressed later in the project. An approach similar to the anomaly detection using Prometheus [5] will be evaluated. In this case, requests to Kubernetes API will be monitored.

Queries

```
apiserver_request_total{
  endpoint="https",resource="pods",verb=~"POST|PUT|PATCH",subresource!="status"
}
```

Units: integer

Notes

It is exposed via kube-apiserver exporter

<u>**Jobs scheduled on a node**</u>

Description

Number of jobs scheduled on the nodes of the cluster

Queries

`sum (kube_pod_info{namespace!="kube-system"}) by (node)`

The output of the query will contain the "node" label and the number of pods on each node

Units: integer

Notes

It is exposed via kube-state-metrics exporter

<u>**Scheduler Performance**</u>

Description

Performance metrics of the kube-scheduler

Queries

Latency for running all plugins of a specific extension point:

`scheduler_framework_extension_point_duration_seconds_sum{namespace!="kube-system"}`

Units: seconds

Number of pending pods, by the queue type:

`scheduler_pending_pods{namespace!="kube-system"}`

Units: integer

Number of attempts to successfully schedule a pod:

`scheduler_pod_scheduling_attempts_sum{namespace!="kube-system"}`

Units: integer

End-to-end latency for a pod being scheduled which may include multiple scheduling attempts:

`scheduler_pod_scheduling_duration_seconds_sum{namespace!="kube-system"}`

Units: seconds

Total preemption attempts in the cluster till now:

`scheduler_preemption_attempts_total{namespace!="kube-system"}`

Units: integer

Number of selected preemption victims:

`scheduler_preemption_victims_sum{namespace!="kube-system"}`

Units: integer

Number of pods added to scheduling queues by event and queue type:
`scheduler_queue_incoming_pods_total{namespace!="kube-system"}`
Units: integer

Number of attempts to schedule pods, by the result:
`scheduler_schedule_attempts_total{namespace!="kube-system"}`
Units: integer

Scheduling attempt latency in seconds (scheduling algorithm + binding):
`scheduler_scheduling_attempt_duration_seconds_sum{namespace!="kube-system"}`
Units: seconds

Notes
It is exposed via kube-apiserver exporter

7.3.2 Hardware Resources

The following metrics are described below:

- Network
- CPU usage and type
- Memory usage
- Storage

Network

Description
Network traffic received on a given node

Queries
Network traffic received in bits per second in the last minute
```
rate(node_network_receive_bytes_total{job="node-exporter",
instance="10.254.X.XX:XXXX", device="ens3"}[1m]) * 8
```
Units: bits per second

Network traffic transmitted in bits per second in the last minute
```
rate(node_network_transmit_bytes_total{job="node-exporter",
instance="10.254.X.XX:XXXX", device="ens3"}[1m]) * 8
```
Units: bits per second

Notes
Real IP address redacted

CPU usage and type

Description
CPU usage and load.

Queries

CPU Usage

```
(1 - sum without (mode) (rate(node_cpu_seconds_total{job="node-exporter",
mode=~"idle$\vert$iowait$\vert$steal", instance="10.254.X.XX:XXXX"}[1m])))
ignoring(CPU) group_left count without (cpu, mode) node_cpu_seconds_total
({job="node-exporter", mode="idle", instance="10.254.X.XX:XXXX"})
```

Units: Seconds

Load Average over 1 minute

```
node_load1{job="node-exporter", instance="10.254.X.XX:XXXX"}
```

Units: integer

Notes

Real IP address redacted

Memory usage

Description

Memory usage in bytes

Queries

```
100 - (avg(node_memory_MemAvailable_bytes{job="node-exporter",
 instance="10.254.X.XX:XXXX"}) / avg(node_memory_MemTotal_bytes{job="node-exporter",
instance="10.254.X.XX:XXXX"}) * 100 )
```

Units: bytes

Notes

Real IP address redacted

Storage

Description

Disk Available Space on root file system in bytes

Queries

```
node_filesystem_avail_bytes{job="node-exporter", instance="10.254.X.XX:XXXX",
mountpoint="/"}
```

Units: bytes

Notes

Real IP address redacted

7.3.3 Kubernetes

Job and Queue metrics, presented in Section 7.3.1, overlap with Kubernetes metrics, however, there are additional Kubernetes specific metrics that are needed for DECICE. The following Kubernetes specific metrics are described below:

- Pod Scheduling Latency

- Pod Startup Time
- Resource Utilization
- Pod/Container Resource Consumption
- Network Performance
- Pod Health

Pod Scheduling Latency

See *End-to-end latency for a pod being scheduled which may include multiple scheduling attempts* in Section 7.3.1

Pod Startup Time

Description
Start time as Unix timestamp for a pod

Queries
`kube_pod_start_time{namespace!="kube-system"}`
Units: integer, seconds

Notes
Exposed via kube-state-metrics exporter

Resource Utilization

Description
CPU and Memory usage in the cluster

Queries

CPU Usage
`cluster:node_cpu:ratio_rate5m`
Units: utilization ration, can be greater than 1.0

Memory Usage
```
1 - sum(:node_memory_MemAvailable_bytes:sum{cluster=""}) /
    sum(node_memory_MemTotal_bytes{job="node-exporter",cluster=""})
```
Units: utilization fraction

Pod/Container Resource Consumption

Description
Rsource consumption per container or per pod for certain pods and/or on certain nodes

Queries
CPU usage by container
```
sum(node_namespace_pod_container:container_cpu_usage_seconds_total:sum_irate{
  namespace!="kube-system", pod="nginx", cluster=""
  }) by (container)
```
Units: seconds

CPU usage of pods, with a node filter
```
sum(node_namespace_pod_container:container_cpu_usage_seconds_total:sum_irate{
  node=~"k8s-master1"
  }) by (pod)
```
Units: second

Memory usage by pod, e.g. in the "decice" namespace
```
sum(container_memory_working_set_bytes{
  namespace="decice", pod="nginx", container!="", image!=""
  }) by (container)
```
Units: Bytes

Memory usage of pods, with a node filter
```
sum(node_namespace_pod_container:container_memory_working_set_bytes{
  node=~"k8s-master1", container!=""
  }) by (pod)
```
Units: Bytes

Network Performance

Description
Network utilization per pod, e.g. in the "decice" namespace

Queries
```
sum(irate(container_network_receive_bytes_total{
  namespace=~"decice"}[4h:5m])) by (pod)
sum(irate(container_network_transmit_bytes_total{
  namespace=~"decice"}[4h:5m])) by (pod)
```
Units: Bytes/second

Pod Health

Description
Queries for discovering issues with pods. "Pending" pods query can be used by custom scheduler. Further metrics are available in the documentation [8]

Queries

Pods in "Unknown" state
```
sum(kube_pod_status_phase{phase="Unknown"}) by (namespace, pod) or
  (count(kube_pod_deletion_timestamp) by (namespace, pod) *
  sum(kube_pod_status_reason{reason="NodeLost"}) by(namespace, pod))
```

Pods in "Pending" state (not scheduled
```
sum(kube_pod_status_phase{phase="Pending"}) by (namespace, pod) == 1
```

8 References

[1] Prometheus Authors. *Overview — Prometheus*. https://prometheus.io/docs/introduction/overview/. [Online; accessed 2023-11-28].

[2] *DeciceCode*. https://github.com/DECICE-project. [Online; accessed 2023-11-28].

[3] Prometheus Community Eric Richardson. *Elasticsearch Exporter*. https://github.com/prometheus-community/elasticsearch_exporter. [Online; accessed 2023-11-28].

[4] *Hands-on tutorial*. https://github.com/rho770/oehi-cc-training. [Online; accessed 2023-11-28].

[5] *Hands-on tutorial*. https://about.gitlab.com/blog/2019/07/23/anomaly-detection-using-prometheus/. [Online; accessed 2023-11-28].

[6] *KubeSM*. https://github.com/kubernetes/kube-state-metrics. [Online; accessed 2023-11-28].

[7] *NodeExprt*. https://github.com/prometheus/node_exporter. [Online; accessed 2023-11-28].

[8] *PodMetrics*. https://github.com/kubernetes/kube-state-metrics/blob/main/docs/pod-metrics.md. [Online; accessed 2023-11-28].

[9] *Prometheus FastAPI Instrumentator*. https://github.com/trallnag/prometheus-fastapi-instrumentator. [Online; accessed 2023-11-28].

[10] *PromOperator*. https://github.com/prometheus-operator/prometheus-operator. [Online; accessed 2023-11-28].

[11] *PromStack*. https://github.com/prometheus-community/helm-charts/tree/main/charts/kube-prometheus-stack. [Online; accessed 2023-11-28].

[12] *Pushing metrics — Prometheus*. https://prometheus.io/docs/instrumenting/pushing/. [Online; accessed 2023-11-28].

[13] *PushProx*. https://github.com/prometheus-community/PushProx. [Online; accessed 2023-11-28].