DECICE

DEVICE-EDGE-CLOUD INTELLIGENT COLLABORATION FRAMEWORK

Grant Agreement: 101092582

D2.1 Specification Of The Optimization Scope

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:	D2.1
Deliverable title:	Specification Of The Optimization Scope
Deliverable version:	1.0
Work Package number:	WP2
Work Package title:	AI-Scheduler for Optimization and Adaption
Responsible partner	The Numerical Algorithms Group (NAG) Ltd.
Due Date of delivery:	2023-08-31
Actual date of delivery:	2023-08-31
Dissemination level:	PU
Type:	R
Editor(s):	NAG
Contributor(s):	UGOE GWDG UNIBO
Reviewer(s):	Dirk Pleiter (KTH) Karthee Sivalingam (Huawei) Jonathan Decker (UGOE)
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	2023-08-24	NAG, UGOE, GWDG	Initial Draft
0.2	2023-08-26	NAG, UGOE, GWDG, UNIBO, KTH	Second Draft and Review
1.0	2023-08-31	NAG, UGOE, GWDG, UNIBO, KTH	Final Version

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

Executive Summary

The main aim of this deliverable is to define, discuss and evaluate ways to optimize the DECICE project by proposing a structural framework as well as possible improvements to it. The document provides an overview of the project's architecture, analyzes its core elements, and explains the scheduling process mechanism. The primary focus, however, is on the discussion and evaluation of optimization strategies. Methods for scheduling and optimization are summarized, along with an in-depth examination of architectural elements such as the AI scheduler, Digital Twin, and synthetic test environment. Furthermore, the document emphasizes the significance of metric aggregation using Prometheus, which is crucial for designing the digital twin and training the AI scheduler as well as enabling system health checks. Finally, the deliverable considers options for enhancing I/O performance and highlights areas for further exploration such as federated learning and container migration. As a result, the deliverable provides insights into the DECICE architecture, acts as a roadmap for future improvements, and identifies potential challenges.

Contents

1 Purpose and Scope of the Deliverable

The purpose of deliverable 2.1: *Specification of The Optimization Scope* is to give an overview of the architectural components within the DECICE framework and to define how these components can potentially be optimized. By investigating the core components of the framework, we aim to point out inefficiencies, bottlenecks, and opportunities for improvement. Moreover, this analysis will aid in defining the scope of our improvement efforts, providing a better approach for the project's development.

2 Abstract / publishable summary

This deliverable presents definitions and consideration to find and discuss the key concepts and ideas for optimization potential within the DECICE project with a focus on various core components of the framework itself. We are proposing enhancements for refining its integral structure and also improving the efficiency of the compute plane components. The document provides an overview of the high-level architecture that represents the DECICE project, along with a comprehensive analysis of its core elements and potential room for optimisation. We have also covered the scheduling process mechanism for this architectural design. The key to our effort, though, is to improve these components' performance. As a result, we have provided a brief summary of the scheduling and optimization methods. Additionally, this necessitates a thorough examination of crucial architectural elements including the AI scheduler, the digital twin, and the synthetic test environment, each of which is discussed in detail in its own part. Additionally, we covered the topic of metric aggregation via Prometheus. These metrics play a crucial role in designing the digital twin that represents the system digitally and to provide more data to the AI scheduler that needs to be trained with cluster data. Our work also explores the critical parameters needed to boost I/O performance, a crucial element in maximizing disk efficiency. We explore potential areas for additional thought after our analysis is finished. The terms federated learning and container migration stand out among these. These complex subjects have the ability to redefine the paradigm for edge devices, crucial elements of the compute plane of the DECICE project. In conclusion, the work in this deliverable helps to understand the functioning of the core parts of the DECICE architecture and explains them in detail. It also acts as a road map for locating future bottlenecks and improvement opportunities.

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:

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.

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.	The deliverable defines the high-level architecture of DECICE and specifies the optimization scope of its components.
(O2) Develop a scheduler supporting dynamic load balancing for energy-efficient compute orchestration and automated deployment.	Defining an optimization scope helps to collect valuable and suitable metrics and to improve the schedulers' training quality.
(O3) Design and implement an API that increases control over network, computing and data resources.	The deliverable depicts a Virtual Training Environment that gives developers an environment to design and improve APIs.
(O4) Design and implement a Dynamic Digital Twin of the system with AI-based prediction capabilities as integral part of the solution.	The deliverable defines the Digital Twin components and explains how these components can be potentially improved.
(O5) Demonstrate the usability and benefits of the DECICE solution for real-life use cases.	Designing optimized hardware and software increases the performance of real-life use cases.
(O6) Design a solution that enables service deployment with a high level of trustworthiness and compliance with relevant security frameworks.	Optimization of the scheduler will help to deploy more reliable services and a security API will apply security rules to the entire ecosystem.

5 Sustainability

Design and optimization of components in the project are tightly coupled to multiple WPs such as WP2, WP3 and WP4. Every partner in each work package will communicate their result regularly for an optimal integration of each component into the 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.

6.3 Publications in preparation OR submitted

See Table 2.

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	0

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

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	

Table 2: Publications related to this deliverable

6.4 Intellectual property rights resulting from this deliverable

None.

7 Detailed report on the deliverable

This deliverable deals with the technical description of the system characteristics and metrics necessary for the scheduler and the supported user-requirements for jobs and workloads. For this purpose the deliverable is divided into four major parts. The first section deals with the DECICE framework and provides a detailed overview of the high-level architecture which is separated into two parts, the framework itself and the compute plane. Afterwards a short description of an exemplary workflow for a job submission is given. The second part of the deliverable deals with scheduling and optimization algorithms which ultimately lay the foundation for the AI-scheduler of the DECICE model. An outline for an optimal scheduling decision and how to achieve it is described and put into context together with potential limitations and restrictions that might appear during a scheduling decision.

The third part of this deliverable details the core components of the DECICE framework regarding the optimization scope. First the digital twin, which will represent the compute plane and serve as the foundation for job placement for the AI scheduler, together with its internal structure and requirements is discussed. Following the digital twin, the virtual training environment is introduced and described. Its purpose is paving the construction of a sophisticated digital twin and AI-scheduler by providing a training and testing ground with different scenarios and generated or accumulated training data. This way different operational conditions can be build ranging from network or hardware restrictions. The last chapter concludes the deliverable by giving insights into what kind of metrics are necessary for a successful operation of the framework and thus need to be collected. The metric section gives a concise overview of how data is being collected, processed and stored using Prometheus as the backbone monitoring system for the DECICE framework.

A last chapter deals with considerations that have been discussed during the elicitation of this deliverable and could lead to potential performance improvements of some components of the DECICE framework but are outside of the scope of the project.

7.1 Overview of DECICE Architecture

The following chapter deals with the structure, description and operation of the DECICE architecture. Figure 1 depicts an initial overview of this architecture. The framework is divided into two main components: The DECICE framework itself, which is responsible for preparing and arranging computational jobs, workloads and services, and the compute plane, which is responsible for the actual execution of work and running services.

The subsequent color legend shows the interaction of the individual components in the framework and illustrates the flow of data and information within the architecture. Any incoming arrows represent that a component is listening for requests, while outgoing arrows represent active requests from a component to another. Therefore, components that do not have any outgoing arrows are purely passive but may respond to requests, while components that do not have any incoming arrows, do not expose any interfaces.

The following Section 7.1.1 presents the DECICE framework in more detail, while Section 7.1.2 is devoted to the compute plane and its description and functionality. Finally, in Section 7.1.3, an example is given to characterize how the scheduling process works using the DECICE framework.

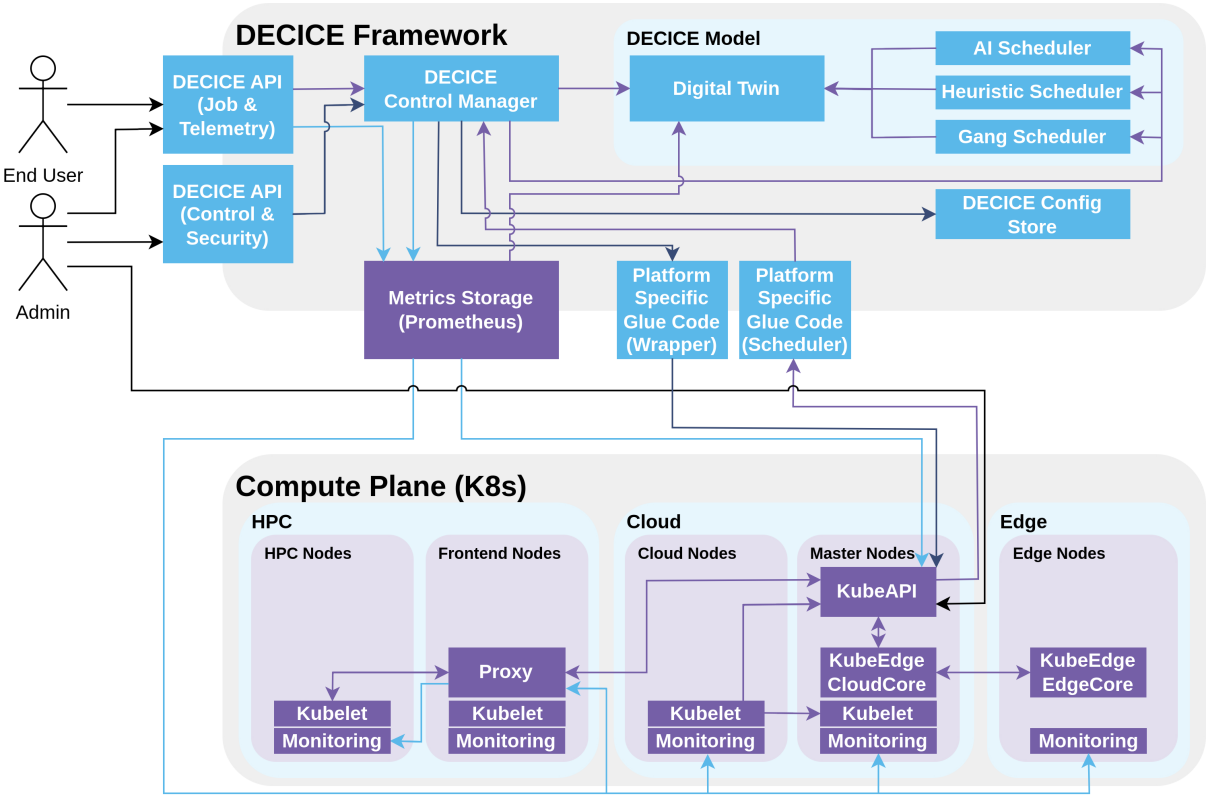


Figure 1: High-level DECICE Architecture

Color Legend

- Light Blue boxes: Components that are developed through the DECICE project
- Purple boxes: Components that already exist in some form and need to be deployed and configured through the DECICE project
- Light Blue arrows: Part of system monitoring
- Purple arrows: Part of the scheduling and other automatic control loops
- Dark Blue arrows: Part of system administration invoked automatically or by admins
- Black arrows: Users and admins interacting with the system

7.1.1 DECICE Framework

The DECICE framework favors a component based composition for its architectural approach, which enables switching between different implementations of the same component. Therefore, the platform can be specified and implemented as a set of interfaces that reflect the organizational structure of the project. Additionally, components interacting with external interfaces can be swapped out or updated if the external interfaces change without affecting the internal components.

Whereas the compute plane consists of already existing software and libraries, the DECICE framework does not come with pre-built implementations. Thus a development from the ground up is necessary and consequently involves requirements engineering that reflects all conditions and use-cases the framework needs to fulfill. Even though DECICE has three defined and worked out use-cases for the validation of the framework (cf. D5.1) it is still missing an elicitation and specification of requirements for the software architecture itself that is part of a future deliverable and will be published as soon as they have been defined. For the time being Figure 1 serves as a basis for understanding the minimal set of components that are required for the DECICE framework and what interfaces to other components they need to provide.

Starting with the heart of the framework, the DECICE Control Manager is responsible for dealing with the majority of the business logic and is central to almost all control flows in the framework. In order to store settings, configuration and data relevant for the DECICE framework, the control manager has its own database, the DECICE Config Store.

Another component the control manager might send requests to or receive from is the Platform Specific Glue Code for the wrapper as well as the scheduler. These interfaces serve as an integration of the control manager with the underlying platform. Here the compute plane is managed by Kubernetes but by swapping them out, they could also support another orchestration platform such as Hashicorp Nomad or Apache Mesos. The wrapper glue code serves as an abstraction for controlling the KubeAPI without having to internally adhere to its expected syntax and having the wrapper glue code translate requests from the internal representation to the format expected by the KubeAPI. The scheduler glue code serves as an entry point for the underlying platform to request scheduling decisions from the DECICE framework.

The DECICE API exposes two components for interactions with the underlying platform, the end user API (job and telemetry) and the admin API (control and security). The end user API provides interfaces to submit jobs, deployments and workloads. These submissions are processed by the

control manager and then stored to be considered during future scheduling decisions. Furthermore, it can also be used to send queries to the metrics storage, while considering user permission to determine what queries are allowed.

The admin API is capable of sending and receiving requests from the underlying platform admin interface for further configuration and management. Admins may also bypass the DECICE API entirely and work directly with the lower components if necessary.

The Metrics Storage utilizes endpoints to aggregate monitoring data from the system and regularly pushes aggregated data to the Digital Twin (DT) and also queries the KubeAPI for additional monitoring data. At its core this component will make use of Prometheus, a monitoring application that records metrics in a time series database.

The DECICE model consists of the Digital Twin and potentially one or more scheduler options. The DT contains aggregated metrics and system settings that are relevant for scheduling and job queues. It is updated by the metrics storage and the control manager and queried by the scheduler to get data for scheduling decisions. The DECICE scheduler is the key component for scheduling decisions and job placement and it queries the digital twin to collect data for a sophisticated scheduling decision and sends its results back to the control manager. It is capable of scheduling any number of pods/jobs/workloads. For testing purposes or different use case scenarios additional schedulers may be provided that utilize different scheduling methods (gang/batch scheduling, heuristic scheduling, AI scheduling).

7.1.2 Compute Plane

The compute plane in a continuum compute platform is used to run user workloads across heterogeneous resources, including Cloud, High-Performance Computing (HPC), and Edge environments. The compute plane represents a resource orchestration platform that is able to run containerized jobs across a number of physical systems. In our case we consider Kubernetes as the resource orchestrator and expand it with KubeEdge to be able to manage autonomous edge devices. The compute plane utilizes Prometheus to gather metrics from devices as well as through Kubernetes/KubeEdge interfaces.

The compute plane architecture may encompass various types of nodes within the compute continuum, however, all nodes of the continuum have a monitoring stack that captures node-local data and makes it available for metrics storage via Prometheus exporters (see Section 7.4.1). The Kubelet component of Kubernetes is also deployed on all nodes, except for the Edge Nodes, which use EdgeCore (a KubeEdge component) instead.

The Cloud acts as the central component, connecting HPC and Edge environments. Cloud Master Nodes run Kubernetes Control Plane with KubeAPI and etcd. The KubeAPI receives requests from the DECICE Control Manager and sends scheduling requests to Platform-Specific Glue Code (Wrapper), which is handled by the DECICE framework.

KubeEdge CloudCore exposes an endpoint that is reachable from the Edge Nodes. Edge Nodes do not have to expose an endpoint to be able to communicate with the main cluster, instead they regularly send requests to the CloudCore endpoint to receive new tasks. KubeEdge EdgeCore is

capable of autonomous operation of edge devices, even during disconnections from the Cloud, while CloudCore ensures that the main cluster does not try to reschedule workloads running on an edge, which has temporarily been disconnected.

In the Cloud, there are Cloud Worker Nodes that are part of the cloud infrastructure. In the HPC environment, a Kubernetes stack is run on each node, and HPC Nodes are potentially equipped with accelerators such as GPUs. In many HPC systems, the nodes do not have direct internet access and rely on a proxy on the frontend nodes to access the internet in order to reduce the attack surface and potential for malicious usage. This is considered in the design as care has to be taken to enable the master nodes to use the proxy to communicate with the Kublets running on the HPC nodes. Furthermore, the proxy is also used for communicating monitoring data and for nodes on the HPC system to download data such as container images from the internet.

During later stages of the project, an integration with an HPC batch schedulers, such as Slurm, will be further explored. Such an integration would require a bridge that is able to express Slurm resources such that Kubernetes can work with them. This would potentially allow a dual usage of HPC resources both as part of a compute continuum managed by the DECICE framework and directly via the Slurm user interfaces.

The DECICE framework itself with its components also needs to run somewhere. Theoretically, the DECICE framework could also run on the compute plane one the Master Nodes next to the Kubernetes Control Plane. However, in order to provide better recovery from Master Node failure in a production setup, the DECICE framework should run on extra nodes that a separate from the compute plane.

7.1.3 Scheduling Workflow

1. The DECICE Control Manager updates the cluster via the platform-specific glue code wrapper to set up new pods in response to a new requests coming into the system or rescheduling
2. Kubernetes notices that pods are pending that have no node assigned
3. Kubernetes requests a scheduling decision
4. The request is forwarded to the custom endpoint exposed by the platform-specific glue code
5. The glue code translates the request into DECICE's internal representation and forwards it to the DECICE Control Manager
6. The DECICE Control Manager considers relevant configurations from the DECICE Config Store
7. The DECICE Control Manager triggers a refresh of the metrics for the Digital Twin from the Metrics Storage
8. The DECICE Control Manager forwards the scheduling request to a scheduler installed in the DECICE Model
9. The DECICE Scheduler pulls in the current system state from the Digital Twin

10. The scheduler considers the current system state and the queue of unscheduled jobs/pods and then generates a scheduling decisions
11. The scheduler sends the scheduling decision back to the DECICE Control Manager
12. The DECICE Control Manager updates the Digital Twin accordingly and sends the scheduling decision as response back to the platform-specific glue code
13. The glue code translates the response to a valid responds for the scheduling request and responds to the original scheduling request
14. Kubernetes persists the response and acts on it

7.2 Overview of Scheduling and Optimization Algorithms

The second major objective of the DECICE project besides connecting Cloud, Edge and HPC into a compute continuum is to establish optimized AI-driven scheduling systems that are capable of finding near-optimal scheduling decisions for workloads across the compute continuum. As the underlying platform for the prototype of the DECICE system, Kubernetes, and all the considered alternatives (Apache Mesos, Hashicorp Nomad) are container orchestration platforms, the target for scheduling decisions are containers. More precisely, pods, in the case of Kubernetes, which are single containers or groups of containers that operate closely together. For example, one container might provide a service and another container in the same pod might provide a proxy with queuing and rate-limiting for that service. Kubernetes was chosen as it has become the de facto standard in terms of container orchestration platforms [Car22] and is part of the graduated projects of the Cloud Native Computing Foundation.

The task of a scheduler is to find a placement for a pod across all nodes that are part of a cluster. A pod commonly has resource requirements in terms of memory and CPU time that should be available on the target node. Furthermore, a pod might require more specific resources such as access to a GPU or other special resources exposed to Kubernetes. Depending on the required resources and the state of the cluster, a scheduling decision might not always be possible when no node is able to fulfill the resource requirements. Then the pod has to wait for the cluster state to change such that its required resources become available before it can be scheduled.

In addition to limitations in terms of availability, further scheduling restrictions might apply, such as pods only being able to be scheduled on certain nodes or not on the same node as other pods. The first case could be used to ensure that in a cluster with worker nodes with and without GPUs, workloads that do not require a GPU, are not scheduled on the GPU nodes. The second case could be used to ensure better availability of a service by distributing multiple instances of the service across multiple nodes such that a node failure only takes down a single instance of the service and not potentially all instances on a single node causing a service disruption or a complete failure.

When multiple pods await scheduling, the scheduler should consider the priority of the pods. Some pods might provide cluster critical services, which are necessary for other pods to access resources or for the cluster to function at all. The scheduler should place these pods first and it might even be desirable to displace non-critical pods in favor of cluster critical pods if necessary.

After applying all the points listed above to not consider nodes that do not have the required resources or that cannot be used due to scheduling restrictions, the first phase of common scheduling approaches, *Filtering*, is completed. Now the scheduler can consider on which of the remaining possible nodes the pod should be scheduled. In general, this process is called *Scoring* and involves considering information about the nodes such as available memory and CPU time or the number of pods already running on a node. Finally, the node with the highest score becomes the new home for the scheduled pod.

Various techniques and algorithms exist that may be used to score a given node. Overall, according to Ahmad et al. [Ahm+22] scheduling techniques can be classified into the following four different categories:

1. Mathematical modeling: In this category, a scheduling problem is analysed as a set of constrained equations and solved using standard techniques such as Linear Programming or Integer Linear Programming (ILP). Finding the optimal solution via ILP is considered an NP-complete problem and, therefore, significantly limits the applicability of ILP to large clusters. ILP techniques are discussed in greater detail in Section 7.2.1.
2. Heuristics: These are low complexity techniques, which generate scheduling solutions in a reasonable time frame. They use a simplified set of rules to score nodes, which are not guaranteed to deliver optimal results but are due to their simplicity and fast computation good enough to be the default scheduler of Kubernetes. Well known heuristic strategies include: first-fit, first-in-first-out, and Dominant Resource Fairness (DRF) [Car22].
3. Meta-heuristics: Are advanced approaches or heuristic methods developed to discover, create, adjust, or choose a heuristic (a partial search algorithm) that can offer an acceptably effective solution for optimization or machine learning challenges. This is particularly useful when dealing with situations involving incomplete or imperfect data, or when computational resources are restricted. Examples include genetic, particle swarm and ant colony optimisation. Nevertheless, the increased complexity of theses approaches in many cases limits their applicability to large clusters due to high computational cost.
4. Machine Learning: ML techniques deal with training models based on available data and then inferring scheduling decisions from said models. Deep neural networks are a promising method, however, depending on the design and optimizations applied, this approach has the potential to be computed in a reasonable amount of time or become as costly as ILP.

The quality of the decisions a scheduler can make in terms of optimizing resource utilization, stability and performance depend not only on the technique used to reach the decision but also on the data fed into the scheduler. Common factors such as memory usage, CPU time reservation and number of pods already running on a node are used. Depending on the nature of the cluster, this might be insufficient to reflect all potential angles for optimization, especially if the nodes of the cluster are heterogeneous and excel at different kinds of tasks. For the DECICE project this is one of the central reasons for creating a custom scheduler as Kubernetes by default expects all nodes to be very similar and well connected but on a compute continuum, including Cloud, HPC and Edge nodes, this does not hold true. See Section 7.4 for an in-depth discussion of how and what metrics are to

be collected in the DECICE project.

Finally, it should also be noted that scheduling by considering one pod at a time is not possible in some cases. Some use cases require batches of pods to be scheduled at once when a workload is only able to start operating when a certain number of pods is running at the same time. Should two such workloads come into the cluster at the same time, a classical scheduler might end up giving 50% of the available resources to each of the two workloads but both workloads might require at least 70% of the available resources to start processing. With both workloads waiting to be allocated the remaining requested resources, they cannot start processing and the cluster is effectively in a dead lock. The solution to this is to only start scheduling such a workload if a certain amount resources are actually available and then to schedule them all at once. This is called gang-scheduling.

7.2.1 Linear Programming (LP)

Linear programming is a mathematical modelling technique in which a linear cost function is minimised (or maximised) when subjected to various constraints [NAG23]. Mathematically, it can be expressed as follows:

$$\min \ \{F(x)\} = \sum_{j=1}^{n} c_j x_j, \tag{1}$$

where the total cost function $F(x)$ associated with n continuous decision variables x are be minimised subject to any linear constraints of the following form,

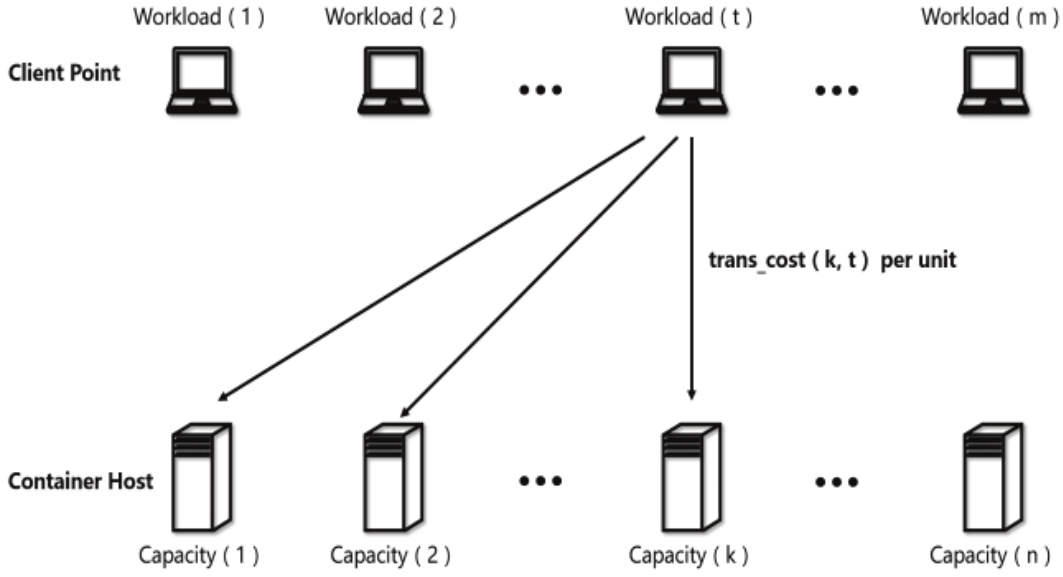
$$\begin{aligned}
\sum_{j=1}^{n} a_{ij} x_j &= b_i, && i = 1, 2, .., m_1 && \text{(equality)} \\
\sum_{j=1}^{n} a_{ij} x_j &\leq b_i, && i = m_1 + 1, .., m_2 && \text{(inequality)} \\
\sum_{j=1}^{n} a_{ij} x_j &\geq b_i, && i = m_2 + 1, .., m && \text{(inequality)} \\
x_j &\geq l_j, && j = 1, 2, ..., n && \text{(simple bound)} \\
x_j &\leq u_j, && j = 1, 2, ..., n && \text{(simple bound)}
\end{aligned} \tag{2}$$

An extension of LP, Integer Linear Programming (ILP) is a standard technique which could be employed for optimising different performance metrics while designing a container based scheduler. Apart from satisfying equation 2, in ILP, the decision variables x are further restricted to take only integer values. The scheduling problem is now recasted as a multi-objective optimisation problem (MOOP).

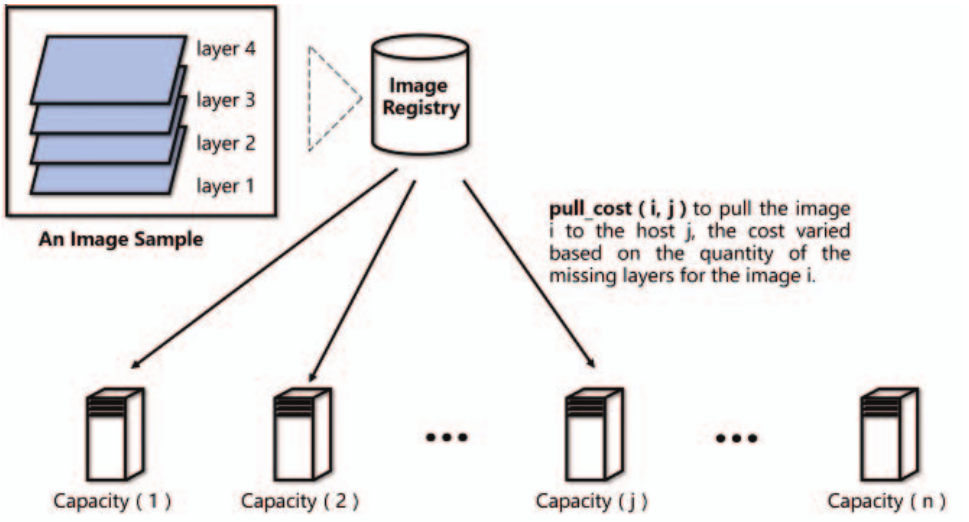
Then equation 1 could be rewritten as

$$\min \ \{F(x)\} = \sum_{j=1}^{n} c_j f_j(x_i); \quad (i = 1, 2, .., m) \tag{3}$$

Thus the total cost function $F(x)$ is minimised by optimising individual objective functions $f_j(x_i)$ which in turn, depends on m decision variables x_i.



(a)



(b)

Figure 2: (2a) Typical scenario of containerized application and (2b) Image pulling registry. Adapted from [Zha+17]

To understand ILP application in the scheduling problem, let us consider a typical container deployment scenario discussed by Zhang et al. [Zha+17] and illustrated in Figure 2 [Zha+17]. Suppose there are m client points such as networked desktops or Internet of Things (IoT) sensors. Each client requests a job of a particular workload. A t^{th} client requests a workload of $\mathrm{Workload}(t)$ units. This job is scheduled in a data centre composed of n container hosts. The k^{th} host has computing capacity of $\mathrm{Capacity}(k)$ units (Figure 2a). There exists a network transition costs between the

client and host due to latency, bandwidth *etc.* Let the cost between k-th host and t-th client be $\mathrm{cost}(k,t)$ units. To launch a container j, each host must pull an image i from the registry resulting in $\mathrm{pull\_cost}(i,j)$. A container image is made up of several layers of file. So, $\mathrm{pull\_cost}$ is zero if all the required layers are already present on the host. However, if some or all the layers of the file are missing, then there exist an extra cost illustrated in Figure 2b.

Zhang et al. [Zha+17] further discuss that when in a given production environment, the t-th client creates a task request with $\mathrm{Workload}(t)$, the workloads are split into batches to a number of container hosts. After pulling the container images from the image registry, if not present, the container hosts launch containers to execute the workloads.

The key issue in the whole workflow is the design of a scheduler to dispatch the workloads to container hosts, which is able to meet the following criteria:

1. All the workloads from clients are containerized and processed in the hosts.
2. The workloads scheduled to a specific host cannot exceed the host's computing capacity.

The goal is to optimize the energy consumption by the hosts, and the cost associated with pulling an image from container registry and network transition of the workload. Due to constraints 1 and 2, we get the following two equations:

$$\sum_{k=1}^{n} x(k,t) = \mathrm{Workload}(t), \tag{4}$$

where natural number $x(k,t)$ is the workload scheduled on the k-th host by t-th client.

$$\sum_{t=1}^{m} x(k,t) \leq \mathrm{Capacity}(k). \tag{5}$$

Now, the cost of workload transition over the network could be defined as

$$f_1(x) = \sum_{t=1}^{m}\sum_{k=1}^{n} \mathrm{trans\_cost}(k,t)\ x(k,t) \tag{6}$$

The energy cost due to power consumption is defined as

$$f_2(x) = \sum_{k=1}^{n} \left[(\mathrm{P_{k,max}} - \mathrm{P_{k,idle}}) \times u + \mathrm{P_{k,idle}}\right], \tag{7}$$

where $\mathrm{P_{k,idle}}$ and $\mathrm{P_{k,max}}$ are the average power values for the k-th host, when the system is, respectively, idle and fully utilised. The utilisation rate u in eq. 7 is defined as

$$u = \frac{\sum_{t=1}^{m} x(k,t)}{\mathrm{Capacity}(k)} \tag{8}$$

Similarly, the cost of pulling the container image is given by,

$$f_3(x) = \sum_{t=1}^{m} \sum_{k=1}^{n} \text{pull\_cost}(k,t)\ y(k,t), \tag{9}$$

where,

$$\text{binary variable } y(k,t) = \begin{cases} 1, & \text{if the node } k \text{ pulls the image } t \text{ from the image registry} \\ 0, & \text{otherwise.} \end{cases} \tag{10}$$

If $y(k,t) = 0$ then $x(k,t) = 0$. Taking into account equations 6, 7 and 9, the total cost function (eq. 3) can be rewritten as a Multi-Objective Optimisation Problem (MOOP) in terms of individual cost functions as,

$$F(x) = c_1 f_1(x) + c_2 f_2(x) + c_3 f_3(x), \tag{11}$$

Here the weighting coefficients c_i are introduced to balance the three costs.

Kaur et al. recasted the scheduling tasks of Industrial IoT (IIoT) devices as a MOOP problem. The solution was obtained within ILP formalism where objective functions which represented (i) carbon emission footprints, (ii) sources of interference, and (iii) energy utilisation, were minimised [Kau+20]. Recently, ILP technique was applied to service operational cost minimisation problem when the problem size was small. For larger sizes, ILP constraint was relaxed to a Linear Programming (LP) problem [Li+23].

7.2.2 AI-Scheduler

An AI-scheduler utilizes some form of machine learning in order to develop a model that can then infer scheduling decisions given the state of the cluster. Ahmad et al. [Ahm+22] describes the capability of an ML-based scheduler compared to alternative approaches in terms of potentially better scalability while also reaching better scheduling decisions compared to other scalable scheduling algorithms such as heuristics.

In order to train a capable model, large amounts of training data is required [Sen+23], which is rarely available. Therefore, many other researchers had to rely on synthetic or simulated data to train their AI-schedulers. Depending on the optimization target for the scheduler as well as the nature of the cluster that it will operate on, different data might be required. For example, in order to find near-optimal scheduling decisions across a compute continuum including cloud, edge and HPC, the scheduler would need to be aware of the capabilities of the heterogeneous nodes as well as the network connectivity between them. This means that this data also needs to be available in the training data.

Within machine learning there are multiple approaches [Car22] such as supervised learning, deep learning and reinforcement learning. Deep learning could serve to find complex patterns in the data and use these for improving the quality of scheduling decisions. Reinforcement learning on the other hand would enable the scheduler to continue to learn from experience.

At this point the final strategy for the DECICE scheduler is still to be determined via experimentation

and comparison of these approaches. As the DECICE framework is designed to be able to switch between schedulers at run time, multiple variations for the AI-scheduler can be tested in the same setup. As a baseline for comparison serves the default heuristic scheduler and a naive FIFO scheduler.

Furthermore, [Sen+23] explains that there are multiple options for extending the default Kubernetes scheduler by adding policies, plugins or by replacing it entirely. As by the design of the DECICE framework, the default scheduler would be replaced entirely. Nevertheless, the two phase process of filtering and scoring is still kept intact such that the ML model has the task of performing scheduling decisions for a given system state including the queue of pending pods after filtering per pod has been applied.

As input for inference and as training data serves the Digital Twin with the information on the system it has stored including running jobs, pending jobs, hardware statics, hardware characteristics and network. By utilizing this many metrics, it should enable the AI-scheduler to become context aware as described by [Car22] and reach scheduling decisions that also consider stability and robustness.

7.3 DECICE Architecture Components

7.3.1 Dynamic Digital Twin

This section documents the requirements for the Dynamic Digital Twin. There are various definitions of Digital Twin (DT) used in industry and academia [Jon+20]. In the context of the DECICE project, we investigated the state of the art of digital twins and have arrived at a definition for a digital twin, which will be discussed in the following.

A Digital Twin is an advanced model that represents the current and historical status of a system. A system managed by the DECICE framework may span across a compute continuum, including Cloud, HPC, Edge, and IoT devices. A Digital Twin is a digital representation of an intended or actual real-world physical product, system, or process (a physical/real twin/entity) that serves as the digital counterpart of it for practical purposes, such as simulation, testing, monitoring, forecasting, and maintenance. With respect to DECICE innovation on the management of the computing continuum, the digital representation, which is the core of the Digital Twin, can be expanded with additional modules that support visualization, feature extractions, forecasting, and simulation. In this definition of a Digital Twin, there are two main components: (1) the core Digital Twin, and (2) the Digital Twin modules.

A functional Digital Twin may include baseline data about the infrastructure and live data collected from sensors, as well as automatically collected and updated system characteristics. It can be used for facilitating analysis and decision-making processes as well as data augmentation and training of data-driven computing continuum policies/optimizations (like AI scheduler and Control Manager).

The remainder of the section is structured as follows: (I) Digital Twin key definitions in Software-oriented-Architecture (SoA) and DECICE, (II) DECICE digital twin requirements.

(I) Digital Twin key definitions A survey by Jones et al. [Jon+20] provides a systematic literature review and thematic analysis of 92 digital twin publications, characterizing digital twins and identifying gaps. The authors of the paper provide a list of themes and their associated descriptions,

which we have extended better fit the DECICE context. As DECICE targets the computing systems (SW and HW), we have replaced the term "Physical" with "Real" and we will use "Real" within the deliverable but this can be translated to "Physical" in management and manufacturing contexts.

In the DECICE project, we have different types of software and tools. Parts of these software represent real entities, while others only exist as virtual entities, which gives a view and definition for a DECICE Digital Twin compared to a SoA Digital Twin. For example, the scheduler is a "real" SW component in DECICE and, in general, in computing systems, but it is considered as a "virtual" entity, which provides optimizations in Digital Twins of manufacturing processes. Therefore, in Table 3, we provide clear examples to define the themes that are known in the SoA of Digital Twin and map the components and themes to their counterparts in DECICE.

Real-to-virtual and Virtual-to-real Twinning Processes According to the initial definition, a digital twin is a virtual representation of a real entity that contains information about the real world. The paper [Gri14] extends this definition by adding that a digital twin consists of three components: a real entity, a virtual representation of the real entity, and bi-directional data connections that feed data from the real to the virtual representation and vice versa. According to [Gri14], data and information flow in both directions in a cycle between the real and virtual states, which is known as mirroring or twinning. The virtual environment is composed of sub-spaces that enable specific virtual operations, such as modeling, testing, and optimization. As already mentioned, we adopted the terminology of "Real" instead of "Physical". Figure 3 shows the real-to-virtual and virtual-to-real twinning processes.

The real entity's state is transferred to the virtual environment through real-to-virtual connections. This process involves a Metrology phase, in which the real entity's state is captured, and a Realisation phase, in which the virtual entity is updated to match the real entity's state. The virtual-to-real connection allows the Digital Twin to physically change the state of the real entity. This involves both metrology and realization phases, where virtual processes and metrology methods determine (optimal) parameter values, and realization methods update the state of the real entity accordingly.

In DECICE the virtual-to-real connection is mediated by two additional components that are external to the DECICE Digital Twin, namely the AI scheduler and Control Manager. These two components closely interact with the DECICE digital twin by processing on-demand the digital representation of the computing continuum contained in the digital twin. The actual realization of changes to the real system occurs through job scheduling based on scheduling and re-scheduling requests. The Digital Twin in the DECICE framework is a passive component that is kept in sync with the real system and queried by the DECICE schedulers when making scheduling decisions.

Twinning is the act of synchronizing the real and virtual states of a digital twin. When a change occurs in either the real or virtual entity, it is measured and then realized in the corresponding virtual/real twin. The rate at which twinning occurs is known as the twinning rate, and can be specified in real-time. In DECICE, this rate can be adjusted based on the requirements of the use cases and different entities. Historical data is collected and reused within the virtual environment, allowing the digital twin to learn from its past performance and virtual processes. In the DECICE digital twin, this learning is captured by mean of ML models capable of representing complex relationship and

Table 3: DECICE Digital Twin Themes Descriptions.

Theme	SoA Descriptions from [Jon+20]	DECICE Descriptions
Real Entity	A 'real-world' artifact, e.g., a vehicle, component, product, system, model.	A "real-world" artifact, in DECICE, refers to a computing continuum that spans HPC, Cloud, Edge, and IoT, as well as the dependencies, applications, and software such as AI scheduler that operate within it. All components of DECICE are included except for the Digital Twin components.
Virtual Entity	A computer generated representation of the physical artefact, e.g., a vehicle, component, product, system, model.	This is a computer-generated representation of a physical or real entity. It includes components that are present in the digital twin and digital twin modules. System structure description that includes the available components and their compositions, as well as the current and historical status of the system. Behavioral model and forecasting/evaluation, virtual training environment, visualization/GUI, feature extraction, This part may also contain various tools that utilize different techniques, such as statistical and machine learning methods, to generate new features or information, such as anomaly detection and prediction and power consumption prediction tools.
Real Environment	The measurable 'real-world' environment within which the physical entity exists.	This refers to the measurable "real-world" environment in which a real entity, such as the HPC, Cloud, Edge, or software such as a job scheduler, exists.
Virtual Environment	Any number of virtual 'worlds' or simulations that replicate the state of the physical environment and designed for specific use-case(s), e.g., heath monitoring, production schedule optimisation.	A digital twin is a virtual representation of a real environment. The digital twin is located within a virtual environment.
Fidelity/Levels of Fidelity	The number of parameters transferred between the physical and virtual entities, their accuracy, and their level of abstraction. Examples found in literature include: fully comprehensive, ultra-realistic, high-fidelity, data from multiple sources, micro-atomic level to the macro-geometrical level.	The number of parameters transferred between the real and virtual entities, their accuracy, and their level of abstraction. These may vary based on the significance of the individual parameter and the added overhead on the system for keeping a given parameter in sync between the system and the Digital Twin.
State	The current value of all parameters of either the physical or virtual entity/environment.	The current value of all parameters of either the real or virtual entity/environment.
Parameters	The types of data, information, and processes transferred between entities, e.g., temperature, production scores, processes.	Different types of data, information, and processes transferred between entities, e.g., CPU/GPU/Memory/Network usage, Job and Queue parameters, job real power consumption, job predicted power consumption. In the context of DECICE the term "metric" is mostly used for parameters collected from different real entities.
Real-to-Virtual Connection	The connection from the physical to the virtual environment. Comprises of physical metrology and virtual realisation stages.	The connection from the real to the virtual environment. Comprises of physical metrology and virtual realization stages.
Virtual-to-Real Connection	The connection from the virtual to the physical environment. Comprises of virtual metrology and physical realisation stages.	The connection from the virtual to the real environment. Comprises of virtual metrology and physical realization stages.
Twinning and Twinning Rate	The act of synchronisation between the two entities and the rate with which synchronisation occurs.	This refers to the synchronization rate between two entities. In DECICE, the twinning rate of different metrics (entities) can vary depending on the use case and type of parameter.
Real Processes	The physical purposes and process within which the physical entity engages, e.g., a manufacturing production line.	The real purposes and process within which the real entity engages, e.g., a job scheduling.
Virtual Processes	The computational techniques employed within the virtual-world, e.g., optimisation, prediction, simulation, analysis, integrated multi-physics, multi-scale, probabilistic simulation.	The computational techniques employed within the virtual-world, e.g., visualization, optimization, prediction, simulation, analysis, and feature extraction.
Perceived Benefits	The envisaged advantages achieved in realising the Digital Twin, e.g., improved design, behaviour, structure, manufacturability, conformance, etc.	The envisaged advantages achieved in realizing the Digital Twin, e.g., improved job scheduling, anomaly detection and reduced system outage time, etc.
Digital Twin across the Product Life-Cycle	The life-Cycle of the Digital Twin – (whole life cycle, evolving digital profile, historical data)	The life-Cycle of the Digital Twin – (whole life cycle, evolving digital profile, historical data)
Use-Cases	The applications of the Digital Twin, e.g., reducing cost, improving service, supporting decision making.	The applications of the Digital Twin, e.g., reducing computing cost, improving service, and supporting AI schedulers.
Technical Implementations	The technology used in realising the Digital Twin, e.g., Internet-of-Things.	The technology used in realizing the Digital Twin, e.g., Internet-of-Things, APIs, MQTT protocol, IPMI, etc.
Data Ownership	The legal ownership of the data stored within the Digital Twin	The legal ownership of the data stored within the Digital Twin.
Integration between Virtual Entities	The methods required to enable communication between different virtual entities.	The methods required to enable communication between different virtual entities.

perform feature extraction of not directly observable key performance indicator (energy/performance efficiency, reliability, anomalies) and forcasting.

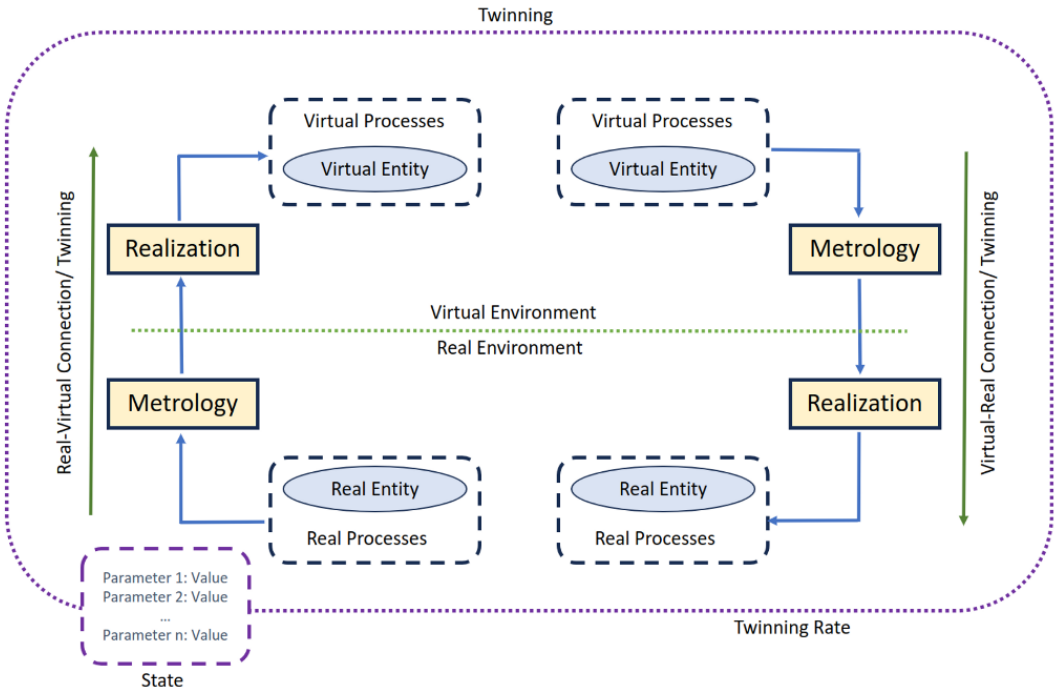


Figure 3: Real-to-virtual and Virtual-to-real Twinning Processes.

(II) DECICE Digital Twin Requirements. Figure 4 shows a block diagram representing the DECICE Digital Twin component. Starting top-down the digital twin has to interface with different parts of the DECICE framework and computing continuum. It receives sensor information from different real HW and SW platforms across the computing continuum and DECICE framework, ranging from HPC to edge. The services provided by the Digital Twin are split into two main parts - the Digital Twin Core and the Digital Twin modules.

Digital Twin Core The digital twin core consists of (i) the system structure description that includes the available components and their compositions, as well as the (ii) current and historical status of the system. The digital twin core receives information from various sensors and APIs located in different parts of the physical/real twin or physical/real entity. It also provides the necessary information to other parts of the DECICE environment, such as the AI scheduler and Control Manager. Therefore, the digital twin core is primarily responsible for receiving the state of real entities as input and providing raw data and processed information to the other parts of the DECICE environment.

Digital Twin Modules The digital twin modules are a set of modules that extend the capabilities of the digital twin. They consume the raw data provided by the digital twin core to provide an augmented representation of the real entity capable of simulating the behaviour in different condition, predicting future trends and estimating hidden states of the system (extracting features which are not directly observable).

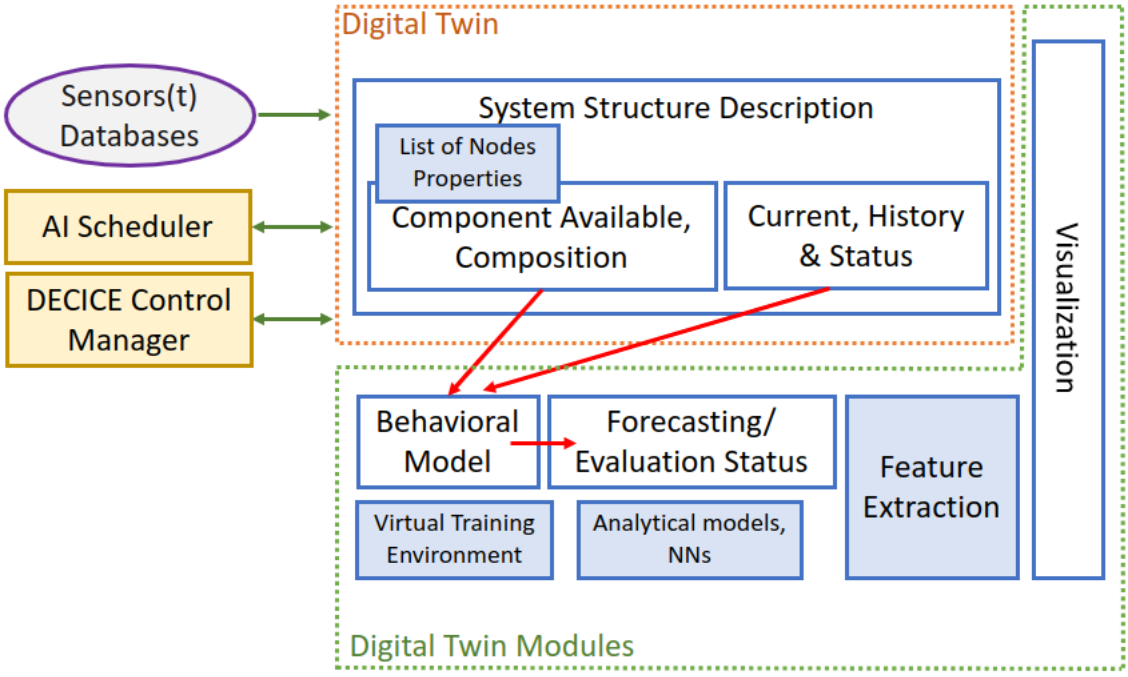


Figure 4: DECICE Digital Twin Component.

The digital twin modules consist of two parts: the Behavioral Model and Forecasting/Evaluation. It creates the Virtual Training Environment that can forecast and evaluate different scenarios (more information in Section 7.3.3), which is useful for conducting what-if analysis.

One of the main functions of this module is to use the current status of the job description and exploit it to forecast future outcomes. Additionally, the digital twin module includes visualization/GUI tools (e.g., Grafana) and feature extraction capabilities for NOT measurable parameters, which we can generate different approaches using human experts, ML tools, and Mathematical tools. This part may also contain various tools that utilize different techniques, such as statistical and machine learning methods, to generate new features or information, such as anomaly detection and prediction, energy and performance efficiency estimation and prediction as well as power consumption prediction tools. Finally, there are also virtual entities, which employ a simulator or trained-model approach to reconstruct a real entity in the digital twin.

Data Models The different tools may generate various data formats and models, but we require an interchangeable format. Different parts of the DECICE framework and computing continuum and those monitoring systems may have varying data formats and models, which can be related to the nature of the data sources, use cases, or the contents of the monitoring system. However, in general, JSON representation is useful for most parts of DECICE framework.

Digital Twin Requirements Imposed By AI Scheduler The scheduler is stateless, which means it relies on the digital twin to understand the current state. It connects to the digital twin in a tightly integrated manner but doesn't make direct changes to it. As a result, the AI scheduler imposes specific requirements, or metrics, on the DT. The monitoring system should provide these metrics and data as input to the DT. To use the AI scheduler effectively, we must first consider the

requirements of our use cases. In general, data management, computation (CPU, GPU, memory, and network), and location awareness are important features that the DT should provide. For hybrid scheduling with Kubernetes and Slurm, the DT should provide additional metrics related to these tools. Also, these metrics can be used to evaluate the performance of the AI scheduler, such as minimizing the number of pending jobs and maximizing the number of idle nodes. For an AI scheduler that handles the placement of data in the compute continuum, the DT system should provide quality storage distribution (historic system state), which should be provided by other parts of the system.

Digital Twin Input Data Requirements In the compute continuum and DECICE framework, there are various real entities. To effectively work with the DT, it should have comprehensive inputs from these entities. and the monitoring system is responsible for providing these inputs to the DT. The detailed DT input requirements are documented in Section 7.3.2, where we documented the monitoring system and important real entities, as well as metrics, APIs, plugins, and monitoring software.

7.3.2 Monitoring System

Monitoring systems refer to the technical implementations that are used to collect and store the information and provide input to the Digital Twin. These implementations include Internet-of-Things, APIs, MQTT protocol, IPMI, and more. So, monitoring (and the metrics that it collects) plays a vital role in ensuring the effectiveness of the Digital Twin. To implement the DECICE DT, we need a holistic monitoring system that fetches updates on the current system state directly from the underlying platform through its interfaces, such as the KubeAPI. The purpose of this document is not to provide complete technical details, as it is impossible to finalize all technical details at this stage of the project. Instead, we aim to give a high-level overview of the requirements and provide examples of technical details and possibilities.

When considering the monitoring system, we need to focus on two key items. The first set of resources and requirements are directly necessary for the monitoring system itself. The second set of requirements is needed by modules that are closely integrated with the monitoring system and DT, such as an AI model that can use historical data. For training purposes, this AI model requires a querying task to extract data from the monitoring system's database. This process leads to additional compute resource utilization in the monitoring system. Additionally, we should consider the HW and SW resources needed for future studies, such as those required for MLOps implementation (a paradigm that aims to deploy and maintain machine learning models in production reliably and efficiently).

The monitoring system comprises three main components: (i) primary data sources from the real environment that the monitoring system must collect important metrics, (ii) APIs/plugins utilized to establish connections, and (iii) software requirements in the DECICE framework, accessible across the computing continuum, which encompasses HPC, Cloud, Edge, and IoT.

Important Real Entities for the Monitoring System In this section, we will provide examples of real entities (tools) that are closely related to the monitoring system. It is important to note that

the monitoring system should be equipped with the capabilities to interact and utilize these tools in order to ensure seamless integration and optimal performance.

Kubernetes Metrics Server: Kubernetes Metrics Server is an open-source component that collects resource utilization data from various components within a Kubernetes cluster. It enables the monitoring and autoscaling of applications running on Kubernetes by providing metrics such as CPU and memory usage for pods, nodes, and containers. It is important to note that Kubernetes Metrics Server is not intended for long-term storage of metrics or complex monitoring and alerting. For more advanced monitoring and analytical capabilities, we consider integrating Metrics Server with external monitoring systems such as Prometheus or using a dedicated monitoring solution.

KubeEdge: KubeEdge is an open-source system for extending native containerized application orchestration capabilities to hosts at Edge. It is built upon Kubernetes and provides fundamental infrastructure support for network, application deployment, and metadata synchronization between cloud and edge.

Slurm (Optional): Slurm (Simple Linux Utility for Resource Management) is an open-source, highly scalable cluster management and job scheduling system. It is primarily used in high-performance computing (HPC) environments to efficiently allocate and manage computing resources across a cluster of machines. Slurm provides a framework for managing and scheduling compute-intensive workloads on a cluster. It allows users to submit jobs to the cluster, which can be a single node or a large-scale cluster with thousands of nodes. Slurm handles the allocation of resources, scheduling of jobs, and monitoring of their execution.

SEDNA (Optional): SEDNA is an open source edge-cloud synergy AI project targeting optimisation and utilization of resources across Cloud-Edge for AI training workloads. Benefiting from the edge-cloud synergy capabilities provided by KubeEdge, SEDNA can implement across edge-cloud collaborative training and collaborative inference capabilities, such as joint inference, incremental learning, federated learning, and lifelong learning. SEDNA supports popular AI frameworks, such as TensorFlow, PyTorch, PaddlePaddle, MindSpore.

Volcano (Optional): Volcano is an open source system built on top of Kubernetes to support high performance batch and elastic workloads. Volcano supports popular computing frameworks such as Spark, TensorFlow, PyTorch, Flink, Argo, MindSpore, and PaddlePaddle on different architectures, such as x86, ARM and GPU. It supports rich scheduling policies and job management features for Machine learning/Deep learning, Bioinformatics/Genomics and other big data applications.

HPC ODA System (Optional): Operational Data Analytics frameworks are a de-facto standard in large-scale HPC installation and consist of a data collection and analysis platform oriented to the management of big data. Their main prerogatives are to manage in a simple way heterogeneous data, both in streaming and batch mode, and to allow the access to these data through a common interface. This simplifies the usability of data supporting applications such as real time anomaly detection, predictive maintenance and efficient resource and energy management using techniques in the domain of machine learning and artificial intelligence. Given the scale of the monitored systems, they have to be scalable and distributed. They are different from Prometheus as they are specialized for collecting data from a scientific computing cluster, which does use batch scheduler and does not

support virtualization nor containerization.

Kubeflow (Optional): Kubeflow is an open-source machine-learning platform built on top of Kubernetes that enables deploying and managing machine learning workflows on Kubernetes clusters. It provides a variety of tools and frameworks to build, deploy, and scale ML models on Kubernetes.

Visualization and User Interface: In the DECICE project, there are various possibilities for GUI/Visualization of the computing continuum. We opt to use Prometheus and Grafana, which are both open-source software. Prometheus is a monitoring system that can be used to collect and store metrics from different sources, and Grafana provides a dashboard for visualizing the data collected by Prometheus. By using these tools, we can have a more comprehensive view of the computing continuum and better understand the different processes and data involved in the project.

Prometheus: Prometheus is an open-source monitoring and alerting toolkit originally developed by SoundCloud. It is widely used in the field of cloud-native and containerized environments, including Kubernetes. Prometheus is designed to monitor and collect metrics from various targets, such as applications, services, and infrastructure components, in a highly scalable and efficient manner. Prometheus is discussed in detail in Section 7.4.1.

Grafana (Optional): Grafana is an open-source data visualization and monitoring platform. 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.

Key features of Grafana include: *Data Visualization:* Grafana provides a rich set of visualization options, including graphs, charts, tables, and gauges. It supports real-time streaming and allows you to create dynamic and interactive dashboards to monitor and analyze data. *Data Source Integration:* Grafana supports integration with numerous data sources, such as Graphite, Prometheus, InfluxDB, Elasticsearch, MySQL, PostgreSQL, and many others. You can connect to these data sources and query data directly from Grafana to build visualizations. *Dashboard Templating:* Grafana allows the creation of dynamic and reusable dashboards by utilizing template variables. These variables can be used to filter data, switch between different data sources, and create more flexible and interactive dashboards. *Alerting and Notifications:* Grafana provides built-in alerting capabilities, allowing you to set up alerts based on specified conditions and thresholds. When an alert is triggered, Grafana can send notifications via various channels such as email, Slack, PagerDuty, and others. *Plugins and Extensibility:* Grafana has a large and active community that develops plugins and extensions to enhance its functionality. One can install plugins to add new visualizations, data sources, authentication methods, and more. *Community and Ecosystem:* Grafana has a vibrant and supportive community that contributes to its development, provides support, and shares dashboards, plugins, and integrations. The Grafana ecosystem is constantly evolving, with a wide range of resources and community-driven projects available.

Grafana is widely used in various domains, including IT operations, DevOps, monitoring and observability, IoT, and industrial applications. It offers a flexible and user-friendly interface for visualizing and analyzing data, making it a popular choice for building monitoring and analytics dashboards.

In the end, we can integrate all of these computing continuum monitoring tools and databases into

Grafana.

H.W. Requirements To meet the HW requirements of the monitoring, we need to consider both the direct requirements of the DECICE project, which show what HW resources we need to implement the monitoring system itself, and also the additional requirements for further R&D studies and investigations that we can conduct using the virtual and real environments provided by DECICE. The virtual environment's lifecycle (which relies on the monitoring system) can even be longer than the real one, and after the real entities are retired or disposed of, parts of the DT and monitoring system can still survive for study purposes.

To store historical data of time series data of monitoring systems for long periods of time, we need TSDB/NoSQL databases like KirosDB/Cassandra, which need storage for storing the data.

To implement most parts of the monitoring system, VMs are required, while Kubernetes resources can be used for parts of the monitoring system and digital twin, such as MLOps. The amount of hardware resources required for the complete implementation of the monitoring system is difficult to define at this stage of the project. However, with VMs and a cloud environment, we can start with minimal requirements and scale up as needed in the future.

There is an ongoing project in Unibo that is working on the hardware and software requirements for the monitoring system and MLOps overhead. This study will reveal the detailed technical requirements for the high-performance computing (HPC) side of the compute continuum of the DECICE project.

7.3.3 Synthetic Test Environment

The virtual training environment (VTE) or synthetic test environment simulates the framework for the DECICE model, which consists of the digital twin and AI-scheduler. By providing artificially generated or curated data accumulated through provided datasets, the VTE is a digital construct that resembles the actual DECICE environment and reflects the compute plane and metric collection without having the need to actually setup a whole compute continuum. This simulated environment allows for a faster, cost efficient and resource-saving training. The VTE constantly provides metrics to the digital twin and AI-scheduler to enable the training process with different test scenarios such as restricted network connections or hardware restrictions. Having a digitized real-world environment enables the capability to to optimize the scheduler for different metrics such as performance, location constraints or energy efficiency. Depending on the use case feature set such as historic information, node architecture or node capabilities can be setup without having the need to actually provide these features via real hardware. Depending on the simulated hardware and environment this can result in vast cost savings during the training process.

The overall training process of the AI scheduler is depicted in Figure 5a. The virtual training environment loads a predefined scenario via a JSON file which is the foundation for the configuration of the digital twin. While the VTE takes care of progressing the training and stepping through the scenarios and providing constant metrics to the DT at the same time, the DT in return adapts to the newly given metrics which can be accessed by the scheduler to perform scheduling decisions [Kun+22].

In contrast to the training cycle Figure 5b shows a real scheduling scenario without a VTE. In this case the scheduling decisions from the AI-scheduler will not be used as a metric for analysis and evaluation of the decisions made by the model, but rather forwarded by the control manager (CM) to the underlying compute plane for a new scheduling task.

Since the prediction quality of the AI-model relies on training data to come up with plausible and valid scheduling decisions, datasets need to be collected, processed and provided to the AI. For this case the DECICE project will rely on several sources of data such as publicly available datasets provided by Alibaba [Ali23] or Google [Goo19] and generated data by data centers from the GWDG.

7.4 Metrics

Since HPC and cloud infrastructures are becoming increasingly complex and powerful with more resources at hand every year, these systems are not only capable of providing more processing power and handling more computationally demanding problems in scientific research and engineering simulations, but also generate a plethora of valuable metrics, which can be collected to optimize the workloads across these systems. HPC and cloud workloads are already known to produce an excessive amount of data when performing large-scale computations [Beh+15]. Collecting these metrics can provide valuable insights into the performance, health and efficiency not only of an HPC cluster but also a full compute continuum such as the DECICE project, which ranges from HPC to cloud to edge devices and works as a heterogeneous compute plane.

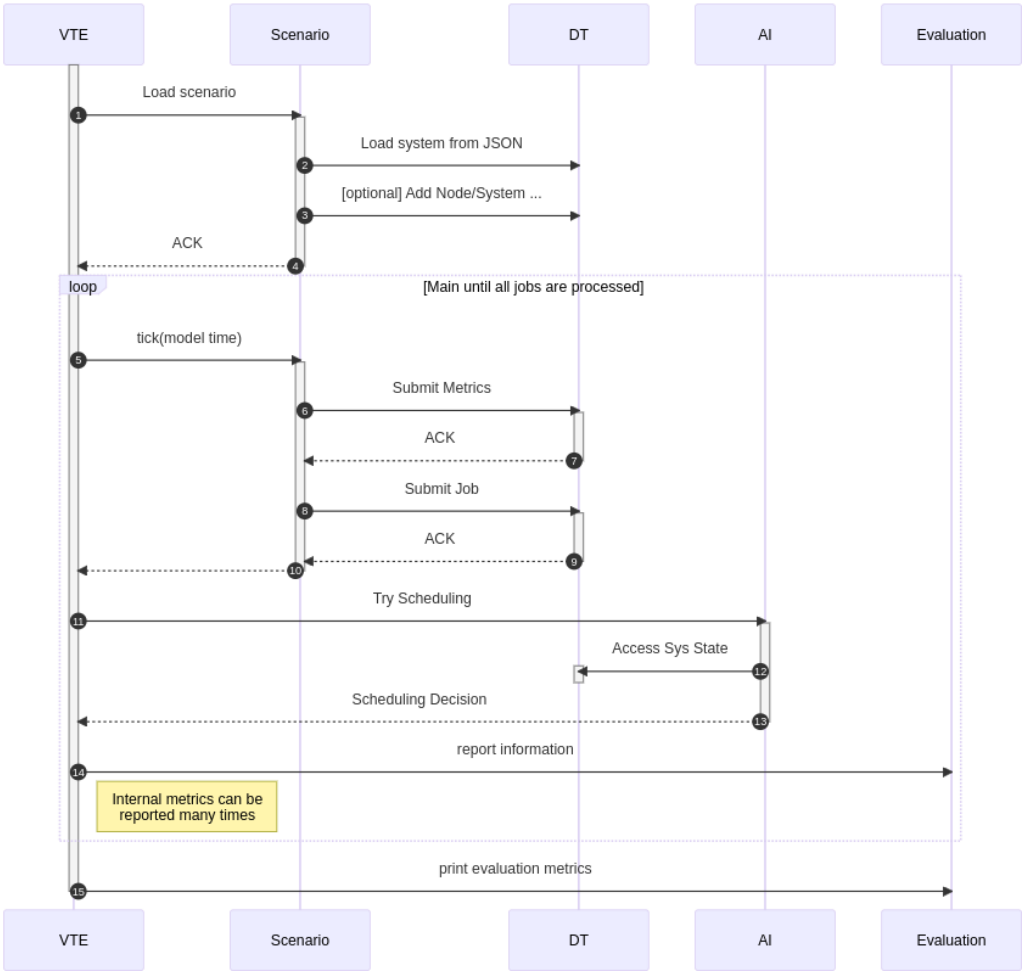
For this task we will rely on Prometheus for the metrics collection. Prometheus is an open source system that allows applications for easy monitoring and alerting capabilities [RV15]. The data generated by the monitoring system will be used to adapt and optimize the AI scheduler and digital twin that in return will enable a comprehensive scenario simulation, accurate bottleneck prediction, and efficient resource allocation optimization across the heterogeneous cluster.

7.4.1 Prometheus

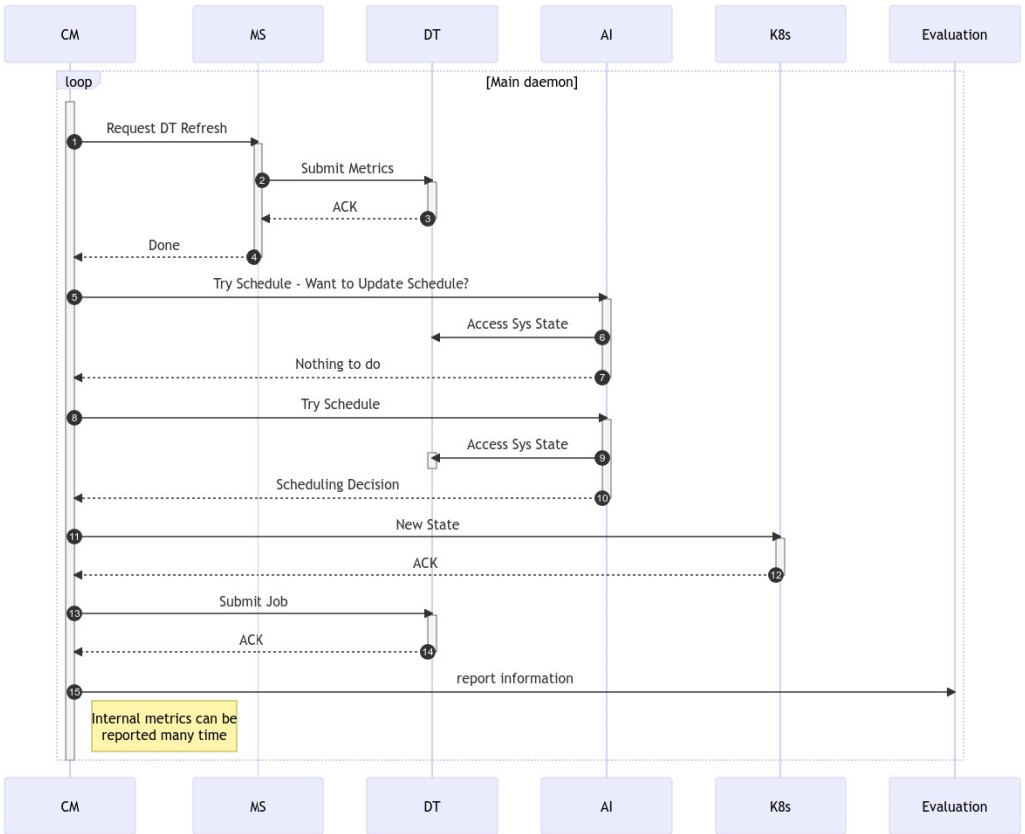
Prometheus utilizes a time series database, is a Cloud Native Computing Foundation (CNCF) graduated project [Fou] and is directly supported by Kubernetes, making it a common choice in the cloud native ecosystem. It has a comprehensive and multi-layered pull-based architecture, which is depicted in Figure 6 with its individual key components are described below.

Data Collection: Prometheus uses a pull-based model for data collection, meaning it periodically scrapes metrics from configured targets over HTTP. These targets can be exporters, which are small processes responsible for exposing specific metrics from applications or systems. A commonly used exporter is the Node Exporter for various server metrics.

Metrics Format: Prometheus stores time-series data in a metric name, a set of key-value pairs called labels, a timestamp, and the actual value.



(a) Sequence Diagram of the Virtual Training Environment



(b) Sequence Diagram of the Scheduler and Digital Twin

Figure 5: Sequence diagrams of the DECICE framework

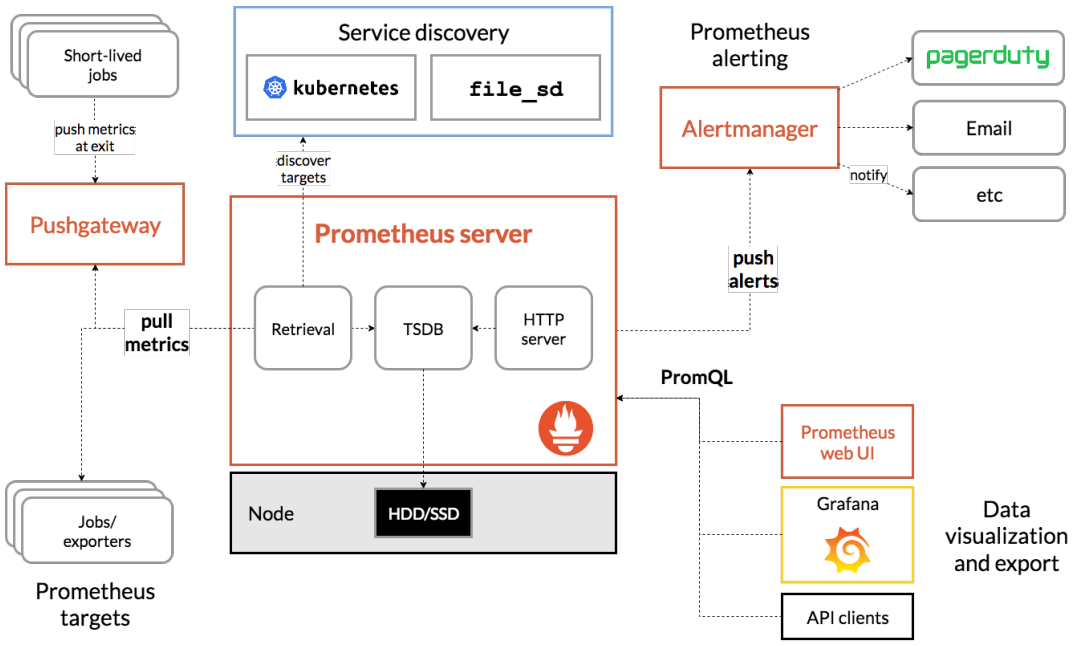


Figure 6: Architecture of Prometheus, adapted from [Pro]

Storage: The scraped metrics are stored locally in a time-series database called the *Prometheus Time-Series Database*. By default, Prometheus uses a local disk-based storage approach.

Querying: Prometheus provides a powerful and flexible query language called *PromQL* (Prometheus Query Language). With PromQL, users can create complex expressions to retrieve and process data.

Alerting: Prometheus comes with an integrated alerting mechanism that allows users to define rules to trigger alerts based on certain conditions and thresholds.

Graphing and Visualization: While Prometheus itself offers a simple expression browser, it is often used in combination with Grafana, an open-source visualization tool.

Service Discovery: Prometheus supports service discovery, which allows it to automatically discover and monitor new instances or targets as they appear in the infrastructure. This can be achieved through various integrations such as Kubernetes service discovery.

Prometheus Agent Mode: As the DECICE project is making use of cloud-native technologies in conjunction with edge devices with a restricted amount of resources, the need for transferring data to remote locations away from where they are actually produced becomes a necessity. Prometheus offers an efficient way of metric forwarding for this use case specifically called *Agent Mode*. This mode is optimized for application scenarios with limited or less powerful resources in which writing is only done remotely (remote write) in order to limit the metrics that have to be stored locally. Furthermore, due to the remote write strategy, the edge node only needs to be able to reach the central Prometheus server and not the other way around. This solves the problem of the edge device being in a different network than the main Prometheus server without forcing every edge device

to expose an endpoint publicly. Instead only the main Prometheus server has to expose a public endpoint that is reachable from the edge networks. The overview is illustrated in Figure 7 below.

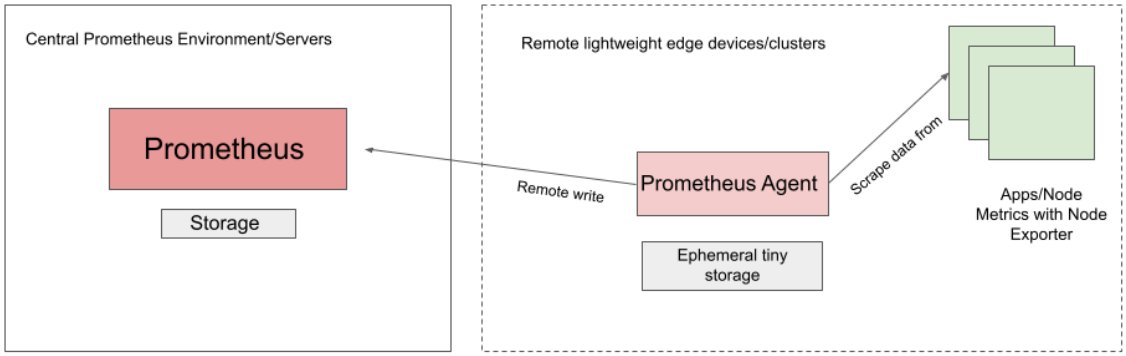


Figure 7: Prometheus Agent mode

The mode does not keep data after writing them to the remote centralized Prometheus server. If an edge device in agent mode cannot reach the Prometheus server, which may happen on edge and IoT devices, it keeps the data on the local disk of the edge device until the remote write gets accessible and data can be pushed again.

7.4.2 Optimization of Metric Collection

Prometheus is able to collect a wide variety of metrics and data about the host system and store it for further purposes such as logging and data analysis. The latter point is particularly interesting, as the stored data serves as the foundation for the training of the AI scheduler and provides the digital twin with constant flow of data. An excerpt of metric collection requirements can be found in Table 4 with a collection of metric names accumulated in Appendix A.

Besides collecting these metrics there is also a risk of putting too much or unnecessary load on servers or edge devices. Collecting metrics can be a resource-intensive task and if too many metrics are collected, it can slow down devices where these metrics are collected from. To prevent a performance decrease or latency increase the following strategies for load reducing can be considered:

- Appropriate Scraping Intervals: Instead of collecting metrics every second, metrics can be collected with larger intervals (e.g., every 15 seconds or 1 minute). For some metrics, a higher scraping interval may be sufficient without sacrificing the accuracy of monitoring.
- Aggregation of Data: Aggregate metrics locally where possible (e.g., sum, average, max) before collecting them to reduce the overall data volume.
- Using Compression: Enable compression for data transfer between the servers and the monitoring system to reduce network usage.
- Using Push-Based Metrics: Where applicable, use push-based metric collection systems (e.g., Pushgateway in Prometheus) to offload the scraping responsibility from the monitored servers.
- Filtering Unnecessary Metrics: Only collect metrics that are essential for monitoring and alerting requirements. Exclude irrelevant or low-priority metrics.

Table 4: Technical Details of Metric Collection (Digital Twin Input Data Requirements).

Monitoring Item	Metrics (Digital Twin Input Data Requirements)	API/Plugin	Monitoring Software
Job and Queue	Job Completion and Status and Queue Metrics • Specification of the jobs • Pending jobs • Running jobs with their mapping to resources • Priority • Restrictions • Run time • Statistics about job arrival • Time of day • Common burst times • Workload model on non-cluster resources required • Jobs scheduled on a node • Scheduler Performance	• HPC – Slurm Plugin – SlurmDBD (Slurm Database Daemon) API – Slurm Daemon Monitoring • Cloud – Kubernetes API • Edge – KubeEdge (Kubernetes API)	• HPC: HPC monitoring Systems (ExaMon, and Grafana), MySQL or MariaDB, but Slurm also supports PostgreSQL and SQLite • Cloud and Edge: Kubernetes Metrics Server and Prometheus and Grafana
HW Resources (hardware statistics - dynamically updated information) (hardware characteristics - static per-node/system information)	• Network – Actual real connections between components – Simplify parallel connections, data center 1 to cloud has 1 large connection – Throughput, latency, error-rate – Reservations for a job • CPU usage/ reservation/ type • GPU usage/ reservation/ type • Memory usage/reservation • Currently available storage – Local storage – Totally available storage – Storage technologies • Metrics depend on platform (HPC, . . . , IoT) – Common metrics: metadata, availability, latency – Links between platforms (bandwidth) • Extensibility for potential future platforms	HPC - IPMI HPC-NVIDIA-SMI, or DCGM, or NVML, or NVPerfKit for net and HPC	HPC monitoring Systems (like ExaMon, and Grafana)
Kubernetes	Kubernetes and KubeEdge(Edge/IoT) • Pod Scheduling Latency • Pod Startup Time • Resource Utilization • Cluster Scalability • Pod/Container Resource Consumption • Cluster Availability • Network Performance • Cluster Autoscaling Efficiency • Pod and Node Health • Cluster Management Overhead	• Kubernetes Metrics Server • Prometheus • Kubernetes API Server: Pod API, Node API, Service API, Deployment API, Namespace API, PersistentVolume API, Ingress API	• Kubernetes Metrics Server • Prometheus • TSDB (InfluxDB, KirosDB, TimescaleDB, OpenTSDB)
AI	Performance results of previously scheduled jobs		

- Optimizing Query Load: Carefully design and optimize PromQL queries to avoid excessive loads on Prometheus during query evaluations.
- Using Blackbox Exporter: For monitoring services or endpoints that are not natively instrumented for Prometheus, consider using the Blackbox Exporter. It can perform lightweight probes without putting additional load on main servers.

7.4.3 I/O Awareness

In Kubernetes, the default scheduler does not consider the heterogeneity of the underlying computing resources when scheduling pods in computing clusters. The standard metrics provided by Kubernetes, which are shown in Figure 8, offer a broad view of the system but lack the granularity needed for comprehensive insights. Studies have shown that by collecting I/O metrics and providing the k8s scheduler with these complementary information, can result in up to 15% better scheduling decisions [Wu22]. This performance increase can also serve as a foundation for the DECICE scheduler to incorporate I/O data into scheduling processes and will be elaborated in future deliverables.

```
kube-master:~$ kubectl top nodes
NAME           CPU(cores)   CPU%      MEMORY(bytes)   MEMORY%
kube-master    303m         3%        6001Mi          18%
kube-worker1   1613m        20%       11522Mi         35%
kube-worker2   451m         5%        8751Mi          27%
kube-worker4   550m         6%        9780Mi          30%
kube-worker5   237m         2%        10607Mi         33%
kube-gateway    849m         10%       9895Mi          30%
kube-service    171m         4%        3711Mi          23%
```

Figure 8: An example of kubectl command for basic resource usage

7.5 Additional Considerations

This section discusses considerations and approaches that could lead to performance improvements in certain cases but are outside of the DECICE project scope.

7.5.1 Federated learning (FL)

FL is a decentralized machine learning model in which a global prediction model is shared with multiple edge devices. Each device trains part of the model with locally available or stored data and only model updates, typically in the form of gradients, are sent to the central server. This scheme reduces the usage of network bandwidth and enhances data privacy as local training data is not transmitted.

As an example see the next-word prediction on a mobile keyboard [Har+18; Li+20], as illustrated in Section 7.5.1. User input can be used to train language models across a large number of mobile phones to predict the next word or phrase, enabling all users to benefit from auto-completing their next word based on the predictions. However, users might wish to protect their privacy and would be reluctant to share their data as well as wish to avoid wasting their limited bandwidth/battery power

on their phone. According to Li et al. [Li+20] federated learning algorithms have the potential to demonstrate the feasibility and benefit of training language models on the historical data available on client devices without exporting sensitive information to the servers.

In the setup shown in Section 7.5.1, each of the mobile phones gathers data based on user input and trains part of the model based on the local data. During each communication round a subset of mobile phones send their locally trained model as an update to the central server. This update is much smaller than the raw data and does not easily allow for deducing the original user input from the model. The updates from the users are incorporated on the server into a new global model, which is then rolled out to a number of devices. Assuming no further updates to the training data, this process lets the network eventually converge on a global model and the training process is completed.

Figure 9: Demonstration of federated learning with mobile phones held by individual users as devices. Adapted from [Li+20].

FL problem formulation Following Li et al. [Li+20], in the FL problem, statistical learning of a single global model is achieved by processing data on, potentially, millions of edge devices. The aim is to achieve updates to the model under the constraint that devices generate data that is processed and stored locally. Only intermediate occasional updates are transmitted to the central server. The goal is to minimize the objective function $F(w)$,

$$\min_w F(w), \text{ where } F(w) := \sum_{k=1}^{m} p_k F_k(w). \tag{12}$$

Here, m is the total number of devices, $p_k \geq 0$ and $\sum_k p_k = 1$. F_k is the local objective function for the kth device, defined as,

$$F_k(w) = \frac{1}{n_k} \sum_{j_k=1}^{n_k} f_{j_k}(w; x_{j_k}, y_{j_k}), \tag{12}$$

where n_k is the number of samples available locally. Let the total number of samples be $n = \sum_k n_k$. Further, let p_k be a user defined term which specifies the relative impact of each device. It can take

two natural values, *viz.,* $p_k = (1/n)$ or $p_k = (n_k/n)$. Here, $x_{j_k} \in \mathbb{R}^{n_{in} \times 1}$, and $y_{j_k} \in \mathbb{R}^{n_{out} \times 1}$ are input and output training data, respectively [Che+21].

Scheduling in edge federated learning Edge devices are inherently heterogeneous. In each iteration of a synchronous federated learning, the participating devices need to update their computational results at the central server. Thus, training speed is dictated by the slowest device and network bandwidth. Hence, an efficient scheduler needs to allocate optimal edge resources for a given application. Currently, there are at least four major research directions to achieve this goal [Xia+21].

1. Participant selection: In this scheme, initially some nodes are randomly selected as participants to perform the computations using their local private data for one iteration and their training speed is analyzed. Further, the node selection is posed as a mathematical optimization problem based on the computational resources and previous training time information. Effect of different scheduling policies, *viz.,* random scheduling (RS), round robin (RR), and proportional fair (PF) etc., are studied for a given problem setting [Yan+20].
2. Resource optimization: Due to the heterogeneity of edge nodes, the available resources in terms of computational power and network bandwidth may vary drastically. To improve resource allocation, nodes with more compute power should be given a bigger share of compute tasks.
3. Asynchronous training: The majority of edge federated learning research is focused on synchronous training, however, asynchronous training could provide an opportunity to significantly increase the capabilities of federated learning in heterogeneous environments.
4. Incentive Mechanism: In environments with independent users, it might be necessary to provide some form of incentive to motivate users to collaborate in a federated learning network. Some researchers have been looking into strategies for providing compensation to users to reward them for their compute power of input data.

Federated Learning in DECICE The two main advantages of FL are privacy, due to not sharing raw training data with peers, and reduced bandwidth usage, as raw training data is processed locally instead of transmitted via a potentially slow network. This comes at the cost of having to provide enough compute power to perform the train locally.

Assuming a number of nodes, including cloud and edge nodes, are controlled via the DECICE framework then these nodes are all also running a Kubernetes or KubeEdge stack in order to connect to the cluster. Furthermore, in this assumption, the edge nodes are connected to sensors that collect data, which is interesting for training a machine learning model. If all the nodes are connected into a cluster, they already have to trust the master nodes as these may submit arbitrary workloads on the other nodes, including accessing the locally stored data. Therefore, the advantage of privacy via FL is negated as there is no privacy for the edge nodes in this scenario from the master nodes.

Nevertheless, considering the second advantage of reduced bandwidth usage, if the training data is, for example, a camera feed, then the network of the edge nodes could be incapable of streaming this data to a central server while maintaining a level of quality required for training a ML model. In this case, the solution would be to perform pre-processing or even training via FL on the edge nodes

to reduce the load on the network. This still requires the edge nodes to be capable of performing said computations themselves.

While it is certainly possible for FL to operate on top of a cluster managed by DECICE and to benefit from local computation, it still highly depends on whether a given use case can actually benefit from FL and whether the hardware setup available also calls for FL. Given this limited applicability, implementing federated learning on top of DECICE is currently not within the scope of the project.

7.5.2 Container Migration

Container migration refers to the concept of relocating a running container from one node to another without restarting the container. This is useful if restarting a given container would mean losing important data while also having to move it to a different node, for example, because the original node has to go down for maintenance.

In general there are two approaches to container migration, cold migration and live migration. For cold migration, the migration process works by freezing the container process (or process tree) and creating a checkpoint of the memory state of the process. This checkpoint along with the container state on disk is send to the new server. There the container state and the memory checkpoint are restored and unfrozen, letting the container continue to operate without a restart.

Cold migration has the problem that while the container is being migrated, it is frozen and if the application running a service that service is disrupted until the migration is completed. If such an interruption is not possible, a live migration is a solution to migrate the container without a restart and without a service interruption.

For a live migration, the container is kept running and instead a duplicate of it is created on the target server. The live migration processes syncs first the disk state and then the memory state, starting with memory pages that have not been changed for the longest time. When the duplicate is in sync, its process is also run and traffic coming into the original container is mirrored into the new container. Once the new container is fully in sync, the mirrored traffic can completely switch over to the new container and shut off the original container. A live migration is much more complicated than a cold migration and should only be used when necessary.

While the ability to perform container migrations, both cold and live, would be a useful addition for certain use cases and it would be possible to implement this functionality as part of the administration tools of the DECICE framework, the actual implementation of such a feature is out of scope for the current objectives of the DECICE project.

8 References

[Ahm+22] Imtiaz Ahmad et al. "Container scheduling techniques: A Survey and assessment". In: *Journal of King Saud University - Computer and Information Sciences* 34.7 (2022), pp. 3934–3947. ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2021.03.002. URL: https://www.sciencedirect.com/science/article/pii/S1319157821000562.

[Ali23] Alibaba. *Alibaba Cluster Trace Program*. 2023. URL: https://github.com/alibaba/clusterdata (visited on 08/28/2023).

[Beh+15] Babak Behzad et al. "Dynamic Model-Driven Parallel I/O Performance Tuning". In: *2015 IEEE International Conference on Cluster Computing*. 2015, pp. 184–193. DOI: 10.1109/CLUSTER.2015.37.

[Car22] Carmen Carrión. "Kubernetes Scheduling: Taxonomy, Ongoing Issues and Challenges". In: *ACM Comput. Surv.* 55.7 (Dec. 2022). ISSN: 0360-0300. DOI: 10.1145/3539606. URL: https://doi.org/10.1145/3539606.

[Che+21] Mingzhe Chen et al. "Convergence Time Optimization for Federated Learning Over Wireless Networks". In: *IEEE Transactions on Wireless Communications* 20.4 (Apr. 2021), pp. 2457–2471. ISSN: 1558-2248. DOI: 10.1109/TWC.2020.3042530.

[Fou] Cloud Native Computing Foundation. *Prometheus Project*. URL: https://www.cncf.io/projects/prometheus/ (visited on 08/25/2023).

[Goo19] Google. *Borg Cluster Workload Traces*. 2019. URL: https://github.com/google/cluster-data (visited on 08/28/2023).

[Gri14] Michael Grieves. "Digital twin: manufacturing excellence through virtual factory replication". In: *White paper* 1.2014 (2014), pp. 1–7.

[Har+18] Andrew Hard et al. "Federated Learning for Mobile Keyboard Prediction". In: *CoRR* abs/1811.03604 (2018). arXiv: 1811.03604. URL: http://arxiv.org/abs/1811.03604.

[Jon+20] David Jones et al. "Characterising the Digital Twin: A systematic literature review". In: *CIRP journal of manufacturing science and technology* 29 (2020), pp. 36–52.

[Kau+20] Kuljeet Kaur et al. "KEIDS: Kubernetes-Based Energy and Interference Driven Scheduler for Industrial IoT in Edge-Cloud Ecosystem". In: *IEEE Internet of Things Journal* 7.5 (May 2020), pp. 4228–4237. ISSN: 2327-4662. DOI: 10.1109/JIOT.2019.2939534.

[Kun+22] Julian Kunkel et al. *Device-Edge-Cloud Intelligent Collaboration framEwork (DECICE)*. https://cordis.europa.eu/project/id/101092582. Accessed: 10/2/2023. 2022. DOI: 10.3030/101092582.

[Li+20] Tian Li et al. "Federated Learning: Challenges, Methods, and Future Directions". In: *IEEE Signal Processing Magazine* 37.3 (May 2020), pp. 50–60. ISSN: 1558-0792. DOI: 10.1109/MSP.2020.2975749.

[Li+23] Jing Li et al. "Service Home Identification of Multiple-Source IoT Applications in Edge Computing". In: *IEEE Transactions on Services Computing* 16.2 (Mar. 2023), pp. 1417–1430. ISSN: 1939-1374. DOI: 10.1109/TSC.2022.3176576.

[NAG23] NAG. *NAG library*. 2023. URL: https://www.nag.com/numeric/nl/nagdoc_latest/flhtml/h/hintro.html (visited on 06/12/2023).

[Pro] Prometheus, an open-source systems monitoring and alerting toolkit. *Architecture*. Online; accessed July 15, 2023. URL: https://prometheus.io/docs/introduction/overview/.

[RV15] Bjorn Rabenstein and Julius Volz. "Prometheus: A Next-Generation Monitoring System". In: Dublin: USENIX Association, May 2015.

[Sen+23] Khaldoun Senjab et al. "A Survey of Kubernetes Scheduling Algorithms". In: *Journal of Cloud Computing* 12.1 (June 13, 2023), p. 87. ISSN: 2192-113X. DOI: 10.1186/s13677-023-00471-1. URL: https://doi.org/10.1186/s13677-023-00471-1 (visited on 08/29/2023).

[Wu22] Zheyun Wu. "An I/O-aware scheduler for containerized data-intensive HPC tasks in Kubernetes-based heterogeneous clusters". KTH, School of Electrical Engineering and Computer Science (EECS), 2022, p. 51.

[Xia+21] Qi Xia et al. "A survey of federated learning for edge computing: Research problems and solutions". In: *High-Confidence Computing* 1.1 (2021), p. 100008. ISSN: 2667-2952. DOI: https://doi.org/10.1016/j.hcc.2021.100008. URL: https://www.sciencedirect.com/science/article/pii/S266729522100009X.

[Yan+20] Howard H. Yang et al. "Scheduling Policies for Federated Learning in Wireless Networks". In: *IEEE Transactions on Communications* 68.1 (Jan. 2020), pp. 317–333. ISSN: 1558-0857. DOI: 10.1109/TCOMM.2019.2944169.

[Zha+17] Dong Zhang et al. "Container oriented job scheduling using linear programming model". In: *3rd International Conference on Information Management (ICIM)*. 2017, pp. 174–180. DOI: 10.1109/INFOMAN.2017.7950370.

A Prometheus Metrics

Metrics	Description
node_disk_reads_completed_total	This is the total number of reads completed successfully
node_disk_reads_merged_total	-
node_disk_writes_merged_total	-
node_disk_discards_merged_total	Reads and writes which are adjacent to each other may be merged for efficiency
node_disk_read_bytes_total	This is the total number of bytes read successfully
node_disk_read_time_seconds_total	This is the total number of seconds spent by all reads
node_disk_writes_completed_total	This is the total number of writes completed successfully
node_disk_written_bytes_total	This is the total number of bytes written successfully
node_disk_write_time_seconds_total	This is the total number of seconds spent by all writes
node_disk_io_now	The only field that should go to zero
node_disk_io_time_seconds_total	Number of seconds spent doing I/Os
node_disk_io_time_weighted_seconds_total	Weighted # of seconds spent doing I/Os
node_disk_discards_completed_total	This is the total number of discards completed successfully
node_disk_discarded_sectors_total	This is the total number of sectors discarded successfully
node_disk_discard_time_seconds_total	This is the total number of seconds spent by all discards
node_disk_flush_requests_total	The total number of flush requests completed successfully
node_disk_flush_requests_time_seconds_total	The total number of seconds spent by all flush requests

Table 5: Examples of I/O metrics

Metrics	Description
node_network_receive_bytes_total	The total number of bytes received by the interface
node_network_transmit_bytes_total	The total number of bytes transmitted by the interface
node_network_receive_packets_total	The total number of packets received by the interface
node_network_transmit_packets_total	The total number of packets transmitted by the interface
node_network_collisions_total	The total number of collisions on the interface
route_table_entries_total	The total number of entries in the routing table
route_table_prefixes_total	The total number of prefixes in the routing table
route_table_last_updated_seconds	The time in seconds since the routing table was last updated
link_rx_errors	The number of receive errors on the link
link_tx_errors	The number of transmit errors on the link
link_duplex	The duplex mode of the link (full-duplex or half-duplex)
link_speed	The speed of the link in Mbps
icmp_ping_count_total	The total number of ICMP ping requests sent
icmp_ping_rtt_seconds	The average round-trip time of ICMP ping requests
icmp_ping_packet_loss	The percentage of ICMP ping requests that were lost

Table 6: Examples of network metrics

Metrics	Description
kube_pod_annotations	Kubernetes annotations converted to Prometheus labels
kube_pod_info	Information about pod
kube_pod_ips	Pod IP addresses
kube_pod_start_time	Start time in unix timestamp for a pod
kube_pod_completion_time	Completion time in unix timestamp for a pod
kube_pod_owner	Information about the Pod's owner
kube_pod_labels	Kubernetes labels converted to Prometheus labels
kube_pod_nodeselectors	Describes the Pod nodeSelectors
kube_pod_status_phase	The pods current phase
kube_pod_status_qos_class	The pods current qosClass
kube_pod_status_ready	Describes whether the pod is ready to serve requests
kube_pod_status_scheduled	Describes the status of the scheduling process for the pod
kube_pod_container_info	Information about a container in a pod
kube_pod_container_status_waiting	Describes whether the container is currently in waiting state
kube_pod_container_status_waiting_reason	Describes the reason the container is currently in waiting state
kube_pod_container_status_running	Describes whether the container is currently in running state
kube_pod_container_state_started	Start time in unix timestamp for a pod container
kube_pod_container_state_started	Start time in unix timestamp for a pod container
kube_pod_container_status_terminated	Describes whether the container is currently in terminated state
kube_pod_container_status_terminated_reason	Describes the reason the container is currently in terminated state
kube_pod_container_status_last_terminated_reason	Describes the last reason the container was in terminated state
kube_pod_container_status_last_terminated_exitcode	Describes the exit code for the last container in terminated state.
kube_pod_container_status_ready	Describes whether the containers readiness check succeeded
kube_pod_status_initialized_time	Time when the pod is initialized.
kube_pod_status_ready_time	Time when pod passed readiness probes
kube_pod_status_container_ready_time	Time when the container of the pod entered Ready state
kube_pod_container_status_restarts_total	The number of container restarts per container

Table 7: Examples of Kubernetes metrics

Metrics	Description
kube_pod_container_resource_requests	The number of requested request resource by a container. It is recommended to use the kube_pod_resource_requests metric exposed by kube-scheduler instead, as it is more precise.
kube_pod_container_resource_limits	The number of requested limit resource by a container. It is recommended to use the kube_pod_resource_limits metric exposed by kube-scheduler instead, as it is more precise
kube_pod_overhead_cpu_cores	The pod overhead in regards to cpu cores associated with running a pod
kube_pod_overhead_memory_bytes	The pod overhead in regards to memory associated with running a pod
kube_pod_runtimeclass_name_info	The runtimeclass associated with the pod
kube_pod_created	Unix creation timestamp
kube_pod_deletion_timestamp	Unix deletion timestamp
kube_pod_restart_policy	Describes the restart policy in use by this pod
kube_pod_init_container_info	Information about an init container in a pod
kube_pod_init_container_status_waiting	Describes whether the init container is currently in waiting state
kube_pod_init_container_status_waiting_reason	Describes the reason the init container is currently in waiting state
kube_pod_init_container_status_running	Describes whether the init container is currently in running state
kube_pod_init_container_status_terminated	Describes whether the init container is currently in terminated state
kube_pod_init_container_status_terminated_reason	Describes the reason the init container is currently in terminated state
kube_pod_init_container_status_last_terminated_reason	Describes the last reason the init container was in terminated state
kube_pod_init_container_status_ready	Describes whether the init containers readiness check succeeded
kube_pod_init_container_status_restarts_total	The number of restarts for the init container
kube_pod_init_container_resource_limits	The number of CPU cores requested limit by an init container
kube_pod_init_container_resource_requests	The number of CPU cores requested by an init container
kube_pod_spec_volumes_persistentvolumeclaims_info	Information about persistentvolumeclaim volumes in a pod
kube_pod_spec_volumes_persistentvolumeclaims_readonly	Describes whether a persistentvolumeclaim is mounted read only
kube_pod_status_reason	The pod status reasons
kube_pod_status_scheduled_time	Unix timestamp when pod moved into scheduled status
kube_pod_status_unschedulable	Describes the unschedulable status for the pod
kube_pod_tolerations	Information about the pod tolerations
kube_pod_service_account	The service account for a pod

Table 8: Examples of Kubernetes metrics (Continued)