DECICE

DEVICE-EDGE-CLOUD INTELLIGENT COLLABORATION FRAMEWORK

Grant Agreement: 101092582

D3.1 Synthetic Test Environment

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:	D3.1
Deliverable title:	Synthetic Test Environment
Deliverable version:	1.0
Work Package number:	WP3
Work Package title:	Open Framework and Virtual Training Environment
Responsible partner	GWDG
Due Date of delivery:	2023-11-30
Actual date of delivery:	2023-11-30
Dissemination level:	PU
Type:	C
Editor(s):	Felix Stein (UGOE) Mirac Aydin (GWDG)
Contributor(s):	
Reviewer(s):	Dirk Pleiter (KTH) Andrea Rivetti (TOP-IX)
Project name:	Device-Edge-Cloud Intelligent Collaboration framEwork
Project Acronym:	DECICE
Project starting date:	2022-12-01
Project duration:	36 months
Rights:	DECICE Consortium

Document History

Version	Date	Partner	Description
0.1	2023-11-15	GWDG/UGOE	First draft of deliverable
0.2	2023-11-17	GWDG/UGOE	Add additional sections
0.3	2023-11-24	KTH/TOP-IX	Internal review
1.0	2023-11-28	GWDG/UGOE	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 purpose of this deliverable is to create a synthetic test environment that can replicate diverse real-world scenarios and serves as a critical enabler for an AI-based scheduler, facilitating accelerated training and optimization. The document provides an overview of the synthetic test environment's architecture, analyzes its core elements, and explains the workflow within the environment. The primary focus is the integration of synthetic test environment with the Digital Twin, scenario database, and scheduler controller through APIs, enabling efficient data retrieval and its integration into the scheduler's training process. Furthermore, the document emphasizes the deployment of the synthetic test environment in a Kubernetes cluster which is crucial for deployment of containerized applications and microservice management in a heterogeneous compute continuum. Finally, the deliverable considers further improvements of the synthetic test environment to optimize the training process of the AI scheduler.

Contents

1 Purpose and Scope of the Deliverable

The purpose of D3.1: *Synthetic Test Environment* is to provide the initial version of code and documentation for the environment that allows the training of AI-schedulers. Slight adaptations may be necessary after the initial version while the overall system architecture is not finalized yet at the time of writing. It also contains a description of the architecture of the environment and software components.

2 Abstract / publishable summary

In the scope of this deliverable, a synthetic test environment and its interactions with a digital twin, an AI-scheduler and a scheduler controller were created. This environment was equipped with the essential APIs required to establish seamless communication with the aforementioned components, providing an integrated and cohesive testing framework.

To ensure the validity and reliability of the AI-scheduler, the MIT SuperCloud Dataset was employed for testing purposes. This approach provided the project with a solid foundation for evaluation and assessment.

In order to evaluate and test API calls within the environment, proper test functions have been implemented. These functions played a pivotal role in the validation of system performance and functionality. To streamline and automate the testing process, a Continuous Integration and Continuous Deployment (CI/CD) pipeline was established. This pipeline enhanced the project's efficiency and contributed to the reliability of the testing procedures.

In a forward-looking move towards improved scalability and management, all software components were containerized, ensuring encapsulation and isolation. These containerized components were then seamlessly deployed on a Kubernetes cluster, offering a robust infrastructure for the entire framework. This deployment strategy further optimized the management of resources, making the system highly adaptable to evolving requirements and workloads.

3 Project objectives

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

Macro-objectives	Contribution of this deliverable
(O1) Develop a solution that allows to leverage a compute continuum ranging from cloud and HPC to edge and IoT.	The Synthetic Test Environment provides the layers and interfaces that are used by other software components which are important to develop the framework.
(O2) Develop a scheduler supporting dynamic load balancing for energy-efficient compute orchestration, improved use of green energy, and automated deployment.	With the help of the Synthetic Test Environment, different scenarios can be simulated for the faster training of the scheduler to support better workload orchestration in the compute continuum.
(O3) Design and implement an API that increases control over network, computing and data resources.	The Synthetic Test Environment integrates the necessary APIs for the communication and data flow between different components such as the scheduler, Dynamic Digital Twin and the scheduler controller.
(O4) Design and implement a Dynamic Digital Twin of the system with AI-based prediction capabilities as integral part of the solution.	The Synthetic Test Environment interacts with the Dynamic Digital Twin and provides the necessary data that is transferred to the AI-based scheduler.
(O5) Demonstrate the usability and benefits of the DECICE solution for real-life use cases.	The Synthetic Test Environment contributes the development of the better components that improve 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.	The improvement and optimization of the AI-based scheduler with the Synthetic Test Environment will provide efficient service deployment in the compute continuum.

4 Changes made and/or difficulties encountered, if any

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

5 Sustainability

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.

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

6.3 Publications in preparation OR submitted

See Table 2.

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 and system characteristics of the synthetic test environment. The outline of this report portrays the implementation details for the DECICE training framework ranging from high level architecture design to implementation and deployment. For this purpose the deliverable is divided into four major parts. The first section deals with the architectural design of the synthetic test environment. It goes into detail on how the training ground was designed from a high level perspective and explains in particular the inner workings of the software component. The section shows how information are being exchanged and concentrates on the data flow within the system.

Section 7.2 goes into detail on what tools, libraries and frameworks were used in order to program the synthetic test environment. This section also describes the information exchange within the framework which was implemented using a micro service approach were individual software components are being developed independently and connected afterwards via HTTP APIs. It also explains how the DECICE project deals with software quality and what standards we want to pursue while developing and deploying the software. The chapter concludes with a description of our documentation, regarding not only code documentation but also features our OpenAPI documentation as well as our DECICE knowledge base to get an overview of the whole project.

The third part of this deliverable - section 7.3 - details the deployment of the synthetic test environment. It shows how the DECICE project deals with deploying the application to a live system by containerizing each micro service. This process requires a coordination service to be able to direct and monitor each service and for this purpose we need the orchestration software Kubernetes. We show how we deployed a fully functional synthetic test environment application on our OpenStack cloud computing instance with the help of Kubernetes and Kubespray.

The last section of this deliverable, section 7.4, deals with the data collection for the digital twin as well as the AI scheduler. We collected and processed data for our other components in the DECICE framework as well as for the synthetic test environment itself. Having real data in the first place omits the need to heavily adapt the training environment at a later stage to account for different data and also serves as test data for complete run-throughs to test the inner workings of the application.

7.1 Architecture

The *Synthetic Test Environment* (STE) of the DECICE framework refers to the training and testing ground for the AI-schedulers. The purpose of an STE is to provide a controlled and repeatable environment for testing systems and software and it is a suitable training method when the costs or risks of real-world training are too high, or when STEs are the only option for providing training [Ros+00; RB01]. In case of this deliverable and future development, it represents a simulation of the production environment for the actual DECICE framework together with its compute continuum. For this deliverable and for our future development process, we are using the terms synthetic test environment and virtual training environment (VTE) interchangeably. These two terms can be considered synonyms and both of them refer to simulated environments that are used to train or test systems and softwares. An outline of the current architecture of the STE is depicted in Figure 1.

As the purpose of the STE is to provide the environment and the control flow for training the reinforcement learning models used by the AI scheduler, the training process of the STE starts by calling the controller entity (1). It takes care of kicking off the whole training process by triggering a metrics refresh on the digital twin (2) by requesting training scenarios from a database (DB) (3). This *ScenarioDB* stores the accumulated training data as different scenarios of HPC/Cloud data in a PostgreSQL database. Scenarios generally refer to various hypothetical or simulated situations that are used to train the AI scheduler and to evaluate the performance, resilience, and adaptability of it. In the first iteration the DB has to be filled manually beforehand but will be replaced by an automated system in later development stages that is able to pull new data automatically from different sources (i.e. pulling metrics from a Prometheus live service). After the data has been pulled from the DB a wrapper takes care of processing the extracted data to a JSON and sends it over an HTTP API to the digital twin (DT) (4). A digital twin typically is a virtual representation of an actual physical product that encompasses its entire life cycle, including its physical and functional characteristics. It acts as a virtual mirror, capable of exchanging and receiving information [TWE18]. In the case of the DECICE training framework it represents accumulated information of node and job metrics (cf. Section 7.4) and holds them in memory for a fast pulling process.

After the scenario data has been pushed to the DT the VTE controller entity informs the scheduler controller that new data for a training process is available (5). The scheduler controller pulls the latest data from the DT (6) and applies a prioritization to create a scheduling queue from all jobs with pending pods, ordering them based on their priority, age and whether parts of them are already running (7). A subsequent filtering step is applied to determine the pool of eligible nodes based on policies and resource requests for each pod in the current job (8). After the post-processing of the JSON data the scheduler controller sends the data of current jobs and the eligible nodes per pod to all schedulers that have been registered in the system (9). The first iteration of the DECICE training framework includes three schedulers: an AI scheduler, an Integer Linear Programming (ILP) scheduler, and a heuristic scheduler. Having three different schedulers in the training environment gives the user the flexibility to observe various scheduler models, performance differences, and to choose one of them. The heuristic scheduler, in this case, serves as a fallback option. Later iterations should also enable the possibility to use, configure, and train your own schedulers. The current number of schedulers serves only the purpose of a proof of concept, and there could be more or fewer schedulers in the future, including your own scheduler. The respective wrapper converts the JSON into a format that can be used by each individual scheduler. In the case of the JSON-to-Tensor Wrapper, the JSON data is converted to tensors that can be processed by machine learning models, which are then send to multiple models that each score the nodes eligible for a given pod. In the case of the JSON-to-ILP Wrapper it is send to multiple ILP models that compute the optimal scheduling decision based on their respective optimization target. In both cases the N.N. Model or N.N. ILP symbolizes further models and ILPs with additional optimization targets (10).

After a run through all three wrappers send their scores to the evaluator component which compares the scores from the AI schedulers to the scores from the ILP scheduler while using the heuristic scheduler as a fall back (11) and sends them back for applying reward scores to the individual models (12). The Aggregator weighs the scores and sends the final scheduling decisions back to the scheduler controller (13).

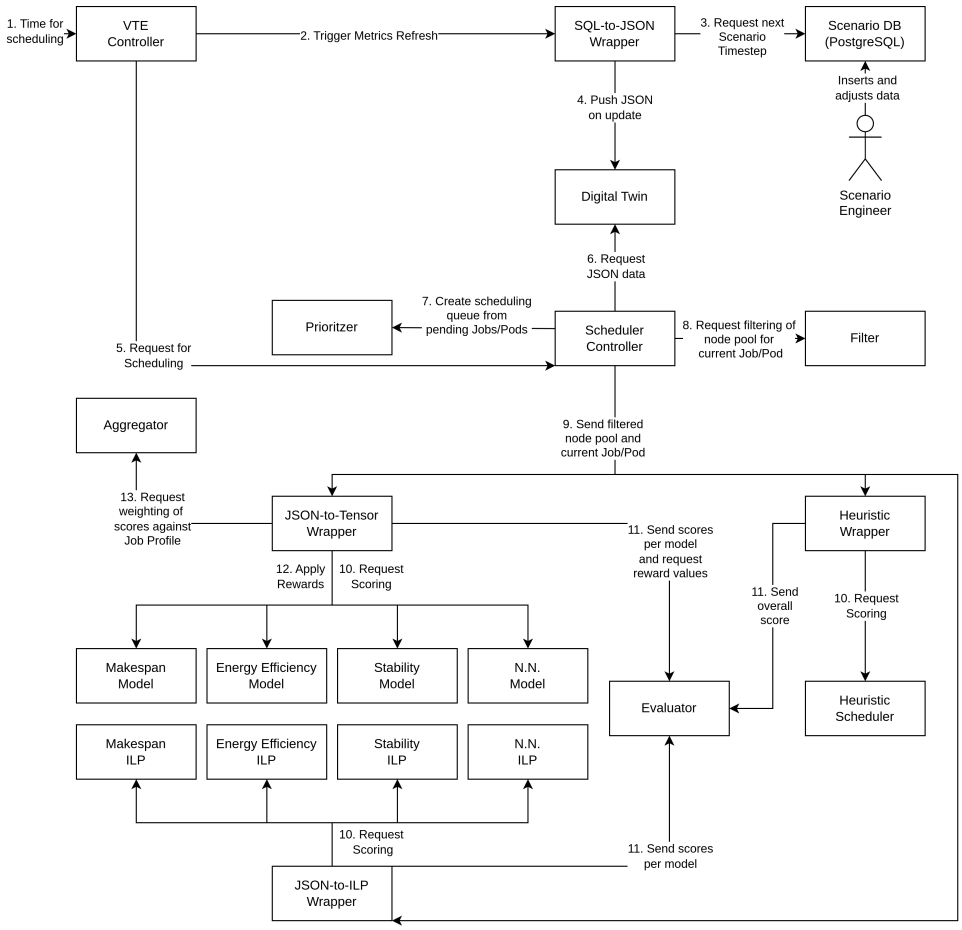

Figure 1: DECICE Model Component Diagram - Training

7.2 Development

The development of a software application is a complex endeavor, encompassing a wide range of activities, from conceptualizing the application's purpose and functionalities to implementing and deploying the final product. This section delves into the process of the software development of the DECICE training framework, providing an overview of the key phases and activities involved. First the key technologies, libraries and techniques used in this project are described and explained and will form the basis of the architecture. The next part deals with the software quality standards which the DECICE project implemented and we want to build upon for upcoming deliverables. This implementation serves as a guideline for later deliverables and implementations and should be used throughout the project to meet the quality standards we set ourselves.

Since the training environment is encapsulated into different software components, the subsequent section deals with the communication of these based on HTTP requests and how they were implemented. The last section deals with the documentation of the project, ranging from code documentation for a better understanding of the intricacies of the code base, to the OpenAPI documentation of our micro services together with the DECICE knowledge where the high-level architecture is explained. This way we are able to provide a sophisticated documentation for external as well as internal partners who are involved in the project.

7.2.1 Software Engineering

The STE and its accompanying software components were programmed in Python. The reason for this is Python being a versatile and widely used programming language that favors readability, has an extensive library ecosystem and allows for rapid prototyping due to its simplicity. For the HTTP backend we chose FastAPI. It is a high-performance web framework for building REST APIs [Ram]. The reason for working with this framework is the build-in *OpenAPI* schema which is automatically generated with a UI documentation, including Swagger UI. It also provides automatic data validation via the *Pydantic* library. Even though Python is a dynamically typed language, meaning that the type of a variable is determined only during run time, *Pydantic* enforces type annotation throughout the code base and thus serves as a guideline for a clean code approach. It allows for some sort of static typing that compiled languages like Rust, Go or C offer without the need of relying on compilation and purely use the Python interpreter for validation. Accordingly, we were able to still use Python and its merits while at the same time being able to avoid demerits that interpreted languages have. We chose *Pytest* as our testing framework. Its ease of use allows for sophisticated and highly customizable unit tests enabling automated test execution and feedback loops during merge requests. This integration ensures that tests are consistently run and results are readily available for monitoring and ensuring integrity of our code base. The code base of the STE can be found in DECICE GitHub repository [DEC].

Each service within the STE runs as a micro service in a containerized environment. This design decision was made to give every partner the flexibility to choose their own approach when implementing software features. While some components are easier to develop in Python, others might benefit from a different language. Especially in the field scheduling decisions Go is often times a preferred candidate. Each micro service receives and sends data via HTTP requests in the form of *CRUD* operations [Moz]. On one hand this allows for an independent development of each software component and on the other hand each service is encapsulated and has its own boundaries to operate without affecting others on a large scale when changes are done.

7.2.2 Software Quality

Software quality is a critical aspect of the software development life cycle, ensuring that the final product meets or exceeds expectations. In order to achieve this goal, an agile approach was also utilized during the development of the code base.

Agile software development methods are a collection of practices that promote iterative, incremental development and continuous improvement. In contrast to traditional waterfall methods, which require all requirements to be defined upfront and then executed in a linear fashion, agile methods emphasize continuous feedback and adaptation. By breaking down the development process into smaller, manageable iterations, teams can regularly inspect and adapt their work. This iterative development style is similar to constructing a prototype that grows and matures over time. As new features and functionalities are introduced, the development team iteratively enhances the software, responding to feedback and changing requirements. Moreover, the incorporation of micro services during implementation aligns with the agile philosophy, enabling the construction of modular and scalable architectures.

To ensure optimum software quality, a set of methods and tools has been integrated into the development of the Synthetic Test Environment:

Ticket System: A centralized tracking system of GitLab has been used to create tickets for every issue and development activity in the project. This systematic approach facilitates efficient issue resolution and provides a transparent overview of project progress.

Branch Protection: Branch protection rules have been implemented to safeguard the integrity of the main branch, preventing unauthorized or accidental changes from being merged directly into the production environment. Merge requests serve as a gate keeping mechanism, requiring code reviews and approvals before changes are integrated into the main branch. This rigorous review process helps maintain code quality and stability, preventing regressions or unexpected errors.

Version Control System: Git and GitLab have been used during the development process. They provide a robust version control system for tracking changes, enabling developers to maintain different versions of the codebase simultaneously. This system allows for reverting to previous versions in case of errors or unexpected behavior, ensuring that developers can always maintain a stable and functional codebase. GitLab's branching feature facilitates parallel development, allowing different teams to work on separate features without affecting the main codebase until the feature is ready for integration.

Hosting Service: GitLab has served as a centralized repository for the project's code, providing a secure and accessible platform for developers to collaborate and contribute. Its integrated issue tracking and merge request system streamline the development process, facilitating communication and ensuring that changes are reviewed and approved before integration.

CI/CD and Automated Testing: A set of unit tests has been developed for the STE. These tests involve sending HTTP requests to the various endpoints within the software component routes. Continuous Integration/Continuous Deployment (CI/CD) pipeline on GitLab has been integrated to automate this testing processes. This automation streamlines the testing phase, providing quick feedback on code changes and contributing to overall development efficiency and reliability.

7.2.3 Documentation

As described in the Section 7.2.1 above, FastAPI has been used to develop the APIs in the STE. FastAPI, in turn, utilizes SwaggerUI for documentation purposes. Each software components has its own documentation accessible through the following link structure: **http://IP:PORT/docs**. By navigating to this link, one can access the SwaggerUI documentation for each specific component. This modular approach allows for the clear and distinct documentation of individual software components, enhancing the overall manageability and understanding of the project. An example of an API documentation of the digital twin in the STE is shown in Figure 2:

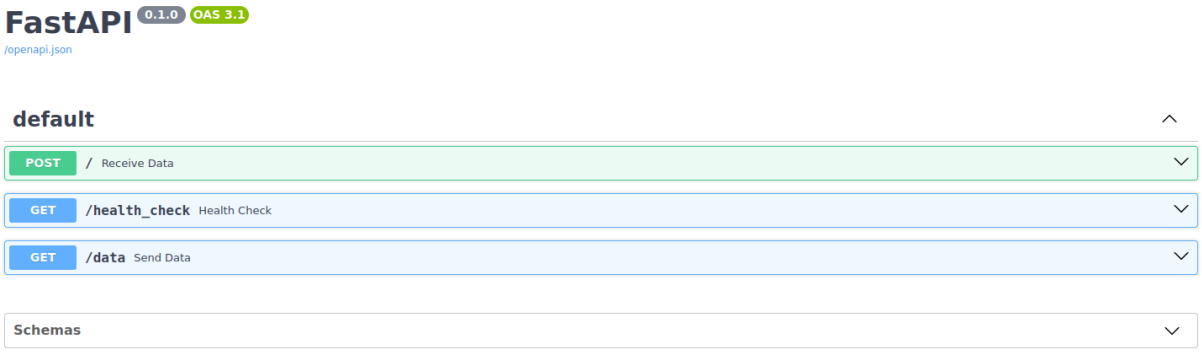

Figure 2: SwaggerUI API documentation of the Digital Twin

In addition to the component-specific SwaggerUI documentation, an internal documentation page for the project has been setup and it is accessible to all project partners. This dedicated page is hosted on GitLab, was built with Hugo [Hug] and serves as a comprehensive resource containing essential information about the project. This centralized documentation hub provides project partners with a convenient and organized source of information, fostering collaboration and ensuring that stakeholders have access to the necessary details for effective engagement with the project.

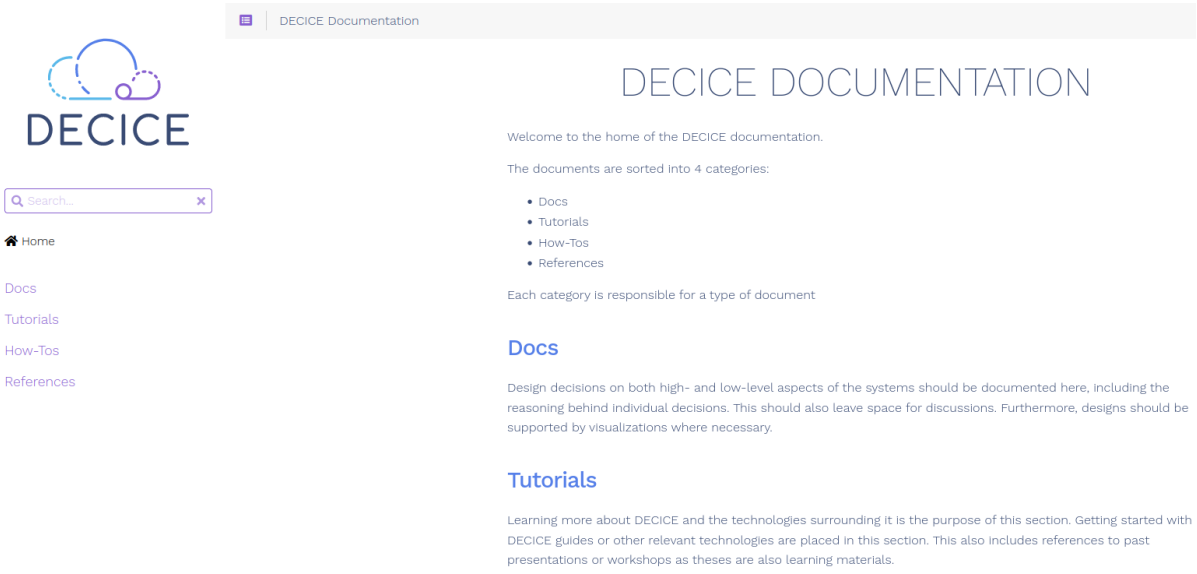

Figure 3: Internal DECICE documentation

The internal documentation page consists of the following sections:

Docs: Design decisions on both high- and low-level aspects of the systems should be documented here, including the reasoning behind individual decisions. This should also leave space for discussions. Furthermore, designs should be supported by visualizations where necessary.

Tutorials: Learning more about DECICE and the technologies surrounding it is the purpose of this section. Getting started with DECICE guides or other relevant technologies are placed in this section. This also includes references to past presentations or workshops as theses are also learning materials.

How-Tos: Getting things done via code or shell snippets or longer instructions is the purpose of this section. The focus lays on productivity, hacks and solutions to common problems. This section

is distinct from Tutorials as it focuses less on learning and more on productivity.

References: API documentation and other references to code that are automatically pulled from the source code of individual projects is placed in this section. It should not be necessary to manually create pages in this section.

7.3 Deployment

In the dynamic landscape of modern software development, containerization has emerged as a transformative technology, providing a streamlined and efficient way to package, distribute and deploy software applications. By encapsulating an application and its dependencies into a lightweight, portable container, developers can ensure consistency across different environments and simplify the deployment process. The orchestration of these containers is where Kubernetes, an open-source container orchestration platform, takes center stage. Kubernetes not only automates the deployment, scaling, and management of containerized applications but also facilitates the seamless coordination of complex, distributed systems.

This chapter explains the containerization and orchestration of the STE. The following section 7.3.1 presents the containerization in detail, while section 7.3.2 is dedicated to orchestration.

7.3.1 Containerization

Containerization has emerged as a transformative technology in the realm of software development and deployment, providing a robust solution to the challenges posed by traditional bare-metal setups. Unlike traditional methods, containerization encapsulates an application and its dependencies into a lightweight, portable unit known as a container. This approach brings several advantages, chief among them being enhanced portability and consistency across various computing environments. One of the key strengths of containerization lies in its ability to simplify deployment procedures. Containers package an application along with its dependencies, libraries, and configurations, creating a self-sufficient unit that can be effortlessly moved from one environment to another.

Complementing containerization is the microservice architecture, a paradigm shift in software design. Microservices advocate breaking down applications into small, independent services, each addressing a specific business capability. These services communicate through well-defined APIs, promoting modularity and agility. One of the primary advantages of microservices lies in scalability. Unlike monolithic architectures, microservices enable individual components to scale independently, optimizing resource allocation based on specific service requirements. Moreover, the fault isolation inherent in microservices enhances system resilience. If one microservice encounters issues, it does not necessarily compromise the entire application, contributing to a more robust and reliable system.

To optimize the management and deployment of diverse software components within the Synthetic Test Environment, the approach of containerization and microservices has been utilized. For each distinct software component, dedicated Dockerfiles were crafted, encapsulating the specific dependencies, libraries, and configurations requisite for its seamless execution. The Dockerfiles, serving as blueprints for container creation, systematically detail step-by-step instructions for configuring the container environment. Leveraging the modularity inherent in Docker, this approach facili-

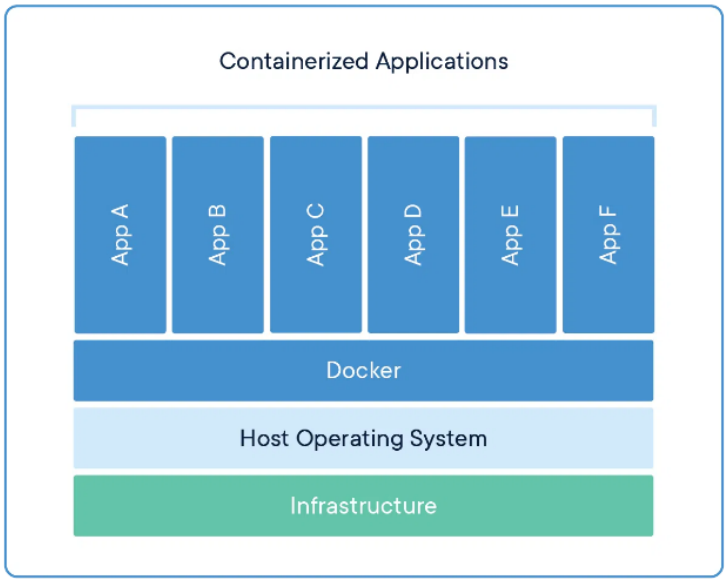

Figure 4: Architecture of containerization [Doc]

tated the creation of discrete and specialized containers tailored to the unique requirements of each software component. The ensuing Docker images, generated from the aforementioned Dockerfiles, encapsulated a self-contained and standardized representation of the respective software, fostering consistency across development, testing, and production environments. This modular containerization strategy not only streamlined the overall development process but also significantly enhanced the maintainability and scalability of the Synthetic Test Environment by decoupling individual software components, affording greater flexibility in terms of updates, scaling, and resource allocation.

7.3.2 Orchestration

Kubernetes stands as a open-source container orchestration platform, meticulously designed to optimize the deployment, scaling, and administration of containerized applications. Kubernetes offers a sophisticated solution for automating the orchestration, load balancing, and self-healing aspects of containerized workloads. Its capabilities, including declarative configuration and automated scaling, empower developers to concentrate on application development, alleviating concerns associated with the intricacies of managing distributed systems. The general architecture of Kubernetes is shown in Figure 5:

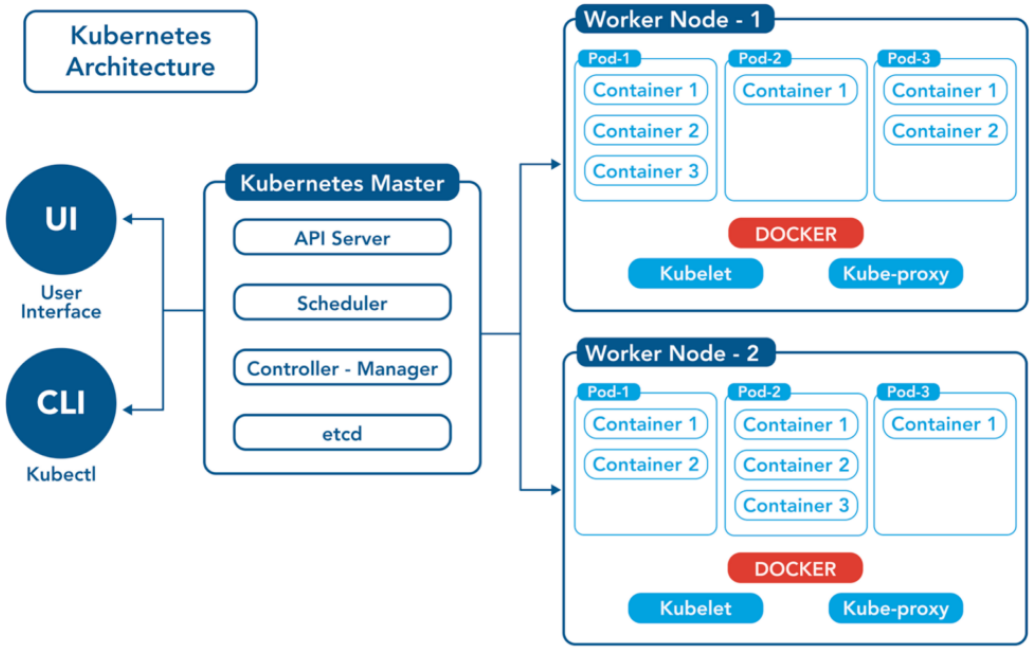

Figure 5: Kubernetes architecture [Ops]

The Kubernetes environment has been deployed on OpenStack. Comprising a bastion node for secure access, a master node for overseeing the cluster's state, and a worker node for executing application workloads, this deployment utilizes OpenStack's capabilities for fast deployment and testing of the environment on virtual machines. The Figure 6 shows the test architecture of Kubernetes on OpenStack:

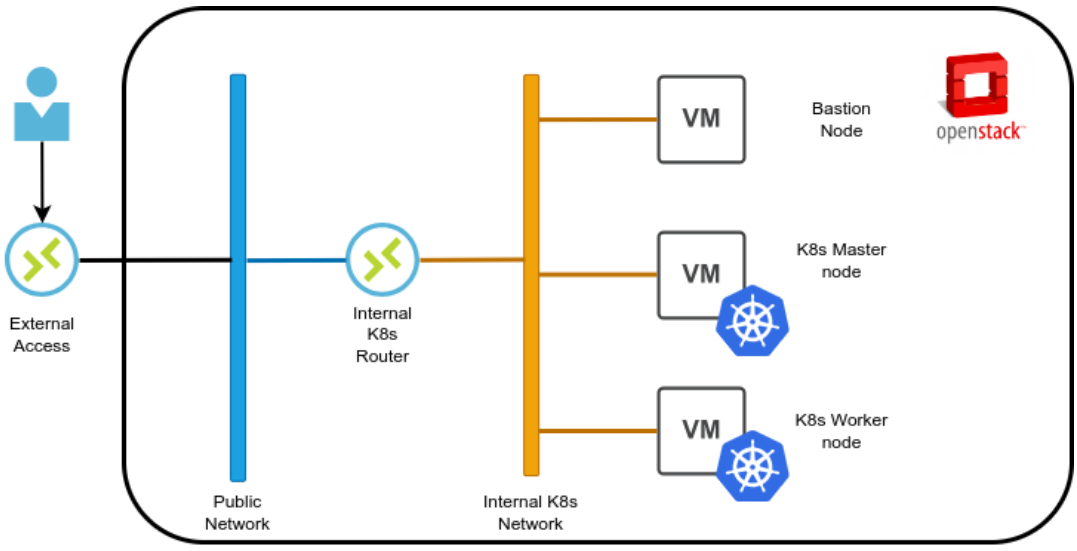

Figure 6: Kubernetes cluster architecture on OpenStack for Synthetic Test Environment

The deployment process was orchestrated seamlessly through the use of Terraform and Kubespray. Terraform, with its infrastructure-as-code approach, enabled the definition and creation of virtual machines, network configurations, and the establishment of port permissions. This allowed for a systematic and repeatable deployment, adhering to best practices in infrastructure management.

This ensured a solid foundation for the Kubernetes cluster.

The orchestration of the Kubernetes cluster was executed by using Kubespray. This tool, designed for deployment at scale, automated the installation and configuration of the cluster components. Through Kubespray, the bastion node, master node, and worker node were seamlessly integrated into a cohesive and functional Kubernetes environment on the OpenStack infrastructure.

7.4 Data

Data is essential in the DECICE project for both the Digital Twin and the AI scheduler. The DT relies on data to accurately represent the current state of the system they are modeling. The AI schedulers use data to learn about the system's behavior and to generate optimal schedules.

Without data, the DT would be unable to provide accurate insights into the system's performance. This would make it difficult to identify and address potential problems, and to optimize the system's operation. Similarly, without data, AI schedulers would be unable to generate effective schedules. This could lead to bottlenecks such as overloaded nodes because of wrong scheduling.

7.4.1 Data Collection

The expectation from the AI scheduler in the DECICE project is to make a job scheduling as accurate as possible. The AI scheduler can learn to identify patterns in job resource usage and deployment patterns with valuable data. This information can then be used to make more informed decisions about how to schedule jobs and deployments to optimize resource utilization and performance. Additionally, the AI scheduler can learn to identify and respond to changes in the Kubernetes environment, such as changes in the number of available nodes, changes in the resource demands of jobs, and changes in the network topology. This information can be used to keep the Kubernetes environment running smoothly and efficiently. To achieve this goal, a well-organized and accurate data is needed.

In order to train the AI scheduler to schedule jobs and deployments in Kubernetes, a small portion of the MIT Supercloud dataset was utilized to train the first version of the AI scheduler. This dataset contains approximately 2TB of detailed monitoring logs from the MIT Supercloud system, including CPU and GPU usage by jobs, memory usage, file system logs, and physical monitoring data. The dataset was released in 2021 to foster innovative AI/ML approaches to the analysis of large-scale HPC and datacenter/cloud operations [Sam+21]. There are also different datasets that are publicly available such as Alibaba Cluster Trace Program [Ali] and Google Cluster Usage Traces [Wil]. However, compared to the MIT Supercloud dataset, these datasets do not provide as much precise information about I/O, resource allocation, and batch workloads. Accessing and processing the necessary data is also made easier by the MIT Supercloud dataset. For this reason, it was selected for the AI scheduler's initial iteration.

Training the AI scheduler on the MIT Supercloud dataset allows it to learn how to efficiently schedule jobs and deployments in Kubernetes, leveraging the richness of data and insights within the dataset for the first iteration of the training. This can lead to improved performance, reliability, and cost-effectiveness of the Kubernetes environment.

7.4.2 Data Processing

As described in Section 7.4.1 above, the MIT SuperCloud dataset was employed to provide real-world data for training and evaluation purposes. The dataset was carefully analyzed and cleaned to extract relevant parameters, including job ID, CPU and memory requirements, core count, total node memory and so on. These extracted parameters were then converted from CSV to JSON format to facilitate data storage and manipulation. The JSON data was subsequently stored in the Digital Twin. Some of these parameters that have been obtained from the MIT dataset are depicted in Table 3 below and an associated JSON schema can be found in appendix A.

Parameter	Explanation	Example Value
job_id	The ID of the job	74426695482326
cpus_req	CPU requirement of the job	5
mem_req	Memory requirement of the job in MB	15360
time_run	Total time elapsed during job run	415
node_id	The ID of the node	r8015356-n976057
cpu_cores	The number of total cores in the node	48
memory	The total amount of memory in the node in MB	393216
cpu_load	Average CPU load on the node	3.62
memory_load	Average memory load on the node	4.08

Table 3: Parameters that were obtained and converted from the MIT SuperCloud dataset

Once job requirements and node information are sorted, the scheduler controller sends the data to the AI scheduler for training. The AI scheduler utilizes this training data to refine its scheduling algorithms, enabling it to generate more efficient and optimized scheduling predictions.

8 References

[Ali] Alibaba. *Alibaba Cluster Trace Program — github.com/alibaba/clusterdata*. https://github.com/alibaba/clusterdata. [Accessed 17-11-2023].

[DEC] DECICE. *Device Edge Cloud Intelligent Collaboration framEwork — github.com/DECICE-project*. https://github.com/DECICE-project. [Accessed 17-11-2023].

[Doc] Docker. *What is a Container? — docker.com*. https://www.docker.com/resources/what-container/. [Accessed 13-11-2023].

[Hug] Hugo. *The World's Fastest Framework for Building Websites — gohugo.io*. https://gohugo.io/. [Accessed 17-11-2023].

[Moz] MozDevNet. *Crud - MDN web docs glossary: Definitions of web-related terms: MDN*. [Accessed 13-11-2023]. URL: https://developer.mozilla.org/en-US/docs/Glossary/CRUD.

[Ops] OpsRamp. *An Overview of Kubernetes Architecture - OpsRamp — opsramp.com*. https://www.opsramp.com/guides/why-kubernetes/kubernetes-architecture. [Accessed 13-11-2023].

[Ram] Sebastián Ramírez. *FastAPI Framework, High Performance, Easy to Learn, Fast to Code, Ready for Production — fastapi.tiangolo.com*. https://fastapi.tiangolo.com/. [Accessed 17-11-2023].

[RB01] Daniela M. Romano and Paul Brna. "Presence and Reflection in Training: Support for Learning to Improve Quality Decision-Making Skills under Time Limitations". In: *CyberPsychology & Behavior* 4.2 (Apr. 2001), pp. 265–277. DOI: 10.1089/109493101300117947. URL: https://doi.org/10.1089/109493101300117947.

[Ros+00] F. D. Rose et al. "Training in virtual environments: transfer to real world tasks and equivalence to real task training". In: *Ergonomics* 43.4 (Apr. 2000), pp. 494–511. DOI: 10.1080/001401300184378. URL: https://doi.org/10.1080/001401300184378.

[Sam+21] Siddharth Samsi et al. "The MIT Supercloud Dataset". In: *CoRR* abs/2108.02037 (2021). arXiv: 2108.02037. URL: https://arxiv.org/abs/2108.02037.

[TWE18] Rajeeth Tharma, Roland Winter, and Martin Eigner. "An approach for the implementation of the digital twin in the automotive wiring harness field". In: (2018). DOI: 10.21278/idc.2018.0188. URL: https://doi.org/10.21278/idc.2018.0188.

[Wil] J. Wilkes. *Clusterdata 2019 Traces — github.com/google/cluster-data*. https://github.com/google/cluster-data. [Accessed 17-11-2023].

A JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Json Schema for NodePool",
  "type": "object",
  "properties": {
    "id": {
      "type": "string"
    },
    "cpu_cores": {
      "type": "number"
    },
    "memory": {
      "type": "number"
    },
    "cpu_load": {
      "type": "number"
    },
    "memory_load": {
      "type": "number"
    }
  },
  "required": [
    "id",
    "cpu_cores",
    "memory",
    "cpu_load",
    "memory_load",
  ]
}
```