DECICE

DEVICE-EDGE-CLOUD INTELLIGENT COLLABORATION FRAMEWORK

Grant Agreement: 101092582

D2.2 Digital Twin

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:	D2.2
Deliverable title:	Digital Twin
Deliverable version:	1.0
Work Package number:	WP2
Work Package title:	AI Scheduler for Optimization and Adaption
Responsible partner	University of Bologna (UNIBO)
Due Date of delivery:	2023-11-30
Actual date of delivery:	2023-11-30
Dissemination level:	PU
Type:	OTHER
Editor(s):	Mohsen Seyedkazemi Ardebili (UNIBO) Andrea Bartolini (UNIBO)
Contributor(s):	
Reviewer(s):	Dirk Pleiter (KTH) Riccardo Cavadini (E4)
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	2023-11-14	UNIBO	Initial Draft
0.2	2023-11-17	UNIBO	Second Draft
0.3	2023-11-25	E4/KTH	Review
1.0	2023-11-28	UNIBO	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

This deliverable describes the implementation of the DECICE Digital Twin core components. The deliverable describes the initial implementation of the Digital Twin and the interfaces for interconnecting with other DECICE components, showcasing its core elements. It delineates the various technological options applicable to different facets of the Digital Twin, offering a nuanced understanding of its capabilities. A detailed examination of data flow between these subcomponents is also presented.

Contents

1 Purpose and Scope of the Deliverable 6

2 Abstract / publishable summary 6

3 Project objectives 7

4 Changes made and/or difficulties encountered, if any 7

5 Sustainability 7

6 Dissemination, Engagement and Uptake of Results 7
6.1 Target audience . . . 7
6.2 Record of dissemination/engagement activities linked to this deliverable . . . 8
6.3 Publications in preparation OR submitted . . . 8
6.4 Intellectual property rights resulting from this deliverable . . . 8

7 Detailed report on the deliverable 10
7.1 Overview of Digital Twin Architecture . . . 10
7.1.1 Monitoring System . . . 11
7.1.2 AI-Scheduler . . . 12
7.1.3 Synthetic Test Environment . . . 12
7.1.4 GitHub . . . 12
7.2 Digital Twin Core . . . 13
7.2.1 Data Collector Component . . . 13
7.2.2 Data Insertion Component . . . 13
7.2.3 Data Querying Component . . . 15
7.2.4 Data Harmonization Component (DHC): Transforming Diverse JSON Files to Standard Format . . . 16
7.3 Digital Twin Modules . . . 16
7.3.1 HTTP Communication API Server . . . 17
7.3.2 Time Series Database . . . 18
7.3.3 ML Modules . . . 20
7.4 Deployment in Kubernetes . . . 20
7.4.1 Prerequisites . . . 20
7.4.2 Steps . . . 20
7.5 Development and Test Environment . . . 22

8 References 23

A Digital Twin Output JSON file 24

1 Purpose and Scope of the Deliverable

The aim of Deliverable 2.2: Digital Twin is to provide an overview of the first implementation of the digital twin and its subcomponents. It also introduces some technical details and technologies that can be utilized for the future development of the digital twin.

2 Abstract / publishable summary

Deliverable 2.2, titled "Digital Twin," begins with a brief review of key definitions associated with the digital twin, such as the monitoring system, AI-scheduler, and virtual training environment. This document then describes the digital twin implementation, encompassing both the digital twin core and digital twin modules.

This deliverable highlights the technologies used in various subcomponents. Additionally, it explores alternative possibilities that can be seamlessly integrated, if necessary, to enhance compatibility with other DECICE components, such as the AI-scheduler and virtual training environments.

Accompanying this deliverable are GitHub repositories containing source codes and the initial implementation of the digital twin. To facilitate a clearer understanding, the document includes examples, excerpts of code, and, in certain instances, procedural steps for the implementation or conversion of the digital twin to the Kubernetes ecosystem.

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. See Table 1.

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 showcases the initial implementation of the digital twin, a key component in DECICE.
(O2) Develop a scheduler supporting dynamic load balancing for energy-efficient compute orchestration, improved use of green energy, and automated deployment.	The Digital Twin contains aggregated metrics and system settings that are relevant for scheduling, in JSON format.
(O3) Design and implement an API that increases control over network, computing and data resources.	The Digital Twin develops an API server to provide data for other DEVICE components.
(O4) Design and implement a Dynamic Digital Twin of the system with AI-based prediction capabilities as integral part of the solution.	The deliverable showcases the initial implementation of the digital twin and explains how these components can be potentially improved.
(O5) Demonstrate the usability and benefits of the DECICE solution for real-life use cases.	Providing data and ML model outputs to other components for optimization purposes enhances 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.	Deploying and porting the digital twin to the Kubernetes ecosystem, addressing security concerns prioritized.

Table 1: Project Macro-objectives and contributions of this deliverable

4 Changes made and/or difficulties encountered, if any

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

5 Sustainability

The design and implementation of Digital Twin 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 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 2.

6.3 Publications in preparation OR submitted

See Table 3.

6.4 Intellectual property rights resulting from this deliverable

None.

Type of dissemination and communication activities	Details	Date and location of the event	Type of audience activities	Zenodo Link	Estimated number of persons reached
News Article	DT News Article on the DECICE Website	-	PU	Link	-
Twitter Post	DT Twitter Post	26.09.2023	PU	Link	70 Impressions
LinkedIn Post	DT LinkedIn Post	26.09.2023	PU	Link	271 Impressions
News Article	Unibo News Article on the DECICE Website	-	PU	Link	-
Twitter Post	Unibo Twitter Post	06.06.2023	PU	Link	135 Impressions
LinkedIn Post	Unibo LinkedIn Post	06.06.2023	PU	Link	427 impressions
News Article	NAG News Article on the DECICE Website	-	PU	Link	-
Twitter Post	NAG Twitter Post	18.07.2023	PU	Link	296 Impressions
LinkedIn Post	NAG LinkedIn Post	18.07.2023	PU	Link	1231 Impressions

Table 2: 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	0

Table 3: Publications related to this deliverable

7 Detailed report on the deliverable

This section documents the requirements for the Dynamic Digital Twin. There are various definitions of Digital Twins (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 spans 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). For more information about the key definitions of Digital Twin and the DEVICE Digital Twin Requirements, please refer to deliverable *D2.1 Specification Of The Optimization Scope*.

The remainder of the section is structured as follows: an overview of the Digital Twin Architecture, Digital Twin Core, Digital Twin Modules, considerations for Deployment in Kubernetes, and the Development and Test Environment. This structured approach aims to provide a clear and systematic understanding of the key components and aspects related to the digital twin implementation within the context of the project.

7.1 Overview of Digital Twin Architecture

Figure 1 depicts the interconnected subcomponents of the digital twin, which include the Digital Twin Core, as well as three digital twin modules: the HTTP Communication API Server, the Machine Learning Modules, and the time series database.

The Digital Twin Core serves as the central hub, collecting data from the monitoring system. After preprocessing, it disseminates the processed data to both the http communication API server and the time series database, facilitating seamless data flow between the components. Within the digital twin core, multiple subcomponents are employed to efficiently gather data from the monitoring system. These subcomponents further refine the collected data for insertion into the database or transmission to the http communication API server.

The HTTP Communication API Server provides essential tools and patterns for developing RESTful APIs (Representational State Transfer API). This approach simplifies communication between diverse systems over HTTP, ensuring a standardized and straightforward exchange of data.

The digital twin system also integrates machine learning modules, enhancing its capabilities by facilitating the composition of different tools. These modules empower the system with intelligent data analysis, predictive modeling, and decision-making capabilities.

Additionally, the system incorporates a time series database, specifically tailored to manage time-stamped or time-series data. This specialized database management system optimizes the storage, retrieval, and processing of data points, making it invaluable for applications dealing with temporal data.

The digital twin is connected to the monitoring system (metrics storage) and interfacing with its database to aggregate monitoring data. The Virtual Training Environment (VTE) Controller has the capability to trigger a wrapper, prompting it to load specific timestamps or updates into the Digital Twin. The AI-Scheduler accesses the Digital Twin to retrieve the current cluster state in JSON format. Within the high-level architecture of the DECICE framework, there is an anticipated connection between the DECICE Controller Manager and the digital twin for triggering data updates. It is important to note that this connection is not yet implemented in the current version of the digital twin, but it is planned for future iterations. In the diagram, the "OUT" represents the GUI and querying capabilities of the database that can be accessed from outside the digital twin.

In the framework of the DECICE MODEL ARCHITECTURE and D2.3 AI Scheduler Prototypes for Storage and Compute, the digital twin functions as a passive element, poised to initiate the data collection process upon cues from the DECICE control manager or VTE Controller trigger. It is noteworthy that the current iteration exhibits a relatively limited capability in terms of data preprocessing within the model architecture. This limitation is acknowledged, and our vision is to fortify this aspect significantly in the forthcoming versions of both implementations and documentation of AI-Schduler, VTE and Digital Twin.

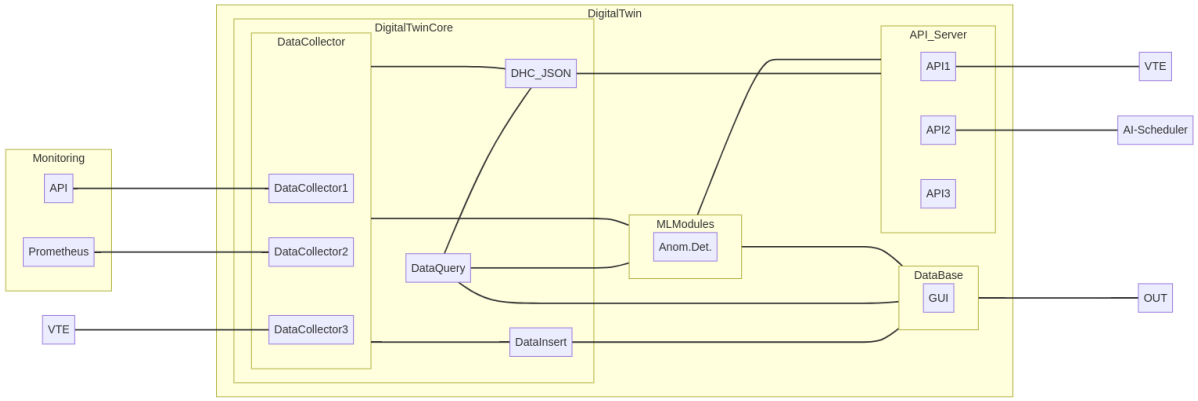

Figure 1: Digital Twin: Its Subcomponents and Interconnections

7.1.1 Monitoring System

In the DECICE project's monitoring system, Prometheus is employed as a crucial tool. Prometheus [Pro23] is an open-source monitoring and alerting toolkit known for its effectiveness in collecting and storing time-series data. This toolkit enables users to seamlessly query and visualize performance metrics. Specially designed for cloud-native environments, Prometheus offers vital insights into the

health and performance of both applications and infrastructure. Utilizing a pull-based model for data collection, Prometheus boasts a robust query language, PromQL, and integrates seamlessly with container orchestration systems such as Kubernetes. For more detailed information about the monitoring system, please refer to the D2.1 Specification Of The Optimization Scope.

7.1.2 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.

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 statistics, hardware characteristics and network. For more detailed information about the AI-Scheduler, please refer to the D2.1 Specification Of The Optimization Scope.

7.1.3 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 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 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]. For more detailed information about the VTE, please refer to the deliverable *D2.1 Specification Of The Optimization Scope*.

7.1.4 GitHub

The source codes and the initial implementation of the digital twin can be found on the DECICE GitHub project page: `https://github.com/DECICE-project`.

7.2 Digital Twin Core

The Digital Twin Core comprises various Python-based subcomponents, including the Data Collector, Data Insertion, Data Querying, and Data Harmonization components (designed to transform diverse JSON files into a standardized format). Functioning as the central hub, the Digital Twin Core utilizes these components to collect data from the monitoring system. Preprocessing, it seamlessly disseminates the processed data to both the HTTP communication API server and the time series database, ensuring smooth data flow across the system. Within the Digital Twin Core, multiple subcomponents efficiently gather and refine data from the monitoring system, enabling effective insertion into the database or transmission to the HTTP communication API server. In the following sections, this document provides more details about each subcomponent of the Digital Twin Core.

7.2.1 Data Collector Component

The Data Collector Component is a crucial module that is responsible for gathering datasets from the monitoring system. This Python code defines a function that retrieves JSON data from a specified API endpoint using the requests library. The requests library simplifies the process of sending HTTP requests and handling responses efficiently, making data retrieval seamless. The function includes robust error handling mechanisms to ensure smooth operation, even in unfavorable conditions. It gracefully manages various types of exceptions that may occur during the API request, providing a reliable and resilient data collection solution.

The Data Collector Component not only collects data from the API of the monitoring system but also includes subcomponents that can query the monitoring system's database and also collect data from VTE.

Given that various components of the DECICE framework are currently in the developmental phase, it is imperative to note that this component will require updates to align with the finalized versions of the monitoring system API, databases, and the VTE API. As development progresses across the framework, ongoing adjustments to this component will be necessary to ensure seamless integration and compatibility with the evolving framework.

7.2.2 Data Insertion Component

This component is responsible for inserting data into the digital twin time series database. It is important to note that the specifics may vary depending on the database being used. However, in general, there are several important considerations that should be taken into account to ensure a successful and efficient process. Here are some key items to consider:

Connection Setup: Establish a connection to the time-series database using appropriate Python libraries (e.g., InfluxDBClient for InfluxDB, influxdb library). Provide connection parameters such as host, port, username, and password.

Data Preparation: Format the data in accordance with the time-series database schema. Ensure that data types align with the database schema requirements.

Timestamps: Ensure that timestamps are in the correct format. Consider time zone issues and maintain consistency.

Batching and Bulk Insert: Optimize performance by batching data for insertion. Many time-series databases offer bulk insert methods; utilize them for improved efficiency.

Error Handling: Implement robust error handling to manage potential issues during data insertion. Log errors for diagnostic purposes.

Data Validation: Validate the integrity of the data before insertion to prevent inconsistencies.

Indexing: Consider database-specific indexing strategies for faster retrieval. Utilize timestamp-based indexing if applicable.

Security: If applicable, ensure that your database connection is secure. Avoid hardcoding sensitive information in the code.

Compression and Optimization: Explore database-specific options for data compression. Optimize data for storage efficiency.

Database-specific Considerations: Be aware of any specific requirements or features of the chosen time-series database (e.g., retention policies, downsampling).

Testing: Perform thorough testing of the data insertion process with a representative dataset. Consider automation for testing as part of your development workflow.

Initial Implementation In the preliminary implementation of the digital twin, the InfluxDB database was selected. Some of the data insertion items mentioned above (such as Connection Setup, Timestamps, Indexing, Database-specific Considerations) were implemented in the initial implementation. However, other items (such as Batching and Bulk Insert, Error Handling, Data Preparation, Data Validation, Compression and Optimization, Testing) need to be implemented, taking into consideration the monitoring system and monitoring data. It is important to reevaluate and incorporate all of these items in future versions, based on the requirements of the final version monitoring system, AI-scheduler, and VTE.

The data insertion component, implemented as a Python module, encompasses the creation of a client object. This object, when instantiated, establishes a connection with the InfluxDB database as specified by the provided URL. Serving as a pivotal gateway, the client object facilitates seamless interaction with the InfluxDB database, empowering essential operations such as data insertion, querying, and the management of database configurations. This modular approach enhances the adaptability and functionality of the digital twin's data handling capabilities within the InfluxDB environment.

```
client = influxdb_client.InfluxDBClient(url=URL, token=INFLUXDB_TOKEN, org=ORG)
```

Here's a breakdown of the parameters utilized when creating the InfluxDBClient object:

URL: This parameter represents the URL of the InfluxDB instance. Typically, it includes the protocol (either HTTP or HTTPS), hostname, and port number where the InfluxDB service is running.

Token: The authentication token is crucial for verifying and authorizing requests with the InfluxDB instance. Tokens serve the purpose of authentication and authorization, ensuring secure interactions between the client and the database.

Organization (org): The organization parameter specifies the name of the organization to which the InfluxDB client is affiliated. In InfluxDB, data is structured within organizations, allowing efficient management and segregation of data based on different contexts or use cases.

Once the client object is initialized with these parameters, it becomes a powerful tool for managing the InfluxDB database. It facilitates essential operations, such as inserting data points into the database, querying existing data, and configuring the database settings.

Additionally, the data insertion component performs the crucial task of converting collected data into the appropriate data schema. This schema ensures compatibility and accuracy when inserting data into the database, making the data insertion process seamless and efficient.

7.2.3 Data Querying Component

This component is entrusted with executing data queries on the timeseries databases of the digital twin.

The Data Querying component, realized as a Python module, empowers users to proficiently query, retrieve, and manipulate data from databases. Similar to the data insertion process, these queries leverage the Python database client library, facilitating seamless interaction with the underlying database infrastructure.

When formulating InfluxDB queries in Python, an array of parameters and functions becomes available, offering extensive capabilities for data filtering and manipulation. Some commonly utilized options include:

- `from(bucket: "bucket_name")`: Specifies the target bucket from which data is retrieved.
- `|> range(start: -1h)`: Filters data based on a specific time range; for example, selecting data from the last hour (1h).
- `|> filter(fn: (r) => r._measurement == "cpu_usage")`: Filters data based on defined conditions, such as specific measurement names. The condition inside the filter function can be modified to match various fields or values.
- `|> group(columns: ["tag_name"])`: Groups data based on specified tag columns.
- `|> aggregateWindow(every: 1h, fn: mean, createEmpty: false)`: Aggregates data over specified time windows, calculating metrics like mean values. This function is particularly valuable for time-series data analysis.

```
client = influxdb_client.InfluxDBClient(url=URL, token=INFLUXDB_TOKEN, org=ORG)
# Example query
query = 'from(bucket: "BUCKET_NAME")")
|> range(start: -1h)
|> filter(fn: (r) => r._measurement == "cpu_usage")'

result = query_api.query(query, org=ORG)
```

Listing 1: Data Querying Example

These flexible querying capabilities empower users to extract valuable insights from the digital twin's timeseries databases, enhancing the analytical prowess of the system.

7.2.4 Data Harmonization Component (DHC): Transforming Diverse JSON Files to Standard Format

The Data Harmonization Component plays a crucial role in this system, responsible for converting disparate JSON file or objects with varying JSON schemas into a unified format compatible with both the VTE and AI-scheduler modules. Before disseminating JSON objects to other components within the framework, this component rigorously validates their JSON schemas, ensuring data integrity and consistency.

Moreover, the Data Harmonization Component includes functionalities designed to preprocess the data effectively. One essential task involves aligning measurements captured at different timestamps. Notably, these measurements might come with different timestamps, creating the need for precise time alignment. Our component seamlessly handles this challenge, ensuring a coherent and synchronized dataset.

By harmonizing data into a standardized JSON schema and aligning timestamps, the Data Harmonization Component ensures that the VTE and AI-scheduler modules receive consistent and properly formatted data. This uniformity is essential for accurate analysis, enabling our system to make informed decisions and deliver optimal performance.

The appendix A contains a segment of the output JSON file generated by our system. This JSON file follows a standardized format, which ensures consistency and makes it easier for downstream applications to interpret.

7.3 Digital Twin Modules

In complement to the Digital Twin Core, additional Digital Twin modules extend the functionality of the system. These modules encompass ML-based tools, API Server, databases, and a graphical user interface (GUI), contributing enhanced capabilities to the Digital Twin. This section provides technical insights into these modules, outlining specific details related to their implementation and functionality. This section explores potential functionalities, emphasizing that module development is ongoing. In the future, we remain adaptable to emerging needs, allowing for the incorporation of additional modules or tools. This flexibility extends to the integration of diverse AI models into the machine learning modules as required.

7.3.1 HTTP Communication API Server

HTTP Communication API Server is a server application that facilitates communication over the HypeTtext Transfer Protocol (HTTP) using Application Programming Interfaces (APIs). HTTP is the foundation of data communication on the World Wide Web. It is an application protocol for distributed, collaborative, and hypermedia information systems. An API is a set of rules and protocols that allows different software applications to communicate with each other. In HTTP API, it means defining how clients can make HTTP requests (using methods like GET, POST, PUT, DELETE) and how the server will respond with data (usually in formats like JSON). The server receives HTTP requests from clients, processes them, and sends back appropriate HTTP responses containing the requested data.

Initial Implementation - Flask For the initial implementation, Flask [Fla23] was chosen as the framework. However, it's worth noting that alternative options, such as FAST API, exist. Flask is a lightweight and flexible Python web framework that provides tools, libraries, and technologies for building web applications. It is classified as a micro-framework because it does not require particular tools or libraries. It has no database abstraction layer, form validation, or any other components where pre-existing third-party libraries provide common functions. However, Flask supports extensions that can add application features as if they were implemented in Flask itself. Flask is often used for building small to medium-sized web applications, APIs, and prototypes.

RESTful API, on the other hand, is an architectural style for designing networked applications. It stands for Representational State Transfer and relies on a stateless, client-server communication model. RESTful APIs use HTTP methods (such as GET, POST, PUT, DELETE) to perform operations on resources (such as data objects), and these resources are identified by URIs (Uniform Resource Identifiers). RESTful APIs are designed to be simple, scalable, and stateless, making them suitable for use in web applications and services.

Flask provides support for building RESTful APIs through its flexible routing system and the use of HTTP methods. Developers can define routes and associate them with Python functions, which are executed when a specific endpoint is accessed with a corresponding HTTP method. This makes it easy to create APIs for web applications, allowing them to handle various client requests and responses efficiently.

Flask was chosen due to its lightweight and flexible micro-framework, making it an ideal choice for our project. Its simplicity and ease of use are particularly advantageous for developing APIs in the context of AI projects. Additionally, Flask benefits from a large and active community, providing valuable support and resources. It's worth noting that while Flask offers simplicity, it follows a philosophy of limited built-in features, allowing for greater customization and adaptability tailored to the specific needs of DECICE project.

For future versions of the digital twin, we plan to utilize the OpenAPI specification to describe and document digital twin APIs. It allows developers to define the structure of their API, including endpoints, operations, request and response formats, authentication methods, and more. This approach will enhance the interoperability and integration capabilities of the digital twin. In this context, FastAPI, a modern web framework that adheres to the OpenAPI standard and supports

asynchronous operations, is also under consideration. This framework could potentially expedite the development process and improve the performance of our APIs.

7.3.2 Time Series Database

A time series database is a specialized type of database designed to efficiently and effectively store, retrieve, and manage time-stamped or time-series data. Time-series data consists of data points collected or recorded over successive time intervals. This type of data is prevalent in various domains, including finance, IoT (Internet of Things), monitoring systems, scientific research, and more. Key features of time series databases include Time-Stamped Data Storage, Time-Based Indexing, Aggregation and Analysis, Scalability, Data Compression.

Given that the monitoring system incorporates Prometheus as the default time series database, the inclusion of timeseries database in the Digital Twin becomes optional. However, it is important to consider the scalability of Prometheus, as challenges may arise in certain scenarios, especially in large and dynamic environments. To address these concerns and to facilitate prolonged and large-scale analysis, the exploration of alternative time-series databases is encouraged. Potential options include Victoriametrics [Vic23], Apache Druid [Apa23b], Cassandra [Apa23a] with KairosDB [Kai23], each offering distinctive features for extended data retention and scalable operations. For the initial implementation, InfluxDB [Inf23] was selected due to its suitability for: (i) Efficient storage and retrieval of time-series data. (ii) Flexibility in handling varying data schemas. (iii) Robust support for high write throughput scenarios.

Initial Implementation - InfluxDB InfluxDB, an open-source time-series database, excels in managing high write and query loads for time-series data. Its organizational structure includes databases, retention policies, measurements, tags, fields, and timestamps. Below is an elucidation of the key components within InfluxDB's data schema. To tailor data collection for monitoring computing nodes—capturing metrics such as memory usage, CPU usage (CPI), and core numbers— this structured approach is followed:

Database: Establish a database, e.g., "monitoring_data," to house all monitoring data.

Measurement: For each type of data to be collected, create a corresponding measurement. Examples include "memory_usage," "cpu_usage," and "core_number," with each measurement representing a distinct data type.

Tags: Enhance measurements with metadata by using tags. Commonly employed in WHERE clauses, tags can differentiate data points. Tagging with, for instance, "node_id," distinguishes data belonging to specific computing nodes. Tags, being indexed, contribute to improved query performance.

Fields: Store actual measurement values in fields. For "memory_usage," a relevant field might be "mem_usage_percent," for "cpu_usage," it could be "cpu_usage_percent," and for "core_number," consider "number_of_cores." Fields accommodate numeric, string, or boolean values, though they are not indexed.

Timestamp: Attach a timestamp to each data point, indicating when the data was collected. In InfluxDB, timestamps are typically in Unix time format (nanoseconds since January 1, 1970, UTC).

This structured approach ensures an organized and efficient setup for monitoring data within InfluxDB, enhancing the database's capability to handle diverse metrics from computing nodes.

Retention Policies: Define retention policies based on how long you want to retain your data. For example, you might want to keep high-resolution data for a short period and lower resolution data for a longer period. You can define different retention policies for different time durations.

Here's an example of how you can structure the data collection:

Measurement 1: memory_usage

Tags: node_id (e.g., node1, node2, ..., node10) Fields: mem_usage_percent (e.g., 70.5) Timestamp: Unix timestamp Retention Policy: Long Term

Measurement 2: cpu_usage

Tags: node_id (e.g., node1, node2, ..., node10) Fields: cpi_usage_percent (e.g., 50.2) Timestamp: Unix timestamp Retention Policy: Long Term

Measurement 3: core_number

Tags: node_id (e.g., node1, node2, ..., node10) Fields: number_of_cores (e.g., 8) Timestamp: Unix timestamp Retention Policy: Long Term

Structuring the data in this manner enables the database to adeptly store and query monitoring data for your computing nodes using InfluxDB. It's important to note that the precise structure may vary depending on the unique characteristics of your use case and specific requirements.

Installation Steps for InfluxDB on Kubernetes:

Create a Namespace for InfluxDB

```
kubectl create namespace influxdb
kubectl config set-context --current --namespace=influxdb
```

Create a ConfigMap for InfluxDB Configuration

```
kubectl apply -f influxdb-config.yaml
```

Create a Persistent Volume for InfluxDB

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: digital-twin-pv
  namespace: digital-twin
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
Persistent Volume Claim
apiVersion: v1
```

```
kind: PersistentVolumeClaim
metadata:
  name: digital-twin-pvc
  namespace: digital-twin
spec:
    accessModes:
        - ReadWriteOnce
    resources:
        requests:
        storage: 1Gi
```

Listing 2: Persistent Volume

```
kubectl apply -f influxdb-pv.yaml
```

Deploy InfluxDB using a StatefulSet

```
kubectl apply -f influxdb-statefulset.yaml
```

Access InfluxDB: To interact with InfluxDB externally, consider exposing it through a Service with either NodePort or LoadBalancer, depending on the specifics of the environment.

```
kubectl apply -f influxdb-service.yaml
```

7.3.3 ML Modules

The role of this component is to store and utilize various ML models, Python tools, libraries, and modules. It leverages real-time monitoring data to forecast future outcomes or extract features for non-measurable parameters. These tools are designed to generate new features and insights, including functions like anomaly detection, prediction of energy and performance efficiency, and tools for predicting power consumption. The development of these tools is still ongoing.

7.4 Deployment in Kubernetes

This section provides a step-by-step guide on deploying Python code in a Kubernetes environment. The deployment of certain components, such as InfluxDB, in Kubernetes has been successfully completed, and the corresponding deployment YAML files are available in GitHub. However, as development progresses on other components, they are currently in the development phase. Once development is concluded, these components will undergo the necessary adaptations for deployment within the Kubernetes environment. The procedure for adapting the Python application that has been developed to the Kubernetes environments is outlined below.

7.4.1 Prerequisites

Ensure the following prerequisites are met before proceeding: (i) A working Kubernetes cluster. (ii) Docker installed on your local machine for building container images.

7.4.2 Steps

1. Dockerize Python App Write a Dockerfile to package your Python application. Use a base image with Python installed and copy your code and dependencies into the image.

```
FROM python:3.10
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
CMD ["python", "app.py"]
```

Listing 3: Example Dockerfile

2. Build and Push Docker Image Build the Docker image using the following command:

```
docker build -t decice-registry/app:latest .
```

Push the image to a container registry:

```
docker push decice-registry/app:latest
```

3. Create Kubernetes Deployment YAML Write a Kubernetes Deployment YAML file specifying the Docker image, replicas, and other settings.

Example deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: app
  template:
    metadata:
      labels:
        app: app
    spec:
      containers:
      - name: app
        image: decice-registry/app:latest
        ports:
        - containerPort: 5000  # Adjust as per your app
```

Listing 4: Deployment YAML Example

4. Apply Deployment to Kubernetes Cluster Apply the Deployment YAML using the following command:

```
kubectl apply -f deployment.yaml
```

5. Expose Service (Optional) If your app needs external access, create a Kubernetes Service and expose it.

Example service.yaml:

```
apiVersion: v1
kind: Service
```

```
metadata:
  name: app-service
spec:
  selector:
    app: your-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5000  # Match with containerPort in deployment.yaml
  type: LoadBalancer
```

Listing 5: YAML Example

Apply the Service YAML:

```
kubectl apply -f service.yaml
```

6. Scale and Manage Adjust the number of replicas using the following command:

```
kubectl scale deployment app --replicas=5
```

Monitor and manage your application using Kubernetes commands. The application is now deployed and accessible within the Kubernetes cluster.

Kubernetes Namespaces In Kubernetes, namespaces act as partitions, segregating cluster resources for distinct users, teams, or projects. They establish a naming scope, requiring resource names to be unique within a namespace but not across namespaces. Essentially, a namespace functions as a virtual cluster within Kubernetes. These namespaces serve as a robust organizational tool, delivering advantages such as isolation, resource quotas, access control, simplified resource management, and support for multi-tenancy.

```
kubectl create namespace digital twin
```

7.5 Development and Test Environment

To establish the development and testing environment, a compact Kubernetes cluster has been provisioned on the GWDG cloud system. The cluster comprises three nodes, consisting of one control plane and two worker nodes. This setup is designed to facilitate efficient development and rigorous testing within the context 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.

[Apa23a] ApacheCassandra. *ApacheCassandra*. https://cassandra.apache.org/. Accessed: November 2023. 2023.

[Apa23b] ApacheDruid. *ApacheDruid*. Accessed: November 2023. 2023. URL: https://druid.apache.org/.

[Fla23] Flask. *Flask*. https://flask.palletsprojects.com/. Accessed: November 2023. 2023.

[Inf23] InfluxDB. *InfluxDB*. https://www.influxdata.com/. Accessed: November 2023. 2023.

[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.

[Kai23] KairosDB. *KairosDB*. https://kairosdb.github.io/. Accessed: November 2023. 2023.

[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.

[Pro23] Prometheus. *Prometheus*. https://prometheus.io/. Accessed: November 2023. 2023.

[Vic23] VictoriaMetrics. *VictoriaMetrics*. https://victoriametrics.com/. Accessed: November 2023. 2023.

A Digital Twin Output JSON file


```
{
  "lastUpdated": "2023-10-26 19:38:23",
  "nodepools": [
    {
      "nodepool": {
        "id": "1",
        "labels": [
          "nodepool1",
          "ABC"
        ],
        "nodes": [
          {
            "node": {
              "id": "1",
              "labels": [
                "Node 1"
              ],
              "metrics": {
                "cpu-usage-percent": 80,
                "memory-usage-percent": 67
              },
              "hardware": {
                "cpu_cores": 4,
                "memory_in_mb": 8192
              }
            }
          },
          {
            "node": {
              "id": "2",
              "labels": [
                "Node 2"
              ],
              "metrics": {
                "cpu-usage-percent": 70,
                "memory-usage-percent": 26
              },
              "hardware": {
                "cpu_cores": 2,
                "memory_in_mb": 4096
              }
            }
          }
        ]
      }
    },
    {
      "nodepool": {
        "id": "2",
        "labels": [
          "nodepool2",
```

```
          "X"
        ],
        "nodes": [
          {
            "node": {
              "id": "3",
              "labels": [
                "Node 3"
              ],
              "metrics": {
                "cpu-usage-percent": 80,
                "memory-usage-percent": 12
              },
              "hardware": {
                "cpu_cores": 4,
                "memory_in_mb": 8192
              }
            }
          },
          {
            "node": {
              "id": "4",
              "labels": [
                "Node 4"
              ],
              "metrics": {
                "cpu-usage-percent": 70,
                "memory-usage-percent": 26
              },
              "hardware": {
                "cpu_cores": 2,
                "memory_in_mb": 4096
              }
            }
          }
        ]
      }
    },
    {
      "nodepool": {
        "id": "3",
        "labels": [
          "nodepool3",
          "Z"
        ],
        "nodes": [
          {
            "node": {
              "id": "5",
              "labels": [
                "Node 5",
                "NEW_LB"
              ],
```

```
              "metrics": {
                "cpu-usage-percent": 80,
                "memory-usage-percent": 51
              },
              "hardware": {
                "cpu_cores": 4,
                "memory_in_mb": 8192
              }
            }
          },
          {
            "node": {
              "id": "6",
              "labels": [
                "Node 6"
              ],
              "metrics": {
                "cpu-usage-percent": 70,
                "memory-usage-percent": 25
              },
              "hardware": {
                "cpu_cores": 2,
                "memory_in_mb": 4096
              }
            }
          }
        ]
      }
    }
  ],
  "networks": [
    {
      "network": {
        "nodepool_a_id": "1",
        "nodepool_b_id": "2"
      }
    },
    {
      "network": {
        "nodepool_a_id": "3",
        "nodepool_b_id": "4"
      }
    },
    {
      "network": {
        "nodepool_a_id": "4",
        "nodepool_b_id": "2"
      }
    }
  ],
  "jobs": [
    {
      "profile": {
```

```
        "name": "profile1",
        "weights": [
          {
            "weight": {
              "model_name": "model1",
              "model_weight": 0.5
            }
          }
        ]
      },
      "job": {
        "id": "1",
        "pods": [
          {
            "pod": {
              "policies": [
                {
                  "policy": {
                    "required_labels": [
                      "required_labels 1"
                    ],
                    "rejected_labels": [
                      "rejected_labels 2"
                    ],
                    "prefered_labels": [
                      "prefered_labels 3"
                    ],
                    "retracted_labels": [
                      "retracted_labels 4"
                    ]
                  }
                }
              ]
            }
          },
          {
            "pod": {
              "policies": [
                {
                  "policy": {
                    "required_labels": [
                      "required_labels 2"
                    ],
                    "rejected_labels": [
                      "rejected_labels 2"
                    ],
                    "prefered_labels": [
                      "prefered_labels 3"
                    ],
                    "retracted_labels": [
                      "retracted_labels 4"
                    ]
                  }
```

```
                    }
                ]
            }
        }
    ]
  }
}
```

Listing 6: Example Output JSON File