An Integrated Platform for Multi-Model Digital Twins

Somayeh Malakuti
ABB Corporate Research Center
Ladenburg, Germany
somayeh.malakuti@de.abb.com

Reuben Borrison
ABB Corporate Research Center
Ladenburg, Germany
reuben.borrison@de.abb.com

Arzam Kotriwala
ABB Corporate Research Center
Ladenburg, Germany
arzam.kotriwala@de.abb.com

Benjamin Kloepper
ABB Corporate Research Center
Ladenburg, Germany
benjamin.kloepper@de.abb.com

Erik Nordlund
ABB Corporate Research Center
Vaesteras, Sweden
erik.nordlund@se.abb.com

Kristian Ronnberg
ABB Corporate Research Center
Vaesteras, Sweden
kristian.k.ronnberg@se.abb.com

ABSTRACT

The notion of *Digital Twin* is known as a means to access otherwise dispersed lifecycle data of industrial devices, and enabling advanced reasoning on top of the data via various kinds of models (e.g. machine learning, simulation). Despite many studies on digital twins, there is still a need for common architectures, platforms and information meta-modelling that enable defining various lifecycle data in a harmonized way, as well as integrating the information with machine learning and simulation models; a gap that is filled by this paper. Our approach for the integration of various digital twin models addresses three known technical debt in machine learning systems: *data pipeline jungle*, *undeclared/unstable data dependencies* and *undeclared consumers*. Adopting such an integrated digital twin platform can reduce the required time and effort to develop and maintain digital twin-based solutions, as well as laying a foundation to support a variety of digital twin-based use cases.

CCS CONCEPTS

• **Computing methodologies** → **Modeling methodologies**; • **Software and its engineering** → **Software design engineering**.

KEYWORDS

Digital twin, data integration, model integration, cloud-based architecture

ACM Reference Format:
Somayeh Malakuti, Reuben Borrison, Arzam Kotriwala, Benjamin Kloepper, Erik Nordlund, and Kristian Ronnberg. 2021. An Integrated Platform for Multi-Model Digital Twins. In *IoT '21: International Conference on the Internet of Things, November 08–12, 2021, Switzerland*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Over the past years, the notion of digital twin has evolved from real-time simulation models that are updated with IoT data from the field, to the digital representation of an entity (e.g., device, system), which enables accessing dispersed lifecycle data via unified APIs of the digital twin for different use cases [7]. Here, the lifecycle data is no longer limited to the IoT data, but also includes engineering and IT data, as well as various kinds of models. In fact, in many industries including the manufacturing industry, digital twin is considered a prerequisite for synchronization of operation and maintenance [22].

As digital twins become a key part of IoT solutions, companies have different levels of maturity with respect to their digital twin solutions. Figure 1 shows five levels of maturity that we defined [11]. Here, the level 0 is the current state in a company, in which there are data silos. The level 1 demands for standardized or at least company-wide digital twin information models and APIs, so that different organizations and use cases adopt same approaches for their digital twins modelling. The level 2 demands for expressing correlations among different models embodied within digital twins. At the level 3, the content of the digital twins can further be extended with more advanced models such as machine learning (ML) and simulation models. The level 4 is the enabler for more advanced use cases, where multiple models are combined together, for example, to have intelligent simulation models to predict the remaining useful life of a motor.

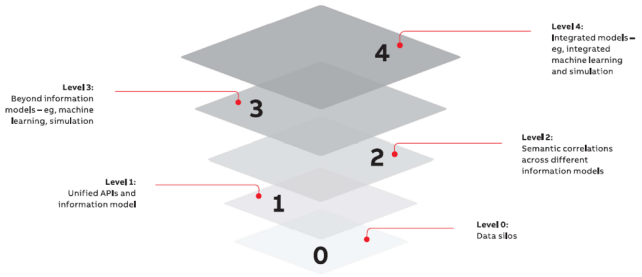

Figure 1: A maturity model for digital twin solutions.

Recent studies [8, 10, 13, 18] show that a significant body of work focuses on the simulation and optimization aspects (i.e., virtual models and services) of digital twins, which map to the levels 3 and/or 4 of our maturity model without covering the levels 1 and 2. By skipping over the levels 1 and 2, each digital twin solution may end up offering a dedicated data pipeline, information modelling, APIs and semantic modelling mechanisms. We have named this as

IoT'21, Nov 08–12, 2021, St. Gallen, CH

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

'digital twin solution silos' [12], which eventually leads to excessive time and effort needed to develop and maintain individual digital twin solutions in companies with a growing number of digital twin-based use cases.

To prevent digital twin solution silos, we proposed an architecture and information meta-model for an integrated digital twin platform [12] in which the maturity level 2 for variety of non-analytic applications is achieved. This paper extends our platform architecture and information meta-model with digital twins that can support individual ML/ simulation models (level 3) as well as interconnected models (level 4). As a result, the following known technical debt in the development of ML [20] as well as digital twin-centric applications are addressed:

- *Data Pipeline Jungle:* Our integrated platform paves the way to establish company-wide data pipelines and digital twin models for various (not limited to ML) use cases.
- *Undeclared/unstable data dependencies*: Our digital twin information meta-model enables defining inter-dependencies of information/ML/simulation models in a declarative way so that automatic reasoning of the inter-dependencies can be facilitated in future.
- *Undeclared consumers*: Our digital twin information model wraps the model with an additional layer of model execution engine and inherits access control by default to keep track of the consumers. This means that any service that needs to access models (whether simulation or ML model) needs to go through our digital twin access control layer and cannot directly call the model. This way there is no scope of undeclared consumers consuming the service.

Addressing these technical debt and adopting an integrated digital twin platform can reduce the required time and effort to develop and maintain digital twin-based solutions, as well as laying a foundation to support a variety of digital twin-based use cases.

This paper extends our previous work [12] in the following manners:

- An information meta-model for digital twins, which supports a) the generic notion of 'functions' as a common way of modelling ML and simulation functions, b) information model elements to map the functions to various digital twin information models to provide/consume data to the ML and simulation functions, and c) information model elements to create a chain of executable models
- Generic model execution components in our digital twin architecture, which enable users to invoke an ML and/or simulation model on demand, possibly leading to a chain of model executions
- A concept for reusable deployment of ML and simulation models to be accessible by our architecture
- Validation of the proposed concepts via an industrial example comprising a combination of ML and simulation models

The rest of this paper is organized as follows. Section 2 describes our use case and problem statement; Section 3 explains our digital twin information meta-model and its application to our use case; our platform architecture is explained in Section 4; Section 5 provides the related work and Section 6 outlines the conclusion and our future directions.

2 USE CASE AND PROBLEM STATEMENT

Simulation of a physical device is a part of many digital twin concepts and architectures. A challenge is however that a simulator does not match a physical device very well without tuning of the simulation parameters. The reasons include e.g. different ambient conditions depending on the place of installation or changes in the device due to aging and degradation. Due to such deviations, the actual behavior of the devices differs from the behavior in the simulation.

Figure 2 shows a combination of simulation and ML to leverage this mismatch between device and simulation to create thermal fingerprints of an electrical motor. The basic idea is to train a ML model to predict the relevant physical parameters from signal data that can be easily collected by sensors installed on the motor. For this purpose, a simulation model specific for the desired motor type is loaded in the simulation runtime and a number of simulations are run with different parametrization representing different instances of the motor type within different states of degradation and different operating conditions of the connected drive.

For the ML model, the set of parameters is the prediction target (y in Fig. 2) it should learn. The simulated sensor values are the input to the ML model from which the parameters should be predicted (X in Fig. 2). For a given installed motor and drive in the field, the real sensor values (X* in Fig. 2) can now be fed to the digital twin cloud and be used as inputs to the ML model. The model will return the parameters (y* in Fig. 2) that most likely represent the current condition of that specific motor. This parameter set is called the thermal fingerprint of the motor, and can be fed into the simulation model that is running on the cloud. In addition, by storing the thermal fingerprints over time, it is possible to monitor the degradation of the electrical motor and improve the planning of preventive maintenance activities.

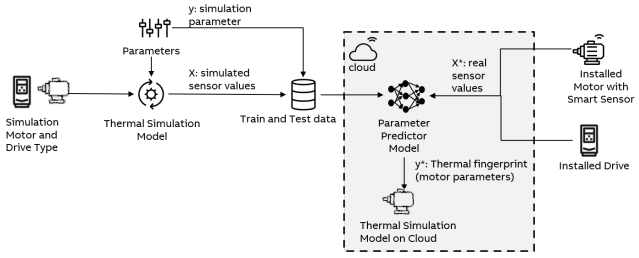

Figure 2: Interacting models.

3 DIGITAL TWINS INFORMATION META-MODEL

In [12] we proposed the information meta-model of our digital twins, with the focus on its elements for making the back-end data accessible via digital twins, to achieve the maturity level 2. In this paper, we extend the meta-model with the elements related to ML and simulation models, as depicted by the classes in the red color in Figure 3. The classes in this diagram represent different concepts that play a role to model lifecycle information of devices, related ML and simulation models, as well as connectivity to external endpoints to fetch the data into digital twins.

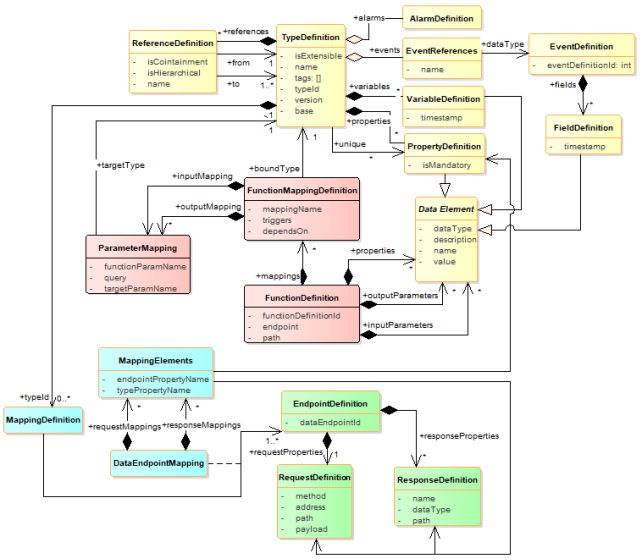

Figure 3: The digital twin information meta-model.

The common information meta-model is based on the JSON format and complies with the object-oriented notions of types, object instances, and relations among the object instances. As the classes in the yellow color depict, each kind of lifecycle data is modeled using a dedicated type; the actual data is modeled as instances of the types. Each type of description consists of a set of properties. Besides, the type descriptions may contain tags to augment them with extra semantic information.

We distinguish between two major kinds of lifecycle data: the one describing an entity along with its properties, and the other describing changing variables, events, or alarms associated with the entity. These can all be modeled in the common information meta-model.

The actual data can be stored in our platform as instance models of the aforementioned types, or the data can be kept in the original data source and retrieved on-demand. This is particularly important when the content of digital twins is stored in various sources (on-premise databases, proprietary IT tools, etc.), and we would like to keep the original data sources as the source of truth without replicating their data within the digital twins. For such cases, the respective digital twin types are marked as 'shadow' types, meaning that it stores the necessary unique properties to distinguish it from other models, but does not have the actual content for the remaining properties. Such a model is associated with the declarative endpoint and mapping descriptions used for fetching the data and mapping it to the model.

The classes in the green color in Figure 3 show the endpoint descriptions. Each endpoint description is designated with a unique identifier, and can have request and response properties. Each communication with an endpoint has two sides: request and response. For the request part, the invocation method (currently HTTP/REST GET or POST), the address of the external server, the URL within that address, and payload are specified. For the response part, the object properties that are returned from that endpoint are defined in the endpoint description.

The response object from an endpoint may be different from the corresponding type description in our platform because the endpoint might be an already existing one that serves multiple other (legacy) applications. Therefore, our solution supports the so-called mapping descriptions, which specify how the response object properties and the corresponding type descriptions map to each other. The classes in the blue color represent the elements of the mapping descriptions. A mapping description is bound to one type and can map the request/response properties of multiple endpoints to the properties of that type.

In our digital twin information modelling, we treat ML and simulation models as functions with specific input and output parameters, which are deployed on the cloud. In the digital twin information modelling, the class *FunctionDefinition* enables defining such a function, its name, endpoint address, and input/output parameters specification. In addition, one may define extra properties for each function on-demand; an example extra property is the URL in which a simulation file is stored.

Each *FunctionDefinition* may be associated to zero or more *FunctionMappingDefinition* that in turn is bound to one device type in our information model. This means that a simulation/ML model may be executed for different device types. Each *FunctionMappingDefinition* has a list of *ParameterMapping* specifications, which has a separate entry for each input/output parameter of the respective function. For each input parameter, the *ParameterMapping* specifies from which properties/variables of which digital twin, the actual value for the input parameter should be fetched. For each output parameter, the *ParameterMapping* specifies in which property/variables of which digital twin type the output values should be stored. The actual digital twin can be specified via the *query* attribute of the *ParameterMapping*.

Each *FunctionMappingDefinition* defines the events that should trigger to execution of a function, as well as execution dependencies of functions, respectively via *triggers* and *dependsOn* properties. The *dependsOn* specification defines the dependencies among models, where the input of on model comes from the output of another model. This specification also implies an execution order among models. The *triggers* can be user invocation, a timer, the variables updates in a digital twin, the occurrence of an event or alarm in a digital twin, or an internal platform event such as the construction of a new digital twin model. The *triggers* specifications enable defining a chain of models that need to be executed in a sequence; for example, when the results of one model is ready, the next model is chain should start executing.

The separation of type definition from function definition in our meta-model has the following benefits: a) The ML applications have an exploratory nature where data scientist experiment with multiple models whose input/output parameters may differ. By separation the function specifications from the type specifications, the latter ones remain intact if the former one evolves. b) The reusability of functions increases as they can be bound to different device types. c) The input/output parameters of a function can be fetched from and stored in multiple types.

Figure 4 shows an example of defining dedicated types for a motor, a smart sensor, and a drive. These types define the properties

and operational variables of the respective device, and are parts of the specification of the devices' digital twin. As we illustrated in [12], depending on the requirements of use cases, dedicated types can be defined to model different lifecycle data (bidding, engineering, installation, etc.) of a device. These types can modularly be added to the description of the digital twin of a device, upon their availability. For a specific device, these types can be instantiated with concrete values for properties and variables; and the totality of the object instances form the digital twin of that device.

The type *abb.motors.M12AB5*, which defines a specific kind of industrial motor, has one relation named as *mountedSmartSensors* that refer to the type *abb.smartsensors.S123BC* to model the smart sensors that are mounted on the motor. In addition to the relations among physical devices, we need to also define how the results of ML and simulation models are stored and linked to the respective devices. On one hand, each model has naturally different kinds of outputs; on the other hand, since ML applications have an inherent exploratory nature, there will be multiple models associated to a device over its lifetime. Therefore, we need to make sure that new models and their results can be flexibly added to the digial twin of a device.

To this aim, we make use of type inheritance in our information model. Here, the type *abb.model.results* is the base type for other types that should be defined exclusively for maintaining the results of a specific ML/simulation model. This type does not define any property or variable, as they have to be defined in inherited types; the type *abb.parameterPredictor.results* shows an example. To be able to associate the results of an ML/simulation model to a device digital twin, we need to define a dedicated relation to the type *abb.model.results* within the digital twin of the device. The relation *executableModels* in the type *abb.motors.M12AB5* shows an example of such a relation. Since this relation is defined to the base type *abb.model.results*, so that any object of its sub-types can be referenced.

Figure 5 shows an example definition of a ML function and its mapping. The function *abb.parameterPredictor* is deployed at the specified URL, and has a set of input/output parameters. The function mapping defines one possible mapping of this function to the motor of the type *abb.motors.M12AB5*. The actual motor for which this function should be executed will be identified via its serial number upon the invocation of this function. The specification of the input parameters defines that value of the input parameter *speed* should be fetched from the respective variable of the smart sensor that is mounted to the motor. Since there can be multiple smart sensors mounted to a motor, we should still explicity identify the smart sensor of interest by its serial number; this can be done either in the mapping specification or explicitly when we invoke the function to make the mappings more reusable. The specification of the input parameters defines that value of the input parameter *torque* should be fetched from a drive; but since we do not have any relation between drive and motor modelled in our digital twins (see Figure 4), we should specify the drive of interest via its serial number during the inovcation of this function. In the next section, we explain how specific devices of interest can be specified upon the invocation of a function.

The specification of the output parameters of this function specifies that the results of this function should be stored in an object

Figure 4: An example type definition.

of the type *abb.parameterPredictor.results* that will internally be referenced by the corresponding motor object.

Figure 5: An example machine learning model definition.

Figure 6 specifies a simulation function and its mapping to the motor of the type *abb.motors.M12AB5*. This mapping specifies that the function execution should be triggered when an object of the type *abb.parameterPredictor.results* is created; i.e., the machine larning model *abb.parameterPredictor* is executed and its results are available. The input parameters of this function are fetched from respective properties of the *abb.parameterPredictor.results*, and the results will be stored in a new object of the type *abb.thermalsimulation.resu*

Figure 6: An example simulation model definition.

4 ARCHITECTURE

Figure 7 depicts the abstract architecture of our solution, which has been implemented based on Microsoft Azure services. In our previous work [12], we explained the details of this architecture and its APIs to applications to ingests backend data into digital twins; this paper extends the previous work with necessary components and APIs for supporting ML and simulation models and their execution. For the sake of completeness, we explain the entire components of the architecture in the following.

4.1 The Common Information Model Components

Type Storage is a MongoDB database in which type descriptions for digital twin models are stored; the component *Type Processor* is a .Net microservice that interfaces this storage. *Object Storage* is a Cosmos DB database in which object instances are stored; the component *Object Processor* is a .Net microservice that interfaces this storage. Both *Type Processor* and *Object Processor* offer dedicated REST APIs to applications for create, read, update and delete (CRUD) operations.

Event Storage is the telemetry storage based on Azure Timeseries Insight technology. The communication with devices takes place

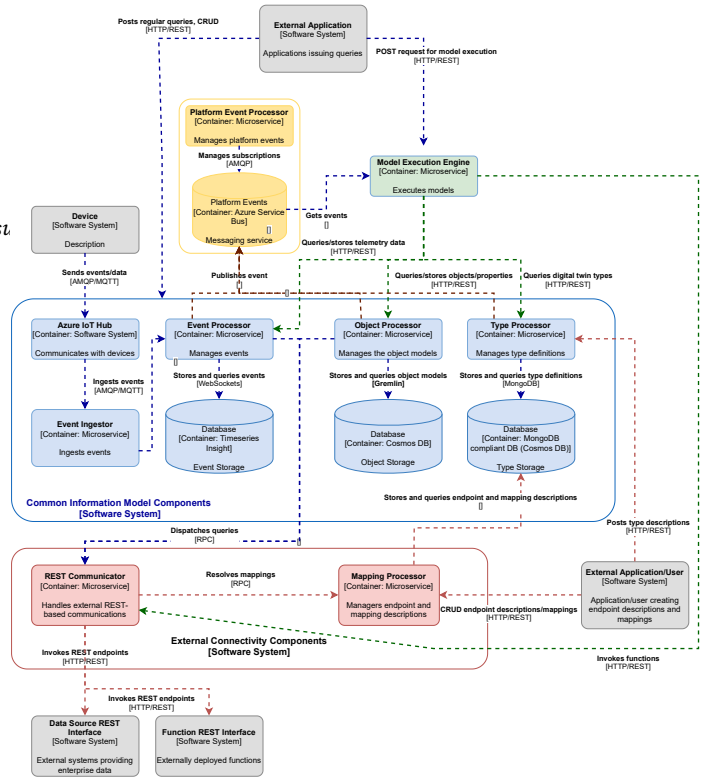

Figure 7: Overall architecture of the digital twin platform.

using AMQP or MQTT protocols and Azure IoT Hub service is used to facilitate the communication. If a device does not natively provide its OT data in our common information model, an industrial edge device can be used as the interface to translate the device OT data model to our common information model format.

The component *Event Ingestor* implements the functionality to receive the events and telemetry data from IoT Hub and to insert them in *Event Storage*. The component *Event Query Processor* interfaces *Event Storage* to enable accessing device events via REST APIs.

4.2 The External Connectivity Components

The function endpoint and mapping descriptions that are explained in the previous section are input to the component *Mapping Processor* via its CRUD REST API, which stores them in *Type Storage*.

The component *REST Communicator* provides the functionality for interpreting the endpoint and mapping descriptions to connect to the external data sources and/or functions, make an invocation to get their results, and map their responses back to the internal types defined in *Type Storage*. Our solution currently supports HTTP/REST protocol for the communications with external data sources and deployed functions.

4.3 The Model Execution Engine Component

The component *Model Execution Engine* offers APIs to receive model execution requests and execute the models accordingly.

4.3.1 The Structure of the Deployed Model. Unlike normal software deployment, ML model deployment [19] consists of :

1. **Code** for data preprocessing and execution of ML model.
2. **ML Model**, which is trained on the historical data and will be used to get predictions.
3. **Data** required for getting predictions from ML model

The problem with the above deployment is that it only focuses on ML model deployment and misses the aspect that the ML pipeline may also consist of additional software and analytics artifacts such as simulation model or grid search which may work together with ML model. Therefore, to accomodate such type of use cases model deployment (hereby model is used for simulation, ML and analytics model) can consist of following:

1. **Code** for data preprocessing and execution of ML, simulation or analytics model.
2. **Model**, which can be a trained ML model, simulation model or analytics model used by the execution code to get results.
3. **Data** required as an input to the model.

The model will be deployed as a microservice in our architecture and the architecture was designed in a way that many different models can be easily deployed and executed in an automated fashion. Here, every deployment comprises a standard structure which is given below and shown in Figure 8:

1. **REST endpoint:** It must be exposed to serve the incoming requests with input data to the model and the response consists of the model output.
2. **I/O definition:** It is the format in which the data must be served to the REST endpoint and is received as the response (predictions) from the model.
3. **Code:** It is the code required to load the model and execute it to get the output. If the I/O definition data format differs from the input and output format of the model, additional preprocessing code must be included in the package for transforming the data formats.
4. **Model artifact:** It should be present in the serialized form to load it for getting the output.

To enforce the structure for the deployment, we used fastAPI [1] to wrap the data preprocessing and model execution code. FastAPI is a web framework which helps to build REST APIs and provides an easy way of defining input and output definition for the requests and response of the API endpoint. Also, FastAPI provides built-in support for handling errors in case a non-compliant input data is sent over the REST endpoints or an exception occurs while execution of the model. Furthermore, FastAPI provides a prebuilt docker image which can be used to package the custom code and model and thereby package it in a new docker image.

The built docker image can then be deployed as a microservice in the architecture environment. The network configuration of the container is done in such as a way that it can be only accessed by other containers in its network and cannot be accessed by the containers outside its network. This way the container is always secure of the external unauthorized request.

4.3.2 The Execution of the Model. Figure 9 shows an example invocation request for our use case. Here, to execute a function (i.e., an ML or simulation model), we must provide the unique identifier

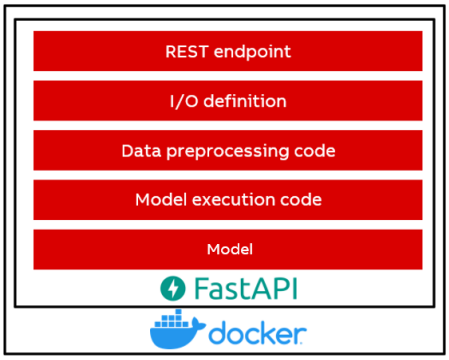

Figure 8: Structure of the ML model deployment with the libraries used for packaging.

of that function in our platform. Since a function can be bound to multiple devices individually, and consequently there will be multiple function mappings for that function, we must also specify the function mapping of interest. In addition, the concrete device for which the function must be executed has to be specified using its *typeId* and serial number in our digital twin descriptions.

In addition, we must specify the digital twin of devices from which the input values of the function should be fetched. In our example mapping in Figure 5, these are a smart sensor and a drive with a specific serial number. If the input values are telemetry data (i.e. modelled as variables in our information meta-model), one needs to also specify the date range for which the data needs to fetched, and the operation that needs to be performed on the data. If the input values are properties in our digital twin models, there will be no need to specify these, because only the latest value of the properties are available in our platform.

```
{
    "function": "parameterPredictor",
    "functionMapping": "parameterPredictor_M12AB5",
    "deviceType": "abb.motors.M12AB5",
    "serialNumber": "123ABC456",
    "inputObjects": [
        {
            "deviceType": "abb.smartsensors.S123BC",
            "serialNumber": "5387BC5"
        },
        {
            "deviceType": "abb.drives.ACS880",
            "serialNumber": "9877BC5"
        }
    ],
     "telemetryValues": {
        "date": {
            "from": "2021-06-01T09:44:47.707Z",
            "to": "2021-06-10T09:44:47.707Z"
        },
        "operation": "avg"
    }
}
```

Figure 9: An example invocation request.

The *Model Execution Engine* component receives the input requests, queries the specified function, mapping, and type descriptions from *Type Processor*. To fetch the values of the function's input parameters, it also queries the specified properties and/or variables from *Object Processor* or *Event Processor*, respectively. If the values

must be fetched from external sources, *Object Processor* or *Event Processor* issue a request to the *External Connectivity Components*; the details of this step is explained in [12].

After fetching the specified input values, *Model Execution Engine* invokes the specified function, which should have been deployed on the cloud beforehand. Based on the mapping specification for the function's outputs, the outputs are stored in *Event Storage* and/or *Object Storage* and are returned to the invoker as well.

As depicted in Figure 6, functions may form a chain through their *dependsOn* or *triggers* specifications. As for triggers, our platform makes use of Azure Service Bus to maintain various kinds of events (either internal platform events or digital twin variable updates/events/alarms), which occur in the platform. Each of the processor components in our platform publish respective events in the service bus, and *Model Execution Engine* subscribes to the specified triggering events and initiates the execution of the respective function. As for models dependencies, *Model Execution Engine* interprets the dependency chain and executes the models from the first model in the chain. The execution terminates after the last model in the chain is successfully executed, or there is any exception in the execution of models in the chain.

5 RELATED WORK

Multiple classifications and surveys exist, which summarize the state-of-the-art on the topic of digital twins [8, 10, 13, 18]. These studies identify that a significant body of work focuses on the simulation aspects of digital twins for specific use cases in a distinct lifecycle phase of devices. The authors in [8] indicate the need for common architectures and platforms that offer means for defining digital twins especially by incorporating early lifecycle data, and enabling interoperable interactions among digital twins. In addition, [22] mention that digital twins can achieve maximal potential benefit from digital transformation if ML models are learnt and integrated to leverage historical data coupled with the current (plant) state. In particular, decision-makers should be empowered to query "what-if" situations to digital twins. The surrogate models mentioned in the use-case (see Section 2) can be used for this purpose.

Whilst our previously proposed platform [12] enables supporting various kinds of digital twin-enabled use cases, this paper shows how our proposal can also support simulation and ML models alongside the information models of digital twins. Our focus is not on a specific simulation or ML model for a specific use case, but on a generic platform in which digital twins with various kinds of inter-linked information models, ML and simulation models can be developed, as well as preventing some technical debt (namely, data pipeline jungle, undeclared/unstable data dependencies, and undeclared consumers), which may occur in the development of both digital twin and ML-based systems to ensure these systems are maintainable during their lifetime.

For instance, when addressing the data silo problem, one may consider (cloud-based) data warehouse solutions such as Azure SQL Data Warehouse and/or Data lakes for Big Data [6] as a potential solution. We observe the following limitations to these solutions compared to ours: Firstly, these solutions are data-centric and are widely adopted for (business) reporting and analytics use cases. However, there are many different kinds of Industrie 4.0 use cases that require a device-centric view on data, need connectivity of data models to physical devices and also must deal with the data silo problem across the device lifecycle [17]. Adopting separate solutions for reporting or analytics use cases leads to the solution silo problem. Secondly, unlike our approach, these solutions usually keep a copy of the data from data sources; hence, the original data sources are not the source of truth and one may need to deal with data inconsistency issues.

To the best knowledge of the authors at the time of writing, the integration and management of simulation and ML models in Digital Twin infrastructures is not given sufficient attention in existing scientific literature. For instance, the Asset Administration Shell (AAS) [16] has been proposed as a digital twin for manufacturing systems. Since the specification of AAS is still under development, we experienced the standardization deadlock [4] problem in adopting AAS due to the late availability of its specifications, and its insufficient expression power for our use cases. For example, the current specifications of AAS do not cover events, various query APIs, nor the strategies to ingest backend data and map it to the AAS. Moreover, supporting digital twins with variety of inter-linked ML and simulation models has not been a focus of AAS until now. As for data interoperability, AAS is designed to enable interoperable data exchange across the vendors. In our solution, we take a two-step approach for the data interoperability: 1) intra-company interoperability using our information meta-model that can represented various kinds of lifecycle data and models, and 2) on-demand translation of our information models to AAS for inter-company interactions as we studied in [15]

Furthermore, in [21], an architecture for intelligent digital twins in cyber-physical systems is proposed. However, the details of how data ingestion from various sources takes place, as well as various APIs that must be offered to applications are not discussed. Alam and Saddik describe C2PS [2], a "digital twin architecture reference model" for cloud-based cyber-physical systems. Their focus is on network communication aspects and controller design. In [14], a service-oriented application for knowledge navigation is presented. The architecture of the application linking different data sources is outlined briefly without mentioning the approach of how to link those data sources together. In [23], a digital twin platform based on a data-centric middleware is defined, whose architecture mostly focuses on communication and data transfer between the physical assets and simulation and not the data silo problem. In [3], a solution is proposed for sensor data integration and information fusion to build digital-twins for cyber-physical manufacturing. Contrary to this solution, our proposal is not limited to specific sensor data, also covers lifecycle data that come from various sources as well as ML and simulation models.

In our contribution, we have provided a generic approach for extending digital twins with simulation and ML models without limiting ourselves to a specific use case or domain. Some related works do exist [5, 9, 24] but with key differences. For instance, [5] propose a 'generic industrial architectural framework of a digital twin' to utilize real-time information from a physical asset as well as structured information (based on Discrete Event Simulation) for the process industry. In addition to their support for telemetry data and

SQL databses, we harmonize heterogeneous enterprise data stored in different external systems via the information meta-model.

The authors in [24] introduce a 'generic CPS system architecture for DT establishment in smart manufacturing' using a 'tri-model-based approach' (i.e. digital model, computational model and graph-based model) which works concurrently to simulate real-world physical behaviour and characteristics of the digital model. They have a more device-centric solution whereby they address efficient ingestion of device data and even enable direct control of devices. However, we cover soliciting data not only from devices but from external enterprise systems into the digital twin. Both, [5] and [24] do not fully explain their information models, making it difficult to judge the expression power of their infrastructures to support diverse use cases.

In [9], an abstract and conceptual architecture is proposed for composing multiple digital twins that are possibly offered by different companies. However, some challenges seem to remain open, such as how would digital twins be composed if companies adopt different digital twin frameworks. We have addressed this topic in our previous work [15], where we enable on-demand interoperability across vendors by translating our digital twin models to the AAS format. The study needs to be extended to also consider digital twin ML and simulation models in future.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed an information meta-model and a cloud-based architecture for creating and managing digital twins, which consist of information models, ML models, simulation models, and/or a combination of these. Adopting a platform-based approach for digital twin solutions help to prevent the *data pipeline jungles* technical debt in companies, because one integrated platform paves the way to establish company-wide data pipelines and digital twin models for various (not limited to ML) use cases. Our information meta-model enables defining various lifecycle data (including IoT data, IT and engineering data), as well as the specification of ML and simulation models and their possible integration; this solution helps to prevent the *undeclared/unstable data dependencies* technical debt and paves the way to in future perform various kinds of analysis on the declared dependencies. As for preventing the *undeclared consumers*, the model service is always called through the model execution engine and never directly by the end consumer service. This restricts access to the services which are not authorized to call the model and therefore prevents undeclared consumers to access the models.

As future work, we would like to experiment with different kinds of simulation as well as other kinds of models such as geometry model, behavioral model, etc. In addition, we would like to explore the implementation of different analytics and ML techniques on our platform e.g. stream analytics.

REFERENCES

[1] 2021. FastAPI framework. https://fastapi.tiangolo.com/.

[2] Kazi Masudul Alam and Abdulmotaleb El Saddik. 2017. C2PS: A digital twin architecture reference model for the cloud-based cyber-physical systems. *IEEE Access* 5 (2017).

[3] Yi Cai, Binil Starly, Paul Cohen, and Yuan-Shin Lee. 2017. Sensor Data and Information Fusion to Construct Digital-twins Virtual Machine Tools for Cyber-physical Manufacturing. *Procedia Manufacturing* 10 (2017), 1031–1042.

[4] Rainer Drath and Mike Barth. 2012. Concept for managing multiple semantics with AutomationML – Maturity level concept of semantic standardization. In *ETFA*. IEEE, 1–8.

[5] Jonathan M Eyre, Tony J Dodd, Chris Freeman, Richard Lanyon-Hogg, Aiden J Lockwood, and Rab W Scott. 2018. Demonstration of an industrial framework for an implementation of a process digital twin. In *ASME International Mechanical Engineering Congress and Exposition*, Vol. 52019. American Society of Mechanical Engineers, V002T02A070.

[6] Alex Gorelik. 2019. *The Enterprise Big Data Lake: Delivering the Promise of Big Data and Data Science.* O'Reilly Media.

[7] Industrial Internet Consortium. 2020. Digital Twins for Industrial Applications.

[8] David Jones, Chris Snider, Aydin Nassehi, Jason Yon, and Ben Hicks. 2020. Characterising the Digital Twin: A systematic literature review. *CIRP Journal of Manufacturing Science and Technology* 29 (2020), 36 – 52.

[9] Francesco Leotta and Massimo Mecella. 2020. Realizing Smart Manufacturing Architectures through Digital Twin Frameworks. (2020).

[10] Mengnan Liu, Shuiliang Fang, Huiyue Dong, and Cunzhi Xu. 2021. Review of digital twin about concepts, technologies, and industrial applications. *Journal of Manufacturing Systems* 58 (2021), 346–361.

[11] Somayeh Malakuti. 2021. Emerging technical debt in digital twin systems. In *26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE.

[12] Somayeh Malakuti, Prerna Juhlin, Jens Doppelhamer, Johannes Schmitt, Thomas Goldschmidt, and Aleksander Ciepal. 2021. An architecture and information meta-model for back-end data access via digital twins. In *26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*.

[13] Elisa Negri, Luca Fumagalli, and Marco Macchi. 2017. *Procedia Manufacturing* 11 (2017), 939 – 948.

[14] Antonio Padovano, Francesco Longo, Letizia Nicoletti, and Giovanni Mirabelli. 2018. A Digital Twin based Service Oriented Application for a 4.0 Knowledge Navigation in the Smart Factory. *IFAC-PapersOnLine* 51 (2018), 631–636.

[15] Marie Platenius-Mohr, Somayeh Malakuti, Sten Grüner, Johannes Schmitt, and Thomas Goldschmidt. 2020. File- and API-based interoperability of digital twins by model transformation: An IIoT case study using asset administration shell. *Future Generation Computer Systems* 113 (2020), 94 – 105.

[16] Plattform Industrie 4.0. 2020. Details of the Asset Administration Shell – Part 1.

[17] Q. Qi and F. Tao. 2018. Digital Twin and Big Data Towards Smart Manufacturing and Industry 4.0: 360 Degree Comparison. *IEEE Access* (2018), 3585–3593.

[18] Qinglin Qi, Fei Tao, Tianliang Hu, Nabil Anwer, Ang Liu, Yongli Wei, Lihui Wang, and A.Y.C. Nee. 2021. Enabling technologies and tools for digital twin. *Journal of Manufacturing Systems* 58 (2021), 3–21.

[19] A.; Windheuser; C. Sato, D.; Wider. 2019. Continuous Delivery for Machine Learning.

[20] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden Technical Debt in Machine Learning Systems *(NIPS'15)*. MIT Press.

[21] Behrang Ashtari Talkhestani, Tobias Jung, Benjamin Lindemann, Nada Sahlab, Nasser Jazdi, Wolfgang Schloegl, and Michael Weyrich. 2019. An architecture of an Intelligent Digital Twin in a Cyber-Physical Production System. *at - Automatisierungstechnik* 67 (2019), 762 – 782.

[22] Jørn Vatn. 2018. Industry 4.0 and real-time synchronization of operation and maintenance. In *Safety and Reliability–Safe Societies in a Changing World*. CRC Press, 681–686.

[23] S. Yun, J. Park, and W. Kim. 2017. Data-centric middleware based digital twin platform for dependable cyber-physical systems. In *Ninth International Conference on Ubiquitous and Future Networks (ICUFN)*.

[24] Pai Zheng and Abinav Shankar Sivabalan. 2020. A generic tri-model-based approach for product-level digital twin development in a smart manufacturing environment. *Robotics and Computer-Integrated Manufacturing* 64 (2020), 101958.