Highlights

HP2C-DT: High-Precision High-Performance Computer-enabled Digital Twin

E. Iraola, M. García-Lorenzo, F. Lordan-Gomis, F. Rossi, E. Prieto-Araujo, R. M. Badia

- Introduces HP2C-DT, a reference architecture integrating HPC into digital twins.

- Proposes a framework enabling real-time DT execution across edge, cloud, and HPC resources.

- Uses COMPSs agents to dynamically orchestrate tasks across heterogeneous infrastructures.

- Demonstrates reduced communication overhead while preserving critical information.

- Validates fast response times and scalable HPC execution in a power grid use case.

HP2C-DT: High-Precision High-Performance Computer-enabled Digital Twin

E. Iraola[a,*], M. García-Lorenzo[a], F. Lordan-Gomis[a], F. Rossi[b], E. Prieto-Araujo[b], R. M. Badia[a]

[a]*Barcelona Supercomputing Center, Plaça d'Eusebi Güell, 1-3, Les Corts, Barcelona, 08034, Catalonia, Spain*
[b]*CITCEA, Universitat Politécnica de Catalunya, Av. Diagonal, 647, Les Corts, Barcelona, 08028, Catalonia, Spain*

Abstract

Digital twins are transforming the way we monitor, analyze, and control physical systems, but designing architectures that balance real-time responsiveness with heavy computational demands remains a challenge. Cloud-based solutions often struggle with latency and resource constraints, while edge-based approaches lack the processing power for complex simulations and data-driven optimizations.

To address this problem, we propose the *High-Precision High-Performance Computer-enabled Digital Twin* (HP2C-DT) reference architecture, which integrates High-Performance Computing (HPC) into the computing continuum. Unlike traditional setups that use HPC only for offline simulations, HP2C-DT makes it an active part of digital twin workflows, dynamically assigning tasks to edge, cloud, or HPC resources based on urgency and computational needs.

Furthermore, to bridge the gap between theory and practice, we introduce the HP2C-DT framework, a working implementation that uses COMPSs for seamless workload distribution across diverse infrastructures. We test it in a power grid use case, showing how it reduces communication bandwidth by an order of magnitude through edge-side data aggregation, improves response times by up to 2x via dynamic offloading, and maintains near-ideal strong scaling for compute-intensive workflows across a practical range of resources. These results demonstrate how an HPC-driven approach can push digital twins beyond their current limitations, making them smarter, faster, and more capable of handling real-world complexity.

Keywords: Computing continuum, Digital twin, HPC, Industry 4.0, Cyber-physical systems, Edge computing, Cloud computing, IoT, Artificial intelligence, Power systems

1. Introduction

Digital twins have emerged as powerful tools to integrate the physical and digital worlds, enabling advanced monitoring, control, and decision-making. Although much research has focused on their simulation and visualization aspects, less attention has been paid to integrating the computational demands of high-fidelity modeling with the need for rapid decision support. This gap becomes particularly relevant in highly distributed systems that need both low-latency responses at the Edge and high computational power to operate.

Therefore, a significant challenge in digital twin architectures is bringing together fast response times with massive computing capacity. Many existing approaches rely on centralized cloud solutions, which can be inadequate for real-time applications due to network latency. On the other hand, purely edge-based solutions lack the computational power needed for complex simulations and large-scale optimizations. The present work addresses these issues by proposing a novel reference architecture for digital twins, the *High-Precision High-Performance Computer-enabled Digital Twin* (HP2C-DT), and an accompanying process framework for its implementation.

A key contribution of our reference architecture is bringing High-Performance Computing (HPC) into the loop, allowing the digital twin to take on heavy computational tasks like probabilistic analysis, synthetic data generation, and model training while still keeping the fast response times needed for real-time decisions at the Edge. HPC stands out from general cloud computing or private clusters by offering dedicated high-performance resources with specialized parallel processing and high-memory bandwidth. This is especially important when quick results are needed, like in short-term simulations that help drive immediate decisions. With HPC, the system can scale its resources as needed, ensuring that even demanding computations can be handled efficiently without slowing down critical operations.

The HP2C-DT reference architecture is designed with three primary focuses: Information Technologies/Operation Technologies (IT/OT) integration, computational infrastructure, and data management. The IT/OT integration dimension ensures that edge components can be deployed on diverse hardware and operate with minimal latency. From an infrastructure perspective, the system dynamically assigns computational tasks to edge, cloud, or HPC resources based on urgency and computational load. A relevant innovation is the adaptive orchestration of tasks, where time-sensitive computations execute immediately at the Edge, while less urgent workloads are scheduled asynchronously on available HPC or cloud resources. At the same time, in terms of data management and communication, HP2C-DT balances computation with data exchange, ensuring that messages and sensor updates are transmitted between edge

and cloud nodes without overwhelming the network. The proposed architecture handles this through a combination of heartbeats to track node status and local aggregation of measurements.

The proposed reference architecture defines a blueprint for the core components, their interactions, and best practices to develop digital twin systems. Building on this foundation, this article introduces the HP2C-DT framework, which has been designed and implemented to bridge the gap between the reference architecture and real-world applications. The framework not only specifies the enabling technologies and provides detailed guidelines for their integration, but also includes a working implementation that simplifies the development of digital twins.

At its core, the HP2C-DT framework integrates edge computing, remote cloud components (cloud or on-premises), and HPC resources into a cohesive system. A key element enabling this integration is COMPSs [1], which orchestrates parallel and distributed execution. Taking advantage of the function-driven approach of the reference architecture, each node—whether at the Edge or in the Cloud—runs a COMPSs agent that coordinates with peers to offload tasks dynamically. This offloading mechanism also includes HPC resources, which allow heavy computations to be assigned to high-performance computing clusters when needed.

A use case of HP2C-DT is in electrical grids, where edge nodes are distributed over vast geographical areas. In such environments, sending data to a central cloud node for analysis is infeasible due to strict latency constraints. Instead, real-time fault detection and response must occur directly at the Edge, while HPC resources handle computationally intensive tasks such as predictive maintenance, load forecasting, and probabilistic risk assessment. Similar scenarios arise in industrial automation, where production line adjustments require millisecond-level reaction times while complex optimization models refine scheduling and resource allocation, and in autonomous transportation networks, where real-time navigation depends on local edge decisions supported by large-scale traffic simulations and data-driven route optimization. Building on the reference architecture and implementation framework, we develop a prototype of a digital twin for power grids to test HP2C-DT's main features in a real-world scenario.

This article is structured as follows. Section 2 provides a concise review of related work on digital twin architectures, their connection to the computing continuum, and their use of HPC resources. Section 3 outlines the proposed reference software architecture, followed by Section 4, which examines the most important features of the reference architecture in detail. Section 5 introduces the HP2C-DT framework, extending the reference architecture for implementation. Section 6 presents an experimental use case in power systems, demonstrating the benefits of HP2C-DT. Finally, Section 7 provides concluding remarks.

2. Related work

Michael Grieves [2] first introduced the concept of a *digital twin* in 2014 as a virtual representation of a physical product, rooted in the management of the product life cycle. This involves a continuous exchange of data: the physical object sends real-time information to its virtual counterpart, while simulations or optimizations performed on the virtual object influence the physical object. In 2017, Grieves expanded this concept by introducing *Digital Twin Instances*, which are individual virtual models of physical products; *Digital Twin Aggregates*, which combine multiple instances; and the *Digital Twin Environment*, a space where digital twins interact in multiphysics simulations [3].

A more recent definition, provided by the CIRP Encyclopedia of Production Engineering [4], describes a digital twin as a digital representation of a unique product, system, or service that captures its characteristics, properties, and behaviors throughout various phases of the life cycle. The digital twin operates through models, information, and data, and enables real-time monitoring, optimization, and decision-making.

Jones et al. [5] further refined the concept of a digital twin by identifying its key elements. A *physical object* or *entity*, such as a machine or system, has a corresponding *virtual object* or *entity* that digitally represents it. These entities exist within *physical and virtual environments*, with different levels of *fidelity* affecting their accuracy. This fidelity can be measured through the number of parameters transferred between physical and virtual objects. The *state* of a digital twin depends on the measured parameters, such as temperature or voltage, which are collected through *metrology*. The digital twin also includes *processes* for updating the physical or virtual model based on new data. The *physical-to-virtual connection* captures real-world data through sensors, while the *virtual-to-physical connection* applies insights from simulations to control real-world devices.

2.1. Twinning

Less emphasis has been placed on how digital twins contribute beyond visualization and monitoring—specifically, in decision-making and model refinement. In Jones et al.'s terms, these capabilities fall under twinning, which refers to the synchronization of physical and virtual instances through metrology (reading an entity's state) and realization (modifying an entity's state). The frequency of twinning varies by application, with high-speed synchronization being critical in fields like power grids.

However, Jones et al. detach twinning from virtual processes. Virtual processes provide computational tools such as optimization, prediction, simulation, and analysis, but these tools are usually not involved in (a) updating the virtual instance when discrepancies arise between expected and measured parameters or (b) determining how the digital twin should respond to such discrepancies in the physical system. Since synchronization relies on continuous data exchange and analysis, the interaction with these computational tools is essential. Yet, how often and in what ways virtual processes should influence synchronization remains unclear.

2.2. Edge computing perspective in digital twins

Integrating digital twins with the computing continuum offers advantages for low-latency applications. Qi and Tao [6] propose a hierarchical architecture that combines edge, fog, and cloud computing to optimize data processing across layers in digital twin-based shop floor environments. This architecture enables real-time control at the Edge, medium-latency data integration at the fog level, and large-scale analysis in the Cloud, and addresses challenges such as bandwidth constraints and latency. Building on this foundation, Groshev et al. [7] introduce the concept of Digital Twin as a Service (DTaaS), highlighting the role of edge computing, network function virtualization (NFV), and 5G technologies. Their study explains how edge computing minimizes end-to-end latency by offloading tasks closer to physical devices, while 5G provides ultra-reliable low-latency communication (URLLC) for real-time applications like remote control and visualization. Together, these studies show how the computing continuum meets the strict low-latency and high-reliability needs of digital twins in Industry 4.0.

2.3. Cloud-computing and HPC

Another important part of digital twins is how they handle heavier computational load, usually leaning on cloud computing for its flexibility and ease of access. This processing power is needed for running complex simulations, real-time data analysis, and predictive modeling, and allows digital twins to mirror and anticipate the behavior of physical systems. But, while the Cloud works well in many cases, it can fall short when extreme performance is needed. High-Performance Computing (HPC), on the other hand, offers more power and efficiency and can handle the heaviest workloads with higher speed and accuracy [8], yet it is still not widely used for digital twins. This is why integrating HPC into digital twin architectures has gained traction as computational demands for high-fidelity modeling and predictive analytics grow. The Digital Twin Engine (DTE) proposed by the European Centre for Medium-Range Weather Forecasts (ECMWF) [9] exemplifies this trend, leveraging EuroHPC supercomputers to deploy climate digital twins for simulations and data handling. The DTE emphasizes code optimization for accelerator technologies like GPUs, providing efficient use of HPC resources while maintaining interoperability across distributed systems. Similarly, recent work by Ares de Parga et al. [10] proposes a parallel Reduced Order Modeling (ROM) workflow using COMPSs [1, 11] on HPC clusters. In general, we see that many efforts focus on using HPC resources for highly complex, physics-based simulations, but there is still a lack of close integration of these resources into the digital twin life cycle from a software perspective.

2.4. Orchestration of asynchronous workflows

Task orchestration plays a key role in managing the diverse resource demands in this area. Nie et al. [12] propose a multi-agent, cloud-edge orchestration framework that uses the digital twin and the Industrial Internet of Things (IIoT) concepts to improve distributed production control. Their approach combines cloud-based production models with self-adaptive strategies to handle dynamic exceptions and make better use of resources. However, while the framework works well for distributed manufacturing, it mainly focuses on static scheduling and does not explore real-time synchronization across distributed resources in depth. Similarly, Nguyen et al. [13] introduce a digital twin orchestration framework with features such as federation, translation, brokering, and synchronization, demonstrating its use in smart city applications such as hotspot prediction and driving assistance. Their work highlights the benefits of cross-domain collaboration and federated learning for AI model sharing, but it remains mostly conceptual and does not address performance optimization in resource-limited environments or large-scale industrial settings. In contrast, Ares de Parga et al. [10] take a more hands-on approach, evaluating the COMPSs framework for orchestrating complex workflows in heterogeneous environments. This way, the authors effectively manage task dependencies and resource allocation in distributed systems, making it a promising option for digital twin orchestration.

2.5. Gaps and novel contributions

Existing architectures often isolate HPC, edge, and cloud resources, resulting in inefficiencies in workflows that demand both low-latency responses and intensive computation. Although domain-specific projects optimize HPC for modeling and simulation, they do not incorporate real-time decision-making at the Edge. Conversely, hyper-distributed platforms prioritize the edge-cloud balance but omit HPC for large-scale simulations. This article proposes a reference software architecture that addresses these gaps by introducing HPC in the loop within a parallel and distributed orchestration component, together with IT/OT integration at the Edge.

3. HP2C-DT reference architecture

The HP2C-DT reference architecture is structured into three layers: Edge, Cloud, and HPC, as shown in Figure 1. The architecture is designed so that the Edge layer focuses on the underlying infrastructure, while the Cloud layer provides a global perspective of the digital twin. Meanwhile, the HPC layer supplies the raw computational power needed for intensive processing.

The Edge layer manages physical devices and local computing resources. It operates autonomously, processing information locally to meet real-time requirements. This layer handles immediate control tasks and real-time monitoring, ensuring quick responses to changes in the environment. For example, sensor measurements collected by edge nodes are transmitted to the Cloud layer using a publish-subscribe messaging model (*sensor measurement* lines in Figure 1). At the same time, actuation commands from the Cloud are sent back to specific edge nodes through point-to-point messaging (*actuation lines* in the figure). This dual communication mechanism ensures efficient data collection together with precise control.

The Cloud layer acts as the brain of the system, integrating various components such as virtual representations of devices, database storage, computation tools, and user interfaces.
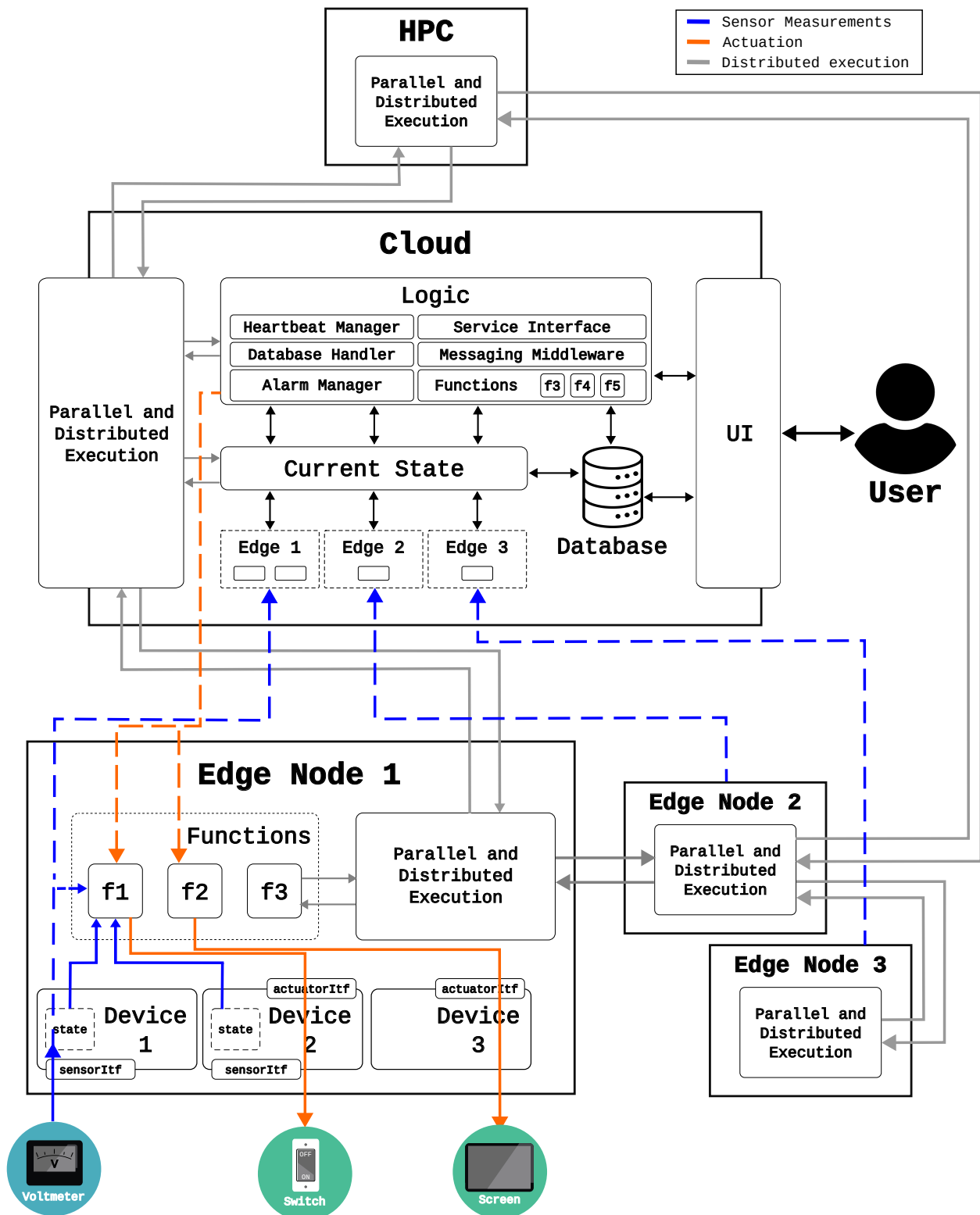
Figure 1: Reference architecture diagram.

It collects and processes information from multiple edge nodes, storing measurements in a time series database, and triggering alarms when needed. The Cloud layer also facilitates advanced computations like simulations and probabilistic analyses by leveraging its access to the full digital twin representation. Its messaging middleware manages communication with edge nodes, while other modules handle tasks like real-time monitoring (via heartbeat signals), alarm logging, and exposing APIs for external services.

Finally, the HPC layer provides high-performance computing capabilities for resource-intensive tasks that exceed the capacity of the Edge or Cloud layers. HPC systems consist of several computing nodes (generally servers with high computing power and a large amount of memory) interconnected with a high-speed network and designed for parallel processing of complex problems. These systems often incorporate specialized hardware accelerators—GPUs (Graphics Processing Units), FPGAs (Field-Programmable Gate Arrays), or TPUs (Tensor Processing Units)—to solve specific problems. Some cutting-edge HPC systems also integrate emerging technologies like quantum computing, which leverages quantum mechanics to solve certain types of problems exponentially faster than classical computers.

HPC systems outperform the Cloud in several ways: they offer heterogeneous hardware better suited to application requirements, lower-latency and higher-bandwidth networking, and optimized software workflows without virtualization overhead. For organizations with consistent high-performance computing needs, in-house HPC can also be more cost-effective than indefinite cloud rentals. In addition, stricter user control in HPC environments enhances data privacy and security by keeping data within the organization and reducing exposure to external threats. Unlike cloud platforms, HPC systems restrict direct service deployment, instead relying on job schedulers like SLURM, which streamline access control and ensure exclusive resource allocation. However, this scheduling mechanism means executions may not start immediately if resources are unavailable.

Beyond the capabilities of each layer, the reference architecture orchestrates computation across them to balance real-time responsiveness and large-scale processing. Computation tasks are referred to as functions and are classified based on latency requirements: synchronous functions execute locally for immediate responses, while asynchronous functions are scheduled across edge, cloud, or HPC nodes depending on resource availability. The Parallel and Distributed Execution component, present on every node, manages this orchestration. For computationally intensive tasks such as large-scale simulations or optimizations, it offloads execution to the HPC layer.

4. Key architectural principles

The HP2C-DT reference architecture balances real-time responsiveness at the Edge with large-scale computational capabilities. This is achieved by integrating operational and computational layers to handle execution, communication, and data flow. This section details the core architectural principles that enable this integration.

4.1. IT/OT integration

A key aspect of the proposed reference architecture is its interaction with the physical environment within a digital twin. For this, the architecture follows a taxonomy of digital twin entities that mirror physical and logical components. The Edge layer is responsible for interacting with *physical objects*, collecting sensor data, and executing control actions on them. To achieve this, edge nodes maintain an image of each connected physical device—a *physically bound digital object*—which keeps a record of the device data received and can communicate with it to apply actions when possible. In contrast, the *virtual digital object* in the Cloud layer provides a more approximate image of the devices. This way, the Edge layer handles a smaller volume of information and remains close to the physical world, while the Cloud layer aggregates data from numerous edge nodes with a lower burden. Consequently, approximation and aggregation are required to manage scalability, reduce communication overhead, and extract higher-level insights from distributed edge data sources to the cloud node. Section 4.3 offers more information about such mechanisms.

Following this logic, each physical object in the digital twin is associated with a *physically bound digital object* and a *virtual digital object*. Within HP2C-DT, and from a software perspective, these are referred to as a *device*. A device functions as a *sensor* if it produces a measurement stream, an *actuator* if it performs actions that modify the physical environment, or both if it supports both roles.

Figure 2a illustrates the hierarchical abstraction of devices. The first conceptual layer consists of the `Sensor`, `Device`, and `Actuator` abstractions, which define the core functionalities required for interaction within the architecture. All sensor-based devices must support triggering predefined sequences of actions when configured to do so, while actuator devices allow actions to be performed on the hardware when required. The second layer introduces domain-specific abstractions, such as `Abstract Sensor 1` and `Abstract Actuator 1`, which refine the base abstractions by specifying data types and measurement magnitudes. Finally, the implementation-specific layer contains concrete classes that inherit from the previous abstractions, integrating hardware- and manufacturer-dependent features. In the figure, `Concrete Sensor 1` extends `Abstract Sensor 1`, while `Concrete Actuator 1` extends `Abstract Actuator 1`. This layer defines low-level interactions with hardware, including communication protocols and data handling. Concrete devices also inherit from vendor-specific classes, such as `Manufacturer Device 1`, which encapsulates hardware drivers and communication details for a specific device family.

Consider the digital twin of a power grid. As represented in Figure 2b, the voltmeter sensor would be represented as an abstract class, `Voltmeter`, in the domain-specific layer, inheriting from the `Device` and `Sensor` abstractions. The `Device` abstraction handles properties such as labeling and positioning
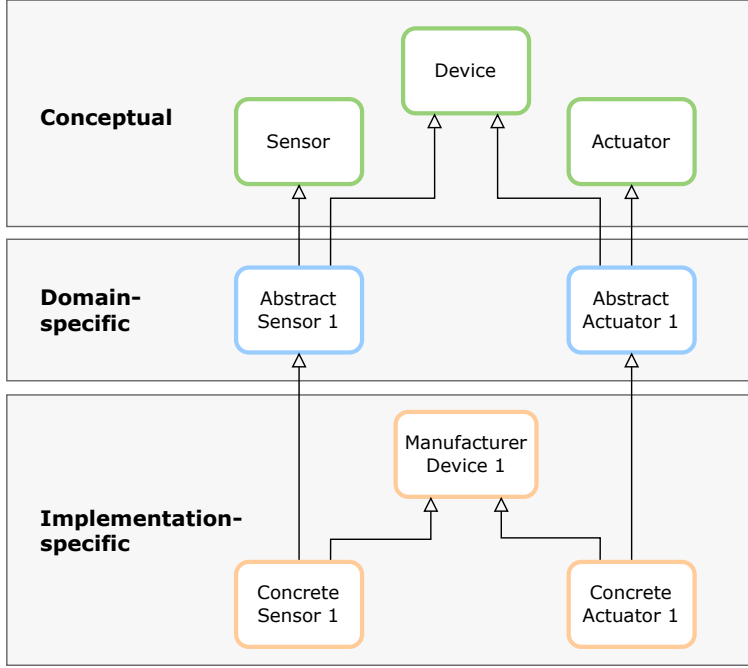
(a) Levels of abstraction abstraction of digital twin objects.
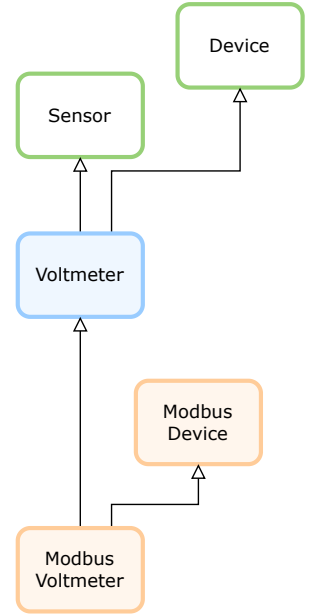
(b) Voltmeter example.

Figure 2: Hierarchical abstraction of digital twin objects.

to ensure proper system integration, while the `Sensor` abstraction standardizes data collection and processing across different sensors. A concrete implementation of the voltmeter abstraction would extend this structure by integrating manufacturer-specific communication details. In the example of Figure 2b, the concrete class `Modbus Voltmeter` would implement the `Modbus Device` class, applicable in cases where multiple devices use the Modbus protocol [14], a common industrial communication standard.

This hierarchical structure enhances flexibility and simplifies system extensibility. It enables a clear separation of concerns, from defining high-level functionalities to managing detailed hardware implementations. This approach also facilitates supporting devices with varying capabilities and those from different manufacturers, without overwhelming the system design. The taxonomy of digital twin objects, based on their proximity to the physical datum, along with the hierarchical structure of their software implementations shown in Figure 2, forms the basis for IT/OT integration. Together, they help map operational data from physical devices to computational processes in the digital twin.

4.2. Data processing management

Functions are modular computation tasks defined by the user in the HP2C-DT architecture. Depending on urgency requirements, they can be configured as synchronous or asynchronous. Because they share the same execution model, they can be used in a similar way at both edge and cloud nodes. Synchronous functions execute locally and immediately on the requesting node. Asynchronous functions, managed by the Parallel and Distributed Execution component, are scheduled and

distributed across the digital twin infrastructure in a Function-as-a-Service (FaaS) manner, executing on available resources or offloading to peer nodes. They may consist of a single task or form complex workflows with multiple dependent or nested tasks, each assigned to a node that meets its constraints. In addition to being synchronous or asynchronous, functions can be triggered in different ways depending on system needs:

- Frequency-based trigger (`onFrequency`): Executes periodically at a fixed interval.

- Event-based trigger (`onRead`): Executes when a new measurement arrives from a specified sensor. It can be configured to run after a set number of measurements instead of every individual reading.

- Change-based trigger (`onChange`): Executes only when a sensor's state changes, reducing redundant computations and communication overhead.

- On-start trigger (`onStart`): Executes once at application deployment, useful for persistent components like messaging listeners that handle actuation signals between edge and cloud nodes.

4.3. Network traffic reduction

The Edge layer can reduce communication overhead through aggregation methods, which is crucial in high-frequency domains, such as power systems. Each physically bound digital sensor object maintains a rolling window that stores recent measurements. Each entry consists of a timestamp and one or more recorded values, depending on the sensor type.

Rolling windows serve multiple purposes within the digital twin architecture. Functions can leverage them to compute results over recent time frames, but their primary role is to enable aggregation. Without aggregation, sending each measurement separately would overwhelm the network as the number of distributed nodes scales. Instead, edge nodes can transmit entire rolling windows—preserving all data while controlling transmission frequency—or apply aggregation methods that further reduce message size. The latter is particularly useful in environments with a high density of sensor information. In Section 5, we explore the implementations of different aggregation methods.

5. HP2C-DT framework

The HP2C-DT framework is a full implementation of the HP2C-DT reference architecture, that provides a modular and extensible software solution to deploy digital twins that take advantage of edge, cloud, and HPC environments. It defines standardized components for device management, computation, communication, and data aggregation. This way, the framework enables developers to implement domain-specific digital twins while benefiting from a robust underlying infrastructure that simplifies integration. Through its modular design, it supports hardware-agnostic deployment, making it suitable for a wide range of industrial applications.

In this framework, edge and cloud nodes run as Java applications inside Docker containers [15]. Java ensures platform independence, efficient concurrency, and access to a rich set of libraries, while the Docker container isolates dependencies and configurations, ensuring that each edge node runs in a controlled and reproducible environment free from compatibility issues. Each of these nodes contains an instance of a COMPSs agent [16] that implements the Parallel and Distributed Execution component of the reference architecture.

For communication, RabbitMQ [17] is used as the messaging middleware, with a central broker running in the Cloud layer. This setup supports both publish-subscribe and point-to-point messaging, as required by the reference architecture. RabbitMQ offers message decoupling while also providing features such as queuing, delivery guarantees, and clustering. Its flexibility and open-source nature allow for partial decentralization, prevent vendor lock-in, and support the hybrid communication model defined by the HP2C-DT reference architecture.

The HP2C-DT framework is available for public use[1]. The following sections go through the most significant features of the framework.

5.1. COMPSs Agents

COMPSs is a programming framework designed to boost developers' productivity working on parallel and distributed applications. By abstracting the complexity of parallel and distributed execution, COMPSs allows developers to focus on algorithm design without worrying about low-level details such

as task synchronization, data transfers, or workload distribution.

To do so, COMPSs follows a task-based approach. Programmers write sequential code in a plain major programming language—currently, it supports Java, Python, and C/C++—with no reference to any specific API to explicitly manage infrastructure and parallelism. By defining an interface or using decorators, programmers select a set of methods whose invocations will be replaced by an invocation to a runtime system that will handle its execution as an asynchronous task. The runtime system supporting the programming model analyzes the data accessed by each invocation to detect potential data dependencies with previously detected tasks and build a task-dependency graph. The runtime orchestrates the execution of all detected tasks across a distributed infrastructure, ensuring the sequential consistency of the original code.

The Colony framework [16] introduces an agent-based deployment strategy for COMPSs, allowing any node participating in the infrastructure to automatically convert sequential functions into task-based workflows. By deploying an agent on each node in the infrastructure—whether at the Edge or in the Cloud layers—the HP2C-DT framework facilitates this transformation, allowing functions (described in Section 4.2) to be seamlessly converted into individual COMPSs tasks or orchestrated into structured workflows with appropriate resource allocation. This ensures that code execution fully leverages the available computing resources, including multiple CPU cores, GPUs, and other accelerators embedded in the node. In addition, the framework supports decentralized peer-to-peer orchestration of task executions. Agents collaborate to distribute the workload, allowing a function triggered on the Edge to offload all or part of its workflow to the Cloud layer or neighboring edge nodes. Likewise, since COMPSs is compatible with HPC environments, any task delegated by cloud or edge nodes can be submitted to the HPC system and expanded into a complex, parallel workflow.

5.2. Edge layer implementation

In the Edge layer, communications are implemented to ensure adequate data exchange between edge nodes and the Cloud layer. Each edge node runs a publisher function that sends messages to a RabbitMQ queue with the routing key `edge.<EDGE_ID>.sensors.<SENSOR_ID>`, which the server reads. Furthermore, a consumer function listens for incoming actuation commands on `edge.<EDGE_ID>.actuators.<SENSOR_ID>`.

Each edge node loads a JSON configuration file. Listing 1 shows part of a configuration file for an edge node labeled `edge1`. The `window-size` field sets the default rolling window size for all devices, though individual sensors can override it (see Section 5.2.3 for details on how the framework handles data management). The `comms` section configures communication protocols and hardware-specific parameters. The field `devices` lists the sensors and actuators. Section 5.2.1 provides more details about this setup. The field `funcs` specifies the computational functions that will run at the edge node, as explained in Section 5.2.2.

---

[1] For intellectual property concerns, a link to the repository will be added after the article is accepted for publication.

Listing 1: Example of global configuration of an edge node (Snippet from setup file `edge1.json`).

```json
{
  "global-properties": {
    "type": "edge",
    "label": "edge1",
    "window-size": 10,
    "comms":{
      "modbus": {
        "ip": "10.14.85.205",
        ...
      }
    }
  },
  "devices": [
    ...
  ],
  "funcs": [
    ...
  ]
}
```

5.2.1. Device implementation hierarchy

The HP2C-DT framework implements the `Device`, `Sensor`, and `Actuator` classes at the conceptual layer of the digital object hierarchy, following Figure 2. These come as a class for `Device` and interfaces for `Sensor` and `Actuator`, ensuring that any digital device can be implemented by extending them. The `Device` class handles instantiation from a JSON setup file, assigns labels and manages status availability. The `Sensor` and `Actuator` interfaces enforce the implementation of sensing and actuating methods. Specifically, `Sensor` requires a `sensed` method that dictates what happens when a new measurement arrives, including triggering computations and storing rolling windows. Similarly, `Actuator` defines how commands are received and executed. Developers can extend `Device` and implement `Sensor` or `Actuator` to support new measurement types while ensuring compatibility with the existing system.

The framework provides pre-implemented domain-specific classes that serve both as examples and as foundations for hardware-specific implementations. The current implementation focuses on power systems, offering abstract classes such as `Ammeter`, `Voltmeter`, `Wattmeter`, `Generator`, and `Switch`. The first three implement the `Sensor` interface and the last two, both `Sensor` and `Actuator`. Developers can use these as templates to create additional domain-specific classes. The built-in `MeasurementWindow`, defined on each of the abstract classes, maintains rolling windows of previous measurements, allowing historical data analysis (see Section 5.2.3). In a similar way, concrete classes that handle communications with hardware are provided at the implementation-specific layer.

Listing 2 shows a continuation of the edge configuration file in Listing 1 in which we set up a voltmeter object. The `label` field assigns a human-readable identifier, such as `Voltmeter Gen1`. The `driver` field specifies the concrete class, in this case, `ConcreteVoltmeter`, an implementation-specific component that ensures correct communication with the hardware on the Edge. The `properties` section defines operational parameters: `comm-type` specifies the Modbus protocol, `indexes` selects measurement channels, `window-size` determines the size of the rolling window (overriding the general window size

set in global properties for this specific device, if set), and `aggregate` configures aggregation, here set to a phasor representation (see Section 5.2.3). Users can specify multiple devices in the configuration file.

Listing 2: Example of configuration of devices (Snippet from setup file `edge1.json`)

```json
{
  "global-properties": {
    ...
  },
  "devices": [
    {
      "label": "Voltmeter Gen1",
      "driver": "es.bsc.hp2c.edge.manufacturer1.
          ConcreteVoltmeter",
      "properties": {
        "comm-type": "modbus",
        "indexes": [1],
        "window-size": 5,
        "aggregate": "phasor"
      }
    },
  ],
  "funcs": [
    ...
  ]
}
```

5.2.2. Functions

HP2C-DT functions can be implemented in Java, the architecture's native language, or Python, which is supported through UNIX domain sockets for local execution. Each function follows a fixed signature, always receiving three arguments: a list or map of sensors it can access, a list or map of actuators it can control, and a map of additional `parameters` specific to the function's requirements. This signature allows users to define various computational operations, such as model training, inference, state verification, or other custom computations. Additionally, the implemented triggers (see Section 4.2) allow the execution of functions on demand.

Listing 3 shows the definition of functions in the `func` block through an example function, `VoltLimitation`, which triggers an action to disconnect the edge node from the grid when overvoltage is detected. The `lang` field specifies that this function is written in Java. The entry `type` specifies whether the function executes immediately or can be deferred. Here, the value `synchronous` indicates that the function's computation is urgent and must be performed as soon as it arrives. Otherwise, for the `asynchronous` cases, the execution would not be immediate and could even be offloaded to other nodes in the architecture, which would, in turn, be handled by the COMPSs agent.

The `method-name` field provides the fully qualified class path where the logic of the function lies. The `parameters` section defines the function's inputs and outputs. In this example, the function processes data from the `Voltmeter Gen1` sensor and controls the `Three-Phase Switch Gen1` actuator to disconnect the system. Furthermore, an additional parameter, `threshold`, is listed in `other`, indicating that the function enforces a voltage limit of 400 volts. The `trigger` section specifies that this function operates in an event-driven manner, using the `onRead` mechanism to execute upon receiving

new sensor data (refer to Section 4.2 for information about the available function-triggering modes). The `trigger-sensor` field ensures that only readings from `Voltmeter Gen1` initiate execution, while the `interval` parameter configures the function to activate every five measurements instead of every single reading.

Listing 3: Configuration of functions (Snippet from setup file `edge1.json`).

```
{
  "global-properties": {
    ...
  },
  "devices": [
    ...
  ],
  "funcs": [
    {
      "label": "VoltLimitation",
      "lang": "Java",
      "type": "synchronous",
      "method-name": "es.bsc.hp2c.edge.funcs.
          VoltLimitation",
      "parameters": {
        "sensors": ["Voltmeter Gen1"],
        "actuators": ["Three-Phase Switch Gen1"],
        "other": {
          "threshold": 400
        }
      },
      "trigger": {
        "type": "onRead",
        "parameters": {
          "trigger-sensor": ["Voltmeter Gen1"],
          "interval": 5
        }
      }
    },
  ]
}
```

5.2.3. Aggregation and rolling windows

For data management, the HP2C-DT framework uses rolling windows that store measurements in each device object. The rolling window is implemented as a circular buffer: when it reaches full capacity, the oldest measurement is automatically replaced by the newest one. This design ensures constant-time insertions and deletions, making it optimal in both time and memory efficiency. Since no memory reallocation or element shifting is required, access to recent data remains fast.

The framework also provides implementations of aggregation methods, as seen in Section 4.3, specifically the domain-agnostic ones: `all`, `average`, `sum`, and `last`. The `all` method is the trivial case where every measurement in the window is sent. `sum` transmits the sum of all values in the window, `average` computes and sends the mean, and `last` retains only the most recent value. Aggregation methods are easily configurable in the edge setup file, as indicated by the `aggregate` field in Listing 2. In addition, domain-specific aggregation methods may be useful, and the framework's modular design allows them to be easily integrated.

A particularly valuable implementation is the phasor representation [18], which provides a compact representation of oscillatory signals. Phasor aggregation represents a periodic signal through a single complex number that encodes amplitude and phase. Given a sinusoidal signal with amplitude A, angular frequency ω, and initial phase ϕ,

$$x(t) = A \cos(\omega t + \phi) \tag{1}$$

its phasor representation is $X = A e^{j\phi}$ or $X = A \angle \phi$. This means that a steady-state sinusoidal signal $x(t)$ can be represented by only two parameters, its amplitude A and its angle ϕ. We implement the `phasor` aggregation method using a Discrete Fourier Transform (DFT) approach to process the time series windows.

Data aggregation induces information loss, which in turn can be assessed using the Mean Squared Error (MSE), ε_{MSE}, a standard metric for signal reconstruction fidelity. For a sensor producing a single value per measurement, given a rolling window $X = \{x_1, x_2, \ldots, x_n\}$ of size n, the information loss introduced by an aggregation function is:

$$\varepsilon_{\text{MSE}} = \mathbb{E}\left[(X - \hat{X})^2\right] \tag{2}$$

where $\hat{X}$ represents the estimated reconstruction of X from the aggregated value $Y = f(X)$.

For average aggregation, the estimate is simply $\hat{x}_i = Y = \frac{1}{n} \sum x_i$, leading to:

$$\varepsilon_{\text{avg}} = \left(1 - \frac{1}{n}\right)\sigma_X^2 \tag{3}$$

which shows that for large n, nearly all variance is lost.

For last-value aggregation, all previous values are ignored. Assuming an autoregressive process with correlation ρ between consecutive samples, the reconstruction error is:

$$\varepsilon_{\text{last}} = (1 - \rho^2)\sigma_X^2 \tag{4}$$

If $\rho \approx 1$, little information is lost, but for uncorrelated data ($\rho \approx 0$), most information is discarded.

The phasor transformation retains full information for pure sinusoidal signals:

$$\varepsilon_{\text{phasor}} \approx 0, \quad \text{if } x(t) \text{ is a steady-state sinusoid} \tag{5}$$

For signals with harmonics or noise, phasor aggregation filters out these components, leading to an effective information loss of:

$$\varepsilon_{\text{phasor}} \approx 1 - \frac{P_{\text{fundamental}}}{P_{\text{total}}} \tag{6}$$

where $P_{\text{fundamental}}$ is the power of the fundamental frequency component and P_{total} is the total signal power.

This analysis shows that information loss increases with n for sum and average aggregation, making them useful for noise smoothing but unsuitable for precise real-time tracking. Last-value aggregation, however, is more effective for highly correlated signals, as it retains the most recent relevant information with minimal error. Domain-specific methods such as the phasor representation can be particularly effective. In these cases, aggregation drastically reduces the size of messages while preserving essential signal characteristics, making it a practical choice for data aggregation.

5.3. Cloud layer implementation

The Cloud layer maintains a collection of virtual digital objects, each representing a physical device linked to an edge node. This collection updates dynamically with every incoming heartbeat and measurement, both of which are managed by the main RabbitMQ broker also running alongside the Cloud. The broker also handles point-to-point actuation towards edge nodes.

To reduce communications beyond data aggregation, the Cloud can perform internal simulations to estimate device states when data is missing, delayed, or intentionally reduced to optimize communication. A two-sided simulation mechanism allows both the Edge and Cloud layers to estimate states independently, triggering data exchange only when discrepancies between estimations and actual measurements arise.

For time series data management, the Cloud layer integrates InfluxDB [19], optimized for fast ingestion and retrieval of sensor data, supporting downsampling and retention policies. The Cloud hosts an instance of this database, along with a dedicated database handler with utility methods for querying and writing data.

Monitoring, visualization, and user interaction rely on Django [20] and Grafana [21]. Django serves as the web interface, allowing operators to monitor the status of edge nodes, track alarms, and perform manual actuation on distributed nodes. It also manages user authentication and integrates with the central database to display system updates in real-time. Grafana complements Django by embedding dynamic monitoring panels directly into the interface, offering real-time visualization of trends, anomalies, and system behavior. To facilitate external interactions, the Cloud layer exposes a REST API, enabling services such as real-time data sharing, status retrieval, and edge node configuration management.

Finally, the Server layer uses a JSON configuration file similar to that in edge nodes, but without a `devices` field. Instead, its knowledge of virtual digital objects is derived dynamically from edge node heartbeats rather than direct physical connections.

6. Experimental evaluation

6.1. Case study

Electric system operators currently use models that capture the actual state of their networks through Supervisory Control and Data Acquisition (SCADA) systems, allowing human operators to make informed decisions from a centralized control room. To make such decisions, operators mainly rely on the current state of the network, along with studies conducted using static (e.g., power flow) and dynamic (e.g., phasor-based) models extracted from the SCADA system.

This centralized approach has several limitations, including low-bandwidth decision-making, limited automation, and restricted control at the Edge. At the same time, the rise of renewable energy is shifting power systems from a centralized, controllable setup to a more variable and decentralized one. Addressing these limitations requires a more distributed

and high-performance approach, which the HP2C-DT framework provides. It supports HPC, which allows for Electromagnetic Transient (EMT) simulations while keeping the traditional power flow and phasor-based models available, ensuring different levels of fidelity. It also enables computations at the Edge, where key assets of power systems—such as generators, power converters, transformers, protection relays, and metering stations—operate as HP2C-DT devices with their corresponding digital replicas.

6.2. Experimental setup

We use a real-time simulator to model the power grid under different operating scenarios. The OPAL-RT hardware platform, OP4512, [22] runs a real-time EMT model of the electrical network using the Hypersim software [23], capturing dynamics in the kilohertz range. This is essential for accurately modeling power systems dominated by renewable energy. In our prototype, the OPAL-RT hardware represents the actual electrical grid and provides input and output interfaces. We create the necessary digital objects of the implementation-specific layer to interact with it, as explained in Section 5.2.1. Edge nodes run on a computing node directly connected to the OPAL-RT, ensuring real-time interaction. The Cloud layer consists of virtual machines hosted on an on-premise server at BSC, each with 4 CPUs and 8 GB of RAM. HPC resources are provided by the general-purpose partition of the MareNostrum 5 supercomputer, which consists of compute nodes equipped with two Intel Xeon Platinum 8480+ processors—each with 56 cores at 2.0 GHz—for a total of 112 cores per node. Each node includes 512 GB of RAM and is interconnected via NVIDIA HDR InfiniBand for high-throughput, low-latency communication.

6.3. Experiments

In this section, we evaluate the system's performance through experiments measuring (1) communication bandwidth, (2) response time, and (3) HPC scalability. For this, we use the experimental setup described above. In particular, for (3), we select an application representative of a digital twin of a power system: stability analysis.

Stability analysis helps adjust the operation point of a power grid to specific operational objectives, such as reducing power generation or maximizing renewable energy consumption, without risking system stability. To enable stability analysis for real-time decision-making, we can leverage machine learning (ML) models, which can be trained to serve as surrogate models. In power systems, ML-based surrogates can replace stability assessment tools and allow stability evaluations to be carried out online on edge and cloud nodes [24].

However, training such ML models requires data. Although the DT continuously collects measurements from the operation of the actual system, such data may be insufficient to train accurate ML-based surrogate models, which requires the creation of a synthetic dataset [25]. We can generate the synthetic dataset offline by exploring the operational space and applying conventional tools for power system stability assessment at each point.
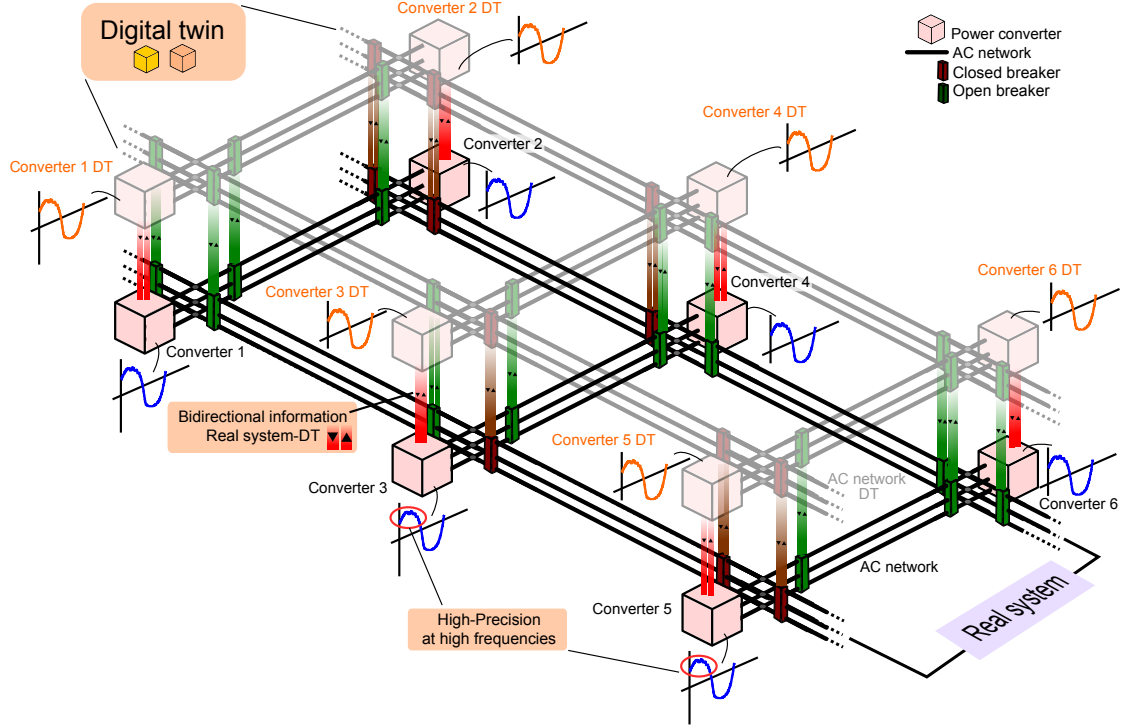
Figure 3: Power grid digital twin concept diagram.

The HP2C-DT framework is well suited for this, as data exploration and generation are highly parallelizable and the framework integrates HPC throughout the digital twin's architecture. Due to this, we propose a data generator tool to evaluate computation scalability on our prototype.

The tool starts by sampling the entire operational space and progressively focuses on regions of higher interest—those intersecting the stability margin of the system—using an entropy-based criterion [26]. After this, the tool recursively divides the variable space into subregions by segmenting the dimensions with a higher impact on system stability. The execution parameters include the depth of exploration and the branching factor. The variable space consists of the power injected by each generator, the power demanded by each load, and several converter control parameters. The tool is implemented in Python using the GridCal library [27], which specializes in power flow calculations.

6.3.1. Experiment 1: Reducing communication overhead

In this experiment, we assess how rolling windows and aggregation methods impact data transmission in the HP2C-DT framework. The experiment varies two parameters: the sampling interval T_s and the aggregation interval T_a. The sampling interval defines how often the hardware collects measurements and sends them to the Edge layer. The aggregation interval determines how frequently the Edge layer transmits data to the Cloud layer. Additionally, for this experiment, we make the aggregation interval define the rolling window size. For example, if the sampling interval is 1 ms and the aggregation interval is 10 ms, each message sent to the cloud node is derived from a rolling window of the last 10 measurements.

We perform the experiment using a single edge node deployed locally, and a cloud node running on an on-premises server at BSC. We assign the edge node a single sensor that generates measurements at predefined acquisition intervals of 1 ms, 10 ms, 100 ms, and 1 s. Each acquisition interval is tested with aggregation intervals of 1 ms, 10 ms, 100 ms, 1 s, and 10 s. The case where both intervals are set to 1 ms represents the baseline, where every measurement is sent individually. The cloud node receives these messages and measures the data transmission rate in bytes per second, including message headers of the RabbitMQ middleware.

Two aggregation strategies are tested. The first one is the `all` aggregation, which groups all measured values without reducing the number of data points. The second is the `phasor` aggregation, which compresses a rolling window into a single amplitude and angle value. Table 1 presents the results for `all` aggregation, while Table 2 shows the results for `phasor` aggregation. Each table displays the measured required bandwidth for different acquisition-aggregation interval pairs, where rows correspond to sampling intervals, and columns to aggregation intervals. The results are the average of five separate executions for each combination of parameters. Note that Table 1 includes cases where $T_s \leq T_a$, covering the limit case in which every measurement is sent as a separate message. In contrast, Table 2 only includes cases where $T_s < T_a$, as the aggregation method requires a minimum number of measurements to operate.

The results for `all` aggregation (Table 1) show that increasing the aggregation interval consistently lowers the required bandwidth. However, the rate of improvement diminishes: in the first row (1 ms acquisition interval), increasing the aggregation interval from 1 ms to 10 ms reduces bandwidth by more

than 5x, but going from 10 ms to 100 ms only reduces it by about 1.7x. This happens because the total number of measurement data remains constant across each row, so the only reduction in required bandwidth comes from decreasing message overhead. The overhead effect is strongest for the smallest aggregation intervals, as the number of messages per second decreases exponentially from left to right. Moving diagonally in the table reveals the true effect of reducing both the number of messages and the number of data points per message, leading to a reduction in required bandwidth proportional to the sampling interval.

Table 2 highlights the impact of actual data aggregation. Since `phasor` aggregation condenses each rolling window into just two floating-point values (amplitude and angle), the results become independent of the sampling interval. The Required bandwidth decreases proportionally with the aggregation interval, achieving a nearly 10x reduction at each step. In contrast, the `all` aggregation was limited by the need to transmit every measured value, leading to diminishing improvements.

This experiment demonstrates the benefits of rolling windows and aggregation in reducing data transmission overhead in the HP2C-DT framework. Rolling windows alone significantly lower the number of messages, reducing the impact of middleware overhead. When combined with aggregation techniques like `phasor`, which, in turn, takes advantage of domain-specific knowledge, the total amount of transmitted data also decreases, achieving a more efficient data flow while preserving essential information. The HP2C-DT framework integrates these features at the Edge layer, making it adaptable to domain-specific digital twins with varying requirements for latency, bandwidth, and data fidelity.

6.3.2. Experiment 2: Response time for IT/OT actions

The second experiment evaluates how functions behave in HP2C-DT, focusing on their flexibility in synchronous/asynchronous and distributed execution. Specifically, we measure response times when an action is triggered by a sensor reading. The response time is the duration between the sensor measurement and the execution of a corresponding action on hardware, such as closing a switch, after processing a computational task. We test three execution modes:

1. The function runs sequentially on the edge node.
2. The function executes as a workflow on the edge node using COMPSs.
3. The function offloads execution as a workflow on a cloud node using COMPSs.

The setup follows the same structure as in Experiment 1. In this case, we configure the system so that the edge node operates with one computing unit for sequential execution (1) and two computing units for workflow execution (2), while the cloud node uses four computing units (3). This setup reflects a real-world edge-cloud environment, where cloud nodes typically have more computational power than edge nodes. The goal is to determine whether offloading computation to the cloud improves performance and under what conditions. It also allows us to assess the benefits of internal parallelization within the edge node.

The experiment examines two sources of overhead: computational overhead from COMPSs task management and communication latency from transmitting execution requests. To assess computational impact, we test functions with different workloads, specifically matrix multiplications of increasing sizes. The computation is parallelized by dividing the multiplication into m^2 blocks of size b^2, where m is the number of blocks per dimension and b is the block size per dimension. Each time the sensor records a measurement, it triggers a computation task executed through one of the three modes. Once the workflow completes, the function signals a device, the `MsgAlert` actuator, printing the result and measuring the total time from measurement to result generation.

Figure 4 shows the results. For small tasks (lower values of b), parallelizing the workflow on the edge node or offloading it to the cloud offers no advantage. As task size increases, there is a break-even point at approximately $b = 16$, beyond which offloading reduces response times. Furthermore, as the number of tasks in the workflow increases (higher values of m), offloading to the cloud becomes increasingly beneficial. For the configuration with $m = 4$ and $b = 256$, we observe a 2x speedup in two scenarios: when comparing edge execution (2 CPUs) to sequential execution (1 CPU), and when comparing server execution (4 CPUs) to edge execution. Note the logarithmic scale on the y-axis, so visual differences in execution time do not directly reflect proportional speedups.
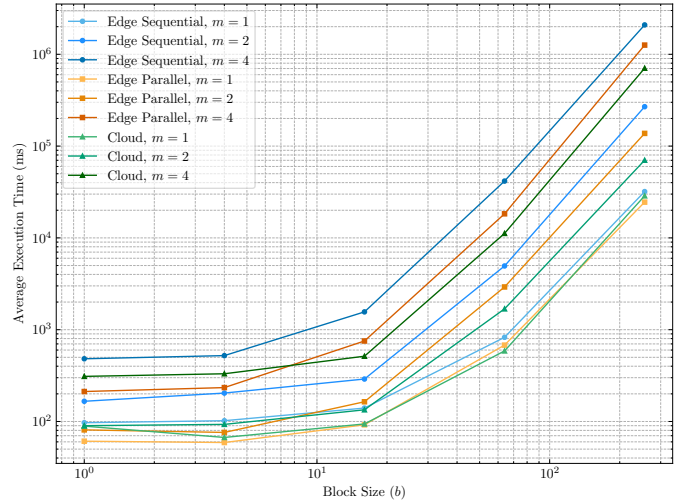

Figure 4: Experiment 2: Response time results.

Note that the three operation modes shown here are automatically transitioned to by the HP2C-DT framework depending on the needs. The synchronous character of each function is easily defined in the setup files of each node in the network. Plus, when asynchronous, the computing is (a) internally parallelized if possible, and (b) orchestrated and distributed to other resources available seamlessly.

Table 1: Required bandwidth (bytes/s) for `all` aggregation method.

Sampling Interval, T_s (ms)	Aggregation Interval, T_a (ms)				
	1	10	100	1000	10000
1	473009	90500	52250	48425	48043
10		47299	9050	5225	4843
100			4730	905	523
1000				473	91
10000					47

Table 2: Required bandwidth (bytes/s) for `phasor` aggregation method.

Sampling Interval, T_s (ms)	Aggregation Interval, T_a (ms)			
	10	100	1000	10000
1	49300	4930	493	49
10		4930	493	49
100			493	49
1000				49

6.3.3. Experiment 3: Scalability of HPC workloads

This experiment evaluates how well HPC resources scale when handling computationally intensive tasks. As a test case, we use the power grid data generator described earlier, applied to the IEEE 118-bus transmission model [28]. The experiment explores a variable space that includes the power delivered by the 53 converter, the proportion of grid-following vs. grid-forming operation for each converter, the power consumed by the 91 loads, and the control parameters of each converter. The computations are parallelized and distributed using the COMPSs framework and executed on the MareNostrum 5 supercomputer at BSC.

We conduct a strong scaling test, keeping the problem size constant while progressively increasing the computational resources. The test starts with a single node, which has 112 CPUs. The number of nodes is then doubled, progressing as follows: 1 node (112 CPUs), 2 nodes (224 CPUs), 4 nodes (448 CPUs), 8 nodes (896 CPUs), 16 nodes (1,792 CPUs), 32 nodes (3,584 CPUs), and 64 nodes (7,168 CPUs). The data generator explores the variable space down to a *depth* of 3 levels, with a *branching factor* of 4 children per parent subregion, and analyzes 170 operation points per subregion. This results in a total of 14,450 grid stability tasks, each lasting an average of 23.62 seconds. The exploratory nature of the problem and the subregion analysis make it a non-trivial parallel problem, as it requires careful coordination and cannot simply be treated as an embarrassingly parallel application.

Figure 5 presents the scalability results in terms of speedup and execution time. Each configuration is tested with five runs. The results show that speedup remains close to ideal up to 16 nodes. At 32 nodes, the speedup remains acceptable but deviates from the ideal curve, while at 64 nodes, performance saturation is evident. This behavior is expected in strong scaling tests, as increasing the number of nodes eventually leads to higher scheduling overhead. In this case, the need to effectively distribute tasks and aggregate results becomes more challenging, and the benefits of additional parallelism diminish at higher scales.

Beyond raw scalability, the practical impact of HPC resources is significant. With 32 nodes, the execution time is reduced to approximately one hour, compared to multiple hours on a single high-performance server. Even with 16 nodes, the execution takes below two hours. In a digital twin scenario, this level of performance would enable real-time grid stability assessments with hourly updates, as opposed to daily recalculations. This higher update frequency would significantly improve prediction accuracy and system responsiveness, which would be impractical in a cloud-based infrastructure due to cost and inconsistent performance.

7. Conclusion

This article presents a reference architecture that enables seamless coordination between physical devices, cloud nodes, and HPC resources. By distributing tasks based on their requirements—real-time processing at the Edge, global coordination in the Cloud, and intensive computation in HPC—the system balances efficiency and scalability. A key feature is the integration of HPC into the digital twin life cycle, allowing for large-scale probabilistic simulations and optimization tasks, alongside exclusive resource allocation and improved data privacy. At the same time, the architecture retains cloud-based tools for user interaction and global monitoring.

The HP2C-DT framework implements this architecture, not just specifying suitable technologies but also providing a flexible structure to integrate domain-specific components. Powered by COMPSs agents, each node in the network can offload tasks to peer edge/cloud nodes or HPC resources as needed. We test the framework through experiments of communication bandwidth, response time, and computing scalability, which demonstrate the framework's effectiveness in a power system application.

Future work will focus on decentralizing the architecture by organizing edge nodes into subclusters to improve availability and enable fail-resilient configurations. Additionally, efforts will focus on refining the lifecycle of digital twin models—whether physics-based or data-driven—to ensure their continuous adaptation and deployment across Edge and Cloud layers. This includes integrating adaptive machine learning capabilities that autonomously learn from sensed data, allowing real-time refinement of the digital twin.
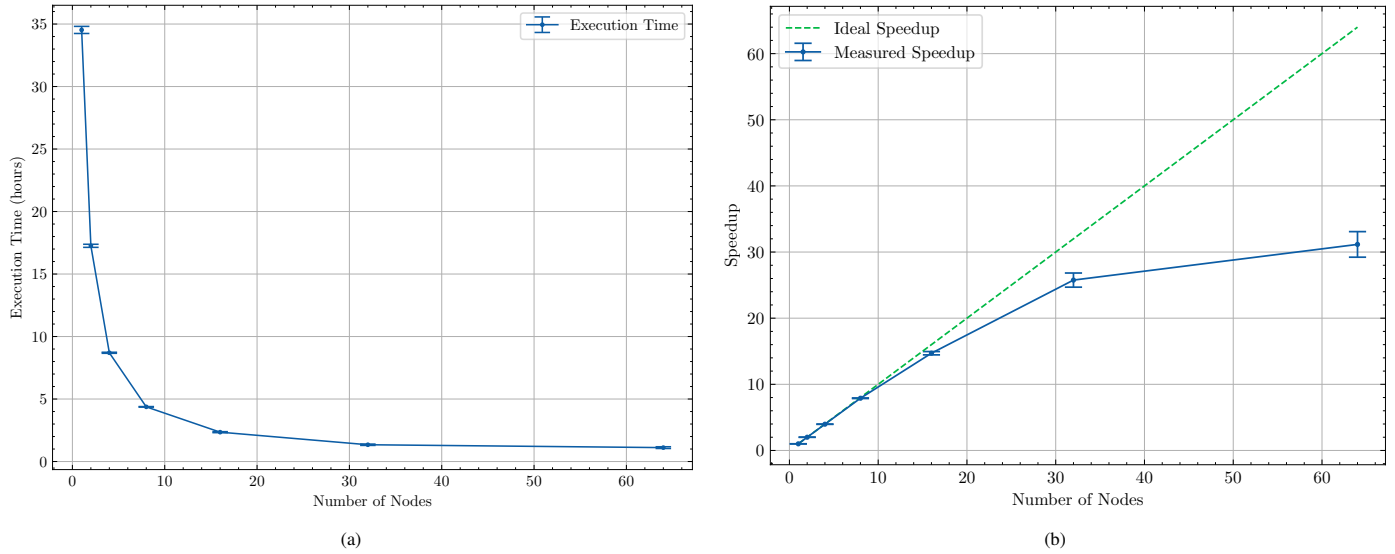
8. Acknowledgments

Figure 5: Experiment 3: Scalability test results. (a) Average execution time as a function of the number of nodes. (b) Speedup for the same range of cases. Each data point represents the average of 5 runs, with error bars indicating confidence intervals.

References

[1] Francesc Lordan, Enric Tejedor, Jorge Ejarque, Roger Rafanell, Javier Álvarez, Fabrizio Marozzo, Daniele Lezzi, Raül Sirvent, Domenico Talia, and Rosa M. Badia, *ServiceSs: An Interoperable Programming Framework for the Cloud*, Journal of Grid Computing, vol. 12, no. 1, pp. 67–91, March 2014. DOI: 10.1007/s10723-013-9272-5.

[2] Michael Grieves, *Digital Twin: Manufacturing Excellence through Virtual Factory Replication*, White Paper, 2014.

[3] Michael Grieves and John Vickers, *Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems*, Transdisciplinary Perspectives on Complex Systems: New Findings and Approaches, pp. 85–113, Springer International Publishing, Cham, 2017. DOI: 10.1007/978-3-319-38756-7_4.

[4] R. Stark and T. Damerau, *Digital Twin*, CIRP Encyclopedia of Production Engineering, Springer, Berlin, Heidelberg, May 2019. DOI: 10.1007/978-3-642-35950-7_16870-1.

[5] David Jones, Chris Snider, Aydin Nassehi, Jason Yon, and Ben Hicks, *Characterising the Digital Twin: A Systematic Literature Review*, CIRP Journal of Manufacturing Science and Technology, vol.29, pp.36–52, May 2020. DOI:10.1016/j.cirpj.2020.02.002.

[6] Qinglin Qi and Fei Tao, *A Smart Manufacturing Service System Based on Edge Computing, Fog Computing, and Cloud Computing*, IEEE Access, vol.7, pp.86769–86777, June. DOI:10.1109/ACCESS.2019.2923610.

[7] Milan Groshev, Carlos Guimaraes, Antonio De La Oliva, and Robert Gazda, *Dissecting the Impact of Information and Communication Technologies on Digital Twins as a Service*, IEEE Access, vol.9, pp.102862–102876. DOI:10.1109/ACCESS.2021.3098109.

[8] Vanderlei Munhoz, Antoine Bonfils, Márcio Castro, and Odorico Mendizabal, "A Performance Comparison of HPC Workloads on Traditional and Cloud-Based HPC Clusters," *2023 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pp.108–114, IEEE, Porto Alegre, Brazil, October. DOI:10.1109/SBAC-PADW60351.2023.00026.

[9] Thomas Geenen et al., *Digital Twins: The Journey of an Operational Weather Prediction System into the Heart of Destination Earth*, Procedia Computer Science, vol.240, pp.99–109. DOI:10.1016/j.procs.2024.07.013.

[10] S. Ares de Parga, J. R. Bravo, N. Sibuet, J. A. Hernandez, R. Rossi, Stefan Boschert, Enrique S. Quintana-Ortí, Andrés E. Tomás, Cristian Cătălin Tatu, Fernando Vázquez-Novoa, Jorge Ejarque, and Rosa M. Badia, *Parallel Reduced Order Modeling for Digital Twins Using High-Performance Computing Workflows*, arXiv, September 2024. DOI: 10.48550/arXiv.2409.09080.

[11] Enric Tejedor, Yolanda Becerra, Guillem Alomar, Anna Queralt, Rosa M. Badia, Jordi Torres, Toni Cortes, and Jesús Labarta, *PyCOMPSs: Parallel Computational Workflows in Python*, The International Journal of High Performance Computing Applications, vol. 31, no. 1, pp. 66–82, January 2017. DOI: 10.1177/1094342015594678.

[12] Qingwei Nie, Dunbing Tang, Changchun Liu, Liping Wang, and Jiaye Song, *A Multi-Agent and Cloud-Edge Orchestration Framework of Digital Twin for Distributed Production Control*, Robotics and Computer-Integrated Manufacturing, vol. 82, pp. 102543, August 2023. DOI: 10.1016/j.rcim.2023.102543.

[13] Do-Van Nguyen, Minh-Son Dao, and Koji Zettsu, *Digital Twin Orchestration: Framework and Smart City Applications*, CEUR Workshop Proceedings, 2022.

[14] Modbus Organization, *MODBUS Application Protocol Specification V1.1b3*, Accessed: March 28, 2025. Available online: https://modbus.org/.

[15] Docker Inc., *Docker*, Accessed: March 28, 2025. Available online: https://www.docker.com/.

[16] Francesc Lordan, Daniele Lezzi, and Rosa M. Badia, *Colony: Parallel Functions as a Service on the Cloud-Edge Continuum*, Euro-Par 2021: Parallel Processing, vol. 12820, pp. 269–284, Springer International Publishing, Cham, 2021. DOI: 10.1007/978-3-030-85665-6_17.

[17] Broadcom, *RabbitMQ*, Accessed: March 28, 2025. Available online: https://www.rabbitmq.com/.

[18] Gerard Ledwich and Arash Vahidnia, *Phasors for Measurement and Control*, Power Systems Series, Springer Cham, February 2021. DOI: 10.1007/978-3-030-67040-5.

[19] InfluxData Inc., *InfluxDB v1.x*, Accessed: March 28, 2025. Available online: https://docs.influxdata.com/influxdb/v1/.

[20] Django Software Foundation, *Django: A Web Framework for Python*, Accessed: March 28, 2025. Available online: https://www.djangoproject.com/.

[21] Grafana Labs, *Grafana OSS*, Accessed: March 28, 2025. Available online: https://grafana.com/.

[22] OPAL-RT Technologies Inc., *OP4512: Compact Entry-Level RCP/HIL FPGA-based Real-Time Simulator*, Accessed: March 28, 2025. Available online: https://www.opal-rt.com/hardware-overview/.

[23] OPAL-RT Technologies Inc., *HYPERSIM*, Accessed: March 28, 2025. Available online: https://www.opal-rt.com/hypersim/.

[24] Adejuyigbe O. Fajemisin, Donato Maragno, and Dick den Hertog, "Optimization with constraint learning: A framework and survey," *European Journal of Operational Research*, vol. 314, no. 1, pp. 1–14, Elsevier, 2024.

[25] Florian Thams, Andreas Venzke, Robert Eriksson, and Spyros Chatzivasileiadis, "Efficient database generation for data-driven security assessment of power systems," *IEEE Trans. Power Syst.*, vol. 35, no. 1, pp. 30–41, IEEE, 2019.

[26] Francesca Rossi, Eduardo Prieto Araujo, Marc Cheah Mañe, and Oriol Gomis Bellmunt, "Data Generation Methodology for Machine Learning-based Power System Stability Studies," *2022 IEEE PES ISGT-Europe*, pp. 1–5, IEEE, 2022.

[27] Santiago Peñate Vera et al., *GridCal: Better Topology and Better ACDC Power Flow (Version: GridCal v5.3.0)*, Zenodo, January, DOI:10.5281/zenodo.14615064.

[28] Ivonne Pena, Carlo Brancucci Martinez-Anido, and Bri-Mathias Hodge, "An Extended IEEE 118-Bus Test System With High Renewable Penetration," *IEEE Transactions on Power Systems*, vol.33, no.1, pp.281–289, January. DOI:10.1109/TPWRS.2017.2695963.

[29] Yan Zhang, *Digital Twin: Architectures, Networks, and Applications*, Simula SpringerBriefs on Computing, vol. 16, Springer Nature Switzerland, Cham, 2024. DOI: 10.1007/978-3-031-51819-5.