Computers in Industry 149 (2023) 103918

Contents lists available at ScienceDirect

Computers in Industry

journal homepage: www.sciencedirect.com/journal/computers-in-industry

Requirements and design patterns for adaptive, autonomous, and context-aware digital twins in industry 4.0 digital factories

Paolo Bellavista [a], Nicola Bicocchi [b], Mattia Fogli [c], Carlo Giannelli [d,*], Marco Mamei [e], Marco Picone [e]

[a] Department of Computer Science and Engineering, University of Bologna, Italy
[b] Department Engineering "Enzo Ferrari", University of Modena and Reggio Emilia, Italy
[c] Department of Engineering, University of Ferrara, Italy
[d] Department of Mathematics and Computer Science, University of Ferrara, Italy
[e] Department of Sciences and Methods for Engineering, University of Modena and Reggio Emilia, Italy

ARTICLE INFO

Keywords:
Industry 4.0
Digital factory
Digital twin
Design patterns
Micro-services

ABSTRACT

Digital factories are poised to achieve unseen levels of resiliency and flexibility, facing increasingly demanding requirements by customers and market conditions. Digital twins are one of the building blocks fueling this vision. They provide a software counterpart for industrial assets enabling control, simulation, analytics and "servitization" functionalities. To effectively fulfill their tasks, digital twins need to embed adaptive, autonomous, and context-awareness functionalities. In this work, we propose an organic vision of digital twin design and implementation with the goal of clearly identifying the primary steps towards this goal. First, we detail how current requirements for digital twins have to be enriched for supporting adaptivity, autonomy, and context-awareness. Second, we propose a set of reusable design patterns mostly popularized in the field of micro-services allowing engineers to meet these new demanding requirements while keeping complexity and management costs under control. Finally, we present our working prototype based on the identified design patterns and implemented with orchestrated micro-services, demonstrating the feasibility of our solution and quantifying its networking and computational overhead.

1. Introduction

Manufacturing companies are increasingly confronted with several challenges, such as shorter product lifecycles, demanding quality standards, sustainability, mass customization, and servitization of physical products. To cope with such issues, the Industry 4.0 guidelines are pushing towards industrial environments composed of heterogeneous machines provided by different manufacturers and interacting one each other in an articulated and flexible manner (Corradi et al., 2019). Digital Twins (DTs) and digital factories are two concepts aiming at alleviating these challenges. DTs can be conceptualized as comprehensive, actionable, digital representations of physical systems (Minerva et al., 2020; Jones et al., 2020; Vuković et al., 2021; Bellavista et al., 2021) providing a software copy of a Physical Object (PO) reflecting its properties, behaviors, and relationships according to the operational context. The physical and software counterparts cooperate and co-evolve for enabling features such as device control, simulation, analytics, and, more generally, the ability to enhance the functionalities of POs. The DT capability of carrying out simulations may help system design (Leng et al., 2021a), improve system flexibility (e.g., rapid reconfiguration to meet ever-changing product orders Leng et al., 2020), and push towards cost-saving (e.g., remote semi-physical commissioning of manufacturing systems Leng et al., 2021b).

The adoption of DTs pushes for a simplified interaction among software and hardware modules. DTs not only allow to hide heterogeneity of machines, but also to make easier their monitoring and reconfiguration. Such aspects are of particular importance, considering that modern industrial environments are characterized by fast-changing requirements. This requires a significant configuration/reconfiguration effort, which is typically carried out manually by practitioners (Bolender et al., 2021). If compared with the recent past, plants are nowadays reconfigured much more frequently, with the ultimate goal of supporting a higher degree of differentiated production, also changing at runtime without significant stopping intervals (Liu et al., 2022b). More specifically, DTs increase the **adaptivity** of industrial environments by interacting with each other (e.g., through uniform interfaces Platenius-Mohr et al., 2020) and providing high-level abstractions for external

* Corresponding author.
E-mail addresses: paolo.bellavista@unibo.it (P. Bellavista), nicola.bicocchi@unimore.it (N. Bicocchi), mattia.fogli@unife.it (M. Fogli), carlo.giannelli@unife.it (C. Giannelli), marco.mamei@unimore.it (M. Mamei), marco.picone@unimore.it (M. Picone).

https://doi.org/10.1016/j.compind.2023.103918
Received 15 April 2022; Received in revised form 16 September 2022; Accepted 3 April 2023
Available online 17 April 2023
0166-3615/© 2023 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

services and applications towards the underlying industrial machines. However, it is worth noting that while the adoption of DTs within industrial environments might simplify the interaction with floor-level equipment (by hiding low-level technical details) and increase their flexibility, it does not guarantee easier management of the environment as a whole. The spread of DTs, in fact, shifts part of the complexity of managing POs to the management of DTs.

To further simplify the management of industrial environments, goal-oriented central control has to be intertwined with some degree of distributed **autonomy** (Azarmipour et al., 2020). In fact, while production managers should be in charge of dictating high-level objectives for the whole industrial environment (e.g., by specifying to devices which product should be crafted and the number and size of lots that should be realized), DTs should be able to autonomously pursue the provided goals by re-configuring industrial machines in a distributed manner, either on their own or as the result of collective interactions (Ding et al., 2019). In other words, production goals are pre-determined and provided in a top-down manner, while DTs (and related PO counterparts) cooperate to pursue such goals by autonomously adapting their behaviors in a distributed manner, also considering the time-varying state of the runtime industrial environment and with little or no centralized coordination.

Such adaptive manufacturing systems require forms of **context-awareness** (Alexopoulos et al., 2016) to support DT autonomous decision-making at the plant production level, based on goals provided by the plant management level. In particular, DTs have to take into account their internal state, the state of their associated machines, and also the distributed state of cooperating machines. More in general, the state of the entire industrial environment at different abstraction levels (from the temperature/vibration of the surrounding physical environment to the currently available network bandwidth) is increasingly needed as the level of required context-awareness increases.

This paper intends to contribute to the evolution of DTs by providing practical guidelines for the design, implementation, and management of adaptive, autonomous, and context-aware DTs in industrial scenarios. In this regard, the contribution of this paper is threefold:

1. We tackle the problem of merging established requirements for DTs with the essential capabilities of adaptivity, autonomy, and context-awareness. Due to their transversal nature, we do not treat adaptivity, autonomy, and context awareness as additional requirements per se but, instead, we elaborate in which ways current requirements, especially those elaborated in Minerva et al. (2020), might be articulated for supporting them.
2. We investigate design patterns and reusable solutions concerning both single components and the whole architecture which can be applied for engineering the DT lifecycle, thus allowing the implementation of increasingly large and complex deployments while keeping operating costs manageable. We also originally present how well-known design patterns for software architectures can be applied to the design and implementation of DTs.
3. We demonstrate the feasibility of our proposals by emulating an industrial environment in which next-generation containerized DTs build on top of documented design patterns and Kubernetes enable adaptive, autonomous, and context-aware behaviors within both single DTs and the system as a whole.

The remainder of the paper is organized is follows. In Section 2, we briefly introduce most relevant aspects of modern industrial environments and how DTs can improve their flexibility. In Section 3, we discuss the requirements for adaptive, autonomous, and context-aware DTs in light of the current consensus about the requirements for general purpose DTs. In Section 4, we illustrate how the identified requirements can be implemented by conceptualizing DTs as containerized software components and making use of the design patterns popularized in the field of micro-services. In Section 5, we use an emulated industrial environment constructed around Kubernetes for discussing feasibility, benefits, and drawbacks of the proposed approaches. Finally, Section 6 presents related work in the field and Section 7 concludes the paper and draws final remarks.

2. Digital twins in modern industrial environments

To better present our solution for DT management in Industry 4.0, this section introduces the primary characteristics of modern industrial environments and how the adoption of DTs can improve their flexibility. Modern industrial environments are typically modeled in three layers: shop floor, plant, and enterprise.

The *shop floor level* hosts industrial machines and is mainly focused on industrial automation. Industrial machines are equipped with sensors (e.g., working temperature, pressure) and actuators (e.g., drills, presses). Industrial machines tend to have extremely long lifetimes (between 10 and 15 years, if not even longer in some cases) and may implement different (proprietary) protocols. In addition, software upgrades may not always be possible, since manufacturers usually forbid software upgrades for safety reasons, or industrial machines may not support them at all. Up-to-date industrial guidelines for cyber security (e.g., IEC 62443, 2013) recommend splitting the network topology into several shop floor subnets accessing the backbone via dedicated gateways, as depicted in Fig. 1. In addition, each shop floor subnet is also composed of Industrial Internet of Things (IIoT) devices and Edge Nodes (ENs). In contrast to industrial machines, IIoT devices are characterized by a substantially shorter lifetime, usually communicate via well-known protocols, and support monitoring and control capabilities while being low cost. It is worth mentioning, however, that IIoT devices also present complex chains of software dependencies (e.g., third-party libraries), thus making integrity mechanisms challenging to be guaranteed (Maggi and Pogliani, 2017). Instead, ENs provide relatively high computational and memory capabilities on the premises (Shi and Dustdar, 2016), close to industrial machines. This may improve, among others, data protection (e.g., by processing sensitive data on-premises rather than in the cloud), real-time responsiveness (since latency at the edge is much lower than at the cloud), and traffic management (e.g., by enforcing fine-grained control over mission-critical traffic flows as they traverse the industrial network) (Fogli et al., 2022a,b).

The *plant level* regards the management of manufacturing processes. The critical component is the Manufacturing Execution System (MES), allowing information flowing upstream and downstream between the shop floor level (where industrial machines produce goods) and the enterprise one (where managers make decisions). In particular, the MES receives instructions about how industrial machines should behave from operators, and then it transmits such instructions downwards, i.e., towards the shop floor.

The *enterprise level* is about making decisions on how to plan business operations. In this regard, decision-makers rely on the Enterprise Resource Planning (ERP), collecting information about supply chains, cash flows, customer orders, and production processes, to decide what, when, and how many products should be crafted.

The control room subnet comprises both plant and enterprise components (i.e., MES and ERP), plus a subnet gateway. In addition, the networking side is evolving towards articulated topologies characterized by backbones connecting several subnets through multiple communication channels to increase performance and fault tolerance.

Typically, industrial machines are configured to receive control messages from the control room (by the MES above all) and send back a few information about their current state (e.g., number of crafted and faulty products). Sporadically, industrial machines also exchange messages with each other, e.g., machines in the same production line share information about the rate of crafted products. In any case, once industrial machines and companion control servers are deployed, their dynamic reconfiguration is not possible, e.g., requiring to stop production to reroute messages towards a different control server. On the

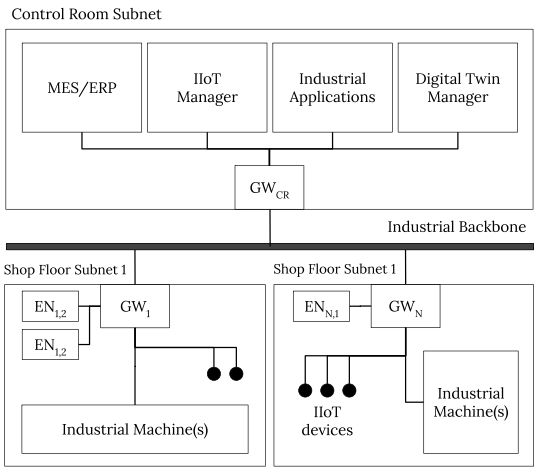


Fig. 1. Target environment. The overall topology is split into multiple subnets, connected via gateways. The control room subnet hosts nodes monitoring and controlling industrial machines and ENs, which run in the shop floor subnets.

contrary, modern industrial environments need to dynamically manage machines and message flows, even at service provisioning time and without imposing any pause to the production process. For instance, since servers can be easily replicated and migrated while industrial machines relocated, it is not possible anymore to support only static deployments of software modules.

As introduced in Section 1, we believe that the adoption of DTs may relevantly improve the flexibility of such environments in terms of autonomy, adaptivity, and context-awareness. Let us note that despite these concepts have been already introduced in the field, no literature specifically addresses adaptivity, autonomy, and context-awareness together in relation to DTs (Hribernik et al., 2021). Most research work in the current literature, in fact, provides specific use cases and applications (Hinduja et al., 2020; Jiang et al., 2021) instead of attempting a systematic definition of their profound meaning in the field of DTs, and the consequences of their formal introduction among other well-established requirements (Minerva et al., 2020; Jones et al., 2020; Fuller et al., 2020).

To promote the large scale adoption of the technology, DTs need to evolve towards active software entities capable of extending the capabilities of their PO counterparts, sensing their environment, proactively communicating with each other, and taking decisions towards cooperative goals, with the paramount objective of adapting themselves and their counterparts to achieve those goals (Hribernik et al., 2021). As a consequence, DTs need to be designed, implemented, and operated as mature software components that embed increasing levels of automation and standardization (Al-Sehrawy and Kumar, 2020; Tekinerdogan and Verdouw, 2020). In particular, we claim that DTs should be implemented and managed by following the micro-services approach (Dinh-Tuan et al., 2019; Dobaj et al., 2018; Ghosh et al., 2021) together with containerization. The primary goal is to de/activate and move software modules among nodes in a completely transparent manner from a machine point of view, e.g., by migrating (part of) a DT from the control room to a shop floor subnet. The adoption of containerization allows to easily split software modules among different components managed in a independent manner one each other. In addition, containerization allows to redeploy software modules to pursue a common goal, by taking into consideration business objectives as well as QoS requirements. In other words, such solution ensures the flexible management of software module (re-)deployment (and also decommissioning) to cope with the dynamic nature and objectives of modern industrial environments.

To support the dynamic lifecycle of DTs in an effective manner and take full advantage of the micro-services approach, there is the need of easily interacting with DTs via well-known interfaces. To this purpose, Fig. 2 presents the overall architecture and primary interfaces of the proposed DTs. The Digital Twin Service is hosted within a Digital Twin Container, in charge of enabling container de/activation and migration. In addition, the Digital Twin Service contains and manages the DT model of the machine (i.e., PO State, Design, Configuration, and Behavior) and interacts with external entities via four interfaces.

The physical and digital interfaces are respectively responsible for communicating with machines and digital services running industrial applications. The digital interface is also used for actuating the manufacturing directives coming from the MES and eventually translated by a dedicated software layer (i.e., the Digital Twin Manager). The storage interface concern is managing past and also future representations of the PO. Finally, the management interface exposes the state of the DT itself thus allowing context-aware, adaptive behaviors mediated by the container orchestrator. Let us anticipate that by clearly defining the primary modules and interfaces of DTs, it is possible to easily deploy them as containerized software modules, with the notable benefit of allowing to manage their lifecycles by exploiting orchestration solutions.

3. Requirements for next-generation industrial digital twins

Although DTs used in industrial applications differ in technical and operational details (Hinduja et al., 2020; Jiang et al., 2021), efforts have been made to define their general properties. In particular, Minerva et al. (2020) recently proposed a set of key requirements summarizing the role of DTs in the cyber–physical systems domain. DTs are characterized as event-driven entities capable of storage capabilities to maintain a log of status changes and offering methods and functionalities to other services mediated by application programming interfaces. Due to their generality, these requirements are focused on functional aspects and do not clearly address the specific role and meaning that adaptation, autonomy, and context-awareness acquire in this context. Moreover, DTs current implementations are still conceived as mostly passive entities representing the state of a PO or, alternatively, as cloud-oriented, platform-specific components delegating the integration of physical and software entities to tailored solutions, fragmented software layers, or even to POs themselves (Fuller et al., 2020). These approaches are frequently based on loose standards and implemented without documented design patterns, limiting reusability and interoperability, as well as increasing long-term maintenance costs (Tekinerdogan and Verdouw, 2020; Washizaki et al., 2020).

We argue that, for supporting dynamic and flexible industrial environments, these aspects need to be addressed and formalized. Because of this, in this section, we better articulate the idea of containerized DTs running within a network of orchestrated components by enriching well-established requirements with a set of additional constraints for supporting adaptation and autonomy both at the DT and the factory levels. It is also worth clarifying that we do not intend to introduce brand-new requirements. Instead, we propose how to apply software engineering design patterns to translate widely accepted requirements in the field, e.g., the ones formerly introduced by Minerva et al. (2020), into actionable tools usable as a foundation for context-aware, adaptive, autonomous DTs.

3.1. Reflection

Definition: The reflection property describes a DT as an entity which *mirrors the behavior and the status of the PO. Each change in status, each event faced by the PO is reflected by the DT. Changes that occur to the DT should be reproduced in the PO.*

Engineering: **(R1)** The DT has to be capable of discovering available POs present within the execution environment and consequently handle the communication and interaction according to the supported protocols and data formats. For example, a DT supporting a specific type of machine should autonomously search for supported entities and either

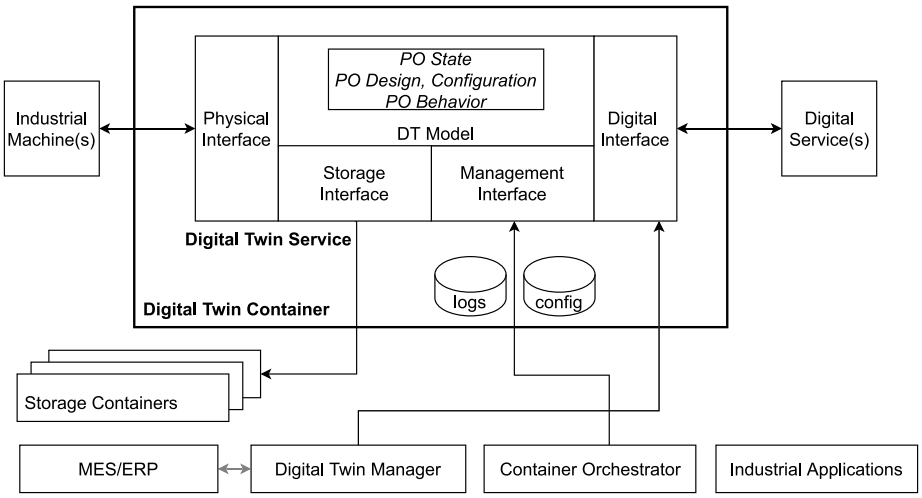


Fig. 2. Representation of a containerized DT. The DT model manages the properties associated with the linked machine. The physical and digital interfaces manage communications. The storage interface stores and retrieves past and future states of POs. The management interface exposes the state and configuration of the container to other services.

establish a permanent connection (i.e., enabling reflection) or ease its configuration process. **(R2)** The DT has to be aware of the quality of the reflection it provides to digital services. This notion, generally identified as *entanglement*, can promote adaptive behaviors aimed at providing determined service levels such as tuning communication protocols, throttling external requests, or even migrating the DT on the basis of internal or environmental conditions.

Impact: The availability of DTs capable of delivering adaptive reflection represents a fundamental enabler towards their use as autonomous entities instead of passive digitalized replicas of POs. Each DT has to be in charge of tasks concerning the reflection requirement, such as discovering the supported PO counterparts, identifying their properties, and maintaining the desired level of entanglement according to internal and environmental conditions. Additionally, DTs can autonomously detect and react to eventual issues (e.g., adapting networking configuration for increasing or decreasing entanglement). Eventually, DTs can also notify external observers about misalignments with the PO. It is worth noting that the autonomous discovery of compatible POs has benefits cascading to other requirements. For example, a composed DT representing thermal-related features of a smart-building might search and connect to all the compatible devices of the building, or their associated DTs, without time-consuming manual interventions.

3.2. Persistency

Definition: The persistency property defines *a DT as an entity which is always available. Its availability exceeds the actual existence of the PO. A DT could be available before the creation, during malfunctioning and crashes, and after the end of life of the PO*.

Engineering: **(R3)** The DT must be resilient and thus organized in decoupled and independent components, represented in Fig. 2, so that a localized fault does not compromise the entire container. **(R4)** The DT has to be highly available, i.e., it must support replication in response to failures, both internal (e.g., the DT model fails) or environmental (e.g., the node running the DT container fails). **(R5)** To minimize the effects of such events, DTs must support autonomous re-configuration. Indeed, their configuration have to be remotely stored and fetched when needed. In this manner, replicas, instead of restarting with the same configuration of a failed container, can eventually retrieve an alternative version.

Impact: Once we adopt DTs to decouple applications from the complexity and fragmentation of the physical layer, we establish an hidden agreement between these two levels. Digital services rely on DTs to interact with POs and any disruption in their functioning potentially represent a critical issue (e.g., the plant control room that suddenly stops receiving telemetry from deployed robots). These requirements represent a crucial pillar to build reliable and dependable networks of DTs.

3.3. Memorization

Definition: The memorization property defines a DT as an entity *storing all the status changes and events occurred to the PO. A DT represents the status of the PO over time and space*.

Engineering: **(R6)** The DT must be able to maintain the current state of the PO internally, acting as a cache between POs and digital services. Indeed, for improving autonomy and minimizing response times to digital services, the current state of the PO has to be held within the DT itself, without using external storage services. **(R7)** The DT must manage the entire history of states and events involving the PO in a context-aware fashion. Furthermore, the DT has to manage different loads of requests from digital services. Thus, the DT has to support different replication strategies for its storage services and autonomously select the most suitable one.

Impact: Observing and efficiently interacting with a PO not only at its current state but also through the navigation of its historical data via a uniform interface segregates responsibilities and has the potential to significantly simplify the design of applications. Memorization can be also exploited to support context-awareness and adaptation directly on DTs via machine learning algorithms capable of predicting future states from past states. Furthermore, the resulting outputs (i.e., the predicted future states) can be memorized within the same data structure, enabling a forward navigation in the predicted "future" of the DT.

3.4. Augmentation

Definition: The augmentation property defines a DT as an entity which *can extend the PO functions and offer them by means of APIs. Augmentation can add new functionalities that the PO does not support or provide access to data in particular formats*.

Engineering: **(R8)** The DT has to be expandable (adaptive) with additional functionalities by supporting dynamic configuration. For example, a complex DT supporting multiple POs or functionalities could be deployed with different configurations on different nodes depending on their resources. The configuration must be updated, without requiring manual interventions, whenever the DT is migrated to a node with different capabilities. **(R9)** The DT must support software updates. At the most basic level, both POs and digital services might

receive updates over time, thus requiring changes in the DT to keep it functional. Furthermore, updates enable the addition of augmentation capabilities to the DT itself without the need of a re-deployment.

Impact: The possibility to easily and dynamically extend the capabilities provided by a DT with respect to the original associated PO counterpart represents the fundamental characteristic of DTs and the reason why they should be seen and modeled as active software components with an independent behavior. Through a dynamic augmentation it is possible to extend interoperability without changes on the PO or without requiring DT re-deployment (e.g., to support new protocols or data formats). It can also allow to introduce intelligent and cognitive functionalities directly on the DT for optimizing both digital and physical layers (e.g., reducing the speed of a production robot or moving the DT to a different EN to improve performance).

3.5. Composability

Definition: The composability property defines a DT as an entity which *must support the correlation of different elementary DTs into complex organizations and provide views on the aggregated DT and individual components*.

Definition: **(R10)** The DT has to be able to manage other DTs as if they were POs. Each change in any DT which is part of a composition scheme (i.e., an observed DT) is communicated to an observing DT. In this way, as soon as one DT detects a change in its PO, the change is propagated towards the upper levels of the composition scheme. Alternatively, in case the composed DT is not observed by any digital service, the lower levels might choose not to communicate the changes to save bandwidth. The same principles have to be also applied to commands that can be propagated from the composed DT to the underlying DTs in order to modify and actuate the physical environment.

Impact: The communication scheme used for composition is strictly tied to reflection, entanglement, and adaptive capabilities. In fact, being a distributed communication scheme, it might require remarkable network resources to guarantee acceptable entanglement levels to digital services observing composed DTs. To avoid network overload, DTs participating in a composition scheme have to coordinate to dynamically select a suitable trade-off between entanglement and networking resources.

3.6. Replication

Definition: The replication property defines a DT as an entity which *can be replicated to serve the needs of different applications. Replicas of the same PO must behave consistently, i.e., they cannot have a different status and they cannot exhibit different behaviors*.

Engineering: **(R11)** The DT, eventually leveraging the container orchestrator, must support replication for facing variable loads of requests coming from digital services. However, as soon as replicas are spawned, two different communication schemes become possible: peer and master–slave. The former implies that all replicas of the same DT communicate directly with the PO. The latter implies that the group of replicas elects a master responsible of managing the PO, while all the others, behaving as slaves, communicate with the master to receive updates about the state of the PO. As a consequence, the DT has to be aware of internal and environmental conditions for autonomously selecting the most suitable approach.

Impact: The DT leverages its awareness of the computational environment for autonomously selecting the most suitable replication scheme. As an example, when dealing with constrained POs, the master–slave approach could be preferred. In contrast, when powerful POs are involved, the peer approach might reduce communication overhead among replicated DTs and avoid all the intricacies of master-election distributed protocols. The same flexibility can also be exploited to handle different visibility and responsibilities levels, allowing to segregate the DT authorized and in charge of communicating with the PO (master) from the others (slaves), which, for example, might be limited to specific interactions with digital services and unauthorized to directly interact with the physical layer.

3.7. Accountability/manageability

Definition: The accountability property defines a DT as an entity which *allows to determine its status and activities and to optimize its execution in the framework in which it is operating. It must provide information about the usage of the PO by the applications associated with it*.

Engineering: **(R12)** The DT has to be observable. Indeed, the DT must be not only aware of its state but also make it available via standard interfaces (e.g., RESTful APIs, event-driven communication patterns). Additionally, the complete event history concerning the DT (i.e., execution logs) has to be exposed in a similar fashion.

Impact: Observability pushes adaptation outside the DT itself. For example, the orchestrator could detect a DT running on limited resources and respond by either migrating it to another node or spawning a replica for the sake of guaranteeing the required entanglement level. Additionally, the availability of the event history concerning the DT enables long-term analytics based on machine learning algorithms, such as failure prediction or anomaly detection.

4. Design patterns

Industrial networks can be composed of several devices, possibly thousands of heterogeneous devices interacting one each other. To exploit DTs as universal interfaces for both monitoring and control, the requirements identified in the previous section have to be engineered in a reliable and dependable way, by following norms and well-documented practices (Wedyan and Abufakher, 2020). In this section, we show how design patterns for both monolithic applications and micro-services can be used for achieving such goals.

4.1. Patterns for single-node, single-container services

Despite we envision industrial deployments of DTs as a distributed network of containerized entities, the DT itself still has to be developed as a monolithic service. Nevertheless, since component isolation (R3, R6) and extensibility (R9) are key requirements, we propose to use the *microkernel pattern*, one of the most modular approaches in the realm of patterns for monolithic services.

The *microkernel pattern* consists of two types of components: a core system (i.e., the DT model) and plug-in modules (i.e., the physical, digital, management, and storage interfaces) as shown in Fig. 3. The core system, usually containing only the minimal functionalities required to make the system operational, manages the state, configuration, and behavior of the PO (R6). The plug-in modules, instead, are independent components enhancing or extending the core system with additional capabilities without the need of re-deployments (R9). As an example, the storage plug-in provides a clear boundary between the management of the present state of the PO which happens inside the DT core and past and predicted states which are, instead, externalized (R7). Generally, plug-in modules should be independent one each other and can be connected to the core in a number of different ways, from point-to-point binding (i.e, the core accepts an object instance of a plug-in) to messaging. This kind of architecture provides decoupled operations and prevents generalized failures (R3). Indeed, the failure of one plug-in does not determine the failure of the whole container. To further improve isolation and decoupling, we propose to implement messaging via an *asynchronous queuing pattern*. The introduction of asynchronous queues allows extremely evolvable and resilient architectures. In fact, protection mechanisms against bursts or sustained rates of excessive requests can be transparently embedded within the queue themselves, thus simplifying the development of the other components.

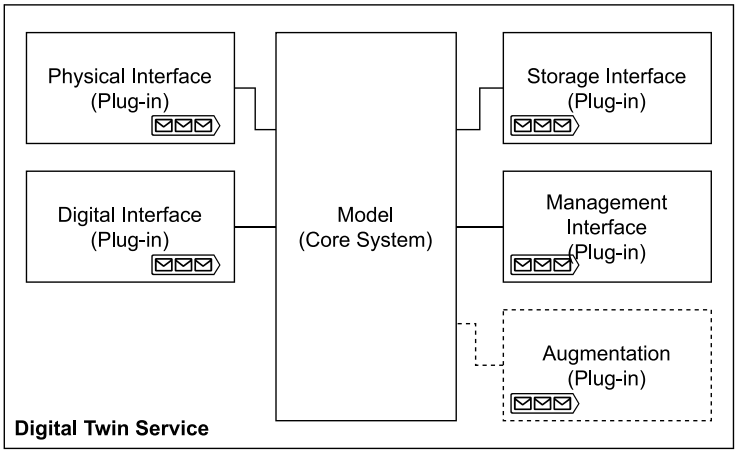


Fig. 3. A DT implemented with the microkernel pattern. The core system containing the state and the logic for interacting with the PO is enhanced by plug-ins implementing additional functionalities.

4.2. Patterns for single-node, multi-container services

Even if there are valid reasons to decompose DTs in multiple components (i.e., core system and plug-ins), there are also reasons for breaking up DTs into multiple containers. Imagine a DT supporting dynamic configuration and exposing its configuration via a standard API. One approach could be to add a specific plug-in to the microkernel architecture described above, while another approach could be to break up the DT into two separate containers: one running the DT itself (i.e., core and basic plugins) and the other running a dynamic configuration daemon. While the former is perfectly legitimate, the latter case has notable advantages. Containers, in fact, establish boundaries around resources (e.g., 8 GB of memory, 6 cores), teams (e.g., one team owns one container image), and concerns (e.g., this image provides dynamic configuration). As an example, using multiple containers allows to assign them different priorities and resource requirements, e.g., ensuring that the configuration daemon uses computing resources only when the DT is offloaded. In addition, containers represent a relatively small and focused piece of code managed by a single team and usually they can be updated, tested, and deployed more easily than complex, monolithic services. Containers can also be easily reused across multiple teams and applications, often leading to high-quality implementations, since they are built once and used in different contexts. For these reasons, we make the case of DTs conceived as multi-container entities, namely *pods* (a term introduced in Kubernetes, currently one of the most prominent container orchestrators). The three patterns we discuss here, represented in Fig. 4, propose to deploy the DT container along with a secondary container responsible for different tasks. In addition to being scheduled on the same machine, the two containers are assumed to have access to shared resources, such as the filesystem and network interfaces.

The *sidecar pattern* considers two containers: the application container (i.e., the DT container) and the sidecar container, augmenting the application usually without accepting or establishing network connections on its behalf. In its simplest form, a sidecar container can be used to add functionalities to a container that might otherwise be difficult to improve. In more articulated cases, sidecars can be used to engineer multi-container services which are inherently more robust and scalable than those structured in a single container. Remote configuration (R5), requiring DTs to store and retrieve their configuration from a remote server, can be implemented with a sidecar container monitoring the configuration files of the DT. The sidecar is responsible for keeping aligned local and remote configurations. If the remote configuration diverges from the local one, it downloads the new configuration and notifies the DT to reconfigure itself using the updated files. Similarly, software updates (R9) can also be implemented using a sidecar container. As an example, it is possible to use a containerized daemon which periodically downloads changes from a git repository, updates the local code of the DT (e.g., the folder containing plug-ins), and triggers the DT to restart itself. As a consequence, pushing updates to a git repository results in updated code being deployed to the running DT in a simple yet reusable fashion.

The *ambassador pattern* uses an ambassador container to act as a broker among the application container and external services. Similarly to sidecars, ambassadors are paired to the primary container and scheduled on the same node. Requirements concerning adaptive reflection (R2) can be implemented using both the ambassador and the microkernel patterns in a complementary fashion. For example, they could be either implemented within the communication plug-ins of the DT (i.e., physical/digital interface plug-ins) or delegated to a specialized daemon running within the ambassador. In the latter case, the communication plug-ins of the DT act as basic network proxies and delegate external connections entirely to the ambassador container. As an intermediate solution, an ambassador could be used for enhancing a DT providing only basic reflection capabilities with more advanced properties, such as autonomously switching among different communication protocols (R2) or automatically searching compatible POs (R1). In addition, an ambassador can be used for providing the communication interfaces of the DT with additional layers of protection from failures of other services. For example, protection patterns such as *throttling*, *circuit breaker*, or *retry* (R3) can be easily implemented within ambassadors without the need of modifying and re-deploying the DT container.

Ambassador containers are not limited to function with digital or physical interfaces. Indeed, they can be used for brokering any connection to external services. The storage interface, for example, is designed for storing and retrieving from external storage services past and future states of the PO. As expressed in (R7), the DT has to be capable of dealing with high request loads from digital services asking for past or future states of the PO as well as with large bodies of data representing its entire history. These functionalities can be implemented within ambassador containers as depicted in Fig. 5. In case of high load of requests, data can be statelessly replicated so that each replica manages the whole history of the DT, thus scaling up the number of manageable requests. Alternatively, in case of large bodies of data, sharding (i.e., partitioning based on content, for example, each shard contains one year of history) can be applied. This approach does not necessarily unload the storage services (i.e., all requests might ask for the same shard) but, instead, allows for scaling up the size of DT's past and future states. In Fig. 5, the two cases are managed by separate ambassadors. Nevertheless, it would be possible to implement both

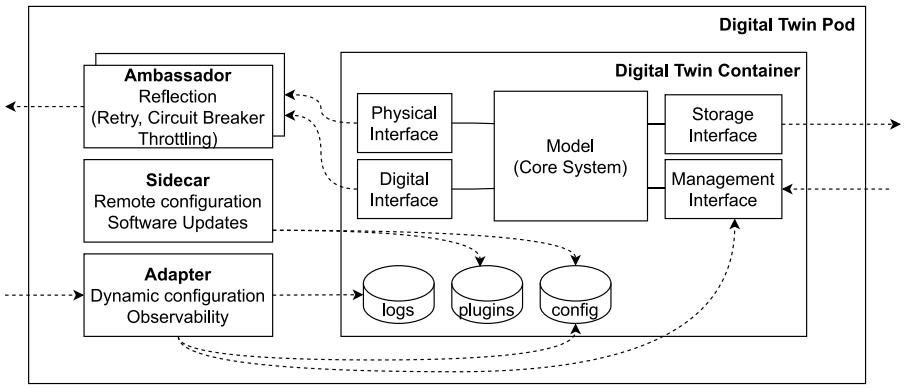


Fig. 4. A DT conceived as multi-container entity using a secondary container for enhancing the primary container. Ambassadors mediate external communications, adapters expose uniform interfaces to other services, while sidecars often act locally.

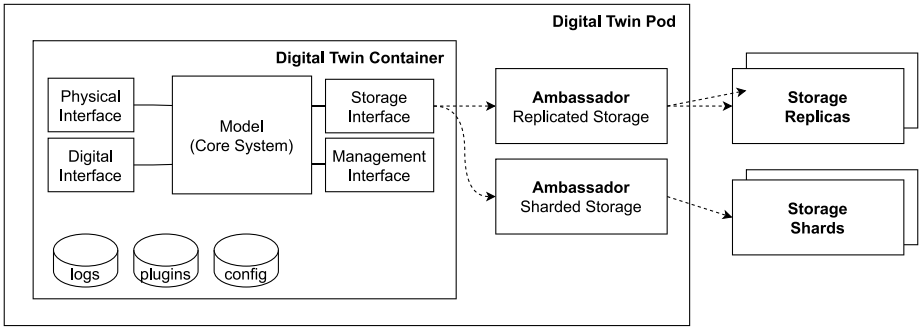


Fig. 5. Two ambassador containers brokering the connection between the storage interface of the DT and two external services adopting different replication strategies.

replication strategies in a single container, capable of choosing the most suitable one.

The *adapter pattern* is used to modify the interface of the primary container so that it conforms to a predefined interface. For example, an adapter might ensure that an application implements a consistent monitoring interface (i.e., all logs saved using the same format). The observability (R12) and dynamic configuration (R8) requirements can be implemented using this approach. Indeed, instead of modifying the DT model or adding plug-ins, a dedicated daemon could be run inside an adapter container. In regard to the observability requirement (R12), a daemon could monitor the logs produced within the DT container and expose them via standard APIs. Accordingly, also the internal state of the DT (exposed via its management interface) can be monitored and exposed to external services, such as the container orchestrator using the same interface. Large factories running massive deployments could greatly benefit from this containerized approach. For example, it could be possible to deploy any kind of DT, possibly produced by different vendors, and then make them uniformly observable by adding a properly crafted adapter to their pod. Dynamic configuration (R8) shares many similarities with remote configuration (R5). A daemon containerized as an adapter can read the configuration files of the DT and expose them via a standard interface. Whenever users or external services apply changes to the configuration, the daemon updates the configuration files within the DT filesystem and signals the DT to reload it.

4.3. Patterns for multi-node services

In this section, we discuss requirements and related design patterns, involving not only containers running within the same scheduling unit (e.g., a pod) but also on different nodes. In particular, we discuss requirements inherently involving multiple nodes, such as persistency, replication, and composition.

The persistency requirement (R4) relates to the availability of software components. Indeed, DTs have to be restarted if their container fails or hangs. If the node running the DT fails, the container has to be migrated and restarted on another node. The implementation of this feature is based on a properly implemented management interface exposing the internal state of the DT. For example, the interface has to provide HTTP endpoints specifying if the container is either ready for execution or actually serving requests. By querying this interface, the container orchestrator can take autonomous decision on whether the DT have to be either restarted or migrated to another node. Another approach is based on the *singleton pattern*. The singleton pattern, despite the different flavors it assumes in different contexts, generally implies that only one instance (of an object, a process, a container, etc.) should exist at any given time for the sake of maintaining integrity and consistency. In the context of containerized services, this pattern implies the use of a load balancer managing only one replica of a service. Since only one instance is running, that instance owns the access right to all the resources (i.e., in this case the PO) without the need for electing a master replica. This simplifies implementation and deployment, but introduces disadvantages in terms of reliability since, in case of issues, such as software updates or migrations, a little downtime is required for reverting to a functioning state. Frequently, however, its simplicity outweighs the reliability trade-off.

The replication requirement (R11) can be similarly implemented by making use of the *load balancer pattern* prescribing the use of a load balancer for splitting requests among a pool of replicas. The pool can be monitored via the management interfaces of the replicas and, depending on environmental conditions, can be enlarged, shrunk, or migrated to different network locations. It is worth noting that stateless replicas are advisable since requests can be routed to any replica regardless of their content or their state. In stateless replication, in fact, each replica is aware of the entire state which, in our context, comprises both the PO and the set of containers storing its future and

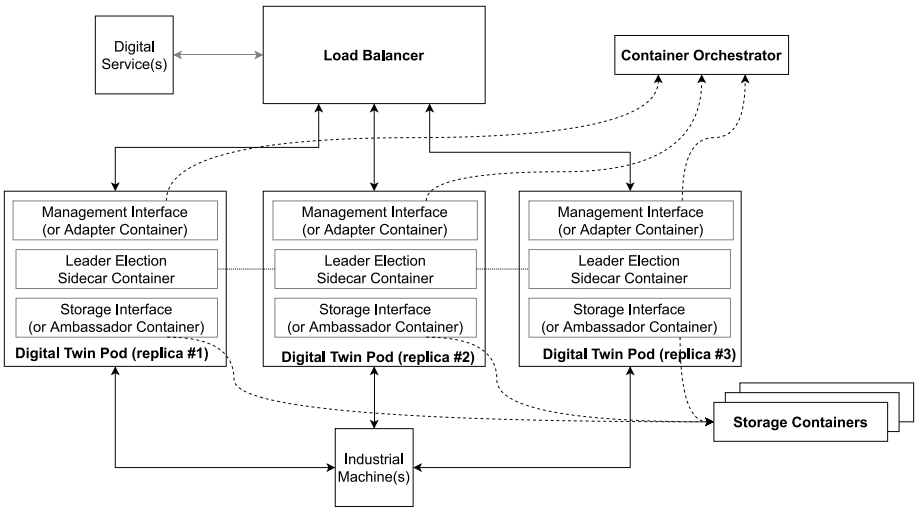


Fig. 6. Three DT replicas handling requests related to same PO. A stateless load balancer splits requests among them. The number of replicas can be either increased or reduced depending on data observed via their management interfaces. Leader election, when needed, can be implemented with a dedicated daemon implementing consensus algorithms packaged in a sidecar container.

past representations. Despite clear benefits in terms of reliability, this approach has some potential issues. For example, the concurrent access of all replicas might overload the PO thus degrading entanglement levels. To prevent this drawback, replicas can adopt a master–slave strategy. The master DT is the single owner of the PO, while slave DTs lose the right of direct access and interact with the PO only via the master DT. In other words, the master enacts a *proxy pattern* between slaves and the PO, thus reducing its load. However, the master–slave approach requires to implement a master election algorithm usually based on distributed consensus algorithms like Paxos or RAFT (Howard and Mortier, 2020). Luckily, there are a number of distributed key–value stores embedding such consensus algorithms without the need of complex implementations within the DT itself. These systems, which can be packaged in a sidecar container as depicted in Fig. 6, provide a replicated and reliable data store comprising the primitives necessary to build election abstractions out-of-the-box.

The composability requirement (R10) prescribes that DTs must receive updates and eventually send commands to a group of peers instead of a single PO. A bare implementation of this feature might require only slight changes to the physical and digital interfaces of the DT for supporting groups of devices instead of a single one. Indeed, each command directed to a PO could be sent to a group of POs or other DTs and each update directed to a digital service can be sent to a group of digital services or other DTs. However, the mere fact of receiving updates or sending commands to a group of peers do not make a composed DT but more a proxy between digital services and the physical environment. What makes composability meaningful is providing applications with composed APIs representing, in a synthetic way, a complex underlying reality. As an example, a composed DT representing a smart-building should provide APIs for querying the average temperature or the presence of fire in the entire building, instead of the bare access to a list of sensors. This problem is often tackled in software engineering with the *API gateway pattern*. This pattern has been in fact proposed to aggregate multiple requests, often directed to different micro-services, into a single one. That is, a digital service attached to a composed DT sends a single request which is decomposed in simpler requests and dispatched to the involved DTs. The received replies are then aggregated and presented to the digital service as one single response. In addition to providing a unified synthetic representation of a complex system, this pattern is also useful in reducing chattiness among involved components.

5. Prototype insights and performance evaluation

This section details the proof-of-concept prototype we implemented and evaluated through an illustrative scenario closely resembling a modern industrial environment. The objective is not only to demonstrate the feasibility of the proposed adaptive, autonomous, and context-aware DT solution, but also to stress how the adoption of the identified design patterns within a realistic environment keeps complexity and costs manageable.

The envisioned experimental scenario puts into action the design patterns presented in Section 4 to enable the requirements laid out in Section 3 through the identification of three phases depicted in Fig. 7 and characterized by: (i) the initial deployment (left); (ii) the context variation (center); and (iii) the deployment adaptation (right). Each of the reference experimental phases are described and detailed in the following paragraphs.

Initial deployment. The initial deployment phase describes a typical industrial setting in a steady state. As illustrated in Fig. 7, we distinguish three logical layers: physical, DT, and application. The physical layer is on the shop floor and comprises three IIoT devices associated to a target deployed industrial machine that publish their status information on a MQTT (2014) message broker, i.e., the so-called IIoT Devices Broker. Such a broker may be defined as a physical broker to refer to its responsibility to exchange data with physical devices. In contrast, the DT layer spreads from the control room to the shop floor. Here, three elementary DTs consume the information published on the physical broker to clone the IIoT devices into software counterparts. Each physical device handles three sub-resources (energy consumption, battery level, and temperature) and publish on independent topics with a configurable message rate (ranging from 10 ms to 100 ms) and an average payload size for each sensor information of 100 Bytes.

The evaluated DTs have been implemented through the creation of a Java DT engine (the core container), which follows the presented design patterns to support built-in modularity and a microkernel oriented structure. The resulting solution is based on a shared multi-thread engine able to effectively implement the DT behavior and to define its shadowing procedures, data processing, and the interaction with external entities through dedicated digital, physical, and management interfaces. Each DT is responsible to digitalize a target MQTT device managing incoming packets to: (i) process and adapt received payloads to the standard Sensor Measurement Lists (SenML) (Jennings et al., 2018) data format; (ii) evaluate and maintain the internal status; and

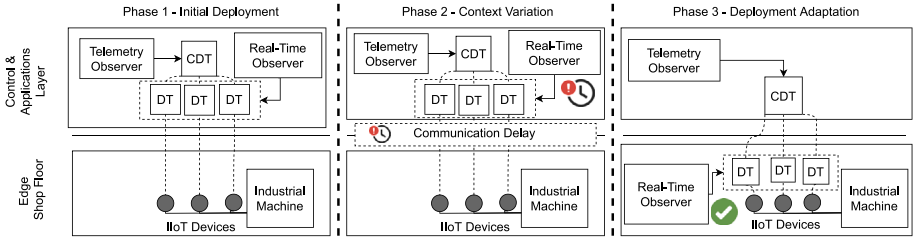


Fig. 7. Phases of the illustrative scenario: initial deployment (left), context variation (center), and deployment adaptation (right).

(iii) handle possible incoming commands and re-configuration requests sent by applications. Such elementary DTs publish their status variation (always using SenML) to a dedicated MQTT message broker, i.e., the so called DTs Broker. Such a broker may be defined as a digital broker to identify its responsibility to handle only packets from DT and applications.

According to the design patterns presented in Section 4, each DT has been structured as a pod where the core engine container is put side by side with a *sidecar* to support communication proxying functionalities and an *ambassador* to handle a uniform and fine-grained metric collection (we recall patterns and ideas presented in Figs. 4 and 5).

Following the same patterns and re-using the DT core engine mentioned above, we defined the behavior of a Composed Digital Twin (CDT). Such a CDT is responsible for periodically aggregating information and statuses from other active DTs and exposing the new computed representation to applications and services interested to have an aggregate representation of the target industrial machine. The CDT directly observes the variations of connected DTs, reading data from the digital broker and publishing its new status on the same broker but on a different topic.

The application layer is about industrial digital services, i.e., those applications built on top (and by means) of the abstractions provided by the DT layer. A telemetry observer and a real-time telemetry observer are the industrial applications part of this illustrative scenario. It is worth noting that such observers differ in the entanglement they demand. Specifically, the real-time telemetry observer demands that observed DTs are strongly entangled with their PO counterparts in order to receive fresh data. This means the information dispatched among IIoT devices and DTs must flow upward and downward as close to real-time as possible.

Context variation. The context variation phase is the result of a drop in network performance, slowing down information circulation between DTs executed in the control room and IIoT devices active at the shop floor. As a result, DTs can no longer guarantee a strong entanglement with their PO counterparts. This generates a misalignment between the physical entity and its digital representation.

The network degradation has a direct impact on different components and multiple metrics. Specifically, it will directly affect the message rate received on each DT and, consequently, the same metric detected by both telemetry and real-time observers. Similarly, the CDT will slow-down the reconstruction of the aggregated representation, resulting misaligned with the respect to the real industrial machine. Note that the acceptable misalignment between DT and PO states is strongly associated to the nature of applications and services, and different tolerance levels may coexist in the same deployment. In our experimental setup, only the real-time observer is affected by a misalignment between POs and DTs, while the telemetry observer is used just for reporting purposes.

Thanks to the possibility of effectively monitoring every single aspect of the involved entities (e.g., through ambassadors on each DT adapting and collecting metrics), we may have multiple decision points able to detect the context variation and react to it by adapting the deployment to restore the target working conditions. In the conducted experiments, this responsibility was delegated to the control room, since it has global awareness on the deployment and thus can determine how and when it should take management actions.

Deployment adaptation. The context variation phase triggers the deployment adaptation phase, which ends with a steady state. The objective of the deployment adaptation phase is to properly react to meet target conditions. In the context of the illustrative scenario, the objective is to dynamically re-configure DTs and applications to meet the demanded level of entanglement through a migration of target components directly on ENs in the shop floor.

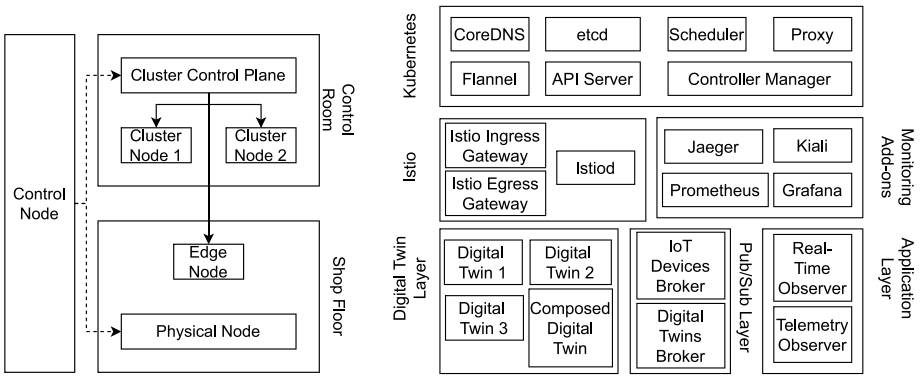
This adaptation requires to: (i) deploy a new MQTT digital broker at the edge and configure it to work in bridged mode with the other one already running in the control room to automatically synchronize target topics; (ii) migrate DTs and the real-time telemetry observer on the edge to be physically close to the IIoT devices; and (iii) re-configure DTs and the observer to work in the new environment with the correct brokers.

5.1. Testbed setup

The testbed consisted of six Virtual Machines (VMs), each running Ubuntu 20.04 LTS and provided with two vCPU and four GB of RAM. Fig. 8(a) shows the physical setup (i.e., roles of the VMs and their physical location), while Fig. 8(b) the cluster pods (i.e., containerized applications deployed in the cluster).

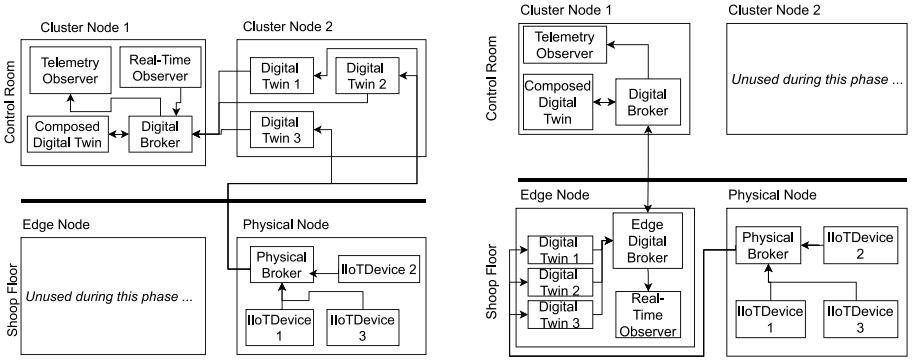
We used Kubernetes (v1.21.11),[1] the de facto industry standard container-orchestration system, to build a cluster of four VMs. The cluster spreads from the control room (a single control plane and two cluster nodes) to the shop floor (one EN). On the one hand, ENs are not as resource-rich as cluster nodes, which typically rely on commodity server technology. On the other hand, ENs are physically closer to data sources. Thus, the rationale for making ENs part of the cluster is not only to add mere resources to the cluster, but more importantly to orchestrate services physically close to data sources (e.g., to improve entanglement). We chose Docker (v20.10.14)[2] as the container runtime and Flannel (v0.17.0)[3] as the network plugin. On top of Kubernetes, we deployed Istio (v1.10.0)[4] as the service mesh. We configured Istio to automatically inject Envoy proxies[5] as sidecars to our services running within the cluster. By doing so, such Envoy proxies intercept all inbound and outbound traffic of the services. This allows Istio to enforce policies and collect telemetry for other monitoring systems. In this regard, we also integrated Istio with telemetry add-ons, i.e., Prometheus (a monitoring system and time series database),[6]

[1] https://github.com/kubernetes/kubernetes.
[2] https://github.com/docker.
[3] https://github.com/flannel-io/flannel.
[4] https://github.com/istio/istio.
[5] https://github.com/envoyproxy/envoy.
[6] https://github.com/prometheus/prometheus.



(a) Physical setup. (b) Cluster pods.

Fig. 8. Comparison between the physical testbed and the structure of involved cluster pods.



(a) Initial deployment. (b) Adaptive deployment.

Fig. 9. Traffic flows at the digital and application level.

Grafana (a monitoring tool for visualizing time series data in dashboards),[7] Kiali (a management console for the service mesh),[8] and Jaeger (an end-to-end distributed tracing system).[9]

We deployed the containerized DTs and the digital broker in the Kubernetes cluster. Note that Kubernetes allows to specify both resource requests and limits. Specifically, a resource request states the minimum amount of a given resource a container needs to run, while a resource limit indicates the maximum amount of that resource a container can take while running. We set the memory request and limit for DTs to 64 MB and 128 MB, respectively.

A distinct VM hosted PO counterparts of DTs. In this case, PO counterparts were IIoT devices and a broker used by such devices to publish MQTT messages. The last VM acted as control node, in charge of configuring the testbed, running the experiments consistently and reproducibly, and gathering the performance results for further analysis. In this regard, we used Ansible (a configuration management tool),[10] Shell scripts, and Python programs.

5.2. Performance results

The performance results we collected detail (i) the overhead introduced by the used technology stack to enable adaptive, autonomous, and context-aware DTs and (ii) how the whole system behaves while adapting to triggering events. The discussion about the performance results revolves around the steady and transitory states the cluster went through while performing the phases depicted by the illustrative scenario. In particular, we measured both resource (i.e., CPU and memory) and network consumption. We extracted the performance results from Prometheus with a per-pod granularity.

Steady state. In the illustrative scenario, a steady state (i.e., a stable cluster configuration over a period) occurs twice: throughout the initial deployment phase (up to the context variation phase) and in the adaptive deployment phase once the new cluster configuration occurs. Fig. 9 details the cluster pods for the digital and application levels during the steady states of the initial (left side) and adaptive (right side) deployment.

Table 1 shows the average resource consumption in a steady state of the following macro-components: Kubernetes, Istio, monitoring addons, and DTs. Fig. 8(b) breaks down such macro-components on a per-pod basis. Kubernetes consumed the majority of resources overall. Specifically, it took 265.75 milliCPU, 2.02 GB (memory), 450.23 KB (traffic in), and 516.06 KB (traffic out). Although the highest in this comparison, the resources allocated for Kubernetes are still minimal. This makes Kubernetes suitable for typical devices within industrial environments. In addition, note that there are also Kubernetes distributions designed explicitly for resource-constrained scenarios. Such distributions may represent a reasonable option for those environments that cannot afford the overhead introduced by vanilla Kubernetes or need to support specific use cases (e.g., semi-autonomous ENs).

An important outcome of the above performance results is that our DT implementation is extremely frugal. It is worth noting that the item "Digital Twins" in Table 1 regards the DTs themselves and also sidecars and ambassadors (deployed as containers within the same pod). In particular, the DTs altogether consumed 43.31 milliCPU, 0.48 GB (memory), 4.74 KB (traffic in), and 15.26 KB (traffic out). The overhead

[7] https://github.com/grafana/grafana.
[8] https://github.com/kiali/kiali.
[9] https://github.com/jaegertracing/jaeger.
[10] https://github.com/ansible/ansible.

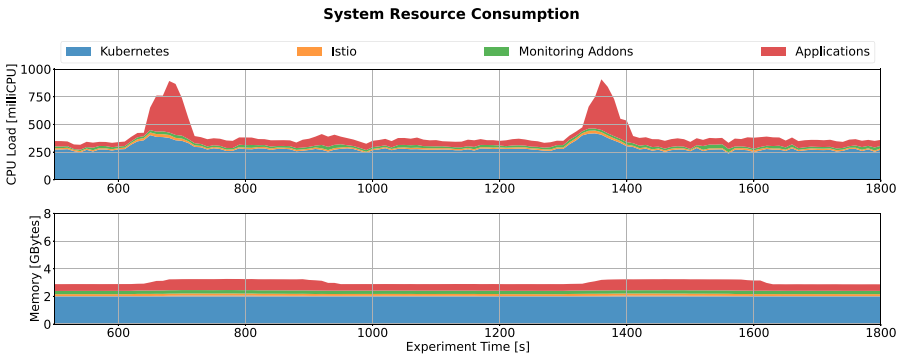


Fig. 10. Introduced overhead in terms of CPU and Memory consumption with respect to a basic Kubernetes deployment during the phases of migration and rollback.

Table 1
Steady state averaged collected performance metrics.

Metric	Unit	Component	AVG	STD
CPU	milliCPU	Kubernetes	265.75	8.05
CPU	milliCPU	Istio	12.00	0.78
CPU	milliCPU	Monitoring addons	4.51	0.78
CPU	milliCPU	Digital twins	43.31	4.38
Memory	GB	Kubernetes	2.02	0.01
Memory	GB	Istio	0.13	0.01
Memory	GB	Monitoring addons	0.23	0.001
Memory	GB	Digital twins	0.48	0.003
Traffic in	KB	Kubernetes	450.23	18.01
Traffic in	KB	Istio	2.62	0.44
Traffic in	KB	Monitoring addons	70.67	7.40
Traffic in	KB	Digital twins	4.74	1.56
Traffic out	KB	Kubernetes	516.06	19.31
Traffic out	KB	Istio	27.47	4.52
Traffic out	KB	Monitoring addons	7.02	0.82
Traffic out	KB	Digital twins	15.26	0.75

Table 2
Average execution time for involved migration steps.

Id	Action	Entity	Location	Exec. [ms]
1	CREATE	Edge digital broker	On edge	7.34
2	CREATE	Digital twin 1	On edge	8.19
3	CREATE	Digital twin 2	On edge	8.13
4	CREATE	Digital twin 3	On edge	8.28
5	CREATE	Real-time observer	On edge	7.36
6	DELETE	Digital twin 1	From control room	3.34
7	DELETE	Digital twin 2	From control room	3.23
8	DELETE	Digital twin 3	From control room	3.20
9	DELETE	Real-time observer	From control room	2.40

Table 3
Average execution time for involved rollback steps.

Id	Action	Entity	Location	Exec. [ms]
1	CREATE	Digital twin 1	On control room	8.19
2	CREATE	Digital twin 2	On control room	8.18
3	CREATE	Digital twin 3	On control room	8.32
4	CREATE	Real-time observer	On control room	7.31
5	DELETE	Digital twin 1	From edge	3.42
6	DELETE	Digital twin 2	From edge	3.22
7	DELETE	Digital twin 3	From edge	3.28
8	DELETE	Real-time observer	From edge	2.39
9	DELETE	Edge digital broker	From edge	2.46

introduced by sidecars and ambassadors is negligible. This notable result fosters the use of design patterns whose benefits go far beyond their costs.

Transitory state. A transitory state happens while moving from one cluster configuration to another, and its analysis allows us to quantify the overhead of a given transition. Typically, a transition is triggered by a context variation, which forces the system to move towards a new configuration. In the illustrative scenario, the transition is from the initial deployment to the adaptive deployment. Such transition occurs since the cluster as it was configured in the initial deployment phase no longer meets the entanglement demanded by the real-time telemetry observer. The proposed autonomous, adaptive, and context-aware DTs can deal with context variation, forcing the cluster to move to a new configuration, i.e., the adaptive deployment.

For completeness, we experimentally assessed both the transition from the initial deployment (Fig. 9(a)) to the adaptive deployment (Fig. 9(b)) and vice versa to rollback to the initial configuration of the deployment. Table 2 itemizes the steps to move from the initial deployment to the adaptive deployment (Table 3 presents the opposite). It is worth mentioning that the container images were pre-pulled on the cluster nodes. Therefore, a CREATE step did not require downloading the related container image from a repository. Also, we executed the steps sequentially, which means the total time of a given transition is the sum of every single step. Note that some steps may be executed concurrently to speed up the transition.

As a result (without parallel step execution), on average the overall migration time was around 55 s and the rollback procedure required about 50 s. Such time intervals include the reported action steps, the execution time required by the container to start internal modules, connect to brokers and start processing incoming data, and the overhead introduced by Ansible.

Graphs in Figs. 10 and 11 depict the resource consumption during the above-mentioned transitory states. The first peak regards the transition from the initial deployment to the adaptive deployment, whereas the second one is the rollback. The peak went over a steady-state resource consumption of around 500 milliCPU, 360 MB (memory), 1600 KB (traffic in), and 1800 KB (traffic out). Graphs in Fig. 12 report instead the distribution of CPU and memory consumption of the DT pods considering both the migration and the rollback procedures. As expected, reported values confirm the trends illustrated in the previous timeline analysis, i.e., the overall limited resource consumption of DT pods and their small variation during the transitions. The CDT kept the same value distribution since it was not directly involved in the migration procedures while DTs increased CPU and memory load only during the transitory period. In both analysis, it is important to stress that the memory occupied by removed containers was released by the virtualization system only after a specific time period (around 5 min). For this reason, in Fig. 10 the occupied memory does not decrease immediately after the transition peak and in Fig. 12 there are two main density areas associated to the memory (which also takes into account the allocation of removed pods).

Let us finally note that the proper management of transition phases is paramount to ensure the fulfillment of demanded requirements in the case of dynamically changing environmental conditions. In this regard, future digital factories will need to activate orchestration mechanisms, while one-shot deployments will not be suitable anymore. Accordingly,

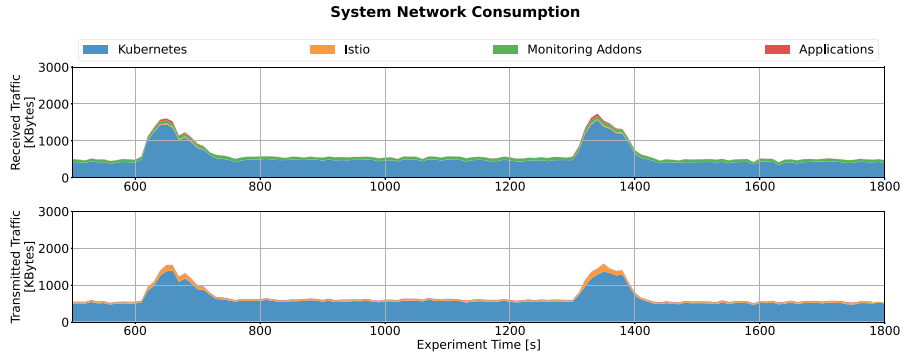
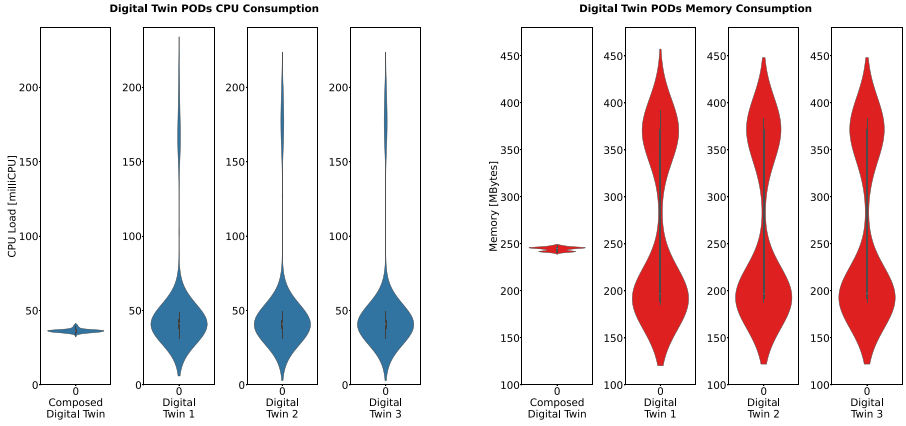


Fig. 11. Received and transmitted KB by Kubernetes, Istio, Monitoring addons and DT applications during the phases of migration and rollback.



(a) CPU Consumption [milliCPU]. (b) Memory Consumption [MB]

Fig. 12. Analysis of DT Pods resource consumption in terms of CPU and Memory during the phases of migration and rollback.

transitory states must be balanced between required resources to absorb the peak, the following estimated steady-state time, and competing application-level requirements.

6. Related work

Patterns are encapsulations of reusable common problems and solutions under specific contexts. The general idea was conceptualized for the first time by Christopher Alexander in 1966 (Alexander, 1966) and it took 20 years before Cunningham et al. started experimenting with patterns applied to programming (Cunningham and Beck, 1986). Popularity of software design patterns grew within the industry since the landmark contribution of Gamma et al. in 1993 (Gamma et al., 1993) while the proper formalization of the concept finally arrived in early 2000s (Baroni et al., 2003). A number of studies have been conducted for categorizing and evaluating patterns both empirically and analytically (Ali and Elish, 2013; Ampatzoglou et al., 2013; Riaz et al., 2015; Mayvan et al., 2017). The impact of design patterns on software quality attributes has been also extensively evaluated along many directions. Wedyan and Abufakher (2020) recently surveyed 50 studies published between 2000 and 2018 and showed how the correct use of well-documented software patterns impact on software qualities.

IIoT applications have recently gained remarkable traction, thus requiring system designers to familiarize with software patterns specific to the IoT and industrial domains. Ray (2016) conducted a survey on IoT cloud platforms and their impacts on industrial solutions. Aly et al. (2018) studied the state-of-practices of industrial IoT technologies and highlighted how integration challenges have significantly shifted the landscape of Internet-based collaborative services and applications. Washizaki et al. (2020) published a comprehensive survey on IoT architecture and design patterns showing, among other things, the relevance of documenting and properly classifying software patterns to promote their widespread adoption.

On top of IIoT solutions, flexible and self-adapting software systems are being developed. However, due to their high complexity, adaptive components might be difficult to design, test, and deploy. Ramirez and Cheng (2010) proposed a catalog of adaptation-oriented design patterns supporting the engineering of systems capable of adaptive behaviors and facilitating the segregation of functional and adaptive logic.

Recently the role of DTs has been re-analyzed and re-shaped both by the scientific and the industrial communities. The primary aim is to clearly identify their definitions and responsibilities as well as to identify new challenges and opportunities among different application domains, in particular in relation to IoT and IIoT (Tao et al., 2019; Barricelli et al., 2019; Bellavista et al., 2022). A shared reference architecture (Malakuti and Grüner, 2018; Souza et al., 2019) has been proposed within the context of the Industrial Internet of Things Consortium, taking into account DT relationships, composition, and main services (e.g., prediction, maintenance, safety). Such architecture also covers different production stages and use cases, in particular related to manufacturing (Kritzinger et al., 2018) and product design (Wagner et al., 2019). DTs are increasingly being considered a part of cyber–physical system architectures, realizing twin models of assets and machines (Josifovska et al., 2019), the computational modules of the physical components of cyber–physical systems (Alam and El Saddik, 2017), or within the RAMI4.0 ecosystem as an important pattern for the manufacturing process and the administration shell (Anderl et al., 2018; Tantik and Anderl, 2017). Minerva and Crespi (2021) proposed a definition and characterization of DTs in relation to software architectures and their platform implementations, together with their

applicability in different industries, while Tekinerdogan and Verdouw (2020) proposed a catalog of software patterns for designing complex systems specifically based on the DT technology. While these papers represent solid groundwork, the investigation and definition of DT-oriented software modeling and design patterns matching DT core properties and responsibilities are still an open research opportunity characterized by a variety of methodological and technical challenges that have not yet been fully analyzed and addressed, as also illustrated in Hribernik et al. (2021) and Koulamas and Kalogeras (2018).

By focusing on the adoption of micro-services for the development of DTs, Siqueira and Davis (2022) provided a thorough analysis of the state-of-the-art literature stressing how such approaches allow to develop the software infrastructure supporting Industry 4.0. In particular, the survey outlines that even if containerization can be regarded as a mature technology for cloud environments, on embedded devices there is still limited support. In addition, there is the need of fully supporting interoperability with orchestrators (and in particular Kubernetes) to ensure DT seamless coordination. Liu et al. (2022a) analyzed the smart meter industrial use case by adopting container-based DTs on the server side to better manage novel applications. In particular, the adoption of containerized DTs of smart meters hosted on server-side distribution grids allows to provide a wider set of applications while minimizing the upgrading procedures on user-side smart meters. Azarmipour et al. (2020) pursued the key objective of developing a dynamic management solution by developing the MES and programmable logic controllers as composition of multiple software modules based on a micro-services approach. In particular, the proposed architecture aims at facilitating the coordination among the MES and control/optimization/management environments, by allowing their communication among service interfaces. To support the deployment of DTs within industrial environments, Damjanovic-Behrendt and Behrendt (2019) recognized the importance of adopting well-known solutions integrated as containerized micro-services. In particular, their proposal focuses on open source projects, by identifying three technology building blocks to develop the DT core components (i.e., Data Manager, Models Manager, and Services Manager). The implementation of the DT core components is based on Apache Kafka, RabbitMQ, Elasticsearch, Grafana, and Hadhoop (among the others). Finally, Wang et al. (2022) presented a notable vehicular use-case demonstrating the suitability of DTs and micro-services in a challenging mobile environment. The proposed architecture is composed of three building blocks in the physical space, i.e., Human, Vehicle, and Traffic, and their related DTs in the digital space implemented as micro-services, i.e., the Human Digital Twin with user management and driver type classification, the Vehicle Digital Twin with cloud-based advanced driver-assistance systems, and the Traffic Digital Twin with traffic flow monitoring and variable speed limit.

In this challenging ecosystem, DTs are conceived as flexible and adaptive software entities that can be exploited to build context-aware, autonomous, and adaptive applications across multiple domains (Hribernik et al., 2021). The ability to design and build DTs that are aware of their own context and capable of autonomous decision-making/adaptation represents a strategic pillar for the next generation of cyber–physical systems, as presented for example in Park et al. (2020) to support personalized production systems and in Cronrath et al. (2019) to enable intelligent manufacturing. In this context, the MAPE-K feedback loop (Arcaini et al., 2015; Kephart and Chess, 2003) represents a well-adopted reference model for managing and controlling autonomous and self-adaptive systems, and its adoption has enabled relevant improvements in autonomous systems over the past decades. Recently, its adoption in the DT ecosystem (Snijders et al., 2020; Feng et al., 2022; Flammini, 2021) has opened up the possibility of designing smart, resilient, and trustworthy DT-driven cyber–physical applications through enhanced awareness and adaptiveness. Even if we support and foster this approach by considering it suitable for possible integration with our research results presented here, at the current stage our proposal primarily focuses on DT-oriented software modeling and design patterns with the aim of enabling a simplified and structured definition of context-aware, autonomous, and adaptive DTs since their foundations.

Overall, the above analysis demonstrates the current interest in developing IIoT solutions and DTs based on clear design principles, with the primary goal of making their development easier and increase their adaptability and flexibility. However, the state-of-the-art literature either only proposes high-level guidelines, e.g., stressing the importance of micro-services architectures but without detailing how to design them, or focuses on specific scenarios, e.g., by proposing vertical solutions for specific markets. Instead, we presented an in-depth analysis of DT requirements to support adaptability, autonomy, and context-awareness in digital factories. Then, we identified best design patterns allowing to satisfy such requirements, thus providing a more general solution that can be adopted in a wide variety of Industry 4.0 environments.

7. Conclusions

In this paper, we discussed how system-wide properties of near coming digital factories such as adaptivity, autonomy, and context-awareness have to be supported by software components and architectures showing, to some extent, the same properties. To this purpose, we translated the general meaning of adaptivity, autonomy, and context-awareness into well-defined, actionable requirements for DTs. We also discussed the relevance of adopting well-understood norms and standards for building deployments counting thousands of DTs and POs and proposed a set of software and architectural patterns that can be used for implementing the identified requirements in a scalable and reliable way.

In light of the benefits that the proposed solution brings to the development and management of industrial DTs, we implemented a working prototype to quantify the costs in terms of networking and computational resources due to required additional software layers. Achieved performance results not only demonstrated the effectiveness of the proposed approach, but also how it can be implemented on top of widely adopted solutions, such as containerization and Kubernetes.

Despite the interesting insights achieved in this work and mainly related to the efficient and standardized modeling/design phases of DTs, the road to the complete definition and efficient support of their life-cycle in industrial contexts is still long. First of all, to encourage the large scale adoption of our approach, we plan to deeply investigate the associated security concerns. Secondly, we intend to further demonstrate the findings originally presented in this paper by assessing and validating them in additional realistic case studies, resembling different and more complex aspects of modern industrial environments. Finally, we plan to further investigating the entanglement property and how it can be actually expressed in numerical terms (and efficiently exploited in practical implementation terms) for re-configuring either DTs or the orchestration environment.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

Al-Sehrawy, R., Kumar, B., 2020. Digital twins in architecture, engineering, construction and operations. a brief review and analysis. In: International Conference on Computing in Civil and Building Engineering. Springer.

Alam, K.M., El Saddik, A., 2017. C2PS: A digital twin architecture reference model for the cloud-based cyber-physical systems. IEEE Access 5, 2050–2062. http://dx.doi.org/10.1109/ACCESS.2017.2657006.

Alexander, C., 1966. The pattern of streets. J. Am. Inst. Plan. 32 (5), 273–278.

Alexopoulos, K., Makris, S., Xanthakis, V., Sipsas, K., Chryssolouris, G., 2016. A concept for context-aware computing in manufacturing: the white goods case. Int. J. Comput. Integr. Manuf. 29 (8), 839–849.

Ali, M., Elish, M.O., 2013. A comparative literature survey of design patterns impact on software quality. In: 2013 International Conference on Information Science and Applications. ICISA, IEEE, pp. 1–7.

Aly, M., Khomh, F., Guéhéneuc, Y.-G., Washizaki, H., Yacout, S., 2018. Is fragmentation a threat to the success of the internet of things? IEEE Internet Things J. 6 (1), 472–487.

Ampatzoglou, A., Charalampidou, S., Stamelos, I., 2013. Research state of the art on GoF design patterns: A mapping study. J. Syst. Softw. 86 (7), 1945–1964.

Anderl, R., Haag, S., Schützer, K., Zancul, E., 2018. Digital twin technology – an approach for industrie 4.0 vertical and horizontal lifecycle integration. It - Inf. Technol. 60 (3), 125–132. http://dx.doi.org/10.1515/itit-2017-0038.

Arcaini, P., Riccobene, E., Scandurra, P., 2015. Modeling and analyzing MAPE-K feedback loops for self-adaptation. In: 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. pp. 13–23. http://dx.doi.org/10.1109/SEAMS.2015.10.

Azarmipour, M., Elfaham, H., Gries, C., Kleinert, T., Epple, U., A service-based architecture for the interaction of control and MES systems in industry 4.0 environment, 2020-July, 217–222, http://dx.doi.org/10.1109/INDIN45582.2020.9442083.

Baroni, A.L., Guéhéneuc, Y.-G., Albin-Amiot, H., 2003. Design Patterns Formalization, Rapport De Recherche. Département D'Informatique, École Des Mines de Nantes, (03/03).

Barricelli, B.R., Casiraghi, E., Fogli, D., 2019. A survey on digital twin: Definitions, characteristics, applications, and design implications. IEEE Access 7, 167653–167671. http://dx.doi.org/10.1109/ACCESS.2019.2953499.

Bellavista, P., Giannelli, C., Mamei, M., Mendula, M., Picone, M., 2021. Application-driven network-aware digital twin management in industrial edge environments. IEEE Trans. Ind. Inform. 17, 7791–7801. http://dx.doi.org/10.1109/TII.2021.3067447.

Bellavista, P., Giannelli, C., Mamei, M., Mendula, M., Picone, M., 2022. Digital twin oriented architecture for secure and QoS aware intelligent communications in industrial environments. Pervasive Mob. Comput. 85, 101646. http://dx.doi.org/10.1016/j.pmcj.2022.101646, URL https://www.sciencedirect.com/science/article/pii/S1574119222000736.

Bolender, T., Burvenich, G., Dalibor, M., Rumpe, B., Wortmann, A., 2021. Self-adaptive manufacturing with digital twins. In: Proceedings - 2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2021. pp. 156–166.

Corradi, A., Foschini, L., Giannelli, C., Lazzarini, R., Stefanelli, C., Tortonesi, M., Virgilli, G., 2019. Smart appliances and RAMI 4.0: Management and servitization of ice cream machines. IEEE Trans. Ind. Inform. 15, http://dx.doi.org/10.1109/TII.2018.2867643.

Cronrath, C., Aderiani, A.R., Lennartson, B., 2019. Enhancing digital twins through reinforcement learning. In: 2019 IEEE 15th International Conference on Automation Science and Engineering. CASE, pp. 293–298. http://dx.doi.org/10.1109/COASE.2019.8842888.

Cunningham, W., Beck, K., 1986. A diagram for object-oriented programs. ACM Sigplan Not. 21 (11), 361–367.

Damjanovic-Behrendt, V., Behrendt, W., 2019. An open source approach to the design and implementation of digital twins for smart manufacturing. Int. J. Comput. Integr. Manuf. 32, 366–384. http://dx.doi.org/10.1080/0951192X.2019.1599436.

Ding, K., Chan, F.T.S., Zhang, X., Zhou, G., Zhang, F., 2019. Defining a digital twin-based cyber-physical production system for autonomous manufacturing in smart shop floors. Int. J. Prod. Res. 57 (20), 6315–6334.

Dinh-Tuan, H., Beierle, F., Garzon, S.R., 2019. MAIA: a microservices-based architecture for industrial data analytics. In: 2019 IEEE International Conference on Industrial Cyber Physical Systems. ICPS, IEEE, pp. 23–30.

Dobaj, J., Iber, J., Krisper, M., Kreiner, C., 2018. A microservice architecture for the industrial internet-of-things. In: Proceedings of the 23rd European Conference on Pattern Languages of Programs. pp. 1–15.

Feng, H., Gomes, C., Gil, S., Mikkelsen, P.H., Tola, D., Larsen, P., Sandberg, M., 2022. Integration of the mape-k loop in digital twins. In: 2022 Annual Modeling and Simulation Conference. ANNSIM, IEEE Computer Society, Los Alamitos, CA, USA, pp. 102–113. http://dx.doi.org/10.23919/ANNSIM55834.2022.9859489.

Flammini, F., 2021. Digital twins as run-time predictive models for the resilience of cyber-physical systems: A conceptual framework. Phil. Trans. R. Soc. A 379, 20200369. http://dx.doi.org/10.1098/rsta.2020.0369.

Fogli, M., Giannelli, C., Stefanelli, C., 2022a. Edge-powered in-network processing for content-based message management in software-defined industrial networks. In: ICC 2022 - IEEE International Conference on Communications. pp. 1438–1443. http://dx.doi.org/10.1109/ICC45855.2022.9838863.

Fogli, M., Giannelli, C., Stefanelli, C., 2022b. Joint orchestration of content-based message management and traffic flow steering in industrial backbones. In: 2022 IEEE 23rd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM). pp. 325–330. http://dx.doi.org/10.1109/WoWMoM54355.2022.00067.

Fuller, A., Fan, Z., Day, C., Barlow, C., 2020. Digital twin: Enabling technologies, challenges and open research. IEEE Access 8, 108952–108971.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1993. Design patterns: Abstraction and reuse of object-oriented design. In: European Conference on Object-Oriented Programming. Springer, pp. 406–431.

Ghosh, A., Mukherjee, A., Misra, S., 2021. SEGA: Secured edge gateway microservices architecture for iIoT-based machine monitoring. IEEE Trans. Ind. Inform. 18 (3), 1949–1956.

Hinduja, H., Kekkar, S., Chourasia, S., Chakrapani, H.B., 2020. Industry 4.0: digital twin and its industrial applications. Int. J. Sci. Eng. Technol. Open Access. J. 8 (4).

Howard, H., Mortier, R., 2020. Paxos vs raft: Have we reached consensus on distributed consensus? In: Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data. pp. 1–9.

Hribernik, K., Cabri, G., Mandreoli, F., Mentzas, G., 2021. Autonomous, context-aware, adaptive digital twins—State of the art and roadmap. Comput. Ind. 133, 103508.

IEC 62443: industrial network and system security, tech. rep., international electrotechnical commission, 2013.

Jennings, C., Shelby, Z., Arkko, J., Keränen, A., Bormann, C., 2018. Sensor measurement lists (senml). http://dx.doi.org/10.17487/RFC8428, RFC 8428. URL https://rfc-editor.org/rfc/rfc8428.txt.

Jiang, Y., Yin, S., Li, K., Luo, H., Kaynak, O., 2021. Industrial applications of digital twins. Phil. Trans. R. Soc. A 379 (2207), 20200360.

Jones, D., Snider, C., Nassehi, A., Yon, J., Hicks, B., 2020. Characterising the digital twin: A systematic literature review. CIRP J. Manuf. Sci. Technol. 29, 36–52.

Josifovska, K., Yigitbas, E., Engels, G., 2019. Reference framework for digital twins within cyber-physical systems. In: 2019 IEEE/ACM 5th International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS). pp. 25–31. http://dx.doi.org/10.1109/SEsCPS.2019.00012.

Kephart, J., Chess, D., 2003. The vision of autonomic computing. Computer 36 (1), 41–50. http://dx.doi.org/10.1109/MC.2003.1160055.

Koulamas, C., Kalogeras, A., 2018. Cyber-physical systems and digital twins in the industrial internet of things [cyber-physical systems]. Computer 51 (11), 95–98. http://dx.doi.org/10.1109/MC.2018.2876181.

Kritzinger, W., Karner, M., Traar, G., Henjes, J., Sihn, W., 2018. Digital twin in manufacturing: A categorical literature review and classification. IFAC-PapersOnLine 51 (11), 1016–1022. http://dx.doi.org/10.1016/j.ifacol.2018.08.474, 16th IFAC Symp. on Information Control Problems in Manufacturing 2018.

Leng, J., Liu, Q., Ye, S., Jing, J., Wang, Y., Zhang, C., Zhang, D., Chen, X., 2020. Digital twin-driven rapid reconfiguration of the automated manufacturing system via an open architecture model. Robot. Comput.-Integr. Manuf. 63.

Leng, J., Wang, D., Shen, W., Li, X., Liu, Q., Chen, X., 2021a. Digital twins-based smart manufacturing system design in industry 4.0: A review. J. Manuf. Syst. 60, 119–137.

Leng, J., Zhou, M., Xiao, Y., Zhang, H., Liu, Q., Shen, W., Su, Q., Li, L., 2021b. Digital twins-based remote semi-physical commissioning of flow-type smart manufacturing systems. J. Clean. Prod. 306.

Liu, L., Ding, Y., Li, X., Wu, H., Xing, L., 2022a. A container-driven service architecture to minimize the upgrading requirements of user-side smart meters in distribution grids. IEEE Trans. Ind. Inform. 18, 719–728. http://dx.doi.org/10.1109/TII.2021.3088135.

Liu, L., Guo, K., Gao, Z., Li, J., Sun, J., 2022b. Digital twin-driven adaptive scheduling for flexible job shops. Sustainability (Switzerland) 14 (9).

Maggi, F., Pogliani, M., 2017. Attacks on Smart Manufacturing Systems. Tech. rep., Trend Micro Research, Shibuya, Japan.

Malakuti, S., Grüner, S., 2018. Architectural aspects of digital twins in iIoT systems. In: Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings. ECSA '18, Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/3241403.3241417.

Mayvan, B.B., Rasoolzadegan, A., Yazdi, Z.G., 2017. The state of the art on design patterns: A systematic mapping of the literature. J. Syst. Softw. 125, 93–118.

Minerva, R., Crespi, N., 2021. Digital twins: Properties, software frameworks, and application scenarios. IT Prof. 23 (1), 51–55. http://dx.doi.org/10.1109/MITP.2020.2982896.

Minerva, R., Lee, G.M., Crespi, N., 2020. Digital twin in the IoT context: a survey on technical features, scenarios, and architectural models. Proc. IEEE 108 (10), 1785–1824.

2014. MQTT Version 3.1.1. URL http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html.

Park, K.T., Lee, J., Kim, H.-J., Noh, S.D., 2020. Digital twin-based cyber physical production system architectural framework for personalized production. Int. J. Adv. Manuf. Technol. 106, 1–24. http://dx.doi.org/10.1007/s00170-019-04653-7.

Platenius-Mohr, M., Malakuti, S., Grüner, S., Schmitt, J., Goldschmidt, T., 2020. File- and API-based interoperability of digital twins by model transformation: An IIoT case study using asset administration shell. Future Gener. Comput. Syst. 113, 94–105.

Ramirez, A.J., Cheng, B.H., 2010. Design patterns for developing dynamically adaptive systems. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems. pp. 49–58.

Ray, P.P., 2016. A survey of IoT cloud platforms. Future Comput. Inform. J. 1 (1–2), 35–46.

Riaz, M., Breaux, T., Williams, L., 2015. How have we evaluated software pattern application? A systematic mapping study of research design practices. Inf. Softw. Technol. 65, 14–38.

Shi, W., Dustdar, S., 2016. The promise of edge computing. Computer 49 (5), 78–81. http://dx.doi.org/10.1109/MC.2016.145.

Siqueira, F., Davis, J., 2022. Service computing for industry 4.0: State of the art, challenges, and research opportunities. ACM Comput. Surv. 54, http://dx.doi.org/10.1145/3478680.

Snijders, R., Pileggi, P., Broekhuijsen, J., Verriet, J., Wiering, M., Kok, K., 2020. Machine learning for digital twins to predict responsiveness of cyber-physical energy systems. In: 2020 8th Workshop on Modeling and Simulation of Cyber-Physical Energy Systems. pp. 1–6. http://dx.doi.org/10.1109/MSCPES49613.2020.9133695.

Souza, V., Cruz, R., Silva, W., Lins, S., Lucena, V., 2019. A digital twin architecture based on the industrial internet of things technologies. In: 2019 IEEE Int. Conf. on Consumer Electronics. ICCE, pp. 1–2.

Tantik, E., Anderl, R., 2017. Potentials of the asset administration shell of industrie 4.0 for service-oriented business models. Proc. CIRP 64, 363–368. http://dx.doi.org/10.1016/j.procir.2017.03.009, 9th CIRP IPSS Conference: Circular Perspectives on PSS. URL https://www.sciencedirect.com/science/article/pii/S2212827117301531.

Tao, F., Zhang, H., Liu, A., Nee, A.Y.C., 2019. Digital twin in industry: State-of-the-art. IEEE Trans. Ind. Inform. 15 (4), 2405–2415. http://dx.doi.org/10.1109/TII.2018.2873186.

Tekinerdogan, B., Verdouw, C., 2020. Systems architecture design pattern catalog for developing digital twins. Sensors 20 (18), 5103.

Vuković, M., Mazzei, D., Chessa, S., Fantoni, G., 2021. Digital twins in industrial IoT: a survey of the state of the art and of relevant standards. In: 2021 IEEE International Conference on Communications Workshops (ICC Workshops). IEEE, pp. 1–6.

Wagner, R., Schleich, B., Haefner, B., Kuhnle, A., Wartzack, S., Lanza, G., 2019. Challenges and potentials of digital twins and industry 4.0 in product design and production for high performance products. Proc. CIRP 84, 88–93. http://dx.doi.org/10.1016/j.procir.2019.04.219, 29th CIRP Design Conference 2019, 08-10 May 2019, Póvoa de Varzim, Portgal.

Wang, Z., Gupta, R., Han, K., Wang, H., Ganlath, A., Ammar, N., Tiwari, P., 2022. Mobility digital twin: Concept, architecture, case study, and future challenges. IEEE Internet Things J. http://dx.doi.org/10.1109/JIOT.2022.3156028.

Washizaki, H., Ogata, S., Hazeyama, A., Okubo, T., Fernandez, E.B., Yoshioka, N., 2020. Landscape of architecture and design patterns for iot systems. IEEE Internet Things J. 7 (10), 10091–10101.

Wedyan, F., Abufakher, S., 2020. Impact of design patterns on software quality: a systematic literature review. IET Softw. 14 (1), 1–17.