

ACCEPTED MANUSCRIPT

# State Reproducibility in Stateful Digital Twin Services

G. Rossi, M. Fogli, R. Spalazzese and C. Giannelli

*24th International Conference on Service-Oriented Computing (ICSOC), 2026*

---

## Cite this as

G. Rossi, M. Fogli, R. Spalazzese and C. Giannelli, “State Reproducibility in Stateful Digital Twin Services,” accepted for publication in *Service-Oriented Computing: 24th International Conference, ICSOC 2026, Łódź, Poland*.

## Notice

This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record will be published by Springer Nature. Use of this Accepted Version is subject to the publisher’s Accepted Manuscript terms of use

<https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms>

# State Reproducibility in Stateful Digital Twin Services

Gaia Rossi<sup>1</sup>, Mattia Fogli<sup>1</sup><sup>\*</sup>, Romina Spalazzese<sup>2</sup>, and Carlo Giannelli<sup>1</sup>

<sup>1</sup> University of Ferrara, Italy

{gaia.rossi, mattia.fogli, carlo.giannelli}@unife.it

<sup>2</sup> Malmö University, Sweden

romina.spalazzese@mau.se

**Abstract.** The Digital Twin as a Service (DTaaS) approach delivers Digital Twin (DT) capabilities as modular, scalable, and on-demand services and requires stateful DTs to remain operational in case of migration, fault recovery, and redeployment. However, when a DT service is disrupted, its runtime state cannot be retrieved from external storage alone: it depends on the sequence of observations received from the physical counterpart, the Physical Twin (PT), which may not be able to regenerate past observations on demand. We argue that state reproducibility is an important property that DTaaS deployments must support. We provide (i) a semi-formal definition of reproducibility grounded in rollback-recovery theory. We also (ii) argue that pessimistic logging (log-based recovery) is the most appropriate recovery strategy when the PT cannot participate in recovery. Finally, we propose (iii) an evaluation methodology on a conveyor belt DT prototype. The results show that state reproduction is achievable at a measurable and predictable cost.

**Keywords:** Digital Twin as a Service · Digital twins · Log-based recovery · State reproducibility.

## 1 Introduction

Digital Twins (DTs) are increasingly deployed as stateful services in Internet of Things (IoT) and Cyber-Physical System (CPS) architectures, providing continuously updated virtual representations of physical assets, the so-called Physical Twin (PT), to downstream consumers [5,3].

The Digital Twin as a Service (DTaaS) paradigm encapsulates DT functions as modular, reusable services that support lifecycle operations such as deployment, migration, and fault recovery [3].

However, supporting these operations for stateful DTs raises an issue that existing DTaaS approaches do not address. When a DT service is migrated, re-deployed, or recovered after a fault, restarting the service or reattaching external storage is not sufficient to restore its state. Consider the DT of the Computer Numerical Control (CNC) machine with a conveyor belt [6]. Its state includes

---

<sup>\*</sup> Corresponding author.

information derived from sequences of PT updates, e.g., average temperature trends and total machine cycles. If part of this sequence is lost or reordered, the resumed DT may no longer represent the state it would have reached without the disruption.

In this paper, we argue that *state reproducibility* is an important property that a DTaaS deployment must support for stateful DTs to keep working across lifecycle operations and disruptions. Informally, a DT is state-reproducible if, given sufficient recorded information about past observations of its PT, its state can be reconstructed at any point in time by replaying those observations in the same order they were originally processed.

It is worth noting that formalizing state reproducibility requires revisiting classical rollback-recovery theory [4]. A DT differs from a classical distributed system in one fundamental way: its state is driven by a PT that may not be able to roll back, may not be pausable, and may not participate in any recovery protocol. Thus, existing theory must be adapted to fit this setting.

Based on these considerations, in this paper we (i) provide a semi-formal definition of DT state reproducibility grounded in rollback-recovery theory, (ii) argue that pessimistic logging is the appropriate recovery strategy when the PT cannot participate in recovery, and (iii) propose an evaluation methodology, which we apply to a conveyor belt DT prototype.

The remainder of the paper is structured as follows. Section 2 provides background and motivation, Section 3 defines state reproducibility, Section 4 describes the implementation, Section 5 presents the evaluation, and Section 6 concludes the paper.

## 2 Background and motivation

**Background.** As defined by Grieves [5], a DT model consists of three elements: a physical element that exists or will exist in the physical space, a virtual counterpart that exists in the virtual space (the DT), and a connection between them. Henceforth, the term PT refers to the physical counterpart of a DT.

Minerva et al. [7] identified a set of characterizing properties of DTs in the context of IoT. Specifically, a subset of those properties was identified as foundational: representativeness and contextualization, mirroring, and entanglement. The **entanglement property**, to which we refer as DT entanglement and recently made available as the Overall Digital Twin Entanglement (ODTE) metric [2], refers to the connection between the DT and its PT. Ideally, this connection makes the two counterparts instantaneously connected, meaning that any change in the PT is instantly reflected in the DT, and vice versa. Practically, this cannot happen: moving data over a network or processing data always takes time. As a rule of thumb, the average time elapsed between two changes should be considered the upper limit for updating the counterpart [7]. When this holds, the two counterparts are said to be entangled.

A stateful DT computes quantities derived from sequences of PT observations, e.g., rolling statistics or cumulative counters. Therefore, its state depends

on the history of past PT observations, not only the most recent one. A DT that has missed or incorrectly processed past observations holds a state that diverges from the current PT condition.

DTaaS platforms encapsulate core DT functions, e.g., data collection, entanglement, modelling, and analytics, as reusable service components, reducing development effort and enabling deployment across heterogeneous infrastructures [1,3]. As services, DTs are subject to lifecycle operations such as deployment, migration, scaling, and fault recovery. These operations may disrupt the DT and require its state to be restored.

Recent work has identified DT lifecycle management as an open challenge. Villegas et al. [8] note that DT research still lacks management frameworks covering their entire lifecycle. Duran et al. [3] frame DTaaS platforms as modular, scalable and service-oriented architectures, and discuss orchestration operations without specifying the conditions under which the internal state of a DT can be correctly reconstructed after such lifecycle operations.

Deterministic replay is a well-established technique in distributed systems. The foundational theory is provided by Elnozahy et al. [4], who classify rollback-recovery protocols and introduce the piecewise deterministic assumption and the concept of determinants. Elnozahy et al. model the outside world as a process that cannot roll back and cannot participate in recovery, but treat it as a peripheral source of inputs. In a DT context, the PT may play the same role. The relevant difference is that the periodic updates sent by the PT are the primary driver of the state transitions in the DT.

***Motivating case and challenges.*** Consider the example of a DT of a CNC machine briefly introduced in Section 1. The PT periodically sends tool wear measurements, spindle speed readings, temperature sensor updates, and machining cycle completions to the DT, which maintains a gathered state mirroring current PT values and a derived state computed from sequences of past observations. In case of a disruption, the DT must be restored to a previous valid state. The derived state cannot be reconstructed from the most recent observation alone: a sufficient sequence of past events must be available. The following scenarios illustrate why.

**C1 - Runtime state reconstruction.** When a DT is migrated to a new host, its runtime state — the volatile, in-memory information representing its current operational context — must be reconstructed at the target. While persistent data stored externally can be reattached transparently, runtime state requires explicit handling.

**C2 - Fault or inconsistent state recovery.** Wrong information from the PT may cause the DT to reach a faulty or inconsistent state, triggering unnecessary or unsafe actions, e.g., a corrupted temperature reading triggering a false maintenance alert or modification in the set temperature. Recovery requires reverting to the last known correct state that preceded the fault.

Both scenarios share a common requirement: given sufficient recorded information, it must be possible to reproduce the DT state at any point in time. We refer to this property as *state reproducibility*.

### 3 Our characterization of DT state reproducibility

We model the DT as a stateful service whose execution unfolds over a totally ordered sequence of events observed from the PT. At any point in time, the DT state reflects the information accumulated from those events since its instantiation. State transitions are governed by a *transition function* that takes the current state and a transition input and produces the next DT state.

#### 3.1 Core definitions

It is worth noting that it is not required that the transition function is deterministic by construction, as it may depend on information not encoded in its arguments, such as internal timers, concurrent writes, or external lookups. In fact, by following Elnozahy et al. [4], we observe that any nondeterministic behavior in the transition function can be resolved by recording sufficient information about the events that trigger state transitions. In log-based rollback-recovery, this information is called the determinant of the nondeterministic event: once the determinant is recorded in a *stable storage*, i.e., storage that survives the failures the system is designed to tolerate, the event can be replayed deterministically during recovery.

**Definition 1 (Determinant).** *The determinant of a nondeterministic event is the information that, when recorded at the time of the event, is sufficient to reproduce the same nondeterministic outcome during replay. A determinant resolves all nondeterminism in the transition function that is not already encoded in the current state.*

We adopt this principle and apply it to the DT context. For a determinant to enable correct replay, the transition input it produces must be sufficient.

**Definition 2 (Sufficient Transition Input).** *A transition input is sufficient if it contains enough information to constrain the transition function to exactly one output state, given the current state. A transition function applied to a sufficient input behaves as a pure function: the same state and the same input always produce the same output.*

The definition does not prescribe what the transition inputs must contain, as that depends on the specific transition function of the DT under consideration. What it requires is that whatever nondeterminism is present in the transition function, the determinants resolve it completely. In the DT context, nondeterminism arises primarily from the ordering in which PT events are processed: the same set of events processed in different orders may produce different states. The determinant must therefore capture sufficient information to establish a total order over processed events. How that order is determined is an implementation choice that affects reproducibility guarantees and must be made explicit.

**Definition 3 (DT State Reproducibility).** *A DT is state-reproducible over a time window if, for every transition within that window, sufficient determinants have been recorded that the DT state can be reproduced by replaying the complete, ordered sequence of those determinants, starting from a known state immediately preceding the first recorded information, without directly observing or copying intermediate DT states.*

Replaying the complete determinant sequence from the initial state guarantees reproducibility but may be costly for long-running DTs with large logs. Following Elnozahy et al. [4], log-based recovery can be combined with *checkpointing*, i.e., periodic snapshots of the DT state saved to stable storage.

Note that a checkpoint is typically much larger than a determinant and thus more expensive to be permanently stored and processed. Reproduction from a checkpoint is correct as long as the sequence of determinants recorded after the checkpoint is complete.

### 3.2 Log-based recovery: pessimistic logging towards a solution

DT state reproducibility does not require the PT to roll back or participate in any recovery procedure. The PT is treated as an outside world process in the sense of Elnozahy et al. [4]: it cannot roll back, it cannot be paused, and it has no awareness of the DT state reproduction process. This is precisely why log-based methods are the appropriate foundation for DT state reproducibility. In fact, log-based recovery is particularly suited to applications that frequently interact with the outside world and, for DTs, the PT continuously generates events that drive DT state transitions.

Among the three flavors of log-based recovery [4], i.e., optimistic, causal, and pessimistic, we state that pessimistic logging is the appropriate choice for DT state reproducibility.

*Optimistic logging* defers persistence of determinants, accepting the risk that a failure may cause some to be lost. In a DT, the PT may not be able to regenerate past observations. While middleware or broker-level mechanisms may buffer messages and partially compensate for this risk, such mechanisms operate outside the DT.

*Causal logging* requires piggybacking dependency information on messages exchanged between coordinated processes, which may have no equivalent in the DT-PT interaction since the PT may not support a recovery protocol.

*Pessimistic logging* requires persisting each determinant before the corresponding event is admitted to the processing pipeline, and therefore it is the only flavor that maintains the reproducibility guarantee given an uncoordinated, non-regenerating PT.

### 3.3 Operational considerations for building a solution

Determinants have to be stored in *stable storage* [4] and state reproducibility imposes a requirement on the nature of the stable storage. A DT may be redeployed or migrated to a different host as part of regular operations, but a local

stable storage tied to a specific host would not survive such operations, making previously recorded determinants unavailable. This work does not address recovery strategies for the case in which state reproduction itself fails, e.g., due to corruption or incomplete determinant logs.

In addition, state reproduction may temporarily break DT-PT entanglement. While the DT replays past determinants, it is no longer processing incoming PT events. During this interval, PT *updates must be buffered* to avoid loss. Once reproduction is complete and the DT resumes processing live events, entanglement is restored naturally as the DT catches up with the current PT state. Alternative architectures may reduce or eliminate this entanglement break, but are outside the scope of this work.

A DT may also issue commands to the PT. Re-issuing them during reproduction may cause unintended physical consequences, since the PT may have already acted on them. This is an instance of the output commit problem [4], which we leave out of scope.

## 4 Prototype implementation

We implemented the DT of a conveyor belt together with an emulated PT. Figure 1 shows the architecture of the prototype and the interaction between components during normal operations and state reproduction. We model the PT as an outside-world process. The prototype consists of an emulated PT, a DT, a Mosquitto Message Queuing Telemetry Transport (MQTT) broker, and a MongoDB instance acting as stable storage, consistent with the requirement established in Section 3.3.

The black solid flow of Figure 1 shows the normal operation flow, in which a DT processes a PT update. The PT, through the *connection handler*, ① publishes an update to the *MQTT broker*, which ② delivers it to the DT *connection handler*. The *connection handler* ③ forwards the update to the *determinant logger*, which creates the determinant and ④ persists it. Once the determinant is safely persisted, the *determinant logger* notifies the *connection handler* that sends the PT update to the *gathered state processor* and *derived state processor* so they can ⑤ execute the state transition.

In our prototype, nondeterminism arises from two sources: the sequence of updates received from the PT and the order in which concurrent handlers commit state-changing transitions. Each state transition is executed atomically and each state-changing event is assigned a monotonically increasing sequence number at receipt, before any processing occurs. The ordering policy adopted in this prototype is FIFO: events are processed in the order they are received by the DT, ensuring that replay follows the same sequence and produces the same state. The determinant of each PT update consists of the event payload and its assigned sequence number.

The blue dashed flow of Figure 1 shows the operations flow during state reproduction. State reproduction is ① triggered by an *external invoker*, e.g., a Kubernetes component or a DTaaS platform manager, via a REST API exposed

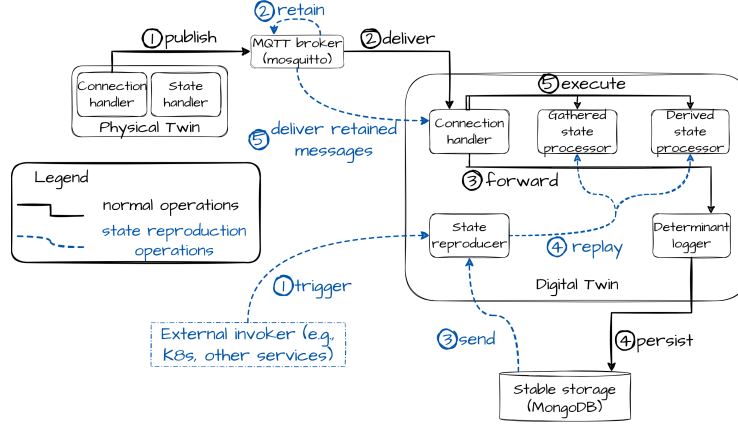


Fig. 1: Architecture of the prototype (some arrows are omitted for readability).

by the DT. First, the DT disconnects from the *MQTT broker* to ensure no new state transition occurs during reproduction. To satisfy the buffering requirement (see Section 3.3), the DT connects to the *MQTT broker* using a persistent session with a fixed client identifier: incoming PT updates are ② retained by the *MQTT broker* while the DT is disconnected. Then, the DT resets to an empty initial state. The *state reproducer* queries *stable storage (MongoDB)*, which ③ sends all stored determinants ordered by the assigned sequence number. The *state reproducer* forwards the determinants to the *gathered state processor* and *derived state processor* so they can ④ replay them through the transition function. Once reproduction is complete, the DT reconnects to the *MQTT broker*, which ⑤ delivers all retained messages before normal operation resumes.

State equality is verified after each reproduction by comparing the logical state of the reproduced DT against the original field by field: the gathered state variables, the derived state buffer, and the connection buffer entries. Execution metadata, i.e., message receipt timestamps and processing times, are excluded from the comparison, as they reflect execution conditions, e.g., network timing and hardware speed, rather than the logical state of the DT.

The DT-PT entanglement is not part of the logical state subject to reproduction. It is undefined during state reproduction, as the connection with the PT through the *MQTT broker* is intentionally suspended.

## 5 Evaluation

**Setup.** The evaluation is conducted using a containerized deployment managed with Docker Compose v5.5.1 and Docker v29.8.0 on a single host (Intel Core Ultra 7 155U, 12 cores / 14 threads, 32 GB RAM, Ubuntu 24.04.5 LTS). The setup consists of four components: the emulated PT, the DT, a MongoDB v6.0.5 instance acting as stable storage, and the Mosquitto MQTT broker v2.0.15. All

experiments are repeated 10 times per configuration. The prototype serves as feasibility validation of the methodology, rather than as a benchmark, and the metrics apply to any DT scenario. We address three **research questions**:

- **RQ1.** How does state reproduction time scale with the number of stored determinants? We measure *state reproduction time*: the elapsed time from the moment state reproduction is triggered to the moment the DT state has been fully reconstructed;
- **RQ2.** What is the per-event latency introduced by pessimistic determinant logging compared to the transition function execution? We measure *write latency*, the time to persist a determinant before the event is admitted to the processing pipeline, and *processing latency*, the time to execute the transition function for a single PT update, excluding persistence;
- **RQ3.** How long does the DT take to become fully operational again after state reproduction? We measure ODTE recovery time: the elapsed time from the moment state reproduction is completed to the moment the ODTE is restored above a threshold of  $\tau = 0.9$ .

To evaluate reproduction time and entanglement recovery, the PT update rate is fixed at  $r = 5$  events per second, the determinant BSON-encoded size is approximately 0.4 KB, and the number of stored determinants is varied across  $N \in \{100, 200, 500, 1000, 2000, 5000, 10000\}$ . To evaluate per-event logging and processing latency, the number of determinants is fixed at 500 and the PT update rate is varied across  $r \in \{1, 5, 10, 20, 50, 100, 200\}$  events per second. To evaluate the effect of determinant size on write latency, the update rate is fixed at  $r = 5$  events per second, the number of determinants is fixed at 500, and the determinant size is varied across  $\{0.4, 10, 100, 500, 1024, 5120, 10240\}$  KB by padding each document to the target BSON-encoded size before persistence. DT-PT entanglement recovery is based on the ODTE metric [2], which ranges from 0 to 1, where 1 denotes perfect entanglement. The ODTE computation window is set to 10s with a refresh interval of 0.5s. Entanglement recovery time is measured from the moment the DT reconnects to the PT through the MQTT broker after state reproduction until ODTE exceeds a threshold of  $\tau = 0.9$ .

**Results. (RQ1, RQ3) — State Reproduction Time and Time to Fully Operational DT.** Figure 2 shows state reproduction time and total time to a fully operational DT as a function of the number of stored determinants  $N$ . State reproduction time grows linearly with  $N$ , ranging from 0.08s at  $N = 100$  to 5.46s at  $N = 10,000$ . This linear scaling is consistent with the sequential nature of determinant replay. State equality was verified in all rounds across all configurations, confirming that the sufficiency condition is satisfied by the recorded determinants in every case.

ODTE recovery time takes one of two values across all runs, 9.10s or 9.61s, independent of  $N$ . The 0.5s difference corresponds to the ODTE refresh interval, at which the crossing of  $\tau$  is detected. This result is consistent with the ODTE computation window of 10s used in the prototype. In deployments with longer computation windows, recovery time may increase accordingly.

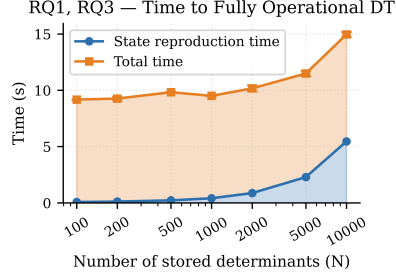


Fig. 2: Time required to fully operational DT after state reproduction.

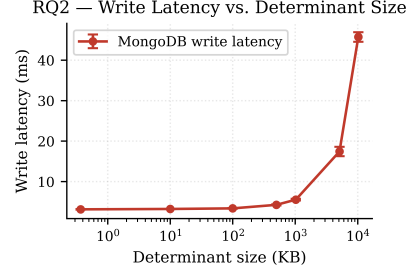


Fig. 3: Write latency as a function of determinant size.

The total time to a fully operational DT, i.e., the sum of reproduction time and entanglement recovery time, grows from 9.2s at  $N = 100$  to 15.0s at  $N = 10,000$ . For small event logs, total time is dominated by entanglement recovery, since few determinants need to be replayed and each is processed with negligible overhead relative to the fixed ODTE computation window. For large logs, reproduction time may become the dominant factor.

**(RQ2) — Per-Event Latency.** Write latency averages  $2.87 \pm 0.38$  ms per event and transition function latency averages  $0.72 \pm 0.07$  ms, for a total per-event cost of approximately 3.6 ms in the prototype. The write latency depends on event payload size and network latency between the DT and its stable storage backend; the reported values are measured in a single-machine deployment with no network latency between components, and represent a lower bound for distributed deployments. In deployments with more complex transition functions, the balance between logging and processing costs may differ.

Figure 3 shows write latency as a function of determinant size. Write latency remains stable at approximately  $3.2 \pm 0.1$  ms for determinant sizes up to 100 KB. Above this threshold, latency increases significantly, reaching  $5.5 \pm 0.2$  ms at 1 MB and  $45.7 \pm 1.2$  ms at 10 MB.

Overall, the results indicate that DT state reproducibility is achievable at a measurable and predictable cost.

## 6 Conclusions

We argued that DT state reproducibility is an important property for dependable lifecycle management of stateful DTs. The key observation is that a DT differs from classical distributed processes because its state is continuously driven by a PT that may not roll back, may not be pausable, and may not be able to regenerate past observations on demand. For this reason, lifecycle operations require explicit conditions under which the DT state can be reconstructed from recorded information.

Building on log-based rollback-recovery theory, we characterized DT state reproducibility in terms of sufficient determinants, ordered replay, and reconstruc-

tion from a known state without copying intermediate states. On a conveyor belt DT prototype, we show that state reproducibility is achievable at a measurable and predictable cost.

Several challenges remain open. Out-of-order events require PT-side ordering semantics, since FIFO ordering at receipt cannot restore causal order. Nondeterminism not captured by determinants must be logged or excluded. Crash consistency across persistence, state update, and acknowledgment, including duplicate deliveries, requires a precise failure model. Checkpointing can bound reproduction time, and DTs that command their PT face the output commit problem. Distributed deployments and comparison with event sourcing and broker-based replay remain to be studied.

**Acknowledgments.** This work was partially funded by the Sustainable Digitalisation Research Centre at Malmö University.

**Disclosure of Interests.** The authors have no competing interests to declare that are relevant to the content of this article.

## References

1. Ahelerooff, S., Xu, X., Zhong, R.Y., Lu, Y.: Digital twin as a service (dtaas) in industry 4.0: An architecture reference model. *Advanced Engineering Informatics* **47** (2021). <https://doi.org/10.1016/j.aei.2020.101225>
2. Bellavista, P., Bicocchi, N., Fogli, M., Giannelli, C., Mamei, M., Picone, M.: Odte: A metric for digital twin entanglement. *IEEE Open Journal of the Communications Society* **5**, 2377–2390 (2024). <https://doi.org/10.1109/OJCOMS.2024.3385659>
3. Duran, K., Verda Cakir, L., Yigit, Y., Huseynov, K., Ram Kusu, S., Ali Erturk, M., Canberk, B.: Toward digital twin-as-a-service (dtaas) platforms: A survey on architecture, design requirements, and performance metrics. *IEEE Communications Surveys and Tutorials* **28**, 1845 – 1878 (2026). <https://doi.org/10.1109/COMST.2025.3635582>
4. Elnozahy, E., Alvisi, L., Wang, Y.M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys* **34**(3), 375 – 408 (2002). <https://doi.org/10.1145/568522.568525>
5. Grieves, M.W.: *Digital Twins: Past, Present, and Future*, vol. 1. Springer Nature (2023). [https://doi.org/10.1007/978-3-031-21343-4\\_4](https://doi.org/10.1007/978-3-031-21343-4_4)
6. Magalhães, L.C., Magalhães, L.C., Ramos, J.B., Moura, L.R., de Moraes, R.E.N., Gonçalves, J.B., Hisatugu, W.H., Souza, M.T., de Lacalle, L.N.L., Ferreira, J.C.E.: Conceiving a digital twin for a flexible manufacturing system. *Applied Sciences (Switzerland)* **12**(19) (2022). <https://doi.org/10.3390/app12199864>
7. Minerva, R., Lee, G.M., Crespi, N.: Digital twin in the iot context: A survey on technical features, scenarios, and architectural models. *Proceedings of the IEEE* **108**(10), 1785–1824 (2020). <https://doi.org/10.1109/JPROC.2020.2998530>
8. Villegas, L.F., Romero, D., Macchi, M., Skoogh, A.: Towards a digital twin lifecycle management framework. *Enterprise Information Systems* **0**(0), 2616681 (2026). <https://doi.org/10.1080/17517575.2026.2616681>