

ACCEPTED MANUSCRIPT

Scale to Stay Entangled: An Empirical Study on Digital Twin Scalability

G. Mazziotta, M. Fogli and C. Giannelli

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

Cite this as

G. Mazziotta, M. Fogli and C. Giannelli, “Scale to Stay Entangled: An Empirical Study on Digital Twin Scalability,” 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>

Scale to Stay Entangled: An Empirical Study on Digital Twin Scalability

Gaia Mazziotta , Mattia Fogli (✉) , and Carlo Giannelli 

University of Ferrara, Italy

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

Abstract. A digital twin (DT) is a virtual element that, for at least part of its life cycle, is entangled with a physical element. Entanglement holds when the physical space and the virtual space reconcile in a timely manner. What “timely” means may vary from application to application. However, the consequence of missing it does not: a DT whose reconciliation with the physical space arrives too late no longer represents its counterpart, and thus no longer works correctly. Keeping reconciliation on time is hard since the workload of a DT changes unpredictably during its life: physical states may grow in volume, new functions may be added, user requests may spike. Scalability, for a DT, is therefore a matter of correctness before being one of performance. Still, scaling decisions are commonly driven by metrics—resource usage, throughput, request rate—that capture only performance, and how architectural design affects the ability of a DT to stay entangled under growing load remains largely unexplored. To bridge this gap, this work proposes twinning isolation—the extent to which the twinning process can scale independently of the rest of the DT—as a lens for comparing service-oriented architectures; distills design principles by examining four canonical patterns (monolithic, microservice, serverless, and hybrid) through this lens; and empirically validates such principles on an implementation of each pattern under growing user requests, measuring DT entanglement along with traditional metrics.

Keywords: Digital twins · Scalability · Service-oriented architectures.

1 Introduction

A digital twin (DT) [18] is a virtual element that, for at least part of its life cycle, is entangled with a physical element, the so-called physical twin (PT). Ideally, DT entanglement requires that changes in the physical space are reflected instantaneously in the virtual space, and vice versa [24]. However, this cannot happen: data takes time to travel between the spaces and to be processed. When reconciliation between the two spaces is “too slow” (what this means may vary widely from application to application), entanglement does not hold. If so, the DT no longer represents its physical counterpart correctly. Therefore, whether a DT works correctly also depends on entanglement, not only on model fidelity, response time, or resource usage.

Based on these considerations, we claim that DT scalability [10,11] is a correctness concern, not just a performance one. In particular, we focus on the *twinning process*, which implements the chain of operations that reconciles the virtual and physical spaces. In other words, the twinning process is the part of a DT that has to keep up with the PT for the entanglement to hold. During the lifetime of a DT, its workload may vary unpredictably (e.g., the volume of physical states may grow, augmentation services may be invoked, or user requests may spike), possibly slowing the twinning process as a side effect. A DT that cannot properly scale therefore risks losing entanglement with its PT.

Service-oriented computing has become the default way to build and operate cloud applications, and DTs increasingly follow the same path: they are built from services, deployed on service platforms, and offered as services themselves—Digital Twin-as-a-Service (DTaaS) [2,12]. Resource usage, throughput, and request rate are traditional metrics typically used to decide whether to scale in or out a cloud service. We claim that such metrics are not sufficient to properly trigger DT scaling and, consequently, ensure DT entanglement. In particular, this work shows across four canonical service-oriented architecture patterns (monolithic, microservice, serverless, and hybrid) that a DT can lose entanglement while traditional performance metrics report a loaded system correctly scaling as designed. To quantify DT entanglement, we use the Overall Digital Twin Entanglement (ODTE) metric [9].

The contributions of this work are the following:

- We propose *twinning isolation*—the extent to which the twinning process can scale independently of the rest of the DT—as a lens for comparing service-oriented architecture patterns when scalability is a matter of DT correctness;
- We distill five design principles by examining four canonical service-oriented architecture patterns through this lens;
- We conduct an empirical study on four implementations of the same DT, one per pattern, showing under which conditions DT entanglement survives scaling driven by growing user requests and that traditional performance metrics fail to detect entanglement degradation.

The rest of the paper is organized as follows. Section 2 introduces DTs, entanglement, and ODTE, and explains why scalability is a correctness problem in this context. Section 3 motivates twinning isolation as the design lens, compares the four patterns through it, and states five design principles. Section 4 describes the implementation of each pattern. Section 5 reports the empirical study: methodology, baseline, and scale-out results. Section 6 positions the work against prior efforts. Section 7 concludes the paper.

2 Background

2.1 Digital Twins and Entanglement

While the concept of DT was first introduced more than twenty years ago by Grieves [17], there is still no general consensus around its definition [19,23,28]. The interested reader may refer to [7] for a survey on the proposed definitions.

Several taxonomies refine the concept [20]; for our purposes, the foundational property is *entanglement* [24]—the property that the physical space and the virtual space reconcile in a timely manner. Note that what “timely” means may vary from application to application. However, regardless of the application context, a DT that is not entangled with its physical counterpart (i.e., DT entanglement does not hold) no longer works correctly.

For example, consider an operator running an offline what-if analysis on a DT: an arbitrarily slow response is tolerable as long as the result remains logically correct. The same operator using the DT to choreograph physical actions on the production line cannot tolerate the same delay—if the actions arrive in the wrong order (a consequence of concurrency [21]) or too late (a consequence of real time [27]), the reconfiguration fails. Order and timing matter just as much in the opposite direction, as physical states are mirrored into the DT. Consider the pose of a robot arm: a state that arrives too late may show the arm where it was rather than where it is, and two states applied in the wrong order may even show it moving backwards.

2.2 Measuring Digital Twin Entanglement

Bellavista et al. [9] proposed ODTE, a $[0, 1]$ -valued measure obtained as the product of three factors observed over a time window TW :

$$\text{ODTE} = T \times R \times A, \quad (1)$$

- *Timeliness* T : the fraction of the physical states received by the DT that arrived within a deadline T_d ;
- *Reliability* R : the fraction of the expected physical states—those implied by the update rate R_e over TW —that the DT received;
- *Availability* A : the fraction of time the PT is observed up from the perspective of the DT.

For example, $T(T_d=1\text{ s}, TW=5\text{ min}) = 0.999$ means that 99.9% of the physical states received over the last 5 minutes arrived within 1 s. We use ODTE operationally: a DT is fit for purpose while its ODTE stays above a threshold, and stops being so when ODTE falls below it—whatever traditional performance metrics report at that moment. Both the threshold and the deadline T_d are application choices. Our empirical study therefore reports ODTE across a range of deadlines. Note that ODTE captures and scores whether and when physical states arrive, not their order, which we treat as a design property (Section 3).

2.3 Why Scalability Is a Correctness Problem

The amount of work a DT handles varies over time. Some load shifts land on the twinning process directly, such as physical states growing in volume or complexity. Others land elsewhere, such as surging user requests on analytical services,

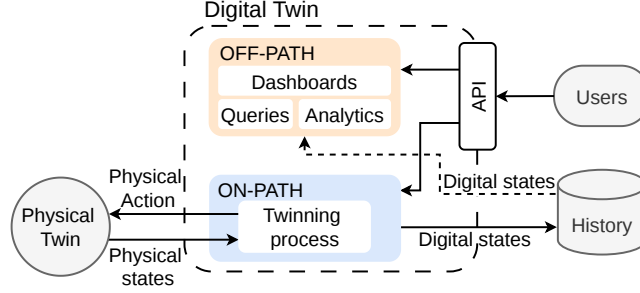


Fig. 1. On-path (bottom) and off-path (top) components of a DT

yet may still slow the twinning process through contention on shared CPU, memory, or network. Either way, a DT that cannot properly absorb the additional workload may lower its entanglement below a desired threshold, which is a DT correctness failure. In fact, although entanglement does not necessarily imply correctness (entanglement may hold but the model may be wrong), when entanglement does not hold, neither does correctness. In other words, entanglement is necessary but not sufficient for correctness.

Bondi [11] distinguishes four types of scalability: load, space, space-time, and structural. We focus on *load scalability*, which for DTs we restate as handling an increasing workload without ODTE degradation rather than without performance degradation, and *structural scalability*, which is the ability to grow without major changes to the core structure. Both can be reasoned about independently of the algorithms and data structures adopted internally to a DT. Space and space-time scalability are tied to those internals and thus are out of scope for this work.

3 Designing Scalable Digital Twins

We claim that some parts of a DT must scale for correctness, and others only for performance. The twinning process is the part in charge of entanglement: any slowdown delays the reconciliation between the two spaces and threatens it. When entanglement breaks, so does correctness—even if every digital state is computed correctly. A state produced too late is still the right value for the physical state that caused it. Yet by the time it lands, the physical space has moved on, so the virtual space no longer matches reality. The rest of the DT, e.g., dashboards, exploratory queries, or batch analytics, scales only for performance: a delay may lower user experience but not the correctness of the responses. We refer to the components implementing the twinning process as *on-path*, and to all the others as *off-path*. Fig. 1 depicts this distinction. The twinning process, shown at the bottom, forms the on-path region: it takes physical states from the PT, transforms them into digital states that it writes to the history store, and sends physical actions back. Here entanglement is at stake. Above it, the off-

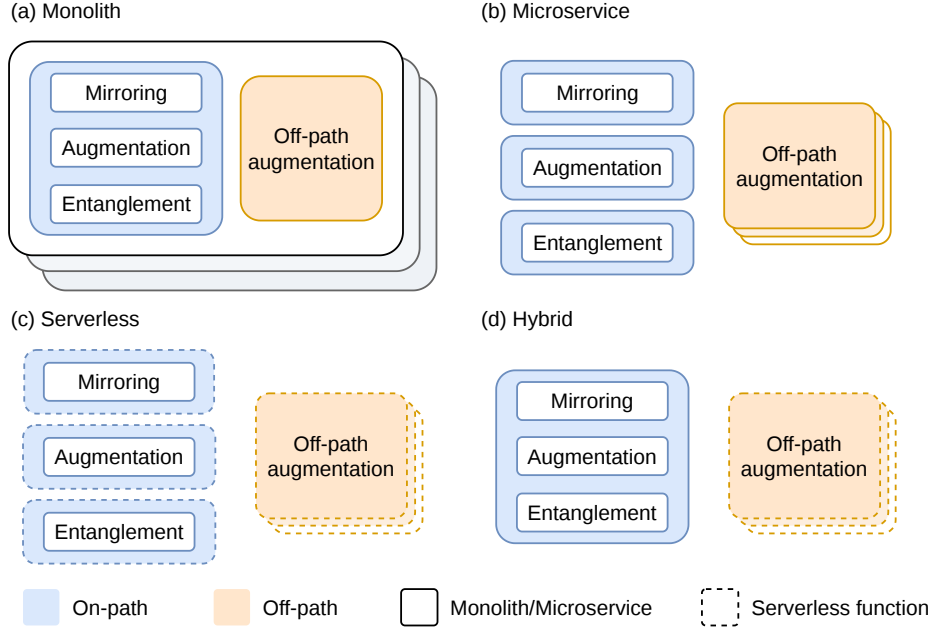


Fig. 2. Service-oriented architecture patterns applied to DT design

path region holds the components that serve the user requests arriving through the DT API, such as dashboards, queries, and analytics over the stored digital states. Here only quality of service (QoS) is at stake.

We call *twinning isolation* the extent to which a service-oriented architecture lets on-path and off-path components be scaled independently. With high twinning isolation, resources can be directed to whichever side is under stress without interfering with the other; with low twinning isolation, they cannot: off-path load then reaches the twinning process, where it delays or drops physical states. We first rate twinning isolation structurally, then observe its effect on T and R in Section 5. This matters since autoscalers act at whatever granularity the architecture exposes. Without twinning isolation, every scaling unit mixes correctness-critical work (the twinning process) with performance-only work (off-path services), so scaling the one necessarily scales the other.

We consider four canonical service-oriented architecture patterns as possible designs of a DT (Fig. 2):

- **Monolith.** A software application whose modules cannot be executed independently [15].
- **Microservice.** A suite of small services, each running in its own process and communicating over lightweight mechanisms [22].
- **Serverless.** An application exposed as short-running, event-triggered functions that the platform invokes on demand and autoscales transparently [13,25].
- **Hybrid.** A combination of two or more of the above.

Table 1. Service-oriented architecture patterns and design axes ($\checkmark$ = yes, $\times$ = no, $\circ$ = depends on the hybrid constituents)

Pattern	Twinning isolation	Scaling granularity	Inter-component calls	State-update ordering
Monolith	$\times$	Application	In-memory	$\checkmark$
Microservice	$\checkmark$	Service	RPC	$\checkmark$
Serverless	$\checkmark$	Function	Framework-mediated RPC	$\times$
Hybrid	$\circ$	Mixed	Mixed	$\circ$

Table 1 compares the four patterns along four design axes. The first two, *twinning isolation* and *scaling granularity*, are properties of the pattern itself; the last two, *inter-component calls* and *state-update ordering*, are the consequences those properties impose on the twinning process. For *state-update ordering*, we assume the weakest common policy: each physical state is processed in the order it was generated, so that the arm of Section 2.1 never appears to move backwards. When a pattern does not preserve this order by design, an external mechanism must restore it—for example, a buffer that releases states in sequence—and the twinning process pays its latency.

Design Principle 1. *Twinning isolation rises with scaling granularity, then saturates.* The monolith has no isolation: the unit of scaling is the entire application, so off-path workloads compete with the twinning process for the same resources. The microservice isolates the twinning process—a single business capability—in its own service, which can scale independently of off-path services. The serverless pattern splits that same capability into several functions. This does not add isolation, but it spreads the twinning process across more components, each adding invocation overhead. Once the twinning process is in its own deployable unit, finer subdivision multiplies on-path components without increasing isolation.

Design Principle 2. *Finer granularity costs more on inter-component communication.* Within the twinning process, calls progress from in-memory (monolith) to RPC (microservice) to framework-mediated RPC (serverless). In-memory calls are the cheapest; RPC adds serialization and network calls; framework mediation adds queueing and invocation machinery on top. The twinning process pays this cost on every physical state, so the cost consumes the latency margin left by the entanglement deadline.

Design Principle 3. *Preserving state-update ordering may be an anti-pattern.* Some patterns preserve state-update ordering for free, while others must fight their own premise to keep it. The monolith preserves it implicitly: a single in-memory chain processes states one after the other. The microservice preserves it by funneling all states through one mirroring service. The serverless does not: each state triggers a function chain that runs in parallel with the chains of its predecessors, and the platform offers no ordering guarantee across invocations.

Restoring order requires external coordination, which works against the stateless, parallel-by-default premise of the pattern [25].

Design Principle 4. *Replicate the twinning process only as a last resort.* This applies when the specification of a PT fixes what it emits, when, and how often: the twinning load is then known in advance and can be provisioned up front, while off-path load, driven by arbitrary user requests, is what needs to scale elastically. Patterns with twinning isolation exploit this: they absorb off-path load by scaling off-path components only, leaving the twinning process unreplicated and preserving ordering at no extra cost. The monolith cannot: it scales by replicating the whole application, so ordering breaks as soon as a second replica processes physical states, unless a coordination mechanism (e.g., leader election) re-imposes it. When the principle does not apply, replication may become necessary, and with it the ordering cost of Design Principle 3.

Design Principle 5. *Pick the best pattern for each side.* Assigning the two sides to different patterns, deployed and scaled independently, yields twinning isolation by construction: off-path scale-out cannot slow the twinning process. Inter-component call cost and state-update ordering are instead inherited from the pattern running the on-path. This resolves the $\circ$ entries of Table 1: they depend on the constituents in general, but are fixed once the split follows the boundary. The principle also yields a design recipe: choose the cheapest ordering-preserving pattern for the on-path, and a pattern with fine scaling granularity for the off-path.

4 Implementation

We implemented one DT per pattern to test the isolation and cost principles of Section 3 (Design Principles 1, 2, 4, and 5). Design Principle 3 concerns physical states that arrive faster than the on-path chain processes them, i.e., a growing twinning load, which this study holds fixed. All four implementations share the same twinning pipeline, the same off-path workload, and the same code base; they differ only in how the pipeline is decomposed into deployable components and in the supporting infrastructure that decomposition entails—call mechanism, broker, autoscaler (Fig. 2). Each pattern is therefore compared as deployed with its idiomatic infrastructure, so measured differences reflect the pattern and its infrastructure, not the code.

The on-path chain applies three stages to every physical state coming in from a PT: (i) *mirror* (ingestion of the physical state and selection of the aspects of interest for the digital state), (ii) *on-path augmentation* (enrichment of the digital state, such as unit conversion or feature extraction, here emulated by a light, fixed-cost CPU-bound task that computes the first 100 primes), (iii) *entanglement* (storage of the digital state and validation of the twinning chain). Each physical state travels as a small JSON envelope carrying the PT identifier, a monotonic sequence number, the emission timestamp, and the state payload. A separate *off-path augmentation*—a CPU-intensive task computing the first 10 000 primes per HTTP request, driven independently of the physical-

state stream—provides the off-path load. A shared Python 3.11 core implements the state data structures, the observation log from which ODTE is computed post-hoc (Section 5.1), the persistence logic, and the prime-computing routine.

The *monolith* runs the three stages and the off-path task in a single process. The *microservice* splits them into four services—mirror, on-path augmentation, entanglement, and off-path augmentation—that communicate over HTTP. The *serverless* realizes the same four components as functions on Fission, an open-source serverless framework for Kubernetes, chaining the on-path ones through message-queue triggers. The *hybrid* combines the two extremes: its on-path chain runs in a single process, exactly as in the monolith, and its off-path augmentation runs as a serverless function.

5 Evaluation

5.1 Methodology

5.1.1 Testbed The testbed consists of a Kubernetes 1.29 cluster on an OpenStack tenancy: one control-plane node (2 vCPU, 4 GiB RAM) and three worker nodes (4 vCPU, 8 GiB RAM each), all on Ubuntu 22.04 with CRI-O as container runtime and Flannel for networking; a separate controller node runs the experiment scripts and the Locust load generator. As is common in production clusters, every pod of every pattern runs an Istio 1.20 service-mesh sidecar, whose latency is included in all measurements, at ingress and on each inter-component call. The ingress broker follows from the pattern: the monolith, microservice, and hybrid ingest physical states via the MQTT broker Mosquitto 2.1, with the monolith relying on an MQTT 5 shared subscription so that, when its pod is replicated, each physical state is delivered to a single replica rather than to every replica; the serverless pattern ingests via RabbitMQ 4.1, as required by the Fission message-queue trigger, and relays inter-function traffic over the same broker. In all four patterns the PT publishes at MQTT QoS 1 (at-least-once delivery). The digital-state store is Redis 8. Off-path autoscaling targets CPU: the monolith and microservice use the Kubernetes Horizontal Pod Autoscaler (HPA); the serverless and hybrid off-path scale through Fission 1.20 with a per-function ceiling. Every HPA runs with the Kubernetes defaults (15 s sync period, 5 min scale-down stabilization). We keep the patterns comparable by holding three conditions equal across them:

1. A *matched workload*. The same physical-state stream drives every pattern, and the same off-path task (prime computation over HTTP), user population, and load ramp generate the off-path load;
2. An *equal CPU budget*. Each DT totals the same CPU request/limit (0.1/1 CPU), split evenly across the pattern components: 0.1/1 for the monolith single pod, 0.05/0.5 per hybrid component, 0.025/0.25 per microservice or serverless component; the HPA set-point is the same for all patterns, so autoscaling pressure is matched even though pods may burst above their request;

3. An *equalized off-path envelope*—each pattern scalable tier is granted the same ~ 6 -core peak, so its replica cap is the envelope divided by the per-pod limit: 6 (monolith), 12 (hybrid), and 24 (microservice, serverless).

The operational realization of twinning isolation (Table 1) is that the microservice, serverless, and hybrid on-path components are pinned to a single replica, so only their off-path scales; the monolith, lacking isolation, must scale the whole application.

5.1.2 Metrics The twinning process writes two lines per physical state into a process-local observation log, joinable on (PT ID, sequence number): an *entanglement* line once the store write succeeds and a *persistence* line once persistence completes. The two-line shape distinguishes three outcomes per state—fully complete, entangled but not persisted, never twinned. Each on-path stage stamps a monotonic per-host clock (for intra-host durations) and a wall-clock (for cross-host comparisons). The three ODTE factors of Eq. (1) are computed from this log. *Timeliness* $T(T_d)$ is the fraction of received states whose end-to-end wall-clock latency is $\leq T_d$; states entangled but never persisted count in the denominator. T_d is a post-hoc parameter, swept at analysis time. *Reliability* R is the fraction of expected physical states received within the window, where the expected count follows from the update rate R_e . Receiving more states than expected does not raise R above 1. *Availability* A is fixed at 1: it measures PT uptime, which no pattern can affect, and states lost on the DT side already lower R . ODTE therefore reduces to $T \times R$ in this study. The analysis window is anchored on the first observed emission, so the window edges do not depend on controller-to-cluster clock alignment. Cluster clocks are NTP-synchronized, with an observed controller-to-cluster offset of about 3 ms. Additionally, we collect, time-aligned, the classical metrics an operator would watch: per-pod CPU utilization as a percentage of the configured CPU request (the quantity the HPA targets) and the current replica count.

5.1.3 Experiments We conduct two types of experiments, which both drive a single PT–DT pair. The study varies only the off-path load: the physical-state rate and the on-path work per state stay fixed, so each pattern is tested on how well it shields the twinning process. Experiment 1 (E1, at-rest baseline) has the PT emitting at 1 state/s with no off-path load or users; each run lasts 360 s with the first 30 s discarded as warm-up, and T_d is swept post-hoc over 1–200 ms and reported at the operating points $T_d \in \{5, 10, 25, 50, 100\}$ ms. Experiment 2 (E2, off-path pressure ramp) adds emulated users that issue the CPU-bound off-path request in a closed loop, each waiting 1 s after a response before its next request; a user thus issues at most 1 request/s, and the request volume a pattern receives falls as its response time grows. In other words, slower patterns face less, not more, off-path traffic. The user count ramps from 0 to 60 over 600 s (spawn rate 0.1 user/s) and is then held, inside a 720 s run with a 120 s warm-up, longer than in E1 to cover the HPA initial scaling rounds; a single HPA CPU target of 70% is

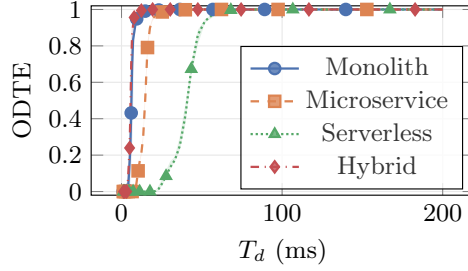


Fig. 3. E1: ODTE with T_d swept from 1 to 200 ms, per pattern

Table 2. E1: ODTE at $T_d \in \{5, 10, 25, 50, 100\}$ ms, per pattern

Pattern	$T_d=5$ ms	10 ms	25 ms	50 ms	100 ms
Monolith	0.153	0.959	0.998	1.000	1.000
Microservice	0.000	0.091	0.985	1.000	1.000
Serverless	0.000	0.000	0.036	0.912	1.000
Hybrid	0.233	0.983	1.000	1.000	1.000

used and physical state emission stays at 1 state/s throughout. Each experiment type is repeated 10 times per pattern, for 80 runs and 12 h of measurement in total.

We report ODTE in E2 at $T_d = 50$ ms, with sensitivity checks at $T_d = 25$ and 100 ms. We treat a DT as entangled when its ODTE is at least 0.9, and $T_d = 50$ ms is the tightest deadline at which all four patterns clear this threshold at rest (Table 2). Unless stated otherwise, reported values are medians across repetitions, and shaded bands, where shown, span the 25th–75th percentile across repetitions. Latency figures instead pool the per-state samples of all repetitions; boxes show quartiles, with whiskers at $1.5 \times$ the interquartile range and outliers omitted. E2 series are bucketed at 15 s (the metrics scrape interval): per-bucket ODTE, T , and R are the definitions above restricted to the bucket, curves are per-bucket medians across repetitions, and peaks and minima are taken over these curves in the post-warm-up window. We also report the off-path request rate each pattern actually served (mean over the full run, median across repetitions). This rate has a ceiling of 35 request/s, which is what the closed-loop users would collectively issue if every response were instantaneous.

All software artifacts—the four implementations with their deployment specifications, the experiment orchestrator, the raw measurements, and the pipeline that regenerates every figure and table—are publicly available.¹

5.2 Results

5.2.1 E1: At-Rest Baseline At rest, $R = 1$ for all four patterns, which means that every emitted state is received and persisted within the analysis window. Thus, ODTE is driven by T alone. Fig. 3 plots ODTE against T_d over the full 1–200 ms sweep, and Table 2 reports the values at $T_d \in \{5, 10, 25, 50, 100\}$ ms. Recall that we treat a DT as entangled when its ODTE is at least 0.9 and as fully entangled when ODTE = 1. The two patterns that run the on-path chain in a single process (monolith, hybrid) become entangled first, exceeding 0.95 by $T_d = 10$ ms; the microservice follows at 25 ms; the serverless is the slowest,

¹ <https://github.com/gaiaMzz/2026-icsoc-scalability-artifacts>

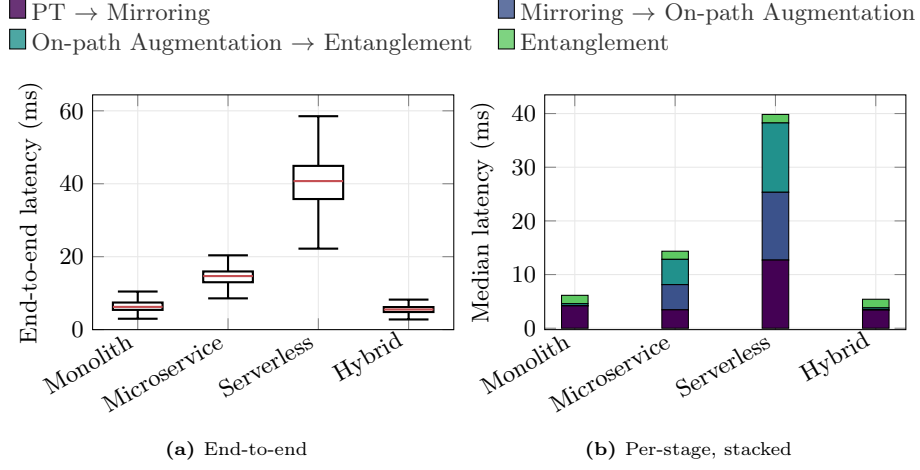


Fig. 4. E1: On-path latency per pattern

reaching only 0.91 at $T_d = 50$ ms and becoming fully entangled only at 100 ms. This latency ordering—hybrid $\approx$ monolith, then microservice, then serverless—is consistent across deadlines.

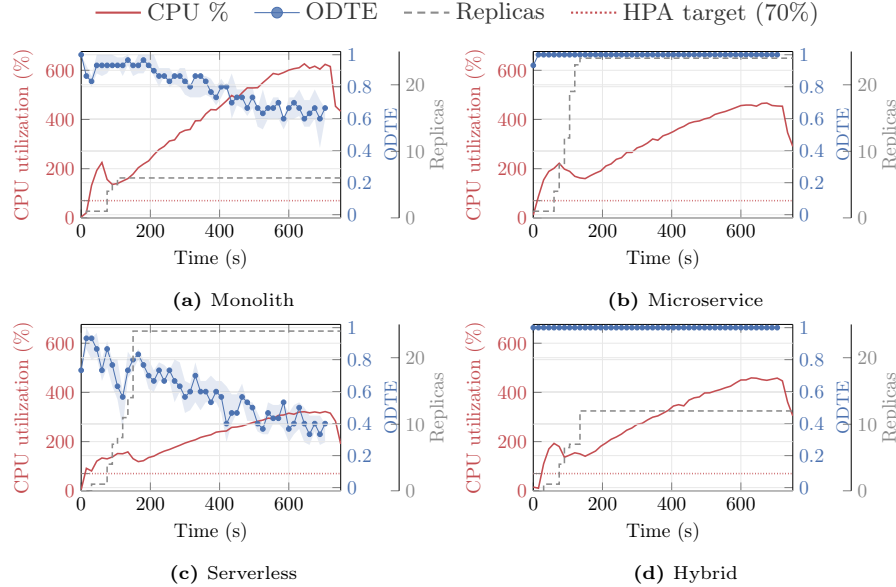
Where the Time Goes. The monolith and hybrid sit at a 5–6 ms median (Fig. 4a), the microservice at ~ 15 ms, the serverless at ~ 41 ms. The per-stage budget (Fig. 4b) localizes the cost. For the monolith and hybrid the inter-component calls collapse to near zero; their latency is ingress transport (~ 3 –4 ms, including the Istio sidecar) plus the entanglement compute (~ 1.6 ms; the persistence stamp that closes the twinning chain adds microseconds and is folded into this stage). The microservice pays two HTTP calls of ~ 4.7 ms each. The serverless pays two RabbitMQ-relayed calls of ~ 12 –13 ms each and a higher ingress (~ 12.7 ms), pushing its median to 41 ms. Since the microservice and serverless on-path are structurally identical (three components, two calls, each pinned to one replica), how the twinning chain is sliced cannot explain the ~ 26 ms gap between them. The gap is framework overhead—the cost of the framework-mediated RPC of Design Principle 2: each serverless call is not a direct HTTP request but a function invocation mediated by the broker (publish, queue, trigger). The amount of this cost, and hence the deadline at which the serverless fails, depends on the platform, here Fission over RabbitMQ, but the cost itself comes with the pattern: every serverless call pays it.

Takeaway. In E1, the load is minimal by design, so on-path latency reflects the infrastructure. The spread across patterns is the inter-component cost, and it sets the margin against the deadline T_d . More splitting and heavier calls leave less margin.

Table 3. E2: Scale-out summary with ODTE at $T_d = 50$ ms per pattern

Pattern	Request/s	Peak CPU	Peak No. of Repl.	Min ODTE	Min T	Min R
Monolith	28.9	626%	6	0.600	0.679	0.800
Microservice	22.1	467%	24	1.000	1.000	1.000
Serverless	20.7	322%	24	0.333	0.333	1.000
Hybrid	25.9	459%	12	1.000	1.000	1.000

5.2.2 E2: Off-Path Pressure Ramp Under the ramp, every pattern drives CPU far above the 70% HPA target—peak fleet utilization of 322–626% of the configured request (Table 3). This reflects pods bursting above their request, not above their limit. The autoscaler scales the off-path tier to (or near) its replica cap. According to the classical metrics, the system is loaded and scaling as designed. ODTE at $T_d = 50$ ms, however, diverges by pattern (Fig. 5): it holds at 1.0 for the hybrid and the microservice, declines to ~ 0.60 for the monolith, and collapses to ~ 0.33 for the serverless. The classical-metric ranking does not predict the ODTE ranking. The serverless shows the lowest peak CPU yet collapses hardest, despite serving the lowest off-path request rate (Table 3), while the hybrid holds at a rate close to the one at which the monolith fails. None of these metrics separates performance degradation from entanglement (and thus correctness) loss.

**Fig. 5.** E2: CPU utilization (left axis), ODTE with $T_d = 50$ ms (inner right axis), and autoscaled-tier replica count (outer right axis) over the load ramp, per pattern

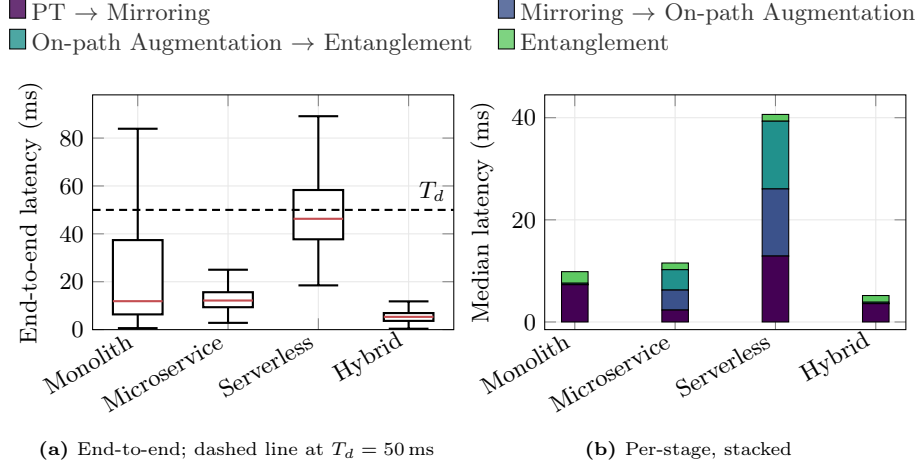


Fig. 6. E2: On-path latency per pattern

Two Failures, Two Mechanisms. The monolith loses *both* timeliness and reliability: T falls to 0.679, R drops to 0.800. This is the only pattern whose R is not constant at 1. Lacking twinning isolation, it can only scale by whole-application replication, so off-path CPU contention bleeds into the on-path chain. The loss sits entirely at ingress: across the ten repetitions, 6.8% of emitted states never reach the chain, while every state that does is entangled and persisted. The drops are scattered single states, not bursts. The publisher and the broker can be ruled out: the same broker delivers every state to the mirroring microservice ($R = 1$). The loss therefore lies in the last leg, broker-to-subscriber delivery to replicas saturated by the off-path work they host. The serverless loses only timeliness: R stays at 1 throughout, but its on-path latency already sits at a 41 ms median at rest, leaving almost no margin against the 50 ms deadline, so any added latency tips a large fraction of states past T_d . The microservice and the hybrid hold entanglement throughout. Fig. 6a shows the resulting end-to-end distributions: the serverless box straddles the deadline, the monolith grows a long contention tail (upper quartile 37 ms), and the microservice and the hybrid stay well below it. Fig. 6b shows where on-path latency concentrates: ingress and entanglement for the monolith, ingress and the two relayed calls for the serverless. However, these values hide the ramp. Specifically, over the ramp the monolith ingress climbs from ~ 4 to ~ 22 ms and entanglement from ~ 1.6 to ~ 6.5 ms as off-path contention loads the pod, the serverless relayed calls climb from ~ 13 to ~ 17 – 18 ms, and the microservice and hybrid stay flat.

The deadline separates the two mechanisms. Loosening it to $T_d = 100$ ms restores the serverless to a minimum ODTE of 0.97 (its states all arrive, only late) but lifts the monolith only to 0.77, since no deadline recovers dropped states ($R = 0.800$). Tightening T_d to 25 ms breaks the serverless outright (ODTE goes to 0, already failing at rest) and costs the microservice a dip to 0.93; the hy-

brid holds at 1 at all three deadlines. The serverless *has* twinning isolation in the structural sense: it pins on-path and scales only off-path, exactly as the microservice does. Yet it collapses. This is the saturation of Design Principle 1 observed empirically: subdividing the twinning process beyond a single deployable unit adds no isolation (off-path scale-out drops no states from the pinned on-path chain, and R holds at 1), while the multiplied on-path components carry the invocation cost of Design Principle 2, which the deadline then reclaims. The results are consistent with isolation being required but not sufficient.

Takeaway. In E2, entanglement holds for the two patterns that isolate the twinning chain *and* have on-path latency margin (hybrid, microservice), while it collapses for the two that lack one of the two conditions (the monolith through whole-application contention, the serverless through exhausted latency margin). The classical metrics do not separate these two outcomes.

6 Related Work

Platforms and Middleware. Ciavotta et al. [14] proposed a microservice-based manufacturing platform whose backend services support DTs throughout their life cycle. Borodulin et al. [12] envisioned a cloud platform that executes DTs as containerized microservices. Atalay [5] designed a smart-grid management framework whose application layer hosts DTs as microservices. Alaasam et al. [4] argued that microservices suit the twinning of event-driven systems, focusing on the middleware that dispatches events rather than on the DT itself. Shahsavari et al. [26] extended an open-source framework for DT-PT data exchange [3], improving its speed and scalability. In these works the microservice pattern is a given, and scalability, where measured, is a property of the platform or middleware; the architecture of the DT itself is not the variable under study.

Digital Twin Architectures. Aziz et al. [6] composed purpose-oriented DTs from individual DTs, each deployed as a single unit—a monolith, in the terms of Section 3. Gao et al. [16] built Plotinus, a microservice-based DT for space-air-ground integrated networks in which each plugin is a distinct microservice. Both works commit to one pattern and leave its scalability unexamined. Bellavista et al. [8] implemented a generic DT model as microservice and serverless variants and evaluated their entanglement across the cloud-to-edge continuum. Their independent variable, however, is placement, not load.

Pattern Comparisons. Outside the DT domain, Blinowski et al. [10] compared monolithic and microservice variants of the same application for performance and scalability, and Adzic and Chatley [1] examined the architectural and economic impact of moving services to serverless platforms. These comparisons rank patterns on throughput, latency, and cost; correctness is not at stake, since the applications compared have no physical counterpart to stay entangled with.

7 Conclusion

In this work, we treated DT scalability as a correctness concern: a DT that falls behind its PT does not merely slow down, it stops working correctly. The four patterns studied give a two-part answer. The architecture must isolate the twinning process, so that load on the rest of the DT is absorbed without touching it. And the twinning process must keep latency margin against the entanglement deadline, since load consumes exactly that margin. Neither half suffices alone. The monolith had margin but no isolation: off-path contention reached the twinning process, and physical states were lost. The serverless had isolation but no margin: its framework-mediated calls had spent most of the deadline before any load arrived. Entanglement survived only where the two met, as in the microservice and hybrid. CPU utilization and replica count reported four loaded systems scaling as designed, showing that such metrics fail to capture DT entanglement.

The natural next step is to grow the load this study held constant: the twinning load itself. Here the PT emitted at a fixed rate and the twinning process did fixed work per state, so the on-path chain never had to replicate. Both can change: on the physical side, a PT may be added, reconfigured, or updated, bringing more states or richer ones; on the digital side, the twinning process may take on heavier work per state, such as a more expensive augmentation. Both kinds of growth are often more predictable than the off-path load studied here, since they follow the PT and DT specifications. A second step follows from the classical metrics failing to capture DT entanglement: if no traditional metric reliably announces its loss, the autoscaler should watch entanglement itself. Whether scaling on ODTE, or on some combination of factors, is a good replacement for CPU utilization remains to be studied, and is a promising path forward for DTs that scale to stay entangled.

References

1. Adzic, G., Chatley, R.: Serverless computing: economic and architectural impact. In: ESEC/FSE 2017. pp. 884–889 (2017)
2. Ahelerooff, S., Xu, X., Zhong, R.Y., Lu, Y.: Digital twin as a service (DTaaS) in industry 4.0: An architecture reference model. *Adv. Eng. Inform.* **47**, 101225 (2021)
3. Aho, P., Immonen, E.: ModelConductor: An on-line data management architecture for digital twins. In: ETFA 2020. pp. 1397–1400 (2020)
4. Alaasam, A.B.A., Radchenko, G., Tchernykh, A.: Stateful stream processing for digital twins: Microservice-based Kafka Stream DSL. In: SIBIRCON 2019. pp. 804–809 (2019)
5. Atalay, M.: A service-oriented digital twins framework for smart grid management. In: SRMC 2022. pp. 9–17 (2022)
6. Aziz, A., Chouhan, S.S., Schelén, O., Bodin, U.: Distributed digital twins as proxies-unlocking composability and flexibility for purpose-oriented digital twins. *IEEE Access* **11**, 137577–137593 (2023)
7. Barricelli, B.R., Casiraghi, E., Fogli, D.: A survey on digital twin: Definitions, characteristics, applications, and design implications. *IEEE Access* **7**, 167653–167671 (2019)

8. Bellavista, P., Bicocchi, N., Fogli, M., et al.: Exploiting microservices and serverless for digital twins in the cloud-to-edge continuum. *Future Gener. Comput. Syst.* **157**, 275–287 (2024)
9. Bellavista, P., Bicocchi, N., Fogli, M., et al.: ODTE: A metric for digital twin entanglement. *IEEE Open J. Commun. Soc.* **5**, 2377–2390 (2024)
10. Blinowski, G., Ojdowska, A., Przybyłek, A.: Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access* **10**, 20357–20374 (2022)
11. Bondi, A.B.: Characteristics of scalability and their impact on performance. In: *WOSP 2000*. pp. 195–203 (2000)
12. Borodulin, K., Radchenko, G., Shestakov, A., et al.: Towards digital twins cloud platform: Microservices and computational workflows to rule a smart factory. In: *UCC 2017*. pp. 209–210 (2017)
13. Castro, P., Ishakian, V., Muthusamy, V., Slominski, A.: The rise of serverless computing. *Commun. ACM* **62**(12), 44–54 (2019)
14. Ciavotta, M., Alge, M., Menato, S., et al.: A microservice-based middleware for the digital factory. *Procedia Manuf.* **11**, 931–938 (2017)
15. Dragoni, N., Giallorenzo, S., Lafuente, A.L., et al.: Microservices: Yesterday, today, and tomorrow. In: *Present and Ulterior Software Engineering*, pp. 195–216. Springer (2017)
16. Gao, Y., Qiu, K., Chen, Z., et al.: Plotinus: A satellite internet digital twin system. *J. Commun. Inf. Netw.* **9**(1), 24–33 (2024)
17. Grieves, M.: Completing the cycle: Using PLM information in the sales and service functions [slides]. In: *SME Management Forum* (2002)
18. Grieves, M.W.: Digital twins: Past, present, and future. In: *The Digital Twin*, vol. 1, pp. 97–121. Springer (2023)
19. Heluany, J.B., Gkioulos, V.: Survey on digital twins: from concepts to applications. In: *ARES 2023* (2023)
20. Kritzinger, W., Karner, M., Traar, G., et al.: Digital twin in manufacturing: A categorical literature review and classification. *IFAC-PapersOnLine* **51**(11), 1016–1022 (2018)
21. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* **21**(7), 558–565 (1978)
22. Lewis, J., Fowler, M.: Microservices (2014), <https://martinfowler.com/articles/microservices.html>
23. Lim, K.Y.H., Zheng, P., Chen, C.H.: A state-of-the-art survey of digital twin: techniques, engineering product lifecycle management and business innovation perspectives. *J. Intell. Manuf.* **31**(6), 1313–1337 (2020)
24. Minerva, R., Lee, G.M., Crespi, N.: Digital twin in the IoT context: A survey on technical features, scenarios, and architectural models. *Proc. IEEE* **108**(10), 1785–1824 (2020)
25. Schleier-Smith, J., Sreekanti, V., Khandelwal, A., et al.: What serverless computing is and should become: the next phase of cloud computing. *Commun. ACM* **64**(5), 76–84 (2021)
26. Shahsavari, S., Immonen, E., Rabah, M., et al.: MCX – an open-source framework for digital twins. In: *ECMS 2021*. vol. 35, pp. 119–124 (2021)
27. Stankovic, J.: Misconceptions about real-time computing: a serious problem for next-generation systems. *Computer* **21**(10), 10–19 (1988)
28. Wu, Y., Zhang, K., Zhang, Y.: Digital twin networks: A survey. *IEEE Internet Things J.* **8**(18), 13789–13804 (2021)