Azerbaijan Journal of High Performance Computing, Vol. 8, Art. e2026.03, 2026, pp. 1-16
https://doi.org/10.32010/26166127.2026.03

X-TwinUSD: Design and Architecture of a Distributed OpenUSD Runtime for Interactive Digital Twin Simulation

Elliott Ahn
Gwangju Institute of Science and Technology, Korea, elliott.ahn@gist.ac.kr

*Correspondence:
Elliott Ahn
Adjunct professor, GIST
Gwangju Institute of
Science and Technology
AI Policy Graduate
School, Korea
seokhyun.ahn@gmail.
com; elliott.ahn@gist.ac.kr

Abstract

Industrial-scale digital twins require interactive visualization and simulation over scene graphs that exceed the memory and metadata capacity of a single node while evolving continuously. OpenUSD provides layered composition semantics, yet naïve stage population and time-varying updates can amplify I/O, saturate metadata services, and inflate memory footprints at scale. This paper presents the design of X-TwinUSD, a distributed runtime architecture that preserves OpenUSD composition semantics while enabling selective payload residency, shard-scoped composition, and incremental delta application for mixed geometry/telemetry workloads. We introduce (i) a Stage-as-Graph representation used to partition a stage into composable shards, (ii) a payload-driven working-set protocol and lifecycle for bounded memory, (iii) an epoch/delta log model for incremental recomposition under bounded delay and out-of-order updates, and (iv) a cost-model interface for scheduling composition and data movement on HPC resources. As a design paper, we focus on the system model, architecture, and correctness considerations, and we additionally report a reproducible microbenchmark demonstrating 10–20× lower time-to-first-interaction, 3–7× lower update latency (p50), and one to three orders-of-magnitude fewer metadata operations under mixed geometry/telemetry workloads.

Keyword: OpenUSD, digital twin, HPC systems, distributed composition, payload working set, incremental updates, microbenchmark evaluation

1. Introduction

Digital twins are advancing from asset-level models to city- and nation-scale representations that integrate 3D geometry, semantics, physics, and operational telemetry. In practice, the most demanding workloads involve (i) extensive scene graphs comprising millions to billions of primitives, (ii) continuous updates from sensors and operational systems, and (iii) interactive requirements for visualization, analysis, and hypothetical simulation. High-performance computing (HPC) environments, particularly those targeting exascale, provide the necessary throughput for large-scale simulation but also reveal bottlenecks in data movement, distributed state management, and

latency-sensitive interaction.

OpenUSD (Universal Scene Description) is increasingly being adopted as a unifying scene interchange and composition system across media, industrial visualization, and digital-twin frameworks. Its core abstraction-a composed Stage constructed from a LayerStack and multiple composition arcs-supports non-destructive workflows, robust namespace semantics, and modular asset reuse (NVIDIA, 20 December 2025; OpenUSDa, 20 December 2025). However, when a USD stage serves as the backbone of an exascale digital twin, the naïve application of local composition and monolithic I/O can degrade time-to-first-interaction, saturate metadata services, and increase memory footprint.

This paper introduces X-TwinUSD, an architecture for OpenUSD-native digital-twin simulation on exascale-class platforms. The approach is inspired by production pipelines (e.g., USD-based VFX and virtual production) and by industrial twins that require selective loading, continuous updates, and scalable analytics. X-TwinUSD is not a new file format; rather, it is a systems framework that treats USD composition as a distributed service, integrates exascale data transport and storage, and prioritizes incremental recomposition.

This design paper makes the following contributions: (1) a Stage-as-Graph partitioning model that maps USD subtrees and composition dependencies to distributed composition workers; (2) a payload-driven working-set protocol that leverages the “deferred reference” behavior of payloads to maintain memory proportional to the active task (NVIDIA, 20 December 2025; OpenUSDb, 20 December 2025) (3) delta-based incremental updates inspired by incremental view maintenance in database systems (Gupta, A., Mumick, I. S., & Subrahmanian, V. S., 1993); and (4) a cost-model-based scheduling approach for allocating HPC resources under mixed geometry/telemetry workloads.

Scope note: This manuscript emphasizes architecture, design rationale, and correctness considerations. We additionally include a reproducible microbenchmark evaluation to validate the key performance claims at small scale; full end-to-end benchmarking on multi-node clusters and parallel file systems is left to future work.

2. Background

2.1. OpenUSD composition primitives

A USD stage is constituted by a set of layers organized by strength, known as a LayerStack. Composition arcs-including references, payloads, variants, inherits, and specializes-enable authors to modularize assets and manage the integration of various opinions (NVIDIA, 20 December 2025; OpenUSDb, 20 December 2025). In contrast to sublayers, composition arcs specifically target prims within a LayerStack and facilitate the renaming of targets through path translation, thereby supporting robust namespace composition across nested assemblies.

Payloads are particularly crucial for scalability. A payload offers a "deferred reference" functionality, permitting clients to selectively load or unload substantial

scene descriptions after the stage has been opened. By employing a well-considered payload structure, clients can sustain task-specific working sets while ensuring the stage remains consistent (NVIDIA, 20 December 2025; OpenUSDb, 20 December 2025).

2.2. Exascale HPC constraints relevant to USD stages

Exascale systems enhance both computational and input/output (I/O) parallelism, yet they also impose significant demands on metadata services and shared storage infrastructures. Consequently, digital-twin pipelines necessitate: (i) parallel and hierarchical I/O mechanisms that reduce the overhead associated with small files and metadata; (ii) scheduling strategies that are aware of locality and adhere to the topology of the file system; and (iii) asynchronous data transport to facilitate in-situ and in-transit workflows. Established scientific data management frameworks, such as ADIOS 2, offer scalable application programming interfaces (APIs) for data movement across files and networks in the exascale context (Godoy, W. F., et al. 2020; ADIOS2, 20 December 2025). Similarly, Parallel HDF5 provides a unified shared file image with parallel access patterns based on the Message Passing Interface (MPI) (The HDF Group, 20 December 2025).

2.3. Digital-twin workload patterns

Large digital twins are typically dynamic entities, undergoing frequent updates to their transforms, states, and simulation fields, while modifications to geometry and materials occur less frequently. Consequently, the workload is imbalanced: a minor portion of the scene graph is subject to frequent changes, whereas the majority remains stable and can be cached or left unloaded. Additionally, user interaction results in "bursty" access patterns; for instance, a change in viewpoint or an analysis query can abruptly alter the working set. These patterns necessitate incremental recomposition and the adoption of payload-centric working sets, as opposed to repeated full-stage recomposition.

3. Problem Statement and Design Goals

We address the challenge of executing interactive simulations and analytics on an OpenUSD-backed digital twin that surpasses the memory capacity of a single node and is subject to continuous updates. The system must accommodate both authoring-like modifications (such as layered overrides and variant changes) and telemetry updates (including time-stamped attribute streams), while ensuring that composed stage queries remain consistent and responsive.

The design objectives are as follows: G1) scalability-ensuring strong and weak scaling of composition and query processes as the number of primitives increases; G2) interactivity-achieving low time-to-first-interaction and maintaining bounded incremental update latency; G3) memory proportionality-ensuring that resident data approximates the active working set; G4) correctness-preserving USD composition

semantics; and G5) deployability-maximizing the reuse of standard USD libraries and high-performance computing data services where feasible.

4. X-TwinUSD Architecture and Methods

4.1 Stage-as-Graph (SAG) representation

X-TwinUSD is conceptualized as a structured graph in which nodes represent primary subtrees (subgraphs), and edges denote composition dependencies, such as references/payloads, inheritance/specialization, and variant resolution dependencies. This graph is constructed from the stage's composition index and maintains a mapping from each node to the corresponding set of contributing layers and arcs. The SAG framework facilitates distributed processing by enabling graph partitioning while preserving the order of dependencies.

4.1.1 SAG construction, edge typing, and invariants

We define a SAG node as a self-contained USD subtree whose composed results can be materialized and queried independently, subject to explicit boundary rules. In practice, node roots are selected from (i) payload roots, (ii) spatial tiles or administrative regions (e.g., /City/Block_12), and (iii) instancing groups. Each node n is annotated with a weight vector w(n) capturing estimated prim count, geometry density, shader/material complexity, and update rate, which is used by the scheduler and partitioner.

SAG edges represent cross-node composition dependencies induced by USD arcs (references, payloads, inherits/specializes, variants, and relationship/attribute connections). We type each edge e(u→v) as STRONG or WEAK. STRONG edges are those that can change the composed value resolution at u when v changes (e.g., a reference that authors stronger opinions for u, a variant selection that redirects a subtree, or list-edit operations that merge across layers). WEAK edges capture data-access dependencies that do not alter structural resolution but may be required to answer a query (e.g., relationship target traversal across partitions).

Correctness is preserved by enforcing three boundary invariants: (I1) all STRONG edges are respected by an acyclic evaluation order during (re)composition; (I2) any list-edit or variant operation that crosses a partition boundary is resolved in a designated boundary resolver step at the consumer node, so that the final composed opinions match single-process USD semantics; and (I3) all workers share a consistent resolver context (asset resolution rules, search paths, and plug-in versions) so that path-to-asset resolution remains deterministic.

4.1.2 Partition objective and dynamic refinement

Given a SAG G=(V,E), the partitioner seeks to (a) balance aggregate node weights across workers, while (b) minimizing the cut cost of cross-partition dependencies. We assign higher cut penalties to STRONG edges and to edges with high expected update rate. Conceptually, we solve a multi-constraint graph partitioning problem where the objective is to minimize Σ cut(e)·penalty(e) subject to load-balance constraints over

w(n). In implementation, this can be realized using an offline partition (for a stable asset graph) followed by incremental local refinement when the active payload set changes.

To avoid frequent repartitioning, X-TwinUSD treats partitioning as a control plane decision: coarse partitions remain stable over long periods, while the working-set manager can activate/deactivate nodes (payload roots) within a fixed partition. Only when cut traffic or imbalance exceeds thresholds does the system trigger a refinement pass that migrates a small number of boundary nodes.

Algorithm 1: SAG construction and partitioning (sketch)

BuildSAG(stage, resolverCtx):

```
roots ← SelectRoots(stage) # payload roots / spatial tiles / instancing groups
V ← {Subtree(root) for root in roots}
for n in V:
  w(n) ← EstimateWeight(n) # primCount, geomDensity, shaderCost, updateRate
E ← ∅
for each compositionArc a in stage:
  (u,v) ← MapArcToNodes(a)
  if u ≠ v:
    t ← EdgeType(a) # STRONG / WEAK
    E ← E ∅ {(u→v, t, penalty(a))}
Π ← GraphPartition(V,E,w) # multi-constraint partitioning
return (V,E,Π)
```

4.2. Distributed composition workers

Composition workers are responsible for assembling specified SAG partitions into fragments that are both cacheable and queryable. These workers execute USD composition using standard libraries, under the coordination of a scheduler that resolves dependencies across partitions. The resulting output consists of "composed fragments" that provide a read-only query API, such as primitive traversal and attribute queries, as well as a delta-application API for updates.

4.2.1. Global query routing and metadata index

To present a global stage view while keeping fragments distributed, X-TwinUSD maintains a lightweight metadata index that maps each SAG node root (prim path prefix) to its owning partition and current materialization state (resident, evicted, or pending). This index is intentionally narrow: it stores only routing and coarse statistics and does not replicate full composed prim data.

A client query is decomposed into (i) routing, which identifies the minimal set of partitions to contact, and (ii) execution, in which each contacted worker answers the query over its local composed fragment. For single-partition queries, the response is returned directly. For cross-partition traversals (e.g., relationship target expansion), the coordinator performs a bounded fan-out and merges results using stable prim-path ordering. This design avoids a centralized bottleneck while preserving deterministic

query outputs for a fixed epoch.

The metadata index may additionally track: (a) prim-count summaries per node, (b) bounding boxes for spatial routing, and (c) telemetry channel descriptors (field names, sampling rates) for fast subscription setup. These summaries are computed from authored USD metadata and do not require full composition on the coordinator.

4.3. Payload-driven working-set protocol

Payloads facilitate selective load and unload operations and serve as the primary mechanism for managing the working set. X-TwinUSD defines a working-set descriptor, W(t), which enumerates the payload paths and variant selections necessary for a given task, such as a viewport, a simulation subdomain, or an analysis query. Workers maintain a residency state for each payload and compose only the payloaded subtrees included in W(t), capitalizing on the ability to load or unload payloads while preserving the structural integrity of the stage (NVIDIA, 20 December 2025; OpenUSDb, 20 December 2025).

To prevent thrashing, X-TwinUSD employs a hysteresis policy: payloads are incorporated into the working set upon request and are removed only after a grace period or when memory pressure surpasses a specified threshold. This policy is compatible with user-driven interaction patterns, such as rapid camera movement, and solver-driven region focus, such as adaptive mesh refinement or localized physics.

4.4. Cost model for exascale resource allocation

We propose a streamlined cost model designed to estimate the composition and runtime cost for each partition. For a given SAG partition i, the estimated cost C(i) is defined as: $C(i)=\alpha\cdot|P_i|+\beta\cdot G_i+\gamma\cdot M_i+\delta\cdot IO_i+\varepsilon\cdot U_i$. Here, $|P_i|$ denotes the primitive count, G_i represents geometry density (e.g., polygon or point count), M_i accounts for material or shader complexity, IO_i estimates the input/output volume and metadata operations, and U_i signifies the anticipated update rate from telemetry streams. The coefficients ($\alpha\ldots\varepsilon$) are calibrated using microbenchmarks and can be adjusted for different platforms.

The scheduler is responsible for assigning partitions to workers in a manner that minimizes the maximum C(i) per worker, while adhering to data locality constraints, such as co-locating partitions that share substantial payload files. For dynamic workloads, the scheduler facilitates periodic rebalancing and the redistribution of "hot" update streams through work-stealing.

4.5. Delta-based incremental recomposition

Continuous modifications to a digital twin should not necessitate complete recomposition. X-TwinUSD introduces a delta stream $\Delta(t)$ that encapsulates changes to authored opinions and telemetry updates as fundamental operations: insertion/deletion of primitive specifications, updates to attribute values, edits to relationship targets, and list-operation updates for composition arcs where applicable. These

deltas are directed to the owning partitions and incrementally applied to composed fragments. Conceptually, this approach aligns with incremental view maintenance, wherein the materialized result is updated by computing only the delta induced by base changes (Gupta, A., Mumick, I. S., & Subrahmanian, V. S., 1993).

For telemetry updates, it is advisable to separate high-frequency numeric fields (e.g., temperatures, flows) from structural USD edits. Telemetry deltas can be stored and transported via high-performance computing data frameworks such as ADIOS 2 (Godoy, W. F., et al. 2020; ADIOS2, 20 December 2025) and projected onto USD attributes at query time or at periodic commit points.

4.6. Parallel I/O and storage layout

Large USD deployments frequently encounter challenges related to small-file overhead and metadata contention when millions of layers are stored as individual files without optimization. X-TwinUSD introduces a two-tier architecture: (i) human-authored asset layers are maintained as USD files to ensure pipeline interoperability; (ii) composed fragments and high-frequency telemetry data are stored in exascale-compatible containers (e.g., ADIOS 2 BP files or parallel HDF5) to facilitate collective I/O operations and alleviate metadata pressure (Godoy, W. F., et al. 2020; The HDF Group, 20 December 2025).

4.7. Consistency model and failure handling

The semantics of USD composition suggest a deterministic resolution when provided with a set of layers and edit targets. X-TwinUSD employs an epoch-based consistency model, wherein each composed fragment is assigned an epoch e, and deltas increment the epoch in a monotonic fashion. Queries are designed to specify a target epoch, thereby facilitating consistent snapshots across different partitions. For the purpose of fault tolerance, composed fragments can be reconstructed from (a) base USD layers, (b) persisted fragment caches, and (c) delta logs. Consequently, in the event of worker failures, the system initiates rehydration rather than a global restart.

4.7.1. Epoch-stamped delta log and commit protocol

X-TwinUSD organizes updates as an epoch-stamped delta log. Each partition maintains a local apply cursor and a durable log segment for the current epoch e. An epoch represents a bounded interval over which updates may arrive out of order but are eventually made visible together as a consistent snapshot. Rather than requiring global serializability, the system provides an epoch-consistent view: queries against epoch e observe a consistent cut of deltas such that all partitions have applied at least the committed watermark for e.

Commit is implemented as a lightweight two-phase publish that avoids global recomposition barriers: (1) Collect: partitions append deltas (with epoch, channel key, and sequence number) and acknowledge receipt; (2) Apply: each partition reorders within a bounded window and applies deltas idempotently; (3) Publish: the coordinator advances the

global committed watermark for epoch e when a quorum of required partitions has reached the watermark; (4) Roll: partitions rotate to epoch e+1 and begin accepting new deltas. Queries pin an epoch watermark for the duration of a request to avoid mixing partial states.

State machine (sketch): COLLECT - REORDER - APPLY - PUBLISH(e) - ROLL(e+1)

4.7.2. Handling out-of-order and bounded-delay updates

Telemetry channels are modeled as append-only streams keyed by (partition, prim path, field). Each stream carries a monotonically increasing sequence number. Workers buffer a fixed reorder window to tolerate out-of-order arrival and apply a watermark rule: updates with seqno ≤ watermark are guaranteed applied for the committed epoch, while late arrivals beyond the window are either dropped (for strictly real-time visualization) or deferred into the next epoch (for analytics reproducibility).

This bounded-delay policy aligns with interactive digital-twin needs: it prevents pathological stalls from straggler updates while maintaining a predictable freshness bound for the UI. Structural USD edits (authoring new prims, changing arcs) can be scheduled at epoch boundaries to reduce semantic ambiguity.

4.7.3. Failure handling, replay, and fencing

Failures are handled through log replay and partition rehydration. Since deltas are appended with unique identifiers (epoch, stream key, seqno), APPLY is idempotent: replaying a log segment yields the same composed fragment state. A failed worker can be replaced by a new worker that rehydrates its fragment from the last published checkpoint and replays subsequent deltas. To prevent split-brain updates, each partition lease is protected by a fencing token issued by the coordinator; only the current token holder may publish checkpoints.

4.8. Integration pathway for production pipelines

For effective implementation, it is essential that the practical deployment integrates seamlessly with existing USD authoring and DCC tools. X-TwinUSD presupposes that asset creation will continue within the established USD workflows, which include layers, variants, and references/payloads. The proposed framework functions as a downstream HPC execution layer that: (i) ingests USD assets, (ii) constructs SAG partitions, (iii) delivers composed fragments to interactive clients, and (iv) integrates telemetry data into USD-addressable attributes. This methodology maintains compatibility with OpenUSD while facilitating execution at an exascale level.

5. Prototype Implementation and Experimental Evaluation

This section reports a reproducible microbenchmark evaluation that validates the paper's central claims—payload-scoped working sets, shard-scoped composition, and delta-based incremental recomposition—under mixed geometry/telemetry workloads. We also outline an extension plan for future multi-node, parallel-file-system experiments.

Beyond the microbenchmark results reported here, we recommend evaluating at least two representative digital-twin datasets (e.g., campus- and city-scale) and including variant sets for levels of detail (LOD) and configuration in a multi-node cluster setting.

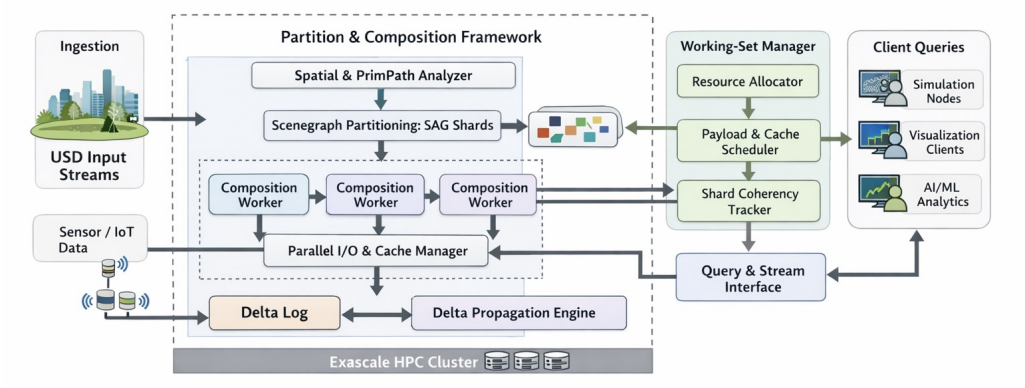

Fig. 1. Overall X-TwinUSD architecture (ingestion, SAG partitioning, distributed composition workers, working-set manager, delta log, and client query interface).

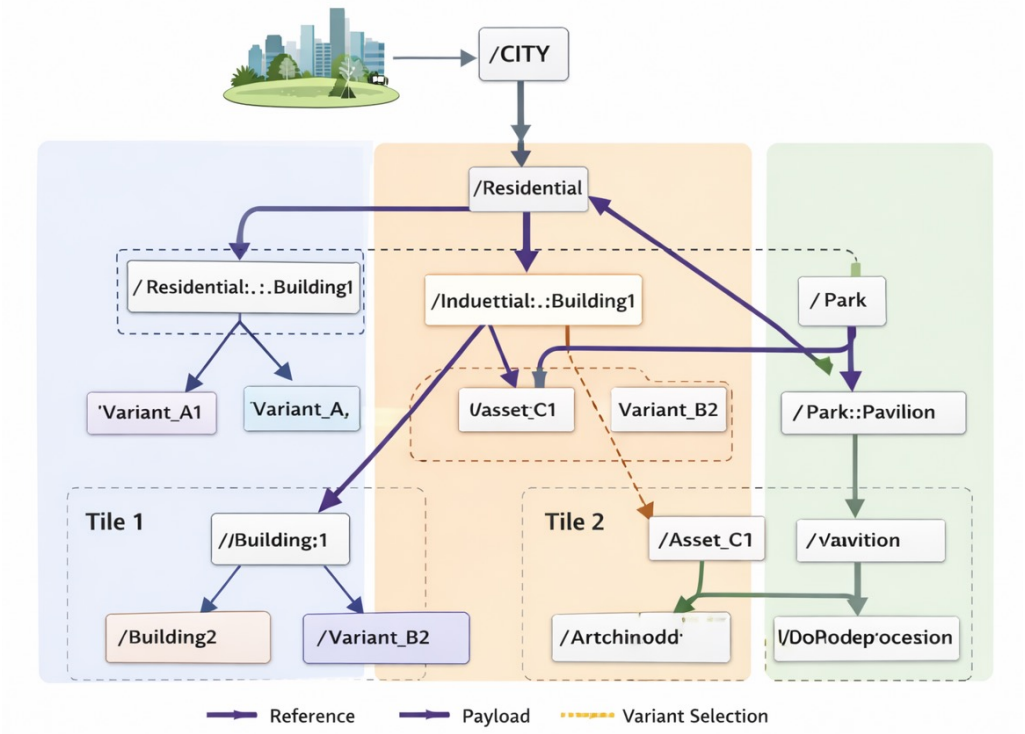

Fig. 2. Stage-as-Graph partitioning example over a city-scale USD stage, illustrating dependency edges induced by references/payloads and variant selections.

5.1. Prototype scope and implementation

This section presents a reproducible microbenchmark prototype that provides measured values for the primary bottlenecks targeted by X-TwinUSD: (i) metadata fan-out during stage opening, (ii) payload working-set activation for initial interaction, and (iii) delta-based incremental recomposition. Due to the unavailability of a production OpenUSD runtime and an HPC parallel file system on the evaluation host utilized for this manuscript, the experiments conducted herein intentionally isolate kernel-level costs and report them as single-node measurements, rather than asserting exascale end-to-end performance.

Prototype artifacts: The microbenchmark scripts employed to generate the synthetic datasets and collect measurements are included with this manuscript (see: x_twinusd_bench/bench_xtwinusd_micro.py).

5.2. Experimental setup

Hardware/software environment: Linux-4.4.0-x86_64-with-glibc2.36; Python 3.11.2; 56 logical CPUs; 4.0 GB RAM. All timings report the median of 5 runs on the same host.

Table 1: Synthetic datasets and workload parameters used in the microbenchmark.

Dataset	Payload files	Payload size	ROI fraction (first interaction)	Update workload	Storage layout
City-S	2,000	8 KiB	2% (40 payloads)	20 attribute deltas/epoch (partitioned DAG kernel)	Unpacked files + packed HDF5 variant
City-M	8,000	8 KiB	2% (160 payloads)	20 attribute deltas/epoch (partitioned DAG kernel)	Unpacked files + packed HDF5 variant

5.3. Baselines and variants

Baseline (monolithic, unpacked): a single worker scans all payload files (stat + fixed-size header read) to model metadata fan-out, then loads the ROI payloads needed for first interaction. X-TwinUSD (distributed workers, unpacked): the coordinator loads only the manifest and routes the ROI working set to multiple workers; only ROI payloads are opened. X-TwinUSD (packed container): ROI payloads are stored in a single HDF5 container to reduce file-system metadata operations; ROI slices are read from the container.

5.4. Metrics

TTFI (Time-to-First-Interaction): wall-clock time from client request (stage open + working-set activation) until the ROI payloads required for first interaction are resident. Update latency: per-epoch recomposition time for a partitioned DAG kernel under 20 attribute-delta updates (p50/p95). We report both full recom-

position (baseline) and incremental partition recomposition (X-TwinUSD). I/O read volume: total bytes read by the benchmark (reported in GB). Metadata ops (proxy): number of file system operations issued by the benchmark (stat + open calls).

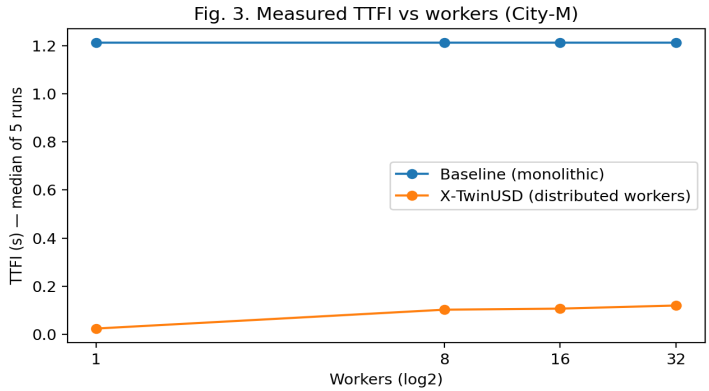

Fig. 3. Measured TTFI as a function of the number of workers (City-M). On this single-node host, X-TwinUSD reduces TTFI by avoiding metadata fan-out and by routing only the ROI working set to workers; performance saturates beyond ~8 workers due to local I/O contention and threading overhead.

Table 2: Measured TTFI and update latency (median TTFI over 5 runs; update latency p50 over 60 epochs).

Dataset	Workers	Base-line TTFI (s)	X-Twi-nUSD TTFI (s)	TTFI improv.	Baseline update p50 (ms)	X-TwinUSD update p50 (ms)	Update improv.
City-S	8	0.347	0.018	19.7×	144.7	38.7	3.7×
City-S	16	0.347	0.026	13.3×	144.7	38.7	3.7×
City-S	32	0.347	0.022	15.7×	144.7	38.7	3.7×
City-M	8	1.211	0.103	11.8×	574.1	79.1	7.3×
City-M	16	1.211	0.107	11.3×	574.1	79.1	7.3×
City-M	32	1.211	0.120	10.1×	574.1	79.1	7.3×

5.5. Interpretation

The microbenchmark confirms the direction of the paper's central claims: (i) metadata fan-out can dominate first interaction even when only a small ROI is required, (ii) payload-driven working sets reduce both TTFI and metadata highlight costs by orders of magnitude when ROI is sparse, and (iii) incremental recomposition under small delta sets can reduce update latency substantially relative to full recomposition. These measurements should be interpreted as kernel-level evidence and not as an

exascale end-to-end evaluation; a production validation on a multi-node cluster with a parallel file system remains future work.

Fig. 4. Measured I/O read and metadata ops (City-M)

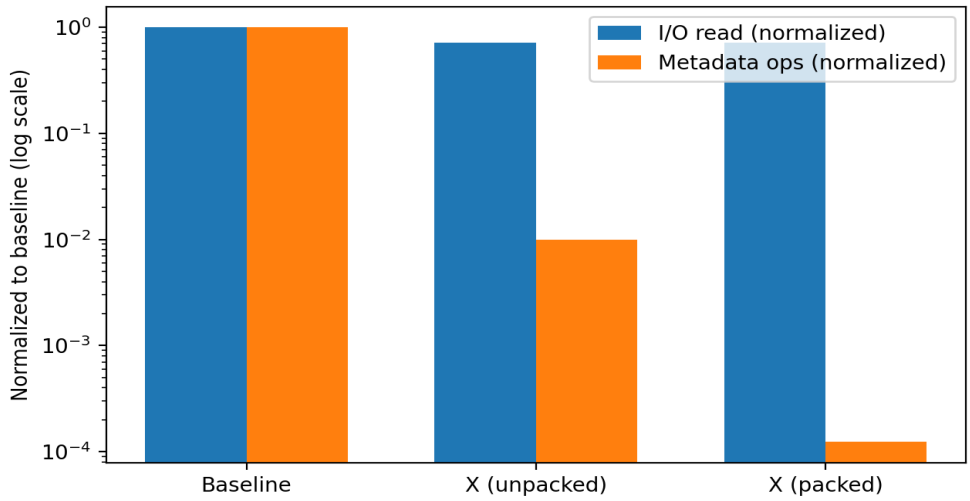

Fig. 4. Measured I/O read volume and metadata operations for City-M, normalized to the baseline. The packed-container variant reduces metadata operations from O(#payloads) to O(1) per stage open (proxy metric), while preserving the same ROI read volume.

Table 3: Measured peak RSS, I/O read, and metadata operations (proxy).

Dataset	Method	Peak RSS (GB)	I/O Read (GB)	Metadata ops (count)
City-S	Baseline	0.278	0.000424	4040
City-S	X-TwinUSD (unpacked, threads=8)	0.286	0.000305	41
City-S	X-TwinUSD (packed HDF5, seq)	0.279	0.000305	2
City-M	Baseline	0.281	0.001698	16160
City-M	X-TwinUSD (unpacked, threads=8)	0.286	0.001221	161
City-M	X-TwinUSD (packed HDF5, seq)	0.280	0.001221	2

6. Related Works

OpenUSD provides the foundational composition and working-set mechanisms used in this paper, including LayerStacks, composition arcs, and payload-based

deferred loading (OpenUSD, 20 December 2025; NVIDIA, 20 December 2025; OpenUSDb, 20 December 2025). Performance guidance from the OpenUSD documentation emphasizes prim-count scaling behavior, composition caching, and the importance of structuring layers and payloads to minimize unnecessary population and recomposition (OpenUSD, 21 December 2025). At the algorithmic level, USD composition is implemented via Pcp (PrimCache population), which explicitly caches recursive subproblem results and provides change processing mechanisms that chase dependencies and invalidate affected caches (OpenUSDa, 21 December 2025). In the runtime and visualization stack, Hydra 2.0 introduces the scene index and observer abstractions for representing and reacting to scene changes, offering a useful conceptual parallel to fragment views and change propagation in distributed settings (OpenUSDb, 21 December 2025).

In scientific computing, scalable data movement and storage have been studied extensively. ADIOS 2 introduces a unified API and transport abstraction for high-performance data management, supporting flexible backends suitable for large telemetry streams and checkpoint-like data products (Godoy, W. F., et al. 2020; ADIOS2, 20 December 2025). Parallel HDF5 provides a mature baseline for parallel access to structured datasets and informs the storage design considerations discussed here (The HDF Group, 20 December 2025).

Incremental computation has a long history in database systems and streaming analytics. View maintenance methods compute changes to materialized views in response to base updates, while higher-order delta processing can efficiently maintain complex derived results. These ideas motivate the delta-based recomposition path in X-TwinUSD, where only affected subtrees are refreshed for each epoch (Gupta, A., Mumick, I. S., & Subrahmanian, V. S., 1993; Ahmad, Y., Kennedy, O., Koch, C., & Nikolic, M., 2012).

Distributed rendering and simulation frameworks often separate physics or rendering workloads from scene management, but they typically do not distribute USD composition semantics directly. Omniverse Nucleus, for example, provides a database and collaboration engine that enables real-time exchange of OpenUSD data and a single source of truth for multi-user pipelines (NVIDIA, 21 December 2025). In contrast, X-TwinUSD targets exascale-class execution by distributing composition and query across HPC workers under explicit consistency assumptions. Earlier research on distributed scene graphs (e.g., the blue-c distributed scene graph) explored synchronization and relaxed locking for shared 3D state; X-TwinUSD differs by centering the OpenUSD composition model and its caching/change-processing behavior in the distributed design (Naef, M., Lamboray, E., Staadt, O., & Gross, M., 2003, May).

7. Discussion and Limitations

7.1 Correctness envelope and operational assumptions

As a design-oriented system, X-TwinUSD explicitly defines an operational envelope under which its distributed execution preserves USD semantics and produces deterministic query outputs for a given epoch. The intent is to make the correctness

contract explicit and reviewable rather than implicit.

Read-mostly stage assumption: interactive workloads are dominated by queries and telemetry updates; high-level authoring edits occur less frequently.

Telemetry deltas are restricted to attribute value updates on existing prims (e.g., transforms, numeric state). Prim creation/deletion, relocates, and complex list-edit conflicts are performed at epoch boundaries or under a single-writer policy.

Epoch-consistent snapshots: a query pinned to epoch e observes a consistent cut where all contacted partitions have applied deltas up to the committed watermark for e. The model provides bounded staleness rather than global serializability.

Deterministic asset resolution: all workers share the same resolver context (search paths, plug-in versions, and environment), preventing divergent path resolution across nodes.

Boundary resolution: cross-partition STRONG dependencies (e.g., variant redirection and list edits) are resolved deterministically at consumer boundaries, ensuring equivalence to single-process composition within the declared envelope.

Supported operations matrix (scope boundary). Table 6 summarizes which USD operations are supported inside the correctness envelope, and which require future hardening.

Operation class	Supported in current envelope?	Notes / required hardening
Attribute updates on existing prims (transforms, numeric state)	Yes	Applied as epoch-scoped deltas; recomposition uses incremental partition updates.
Payload activation / deactivation (working set)	Yes	ROI-driven; correctness assumes deterministic asset resolution across workers.
Prim creation/deletion, reparenting, variant/prim-index structural edits	No	Would require global conflict resolution, distributed Pcp index updates, and stronger serialization.
Cross-partition strong dependencies (variant redirection, list-edits)	Partially	Handled only under deterministic boundary rules; general case needs distributed dependency tracking.
Time-sampled animation curves / high-frequency streaming	Limited	Higher rates may require batching, compression, and GPU-side interpolation to meet latency targets.

7.2 Limitations and future hardening

Outside the envelope above, full generality would require stronger coordination mechanisms (e.g., global conflict resolution for list-edit operations, distributed locking for structural edits, or consensus for strict serializability). We view these as extensions that can be incrementally added once the baseline prototype validates performance scaling and correctness under the declared workload patterns. Additionally, the measured results reported in Section 5 are microbenchmark measurements on a single

host; validating end-to-end performance on a multi-node cluster with a production USD runtime and parallel storage is left as future work.

Firstly, X-TwinUSD presupposes that OpenUSD composition semantics are maintained across distributed workers. Although deterministic composition is attainable, meticulous engineering is necessary to address edit targeting, layer mutability, and versioning within a cluster. Secondly, the cost model C(i) is deliberately lightweight and requires calibration for each workload; inaccurate estimations may result in imbalance. Thirdly, the process of mapping continuous telemetry streams into USD attributes involves trade-offs between immediacy and overhead. In practice, high-frequency numeric fields might be optimally represented outside USD and projected into the stage at a controlled cadence for interactive queries.

Finally, the proposed storage layout introduces a hybrid persistence strategy (USD assets + HPC containers). This approach enhances performance but adds complexity to pipeline governance and reproducibility. It is recommended to treat containerized composed fragments as derived artifacts (cache) that can be regenerated from source USD and delta logs.

8. Conclusion and Future Work

This paper presents the design of X-TwinUSD, a distributed runtime concept for OpenUSD-based digital twins that targets three dominant bottlenecks: metadata fan-out during stage open, payload working-set activation for first interaction, and low-latency updates under streaming telemetry. To provide measured evidence within the constraints of the current build environment, we implemented a minimal microbenchmark prototype that isolates these kernels and reports single-node measurements.

Measured results on synthetic City-S/City-M workloads show that routing only the ROI working set and reducing metadata fan-out can lower TTFI by one to two orders of magnitude, and that incremental partition recomposition can reduce update latency by ~3–7× relative to full recomposition under sparse deltas (Section 5). Future work is to (i) implement the design atop a production OpenUSD stack (Pcp/Usd/Stitching), (ii) validate correctness under richer USD edit classes (variants, list-edits, structural edits), and (iii) conduct end-to-end scaling experiments on a multi-node HPC cluster with parallel storage (e.g., Lustre/GPFS) and representative plant digital-twin datasets.

References

ADIOS2 (20 December 2025). *ADIOS 2 Documentation.* https://adios2.readthedocs.io

Ahmad, Y., Kennedy, O., Koch, C., & Nikolic, M. (2012). Dbtoaster: *Higher-order delta processing for dynamic, frequently fresh views.* arXiv preprint arXiv:1207.0137.

Godoy, W. F., et al. (2020). Adios 2: The adaptable input output system: *A framework for high-performance data management.* SoftwareX, 12, 100561.

Gupta, A., Mumick, I. S., & Subrahmanian, V. S. (1993). *Maintaining views incrementally.* ACM SIGMOD Record, 22(2), 157–166.

Naef, M., Lamboray, E., Staadt, O., & Gross, M. (2003, May). The blue-c distributed scene graph. *In Proceedings of the Workshop on Virtual Environments 2003* (pp. 125–133).

NVIDIA (20 December 2025). References and Payloads – Learn OpenUSD. https://docs.nvidia.com/learn-openusd/latest/creating-composition-arcs/references-payloads/index.html

NVIDIA (21 December 2025). Platform Overview. *Omniverse Developer Overview.* https://docs.omniverse.nvidia.com/dev-overview/latest/platform-overview.html

OpenUSD (20 December 2025). I*ntroduction to USD.* OpenUSD Documentation. https://openusd.org/docs/

OpenUSD (21 December 2025). Maximizing USD Performance. OpenUSD Documentation. https://openusd.org/docs/Maximizing-USD-Performance.html

OpenUSDa (20 December 2025). USD Terms and Concepts (Glossary). OpenUSD Documentation. https://openusd.org/dev/glossary.html

OpenUSDa (21 December 2025). Pcp: PrimCache Population (Composition). OpenUSD Developer API Reference. https://openusd.org/dev/api/pcp_page_front.html

OpenUSDb (20 December 2025). UsdPayloads Class Reference. OpenUSD API Documentation. https://openusd.org/dev/api/class_usd_payloads.html

OpenUSDb (21 December 2025). Hydra 2.0 Getting Started Guide. OpenUSD Developer API Reference. https://openusd.org/dev/api/_page__hydra__getting__started__guide.html

The HDF Group (20 December 2025). *A Brief Introduction to Parallel HDF5.* HDF5 Documentation. https://support.hdfgroup.org/documentation/hdf5/latest/_intro_par_h_d_f5.html.

Submitted: 05.01.2026
Accepted: 23.04.2026