ORCA: Steerable Observability for Bulk-Synchronous Parallel Applications

Ankush Jain*, Sheng Jiang*, Charles D. Cranor*, Qing Zheng†
Brian Atkinson†, George Amvrosiadis*, Gary A. Grider†
*Carnegie Mellon University, †Los Alamos National Laboratory
ankush.jain@databricks.com[1], {shengjiang, chuck, gamvrosi}@cmu.edu, {qzheng, batkinson, ggrider}@lanl.gov

***Abstract*—The *bulk-synchronous parallel* (BSP) paradigm powers critical computational workloads from scientific simulations to large-scale machine learning training. Running on tens of thousands of processors, these systems are uniquely sensitive to performance variations that current observability approaches struggle to diagnose—post-hoc analysis of massive traces introduces latency and rigidity that obstructs timely diagnosis.**

To address these challenges, we present ORCA, a system for always-on, steerable BSP observability. ORCA enables operators to materialize relevant views from running applications on demand and intervene in response, via a tree-based overlay providing coordinated collection, in-situ SQL analytics, and timestep-consistent control. ORCA accelerates observability workflows across the spectrum. On a real AMR code at 4096 ranks, ORCA collects equivalent traces at ∼2% overhead (vs 56–81% for tracing tools), enables up to 7275× faster queries, and 99.999% in-situ data reduction. In an LLM-driven evaluation of steerability, a performance anomaly that took us months to diagnose manually is localized in 27 minutes.

***Index Terms*—Observability, distributed tracing, in-situ analytics, bulk-synchronous parallelism, computational steering.**

I. Introduction

The *bulk-synchronous parallel* (BSP) [1] paradigm powers critical computational workloads, from scientific simulations to large-scale machine learning training. These systems are uniquely fragile: at the scale of tens of thousands of processors, lockstep execution means a single straggler can block cluster-wide progress at every synchronization point. As ML training pushes BSP toward 100K GPUs [2] and scientific workloads grow in tandem [3, 4], the cost of undiagnosed performance pathologies has become economically untenable. Yet the causes are diverse and highly context-dependent [5–24] —load imbalance, poor task overlap, hardware variability—varying across clusters, workloads, and scales. Metrics [25, 26] provide baseline visibility but rarely enough workload structure to attribute causes. Tracing [27–31] can capture the fine-grained patterns needed for diagnosis, but prevailing approaches—exhaustive collection to disk followed by expensive *extract-transform-load* (ETL) [10, 32] —cannot provide feedback at the speed problems demand. In a prior study of placement in AMR codes [10], we found that isolating the root causes of several performance artifacts took months, while fixes typically took days. Rapid visibility into production pathologies remains one of the highest-yield paths to efficiency.

We argue for always-on, steerable observability as a production capability for BSP workloads, not just faster post-hoc analysis, but continuous visibility with the ability to escalate on demand. The evolution of distributed tracing [33–35] offers an instructive parallel. In microservices, tracing evolved to preserve causal relationships between requests (the unit of work in asynchronous systems) so tail latency is continuously monitored and attributable to specific services. In BSP, causality is induced by synchronization: performance anomalies manifest as stragglers that stall collectives. The same synchronization creates natural windows for both observation and intervention, enabling operators to steer collection on demand, test hypotheses live, and act on feedback while the job runs. Real-time diagnostics are already proving valuable in ML training [16–24] for problems ranging from straggler localization to training correctness, but each system implements a bespoke collection and analytics pipeline. BSP workloads need a common observability substrate, providing low-overhead on-fabric transport, BSP-aware coordination semantics for telemetry aggregation and control, and programmable analytics interfaces (§III).

In this paper, we present a tree-based overlay [36] for rapid feedback loops over large-scale BSP applications called ORCA (*Observability with Real-time Control and Aggregation*). ORCA treats telemetry as columnar streams and provides in-situ SQL-based analytics, faster offline analytics, and a control plane for online steerability. Central to the design is treating synchronization as the fundamental consistency boundary. On the data path, this yields *timestep dataframes* (§IV-A): a per-timestep grouping of all ranks' telemetry into a single logical analysis unit. Because synchronization partitions BSP execution into causally independent windows, timestep dataframes support both in-situ analysis during execution and timestep-partitioned persistence for faster offline queries. Combined with columnar formats (Arrow [37] in transit, Parquet [38] at rest), this eliminates the ETL that dominates conventional tracing workflows. An RDMA-based transport (§IV-B) minimizes collection overhead, while distributed query planning with operator pushdown minimizes data movement (§IV-C), On the control path, a timestep-consistent commit protocol (§IV-D) enables atomic interventions—new analytics, probes, or parameter updates—applied asynchronously at the same timestep across all ranks.

[1]Work done while at CMU.

SC26, November 15-20, 2026, Chicago, Illinois, USA
979-8-3195-4789-7/26/$31.00 ©2026 IEEE

We evaluate ORCA on a real AMR code at 512–4096 CPU ranks across tracing, in-situ, and agentic workflows:

- In conventional tracing workflows, ORCA captures equivalent telemetry at ∼2% runtime overhead (vs 56–81% for state-of-the-art tracers), produces up to 42× smaller traces, and enables up to 7275× faster offline analyses (§VI-A).
- In-situ analytics enable low-overhead needle-in-the-haystack workflows—outlier `MPI_Wait` calls are isolated in real-time at sub-2% runtime overhead, while discarding up to 99.999% of data at MPI ranks (§VI-B).
- In an agentic evaluation of its steerable capabilities using an LLM agent, ORCA enables localizing a performance anomaly—the same bug that took us months to diagnose in a prior study [10]—in 27 minutes, while reducing data volume by up to 144× (§VI-C).

More broadly, ORCA demonstrates that declarative SQL analytics can be embedded on the fabric alongside a live MPI application, enabling streaming analytics workflows beyond observability (§VIII). ORCA is open source [39].

II. Background and Motivation

Bulk synchronous parallel (BSP) [1] workloads in both scientific computing and machine learning training pose unique efficiency challenges. These jobs, increasingly spanning tens of thousands of nodes, execute in lockstep and synchronize frequently via collective communication. At these scales, BSP is uniquely fragile: a single straggler can block cluster-wide progress at synchronization points. Their root causes are highly context-dependent, ranging from load imbalance and suboptimal overlap to tuning-induced variability, and have been documented extensively [5–24]. Despite extensive case studies, performance pathologies persist and remain costly to diagnose and fix.

We argue this reflects a structural limitation in how BSP observability is practiced. While monitoring infrastructure is widely deployed [25, 26], it targets *known unknowns* via a fixed set of metrics such as CPU utilization. Performance pathologies are dominated by *unknown unknowns* [40–42] — they cannot be anticipated by predefined metrics, and resolving them requires iterative investigation at varying resolutions: from high-level aggregates to fine-grained detail when needed. Current approaches to capturing fine-grained telemetry — large event traces followed by post-mortem analyses — incur prohibitive collection and analysis costs [10, 32], delaying interventions. As a result, operators are forced to choose between expensive investigations and ad-hoc workarounds. This gap motivates a different model: always-on application observability as a production capability, with relevant views constructed on demand in real-time, hypotheses tested live, and collection steered accordingly while the job is still running — a paradigm we call *steerable observability*.

We now illustrate this gap with case studies from HPC and ML training (§II-A), survey existing approaches and their limitations (§II-B), and use our experiences with eBPF [43] to motivate a programmable substrate for real-time visibility at workload scale (§II-C).

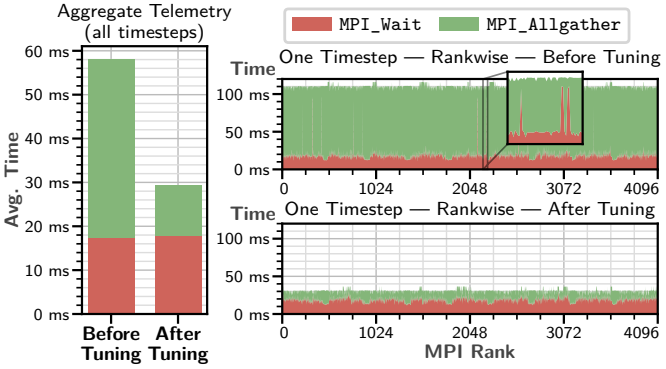


Fig. 1: A performance pathology we identified in an AMR code [10]. A volatile `MPI_Wait` preceding `MPI_Allgather` is misattributed as elevated collective cost in aggregate telemetry (left). Rank-level telemetry for a single timestep (right) reveals the true cause but requires joining all ranks' data.

A. Case Studies from HPC and ML

1) Optimizing Placement in AMR: A prior study of placement and load-balancing in *adaptive mesh refinement* (AMR) codes [10] illustrates this gap concretely. Although AMR codes are widely known for load-imbalance challenges [44], we found load imbalance indistinguishable from other hardware, software, and tuning artifacts without fine-grained telemetry. Fig. 1 shows one such example: tail latency spikes in `MPI_Wait` that only manifested at scale would get attributed to collectives in aggregate telemetry. Each artifact took us months to diagnose, while mitigations typically took days.

The primary diagnostic bottleneck in our study was a slow feedback loop: each iteration required running the codes, collecting large event traces, followed by post-mortem trace analyses bottlenecked on expensive ETL (*extract-transform-load*) from formats such as CSV. Mitigations identified were cluster- and workload-specific. Even after these artifacts were fixed, placement demonstrated a fundamental tradeoff between compute load balance and communication locality — the right balance between the two varies across workloads, scales, and architectures. Translating these optimizations to production environments would require repeating this time-consuming exercise — underscoring the need for infrastructure that makes such investigations routine.

2) Parallels from the Pytorch Ecosystem: The PyTorch [45] ML training ecosystem exhibits the same offline feedback loop. Kineto [29] captures fine-grained traces on demand as JSON, triggered via a node-local control daemon called Dynolog [46]. *Holistic Trace Analysis* (HTA) [47] computes derived views such as activity breakdowns and straggler detection. Together, these cover capture, actuation, and analysis — but HTA is a single-node Pandas [48] script that cannot scale to cluster-wide traces, and Dynolog provides best-effort actuation with no workload-level consistency guarantees.

Recent ML training systems have begun addressing specific pathologies online — straggler localization [16–20], fault diagnosis [22, 23], and training correctness [24] — but each reimplements subsets of collection, aggregation, and control from scratch. A common programmable substrate would gen-

eralize these diagnostics, allowing detection strategies to be expressed as analytical queries over system state, composable without rebuilding the underlying infrastructure.

B. Existing Approaches and Gaps

Monitoring. Monitoring enables basic cluster visibility via periodically captured metrics, but lacks the workload context necessary to construct views like Fig. 1. Widely deployed infrastructure for metrics includes Prometheus [25], a time-series database, and LDMS [26], an on-fabric hierarchical metric collection infrastructure used in HPC clusters.

Profiling and Tracing. Tools such as TAU [27], Score-P [28], and Kineto [29] can provide both high-level aggregates via profiles or fine-grained telemetry via traces, but are post hoc by design. Traces can characterize short runs, but cannot provide always-on visibility over long-running jobs. In the absence of any real-time feedback to guide collection, traces simultaneously overcollect and undercollect — despite capturing every `MPI_Wait`, they can not explain which sub-functions within anomalous `MPI_Wait` caused the latency spikes. Formats like OTF2 [49] and JSON [50] require expensive transforms for dataframe-style analytics [10, 32], and visual inspection for anomalies is impractical for cluster-scale workloads.

Dataframe Analytics. Recent tools such as Caliper [31], DFTracer [30], Pipit [51], and Pytorch HTA [47] have adopted Pandas [48] and Dask-based [52] dataframe analytics interfaces to analyze telemetry, but require expensive ETL [10, 32] — the working set for which scales with *ranks* $\times$ *timesteps*. Columnar formats like Parquet [38] offer efficient post-hoc analytics, but do not by themselves address overcollection, working set scaling, or real-time feedback, and remain largely unadopted in tracing workflows. In a simple internal port of HTA analyses from JSON to Parquet, we observed $12\times$ faster data loading and $8\times$ faster end-to-end analysis.

Streaming and In-situ Analytics. ADIOS [53] enables fixed-function in-situ workflows [54] on scientific data, typically run as another MPI application. MRNet [55, 56] is a tree-based overlay used as a reduction backend for commercial debuggers, but provides a compiled packet-based programming model. Neither enable runtime-programmable dataframe-style analytics needed for real-time BSP observability. Cloud streaming systems such as Flink [57] and Kafka [58] are designed for independent event streams over IP networks They lack fabric-aware transport and BSP-aligned coordinated collection, precluding views like Fig. 1.

C. The Missing Substrate

Diagnosing the root cause behind Fig. 1 required capabilities beyond what any single tool in the taxonomy provides. At node scale, eBPF [43] provides a template: dynamic probes, programmatic aggregation, low overhead in the common case with depth when needed. It enabled controlled and targeted access to telemetry within `MPI_Wait` after alternatives were exhausted. Hypotheses about interrupts and scheduler effects were tested and discarded in hours. But eBPF is node-local, and our workflow involved ad-hoc orchestration, uncoordinated collection, and manual post-hoc correlation. Extending this to workload scale requires a substrate for real-time *feedback*, treating observability not as distributed logging, but as materializing views directly from running applications. But visibility alone is insufficient for steerability. Interventions — whether additional views or parameter adjustments — require a *control plane* back to the running application, and may be driven by humans, automated agents, or policies. The next section states the requirements such a substrate must satisfy.

III. Requirements

Our goal is to accelerate the full spectrum of observability workflows, from offline trace analysis to real-time diagnosis. This requires analytics to transform fine-grained telemetry in situ, terminating in aggregated views or detailed traces but reorganized for efficient offline analyses. Additionally, control semantics are needed for interventions ranging from steering application views to steering the application itself. We model BSP as consisting of timesteps punctuated by synchronous collectives [59, 60], with timesteps containing one or more collectives. Steerable observability for this model must support three key requirements.

BSP-Aware Semantic Guarantees. Both the data and control paths must be synchronization-aware. On the data path, both transport and analytics must offer consistency guarantees, as all telemetry for a timestep must be collected and analyzed as a single unit. Control inputs must similarly be applied consistently and atomically at timestep boundaries across all ranks. While uncoordinated updates may be tolerable in some cases, they break correctness for interventions such as hyperparameter changes — a timestep-consistent control plane generalizes across use cases.

Low-Cost, Low-Latency Feedback. For always-on view-driven observability to be viable, overhead must be low and proportional to view complexity. Induced jitter must be minimized, so that continuously monitored aggregates incur minimal cost, while detailed traces remain available on demand. Analytics must execute in real-time, as feedback latency bounds intervention latency.

Declarative and Flexible Analytics. No predefined set of analyses can anticipate every pathology. Operators must be able to specify views and collection policies via a programmable interface at runtime, without recompilation or restart. Collection and analytics must be schema-agnostic, enabling visibility into any layer of the runtime stack — from MPI and fabric libraries to the application itself.

At scale, both data and control paths necessitate distributed execution, requiring a careful balance of consistency guarantees with asynchrony for low overhead. Our design leverages modern columnar primitives (Arrow [37], Parquet [38]), an extensible query engine called DataFusion [61], and RDMA for low-overhead transport, with a tree-based topology for reductions and broadcast.

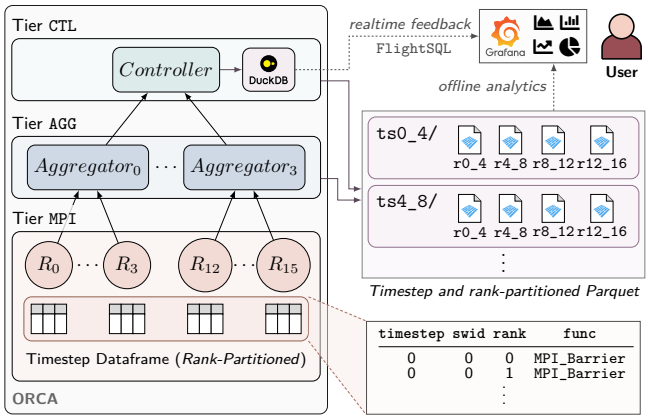


Fig. 2: ORCA architecture. A three-tier TBON with application ranks (MPI) at the leaves, intermediate aggregators (AGG), and a controller (CTL) at the root. Telemetry originates as columnar timestep dataframes, is transformed by SQL plan operators at each tier, and persisted via DuckDB and Parquet sinks — enabling workflows from real-time feedback via lightweight metrics to zero-ETL post-hoc analytics.

IV. Design

ORCA (Fig. 2) is a programmable *tree-based overlay network* (TBON) with a controller (CTL) at the root and aggregators (AGG) as the intermediate scale-out tier, attached to application ranks (MPI) as the leaf tier. CTL and AGG run on dedicated nodes outside the MPI domain and communicate with the application via on-fabric RPCs. The TBON provides asynchronous data and control planes. Data originates as columnar Arrow [37] streams at leaf ranks, flows through the TBON, and is persisted via sinks. Two sinks are currently implemented: Parquet [38] for high-volume telemetry, and DuckDB [62] for real-time metrics streamed to dashboards. We present ORCA as a progression of capabilities that successively shorten the observability feedback loop:

- Timestep-consistent, low-overhead collection is provided via a timestep-based data model (§IV-A) along with an RDMA-based transport (§IV-B). By themselves, they write timestep-partitioned traces, enabling a workflow similar to conventional tracers, but more efficient to collect and analyze.
- A SQL-based interface called *OrcaFlow* (§IV-C) provides declarative in-situ analytics over data en-route to sinks, with automatic operator pushdowns to minimize data movement.
- Timestep-consistent online interventions are provided via a speculative two-phase commit variant called TS2PC (*timestep-linked two-phase commit*), ensuring atomic commit across all ranks at the same timestep (§IV-D).

A. Data Model: Timestep Dataframes

ORCA workloads emit telemetry via named and *typed* streams generating *timestep dataframes*, which are the logical unit of collection, analysis, and persistence. Applications can have any number of streams, each generating events with a fixed schema. Streams are flushed at the end of each timestep to form these dataframes, comprising all events from that timestep. Timestep dataframes are logical constructs consisting of rank-partitioned shards that are transported over the TBON asynchronously, but analyzed and persisted as a single unit. Timesteps can contain multiple collectives, dividing execution further into causally-independent *synchronization windows*, assigned unique monotonic identifiers called `swids`.

Timestep dataframes enable efficient analytics by capturing the causal structure of BSP execution. At blocking collectives, all ranks wait for stragglers and execution resets. Each timestep is therefore, naturally, an independent analysis unit. Pathologies are first attributed to a straggler rank that delayed a collective, then to events in the dataframe that led to that straggler. This mirrors the evolution of distributed tracing in loosely-coupled microservices — from treating traces as bags of events [33] to capturing causal edges between events via *request IDs* [34, 35]. For BSP workloads, this timestep-centric dataframe abstraction confers two analytical advantages:

- In transit, data can be represented as Arrow [37], whose contiguous columnar layout enables efficient serialization into RDMA-registered memory, zero-copy deserialization and in-situ analytics, while types such as dictionary arrays map naturally to repeated symbols in trace data.
- At rest, data is persisted as *timestep-partitioned* Parquet [38] (see Fig. 2), a columnar analytics-oriented format and a natural counterpart to Arrow. It not only enables zero-ETL analytics, but also reduces data read via rowgroup statistics and operator pushdowns.

Conventional tracers [27–31] not only employ ETL-heavy formats [49, 50], but create rank-major layouts that fragment a single timestep across thousands of per-rank logs. ORCA's partitioning aligns the primary key of the storage layout with the causal structure of BSP telemetry. Among prior work, Hatchet [63] postprocesses traces to build callflow graph indexes. Callflow graphs are a secondary index for BSP telemetry, complementary but optional when the primary key (`timestep, rank`) is effective.

B. TBON-Optimized RDMA Transport

ORCA's transport layer is designed to move Arrow dataframes asynchronously over an RDMA fabric across TBON tiers. Application overhead is minimized via an incast-optimized, receiver-driven mechanism that avoids synchronized pushes and overlaps data movement with application execution. As the CTL and AGG nodes are fully asynchronous and throughput-optimized, fabric QoS features [64] can be used to relegate ORCA traffic to a lower priority class, relying on idle fabric capacity for low-perturbation transfer.

The transport layer is agnostic to how telemetry is generated. Different profiling mechanisms may accumulate telemetry into per-stream *collectors* — local buffers that emit Arrow dataframes when flushed. After each timestep, all collectors on each rank are flushed to generate per-stream Arrow dataframes, which are then serialized into RDMA-registered memory. Ranks then send an RPC to their aggregators containing RDMA descriptors for the serialized data. Ranks then resume execution, while aggregators collect these dataframes

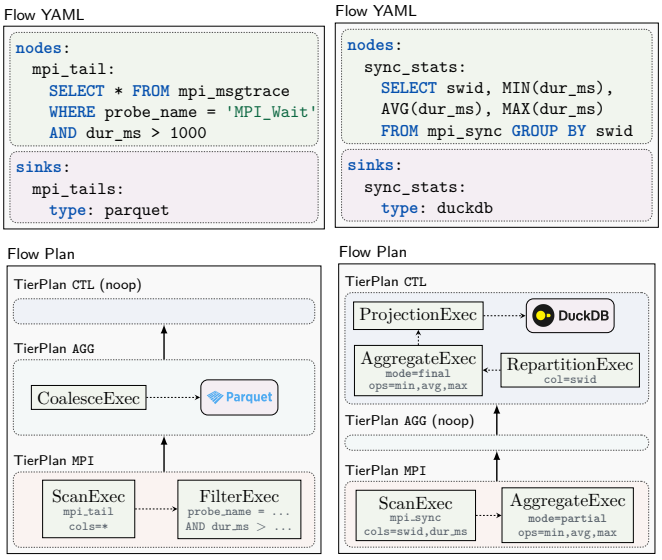


Fig. 3: OrcaFlow compilation showing flows and their compiled tier plans. Flows chain SQL transforms over telemetry streams en route to configurable sinks. Both operators and sinks are placed automatically. (Left) A selective filter pushed to MPI, discarding most data at source. (Right) Aggregations also reduce data volume — pushed-down partial aggregations emit fixed-size intermediates regardless of input volume.

via asynchronous RDMA GETs. After multiple iterations, we developed three key mechanisms for stable TBON operation:

- Aggregators must issue RDMA GETs at controlled concurrency to prevent catastrophic throughput collapse, as NICs have limited resources to track active RDMA operations (1–2 on older fabrics). The pull-based model, which resembles receiver-driven flow control, lets aggregators stay in control of data volume and is essential for stable incast operation.
- Because the collection pipeline yields dataframes in timestep order, it is vulnerable to head-of-line blocking from missing shards, risking *out-of-memory* errors. We address this with a priority queue at aggregators that prioritizes lower timesteps.
- While the system must be provisioned with adequate bandwidth to ensure minimal backpressure, flow control is essential for stability. Aggregators communicate backpressure by withholding RPC ACKs, and ranks block if outstanding RPC count exceeds a configured limit.

For offline analyses, aggregators compress and persist data at preconfigured timestep intervals. As the incoming data is already partitioned by timestep and rank, the TBON simply preserves this partitioning.

C. In-Situ Analytics for Real-time Feedback

The timestep dataframe abstraction also enables zero-copy in-situ SQL analytics at every TBON tier, enabling low-latency materialization of views and drastically reducing persisted data volume. OrcaFlow (Fig. 3) is a SQL-based programming interface that turns the TBON into a programmable reduction tree, transforming dataframes en route to sinks — currently partitioned Parquet for offline analysis and DuckDB for real-time metrics accessed by dashboards [65]. For analyses difficult to express in SQL, *map* operators allow invoking arbitrary precompiled kernels.

OrcaFlow is backed by a TBON-aware distributed planner and query engine built on Apache DataFusion [61]. Each flow compiles into *tier plans* describing transforms to apply, data to forward, and sinks to execute locally. The planner automatically places both transforms and sinks, pushing operators that reduce data volume as close to the source as possible. MPI and AGG operate on rank-partitioned shards and execute per-partition operators: selection, projection, partial aggregation. Cross-partition operations like joins must run on CTL, the root aggregator. Sinks are placed on the lowest tier that produces their stream (no sinks on MPI, DuckDB only on CTL). Fig. 3 illustrates with two flows:

- *(Left)* A selective tracing flow for MPI_Waits $>$ 10 ms pushes filters all the way to the MPI tier. Results are coalesced at AGG and written as Parquet. Most telemetry for pathologies like Fig. 1 gets discarded at source.
- *(Right)* An aggregation computing per-collective statistics splits into a partial aggregation at MPI and a final merge at CTL. Each leaf emits one row per group regardless of input volume — sketches and quantile digests fit the same model, producing fixed-size intermediates.

The programming model resembles batch analytics frameworks like Spark [66], but targets a multi-tier TBON, remains in-memory until sink ingestion, and avoids distributed shuffling to maintain predictable throughput. All tiers collect dataframes asynchronously but analyze them in timestep order. Per-partition compute budget scales with MPI and AGG, while cross-partition work is limited to a single node (CTL). This suffices for residual work after filtering and pre-aggregation. Post-mortems remain the preferred workflow for expensive cross-partition analyses, but still benefit from ORCA's in-situ filtering, preaggregation, and timestep-partitioned Parquet.

D. Control Plane for Steerable Observability

The control plane enables users to intervene in response to feedback, closing the loop between observation and action. Interventions range from steering collection to steering the application itself in response to observed behavior. Prior work has explored auto-tuning for specific use cases [67]. ORCA instead provides general mechanisms for timestep-consistent feedback and control, on which such policies can be built.

Requirements. The control plane must be asynchronous, as the controller is not part of the collective domain. It must also be atomic and timestep-consistent — commands must be acknowledged by all ranks before being executed, and they must execute on all ranks at the same timestep for correctness.

The TS2PC Protocol. ORCA implements the control plane as a replicated state machine [68] using *timestep-linked two-phase commit* (TS2PC). Commands are opaque payloads — the protocol only decides when to commit and apply them. As the controller is asynchronous, it maintains a lagging view of the application's current timestep ($= T_{app}$). TS2PC proposes commands for a future timestep $T_{cmd} = T_{app} + \Delta$, where Δ

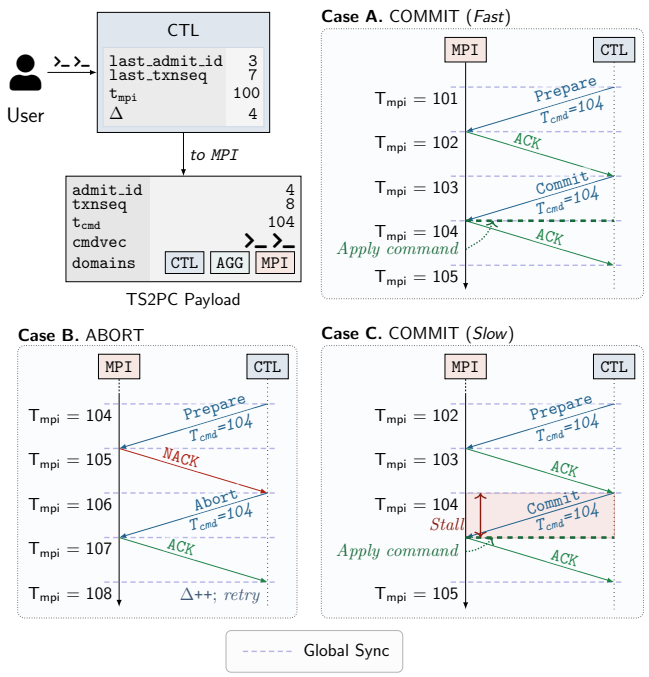


Fig. 4: Timestep-linked two-phase commit (TS2PC). Commands are proposed for a future timestep $T_{cmd} = T_{mpi} + \Delta$. (A) Normal case: ranks agree and apply the command before executing T_{cmd}. (B) Abort: ranks have passed T_{cmd} and vote to abort, command is retried with a larger Δ. (C) Stall: ranks have prepared but not received commit/abort. Execution blocks until resolution to preserve correctness.

is an automatically learned speculative margin. The TBON assists the protocol by broadcasting requests and reducing responses — any NAK in the subtree results in a single NAK at the controller. Fig. 4 illustrates the protocol via three cases.

- *Case A (Commit):* Ranks agree to commit if they have not yet reached T_{cmd}, and apply the command just before T_{cmd}.
- *Case B (Abort):* If ranks are past T_{cmd}, they vote to abort. The controller processes the abort, increments Δ, and automatically retries the transaction. As a result, Δ quickly converges to the natural latency of the TBON.
- *Case C (Stall):* A subtlety arises when ranks have prepared but not yet received commit/abort by T_{cmd}. Correctness requires stalling execution until commit/abort is received, so the prepared command can be applied before T_{cmd}.

More implementation-specific details are discussed in §V-B.

V. Implementation

The core ORCA framework consists of 40K lines of code (80% C++, 20% Rust). ORCA uses the Mercury [69] RPC library for on-fabric communication via libfabric [70] and UCX [71] providers. The Rust portion consists of OrcaFlow's DataFusion-based query planning and execution on each node, while the bulk of the functionality — bootstrap, transport, and control — lies on the C++ side.

A. Implementation Overview

Integration. Two integration surfaces exist: the *OrcaClient* shim for applications, and collectors for profiling interfaces.

- *OrcaClient:* The *OrcaClient* shim links with the application and exposes a `PostTimestepAdvance()` hook, called after every timestep. ORCA can be activated at runtime by preloading the full library, otherwise these hooks become no-ops. `PostTimestepAdvance()` captures all data path work, including draining collectors, executing tier plans, dispatching data, and handling control messages.
- *Collectors:* Streams are implemented via collectors that accumulate events locally and emit Arrow dataframes when drained. At the end of each timestep, ORCA drains all collectors and enqueues the output for analysis. Our implementation currently packages basic collectors for MPI [59] and Kokkos [72]. Additional collectors may be defined by clients. Collectors can also implement dynamic instrumentation via interfaces like eBPF [43] and DynInst [73].

Overlay. The overlay bootstraps via a TCP-based protocol and switches to on-fabric RPCs after the initial handshake. While it supports additional topologies for use cases like intermediate fan-in reduction, only one is discussed for simplicity.

Data Path. The data path is completely asynchronous but timestep-ordered. Each tier executes its tier plans on dataframes in timestep order independently, and forwards relevant data to the next tier. Despite its asynchronous nature, flow updates across tiers also guarantee timestep-consistency.

B. TS2PC: Implementation Details

TS2PC is implemented as separate *inner* and *outer* layers. The inner layer enables a single transaction with a unique monotonic ID called `txnseq`, targeting a particular T_{cmd}. The outer layer adds features like automatic Δ-learning (discussed in §IV-D) and *control domains* for cross-tier consistency.

Concurrency. Correctness necessitates that only one transaction be in flight at any given time, as intermittent failures and automatic retries can result in commands being committed out of order. However, a single transaction can contain an arbitrary number of commands applied as a batch.

Handling Pauses. The control plane supports pausing and resuming the application between timesteps for basic lifecycle management, and as primitives for more complex workflows. An interesting case emerges from the interaction of the paused state with TS2PC — resume commands in this state will be committed for a future timestep never reached by a paused application, creating a deadlock. Our solution recognizes that the paused state makes the application synchronous with the user and renders TS2PC unnecessary for consistency. Commands in this state circumvent the protocol and are executed directly.

Cross-tier Consistency. Commands such as flow updates trigger tier plan updates at all tiers, and must be applied at a consistent timestep across the entire TBON. This is challenging as the higher tiers are asynchronous and observe application timesteps at different times. A tier is defined to observe a timestep when it processes its dataframes.

Cross-tier consistency is supported via *control domains*, which make each overlay tier independently addressable by the control plane in a timestep-consistent manner. Each tier corresponds to a domain, and each transaction carries a bitmask specifying which domains it targets — flow updates, for example, target all domains. The inner protocol remains unchanged, with MPI making all commit decisions as the single source of truth on the application timestep, regardless of the domain. Higher tiers snoop on protocol traffic and apply commands only if their domain bit is set.

VI. Evaluation

We evaluate ORCA by progressively enabling its capabilities. As a passive telemetry collector, we compare runtime overhead (§VI-A1), storage efficiency (§VI-A2), and analytics performance (§VI-A3) against state-of-the-art tracers. We then demonstrate online analytics (§VI-B) and control workflows (§VI-C) — capabilities with no existing counterparts.

Hardware Setup. Experiments run on a 600-node research cluster with QLogic Truescale fabric (Intel Xeon E5-2670, 16 cores/node, 64 GB RAM, 40 Gbps QLogic 7340 NICs) and a 20-node Lustre filesystem with 3GB/s aggregate bandwidth.

Software Setup. We use the Sedov Blast Wave 3D [74] AMR code from the Parthenon [75] and Phoebus [76] frameworks at 512–4096 CPU ranks, with 16 ranks/node. ORCA uses one aggregator node per 1024 ranks plus one controller node. All experiments use MVAPICH2 v2.3.7 [77] with a tuned PSM [78] library for our fabric. While simulation nodes use PSM, ORCA communication uses Mercury [69] over verbs because of fabric restrictions.

This setup represents a strenuous observability workload: CPU codes make every cycle visible as runtime overhead, the legacy fabric lacks QoS support and forces ORCA onto a non-native interface (verbs), and the code's multiple communication patterns — multiple collectives, halo exchanges, and frequent load balancing — make it highly jitter-sensitive.

A. Tracing Overhead and Efficiency

We compare ORCA in conventional tracing workflows with four state-of-the-art tracing tools (TAU [27], Score-P [28], Caliper [31], and DFTracer [30]) along three axes: runtime overhead, storage efficiency, and analytics performance.

Collection Profiles. ORCA is set up with two tracing profiles: a lightweight profile that only captures MPI collectives, and a detailed profile that also captures all MPI point-to-point messages and Kokkos events. Extremely high-volume events like `MPI_Test` and `MPI_Iprobe` are excluded. For ORCA, the lightweight profile measures the common-case overhead of always-on observability, while the detailed profile measures the worst-case overhead of capturing all MPI and Kokkos events. All other tracing tools are configured with the same detailed collection profile. Unless otherwise specified, all experiments run for 2000 timesteps, and we report the average runtime from three runs. Unlike tracing tools, ORCA is deployed with additional resources: one aggregator node per 1024 ranks, plus one controller node.

1) Runtime Overhead: Fig. 5 compares the runtime overhead of ORCA and other tracing tools for 512–4096 ranks.

***Takeaway:** ORCA incurs an overhead of 0.8–2% across all scales and profiles, vs up to 56–81% for TAU and DFTracer.*

- ORCA remains within 0.8–2% overhead across all scales and profiles, while that of TAU and DFTracer degrades, from 18–20% at 512 ranks to 56–81% at 4096 ranks (Score-P and Caliper discussed separately below).
- At 4096 ranks, the 2% overhead is despite sharing the fabric with a real AMR code with no QoS support, transporting 1.8 GB/s of telemetry (550 GB on-wire, 111 GB after compression, over 308 s).
- Overhead increases only 0.4% between ORCA's lightweight and detailed profiles at 4096 ranks, despite 200× more telemetry (524 MB vs 111 GB), enabled by Arrow's serialization efficiency and a TBON-optimized RDMA transport.

Why ORCA Works. TAU and DFTracer degrade steadily because they flush trace buffers to storage as they fill, which produces uncoordinated flushing patterns. This, coupled with a kernel-based data path for storage, introduces jitter that compounds at scale. ORCA, in comparison, uses an inherently efficient format, serializes synchronously into RDMA-registered buffers, and transports via efficient RDMA GETs.

Caveats on Score-P and Caliper. Because of difficulties running them for the full 2000 timesteps, all numbers for Score-P and Caliper are extrapolated from 200 timesteps. Both tools are designed for short characterization runs where data is buffered in memory and flushed once at the end. Score-P also requires an additional workload pass to estimate trace buffer sizes, and frequently crashed with out-of-memory errors despite generously configured buffer sizes. While Caliper supports periodic flushing, the path was buggy and performance was significantly degraded even after patches.

2) Storage Efficiency: Fig. 6 compares the total size, and the per-record size of the traces emitted by ORCA and other tracers. ORCA writes Parquet, TAU and Score-P are configured for OTF2, DFTracer writes `jsonl`, and Caliper writes its custom plaintext `cali` format. Key findings:

***Takeaway:** ORCA's Parquet traces are up to 6.4× and 44.3× smaller than OTF2 and plaintext formats, respectively.*

- ORCA consistently emits the smallest traces, 3.6–3.8× smaller than TAU, and up to 44.3× smaller than Caliper.
- Differences can be partly explained by format efficiency. Per-record footprint of OTF2, being a binary format, is only 1.4× larger, while that of plaintext formats is 7–13× larger.

Record Amplification. Accounting for format efficiency, other tracing tools emit 2.5–3.3× more records than ORCA, despite the same collection profile.

1) Most tracers generate separate entry and exit records per traced function call, while ORCA and DFTracer generate one record containing both start time and duration. Latter is both more space-efficient and allows duration-based filters without additional preprocessing.

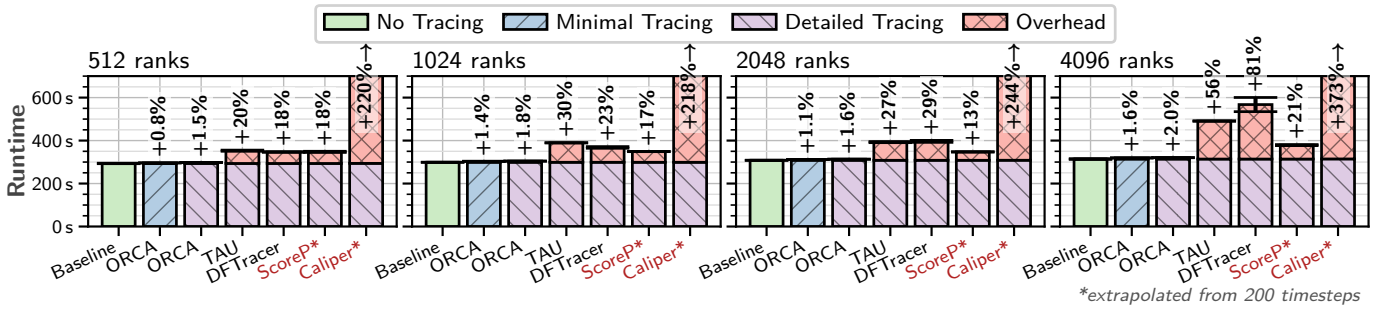


Fig. 5: Runtime overhead for a detailed tracing profile on an AMR code at 512–4096 ranks. An additional minimal profile (collective-only) for ORCA shows always-on overhead. ORCA maintains 0.8–2% overhead across both profiles, while TAU and DFTracer degrade from 18–20% at 512 ranks to 56–81% at 4096 ranks. **extrapolated from 200 timesteps due to stability issues.*

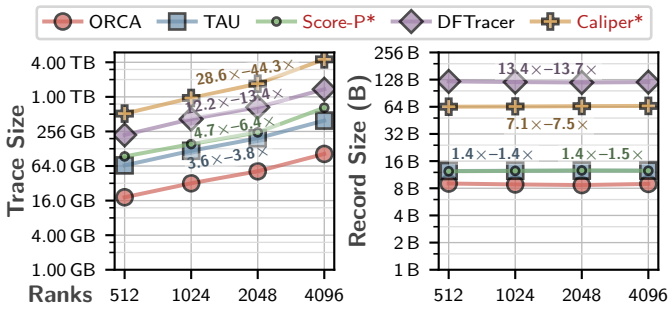


Fig. 6: Trace size (left) and per-record size (right) comparison for detailed profile. ORCA traces are 3.6–44× smaller than other formats. Per-record size isolates format efficiency from record amplification — other tracers emit 2.5–3.3× more records than ORCA for the same collection profile.

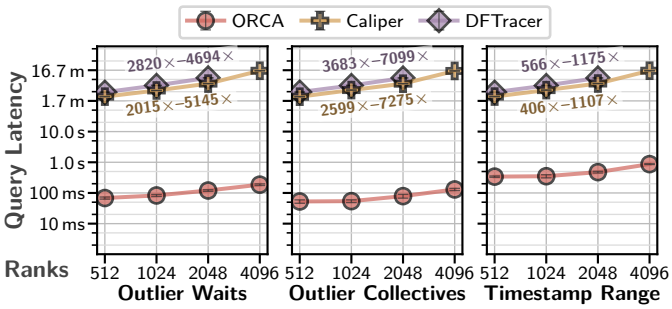


Fig. 7: Post-mortem query latencies on 20 timesteps. ORCA queries execute in O(100 ms), 3–4 orders of magnitude faster than DFTracer and Caliper. Performance for plaintext formats is dominated by parsing, not query complexity. TAU and Score-P are omitted as OTF2 is not a queryable format.

2) TAU incurs additional amplification from hardcoded derived events: for every MPI send and receive, it emits a function event, a message event, and an event for messages received during certain phases such as `MPI_Wait`.
3) Flush-induced jitter with Score-P and Caliper creates stragglers, resulting in additional wait events from other ranks.

Why ORCA Works. ORCA's storage efficiency stems from two design choices. First, Parquet yields an inherently compact representation through binary columnar layout, dictionary encoding, and Snappy [79] compression. Second, separating derived events from collection — computing them via OrcaFlow rather than baking them into the tracer — prevents record amplification from creeping in as features accumulate. We discuss compression further in §VI-A4.

3) Query Performance: We evaluate analytics performance using a suite of post-mortem queries (Fig. 7). All queries run on traces from a smaller 20-timestep run, executed cold with caches cleared, latencies averaged over 3 runs.

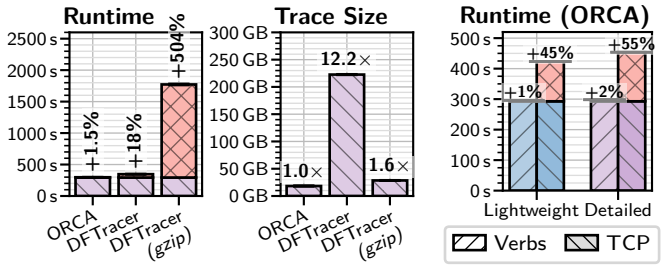
***Takeaway:** ORCA traces enable 3–4 orders of magnitude faster analytics compared to DFTracer and Caliper.*

Queries. We use three queries of increasing complexity. *Outlier Waits* is a simple filter identifying `MPI_Wait` events exceeding 1 ms. *Outlier Collectives* aggregates collective events by `swid` and filters for outliers (max duration $>$ 10 ms). *Timestamp Range* returns all events within the first second of execution—a query that might power a visualization frontend, and notably not a dimension ORCA partitions on.

Analytics Tools. All queries execute on a single 16-core node. DFTracer and Caliper queries use their bundled Python-based analytics libraries. ORCA queries use an off-the-shelf query engine [80] with no partition awareness for simplicity. TAU and Score-P are excluded as OTF2 does not support dataframe analytics. Findings:

- ORCA queries complete in hundreds of milliseconds, 3–4 orders of magnitude faster than DFTracer and Caliper. The gap widens with scale.
- Query times for DFTracer and Caliper do not vary with query complexity. This is because performance is bound by parsing, not computation — entire plaintext traces need to be parsed and loaded as dataframes for queries.

Why ORCA Works. Plaintext formats used by analytics-oriented tracers [30, 31] require expensive ETL to materialize dataframes before any analyses, while Parquet's columnar layout enables zero-ETL analytics. Additionally, ORCA's layout enables both cross-file and intra-file pruning via partitioning and rowgroup statistics. The outperformance in Fig. 7 is despite the query suite not exploiting the former — unmodified Polars with lazy execution must scan all footers. Open table metadata formats [81, 82] would accelerate queries further by adding a catalog layer over Parquet, enabling standard query engines [62, 80] to automatically leverage ORCA's partition-



(a) Compression impact. (b) Transport impact.

Fig. 8: (Left) Compression reduces trace size but induces severe jitter on simulation ranks.ORCA avoids this by offloading compression to dedicated aggregators. (Right) TCP transport increases ORCA overhead from 1–2% to 45–55%. Arrow-RDMA co-design is essential for low overhead.

Flow	Query
Outlier Wait Counts	`SELECT COUNT(*) FROM mpi_wait WHERE duration > 1ms` → *DuckDB*
Outlier Wait Traces	`SELECT * FROM mpi_wait WHERE duration > 1 ms` → *Parquet*
Load Imbalance	`SELECT rank, SUM(duration) FROM kokkos WHERE name IN <compute kernels> GROUP BY rank` → `stats` → *Parquet*
	`PERCENTILES(stats, 50/90/99th)` → *DuckDB*

TABLE I: In-situ workflows evaluated in §VI-B, ranging from lightweight metrics and selective tracing on the communication path to aggregates over compute kernels.

ing. Conventional formats offer no such benefits, requiring larger analytics clusters for ETL as traces grow.

4) Tradeoff Analysis: We now isolate the impact of key architectural decisions using a 512-rank run (Fig. 8).

Takeaway: *Architectural choices — dedicated nodes and RDMA — are key to compression and low runtime overhead.*

Compression. DFTracer supports gzip compression, which reduces its trace size from 12.2× that of ORCA to 1.6× (fig. 8a), but at the cost of increasing runtime overhead from 18% to 504%. Compression on MPI ranks is unviable because it competes with the application for CPU, amplifying jitter. ORCA offloads compression to dedicated aggregator cores, keeping it off the critical path. Additionally, most compressed formats cannot be queried without full decompression, while ORCA queries only require decompressing the files and row-groups that match query predicates.

Transport Layer. Switching from `verbs` to libfabric's `TCP` provider increases ORCA's overhead from 1–2% to 45–55% (fig. 8b). Even with the same Arrow-based data path, TCP suffers because retrieval incurs CPU overhead on simulation ranks, introducing jitter. These results underscore the cost of TCP-based transport for tightly-coupled BSP workloads. While GPU codes may have more tolerance for the transport overhead of conventional event-based streaming systems [57, 58], they still lack the columnar, causally-partitioned data path that underpins ORCA's analytical efficiency.

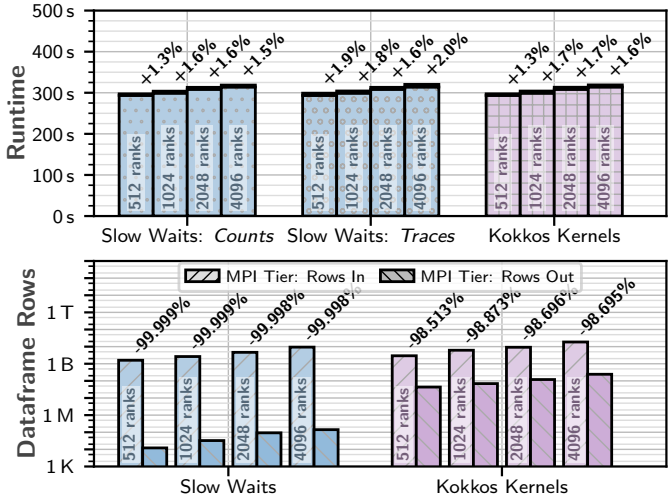


Fig. 9: Runtime overhead (top) and MPI tier data reduction (bottom) for in-situ analytics with operator pushdown. Flows range from monitoring communication outliers to compute load balance. Pushdown discards 98.5–99.999% of data at sub-2% overhead, despite operators executing on MPI ranks.

B. ORCA-Native In-Situ Workflows

While it can enable conventional tracing workflows, ORCA is fundamentally designed for in-situ analytics — directly materializing views from a running application. We now evaluate these capabilities, measuring the runtime overhead of operator pushdown, and its impact on data movement.

Takeaway: *In-situ analytics and operator pushdown reduce data movement by 98.5–99.999%, at sub-2% overhead.*

Workflows. Table I shows three flows of increasing complexity, designed for AMR codes. The first demonstrates a monitoring workflow: measuring the count of `MPI_Wait` exceeding 1 ms each timestep, while the second persists outlier events to Parquet. The third flow tracks compute load imbalance in AMR codes — it filters for known compute kernels, computes duration aggregates by rank, and streams details to Parquet and percentiles to DuckDB for real-time dashboards.

Observations. Fig. 9 shows the runtime overhead and the data reduction at `MPI` for the three flows.

- All workflows incur an overhead of $\leq 2\%$ across all scales despite executing operators on CPU ranks, illustrating the efficiency of Arrow-based in-situ analytics. Computation budget for pushdown only grows with modern GPU codes.
- Data movement from MPI ranks is reduced by $98.5\% - 99.999\%$. Designing observability around materialized views reduces data movement and improves feedback latency. Most telemetry is redundant for monitoring and debugging — a small fraction of aggregated and filtered data suffices to materialize the required views.
- The `MPI_Wait` flows do not sample—they perform continuous predicate evaluation for every single `MPI_Wait`, emitting only outliers. Comprehensive continuous observability of this type is impractical without pushdown.

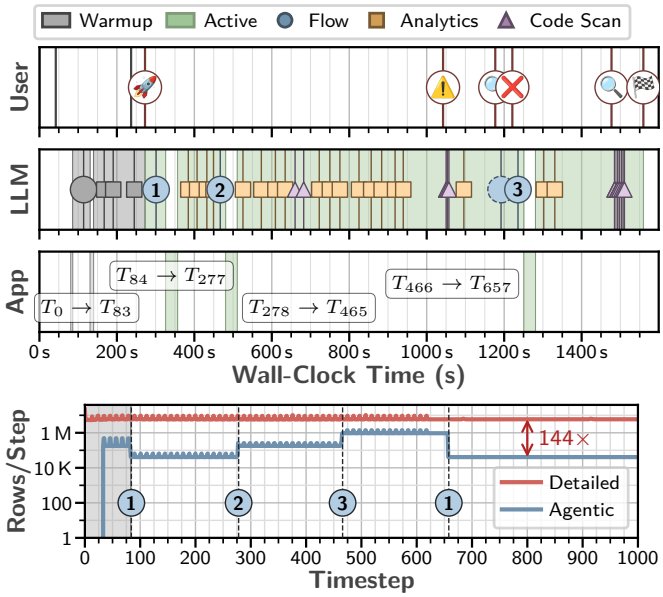


Fig. 10: Visual transcript of an agentic debugging session demonstrating closed-loop capabilities. An LLM agent is tasked with localizing the anomaly from Fig. 1 using high-level metrics, progressively collecting targeted telemetry to identify the source. The agent operates largely autonomously, producing the complete call path in 27 minutes. Reverting to baseline monitoring reduces data volume by 144×.

C. Closing the Loop: Agentic Debugging

While ORCA was designed for human operators, the emergence of LLM-based coding agents, capable of issuing commands and analyzing outputs, offers a way to validate the steerability paradigm. We task an LLM agent with localizing the 4096-rank AMR anomaly shown in Fig. 1, which we previously investigated [10].

Takeaway: *Steerable collection and in-situ analytics compress our months-long manual investigation into 27 minutes.*

Setup. Claude Code (Opus 4.5) [83] ran on the head node with access to the running application instance via ORCA, and a copy of the codebase. The agent was asked to monitor only collective statistics initially, collecting additional Kokkos/MPI telemetry only if outliers were detected. The LLM wrote and installed all ORCA flows and used Polars [80] on the head node for all analyses.

Timeline. Fig. 10 shows the debugging session: user and LLM interactions at top, per-timestep event volume at bottom. After a warmup phase, the LLM began the localization exercise at 272 s by monitoring collectives for outliers (①).

1) At 384 s, it analyzed collective percentiles, identified stragglers, and installed a flow for Kokkos events > 1 ms (②).
2) Between 510–1000 s, it used the Kokkos events to identify a `map.clear()` in AMR redistribution as the culprit.
3) At 1042 s, user rejected this as inadequate given the magnitude of the anomaly (50–100 ms). LLM then found a `MPI_Wait` in a buffer destructor triggered by the `clear()`.
4) At 1191 s, LLM proposed capturing all `MPI_Wait` events to confirm, user rejected and asked for a duration-based filter. LLM then installed a flow for `MPI_Waits` > 1 ms.
5) At 1335 s, telemetry confirmed waits as the root cause. LLM reported the complete call flow at 1605 s. Application then resumed with baseline collective monitoring (①).

Observations. The agent autonomously correlated high-level telemetry with code, requested targeted detail, and traced the root cause to where instrumentation ended — completing in 27 minutes an investigation that previously took us months. Reverting to baseline monitoring afterwards reduced data volume by 144×. User interventions were minor: rejecting an incomplete explanation and asking for basic filtering. While the exercise relied on pre-existing annotations in relevant regions, dynamic instrumentation [43, 73] would enable investigations to continue into the MPI and fabric layers. Beyond localization, the same anomaly signature can be deployed as an always-on metric, allowing regressions to be detected immediately.

VII. Discussion

While ORCA is evaluated primarily as an observability system, its architecture also suggests broader implications for BSP execution, ML training, streaming analytics, and scientific data management. We discuss these implications next, moving from synchronization semantics and ML training to general streaming analytics and scientific data management.

A. Synchronization as a Semantic Interface

While global synchronization makes BSP applications unusually sensitive to performance variability, it also provides unusually strong semantics for observability and control. Collectives enable materializing a consistent global snapshot of system state, while also acting as execution fences, creating natural boundaries for consistent intervention. Conventional observability tooling largely discards this structure, collecting telemetry as per-rank logs that must be correlated after the fact. ORCA instead models observability as analytics over timestep-consistent slices of application state, attaching asynchronously to the application heartbeat while preserving the same structure for offline telemetry.

B. Applicability to ML Training Workloads

Large-scale ML training is itself BSP, and a recent burst of observability systems addresses straggler localization [16, 17, 20], fault attribution [19, 21–23, 84], and training correctness [24] while reimplementing subsets of collection, detection and actuation. Table II compares these systems with ORCA and shows how these workflows can instead be expressed via composable operators over over a common substrate for aligned collection, analysis, and control. The same substrate need not stop at performance telemetry: Arrow–DLPack interoperability [37, 85] and GPU-native OLAP kernels [86] enable pushed-down analytics over model state itself, supporting workflows ranging from numerical stability to evolution of weights and gradients without exporting full tensors.

System	Diagnostic Task	Their Approach	ORCA Equivalent
Greyhound [17] *(Fail-slows)*	• Iteration boundary detection • Fail-slow detection • Straggler localization • Straggler cause disambiguation • Online interventions	• Autocorrelation over collectives • Track iteration times • Aggregate NCCL events (off-fabric) • Microbenchmarks • Tune micro-batch distribution or parallelism	• `PostTimestepAdvance` • Track iteration times • `GROUPBY swid` (on-fabric) • Microbenchmarks or trace events • Supported via control plane
Flare [84] *(Training anomalies)*	• Stall detection/disambiguation • Slowdown analysis	• Call stack analysis • Periodic metrics	• In-situ reductions • In-situ trace event aggregation
Holmes [16] *(Fail-slows)*	• Outlier collective detection • Localization	• Random forest model • Graph search	• Policy-dependent • Learn collective topology at bootstrap, `GROUPBY swid`, trace critical path in reverse
Mycroft [20] *(Training anomalies)*	• Straggler detection/localization	• Fine-grained trace NCCL events buffered locally, flushed to Kafka if heuristics trigger	• Aligned dataframes analyzed in-situ
TrainCheck [24] *(Training correctness)*	• Correctness invariants	• Offline trace analysis for invariant generation, online enforcement	• Timestep-aligned traces for offline generation, in-situ enforcement via OrcaFlow
Reveal [21] *(Training anomalies)*	• Anomaly detection and attribution	• Time-windowed metrics + unsupervised learning	• Workload-level trace events
Minder[#] [23] *(Fault diagnosis)*	• Fail-slow hardware	• Cluster-level metrics and anomaly detection	• Workload-level trace events
Aegis[#] [22] *(Fault diagnosis)*	• Fail-stop hardware	• Per-node NCCL counters pulled on timeouts	• Workload-level trace events
EROICA[#] [19] *(Training anomalies)*	• Anomaly detection and attribution	• Always-on iteration monitoring • Trigger-driven detailed capture • In-situ telemetry reductions	• Identical workflow, but workload-level

[#] *Minder, Aegis, and EROICA target multi-tenant environments. ORCA currently supports a single large workload.*

TABLE II: Diagnostic approaches in ML training observability and their ORCA equivalents.

ML training also exposes additional optimization opportunities through its multi-dimensional collective structure. Data (DP), tensor (TP), and expert parallelism (EP) form hierarchical collective domains [87, 88]. This structure enables topology-aware pushdowns: for example, a reduction scoped to a TP group can terminate deeper in the hierarchy than one spanning DP groups. Unlike the common interfaces provided by MPI [59] and NCCL [60] for individual collective domains, no standard interface exists for describing multi-dimensional parallel structure, requiring framework-specific integrations to exploit it.

C. ORCA as a Computational Model

ORCA's broader lesson is that in-situ analytics can be treated as a streaming ORCA can be understood as a general model for streaming and in-situ analytics, with observability as one application. It executes dataflows of reusable kernels over a tree-based overlay, with operators placed at runtime by a tier-aware planner. SQL is only a convenient frontend for constructing these dataflows from relational operators.

- ORCA instantiates a tree-based variant of MapReduce. MapReduce frameworks [52, 66] typically use a single tier of executors communicating using all-to-all shuffles. ORCA instead maps reducible computation onto its tree hierarchy. Shuffling naturally composes with this model as an intra-tier operation. Prior work [89, 90] demonstrates the viability of all-to-all shuffling for streaming reorganization on MPI ranks. A tier- and phase-aware planner can leverage both, turning placement into a planning decision—checkpoint phases may favor the aggregate resources of the MPI tier, while compute phases may favor auxiliary tiers to minimize application interference.
- Compared with in-situ coupling systems such as ADIOS [53], ORCA differs in three key ways. First, downstream computation is asynchronous while retaining timestep consistency, and need not execute as a second timestep-synchronous MPI application. Second, the simulation itself is an execution target, allowing operators to be pushed into application ranks to reduce data before movement. Third, a dedicated root controller has consistent authority over the overlay, enabling coordinated dataflow placement and updates as well as application steering.

Together, these observations suggest a path beyond today's separate in-situ pipelines for observability, data reorganization and indexing [89, 90], and visualization [91, 92]. These workloads can instead be expressed as dataflows over reusable kernel libraries shared across in-situ and post-hoc analyses. While prior work [93–95] suggests that even mesh-based workloads may admit a common algebra, defining one upfront is unnecessary: useful dataflows can be composed directly, with richer algebras and optimizers emerging as common operators and patterns develop.

D. OLAP Lessons for Scientific Data Management

The evolution of analytical data systems offers useful lessons for challenges in scientific data management. Modern lakehouses decouple query engines from data at rest, with both data and dataset metadata in open formats: Parquet for files, and Iceberg/Delta Lake [81, 82] for logical datasets. These table formats not only bundle related Parquet files, but also expose statistics and layout information that query planners can exploit to accelerate analytics. This metadata layer is a natural counterpart to the reusable-kernel model above: standardized metadata lets planners push common

analytical operators toward the data without coupling either data or computation to a particular engine. This separation closely resembles scientific workflows, where data outlives the simulation that produced it and is subsequently consumed by independent analysis tools.

Adopting the same separation could address long-standing fragmentation in scientific data management without requiring a particular storage substrate. The same logical dataset could span filesystems such as Lustre [96] and object stores from DAOS [97] to the cloud, while exposing common metadata to downstream analytics. Zarr [98] already moves in this direction for N-D arrays by separating logical array metadata and chunks from the storage backend. These dataset metadata layers can in turn be complemented by a filesystem-backed catalog [99]: a higher-level namespace for dataset discovery, governance, and lifecycle management. Combined with reusable dataflows, this offers a path toward a common scientific data platform rather than separate stacks for each stage of the data lifecycle.

E. Beyond the Relational Model

The obvious concern with applying OLAP tools to HPC is that scientific data and its decomposition are richer than the relational model. While genuine differences exist at the logical level, two observations suggest that they need not require separate physical and execution substrates:

- N-D arrays, tensors, and meshes ultimately reduce to linear buffers plus compact metadata describing properties such as shape, dimensionality, and layout. Their logical structure therefore need not dictate a different physical container.
- While scientific partitioning may encode complex spatial or topological structure that does not map cleanly to hash or range partitioning, the simulation has already decomposed the problem domain to execute at scale. In-situ analytics can inherit that decomposition as their native partitioning, requiring shuffle or repartitioning only when an analysis needs a different organization.

VIII. Related Work

Tool Infrastructure. MRNet [55] introduced TBONs for scalable distributed tool infrastructure, but its compiled C++ filter model predates modern analytical interfaces. eBPF and DynInst [43, 73] provide complementary mechanisms for dynamically instrumenting symbols at runtime. SmartNICs are also being explored as execution targets for Arrow-based data pipelines [100, 101]. A declarative engine like ORCA's could offload supported plan fragments to such devices transparently.

Trace Analytics. Fay [102] and Snicket [103] compile declarative queries into distributed probes that filter and aggregate telemetry at source, while Pivot Tracing [104] adds happened-before joins to preserve causal relationships across distributed events. ORCA's corresponding organizing primitive is timestep partitioning induced directly by BSP synchronization.

DiTing [105] adopts a tiered, lakehouse-style architecture after hitting ingestion limits with a centralized OLAP deployment, and a custom columnar format to avoid Parquet footer overheads for extremely wide metrics, a limitation substantially addressed by recent Parquet metadata optimizations [106]. Its broader production experience nevertheless reinforces ORCA's architectural direction of pushing analytics toward telemetry sources while using higher tiers for aggregation and persistence.

Hatchet [63] postprocesses traces into graph-based columnar dataframes. Such structure composes naturally as a secondary index, while $(\text{timestep}, \text{rank})$ is the more useful primary key for BSP, directly enabling timestep dataframes and in-situ analytics. Recent work also explores accelerating telemetry queries using approximate proxy statistics [107]. Such summaries fit naturally into declarative query plans as approximate replacements for exact operators.

Computational Steering. Computational steering—interactive control of running simulations—dates to the early 1990s [108, 109] and continues through modern in-situ visualization frameworks [91, 92]. ORCA extends this lineage with OLAP-style analytics and timestep-consistent control.

IX. Conclusions

BSP workloads require always-on observability as a production capability, capable of materializing actionable views and enabling rapid intervention. Traditional tracing works against this: the post-hoc paradigm incentivizes overcollection, requires large analytics clusters, and delays feedback. ORCA inverts this by pushing analytics to the source. Timestep dataframes and columnar representation enable multi-tier SQL analytics with operator pushdown—sub-2% overhead, up to 99.999% data reduction at source, agentic anomaly localization in minutes rather than months—all while contending for cluster resources with a live CPU code. GPU workloads and modern fabric QoS capabilities only increase the headroom available for such pushed-down workflows. Views materialize on demand, so only the stories—not the raw data—exit the fabric. The hero run need not be followed by a hero analysis.

Acknowledgments

We thank the anonymous reviewers of our paper for their valuable comments on improving the manuscript. We also thank Vyas Sekar, Milind Srivastava, Theo Benson, Lucas Castanheira, Phil Carns, and Rob Ross for insightful discussions that helped shape this work. This manuscript has been approved for unlimited release and has been assigned LA-UR-26-27392. We also thank the member companies of the PDL Consortium (Bloomberg LP, Everpure, Google, Jane Street, LayerZero Labs, Meta, Microsoft Research, Oracle Corporation, Salesforce, Uber, and Western Digital) for their interest, insights, feedback, and support. The authors used ChatGPT and Claude throughout this work to assist with development, analysis, and writing.

References

[1] Leslie G. Valiant. "A Bridging Model for Parallel Computation". In: *Commun. ACM* 33.8 (Aug. 1, 1990), pp. 103–111. ISSN: 0001-0782. DOI: 10.1145/79173.79181. URL: https://dl.acm.org/doi/10.1145/79173.79181.

[2] Omkar Salpekar et al. *Training LLMs with Fault Tolerant HSDP on 100,000 GPUs*. Version 1. Jan. 30, 2026. DOI: 10.48550/arXiv.2602.00277. arXiv: 2602.00277 [cs]. URL: http://arxiv.org/abs/2602.00277. Pre-published.

[3] Sambit Das et al. "Large-Scale Materials Modeling at Quantum Accuracy: Ab Initio Simulations of Quasicrystals and Interacting Extended Defects in Metallic Alloys". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '23: International Conference for High Performance Computing, Networking, Storage and Analysis. Denver CO USA: ACM, Nov. 12, 2023, pp. 1–12. ISBN: 979-8-4007-0109-2. DOI: 10.1145/3581784.3627037. URL: https://dl.acm.org/doi/10.1145/3581784.3627037.

[4] Ryan Stocks et al. "Breaking the Million-Electron and 1 EFLOP/s Barriers: Biomolecular-Scale Ab Initio Molecular Dynamics Using MP2 Potentials". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. SC '24: International Conference for High Performance Computing, Networking, Storage and Analysis. SC '24. Atlanta, GA, USA: IEEE Press, Nov. 17, 2024, pp. 1–12. ISBN: 979-8-3503-5291-7. DOI: 10.1109/SC41406.2024.00015. URL: https://dl.acm.org/doi/10.1109/SC41406.2024.00015.

[5] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q". In: *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*. SC '03: International Conference for High Performance Computing, Networking, Storage and Analysis. Phoenix AZ USA: ACM, Nov. 15, 2003, p. 55. DOI: 10.1145/1048935.1050204. URL: https://dl.acm.org/doi/10.1145/1048935.1050204.

[6] Bilge Acun, Phil Miller, and Laxmikant V. Kale. "Variation Among Processors Under Turbo Boost in HPC Systems". In: *Proceedings of the 2016 International Conference on Supercomputing*. ICS '16: 2016 International Conference on Supercomputing. Istanbul Turkey: ACM, June 2016, pp. 1–12. ISBN: 978-1-4503-4361-9. DOI: 10.1145/2925426.2926289. URL: https://dl.acm.org/doi/10.1145/2925426.2926289.

[7] Haryadi S. Gunawi et al. "Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems". In: *ACM Transactions on Storage* 14.3 (Aug. 31, 2018), pp. 1–26. ISSN: 1553-3077, 1553-3093. DOI: 10.1145/3242086. URL: https://dl.acm.org/doi/10.1145/3242086.

[8] Abhinav Bhatele et al. "The Case of Performance Variability on Dragonfly-based Systems". In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS). May 2020, pp. 896–905. DOI: 10.1109/IPDPS47924.2020.00096. URL: https://ieeexplore.ieee.org/document/9139880.

[9] Brian Van Straalen et al. "Scalability Challenges for Massively Parallel AMR Applications". In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. Distributed Processing (IPDPS). Rome, Italy: IEEE, May 2009, pp. 1–12. ISBN: 978-1-4244-3751-1. DOI: 10.1109/IPDPS.2009.5161014. URL: http://ieeexplore.ieee.org/document/5161014/.

[10] Ankush Jain et al. "Lessons from Profiling and Optimizing Placement in AMR Codes". In: *2025 IEEE International Conference on Cluster Computing (CLUSTER)*. Edinburgh, UK: IEEE, Sept. 1–30, 2025. DOI: 10.1109/cluster59342.2025.11186462.

[11] Xinhao Kong et al. "Collie: Finding Performance Anomalies in RDMA Subsystems". In: 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22). 2022, pp. 287–305. ISBN: 978-1-939133-27-4. URL: https://www.usenix.org/conference/nsdi22/presentation/kong.

[12] Adithya Gangidi et al. "RDMA over Ethernet for Distributed Training at Meta Scale". In: *Proceedings of the ACM SIGCOMM 2024 Conference*. ACM SIGCOMM '24: ACM SIGCOMM 2024 Conference. Sydney NSW Australia: ACM, Aug. 4, 2024, pp. 57–70. ISBN: 979-8-4007-0614-1. DOI: 10.1145/3651890.3672233. URL: https://dl.acm.org/doi/10.1145/3651890.3672233.

[13] Xin You et al. "GVARP: Detecting Performance Variance on Large-Scale Heterogeneous Systems". In: *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. SC24: International Conference for High Performance Computing, Networking, Storage and Analysis. Nov. 2024, pp. 1–16. DOI: 10.1109/SC41406.2024.00063. URL: https://ieeexplore.ieee.org/document/10793232/.

[14] Ruiming Lu et al. "PERSEUS: A Fail-Slow Detection Framework for Cloud Storage Systems". In: *Proceedings of the 21st USENIX Conference on File and Storage Technologies*. FAST'23. USA: USENIX Association, Feb. 21, 2023, pp. 49–63. ISBN: 978-1-939133-32-8.

[15] Orcun Yildiz et al. "On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems". In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS). May 2016, pp. 750–759. DOI: 10.1109/IPDPS.2016.50. URL: https://ieeexplore.ieee.org/document/7516071.

[16] Zhiyi Yao et al. "Holmes: Localizing Irregularities in LLM Training with Mega-scale GPU Clusters". In: 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25). 2025, pp. 523–540. ISBN: 978-1-939133-46-5. URL: https://www.usenix.org/conference/nsdi25/presentation/yao.

[17] Tianyuan Wu et al. "GREYHOUND: Hunting Fail-Slows in Hybrid-Parallel Training at Scale". In: 2025 USENIX Annual Technical Conference (USENIX ATC 25). 2025, pp. 731–747. ISBN: 978-1-939133-48-9. URL: https://www.usenix.org/conference/atc25/presentation/wu-tianyuan.

[18] Weihao Cui et al. *XPUTimer: Anomaly Diagnostics for Divergent LLM Training in GPU Clusters of Thousand-Plus Scale*. Version 1. Feb. 8, 2025. DOI: 10.48550/arXiv.2502.05413. arXiv: 2502.05413. URL: http://arxiv.org/abs/2502.05413. Pre-published.

[19] Yu Guan et al. "EROICA: Online Performance Troubleshooting for Large-scale Model Training". In: 23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26). 2026, pp. 1113–1130. ISBN: 978-1-939133-54-0. URL: https://www.usenix.org/conference/nsdi26/presentation/guan-yu.

[20] Yangtao Deng et al. "Mycroft: Tracing Dependencies in Collective Communication Towards Reliable LLM Training". In: *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. SOSP '25. New York, NY, USA: Association for Computing Machinery, Oct. 12, 2025, pp. 254–269. ISBN: 979-8-4007-1870-0. DOI: 10.1145/3731569.3764848. URL: https://doi.org/10.1145/3731569.3764848.

[21] Ziji Chen et al. *Detecting Anomalies in Machine Learning Infrastructure via Hardware Telemetry*. Oct. 31, 2025. DOI: 10.48550/arXiv.2510.26008. arXiv: 2510.26008 [cs]. URL: http://arxiv.org/abs/2510.26008. Pre-published.

[22] Jianbo Dong et al. "Evolution of Aegis: Fault Diagnosis for AI Model Training Service in Production". In: 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25). 2025, pp. 865–881. ISBN: 978-1-939133-46-5. URL: https://www.usenix.org/conference/nsdi25/presentation/dong.

[23] Yangtao Deng et al. "Minder: Faulty Machine Detection for Large-scale Distributed Model Training". In: 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25). USENIX Association, 2025, pp. 505–521. ISBN: 978-1-939133-46-5. URL: https://www.usenix.org/conference/nsdi25/presentation/deng.

[24] Yuxuan Jiang et al. "Training with Confidence: Catching Silent Errors in Deep Learning Training with Automated Proactive Checks". In: 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25). 2025, pp. 313–329. ISBN: 978-1-939133-47-2. URL: https://www.usenix.org/conference/osdi25/presentation/jiang.

[25] *Prometheus - Monitoring System & Time Series Database*. URL: https://prometheus.io/.

[26] Benjamin Schwaller. *Integrated System and Application Continuous Performance Monitoring and Analysis Capability (Final)*. SAND2021-11954, 1822583, 700207. Sept. 1, 2021, SAND2021–11954, 1822583, 700207. URL: https://www.osti.gov/servlets/purl/1822583/.

[27] Sameer S. Shende and Allen D. Malony. "The Tau Parallel Performance System". In: *Int. J. High Perform. Comput. Appl.* 20.2 (May 1, 2006), pp. 287–311. ISSN: 1094-3420. DOI: 10.1177/1094342006064482. URL: https://doi.org/10.1177/1094342006064482.

[28] Andreas Knüpfer et al. "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir". In: *Tools for High Performance Computing 2011*. Ed. by Holger Brunst et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 79–91. ISBN: 978-3-642-31476-6. DOI: 10.1007/978-3-642-31476-6_7. URL: http://link.springer.com/10.1007/978-3-642-31476-6_7.

[29] *Pytorch Kineto*. pytorch, Dec. 4, 2025. URL: https://github.com/pytorch/kineto.

[30] Hariharan Devarajan et al. "DFTracer: An Analysis-Friendly Data Flow Tracer for AI-Driven Workflows". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. SC '24. Atlanta, GA, USA: IEEE Press, Nov. 17, 2024, pp. 1–24. ISBN: 979-8-3503-5291-7. DOI: 10.1109/SC41406.2024.00023. URL: https://dl.acm.org/doi/10.1109/SC41406.2024.00023.

[31] David Boehme et al. "Caliper: Performance Introspection for HPC Software Stacks". In: *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '16: The International Conference for High Performance Computing, Networking, Storage and Analysis. Nov. 2016, pp. 550–560. DOI: 10.1109/SC.2016.46. URL: https://ieeexplore.ieee.org/abstract/document/7877125.

[32] Izzet Yildirim et al. "IOMax: Maximizing Out-of-Core I/O Analysis Performance on HPC Systems". In: *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. SC-W '23. New York, NY, USA: Association for Computing Machinery, Nov. 12, 2023, pp. 1209–1215. ISBN: 979-8-4007-0785-8. DOI: 10.1145/3624062.3624191. URL: https://dl.acm.org/doi/10.1145/3624062.3624191.

[33] Paul Barham et al. "Using Magpie for Request Extraction and Workload Modelling". In: 6th Symposium on Operating Systems Design & Implementation (OSDI 04). 2004. URL: https://www.usenix.org/conference/osdi-04/using-magpie-request-extraction-and-workload-modelling.

[34] Rodrigo Fonseca et al. "X-Trace: A Pervasive Network Tracing Framework". In: 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07). 2007. URL: https://www.usenix.org/conference/nsdi-07/x-trace-pervasive-network-tracing-framework.

[35] Benjamin H. Sigelman et al. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Google, Inc., 2010. URL: https://research.google.com/archive/papers/dapper-2010-1.pdf.

[36] Dorian C. Arnold, Gary D. Pack, and Barton P. Miller. "Tree-Based Overlay Networks for Scalable Applications". In: *Proceedings of the 20th International Conference on Parallel and Distributed Processing*. IPDPS'06. USA: IEEE Computer Society, Apr. 25, 2006, p. 223. ISBN: 978-1-4244-0054-6. DOI: 10.1109/IPDPS.2006.1639493.

[37] *Apache Arrow*. Apache Arrow. URL: https://arrow.apache.org/.

[38] *Apache Parquet*. Apache Parquet. URL: https://parquet.apache.org/docs/overview/.

[39] Ankush Jain, Sheng Jiang, and Chuck Cranor. *ORCA: Observability with Real-Time Control and Aggregation*. 2026. URL: https://github.com/pdlfs/orca-umbrella.

[40] Charity Majors. *Observability: What's in a Name?* Honeycomb. Aug. 22, 2017. URL: https://www.honeycomb.io/blog/observability-whats-in-a-name.

[41] *Observability Primer*. OpenTelemetry. URL: https://opentelemetry.io/docs/concepts/observability-primer/.

[42] Cindy Sridharan. *Distributed Systems Observability*. ISBN: 978-1-4920-3343-1. URL: https://www.oreilly.com/library/view/distributed-systems-observability/9781492033431/.

[43] *What Is eBPF? An Introduction and Deep Dive into the eBPF Technology*. URL: https://ebpf.io/what-is-ebpf/.

[44] Anshu Dubey et al. "A Survey of High Level Frameworks in Block-Structured Adaptive Mesh Refinement Packages". In: *Journal of Parallel and Distributed Computing* 74.12 (Dec. 2014), pp. 3217–3227. ISSN: 07437315. DOI: 10.1016/j.jpdc.2014.07.001. URL: https://linkinghub.elsevier.com/retrieve/pii/S0743731514001178.

[45] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

[46] *Facebookincubator/Dynolog*. Meta Incubator, Dec. 4, 2025. URL: https://github.com/facebookincubator/dynolog.

[47] *Facebookresearch/HolisticTraceAnalysis*. Meta Research, Dec. 8, 2025. URL: https://github.com/facebookresearch/HolisticTraceAnalysis.

[48] *Pandas - Python Data Analysis Library*. URL: https://pandas.pydata.org/.

[49] Eschweiler Dominic et al. "Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries". In: *Advances in Parallel Computing*. IOS Press, 2012. DOI: 10.3233/978-1-61499-041-3-481. URL: https://www.medra.org/servlet/aliasResolver?alias=iospressISSNISBN&issn=0927-5452&volume=22&spage=481.

[50] *Catapult - The Trace-Event File Format*. URL: https://chromium.googlesource.com/catapult/+/HEAD/docs/trace-event-format.md.

[51] Abhinav Bhatele et al. *Pipit: Scripting the Analysis of Parallel Execution Traces*. Version 2. May 14, 2024. DOI: 10.48550/arXiv.2306.11177. arXiv: 2306.11177 [cs]. URL: http://arxiv.org/abs/2306.11177. Pre-published.

[52] Matthew Rocklin. "Dask: Parallel Computation with Blocked Algorithms and Task Scheduling". In: Python in Science Conference. Austin, Texas, 2015, pp. 126–132. DOI: 10.25080/Majora-7b98e3ed-013. URL: https://doi.curvenote.com/10.25080/Majora-7b98e3ed-013.

[53] William F. Godoy et al. "ADIOS 2: The Adaptable Input Output System. A Framework for High-Performance Data Management". In: *SoftwareX* 12 (July 1, 2020). ISSN: 2352-7110. DOI: 10.1016/j.softx.2020.100561. URL: https://www.softxjournal.com/article/S2352-7110(19)30256-0/fulltext.

[54] Greg Eisenhauer et al. "Streaming Data in HPC Workflows Using ADIOS". In: *Proceedings of the Cray User Group*. CUG '24. New York, NY, USA: Association for Computing Machinery, Aug. 15, 2025, pp. 31–43. ISBN: 979-8-4007-1328-6. DOI: 10.1145/3725789.3725793. URL: https://dl.acm.org/doi/10.1145/3725789.3725793.

[55] P.C. Roth, D.C. Arnold, and B.P. Miller. "MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools". In: *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*. SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing. Nov. 2003, pp. 21–21. DOI: 10.1145/1048935.1050172. URL: https://ieeexplore.ieee.org/abstract/document/1592924.

[56] William R. Williams et al. "Dyninst and MRNet: Foundational Infrastructure for Parallel Tools". In: *Tools for High Performance Computing 2015*. Ed. by Andreas Knüpfer et al. Cham: Springer International Publishing, 2016, pp. 1–16. ISBN: 978-3-319-39589-0. DOI: 10.1007/978-3-319-39589-0_1.

[57] Paris Carbone et al. "Apache Flink™: Stream and Batch Processing in a Single Engine". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 38.4 (2015), pp. 28–38. URL: https://asterios.katsifodimos.com/assets/publications/flink-deb.pdf.

[58] Jay Kreps, Neha Narkhede, and Jun Rao. "Kafka: A Distributed Messaging System for Log Processing". In: *Proceedings of the 6th International Workshop on Networking Meets Databases*. NetDB '11. Athens, Greece: ACM, June 12, 2011, pp. 1–7. ISBN: 978-1-4503-0652-2. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2017/09/Kafka.pdf.

[59] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.1*. Message Passing Interface Forum, Nov. 2023. URL: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf.

[60] *NVIDIA Collective Communications Library (NCCL)*. NVIDIA Developer. URL: https://developer.nvidia.com/nccl.

[61] Andrew Lamb et al. "Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine". In: *Companion of the 2024 International Conference on Management of Data*. SIGMOD/PODS '24: International Conference on Management of Data. Santiago AA Chile: ACM, June 9, 2024, pp. 5–17. ISBN: 979-8-4007-0422-2. DOI: 10.1145/3626246.3653368. URL: https://dl.acm.org/doi/10.1145/3626246.3653368.

[62] Mark Raasveldt and Hannes Mühleisen. "DuckDB: An Embeddable Analytical Database". In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD/PODS '19: International Conference on Management of Data. Amsterdam Netherlands: ACM, June 25, 2019, pp. 1981–1984. ISBN: 978-1-4503-5643-5. DOI: 10.1145/3299869.3320212. URL: https://dl.acm.org/doi/10.1145/3299869.3320212.

[63] Abhinav Bhatele, Stephanie Brink, and Todd Gamblin. "Hatchet: Pruning the Overgrowth in Parallel Profiles". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '19: The International Conference for High Performance Computing, Networking, Storage, and Analysis. Denver Colorado: ACM, Nov. 17, 2019, pp. 1–21. DOI: 10.1145/3295500.3356219. URL: https://dl.acm.org/doi/10.1145/3295500.3356219.

[64] Daniele De Sensi et al. "An In-Depth Analysis of the Slingshot Interconnect". In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1 vols. New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 1–14. DOI: 10.1109/SC41405.2020.00039. URL: https://doi.org/10.1109/SC41405.2020.00039.

[65] *Grafana | Query, Visualize, Alerting Observability Platform*. Grafana Labs. URL: https://grafana.com/grafana/.

[66] Matei Zaharia et al. "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. USA: USENIX Association, June 22, 2010, p. 10. URL: https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets.

[67] Shajulin Benedict, Ventsislav Petkov, and Michael Gerndt. "PERISCOPE: An Online-Based Distributed Performance Analysis Tool". In: *Tools for High Performance Computing 2009*. Ed. by Matthias S. Müller et al. Berlin, Heidelberg: Springer, 2010, pp. 1–16. ISBN: 978-3-642-11261-4. DOI: 10.1007/978-3-642-11261-4_1. URL: https://doi.org/10.1007/978-3-642-11261-4_1.

[68] Fred B. Schneider. "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial". In: *ACM Computing Surveys* 22.4 (Dec. 1990), pp. 299–319. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/98163.98167. URL: https://dl.acm.org/doi/10.1145/98163.98167.

[69] Jerome Soumagne et al. "Mercury: Enabling Remote Procedure Call for High-Performance Computing". In: *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. 2013 IEEE International Conference on Cluster Computing (CLUSTER). Indianapolis, IN, USA: IEEE, Sept. 2013, pp. 1–8. ISBN: 978-1-4799-0898-1. DOI: 10.1109/CLUSTER.2013.6702617. URL: http://ieeexplore.ieee.org/document/6702617/.

[70] Paul Grun et al. "A Brief Introduction to the OpenFabrics Interfaces - A New Network API for Maximizing High Performance Application Efficiency". In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects. Aug. 2015, pp. 34–39. DOI: 10.1109/HOTI.2015.19. URL: https://ieeexplore.ieee.org/document/7312664.

[71] Pavel Shamis et al. "UCX: An Open Source Framework for HPC Network APIs and Beyond". In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects. Aug. 2015, pp. 40–43. DOI: 10.1109/HOTI.2015.13. URL: https://ieeexplore.ieee.org/document/7312665.

[72] Christian R. Trott et al. "Kokkos 3: Programming Model Extensions for the Exascale Era". In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (Apr. 2022), pp. 805–817. ISSN: 1558-2183. DOI: 10.1109/TPDS.2021.3097283. URL: https://ieeexplore.ieee.org/document/9485033.

[73] Andrew R. Bernat and Barton P. Miller. "Anywhere, Any-Time Binary Instrumentation". In: *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*. PASTE '11. New York, NY, USA: Association for Computing Machinery, Sept. 5, 2011, pp. 9–16.

ISBN: 978-1-4503-0849-6. DOI: 10.1145/2024569.2024572. URL: https://dl.acm.org/doi/10.1145/2024569.2024572.

[74] Leonid Ivanovich Sedov. *Similarity and Dimensional Methods in Mechanics*. CRC Press, 1959. DOI: 10.1201/9780203739730. URL: http://archive.org/details/l-i-sedov.-similarity-and-dimensional-methods-in-mechanics.

[75] Philipp Grete et al. "Parthenon—a Performance Portable Block-Structured Adaptive Mesh Refinement Framework". In: *The International Journal of High Performance Computing Applications* 37.5 (Sept. 1, 2023), pp. 465–486. ISSN: 1094-3420. DOI: 10.1177/10943420221143775. URL: https://doi.org/10.1177/10943420221143775.

[76] Brandon Barker et al. *Phoebus: Performance Portable GRRMHD for Relativistic Astrophysics*. Version 2. Oct. 15, 2024. DOI: 10.48550/arXiv.2410.09146. arXiv: 2410.09146 [astro-ph]. URL: http://arxiv.org/abs/2410.09146. Pre-published.

[77] Dhabaleswar Kumar Panda et al. "The MVAPICH Project: Transforming Research into High-Performance MPI Library for HPC Community". In: *Journal of Computational Science*. Case Studies in Translational Computer Science 52 (May 1, 2021), p. 101208. ISSN: 1877-7503. DOI: 10.1016/j.jocs.2020.101208. URL: https://www.sciencedirect.com/science/article/pii/S1877750320305093.

[78] *PSM: Performance Scaled Messaging*. Intel® Corporation, Dec. 5, 2025. URL: https://github.com/intel/psm.

[79] *Snappy: A Fast Compressor/Decompressor*. Google, Mar. 29, 2026. URL: https://github.com/google/snappy.

[80] *Polars*. URL: https://www.pola.rs/.

[81] *Apache Iceberg - Apache Iceberg™*. URL: https://iceberg.apache.org/.

[82] Michael Armbrust et al. "Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores". In: *Proceedings of the VLDB Endowment* 13.12 (Aug. 2020), pp. 3411–3424. ISSN: 2150-8097. DOI: 10.14778/3415478.3415560. URL: https://dl.acm.org/doi/10.14778/3415478.3415560.

[83] *Claude Code Overview*. Claude Code Docs. URL: https://code.claude.com/docs/en/overview.

[84] Weihao Cui et al. "Flare: Anomaly Diagnostics for Divergent LLM Training in GPU Clusters of Thousand-Plus Scale". In: *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. 23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26). 2026. URL: https://www.usenix.org/conference/nsdi26/presentation/cui.

[85] *The DLPack Protocol — Apache Arrow V22.0.0*. URL: https://arrow.apache.org/docs/python/dlpack.html.

[86] RAPIDS cuDF Developers. *cuDF: A GPU-Accelerated DataFrame Library for Tabular Data Processing*. 2026. URL: https://github.com/NVIDIA/cudf.

[87] Mohammad Shoeybi et al. *Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism*. Mar. 13, 2020. DOI: 10.48550/arXiv.1909.08053. arXiv: 1909.08053 [cs]. URL: http://arxiv.org/abs/1909.08053. Pre-published.

[88] *DeepSpeed: Extreme Speed and Scale for DL Training*. DeepSpeed. URL: https://www.deepspeed.ai/.

[89] Qing Zheng et al. "Scaling Embedded In-Situ Indexing with DeltaFS". In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. Nov. 2018, pp. 30–44. DOI: 10.1109/SC.2018.00006. URL: https://ieeexplore.ieee.org/document/8665820.

[90] Ankush Jain et al. "CARP: Range Query-Optimized Indexing for Streaming Data". In: *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. SC24: International Conference for High Performance Computing, Networking, Storage and Analysis. Nov. 2024, pp. 1–19. DOI: 10.1109/SC41406.2024.00093. URL: https://ieeexplore.ieee.org/document/10793150.

[91] Matthew Larsen et al. "Ascent: A Flyweight In Situ Library for Exascale Simulations". In: *In Situ Visualization for Computational Science*. Ed. by Hank Childs, Janine C. Bennett, and Christoph Garth. Cham: Springer International Publishing, 2022, pp. 255–279. ISBN: 978-3-030-81627-8. DOI: 10.1007/978-3-030-81627-8_12. URL: https://doi.org/10.1007/978-3-030-81627-8_12.

[92] Utkarsh Ayachit et al. "ParaView Catalyst: Enabling In Situ Data Analysis and Visualization". In: *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*. SC15: The International Conference for High Performance Computing, Networking, Storage and Analysis. Austin TX USA: ACM, Nov. 15, 2015, pp. 25–29. ISBN: 978-1-4503-4003-8. DOI: 10.1145/2828612.2828624. URL: https://dl.acm.org/doi/10.1145/2828612.2828624.

[93] M. Kersten et al. "SciQL, a Query Language for Science Applications". In: *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*. AD '11. New York, NY, USA: Association for Computing Machinery, Mar. 25, 2011, pp. 1–12. ISBN: 978-1-4503-0614-0. DOI: 10.1145/1966895.1966896. URL: https://dl.acm.org/doi/10.1145/1966895.1966896.

[94] Byung S Lee and Ron Musick. "MeshSQL: The Query Language for Simulation Mesh Data". In: *Information Sciences* 159.3 (Feb. 15, 2004), pp. 177–202. ISSN: 0020-0255. DOI: 10.1016/j.ins.2003.02.002. URL: https://www.sciencedirect.com/science/article/pii/S0020025503001981.

[95] Alireza Rezaei Mahdiraji. "Toward Unstructured Mesh Algebra and Query Language". In: *Proceedings of the 2014 SIGMOD PhD Symposium*. SIGMOD'14 PhD Symposium. New York, NY, USA: Association for Computing Machinery, June 18, 2014, pp. 16–20. ISBN: 978-1-4503-2924-8. DOI: 10.1145/2602622.2602626. URL: https://dl.acm.org/doi/10.1145/2602622.2602626.

[96] Peter J Braam and Philip Schwan. "Lustre: The Intergalactic File System". In: *Ottawa Linux Symposium*. Vol. 8. 11. 2002, pp. 3429–3441.

[97] Zhen Liang et al. "DAOS: A Scale-Out High Performance Storage Stack for Storage Class Memory". In: *Supercomputing Frontiers*. Ed. by Dhabaleswar K. Panda. Cham: Springer International Publishing, 2020, pp. 40–54. ISBN: 978-3-030-48842-0. DOI: 10.1007/978-3-030-48842-0_3.

[98] *Zarr*. Zarr. URL: https://zarr.dev/.

[99] *HadoopCatalog*. URL: https://iceberg.apache.org/javadoc/latest/org/apache/iceberg/hadoop/HadoopCatalog.html.

[100] Craig Ulmer et al. "Extending Composable Data Services into SmartNICs". In: *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). May 2023, pp. 953–959. DOI: 10.1109/IPDPSW59300.2023.00158. URL: https://ieeexplore.ieee.org/document/10196516.

[101] Jianshen Liu et al. *Processing Particle Data Flows with SmartNICs*. Oct. 13, 2022. arXiv: 2210.06827 [cs]. URL: http://arxiv.org/abs/2210.06827. Pre-published.

[102] Úlfar Erlingsson et al. "Fay: Extensible Distributed Tracing from Kernels to Clusters". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11: ACM SIGOPS 23nd Symposium on Operating Systems Principles. Cascais Portugal: ACM, Oct. 23, 2011, pp. 311–326. ISBN: 978-1-4503-0977-6. DOI: 10.1145/

2043556.2043585. URL: https://dl.acm.org/doi/10.1145/2043556.2043585.

[103] Jessica Berg et al. "Snicket: Query-Driven Distributed Tracing". In: *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*. HotNets '21: The 20th ACM Workshop on Hot Topics in Networks. Virtual Event United Kingdom: ACM, Nov. 10, 2021, pp. 206–212. ISBN: 978-1-4503-9087-3. DOI: 10.1145/3484266.3487393. URL: https://dl.acm.org/doi/10.1145/3484266.3487393.

[104] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. "Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. New York, NY, USA: Association for Computing Machinery, Oct. 4, 2015, pp. 378–393. ISBN: 978-1-4503-3834-9. DOI: 10.1145/2815400.2815415. URL: https://dl.acm.org/doi/10.1145/2815400.2815415.

[105] Zhenyu Ren et al. "All Along the Watchtower: Achieving the Trinity of Observability in Cloud with DiTing". In: 20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26). 2026, pp. 2501–2514. ISBN: 978-1-939133-55-7. URL: https://www.usenix.org/conference/osdi26/presentation/ren.

[106] Andrew Lamb. *3x-9x Faster Apache Parquet Footer Metadata Using a Custom Thrift Parser in Rust*. Apache Arrow. Oct. 23, 2025. URL: https://arrow.apache.org/blog/2025/10/23/rust-parquet-metadata/.

[107] Milind Srivastava et al. *ASAP: Reimagining the Data Lifecycle Using Application Semantic-Aware Processing*. Aug. 14, 2026. DOI: 10.1184/R1/33179324.v1. URL: https://kilthub.cmu.edu/articles/preprint/ASAP_Reimagining_the_Data_Lifecycle_using_Application_Semantic-Aware_Processing/33179324/1. Pre-published.

[108] M. Miller et al. "Simulation Steering with SCIRun in a Distributed Environment". In: *Proceedings. The Seventh International Symposium on High Performance Distributed Computing (Cat. No.98TB100244)*. . The Seventh International Symposium on High Performance Distributed Computing (Cat. No.98TB100244). July 1998, pp. 364–365. DOI: 10.1109/HPDC.1998.710032. URL: https://ieeexplore.ieee.org/document/710032.

[109] G.A. Geist, James Arthur Kohl, and Philip M. Papadopoulos. "CUMULVS: Providing Fault Tolerance, Visualization, and Steering of Parallel Applications". In: *The International Journal of Supercomputer Applications and High Performance Computing* 11.3 (Sept. 1997), pp. 224–235. ISSN: 1078-3482. DOI: 10.1177/109434209701100305. URL: https://journals.sagepub.com/doi/10.1177/109434209701100305.