Data Consistency Challenges in AI Applications

Ken Birman
Cornell University
Ithaca, New York, USA

Jamal Hashim
Cornell University
Ithaca, New York, USA

Edward Tremel
Augusta University
Augusta, Georgia, USA

Yuting Yang
Cornell University
Ithaca, New York, USA

Abstract

Although the AI and ML communities frequently discuss semantic consistency issues, present-generation systems rarely work with dynamically evolving data and hence have not grappled directly with data consistency. However, this may soon change. AI applications that support real-world data streams will require new options, particularly if they execute as distributed pipelines of ML components. The Vortex system offers two modes of operation: serializable snapshot isolation for non-temporal data, and temporally-indexed querying for real-time tasks. We justify this design, review our architecture, and demonstrate that consistency properties emerging from our system benefit unmodified ML logic executing in an environment abstracted away from Vortex by multiple layers of infrastructure. Vortex requires minimal changes to ML code (similar to other serving platforms) and significantly outperforms baseline systems. The work demonstrates that stronger consistency is valuable for ML, practical, and need not be costly.

1 Introduction

Data consistency models are explanatory: they enable us to create complex distributed systems that support concurrent data updates but to explain the resulting behavior in an intuitive, mathematically rigorous way. The underlying models trace their lineage to state machine replication [17, 22] and linearizability [13]. Prevailing AI deployment models lack explicit data consistency models, but this is not to say that they struggle with platform-caused inconsistencies. Instead, the most widely used architectures push consistency questions into storage frameworks. The AI components require little more than preservation of FIFO event orderings, a means of creating consistent checkpoints, and rollback. Moreover, modern AIs are often monolithic, running in a single container deployed on a single server, which simplifies the corresponding logic. Training data is static, and consistency within a training framework generally reduces to a simple read-what-you-wrote policy. When deployed, many AIs perform database or vector database queries during inference and knowledge retrieval, but the underlying databases are typically updated offline, and consequently do not change during inference.

Figure 1. A physician's AI assistant to dynamically visualize medical data can highlight regions of interest, but rigorously correct and consistent behavior would be obligatory.

Our core premise is that as AIs evolve and the desired deployment scenarios change, new infrastructure requirements are emerging. Notably, we see a shift towards *real-time world models*—contexts into which data flows continuously, such as imaging and telemetry captured during medical examinations and procedures (Figure 1), events on a smart farm, and financial market feeds. These real time worlds can be queried by an AI while performing inference and can even be used as sources of real-world feedback during reinforcement training. At the same time, AI architectures are evolving, with monolithic approaches giving way to distributed microservice pipelines. The resulting systems will depart from prior approaches by requiring consistency models that enforce properties across multiple components that share a rapidly evolving state and are spread over distributed hardware.

Our open-source Cornell-based research effort explores non-intrusive data consistency enhancements that overcome limitations seen when such systems are built using existing stream processing, messaging, and AI serving frameworks. For example, NVIDIA's DeepStream provides GPU-accelerated video AI pipelines, but its batching mechanism operates on arrival order rather than sensor timestamps, offering no mechanism for sensor time ordering or lateness bounds. Apache Kafka guarantees message ordering only by arrival order within partitions; it has no concept of event time at the sensor level, delegating temporal reordering entirely to downstream consumers. Apache Flink provides robust event-time semantics with sensor time and configurable allowed

[0]13th Workshop on Principles and Practice of Consistency for Distributed Data (PAPOC 2026), Edinburgh, UK. Contact author: ken@cs.cornell.edu

lateness, but does not act as a data store, meaning supplemental data must be fetched externally. Ray Serve optimizes inference throughput through batching and autoscaling but provides no temporal ordering.

The Vortex system can run the same jobs with stronger consistency guarantees, faster data access paths and higher throughput. The systems community refers to service level objectives (SLOs) as a catch-all for performance; we fully assess Vortex from an SLO perspective in [26].

Here, our central premise is that most forms of AI data *require* a consistency model centered on serializable snapshot isolation [5] extended by a *read-what-you-wrote* property. Today's AI leave their requirements unstated yet achieve strong consistency because data is updated offline. Our work starts by explicitly supporting streaming updates and explicitly implementing serializable snapshot isolation in a single framework that hosts both data and AI computation on the same servers. However, something slightly different is required for time-indexed data queries that run on data captured or generated from external sources. These cases require temporal indexing focused not just on speed, but also on temporal accuracy and reproducibility. This mix of properties turns out to be compatible with serializable snapshot isolation, but irreconcilable with read-what-you-wrote.

Vortex responds by offering both consistency models: the application will receive read-what-you-wrote guarantees unless it does time-indexed data access, which offers all guarantees except the read-what-you-wrote property. This yields a flexible yet non-intrusive framework that can support high-rate incoming data streams even as queries run on past data (including data captured as recently as a few milliseconds in the past), always guarantees serializable snapshot isolation, and is useful both for non-temporal and temporal AI tasks. Moreover, as remarked earlier, the costs are low: Vortex outperforms baseline solutions both for traditional AI jobs and for time-oriented tasks (and sometimes by dramatic margins). Whereas many AI developers assume that strong guarantees will cost more and require extensive changes to existing AI code, our effort is one of the rare situations in which this is not the case.

2 System Design

2.1 An AI Physician's Assistant

Data consistency is trivial in systems with static data, but becomes more challenging in systems that confront evolving system state. The issue arises in many emerging use cases: AIs fine-tuned for enterprise customers often must handle incoming event streams, be they changes in the financial or commodities markets, updates on product or supplies shipments, changing legislation, or even engagement with customers over the details of new projects. Across the board, contextualizing the AI responses to reflect the most recent

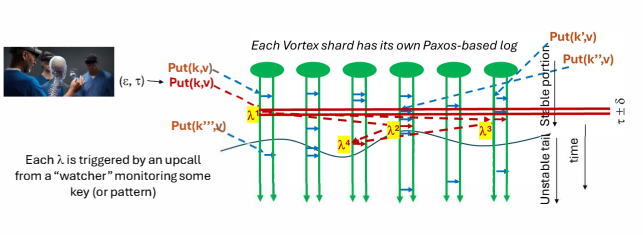


Figure 2. Event flow internal to the physician's assistant. Time flows from top to bottom.

data - the most accurate knowledge of the state and needs of the enterprise - is becoming a pervasive obligation.

For our purposes here, it will be useful to outline a running example that is illustrative, yet representative of this category of emerging AI use cases. Consider a medical assistant that supports a continuous inflow of images and other data captured by sensing devices (Figure 1). The AI enables conversational and interactive engagement with a dynamic visualization of musculoskeletal, circulatory, and nerve imaging. It dynamically highlights possible abnormal findings and tags the VR display with relevant documents (lab reports, patient history, best-practice intervention options, insurance coverage restrictions, etc.). The need for data consistency is evident and would be required as part of any safety review.

2.2 Event Flow Within the Solution

Focusing on events and the task flow, Figure 2 shows telemetry captured from devices and then relayed through components that use Vortex's key-value `put` API, intermixed with queries from the VR display. Each distinct type of data ends up appended to a corresponding file. There is no real limit on how many such files can be stored: Vortex scales out using a "sharded" approach in which data maps to a small replica group (in the figure, we see two replicas per shard). Data written into Vortex is automatically tagged with metadata, notably including a timestamp. As seen in the illustration, AI computations are hosted on the same Vortex servers that host this sharded data. This design minimizes delays to fetch data over the network.

The AI queries illustrate two categories of behavior. In the example, we see a highlighted cervical disk compression that may be impinging on nerves or blood vessels. The display must track while the patient and physician are moving and interacting, hence an AI visualization role is to rotate and align previously captured skeletal imaging data to reflect the patient's current orientation, limb positioning and movement — all forms of real-time telemetry that are processed interactively as the physician selects and reorients portions of the display to zoom in on the possible issue. The second role arises when the physician issues commands that involve tasks like review and summarization of databases of patient records, current medications, laboratory reports, insurance coverage rules, medical literature, etc. These databases and

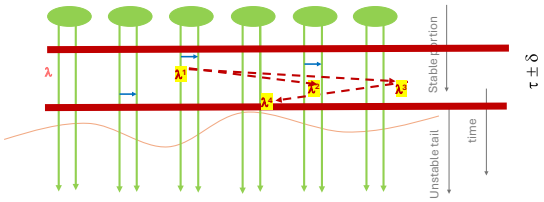


Figure 3. Zoom of a Vortex time-indexed **get**. Data versions are accessed along a stable consistent cut.

external services presumably would offer consistency based on database serializability.

In Figure 2, an AI query initiated by the physician is highlighted. Notice that although the physician's assistant might employ dozens of AIs, the figure only shows four lambdas. This centers on a form of AI packaging: AI code written in a framework like PyTorch or TensorFlow has very little host-resident code (many are little more than computational recipes that dispatch all the hard work to GPUs or shared host-computing libraries). In Vortex, we are relaxed about loading AI code into multiple servers; it is reasonable to assume that every Vortex server hosts a copy of the code for every AI component. However, models and other data dependencies are huge. Preloading all of these models on every server is infeasible, yet loading them at the last moment would be prohibitively slow.

How exactly might these AIs be implemented? Brevity precludes a detailed discussion (see [26] for details), but the two AI pipelines we experimented on are examples of custom AIs built from off-the-shelf components. The first, PreFMLR, is a multimodal retrieval-augmented (RAG) pipeline that retrieves medical literature or other documents relevant to queries about imaging, while the second, SpeechSense, offers recommendations in response to audio queries on data streams or video. Both employ open-source models for language and vision tasks (text and image encoding, cross-attention, inference, knowledge retrieval). Data storage and retrieval occurs through our key-value store, augmented by a standard library for approximate nearest-neighbor search. No changes to the existing AI logic were required, and few (if any) API changes. These examples are representative of a broad category of systems in which AI is combined with rapidly evolving external state. We have experience developing AIs of a similar kind for applications in digital agriculture (dairy farming) and factory automation, and the same needs arise in domains as remote as business intelligence.

Given that our underlying storage is sharded for scalability, this suggests an opportunity. Consider a set of AIs that share data dependencies, operate on the same large input objects, or exchange data with one another. If the associated models will fit in memory, grouping them into a single package can offer efficiencies. Now, when we activate that package on some shard, the replicas in the shard will find needed data locally (if the data isn't local, the Vortex `get` will fetch it over the network, but we try to minimize that case). For a flow of events that will visit the same package and models again and again, we can cache the data dependencies, eliminating even the need to load them from storage. Indeed, we can often retain the largest kinds of objects (model parameters and hyperparameters and Low Rank Adapters (LoRAs)) directly in GPU memory. This is what the lambdas in the illustration represent: a grouping of AIs and the associated models and data objects, selected because it is efficient to run them side by side. With preplanning it is often possible to partition a shared GPU into mini-GPUs, a technique that enables AI models within the same package to run concurrently.

Although reminiscent of web microservices architectures, our approach differs in important ways. First, whereas web microservice architectures casually launch new instances on the fly in large server pools, AI elasticity hinges on prepositioning needed objects, including the ML models themselves which can be huge. When a lambda is launched, the required data must be either already in memory or at least local. For Vortex, we solve this by launching task instances within shards that already hold any needed models and data.

Our approach also avoids the need for so-called *incoherent* caching, an application in which applications trust cached data without checking for newer versions. Incoherent caching is widely used in today's standard web microservices frameworks, which favor an approach called the CAP and BASE methodology [4, 20]. That approach embraces staleness as a feature, hence would not be appropriate in a medical setting, or any other context where the AI needs correct, explicable behavior in the face of incoming data.

2.3 Consistency Obligations

Our task is not dominated by engineering puzzles. AI applications are heavily layered, with much of the code written in higher level frameworks like PyTorch or TensorFlow. Each brings its own runtime environment that accesses data through APIs that map to storage layers such as key-value storage and networking layers such as TCP or RDMA (Infiniband). Thus while we do need to introduce a low-level consistency model, we also want that model to benefit higher level code that accesses data only through these preexisting frameworks. Success centers on leveraging guarantees already inherent to the frameworks. By doing so, we can run an existing AI platform *unchanged*, and it will inherit stronger properties from our Vortex runtime.

To see how this plays out, revisit Figure 2. The AI logic will have been coded using standard frameworks, then imported to run in Vortex. Although the AI logic is structured as a pipeline, AI subtasks are grouped into packages. In such a package, the AI subtasks are event-triggered and non-interfering even when concurrency occurs. AIs are typically

implemented as Python modules with independent namespaces. Data sharing between AIs occurs through request-response calls from one stage to another, lock-free sharing of files, explicit message queuing interactions (normally via Kafka), or key-value `put` that triggers a matching key-value `get` in the next AI subtask.

A further insight is that in modern ML, all of these abstractions (files, Kafka messaging, etc.) map to key-value operations. Moreover, the underlying key-value objects are *immutable*, but organized into append-only logs. Each time an object is stored via a `put` the system checks for an existing version; it either creates a new log or appends to the existing log. Many AIs fit precisely with this model, for example working with POSIX files by limiting themselves to append-only updates or full-file replace. The mapping to a key-value store is trivial: The keys identify the file, and the values are deltas that define the versions of the file. This observation may sound extremely detailed and low level, but it will prove to be significant because it enables us to focus our consistency model on a key-value abstraction.

Indeed, in a thousand-foot view of today's AI infrastructures, the AI computations themselves are functional: they accept some input state and transform it into an output state, and the state is entirely defined in terms of immutable objects and sequences of object versions. That AIs should universally exhibit such a simple underlying data model may be surprising, but makes sense if we consider the way AIs evolved over the past few decades. Heavy use of checkpoint and rollback within AI training and autoregressive language generation has favored a style of coding that facilitates rollback and preserves FIFO event orderings.

Our task is thus to offer a highly efficient key-value store that leverages this functional computing model and the immutable nature of the objects being managed to offer stronger forms of consistency. To relate this to the figure, recall that Vortex has configured the AI pipeline for the overall AI job into a dataflow between four lambdas, which it scheduled onto four servers where the data dependencies of the respective lambdas can be locally satisfied. As we learned earlier, the AI as a whole will be a data flow graph, but that graph has been partitioned through an offline analysis into the four packages shown. Inside a lambda, we would find subsets of the data flow graph consisting of independently callable AI stages that are individually FIFO-preserving and execute in a functional manner. They apply input to the AI logic, and any outputs they create can be understood as newly minted objects (or new versions that fully replace existing objects).

This data-flow of functional AI tasks is interconnected by directed edges, representing the passing of data from one task to the next. We now know that even when the specific APIs used vary, they instantiate a single unifying pattern: one node in the dataflow pipeline triggers the next, and the behavior maps to a unified key-value storage model centered on functional events that extend logs of immutable object versions or deltas. Indeed, because the objects themselves are immutable once created, we can understand the (sharded) storage layer as having a single append-only log per shard: the logs for individual objects can be filtered out of the shared log. Each new object or new version of an existing object adds a log record, hence as the updates stabilize, the immutable portion of the history of the system as a whole grows monotonically. Figure 2 shows this as a wavy line, with the stable immutable state on top (in the older portion of the system state) and the evolving (newer) log appends below. This insight is extremely convenient when reasoning about consistency.

2.4 Runtime Model

Vortex tolerates benign crash failures, but not Byzantine faults or undetectable data corruption. In keeping with the virtual synchrony model, faults are sensed by timeout and reported to a membership tracking service. This service updates the membership and notifies all system components in a consistent, coordinated manner guaranteed free of partitioning risk. For timestamping, we assume clock synchronization with known bound δ on clock skew and network links with bounded delay ϵ.

2.5 Offline Planning

The grouping of AIs into the lambdas as seen in the timeline figure occurs offline and is one of several such offline actions. They all center on analysis of a description of the job, represented as an annotated dataflow graph. These graphs can often be created automatically, or with at most a small amount of human help. Given such a graph, our offline analysis tool enumerates possible concurrency patterns. An optimizer then determines the most efficient packing of AI tasks into packages and identifies shards on which each package will run, respecting constraints such as required GPU memory or compute. The corresponding AI models and other data dependencies can now be grouped using what we call an *affinity tag* [11] and then stored into Vortex. Objects in the same affinity group will be collocated on the same shard and loaded as a set on first access. The AIs within each of these packages initialize themselves by registering upcall trigger requests: they specify a key or pattern, and in the event of a key-value `put` or `trigger` using a matching key, Vortex on the node(s) where that match occurred will upcall to the AI logic. This upcall registration occurs on every Vortex node, but the AI will then be idle awaiting an upcall.

Vortex has been designed so that its APIs and infrastructure will preserve FIFO ordering for data accesses. As a result, by centering Vortex on a data-flow view of the AI and on its interaction with key-value storage, consistency of the key-value layer is elevated and becomes a consistency model "used by the application" — even for an application created by a developer who gave minimal thought to consistency.

2.6 External Devices and Clients

To recap, let us walk through the example shown in the figure. Data is captured from medical equipment, and queries from end-users. The associated external client applications bind to Vortex servers, picked at random from our distributed pool of servers to spread the associated overhead evenly. For each client, the selected Vortex server node will serve as a point of ingress for the event or flow of requests it issues. When the first event occurs in a new flow, the preplanned routing is retrieved. For each AI task, the ingress node identifies the shard where the corresponding affinity group resides and selects a member to perform AI tasks for the entire flow. Because Vortex makes routing planning decisions offline, all that remains at runtime is to decide which instance within a shard — which replica — will execute the required AI tasks. This decision occurs once for the entire flow when it is first instantiated; after that, all events follow identical routing. Load balancing still occurs between flows, but per-flow scheduling allows Vortex to preserve FIFO ordering even when the AI dataflow graph branches and two or more AI subtasks run concurrently.

After this first event, each time the ingress node receives a new update or query in an existing flow, it reuses the flow activation by sending a key-value pair representing the event to the ingress AI task (λ_1 in our example). The key in this pair represents the flow-id, while the value will be the serialized data associated with the event. It then forwards this key-value pair to the desired shard, selecting a representative to handle the request in accordance with the predetermined route. To store an update, the representative replicates the action across the shard using a virtually synchronous persistent replication protocol [14]. If the key matches an existing upcall trigger, one of the shard members can then run a follow-on AI task. For a query, a point to point transmission is used to trigger the AI upcall.

In the figure, this triggers λ_1 which then triggers λ_2 and λ_3. Notice that λ_4 requires two inputs, one from each of λ_2 and λ_3, and that these must result from AI computation associated with the same query event (a requirement that also arises in the AllReduce collective communication (CCL) pattern). Our design facilitates doing so: flows are FIFO-preserving, hence the inputs arrive asynchronously but in the same order.

2.7 Data Versioning

The preservation of FIFO order can be viewed as the first of a series of consistency properties offered in Vortex. In what follows we ask what data access consistency model is most appropriate for the two types of data at play: time-indexed data arising from sensors, and data hosted in the Vortex storage layer. Our discussion will center on data versioning.

We noted that versioning is pervasive and that AIs can be viewed as functional elements that create new versions of objects but never edit an existing object in place, but why exactly is this the case and why should it matter? Fundamentally, this pattern occurs because a versioned data model makes it easy for an AI training framework to explore modifications to model parameters but then to revert to the prior state if the outcome is judged to be poor. Versioning also facilitates checkpointing and rollback for fault-tolerance: we simply discard any new files created since the checkpoint was formed and truncate files or logs to the lengths recorded in the checkpoint. At inference time, the autoregressive token generation structure has a similar feel, particularly for large language models.

Many AI data frameworks implement rollback by deleting files or versioned AI objects, but in Vortex a rollback simply creates a reversion record, enabling us to preserve the immutability of the stable past. Object version numbers thus advance monotonically even if some new version is really an exact duplicate of a prior version to which the system has reverted. But we have never observed an existing AI that cares about object versioning. Instead, AIs access the "most current" key value object instance. Version numbers underlie iterators, but the details are invisible to the code using the iterator (often written using a package like Pandas).

A monotonic versioning model also fits well with the devices seen in our medical example. Consider a video camera: video streams are represented as sequences of keyframes and deltas, transmitted over a FIFO-preserving networking protocol like TCP, and either consumed from the stream or sequentially stored into files using file appends. Meta-data tags will often carry ancillary information such as timestamps. Updates to the patient's electronic health records append to the record but never revise past information. The same is true for lab results, updates to medical literature, and advisories from drug companies, legal authorities, and insurance companies. Indeed, FIFO record ordering sometimes reflects causality: a new patient will have an ID entered into the medical records system before records using that ID appear; a lab result will occur after the observation that caused the lab test to be ordered.

Vortex respects and enhances basic FIFO guarantees in the following manner:

1. At the lowest level, the system relies on Derecho [14], which offers FIFO-ordered reliable message streams as well as a multi-destination stream from a sender to a set of destinations. Derecho in turn is built using LibFabrics, a library maintained by the HPC community that manages the Message Passing Interface (MPI) framework, and LibFabric maps networking operations to hardware accelerators like remote direct memory access (RDMA) or compute express link (CXL) when possible, with TCP as a fallback.
2. Over these channels, Derecho implements *virtually synchronous* membership tracking [2, 3], totally ordered in-memory atomic multicast within sharded

subgroups and Paxos-style totally-ordered data replication. The latter is used to support the replicated append-only logs used in Vortex.

3. A layer we call Cascade introduces event handlers over the base offered by Derecho to support versioned key-value objects. If desired, it will first check that the old version number is the one expected and if not, reject the request: an atomic *compare and swap*. To save space when appending data to some existing object, only the bytes in the new suffix are stored.
4. Affinity grouping [11] is used to arrange that related objects will co-reside on the same shard, even if the objects have keys that would otherwise hash to different shards. A single `put` can atomically update a list of objects provided that belong to the same affinity set.
5. Last, Vortex hosts AIs over the Derecho and Cascade layers, offering both the native Cascade key-value API and the wrapper APIs mentioned earlier: POSIX files, pub-sub, etc. Consistency as perceived by a Vortex application thus emerges from the underlying properties offered by Derecho and Cascade.

   We have seen that a diversity of application-visible APIs used within AI dataflows all map to `put` and `get`, and these in turn are mapped to Derecho communication. A pending update (`put`) is initially *unstable,* meaning that it may be finalized in some logs but still underway in others. A stable update is one that has reached a point of being fully replicated and hence deliverable to applications monitoring the corresponding key (curving line in Figure 2). By default, `put` waits until the underlying update is completed, and `get` guarantees that it will retrieve the most current stable version of any object. Iterators always return lists of stable versions, and attempts to directly access an unstable object (for example, via a temporal index into the unstable portion of the timeline) will silently wait until stability is achieved. We measure this delay in our experimental section, and show it to be at most a few milliseconds.

2.8 Serializable snapshot isolation

These observations set the stage for the Vortex implementation of serializable snapshot isolation.

Updates. Applications can issue updates via `put` to a single (key,value) object or a list of objects in the same affinity set. The request will be relayed to the shard hosting the object(s), where the update protocol executes them in a FIFO-preserving total order. This yields a transactional behavior on a per-shard basis and is strong enough to support mechanisms such as the Kafka pub-sub message queue transactions popular in microservice architectures.

Also available but seemingly unnecessary for AI pipelines is a full database-style transaction algorithm based on an approach developed in Heron [10]. This method splits a transaction into two stages. First, the transaction runs speculatively, yielding a list of key-value objects read (with their version numbers) and a list of key-value objects updated (with version numbers and desired new values). The list is sorted into ascending order by shard number. A chained transaction protocol then visits shards in order, locks the local key-value objects read or written against access, and validates that the version numbers are unchanged. If the procedure reaches the tail of the list of shards, the transaction commits from the tail of the sequence back to the head, much like in Chain Replication [24]. If a version number mismatch is sensed, the transaction aborts. The ordered locking policy ensures deadlock freedom.

Queries. Our AI dataflow graph routing planner ensures that Vortex dataflows will not loop back and forth between shards. As a result, an application that issues a series of `get` operations will read data along stable consistent snapshots: a set of log prefixes that have each become immutable and in which updates conform to the virtually synchronous variant of the state machine replication model—totally ordered, fault-tolerant, and fully replicated within the relevant service membership view (i.e., the assignment of service nodes to shards). No locking is needed because of the immutability guarantee: queries see read-only immutable data.

Earlier, we stressed the importance of the read-what-you-wrote property for applications concerned with logical consistency. In Vortex, this property holds because when an application reads an object, the system always returns the most current version, and the default `put` API is synchronous. It follows that in a dataflow pipeline, a downstream `get` will see any object or version created by a task that ran earlier in the flow.

2.9 Temporal indexing

Temporal versioning poses a challenge that proves to be irreconcilable with the model used for FIFO serializable snapshot isolation. Consider a stream of data (gestures, remarks, captured imaging) originating at a variety of devices and passing through a variety of ingress nodes. FIFO ordering ensures that the logical ordering of these updates in shard logs respects the sequence in which events reached individual Vortex nodes. However, concurrent updates can still interleave in any pattern that respects the FIFO restriction. Moreover, clock synchronization is never perfect, and network links can introduce unpredictable delay. Thus, object A might be ordered before B by Vortex even if B carries the earlier timestamp. Thus, it is entirely possible that A will be stable and prior to B in some log, yet an iterator using timestamp order should order B before A.

Vortex addresses this issue by offering two query modes. **Logically-prioritized** reads employ a FIFO-preserving serializable snapshot isolation that respects the read-what-you-wrote model of Section 2.8, but by doing so might violate timestamp ordering. **Temporally-prioritized** query mode arises when we use data timestamps and an index to identify the version of an object that existed at a specific point in time. By default, `get` operations issued by the AI logic use the timestamp of the initiating event as their temporal index. However, an explicitly time-indexed form of `get` is also provided, allowing applications to fetch data for a specific point in time or iterate over a range of timestamps.

The Vortex temporal-priority query policy waits for *temporal stability* for the time period being accessed. It estimates the time required for data to transit through Vortex (accounting for possible clock skew) and then prevents access to the suffix of a log where out-of-temporal-order events might still be arriving. We do so by creating a temporal index that reflects a stable temporal order over the logically stable prefix of the log.

Our basic rule is simple. Recall that clocks are synchronized to a maximum error of δ, that relaying a message from an external client to Vortex takes at most time ϵ, and that `put` will complete in maximum delay Δ. Then Vortex can behave as follows:

1. On the Vortex node where the request first arrives, reject a new `put` if the timestamp is impossibly far into the past or future. On a server node with current time *now*, a `put` would reject an item with timestamp $\tau < now - \delta - \epsilon$ or $\tau > now + \delta$. This rule is slightly distrusting of external clients and devices: had the data originated on a Vortex node, we would trust the timestamp.
2. To perform the `put`, a Vortex server node will first relay the request to a server node belonging to the target shard and then run the normal Derecho atomic multicast or persistently logged replication protocol. The protocol will terminate when a message becomes logically stable, creating the new object version. Thus a maximum of $\epsilon + \Delta$ time elapses. Next, in support of temporal indexing Vortex additionally tracks a temporal stability frontier, allowing access to a prefix of the log only for queries indexing at least $\Delta + 2\delta + \epsilon$ time into the past. This accounts for the full delay from the external client to the completed creation of the new object version on the server where this stabilization frontier is tracked, as well as the possibility that the client's clock is running slow. Recall that we are assuming ϵ, δ and Δ are all correct. This implies that there cannot be any updates still propagating through the system with smaller timestamps, hence that prefix of the log is both logically stable and temporally stable.
3. For the prefix of the log that is both logically and temporally stable, incrementally update a temporal index that tracks the timestamp sort order.

Notice that temporal data access experiences a minimum delay from when data reaches the system until when it can be seen in temporal queries. Even if the system magically could achieve logical stability with 0 delay, the temporal stability window would force a delay of $\Delta + 2\delta + \epsilon$ prior to the data becoming accessible via temporal indexing. In contrast, the non-temporal `get` incurs no delay beyond the cost of the replication protocol. Our experiments reveal that on modern hardware, ϵ, Δ and δ are all very small, and the formula yields a delay of less than $5ms$ even for storage of large objects like video frames. Humans interacting with VR generally tolerate delays smaller than $10ms$, hence the temporal stabilization delays built into Vortex should be tolerable.

Vortex's temporal stability technique is a variation on the approach used by Cristian, Aghili, Strong and Dolev in [9], but whereas that paper considered a variety of failure models, our solution is simplified by the assumption of a single and rather benign failure model.

2.10 The Puzzle of Mixing Logical and Temporal Consistency

Given that $\Delta + 2\delta + \epsilon$ is small, why offer two query modes rather than simply combining them? Our main concern centers on the read-what-you-wrote problem. Suppose that λ_1 creates some object A, and λ_2 attempts to read A. We showed that λ_2 will read the proper version of A when using the default logical mode access. But this turns out to be incompatible with the temporal query mode. The central issue is that AIs take time to execute. Suppose that our AI query is interrogating the system state as of time τ, and that λ_1 ran first and computed for time υ before writing A at time $\tau + \upsilon$. Under the default temporal indexing policy, λ_2 will try to read A at time τ, which is before λ_1 wrote A. If we wish to respect "read your own writes," we must satisfy this `get` request from the version of A written by λ_1. Yet if A preexisted and λ_1 overwrote it with a new version, the version closest to τ would be the old one. Moreover, we desire determinism, yet approaches other than pure time indexing would result in non-determinism for temporally indexed queries. If we believe that A is being updated repeatedly, we should assume that we are glimpsing a transient state. The version of A that is stable and closest in time to τ is the only one deterministically and reproducibly identifiable. Worst of all would be a query that doesn't specify the desired version of A. In logical query mode, this yields the most current version of A, but if this is how we handle a temporally indexed `get`, temporal indexing will lose its reproducibility guarantee. Conversely, at the time the temporal query is indexing, the new version of A created by λ_1 may not have existed. Accordingly, our temporal query mode opts for the version closest in time

(but not prior) to τ. In effect, explicit temporal indexing takes priority over read-what-you-wrote.

Butler Lampson famously advised OS designers to optimize for the base case and resist the urge to solve edge cases, arguing that high performance for the base case would far outweigh any performance loss from working around limitations in obscure applications that use interfaces in unusual ways [18]. We have taken that advice to heart. Rather than struggling to unify the models at what would probably be a high latency cost, we simply offer two models.

The same philosophy extends to concerns about timing errors caused by overly aggressive choice of ϵ, δ and Δ. If these parameters are too small, our implementation might allow a temporal query with time-index *now* to go forward as soon as logical stability is achieved but before genuine temporal stability is achieved. In such cases, late-arriving temporally-tagged data could change the time index used by Vortex after the application has already read the corresponding object, exposing a visible temporal inconsistency (this could never occur for accesses to old data, which will have long since stabilized). We explore this question in our experimental section and will show that the likelihood of observing such an error is extremely low provided that the configuration parameters capture the most typical runtime conditions.

2.11 Medical Assistant Example

Consider the medical assistant example from Figure 1. The preceding discussion makes it clear that this application will use a mixture of queries. For visualizations like the one shown, temporal query mode will yield a reproducible and predictable result: a visualization of the state the system was in at time τ. The few milliseconds of lag before updates become accessible is a low enough delay that the physician should not even be aware of it. But for non-time-indexed tasks, including patient records and lab results that might be shown through the physician's VR display, the logical query mode is probably more suitable. Indeed, these queries might even default to always use the most current stable versions of any evolving data. Intuitively, one expects medical equipment to reflect the "current" state in most respects, which obviously would include the most current data, no matter what the timestamps. Thus our choice for continuity in a visual display simply favors a different tradeoff than the one we would use for other AI tasks. By offering both options, Vortex enables an application like this and avoids behavior that would otherwise feel anomalous and that might even endanger the patient.

3 Experiments

This section presents an experimental evaluation of Vortex, focusing on its consistency guarantees and overall performance characteristics.

3.1 Experimental Environment

We conducted our experiments on a dedicated cluster of five servers, each equipped with Mellanox ConnectX-4 VPI NIC cards operating in RoCE mode. The servers communicate through an Ethernet switch providing a 100 Gbps network backbone. Each server contains dual Intel Xeon Gold 6242 processors (2.8 GHz, 32 cores total) with 192 GB of memory. The RDMA stack uses MLNX_OFED 23.10. Server clocks were synchronized using PTP, achieving a maximum clock skew δ of 70 ns between any two servers. In our calculations below we conservatively round this up to 100 ns.

3.2 Timestamped Puts vs. Standard Puts

To validate the suitability of Vortex for real-time applications, we first evaluate its performance under workloads representative of the medical assistant scenario described in Section 2.11. Video processing models in AI pipelines are typically trained on temporally ordered frame sequences, making correct frame ordering essential at inference time. A system like Vortex ensures that models observe frames in their proper temporal order despite variable network and processing delays. For such latency-sensitive applications, we assume an on-premises deployment with a well-maintained, stable cluster that avoids major load spikes—conditions reflected in our experimental environment. These experiments also use a replication factor of 3 for fault tolerance. We selected two configurations: 30 KB messages at 20 qps to simulate low-resolution video streaming at 20 fps, and 512 KB messages at 20 qps to simulate higher-resolution video streaming. As shown in Table 1, timestamped puts incur no measurable overhead compared to standard puts.

3.3 Choosing Δ and ϵ

To deploy Vortex, we must translate observed performance into a concrete parameterization. By measuring latencies for our target workload, the results in Table 1 allow us to choose Δ and ϵ as fixed bounds that the system uses to compute stabilization delays. The discussion below explains how we map these results into parameter values and what consistency–latency tradeoffs arise when choosing conservative versus aggressive bounds.

For the 30 KB case, the maximum observed latency from server receipt to persistence was 2131 μs, suggesting Δ = 2500 μs as a reasonable bound. Similarly, the maximum client-to-server latency of 154 μs suggests ϵ = 500 μs. Applying the same reasoning to the 512 KB case yields Δ = 10000 μs and ϵ = 500 μs. With δ = 100 ns, the resulting stabilization delays ($\Delta + 2\delta + \epsilon$) are approximately 3 ms and 10.5 ms for the 30 KB and 512 KB cases, respectively — well within the latency tolerance for VR applications.

The choice of Δ and the estimated values for ϵ and δ involve a fundamental tradeoff between temporal consistency

Table 1. Latency Statistics (μs) at 20 qps

Size	Mode	Operation	Min	Max	Mean	Median	P95	P99
30 KB	Timestamped	Client Sent → Server Received	110.34	154.62	123.81	123.14	132.75	144.27
		Server Received → Persisted	775.68	2131.46	910.41	911.62	993.24	1087.23
		E2E Put	896.00	2254.85	1034.23	1035.65	1118.49	1215.51
		E2E Get	232.00	278.00	253.57	253.00	260.00	271.00
	Standard	Client Sent → Server Received	109.31	157.95	124.34	122.88	134.91	140.60
		Server Received → Persisted	841.22	1677.57	946.55	939.26	1022.52	1109.95
		E2E Put	961.28	1797.12	1070.88	1064.70	1148.84	1231.84
		E2E Get	231.00	284.00	249.94	249.00	257.00	267.00
512 KB	Timestamped	Client Sent → Server Received	128.51	238.08	181.48	181.25	202.24	212.10
		Server Received → Persisted	2524.67	9754.62	3002.78	2945.54	3102.46	4404.33
		E2E Put	2705.66	9963.52	3184.25	3127.55	3282.30	4590.95
		E2E Get	490.00	755.00	640.52	652.00	708.00	722.00
	Standard	Client Sent → Server Received	145.66	222.46	169.71	167.17	194.69	210.82
		Server Received → Persisted	2812.67	8299.78	3093.02	3016.70	3261.70	6168.19
		E2E Put	2967.04	8466.94	3262.73	3187.46	3443.33	6361.65
		E2E Get	488.00	840.00	660.89	654.00	692.00	703.00

Replication factor of 3. 1000 total requests sent; first 10 treated as warmup.

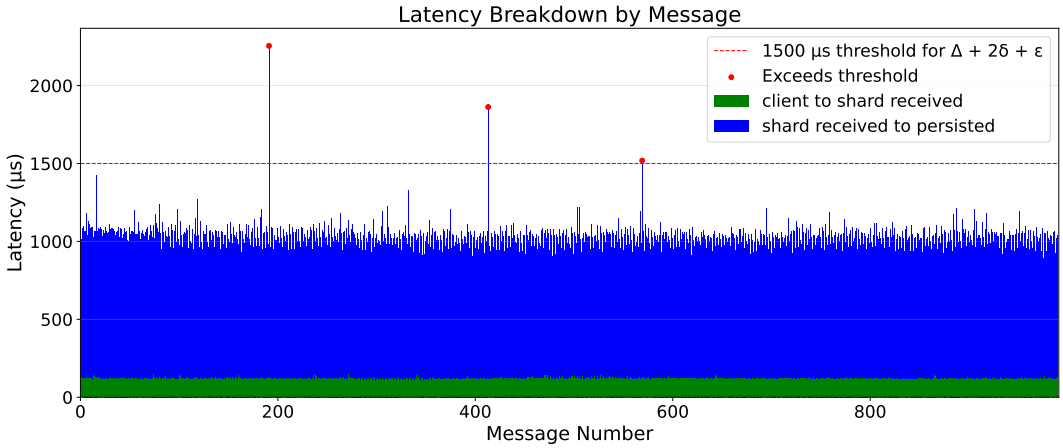


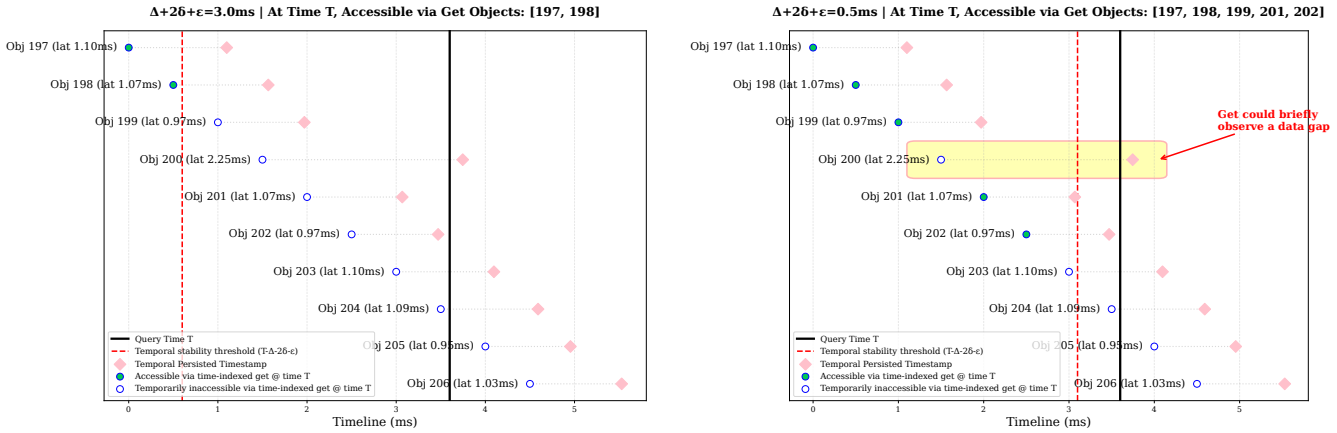
Figure 4. Latency breakdown for 30 KB messages at 20 qps, showing the 1500 μs threshold. Of 990 messages (excluding 10 warmup requests), only 3 exceeded the threshold, supporting our contention that strong consistency guarantees need not harm throughput or latency-oriented SLOs.

guarantees and query latency. We illustrate this tradeoff using the 30 KB workload from Table 1.

A conservative choice of $\Delta + \epsilon = 5$ ms virtually guarantees that all puts will complete within the stabilization window, ensuring strict temporal consistency. Applications that cannot tolerate any temporal inconsistencies—where a get might initially miss a value that has not yet persisted, only to return it later—should adopt such conservative parameters.

Applications that can tolerate occasional inconsistencies may benefit from more aggressive settings. Setting $\Delta + \epsilon = 3$ ms provides strong consistency under normal conditions while accepting rare violations during latency spikes. For latency-sensitive applications willing to accept more frequent inconsistencies, $\Delta + \epsilon = 1.5$ ms minimizes stabilization delay. To assist in visualizing the resultant risk, Figure 4 illustrates the consequences of choosing the aggressive 1.5 ms threshold for our 30 KB experiment. Of 1000 messages, only 3 exceeded the threshold. For these messages, a temporally-indexed get issued between the 1500 μs threshold and the actual persistence time would race with the update. Whether this is acceptable depends on the application's requirements.

Figures 5a and 5b further illustrate this tradeoff. In Figure 5a, setting $\Delta + 2\delta + \epsilon = 3$ ms ensures temporal consistency: no gets will return out-of-order results. However, at query time 3.6ms, objects 199–202 remain invisible to the client despite being persisted, because their timestamps fall after the visibility threshold. Note that object 200 persisted after objects 201 and 202, but they will be processed in order



(a) Conservative threshold ($\Delta + \epsilon + 2\delta = 3$ ms). Larger stabilization window ensures temporal consistency but impacts perceived latency.

(b) Aggressive threshold (0.5 ms). Smaller stabilization window reduces latency but may briefly expose incomplete update timeline.

Figure 5. Temporal query behavior under different stabilization thresholds, showing measured put latencies but with an artificial inter-put latency intended to illustrate tradeoffs. An object is considered accessible only after two conditions are satisfied: (1) its device timestamp is no later than the temporal consistency threshold (red dotted line), and (2) its persisted timestamp is no later than the query time (black solid line).

because the threshold is set large enough that by the time object 201 is visible, object 200 is also visible. This introduces additional perceived latency. In contrast, Figure 5b shows the effect of an aggressive threshold of 0.5 ms. While client-perceived latency is minimized, temporal consistency may be violated: object 200 has not yet persisted, but objects 201 and 202—with later timestamps—are already visible.

Ultimately, the appropriate choice of values to use for Δ, δ and ϵ depends on the specific requirements and constraints of the target application. Vortex provides the flexibility for developers to make this tradeoff explicitly.

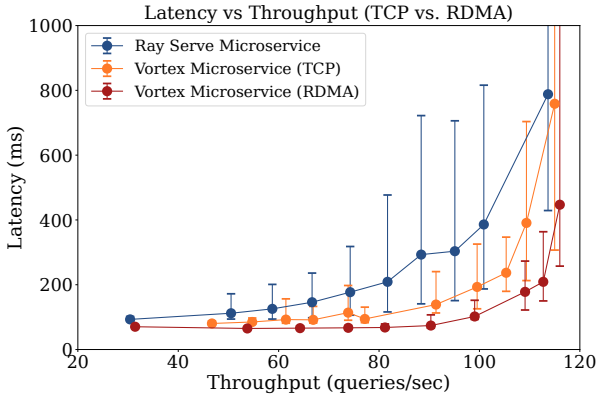


Figure 6. Performance comparison across serving paradigms and network protocols. The x-axis gives the presented load, and the y-axis median latencies with 95% and 5% bars.

3.4 Vortex vs. Baseline AI Pipeline Serving System

The preceding experiments demonstrated that Vortex's temporal consistency mechanisms add negligible overhead to storage operations. However, low storage-layer overhead is of limited value if end-to-end application performance is poor. We now evaluate whether Vortex performs well at the application level by comparing Vortex against Ray Serve, a widely-used AI serving framework, on an end-to-end AI pipeline.

Unlike the previous experiments, which isolated storage-layer latency, this section reports end-to-end latency: the total time from query arrival through AI inference completion. As throughput increases, latencies rise due to batching in the AI components rather than slower storage operations.

Figure 6 compares both systems for an AI pipeline supporting queries of the kind arising in Figure 1. In PreFLMR [19], (query, image) pairs are encoded, cross-attention runs, and finally a knowledge retrieval action occurs. Vortex outperforms Ray Serve across both TCP and RDMA configurations. At 100 QPS with a 200 ms SLO target, Ray Serve's miss rate reaches 93.8%, while Vortex with RDMA maintains just 0.89%. For a 500 ms target, Ray Serve misses 26.9% of requests; Vortex misses none. The tighter latency distribution under RDMA also has implications for temporal consistency: as discussed in Section 3.3, predictable latencies enable more aggressive choices of Δ and ϵ, reducing the stabilization delay visible to applications.

These results show that Vortex's consistency guarantees impose no meaningful performance penalty, and that its optimizations actually yield substantially better throughput and SLO compliance than the baseline. Moreover, Ray Serve does not natively support ordering requests by sensor timestamps, a capability that Vortex provides without sacrificing performance.

3.4.1 The role of RDMA in consistency and performance. The choice of Δ and ϵ depends directly on storage-layer latency characteristics: tighter thresholds require faster and more predictable put persistence. RDMA is valuable to achieving both. Under low load, Vortex achieves an average end-to-end latency of 70 ms compared to Ray Serve's 93 ms—a 33% improvement. More importantly for temporal consistency, Vortex's stage-to-stage data transfers complete in under 2 ms, whereas Ray Serve requires 5–13 ms over TCP. This gap persists even for large intermediary results such as vision encoder outputs. The faster transfer times directly enable smaller Δ values: puts reach the storage layer and persist more quickly, shrinking the stabilization window required for temporal consistency.

RDMA also reduces latency variability, which affects how aggressively $\Delta + \epsilon$ can be set. TCP latencies exhibit inherent instability due to congestion control's sawtooth behavior. Even when median latency is acceptable, worst-case latencies force conservative parameter choices. Consider the error bars in Figure 6: TCP configurations show broad latency spreads, whereas RDMA maintains tight distributions across the full range of input rates until backlog formation begins around 115 qps. This predictability means that a $\Delta + \epsilon$ threshold calibrated to the 99th percentile under RDMA can be set much closer to the median than would be safe under TCP.

The SLO implications are significant. At 100 qps, Ray Serve achieves a median latency of 400 ms, but meeting a 5% SLO miss rate would require a target of at least 800 ms. Vortex on RDMA maintains latencies close to its median with far fewer outliers, enabling tighter SLO targets without increased violation rates. For applications requiring both temporal consistency and strict SLOs, RDMA's contribution is twofold: it enables smaller stabilization delays through faster persistence, and it makes aggressive thresholds viable through reduced variance.

When configured to use TCP rather than RDMA, Vortex's throughput drops by 1.7% and latency increases by 1.23–1.9×. Even so, Vortex-TCP outperforms Ray Serve, indicating that our optimizations—zero-copy data paths and lock-free critical paths—provide benefits independent of RDMA. However, the consistency advantages of predictable, low-variance latencies are most fully realized with RDMA.

4 Discussion

Notice that the mechanisms and subsystems over which Vortex was constructed draw on ideas that are mature and familiar to the data consistency community:

- The theory of state-machine replication dates to Lamport's original 1978 paper and already argues that real-time systems are deeply at odds with logical consistency [17]. The model we adopt can be traced back to the Isis Toolkit, which offered the first implementation of virtual synchrony, continuous membership tracking and self-repair, and a durable replicated update called `gbcast` (1984, but first published in 1987 [3]). Of course, protocols have evolved during the ensuing 45 years. Vertical Paxos, which brings a form of virtual synchrony into a Paxos setting, was created in 2009 by Lamport, Malkhi and Zhou. By 2012, with the help of Malkhi and Lamport, protocol II of that paper was expanded into a new virtually synchronous atomic multicast protocol, presented in a temporal logic formalism in Chapter 12 of Birman's textbook on distributed systems [2]. That protocol was then further modified to optimize its functionality for fast user-mode communication (by layering Derecho on LibFabric, an open-source universal adaptor, the resulting logic can run on TCP, RDMA and several other low-level mechanisms) and implemented in Derecho in 2019 [14]. After so much evolution, it becomes important to use formal methods to prove each new protocol correct. This protocol was formalized and formally proved: first using a theorem prover called Ivy, then using a higher-order specification tool and prover called DistAlg [23], and finally in Coq [21]. The result is a series of proofs that include two machine-checked ones (the most comprehensive is the DistAlg specification and proof).
- Meanwhile, Keidar and Shraer developed a temporal logic formalism within which they explored lower bounds for the dynamically uniform agreement at the crux of this form of atomic multicast, publishing this work in 2006 [15]. Upon reviewing the Derecho protocols, Gregory Chockler then offered a message-counting argument showing that the Derecho RDMA-oriented atomic multicast achieves this lower bound. Some theoretical analyses of this kind only hold asymptotically, but in head-to-head comparisons, Derecho significantly outperforms prior Paxos-based protocols including RDMA Paxos libraries, RaFT, and Zookeeper at a wide range of object sizes and system deployment scales, often offering orders of magnitude speedup both in the sense of higher throughput and lower latency [14].
- Turning now to the data consistency problem, while atomic updates offer a trivial form of serializability for updates to data replicated in shards, we need something more for applications that run as pipelines spread over multiple hosts and querying multiple saved objects. Here, again, the deep theoretical background of the area offers an answer. Lamport's potential causality operator dates to 1978, and the formalization of a consistent cut to 1985 [7]. Jointly, as we saw in the above event-timeline illustrations, this theory and the time-indexing methodology enable Vortex to "situate" distributed queries. Even though the computation is spread over multiple machines in a compute cluster, the data accessed for a single query will be as current as possible while also being complete (no missing data) and gap-free under a causality analysis.

- Hellerstein and Alvaro's work on Bloom and Calm stress the value of monotonicity in a functional computing model and develop a comprehensive data consistency framework based on these properties [1, 12]. Although Vortex doesn't expose the model directly to users, and its primary API centers on a key-value store, the same sort of asynchronous and monotonic evolution of state arises in Derecho and hence Vortex can be understood as another in the same family of solutions, and for many of the same reasons. Turning to the specific model we favor, Cahill *et.al.* were first to show that Snapshot Consistency could support serializability [6].
- Prior efforts to reconcile time with logical consistency models have confronted the same issue of distributed version synchronization we discussed. Google Spanner [8] solves the problem using a mix of TrueTime timestamps (similar to the timestamps we employ, but backed by atomic clocks on a network of satellites) and a delaying mechanism enforced before transaction execution. This is similar to our Δ delay but imposed over WAN links with a far larger delay factor. For us, Δ could be as small as $5\mu s$. For Spanner, delays can be in the seconds. Kulkarni proposed a form of hybrid timestamps that blend synchronized clocks with Lamport-style logical clocks [16]. We considered using hybrid timestamps, but because they only track causality over short intervals, we found them to be unsuitable in our setting.
- The Delta-T atomic broadcast protocol [9] offered temporal guarantees but lacked logical ones. The 1995 US Advanced Automation System, a prototype for a new air traffic control framework, stumbled over that issue [25], underscoring the importance of logical properties for applications requiring state machine replication of data or computational tasks.

This summary isn't exhaustive but it touches on the key results relevant to Vortex. For us, the ultimate insight is that given a problem statement requiring consistency, we now have the conceptual building blocks to assemble high-quality solutions. However, merely having building blocks is quite far from deploying them in a way that guarantees consistency for an application—particularly for an AI implemented in PyTorch or TensorFlow that potentially links stages of its dataflow computation using file passing or pub-sub message queuing. Conversely, these AIs implement computations in a functional style, and we can understand the data they write as objects appended to an immutable versioned multilog with one append-only log per shard. Appreciating this enables us to deploy our building blocks in a manner that conveys the underlying consistency model up to the ML—in a way that even benefits a pipelined AI with subtasks running on different servers in a distributed deployment.

5 Conclusion

The evolution of AI and its integration into data streaming settings is bringing a new mix of needs: AIs are increasingly distributed over multiple machines both during training and serving, and data increasingly includes streams captured from cameras and other sensors. Lacking a consistency model, applications could be exposed to unstable data with temporal sequence gaps.

To address this emerging need, Vortex offers two options. The basic framework employs a serializable, event-order preserving form of snapshot isolation that leverages the predominantly functional AI coding style, immutability of objects and monotonic growth of the stable history of the system. A second option, slightly more costly, adds a temporal stability guarantee and offers the ability to perform temporally-indexed queries; although layered on the logically consistent framework, it nonetheless exposes conflicting goals that we resolve in favor of temporality and reproducibility. Our belief is that most AI tasks would use the logical model, but visualization tasks and other strongly temporal functionality would employ temporally indexed queries. A single application can mix the two. Vortex outperforms widely popular baseline options in terms of per-request latency, variability of latency, and throughput. The performance cost of consistency is remarkably low.

Looking to the future, much more can be done to explore consistency for AIs that run in settings with highly dynamic data inflows, a need that will become pressing as the concept of AI "world models" rolls out. These include understanding how data consistency impacts model convergence in distributed AI training and inference systems, understanding concepts of optimality for AI consistency models used in training and inference at very large scale, and perhaps identifying new abstractions that might be used as world models grow to span geographic scales at which latency becomes a serious barrier. At scale, a point will surely be reached when we will be forced to consider options for tolerating faults, including adversarial behaviors that would never be seen in a single compute cluster or data center. Conversely, decentralized AI training is already revealing interesting hybrid consistency models that combine edge learning and inference with centralized integrative computation, a trend that could also be useful in other distributed applications.

Vortex is open source and can be downloaded (along with the Derecho and Cascade frameworks, on which it depends) from https://GitHub.com/Derecho-Project.

6 Acknowledgements

We are grateful to Microsoft, IBM, and NVIDIA for providing funding and resources that supported this work. We also thank Weijia Song, Tiancheng Yuan, Thiago Garrett, and Professor Christopher de Sa for valuable observations that helped shape this paper. Tiancheng Yuan additionally played a lead role in creating and evaluating the AI pipelines used in [26]. Some of the data reported in our experimental section is actually drawn from that prior paper.

References

[1] Alvaro, P., Conway, N., Hellerstein, J. M., and Marczak, W. R. Consistency analysis in bloom: a CALM and collected approach. In *Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings* (2011), pp. 249–260.

[2] Birman, K. P. *Guide to reliable distributed systems: building high-assurance applications and cloud-hosted services*. Springer Science & Business Media, 2012.

[3] Birman, K. P., and Joseph, T. A. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst. 5*, 1 (Jan. 1987), 47–76.

[4] Brewer, E. Lessons from internet services: ACID vs. BASE. archived from the original on 2008-06-24. retrieved 2008-11-06.

[5] Cahill, M. J., Röhm, U., and Fekete, A. D. Making snapshot isolation serializable. *ACM Transactions on Database Systems 34*, 4 (2008), 20:1–20:42.

[6] Cahill, M. J., Röhm, U., and Fekete, A. D. Serializable isolation for snapshot databases. *ACM Trans. Database Syst. 34*, 4 (Dec. 2009).

[7] Chandy, K. M., and Lamport, L. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst. 3*, 1 (Feb. 1985), 63–75.

[8] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS) 31*, 3 (2013), 1–22.

[9] Cristian, F., Aghili, H., Strong, R., and Dolev, D. Atomic broadcast: From simple message diffusion to byzantine agreement. *Information and Computation 118*, 1 (1995), 158–179.

[10] Eslahi-Kelorazi, M., Le, L. H., and Pedone, F. Heron: Scalable state machine replication on shared memory. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2023), pp. 138–150.

[11] Garrett, T., Song, W., Vitenberg, R., and Birman, K. Keep your friends close: Leveraging affinity groups to accelerate AI inference workflows. In *Proceedings of the 18th ACM International Systems and Storage Conference* (New York, NY, USA, 2025), SYSTOR '25, Association for Computing Machinery, p. 1–15.

[12] Hellerstein, J. M., and Alvaro, P. Keeping CALM: when distributed consistency is easy. *Commun. ACM 63*, 9 (2020), 72–81.

[13] Herlihy, M. P., and Wing, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990), 463–492.

[14] Jha, S., Behrens, J., Gkountouvas, T., Milano, M., Song, W., Tremel, E., Renesse, R. V., Zink, S., and Birman, K. P. Derecho: Fast state machine replication for cloud services. *ACM Trans. Comput. Syst. 36*, 2 (Apr. 2019).

[15] Keidar, I., and Shraer, A. Timeliness, failure-detectors, and consensus performance. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2006), PODC '06, Association for Computing Machinery, p. 169–178.

[16] Kulkarni, S. S., Demirbas, M., Madappa, D., Avva, B., and Leone, M. Logical physical clocks. In *International Conference on Principles of Distributed Systems* (2014), Springer, pp. 17–32.

[17] Lamport, L. Time, clocks, and the ordering of events in a distributed. *Communications of the ACM 21*, 7 (1978), 8.

[18] Lampson, B. W. Hints for computer system design. *SIGOPS Oper. Syst. Rev. 17*, 5 (Oct. 1983), 33–48.

[19] Lin, W., Mei, J., Chen, J., and Byrne, B. Preflmr: Scaling up fine-grained late-interaction multi-modal retrievers. *arXiv preprint arXiv:2402.08327* (2024).

[20] Mao, Z., Ellithorpe, J., Adya, A., Iyer, R., Zaharia, M., Shenker, S., and Stoica, I. Rethinking the cost of distributed caches for datacenter services. HotNets '25, Association for Computing Machinery, p. 317–325.

[21] Nagasamudram, R., Beringer, L., Birman, K., Milano, M., and Naumann, D. A. Verifying a C implementation of Derecho's coordination mechanism using VST and Coq. In *NASA Formal Methods* (Cham, 2024), N. Benz, D. Gopinath, and N. Shi, Eds., Springer Nature Switzerland, pp. 99–117.

[22] Schneider, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv. 22*, 4 (Dec. 1990), 299–319.

[23] Shivam, K., Paladugu, V., and Liu, Y. A. Specification and runtime checking of Derecho, a protocol for fast replication for cloud services. In *Proceedings of the 5th Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating Algorithms for Distributed Systems* (New York, NY, USA, 2023), ApPLIED 2023, Association for Computing Machinery.

[24] van Renesse, R., and Schneider, F. B. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6* (USA, 2004), OSDI'04, USENIX Association, p. 7.

[25] Wald, M. Flight to Nowhere: A Special Report; Ambitious Effort Becomes a Fiasco, 1996.

[26] Yang, Y., Yuan, T., Hashim, J., Garrett, T., Qian, J., Zhang, A., Wang, Y., Song, W., and Birman, K. Vortex: Hosting ML inference and knowledge retrieval services with tight latency and throughput requirements. *ArXiV preprint* (2025).