Cascade: A Platform for Delay-Sensitive Edge Intelligence

Weijia Song
wsong@cornell.edu

Thiago Garrett
thiagoga@ifi.uio.no

Alicia Yang
yy354@cornell.edu

Mingzhao Liu
ml2579@cornell.edu

Edward Tremel
etremel@augusta.edu

Lorenzo Rosa
lorenzo.rosa@unibo.it

Andrea Merlina
andremer@ifi.uio.no

Roman Vitenberg
romanvi@ifi.uio.no

Ken Birman
ken@cs.cornell.edu

Abstract

Interest in intelligent edge computing is surging, driven by improving connectivity and hardware advances. This is creating a need: today's cloud platforms optimize for high bandwidth via batching even at the price of high latency, but edge systems often must react quickly as events occur. A further challenge is that many intelligent applications are locked into ML computing frameworks that cannot easily be modified. We created a new system, Cascade, to untangle this puzzle. Innovations include a legacy-friendly storage and computing model that lets applications directly access Cascade-hosted data, data paths that leverage RDMA and GPU accelerators, and a separation of updates from queries to minimize locking and copying. Our evaluation includes cases where Cascade reduces latency by as much as 3 orders of magnitude with no loss of throughput.

1 Introduction

Latency concerns are pushing developers to consider edge deployments, where they benefit from fast network connectivity and can offer rapid responses to end users while also quickly filtering and discarding uninteresting information [23, 32, 36, 43]. The burst of excitement around large language models is accelerating this trend: Although ChatGPT is not a real-time application, it plays interactive roles in which synthesized text should reflect current state. These edge databases will become large, will receive high rates of updates, and the AIs that query them will depend on extremely rapid responses.

Any platform that responds to these needs will be further constrained by the need to maintain compatibility. Many applications are locked into big-data ML frameworks such as PyTorch or Apache. This argues for a "platform as a service" approach that enables edge-focused data hosting and execution environment but remains compatible with standard ML tools and data curation infrastructures.

The challenge is evident in Fig. 1 (drawn from an experiment discussed in Sec. 5.1). Kafka is a pub/sub tool for interconnecting microservices, and Kafka Direct is an RDMA version that sets records [10, 33, 41]. The experiment streams objects to a no-op task. Cascade, designed with these requirements in mind, is compared with Kafka Direct on the left. It maintains a tight latency distribution, while Kafka Direct exhibits huge tail latencies. On the the right we break those delays down, revealing that they arise at every layer of the componentized system. The finding is not unusual: existing data-ingress platforms run as distinct components and introduce batching as part of a strategy that optimizes throughput at the expense of tail latency (see, for example, the performance benchmarks in [4, 7, 9, 15, 33]). This will be the first of a series of "insights" about existing platforms that shaped our work on Cascade.

Cascade is a scalable distributed platform optimized for delay-sensitive edge intelligence. The core of the system is an RDMA-accelerated key-value (K/V) store. Thin APIs allow it to be treated like a file system or a pub/sub framework. Unusually, Cascade also can host user-developed logic which will run in an address space with direct access to the Cascade address space, so that data reads return pointers without doing any copying. This paper focuses on "engineering" aspects:

1. Cascade can share objects via memory-mapped segments, enabling access via pointers and eliminating copying. Objects accessed as a group are managed jointly, and jobs are scheduled close to data they access.
2. Although the APIs seen by users are standard (K/V store file system, pub/sub), Cascade's novel fast path architecture avoids interposing a separate service between stages of a multi-stage API pipeline or graph.
3. Throughout the system design choices minimize latency. Cascade has a single scheduler thread per server, queries and updates have distinct data paths, actions are batched and locking is minimized.

This paper presents the system, then evaluates applications (a messaging service and two AI/ML pipelines) to confirm that it outperforms popular baseline options.

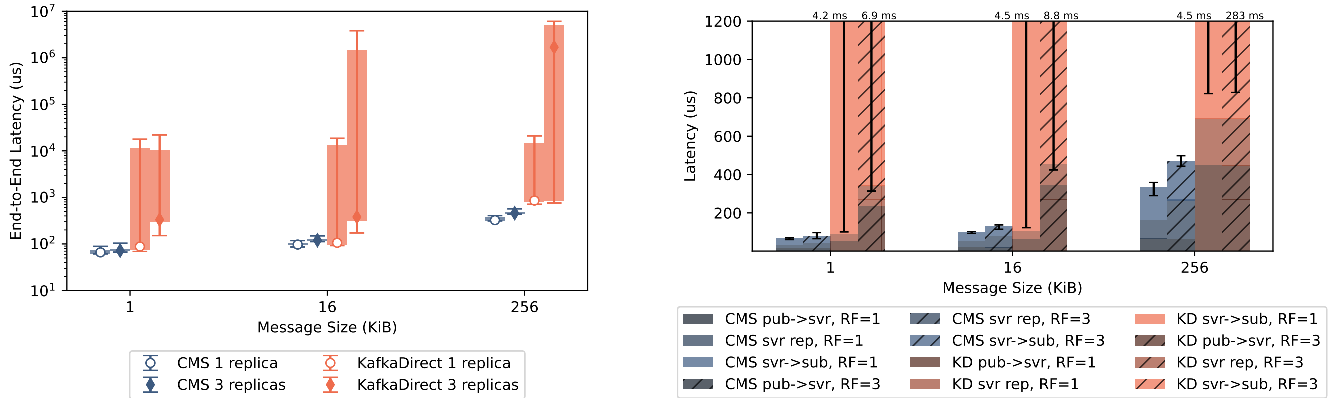


Figure 1: Comparison of the Cascade messaging system (CMS) and Kafka Direct over 100Gbps RDMA. For a single-stage no-op task, Cascade latency is less than 100us, whereas Kafka-Direct has huge tail latencies (left), and they arise at every layer (right). The notation to describe the layers and full details of the experiment are presented in Sec. 5.1

2 Intelligent-Edge Computing Today

Suppose that one were to build an intelligent edge application today. In this section we identify issues that arise due to batching, accelerators, caching, and object placement. Each leads to an insight concerning requirements that future edge infrastructures will need to address.

2.1 Stream Processing Systems

Stream processing for the intelligent edge is different from traditional batch processing [17,24,45,47]. Edge applications are *event-triggered*: as events stream in, the system continuously responds on an event-by-event basis. This makes it hard for edge applications to run on platforms evolved from batched big data infrastructures. For example Online MapReduce [16] and Spark Streaming [3, 9] were created by extending offline batched versions to support mini-batching. They offer legacy compatibility, but many events are still delayed while waiting to fill the next mini-batch. Even stream-native systems such as Google DataFlow [8], Apache Flink [14], and Apache Storm [1] often batch, incurring significant per-event delays.

Insight 1: Systems using rigid batching patterns to improve throughput incur a latency penalty.

2.2 Accelerators for the Edge Cloud

RDMA is available on cloud-based HPC clusters [40, 42], and RDMA are available on edge clusters [25]. As the cloud expands to embrace edge computing, we believe these cases will converge, yielding a single platform that can leverage accelerators. The easy option is to hide the accelerator under IP wrappers like IPoIB [30], but doing so often harms performance. To gain the full benefit, an edge platform must be end-to-end zero-copy with minimal locking, using optimized data layouts such as cache-aligned memory objects in DMA-mapped, pinned pages [10, 29, 34, 46, 48]. One option is to create a suite of new accelerated infrastructure tools, similar to Kafka-Direct [19, 26, 28, 41, 50]. The alternative we favor builds a highly efficient core infrastructure component, then maps standard APIs to this core.

Insight 2: Even if it adheres to standard APIs, an accelerated edge demands a specialized ground-up architecture.

2.3 The Edge Consistency Puzzle

Today's cloud is biased toward fast response but sometimes cuts corners by tolerating weak consistency. We see this dynamic in the first tiers of the existing cloud, which center on an elastic microservices model that makes extensive use of weakly coherent (potentially stale) caches. A cloud-integrated low latency edge will need to preserve many aspects of this popular and highly scalable model. Yet the intelligent edge brings its own priorities, including an emphasis on data freshness. Edge applications that take real-world actions often need stronger consistency: an action based on a confused perception of the external state can cause harm.

Insight 3: The edge cloud needs data freshness and consistency guarantees provided that they can be offered with low latency and high throughout.

2.4 Collocation of Data and Computing

Latency-sensitive tasks are at risk of stalling when fetching objects from storage. This is a particularly serious issue with intelligent applications: ML hyperparameters and models are often large objects, and copying them over a network will be costly even if the network is fast. ML training is so iterative that this overhead is mostly ignored: a training run will fetch objects once, but subsequent iterations will find copies in

cache. An event-triggered edge would have less temporal locality and hence lower hit rates.

Insight 4: The edge compels us to group objects that will be accessed as a set and to compute close to the data rather than moving data from storage to compute nodes.

3 Cascade Design and Implementation

These insights motivate Cascade: a combined storage and compute framework optimized for low latency and high bandwidth, and offering strong consistency. Our innovations center on the way the system is engineered and enable Cascade to host applications near the data: either directly in Cascade's address space, or in a docker container on the same compute node as one of our servers. This "fast-path" is not obligatory, but brings a huge speedup.

3.1 K/V Store and Data Organization

Cascade runs as a distributed, scalable service on a cluster of nodes equipped both for storage and computation, using a library called Derecho [26] for RDMA messaging, multicast and replicated logging using a variant of Paxos. The native storage abstraction is that of a sharded, fault-tolerant K/V store. APIs such as the POSIX/FUSE file system, pub/sub and adaptors used by language-level embedded query packages like LINQ are mapped to our native key-value mode (sometimes this is trivial, in other cases it requires more ingenuity and brings some overhead). Users can specify a replication level, in which case we replicate data across shard members using self-managed state-machine replication: every member has the full data, and can accept any job assigned to it. Should a shard become overloaded, Cascade's scheduler can also "spill over" and place tasks on members of other shards to balance load (Sec. 3.6). These servers would fetch, then cache, any needed objects.

Cascade's native API is compact. To store a K/V object `put` is invoked. To fetch one `get` is called. Both can be invoked by any node; if the caller is not in the target shard, a point-to-point send relays the request to a member. A new K/V operation called `trigger` will cause Cascade to upcall to any lambdas watching the given key. To facilitate interoperability with applications designed to read and write files, Cascade keys are strings, and can look like POSIX pathnames.

Objects are managed in *pools*. A pool is identified by a path prefix, and has a single replication factor, persistence/logging properties, and sharding policy. Each object has an opaque application-supplied vector of bytes, and a meta-data record holding version numbers and timestamps. Versioning permits Cascade to offer an exactly-once `put` semantic: Should a relayed `put` be disrupted by a failure, it can simply be reissued with the same version number. If the version already exists the duplicated request is ignored. The timestamps are useful because many edge applications track the evolution of some phenomenon over time, such as trajectories of vehicles or changing temperature and pressure in a complex piece of equipment. Cascade's `get` can be time-indexed, making it easy to express temporal computations.

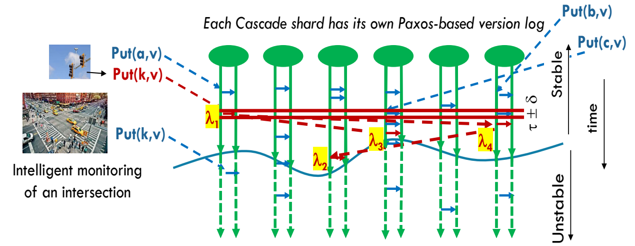


Figure 2: Serializable snapshot isolation

3.2 Computing on Cascade Objects

Cascade inherits the very strong form of consistency illustrated in Fig. 2 from Derecho. Here we see a traffic camera uploading an image. A lambda was watching for this event. It triggers, initiating a series of actions. These lambdas access data by time, enabling them to fetch images from before and after the event time τ. Lambdas accessing the same time (the red bar of width corresponding to clock precision δ) observe data along a consistent cut: a form of linearizable snapshot isolation [12]. Thus the computation will analyze a complete and consistent view of recent data. Data persists after as little as 50*us*, enabling very high-speed safety monitoring.

3.3 Fast Path Lambdas

The traffic camera example involved a series of user-supplied lambdas. For example, perhaps λ_1 does a quick scan of the scene and discards completely empty images, or ones that cannot be analyzed due to extreme weather. For interesting images, λ_3 and λ_4 run to detect pedestrians, cars, buses, bikers, etc. This leads to triggering λ_2 (potential collisions). This pattern is typical of the sequences of intelligent computations Cascade is designed to host: each lambda implements some form of application logic or artificial intelligence, yielding a data-flow graph (DFG) in which each directed edge represents a flow of objects from one lambda to another.

A lambda can be quite elaborate: our experiments package the image classifier YOLO as a lambda, and ChatGPT could also operate this way. Some AI lambdas already take inputs as command line arguments and from files and need no modifications; others can be ported by building "helper" programs and still others can be recompiled to access our native K/V layer directly. For this purpose, they would use our native C++ APIs, or access them via wrappers from Python, Java, or C# dotnet.

Cascade can trigger a lambda in three ways. (1) In the first case, the developer intends to trigger some lambda but

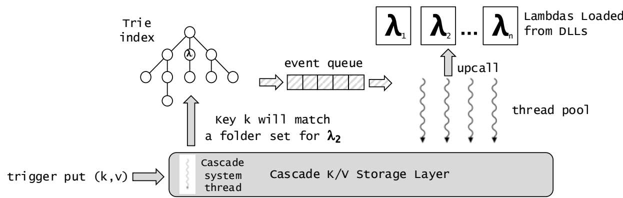


Figure 3: Fast path in a Cascade Node

without saving any data. For this purpose, `trigger` is used; it has the usual K/V arguments, but is supported as a point-to-point asynchronous send. (2) A volatile `put` atomically replaces the current version (if any) of an in-memory object replicated with some shard. Here, the triggered lambda will run after these updates occur. The developer selects between two options: we can do the upcall on one node or it can occur on *all* shard members. (3) A persistent `put` uses Derecho's Paxos protocol to update a versioned, logged object.

Fig. 3 shows details of this design for a single Cascade server, illustrating the case where the user's logic is packaged into a DLL with upcallable functions. On the bottom left data flows into Cascade K/V storage layer. These updates correspond to `put` and `trigger put` operations, so the system thread checks for a match between the key and locally registered lambdas (discussed in Sec. 3.1). When a match is found an event pairing the data and the lambda is enqueued. Threads in the lambda upcall pool monitor this queue, removing elements and doing upcalls. The key and value are passed as C++ shared pointers, avoiding copying.

Our containerized colocation option is not yet finished, but is designed to mimic the DLL approach. The user's logic will run in a docker container. It accesses Cascade either by treating it as a file system, pub/sub layer or K/V system, or via a version of the "internal" API we offer to DLLs. Data is moved between the container and Cascade using a pair of shared-memory segments. One, restricted to read-only access, selectively reveals Cascade-hosted data to the application. The other lets the application send data to Cascade. Both include pools of 2MB pages; the R/W one additionally holds two lock-free FIFO queues, one for requests by the application, and one for responses and notifications. In preliminary experiments, DLL hosting for lambdas is slightly faster, but less secure. The containerized approach benefits from better security, but system calls are slightly more costly: RPC though our shared memory FIFOs incur round-trip latencies varying from about $0.5us$ to slightly more than $1.5us$, depending on the server we test on.

3.4 Minimizing data movement

Throughout Cascade, the need to pass objects from place to place risked causing a great deal of copying. Reasons include data marshalling, the need to collect data from various memory locations, and cacheline or page alignment needs [27].

One example of this arises with containerized AI lambdas. With these, Cascade leverages the shared page pools described above. We share large page-aligned memory-mapped objects by remapping pages via `mremap`, so that they appear to reside in the page pool, yet are actually read-only duplicates of pages that really reside in Cascade itself. We can do the opposite as well: data that really lives in the container client can be made visible for Cascade to read, or even for it to use as the origin of a direct RDMA transfer. With the most recent NVIDIA GPUs, we use `cuMemMap` to share GPU memory pages.

Another example of how we avoid unecessary copying arises with in-place C++ object construction. Here we seek to avoid copying some structure or multifield object. The classic issue is handling of hardware-specific byte ordering and alignment. Our approach assumes that servers will all use the same native data layout rules. External clients respect server policies even if they differ from the native ones, potentially paying a cost similar to standard marshalling if they run on hardware that favors some other native data formatting. This enables RDMA transfers without marshalling or copying. Cascade publishes the service's desired byte layout and alignment when an external client connects, enabling clients to preconstruct efficient marshallers for native types at binding time. For small amounts of data, the costs will be negligible. Large objects like photos or video often have architecture-independent encodings; here, the main need is to ensure that when such an object is first uploaded by the device, it will already be saved in a suitable memory region. Thus it is not necessary to marshal objects.

Although this is not always feasible in a back-compatible and fully transparent manner, all applications using Cascade APIs can avoid copying by having a message sender asynchronously allocate memory within the message buffer region Cascade employs for communications, which is pinned and preregistered for RDMA transfers. The caller supplies the desired constructor as a function. When space becomes available, Cascade upcalls the constructor on a thread that is permitted to block. The client can either construct the object in place immediately, or delay until data is available (for example a client managing a camera may need to wait until motion is detected or a new photo is acquired). When the object is ready to send, the constructor returns control to Cascade, which marks the message as complete. Meanwhile, Cascade's core thread polls "under construction" message slots. The next time it checks, Cascade will notice that a message (or group of messages) is now ready for transmission, and will enqueue an RDMA send that respects the FIFO order in which the message memory slots were requested.

3.5 Collocation of Related Data Objects

The next question focuses on the locations at which data is stored and where computation will occur. Most K/V sys-

tems hash the object keys to obtain shard numbers. Hashing randomizes object placement onto the available shards. The concern is that if a lambda depends on multiple objects, randomized object placement would defeat the goal of collocating computation with storage. ML models can be huge, so the importance of collocation is significant – with a 10GB model, even transferring at 100Gb/s takes one second.

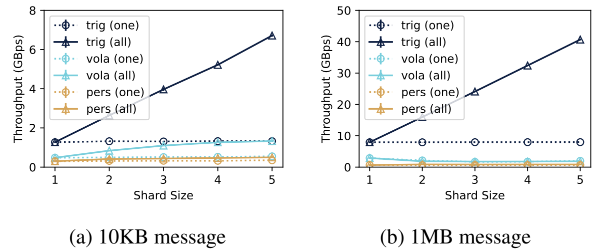
With this in mind, Cascade allows developers to associate an *affinity set key* with each object: a second pathname. If none is specified, the object's name is its affinity key. Objects can still be accessed one by one, but if desired, objects that employ the same affinity set will be collocated, moved, cached in memory or evicted as a set. They can also be updated as a set: for example, an ML system that refines its model might also update hyperparameters: two distinct objects, but they will have the same affinity set key. In support of such cases, `put` also offers an API that will send a list of new objects sharing an affinity key as a single message. Cascade's `put` implementation is single-threaded, hence the set of updates will appear to happen atomically.

Even with affinity groups, objects still have unique names. The affinity key is only used to ensure co-hosting. If desired, the Cascade *meta-data store* can be queried to obtain the affinity key associated with a given object. The meta-data store is also able to list all objects associated with an affinity set, to track and query access patterns, and to track cache status for sets that are cached outside their home shard.

3.6 TIDE scheduler

Cascade makes scheduling decisions at several stages of event and event-stream processing. Consider a data-flow graph (DFG) representing application data flows and data dependencies (an example can be seen in Fig. 9 in Sec. 5). A single DFG can represent processing of a single event, but can also describe an event stream, and it is even possible to create a form of template that will be instantiated once per stream. Moreover, an application might use many DFGs. Within these instantiated DFGs, each computational task – each lambda – must be mapped to a node that will perform that task. TIDE's job is to compute an optimized, load-balanced plan that will be replicated onto client systems and used to select the target when a `trigger` put occurs. TIDE also manages caches, both for host memory and CUDA memory on GPU accelerators. The broad goal is to ensure that all the inputs a task requires are available on the node where the task is launched, so that there are no object-fetch delays on the critical path.

TIDE is a sophisticated subsystem, and the constraint of brevity precludes a full discussion of the optimization techniques employed, or a full evaluation of TIDE's effectiveness. Accordingly, we defer a more detailed presentation of the algorithm for future papers: one focused on the single-event case, and a second on an extension we call MTIDE that focuses on scenarios in which a pipeline is instantiated multiple times, resulting in a large number of side-by-side event flows.



(a) 10KB message

(b) 1MB message

Figure 4: K/V Store Put Throughput

`PUT type`	10KB Message	1MB Message
Trigger	12 μs	220 μs
Volatile	70 μs	1100 μs
Persistent	500 μs	4200 μs

Table 1: Typical Put Operation Latencies

4 Evaluation

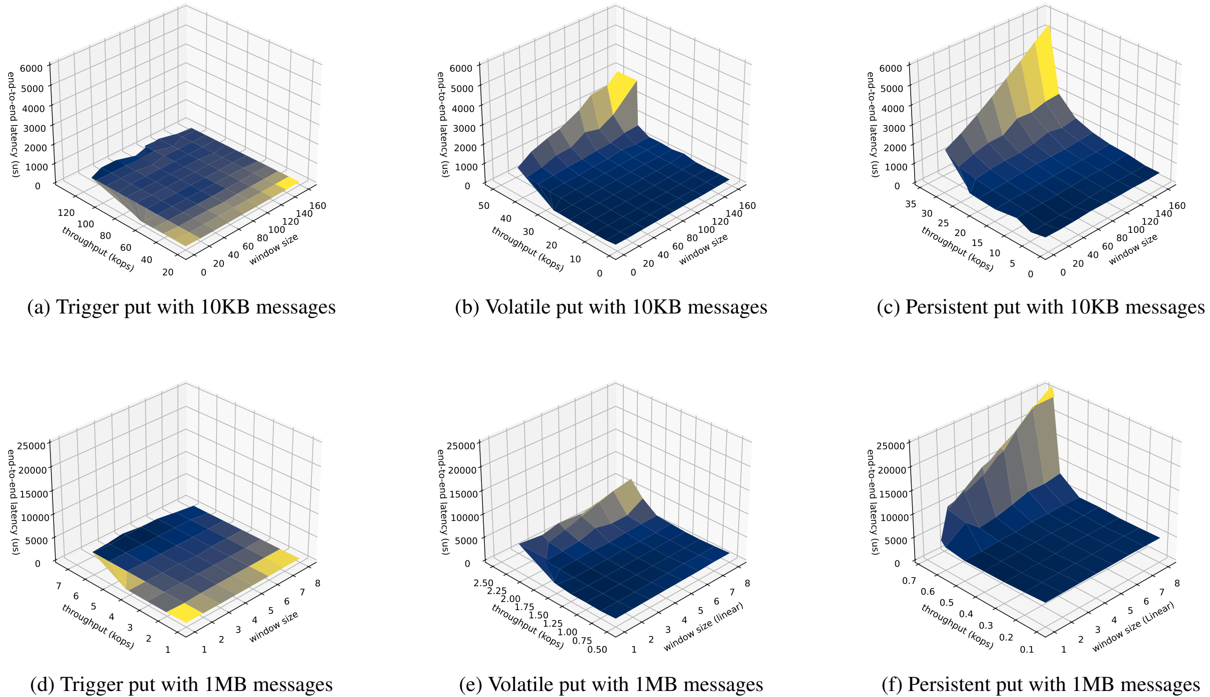
In this section, we start by reviewing microbenchmarks that evaluate the throughput and latency of the Cascade K/V store. Next, we evaluate performance of a data pipeline composed of multiple put operations, employing no-op actions to highlight overheads. Finally, we evaluate three applications.

Our servers are dedicated nodes with Mellanox ConnectX-4 VPI NIC cards connecting to a Mellanox SB7700 InfiniBand switch, which provides an RDMA-capable 100Gbps network backbone. The servers have two configurations. The more powerful configuration matches what edge-hosting systems typically offer; these have dual Intel Xeon Gold 6242 processors, 192 gigabytes of memory, and an NVIDIA Tesla T4 GPU. The lightweight setup emulates a less powerful Cascade client; it has two Intel Xeon E2690 v0 processors and 96 gigabytes of memory but no GPU. For convenience, we use **type A** to denote the larger configuration and **type B** for the basic one. Both types of servers have high-speed NVMe storage (Intel Optane P4800X cards).

All servers run Linux kernel 5.4.0. We synchronize the server clocks with PTP [22] so that the skew among them is in the microseconds, allowing comparison of the timestamps from different servers with sub-millisecond precision.

4.1 Cascade K/V Store Performance

We started by evaluating the performance of Cascade's three types of `put` operations (Sec. 3.3). Clients run on type B servers, issuing requests to Cascade nodes on type A servers at a controlled rate. Then, we varied the shard size as well as the number of clients to test scalability. For some experiments, we used just a single client irrespective of the server shard size (`one`), while others had multiple clients, one per shard member (`all`).



(a) Trigger put with 10KB messages
(b) Volatile put with 10KB messages
(c) Persistent put with 10KB messages
(d) Trigger put with 1MB messages
(e) Volatile put with 1MB messages
(f) Persistent put with 1MB messages

Figure 5: Cascade's K/V Store Put Latency is low over a wide range of message sizes and data rates.

4.1.1 Throughput

Fig. 4 shows the `put` throughput of Cascade K/V store as we vary the shard size (the number of replicas). The Y-axes represent the throughput seen by the application. With only one node in a shard, volatile `put` reaches $\sim$ 500 MBps (50 kops) and $\sim$ 2.8 GBps (2.8 kops) for 10KB and 1MB messages. The throughput for 10KB messages is steady as we vary the shard size from 1 to 5. With 1MB messages, both figures drop slightly: $\sim$ 2.2 GBps (2.2kops) for shards of size 5, reflecting the overhead of replicating large messages.

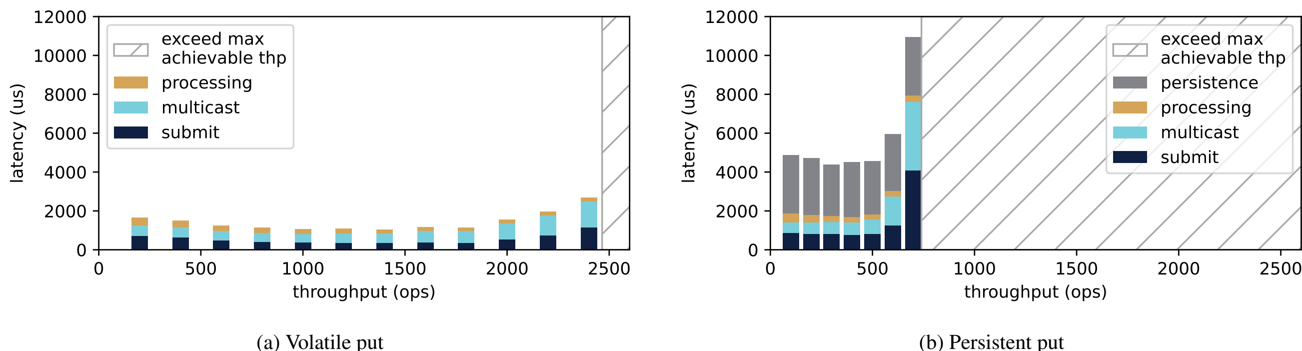
With multiple clients, we achieve even higher throughput. Volatile `put` with 10KB messages rises from $\sim$ 500 MBps (25 kops) to $\sim$ 1.3 GBps (130 kops), a figure at which the replication capacity of the system becomes saturated. In contrast, with 1MB messages, even with multiple clients, throughput remains flat, peaking at $\sim$ 2.7 GBps for 5-member shards. Our studies suggest that the bottleneck is associated with a `memcpy` operation that we use to copy data from the RDMA buffers used for incoming messages to a heap where we store objects that will be passed to developer-supplied lambdas.

The numbers for persistent `put` operations show similar trends, but the actual bandwidths are sharply reduced. Persistent `put` reaches at most $\sim$ 270 MBps (27 kops) for small messages and $\sim$ 800 MBps (800 ops) for large ones. The bottleneck turns out to be a side-effect of the Paxos-based consistency model used in Derecho. Although our NVMe device can achieve sequential write bandwidth of 2.4 GBps, this data rate is only achievable with long, uninterrupted DMA transfers. It turns out that in the persisted mode our update workload incorporates ordering dependencies that the storage layer enforces by periodically pausing until persisted updates are completed, disrupting the DMA transfer scheme (we plan to look at ways of aggregating such actions opportunistically, but this is future work). `Trigger` put operations scale best because these operations avoid all memory copying and replication overheads. A `trigger` put client gets $\sim$ 7.6GBps ($\sim$ 7.6 kops) for 1MB messages, which is close to the RDMA hardware limit, and aggregated throughput grows linearly in the number of clients.

4.1.2 Latency

Table 1 explores `put` latency for our three modes, considering small and large messages. For each volatile `put` operation, we measured time starting when the client sends the request, and ending when all replicas finish updating their in-memory store. Each data point in the figure shows the average latency during a five-second period. Similarly, for each persistent `put` operation, we measure time from when the client first sends the request to when the last replica finishes persisting it;. For `trigger` put we measure until the request reaches a replica that upcalls to a developer-supplied lambda.

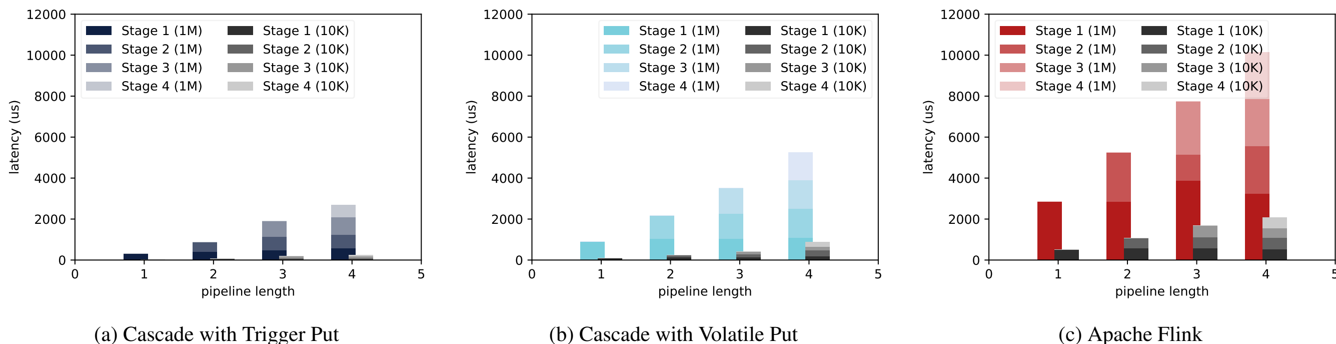
The persistent `put` latency is about four to five times higher



(a) Volatile put

(b) Persistent put

Figure 6: Latency Breakdown for Volatile and Persistent Put



(a) Cascade with Trigger Put

(b) Cascade with Volatile Put

(c) Apache Flink

Figure 7: No-op Pipeline Latency

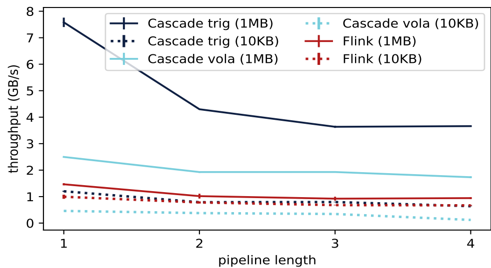


Figure 8: No-op Pipeline Bandwidth

than that of a volatile put. Below, we confirm that the bottleneck is the I/O to our storage devices. Trigger put is one order of magnitude faster than volatile and persistent put because it does not need to replicate or persist any data.

The data in Table 1 reflects performance when the system is not saturated. As the workload is increased and begins to approach the maximum sustainable throughput, latency will rise sharply and without limit. To quantify this effect, we measured the end-to-end latency of Cascade K/V store with three replicas in Fig. 5. The six subfigures show the end-to-end latency for the three operation types and two different message sizes. We control the maximum message rate on the client-side and the Cascade window size (a multicast flow-control parameter) to see how the latency changes.

As shown in figures 5b, 5c, 5e, and 5f, before the system becomes saturated end-to-end latency is consistently low, corresponding to the flat part on the near right part of the curved surfaces. The system is keeping up with the request rate, so there are no queuing backlogs: requests are processed immediately. As the workload grows we see the latency suddenly rise, corresponding to the slope part on the far left part of the curved surfaces. Here, processing becomes bursty and queuing delay dominates the end-to-end latency.

Fig.s 5a and 5d show that the trigger put latency is insensitive to workload and window size. This is a consequence of using a no-op as the triggered action: If we used a lambda that performed a more realistic computation, the computing cost would dominate the end-to-end latency. We will see this effect when we evaluate our dairy image processing pipeline.

Fig. 6 shows the latency breakdown for the volatile and persistent put. We use a setup with a shard of three nodes and a client that uploads 1MB objects. The window size is three. The *submitting* component refers to the latency between the client serializing a put request into the sending buffer and the Cascade server receiving it; the *multicast* component is

the latency of replicating the data among the shard members; the *processing* component is the time spent in updating the in-memory state, and the *persistence* component represents the time between initiation of an update and its persisted commit (across all replicas). Because `trigger` put has only the submitting component, we exclude it in this figure.

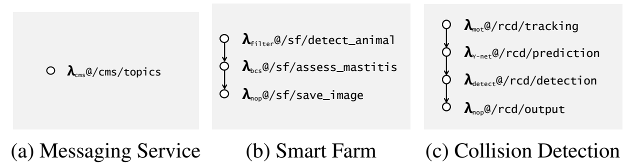
All options have a wide range of message sizes and data rates for which per-event latency remains stable and very low. For volatile `put`, the multicast and submitting components account for most of the end-to-end latency. For persistent `put`, the persistence overhead dominates. When the request rate approaches the maximal achievable single-event throughput, the overhead of submitting and multicast grows suddenly because the messages pile up. In the persistent case the effect is exacerbated by a stage in Paxos that syncs data to storage.

4.2 Pipeline Performance

To microbenchmark a fast path pipeline we created a series of lambdas that relay received data stage-by-stage but perform no other computation, implemented with a `trigger` put or a volatile (multicast) `put`. The shards have three members each: one running on a type A server and two on type B servers. Each lambda runs in a single member, on a dedicated A-type server. The client program runs on a type B server node. A first experiment examines latency from the client to the no-op for 10KB and 1MB messages sent at a low request rate. In Fig. 7a and 7b we show the average latency during a representative 5-second period for varying pipeline lengths. The first-stage numbers match the trigger and volatile put latencies in table 1. In longer pipelines `trigger` is faster, but the overhead of volatile `put` is surprisingly low.

For purpose of comparison, we then configured Apache Flink to mimic Cascade by having it use a single task-processing slot per server, disabling automatic operator chaining [6] to prevent it from merging the tasks. We recompiled Flink to load 1MB at a time (the defaut is 32KB), and set its minibatch delay barrier to zero. This last change goes beyond the norm but without it performance was terrible and the servers had very low CPU utilization levels. Yet even with all of this tuning, Flink's pipeline latency is high (Fig. 7c). The Cascade pipeline with trigger put has a latency below one-eighth that of the Flink version for 10 KByte messages and one-fourth for 1 MByte messages. Indeed, even the (replicated) volatile put on three-member shard has less than half of the latency of Flink, at both messages sizes. Two factors account for this: Flink runs on TCP, not RDMA, and it uses the Java-based Kryo serializer, which copies from Java-managed memory to a network buffer.

We then stressed each pipeline by streaming at the maximum sustainable message rates. This yields the first four throughput series in Fig. 8. Again, the throughput of Cascade's one-stage pipeline matches the trigger and volatile `put` throughput in Fig. 4, dropping slightly as we move to a pipeline with two or more stages. This reflects the extra costs associated with message relaying: the first stage only sends, while inner pipeline stages must send and receive, and the final stage only receives. Performance is sustained as pipeline length grows from two to four, supporting our claim that Cascade scales extremely well. In the same experiment, Flink gives lower and more variable throughput.



(a) Messaging Service (b) Smart Farm (c) Collision Detection

Figure 9: Application DFGs

5 Cascade Applications

We implemented three applications to explore the overall effectiveness of Cascade when compared with existing ways of solving the same problems. We made a major effort to be fair to the comparison platforms, and to configure them exactly as recommended by their developers.

5.1 Cascade Messaging Service

Our first application is the *Cascade Messaging Service (CMS)* that was compared to Kafka-Direct in the introduction. CMS employs a standard Pub/Sub model but maps publish to atomic multicast, enabling strong ordering and fault-tolerance semantics. As seen in Fig. 9a, the CMS DFG has just one vertex: a lambda λ_{cms} binding to folder */cms/topics*. A CMS client publishes to a topic T by calling `put` with key */cms/topics/T*, and will be either volatile or persistently logged at the developer's option. λ_{cms} does no computation; instead, it pushes a notification (including the object data) to clients subscribed to T. λ_{cms} allows concurrent upcalls for distinct topics (any single topic retains FIFO event ordering).

Recall from Sec. 4 that our experimental setup has two categories of servers. For this experiment we hosted both platforms on type A servers, and configured Kafka-Direct as recommended by the developers, with publisher, subscriber and server nodes on distinct machines. To create Fig. 1 (left side) we disabled intentional batching to prioritize latency over throughput (nonetheless, both batch if a backlog forms). To avoid bottlenecks in the persistent storage layer, λ_{cms} uses a volatile folder, while KafkaDirect's log data was stored in ramdisk. We then varied the data rates at each object size. For each size we were able to identify rates that minimized latency for both systems. We then plotted the median (circles), 10%-90% range (box) and an error bar for the 1%-99% range. The right side of the figure breaks delay down by the stage at which it arose.

Fig. 10 shows latency at each presented data rate, but only for CMS; we see steady low latency as long as the system can keep up event-by-event, and then a shift to a batched behavior as backlogs begin to form. We were unable to create a similar graph for Kafka-Direct: latencies were highly variable at every offered rate, consistent with our finding that this Pub/Sub broker becomes bursty even at very low throughput.

We invested significant effort to understand why Kafka-Direct is so prone to queuing. Unfortunately, the root cause is not evident, but it has the effect of delaying the Kafka-Direct RDMA polling thread: Quite frequently,100ms or more passes between rechecking the RDMA message queue (we instrumented the code and have data supporting this observation, over a wide range of operating conditions, and it arises in an unmodified download of the broker configured precisely as recommended, on hardware identical to what they used for their own testing). During these long pauses, queuing backlogs form, resulting in delivery of a batch of delayed messages. Our hypothesis is that the developer team didn't view this as a problem because throughput was their priority – and Kafka-Direct throughput is quite good (this said, Cascade is faster). In future work, we plan to extend our CMS into a drop-in-compatible replacement for Kafka.

5.2 A Smart Farming Application

We also built a dairy health tracking application. The system images animals as they enter or leave the milking area, then employs ML to develop a variety of information streams that are reported via dashboards for farm workers, owner and vets. We obtain a multi-stage pipeline. The first step uses a motion detector and RFID to capture photos and identify the cows. In the second step, selected images are subjected to a full analysis, after which web tools generate dashboard reporting.

Two vision models are involved in this application: a filter model that determines whether a photo has a valid animal image, and a body-condition scoring (BCS) model called CowNet [2] that assesses the health condition of the animal. We developed the λ_{filter} and λ_{bcs} using TensorFlow's C API

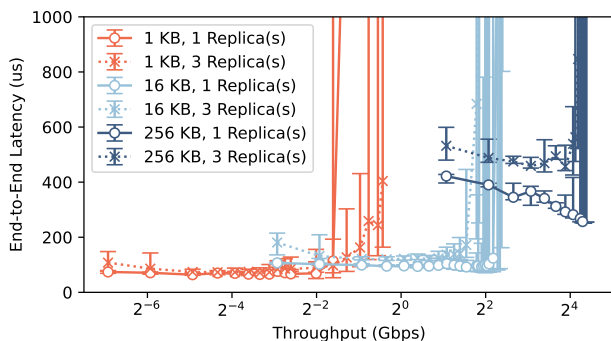


Figure 10: CMS latency: Detailed breakdown

to incorporate the two models. A storage stage is represented as a no-op lambda, giving the three-stage DFG shown in Fig. 9b. Both $/sf/detect_animal$ and $/sf/assess_mastitis$ folders are volatile, since not all original photos are worth saving. The $/sf/save_image$ folder is used to retain photos along with the computed body condition score, RFID, and timestamp. These lambdas do not have any constraint on ordering, hence the upcall pool is configured to permit concurrent upcalls. Nodes in the front-end and compute stages perform image analysis and run on servers of type A. The other nodes run in the type B servers. Each node runs in a dedicated server to avoid resource conflicts.

We first deployed the application with a simple configuration where each stage of the pipeline runs on a single-member shard. We use cow images collected from our research dairy, each with a valid cow image pre-verified by the filter model: it needs to run, but will always select every image for further analysis (this is to avoid variable-length computations that would make the output harder to understand). The raw image size in JPEG format is about 200 KBytes. At the beginning of the experiment, the photo aggregator transforms the raw images into two-dimensional dense arrays in OpenCV `cv::Mat` format, which can be used directly by our inference engine. Interestingly, although this dense array format is larger (about 1 MBytes which is five times the JPEG format), it saves the computation resources and supports faster decision-making, reducing CPU and GPU performance pressure but at the cost of higher data movement costs. The storage folder is volatile, meaning that the most recent version of each object will be held in memory but not persisted.

We then built the same application as a Flink pipeline for comparison. It consists of a photo streaming data source task as the photo aggregator, four filter tasks, four bcs tasks, and a terminating data sink. We use Flink's fine-grained resource management to control the task layout so that tasks of the same type will run on the same server. Moreover, the filter and bcs tasks are placed in Type A servers because the models require GPU resources. To mimic Cascade in-memory storage of the results, the Flink data sink task saves output into an in-memory hashmap. Once again we see that even though Flink's data storage sink is non-replicated and the Cascade version replicates the output, Flink is substantially slower.

Latency Breakdown In this experiment we selected fixed sending rates in the range from 50 to 400 frames per second(fps). For each rate, we run a session that lasts for (at least) five seconds and log the timestamps for each photo at different stages in the pipeline. We recorded the following six timestamps for each photo: (1) the photo aggregator sends a photo; (2) the filter lambda is triggered; (3) the filter lambda terminates; (4) the bcs model is triggered; (4) the bcs model terminates; (5) the result is written by the store folder.

The gray bars in Fig. 11 show the results for Cascade at a low rate and then at the highest sustainable rate (the limiting factor turns out to be the filter and bcs model costs,

which fully load our type A servers). The end-to-end latency is only six milliseconds for both the light workload at ~ 50 fps and the stressed workload at ~ 400 fps. Even when the workload grows to a stressed 650 fps, it only rises slightly to 6.5 milliseconds. The time spent in model inference dominates end-to-end latencies: filter processing time represents about 40% of end-to-end latency; and bcs model processing is even greater, at nearly 43%. The aggregated data forwarding latency represents just 17%, reflecting the efficient fast path.

The red bars in Fig. 11 show the Flink latency breakdown. Although the filter and bcs models consume slightly more time, Flink's data forwarding delays (highlighted with stripes) are far higher. For a stressed load at 400 fps, Cascade's end-to-end latency is about one-eighteenth that of Flink's. Even with light load at 50 fps, Flink's end-to-end latency is 25 *ms*, whereas Cascade is just 6 *ms*, a 75% reduction. The peak achievable Flink throughput was ~450 fps, hence there is no Flink data point for 650 fps.

Throughput Scalability We investigated scalability by varying the number of nodes while tracking throughput. Here, the configuration of the shard responsible for each stage has a significant impact, so we use a tuple to represent a system configuration, where the elements are positive integers representing the number of nodes assigned to each role: frontend (which runs the filter lambda), compute (which runs the bcs scoring lambda). Since the storage is not a performance bottleneck here, we keep the same storage tier set-up as in the previous section (three nodes in the storage shard) in this experiment and skip it in the tuple. For example, $(1,2)$ represents a system configuration with six nodes. The frontend folder is backed by a shard with one node; the compute folder is backed by a shard with two nodes; and, not shown in the tuple, the store folder is backed by a shard with three nodes. Cascade supports a variety of load-balancing policies, including random, static, and round-robin; we selected round-robin. We then graphed the maximum throughput achievable without overloading the pipeline in Fig. 12. For context we benchmarked both lambdas on a single server of type A: filter runs at ~ 900 fps, while bcs runs at ~ 700 fps.

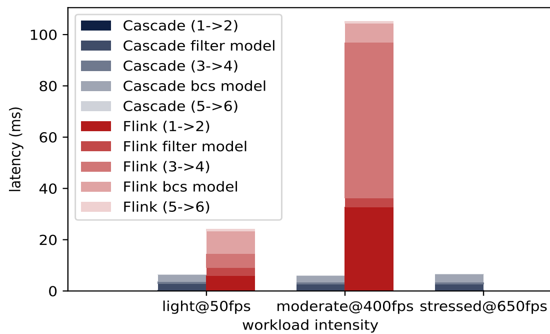


Figure 11: Smart Dairy Latency Breakdown

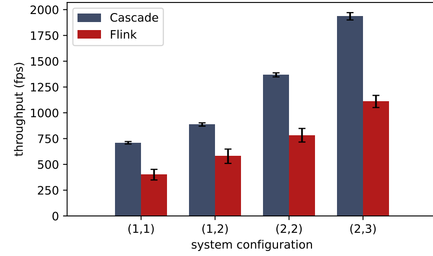


Figure 12: Txns/sec versus stage size (front-end,compute).

Figure 13: Traffic at a busy intersection

The overall trend for Cascade, shown dark gray bars in Fig. 12 is easily understood. The bcs lambda is the bottleneck in the $(1,1)$ configuration. In the $(1,2)$ configuration, the bcs lambda has adequate capacity because it runs on 2 nodes, causing the filter lambda to emerge as the limiting factor: we obtain a maximum throughput of ~ 900. With configuration $(2,2)$ bcs is again the limit. With the $(2,3)$ configuration, the two lambdas are balanced and throughput exceeds one thousand fps. Broadly, these results support our view that Cascade has excellent scalability. We repeat the same experiment using Flink (red bars). Cascade turns out to outperform Flink by $\sim 40\%$, reflecting the benefits of our fast-path architecture.

5.3 Real-time Collision Detection

Next we created a more sophisticated application that monitors city intersections (Fig. 13). We analyze video streams that include several types of agents (pedestrians, cars, cyclists, skateboarders, buses, and golf carts). For each frame, the solution extrapolates agent trajectories to detect imminent collisions a few seconds before they occur, using off-the-shelf ML models in a our stage pipeline (Fig. 9c). The first stage, λ_{mot}, runs a Multi-Object Tracking (MOT) ML model to track trajectories in a video stream. λ_{YNet} predicts each trajectory for the next 4.8 seconds based on the past 3.2 seconds. λ_{detect} predicts potential collisions and requires consistency: stale data could disrupt the algorithm. The final stage stores output.

We implemented λ_{mot} in Python using a multi-object tracker available in [11], which employs a combination of YOLO5 for agent detection, and StrongSORT [21] and OSNet [49] for trajectory tracking. λ_{YNet} was also imple-

mented in Python, using a trajectory prediction model called YNet [35]. Both λ_{mot} and λ_{YNet} use PyTorch. We implemented λ_{detect} in C++. It consists of a simple algorithm that performs a linear interpolation on each trajectory prediction and checks if any interpolated pair of trajectories in the same frame crosses each other: a potential collision. All ML models used in λ_{mot} and λ_{YNet} are trained with the Stanford Drone Dataset (SDD) [39]. In our application, we treat each video as a stream originating at a camera hosted by a distinct client.

All folders are volatile, since data generated will not be accessed for longer than a few seconds after the events happened. The lambdas are sensitive to data consistency: the position of an agent in past frames influences detection and trajectory prediction during the analysis of subsequent ones.

Next, we evaluated the traffic safety application. Ideally, computations (λ_{mot}, λ_{YNet}, and λ_{detect}) should dominate the end-to-end latency of each video frame. λ_{mot} was deployed on one shard containing two type A servers. λ_{YNet} was deployed on one shard containing six type A servers. Since λ_{detect} does not require GPUs, it was deployed on one shard containing three type B servers. Finally, we deployed three clients on type B servers; each streams one randomly selected video from the SDD dataset. Note that this pipeline triggers no replicated updates: it uses `trigger` puts when transmitting data from one stage to the next, hence there is no trade-off between the number of shards and shard size.

We identified an opportunity for parallel event handling (Fig. 14). Each frame triggers λ_{mot}, which detects agents in the frame and match them with agents detected in the previous frame from the same client. The new position of each agent in the new frame is sent to λ_{YNet}, triggering separate λ_{YNet} instances for each new position. These can run in parallel, limited by the λ_{YNet} six-node deployment. Therefore the workload on the second stage depends not only on how many clients are streaming frames, but also on how many agents are detected in each frame. The size of messages containing new positions is negligible, in the order of 10s of bytes.

Clients stream videos in a rate of 2.5 frames per second, since YNet was trained under that framerate. Each frame is sent uncompressed to the first stage of the pipeline, and has a size of roughly 8MB. According to our evaluations, sending compressed frames would reduce their size to the order of 100s of KBs. Although the transmission delay of smaller frames is lower, uncompressing them in λ_{mot} results in a much higher total delay. We executed the application with all three clients streaming simultaneously for approximately 5 minutes. The first 30 seconds worth of frames from each stream were discarded to allow a warmup, leaving an aggregated total of 2073 frames. Fig. 15 shows the frequency of agents across all frames (λ_{mot} typically detected 7 to 16 agents per frame).

Of particular interest is the end-to-end latency of frames, and the cost of Cascade lambdas. Fig. 16 shows the average latency breakdown per frame. We aggregated frames according to the number of agents detected, in groups of five, as

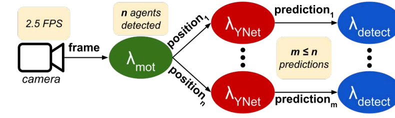


Figure 14: Pipeline triggered by a frame.

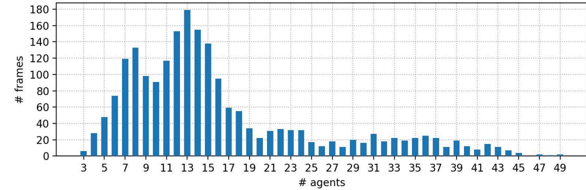


Figure 15: Histogram of agents across 2073 frames.

indicated in the vertical axis. The topmost group corresponds to all frames. The horizontal axis indicates latency in milliseconds. Each horizontal bar corresponds to the average end-to-end latency for a lambda in frames containing the indicated number of agents. In a frame, the end-to-end latency for λ_{mot} corresponds to the time since the lambda started executing (with the frame already available in memory), until all agents in the frame are detected and their trajectories computed. Since there are multiple agents per frame, the end-to-end latency for λ_{YNet} corresponds to the time since the first instance of this lambda starts, until the last instance finishes. Multiple instances are executed, generating predictions in parallel. Thus multiple instances of λ_{detect} start while there are still instances of λ_{YNet} running. The end-to-end latency for λ_{detect} is the time since the first instance started, until the last instance finishes. The gap between time 0ms and λ_{mot} corresponds to the frame transfer from the client. The gap between λ_{mot} and λ_{YNet} includes the transferring of the new agents positions, as well as waiting time due to the servers being busy executing λ_{YNet} for the previous frame.

As expected, latency is higher in general for larger workloads (indicated by the number of agents). However, it is possible to see that the time between the frame is sent by a client until λ_{mot} starts is consistent regardless of the workload: the average is 28ms, with a 95th percentile of 34ms. Similarly, the overhead between λ_{mot} and the first instance of λ_{YNet} is consistent, about 5ms. Focusing on the frames containing 11 to 15 agents, the average end-to-end latency for the pipeline is 229ms, with a 95th percentile of 264ms. Considering all frames, the latency is much less consistent with an average of 241ms and a 95th percentile of 338ms. This variance stems from lambdas that require more processing time when the workload is heavier. We conclude that Cascade incurs a low and consistent overhead in the critical path.

We also deployed our application in Microsoft Azure Cloud. Adhering closely to documentation [38], we created a pipeline that uses Azure Machine Learning (AML), Stream Analytics (SA), and Event Hubs (EHs). The application lambdas

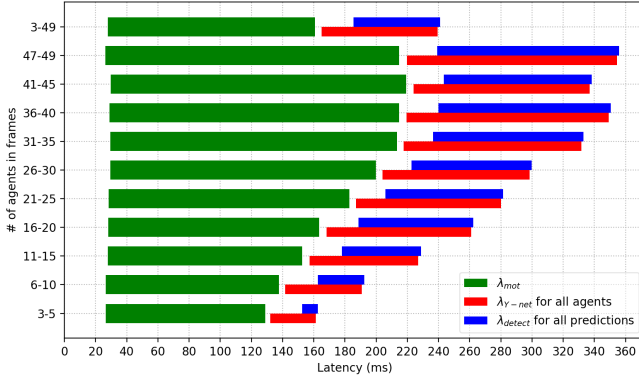
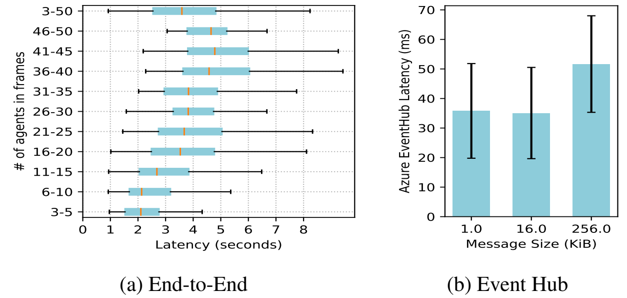


Figure 16: RCD average per-frame latency breakdown.



(a) End-to-End

(b) Event Hub

Figure 17: Latency on Azure Cloud

were deployed as real-time endpoints (which behave as web-services), triggered by SA jobs connected by EHs [31].

RDMA is widely used in Azure's own infrastructure components, although we were not able to determine whether RDMA is used in the specific Azure components we used. Accordingly, the goal of this experiment is really to illustrate expected performance in the absence of a technology like Cascade – we do not think there actually is an existing cloud technology that can be used in a totally fair comparison. In future work we plan to experiment with Cascade hosted on Azure HPC (a costly configuration, but one that allows third-party platforms to leverage RDMA).

In this experiment, we employed only one camera. Videos were pre-loaded in Azure Blob Storage [13]. The simulated camera ran in a virtual machine, triggering the application pipeline by sending frame metadata to an EH at a rate of 2.5 fps. A front-end SA job consuming the metadata invoked λ_{mot}, which downloaded the corresponding frame from the Blob storage and performed object tracking. Results were sent to the next job through another EH. The third job was triggered similarly. The results of the last job were sent back to the camera virtual machine for end-to-end latency measurement.

We used seven *Standard_NC4as_T4_v3* AML instances equipped with the same GPU as in our local environment, one for λ_{mot} and six for λ_{YNet}. Three more instances (type *Standard_DS3_v2*) were deployed for λ_{detect}. We employed premium-tier [37] EHs, and SA jobs ran in a dedicated SA cluster. Measurements confirmed that our lambdas have the same computation cost as in our Cascade experiment.

Although Azure solutions are capable of high throughput, it is quite difficult to disable batching in standard Azure components. Figure 17a shows end-to-end latency, grouped by the number of agents detected in a frame. The topmost bar corresponds to all frames. Medians are all above 2 seconds, with long tails. The minimum was 924.67ms. Figure 17b drills down, showing that the latency originates in the EH: We deployed a consumer and a publisher in an idle Azure virtual machine to stress a premium-tier EH with a single partition. Latencies from sending to receiving is tracked for 30 seconds, using the same message sizes as in Figure 1. We disabled batching and controlled the sending rate for the lowest latency. The average EH latency is tens of milliseconds, over two orders higher than RDMA-enabled CMS, and jitter is common, resulting in the long tails evident in Figure 17a.

6 Additional Related Work

Sections 1 and 2 discussed a number of widely used big-data platforms and the challenges of adapting them for use in event-driven edge settings. Although Kafka Direct, Apache Flink and Storm [14] aim at stream processing, these are not the only prior systems relevant to ours. For example, Spark [47] achieves impressive performance for iterative tasks such as training. It gains this speed through in-memory RDD caching and scheduling, but runs the actual jobs on nodes distinct from the HDFS storage service that hosts data. We think of Cascade as similar in style, but with a primary bias towards low-latency that leads us to group related objects and then to run jobs close to their inputs. Prior work on K/V stores includes RDMA-enabled systems such as FARM [20] and FASST [29] as well as commercial data warehousing products, such as Amazon's DynamoDB [18], Snowflake [5], Microsoft CosmosDB, Databricks Datalake, Cassandra, RocksDB or even the Ceph object-oriented file system, which runs over a key-value store called RADOS [44]. As with Spark, none of these solutions hosts developer-supplied lambdas or deploys GPU accelerators close to the storage system, forcing costly locking, copying and domain crossings. Moreover, few focus on rapid data consistency (Azure has long featured strong storage consistency, and AWS recently introduced a consistency feature, but neither achieves particularly low update latency).

7 Conclusion

We created Cascade to host a new generation of edge computing that depends on large stored objects and other "big data" collections, yet also must carry out computations under time pressure. A central puzzle is that AI developers are locked into cloud tools and platforms, hence any new option must

be as transparent as possible. Cascade's architecture achieves these goals by allowing user code to be hosted very close to our servers and scheduling job execution on nodes that host the required data. The design enables end-to-end zero copy RDMA data paths between the application and our storage model, or between application stages. Cascade also offers data consistency: a guarantee needed in many reactive edge settings, where inconsistency can result in visible errors or real-world harm. Performance is excellent: stage to stage delays can be as low as 33*us* and bandwidth as high as 4.5Gbps. The latency figure improves on today's standard platforms by multiple orders of magnitude, while the throughput number equals or improves upon what today's platforms achieve.

8 Acknowledgments

We are grateful to Professor Julio Giordano, Martin Matias Perez and Daniel Martin for their help in designing the intelligent dairy pipeline we described here. Chris de Sa and Bharath Hariharan made many valuable suggestions early in our design process, and Ranveer Chandra suggested some use cases that shaped the overall solution. Partial support for our work was provided by AFRL/RY under the SWEC program, Microsoft, Nvidia and Siemens.

References

[1] Apache storm. https://storm.apache.org/.

[2] Cownet open source initiative. https://github.com/OpenNNs/CowNet.

[3] Spark streaming programming guide. https://spark.apache.org/docs/latest/streaming-programming-guide.html.

[4] Streaming data solutions on AWS. https://go.aws/3AXIWDJ.

[5] Snowflake. https://snowflake.com, 2012.

[6] Flink architecture: Tasks and operator chains. https://bit.ly/3rTFplD, 2021.

[7] Pub/sub. https://cloud.google.com/pubsub, 2022.

[8] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. 2015.

[9] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured streaming: A declarative API for real-time applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data*, pages 601–613, 2018.

[10] Network-Based Computing Laboratory at the Ohio State University. RDMA-based Apache Kafka (RDMA-kafka). https://hibd.cse.ohio-state.edu/#kafka.

[11] Mikel Broström. Real-time multi-camera multi-object tracker using YOLOv5 and StrongSORT with OSNet. https://github.com/mikel-brostrom/Yolov5_StrongSORT_OSNet, 2022.

[12] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. *SIGMOD*, pages 729–738, 2008.

[13] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157, 2011.

[14] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[15] Miguel Castro, Peter Druschel, A-M Kermarrec, and Antony IT Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications*, 20(8):1489–1499, 2002.

[16] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce online. In *NSDI*, volume 10, page 20, 2010.

[17] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[18] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[19] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Rethinking stateful stream processing with RDMA. *SIGMOD (to appear)*, 2022.

[20] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, pages 401–414, 2014.

[21] Yunhao Du, Yang Song, Bo Yang, and Yanyun Zhao. Strongsort: Make deepsort great again. *arXiv preprint arXiv:2202.13514*, 2022.

[22] John C Eidson, Mike Fischer, and Joe White. IEEE-1588™ standard for a precision clock synchronization protocol for networked measurement and control systems. In *Proceedings of the 34th Annual Precise Time and Time Interval Systems and Applications Meeting*, pages 243–254, 2002.

[23] Christopher Hannon, Deepjyoti Deka, Dong Jin, Marc Vuffray, and Andrey Y Lokhov. Real-time anomaly detection and classification in streaming PMU data. In *2021 IEEE Madrid PowerTech*, pages 1–6. IEEE, 2021.

[24] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, 2007.

[25] MSV Janakiram. Demystifying Edge Computing – Device Edge vs. Cloud Edge. *Forbes*, Sept. 2017.

[26] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P Birman. Derecho: Fast state machine replication for cloud services. *ACM Transactions on Computer Systems (TOCS)*, 36(2):1–49, 2019.

[27] Sagar Jha, Lorenzo Rosa, and Ken Birman. Spindle: Techniques for optimizing atomic multicast on rdma. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, pages 1085–1097, 2022.

[28] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, 2016.

[29] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation OSDI '16*, pages 185–201, 2016.

[30] Vivek Kashyap. IP over InfiniBand (IPoIB) architecture. Technical report, 2006.

[31] Benjamin Kettner and Frank Geisler. Iot hub, event hub, and streaming data. In *Pro Serverless Data Handling with Microsoft Azure: Architecting ETL and Data-Driven Applications in the Cloud*, pages 153–168. Springer, 2022.

[32] Peter Knight, Lucas Partridge, and Honor Powrie. GE aviation use-case–Apache Spark for analytics. In *2021 IEEE International Conference on Prognostics and Health Management (ICPHM)*, pages 1–5. IEEE, 2021.

[33] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7, 2011.

[34] Xiaoyi Lu, Md Wasi Ur Rahman, Nusrat Islam, Dipti Shankar, and Dhabaleswar K. Panda. Accelerating Spark with RDMA for big data processing: Early experiences. pages 9–16. Institute of Electrical and Electronics Engineers Inc., 10 2014.

[35] Karttikeya Mangalam, Yang An, Harshayu Girase, and Jitendra Malik. From goals, waypoints & paths to long term human trajectory forecasting. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 15233–15242, October 2021.

[36] Dario Martinez. From batch processing to streaming processing in aviation. `https://bit.ly/3edcX9G`, 2021.

[37] Microsoft. Azure event hubs quotas and limits, 2022.

[38] Microsoft. Integrate azure stream analytics with azure machine learning, 2022.

[39] Alexandre Robicquet, Amir Sadeghian, Alexandre Alahi, and Silvio Savarese. Learning social etiquette: Human trajectory understanding in crowded scenes. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision – ECCV 2016*, pages 549–565, Cham, 2016. Springer International Publishing.

[40] Anna Kornfeld Simpson, Adriana Szekeres, Jacob Nelson, and Irene Zhang. Securing RDMA for high-performance datacenter storage systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[41] Konstantin Taranov, Steve Byan, Virendra Marathe, and Torsten Hoefler. KafkaDirect: Zero-copy data access for Apache Kafka over RDMA networks. pages 2191–2204. Association for Computing Machinery (ACM), 6 2022.

[42] Shin-Yeh Tsai, Mathias Payer, and Yiying Zhang. Pythia: Remote oracles for the masses. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 693–710, 2019.

[43] Joris van Rooij, Vincenzo Gulisano, and Marina Papatriantafilou. Locovolt: Distributed detection of broken meters in smart grids through stream processing. In *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*, pages 171–182, 2018.

[44] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[45] Tom White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

[46] Jilong Xue, Youshan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. Fast distributed deep learning over RDMA. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–14, 2019.

[47] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache Spark: A unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[48] Ziyu Zhang, Zitan Liu, Qingcai Jiang, Junshi Chen, and Hong An. RDMA-based Apache Storm for high-performance stream data processing. *International Journal of Parallel Programming*, 49(5):671–684, 2021.

[49] Kaiyang Zhou, Yongxin Yang, Andrea Cavallaro, and Tao Xiang. Learning generalisable omni-scale representations for person re-identification. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(9):5056–5069, 2022.

[50] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. Designing distributed tree-based index structures for fast RDMA-capable networks. In *Proceedings of the 2019 International Conference on Management of Data*, pages 741–758, 2019.