Cascade: Fast AI/ML for Delay-Sensitive IoT Pipelines

Paper #60

Abstract

Efforts to deploy AI/ML in IoT settings encounter technology gaps. IoT applications (often implemented as distributed programs on a cluster or federated into pipelines) may need to track rapidly-changing data, yet existing platforms are designed mostly for offline batched use and exhibit problematic behavior under time-pressure. Cascade enables use of AI and ML in high-bandwidth, delay-sensitive environments. Our focus is on the critical path from an IoT sensor through a pipeline of developer-provided machine intelligence, and on the puzzle of minimizing latency while providing strong guarantees. Innovations include a new stored-procedure feature that hosts application lambdas directly in the Cascade address space, object and task collocation controlled using *affinity group keys*, leveraging of RDMA and GPU accelerators, and a separation of updates from queries that minimizes locking and copying. Microbenchmarks confirm that Cascade achieves exceptionally low latency and high throughput.

1 Introduction

As artificial intelligence moves into the edge cloud delay-sensitive and throughput-sensitive data paths are revealed [12]. Consider a smart dairy farm equipped with cameras, sensors and controllable elements (Fig. 1). Goals include identifying each cow, deciding if she needs to be washed, assessing body condition, and so forth. Each will often involve a series of intelligent sub-tasks. For example, when a motion detector senses activity, we ask the camera to capture a thumbnail, but not every thumbnail will show a cow. If the thumbnail is not properly centered, we may need to optimize camera settings before requesting a high-resolution image. For acceptable images, we run analytics, but only save "interesting" images.

Even this very basic application would be problematic on today's platforms. For example, in the Apache platform, the pipeline is defined using a package called Beam, then uploaded for analysis to an event-stream tool (or a "runner"), such as Flink, Spark, or Google Dataflow. The upload introduces delay and storage costs even before application logic sees the data. Like most back-end systems, Flink batches events to improve resource utilization, but this further amplifies delays. Reconfigured to compute instantly on each new event, the underlying HDFS storage layer may serve reads from stale or causally-inconsistent data [32].

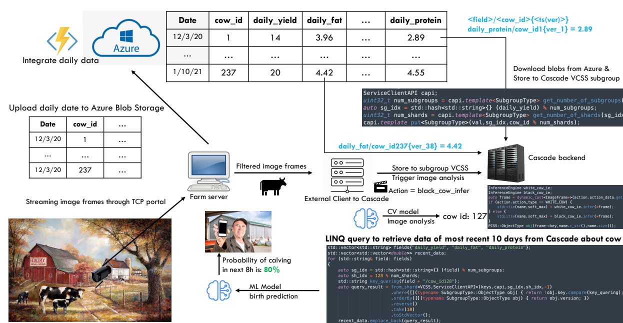


Figure 1: Part of a Cascade Smart-Dairy Application. In the black boxes we see *lambdas* introduced by the developer.

Dairy automation pushes the limits today, but cloud-hosted intelligence for controlling drones, intelligent roadway intersections, power grids and smart homes will require even faster reactions. Moreover many scenarios pose safety obligations that induce correctness obligations in the hosting platform.

Cascade minimizes delays using a ground-up architecture that centers on collocating computation and data while still preserving data freshness and consistency. The design minimizes locking, copying, and distributed data movement, taking advantage of accelerators when feasible. Whereas many platforms offer ways to build a single ML application by composing subtasks, edge frameworks have given less attention to independently-developed components that run separately, where the output of one becomes the input to others: a common situation when MLs are federated. Cascade's fast-path enables rapid sharing between AI or ML components that interoperate within a data-flow graph.

Prior work has focused on individual aspects of this challenge. Most databases support event-oriented computing; a modern example would be Snowflake [8, 16]. At a more mechanistic level, there has been work on wrapping RDMA into easy-to-use RPC APIs so that existing applications can leverage fast data-movement paths [13, 23, 27, 35, 37]. Libraries offer a spectrum of high-speed functionalities including shared-memory [19], key-value (K/V) stores [19, 27, 29], and communication primitives [5, 11, 24]. However, we believe that Cascade is the first-integrated solution built from the ground up specifically for to enable time-pressured intelligent IoT edge computing.

In what follows, we present the detailed architecture, show how Cascade can be used to solve the dairy application from Fig. 1, and then describe a series of microbenchmarked evaluations. These compare our overheads with those of standard tools (in which the AI/ML logic runs in a separate address space), then drill down on the fast path, where lambdas run inside the Cascade address space. Compared to Apache Flink (tuned as recommended in [34]), Cascade reduces latencies by a factor of 16x and bandwidth rises by 4x, even though Cascade guarantees much stronger properties.

2 Background

The core of any intelligent IoT application involves designing and training an AI/ML model, which can then be used to classify new events. Training is often viewed as a *back-end* activity and occurs offline, using packages such as Spark [38], TensorFlow Serving [31], Ray [30], TorchServe [6] and Triton [7]. Yet this often forces applications to use precomputed models. Cascade envisions a mix of edge classifiers and dynamic learning, enabling new IoT applications [12].

IoT data changes rapidly and classifiers can quickly become stale, so as edge platforms become smarter it becomes increasingly important to guarantee freshness of both the data and models used to make decisions: a dimension of data consistency. Edge applications often are graphical structures: simple pipelines or trees, with increasingly complex patterns likely to emerge as edge learning becomes more common. Node-to-node bandwidth may be very high. Today's backend systems gain resource-utilization efficiencies from batching; with event-triggered computing other forms of concurrency will be needed to keep the hardware productively busy.

Yet even as we consider these changes, it is important to keep in mind that developers have no appetite for reimplementing effective solutions. Any successful platform must be able to host the packages popular on back-end cloud platforms such as Tensor Flow [31], AWS SageMaker [26], Azure ML Studio [4], and Google AI [1]. In our work, if some existing package is adequately fast, developers can use it without change. But if a specific component is performance-critical, we offer a simple way to port that logic into what we call the fast-path. The resulting logic will leverage accelerators

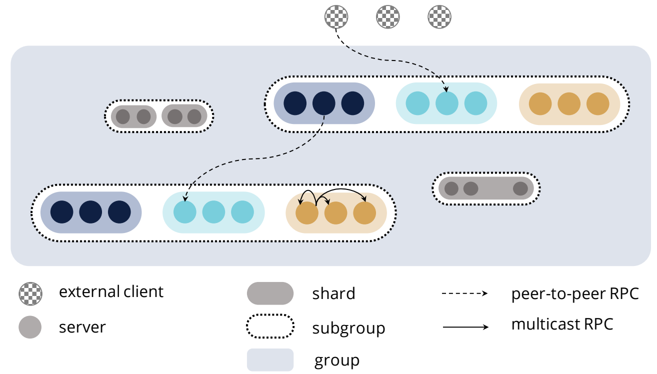


Figure 2: Cascade inherits the Derecho service architecture

to reduce overheads and delays, and yet still can interoperate with the larger "ecosystem" of unchanged legacy components.

To strike this balance, we start by offering Cascade as a standard K/V store, with a secondary option of treating it like a file system. Cascade will still offer benefits, because the legacy code can leverage our graphical computing model, data transfers will run over RDMA (if available), and our approach to code/data collocation can be exploited by configuring the legacy platform to run subtasks on the same nodes where Cascade is hosting any needed inputs. The fast-path, in contrast, goes much further by running AI/ML code *inside the platform address space*, eliminating performance-limiting barriers associated with crossing domain boundaries.

3 Cascade Design and Implementation

Cascade's central innovations are its lock-free RDMA fast-path, mechanisms for collocating objects with computation and platform customization using lambdas that run in the Cascade address space to minimize delay. In this section we start with the big picture, then drill down on the fast-path.

Like any sharded K/V store, Cascade allows key,value objects to be saved (`put`) or fetched (`get`) by any node. If a `put` arises internal to Cascade and targets the local shard, an atomic multicast or durable Paxos operation will be issued directly (the choice depends on whether the shard is configured to track just the most recent version using in-memory storage, or to keep a persistent version history). If the operation

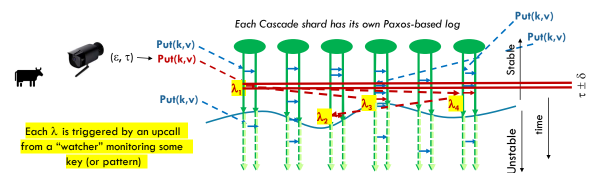


Figure 3: Serializable snapshot isolation in Cascade. The four lambdas will observe consistent data.

targets some other shard or originates in a platform external to Cascade, a P2P relaying operation occurs, selecting one of the shard members as a relaying proxy. Cascade keys are POSIX file pathnames, with support for directory hierarchies, a current working directory for each lambda, a home directory accessed with ~/, etc. This enables K/V data to be used through a file-system API at a slight overhead (more system calls are required for each access).

Cascade objects are arbitrary application objects with associated meta-data, notably version numbers and timestamps. Callers of `put` can supply one or both, or even a *hybrid logical clock* [28]. Applications that leverage version-numbering gain a fault-tolerance benefit. Should a `put` be disrupted by a failure the can originator safely reissue the same request with the same version number. If the version already exists the duplicated request is ignored; if not, the proxy performs the operation. We obtain exactly-once semantics [24, 33], with a very strong form of consistency illustrated in Fig. 3, where computations triggered by an IoT event at time *t* query Cascade-hosted data. These queries run on stable data that is temporally accurate and causally consistent; stabilization takes about 50*us* (dashed vertical timelines).

Cascade inherits some of these guarantees from Derecho [24], an open source C++ RDMA data-replication library that we used when developing our solution. However, Cascade goes well beyond anything Derecho supports by offering a serializable snapshot isolation guarantee [14] on a K/V "database" that supports full SQL queries with the constraint that updates are atomic, but limited to a single shard at a time. Queries run on a separate data path, accessing committed, stable data as was seen in Fig. 3. This is why so little locking is required: once an update becomes stable the entire prefix of the log including that update can be treated as immutable read-only data. When data must be moved RDMA is used if available; if not, all features map efficiently to TCP.

3.1 Fast-Path Programming model

Suppose that we simply want to run existing AI/ML code to implement the dairy image processing pipeline example from the introduction. Today the standard way to do this would be to use Flink, Spark or Tensor Flow, and the most standard option would be to employ a K/V connector that treats Cascade as a storage service. At binding time, the connector would learn the sharding pattern and IP addresses of the server nodes. Then, as IoT events occur, it could use this information to forward the data to the proper shard. Communication will be very fast if RDMA is available but because such services batch events, and have significant overheads other than data transfer, the performance benefit will be limited. Moreover, although collocation of computation and data is a theme for our work, it isn't trivial to configure legacy schedulers to run tasks on the same nodes where Cascade hosts the data those tasks will use.

The central idea of the fast-path programming model is to support lambda functions that are hosted directly within the Cascade server nodes, wiring themselves in by requesting upcalls when "events of interest" occur. Then we arrange to trigger the lambda at a node where dependent objects have been prepositioned. The trigger mechanism itself is similar to monitoring a folder or file in Linux, but by arranging for the trigger to run on a node that already hosts the data it needs delays will be minimized.

3.1.1 In-Place Message Construction

Even before we tackle placement, a first challenge involves delay when the end-to-end application constructs IoT objects or other updates: If we don't address those costs, the game would be over before it starts. Traditional data marshalling, copying and even brief locking can dramatically reduce performance [25]. With this in mind, we implemented an in-place C++ object construction option that lets the application allocate memory within the Cascade message buffer region, preregistered for RDMA transfers. To enable zero-copy sharing, server nodes use a single policy for data representation and alignment, while external clients respect the server policies (they may incur a cost, such as storing integers or floats in a non-native format, but this cost will be spread over the IoT components and hence insignificant). Servers share objects with one-another in the agreed-upon format, avoiding marshalling or copying. Cascade publishes the service's desired byte layout and alignment when an external client connects, enabling connectors to preconstruct efficient marshallers.

The Cascade in-place construction APIs are used by a C++ class as follows. First, a static factory method requests memory from a special allocator, supplying the desired constructor as a lambda. As soon as message buffer space becomes available, Cascade upcalls the lambda, which can now run asynchronously, constructing the object and eventually marking it as ready to send (long delays are common at this step, for example to wait until motion is detected or a new photo is acquired). Cascade polls "under construction" message slots and will transmit a message (or group of messages) when ready, respecting the FIFO order in which the message memory slots were originally requested. Python and Java constructors use this same approach, but access memory via getters and setters.

3.1.2 Collocation of Related Data Objects

Our next step focuses on the locations at which data is stored and where computation will occur. Most K/V systems hash the object keys to obtain shard numbers. Hashing randomizes object placement onto the available shards, and is convenient for parallel computing patterns such as MapReduce, but IoT edge pipelines will often be linear sequences of actions or simple tree-like structures, with less frequent use of MapReduce. The concern is that if a lambda depends on multiple

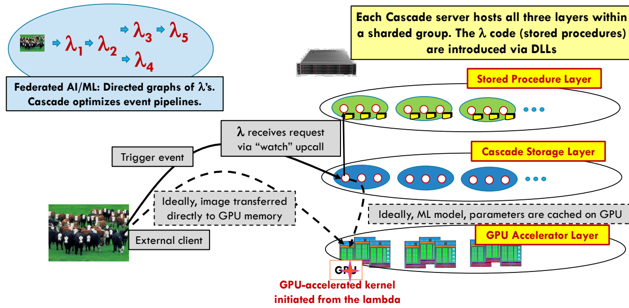


Figure 4: The Cascade Fast Path

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 (a typical RDMA data rate) takes one second. Moreover, whereas a traditional batched AI/ML platform will often reuse dependent objects frequently, giving the caching layer a chance to compensate for high initial data-access costs, Cascade optimizes latency and throughput on an event-by-event basis. Even a single unnecessary object transfer could be a dominating overhead.

With this in mind, the Cascade API allows developers to associate an *affinity set* key with each object: a second pathname. Objects in the same affinity set will be collocated, moved as a set, cached as a set, or evicted from cache as a set. If an object has a non-null affinity key, the affinity key (not the object key) is used to hash and select the hosting shard.

Such objects still have their own unique names, in whatever folder hierarchy the developer employed, but this placement policy physically hosts them on the same Cascade storage nodes. Of course, we still need a way to access those objects by name when some component calls `get`. For this purpose, Cascade maintains a *metadata store* within which affinity set mappings are tracked. In addition to supporting `get` this enables the system to list the 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.

Consider our intelligent dairy scenario. How might affinity sets arise, and be used? Each lambda would have an affinity set for inputs: hyperparameters for its AI/ML model, the trained model, and so forth. Results from prior computational steps could also be saved using affinity set keys. Those objects will all reside in some single shard, hence by simply using the same affinity key when deciding where to initiate the computation, we obtain the desired collocation.

For example, suppose that some host computer external to Cascade has an IoT device that senses movement in the barn, and the host then captures a thumbnail photo. The host computer will be an external client of Cascade, running code that was configured with the affinity key. When this host first binds to Cascade, it downloads the shard mapping, so that later as events occur, it already will have determined the responsible subgroup and shard. Cascade membership can change due to failures, forcing a refresh, but this is infrequent. Thus, when the camera captures the image, the host software simply sends the event to a node within the preselected shard. We offer several automated load-balancing options.

3.1.3 Lambdas

Our next challenge is to efficiently host user-supplied lambdas, and to minimize overheads when they trigger. Many of today's most widely used platforms require that data always be stored before computing can be initiated, but in Cascade we wished to avoid these potentially unneeded costs (IoT inputs are often discarded after analysis). This led us to distinguish three cases. In the first case, the developer intends to trigger some computation but without saving any data. For this purpose, `trigger` is used; it has the usual key and value arguments, but is supported as a P2P operation. The second involves updating a "volatile" (in-memory) object by replacing the current version (if any) with a new version: a replicated update within the shard where the object resides. Here, the triggered code will run after these updates occur either on one node, or on all shard members, depending on the developer's preference. The third case first updates a versioned object logged to persistent storage, then runs any triggered lambdas. For this third case we also support storing the delta of this new version relative to the prior one, compressing it, or even entangling its cryptographic signature with (entangled) signature on the prior version to create a tamper-proof chain.

We obtain the structure illustrated in Fig. 4. This figure shows nine nodes running a sharded service with three members per shard. Within each node one finds three "layers" of logic sharing a single address space: `get` operations return pointers directly to immutable data in the Cascade data region, avoiding locks or copying if the data is available locally. One is the developer-supplied lambda that orchestrates the computation. The second layer is the object storage infrastucture, which will be queried by the lambda when it needs to access input data such as the key and value that triggered the lambda upcall, objects hosted in Cascade, and cached content which might also include data pulled from elsewhere in the cloud. The third layer depicts the GPU accelerator, hosted on the same node and used for expensive computational tasks. GPU memory is available for incoming RDMA transfers, and a portion is set aside as a cache managed by Cascade.

The application depicted in this figure is one stage in an intelligent photo analysis. The initial capture of the photo triggered a lambda to assess photo quality (perhaps, from a thumbnail). That lambda then requested upload of the full image, perhaps after adjusting camera parameters. Further stages then trigger. Thus, even a rather basic task might involve a federated ML with pipelines of lambdas.

In the general ML literature there is a great deal of attention to MapReduce. At present, we don't expect that MapReduce

will be as common as linear sequences, but we do support the pattern: in this case, one event triggers an upcall to a lambda *in each member of a shard*, which could have many members. The lambdas run the map step, then shuffle, sort/group and reduce. The distributed AI that results could be "one task" in a larger federated ML computation.

Lambdas are created much as one creates any AI/ML application, except that the code is compiled to a DLL and then loaded by all the Cascade servers (at startup or dynamically). As such, the feature can be used with existing packages provided that they are capable of running within the address space of an active C++ program. What would normally be the `main` method in a standard AI program will be replaced by an `initializer` for the DLL. To set watches for events of interest, the DLL `initializer` can register methods (perhaps using the C++ "capture" feature to pass additional context information to the lambda for use when it receives an upcall). Note that the DLL will be loaded by all Cascade nodes and may have the identical watches registered everywhere, but upcalls only occur on nodes where a matching key is updated. Moreover, even if all nodes in a shard match a key, Cascade has selectable policies under which the developer can request that just one upcall be triggered, in a load-balanced manner.

Lambda upcalls raise non-trivial issues of thread delays and memory management. Recall that Cascade is implemented using Derecho. Incoming Cascade events and photos are all Derecho messages, hence Cascade itself learns of updates via upcalls on the Derecho predicate thread - a critical path for all of Derecho. Accordingly, Cascade limits itself to doing a fast path match to see if the incoming key-value object matches any pending lambda (a trie lookup). If a match is found, the key and a pointer to the value are enqueued in a FIFO round-robin buffer. A pool of application-dispatch threads monitors this queue, removing elements and doing upcalls to the registered lambdas, preserving FIFO order by ensuring that any single DLL will see all its upcalls on a single thread. Minimal locking is required (to implement the round-robin buffer), and only the C++ shared pointer is copied.

3.2 Coding (and Caching) for Intelligent IoT

Developers of Cascade stored procedures can work in C++ (perhaps leveraging a library like MxNET), Python or Java, but the underlying native API is in C++ 17. Lambdas will often query databases or other services external to Cascade. To support such tasks, our native C++ layer needed a language-embedded SQL option. Curiously, C++ 17 lacks a standard solution to this problem, but we identified a package, "boolinq.hpp", that offers a comprehensive open-source version of the LINQ language-embedded query syntax for SQL-style operations on collections. Lambda developers could also use packages like Tensor Flow, PyTorch and NumPy.

Although LINQ and similar language embeddings for SQL are not new, our use of them in time-pressured IoT settings exposes interesting new choices and tradeoffs, particularly with respect to design of affinity groups, cache management and prefetching. Prior work on back-end, batched systems such as MapReduce, Hadoop and Spark/Databricks (a family of related systems), Tensor Flow and more recently, Ray, showed that caching is central to performance in long-running computations, such as when a large model is trained [10, 17, 30, 38]. Offline training tasks are heavily iterative and may run for hours, so object reuse is to be expected. In contrast, edge logic will be triggered event-by-event under time pressure, and data will often evolve so quickly that even if the code appears to reuse the identical item, it may still need to be recomputed.

Indeed, it makes sense to anticipate two cases. Long-term stable data, such as a machine-learned model for cow recognition, will probably continue to be computed on existing back-end systems, downloaded, and then frequently reused. This sort of object should ideally be downloaded once, then saved in the same affinity group as the object that triggered the lambda. As long as events from a given source follow the same path within the system for each pipeline, they will find needed data inputs either stored locally or cached locally, including data acquired from external sources such as databases or file systems. For other objects generated and hosted entirely within Cascade, but that might be accessed from multiple shards or where there is no natural way to assign an affinity group key, we implement a coherent caching scheme, ensuring that if an object is updated, any stale cached versions will be evicted from cache.

An final nuance involves caching on GPU. Cascade manages accelerator memory, and hence can leave an object in GPU memory for reuse even if the higher-level code doesn't explicitly distinguish where data touching will occur. The option of using GPU Direct to terminate RDMA transfers directly in GPU memory, avoiding a staged transfer through the host and an extra DMA operation from host into GPU, is also of interest. Both topics will be explored in future work.

Operation	Description
`put`	Insert or update an object.
`trigger`	Notify *watchers* tracking the key on a node.
`remove`	Remove an object by key.
`get`	Get the value of an object by key.
`initialize`	Initialize DLL resources.
`destroy`	Destroy DLL resources (on exit).
`register`	Register itself to Cascade context.
`unregister`	Unregister itself from Cascade context.

Table 1: API for K/V operations and registering lambdas.

3.3 Additional functionality

Our focus has been on the Cascade fast-path. To preserve space we were forced to omit discussion of a number of Cascade features associated with support for multitenancy and to avoid naming collisions when distinct lambdas happen to use similar keys. Also important but not covered here are support for federated ML applications described as data flow graphs, support for the MapReduce computing model and other popular D-AI and D-ML computational patterns, wide-area mirroring, security, thread scheduling, NUMA architectural considerations, and system configuration features.

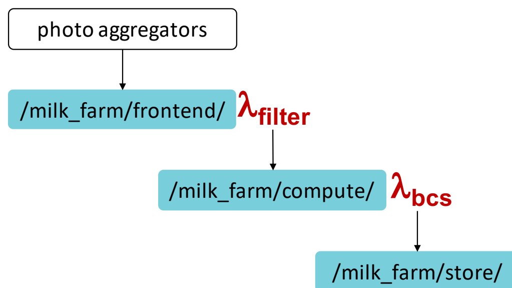


Figure 5: The fast-path in a Smart Dairy Service.

4 Smart Milk Farm with Cascade

Rather than limiting our evaluation to microbenchmarks, we also implemented a dairy intelligence application. The application focuses on one step in an AI framework for dairy productivity and management, and models a research dairy farm like the one illustrated in Fig 1, where cows are milked a few times daily using partially automated systems. To track of the health of each cow, the system images animals as they enter or leave the milking parlor, then employs ML to develop a variety of informatio streams which will be reported via dashboards aimed at 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.

The filter we use scores each image based on whether or not it shows a cow, sharpness, and other criteria. For example, the research farm we are working with requires that we not image farm workers. Storing photos prior to analysis would violate the policy, but a purely online filtering procedure is considered acceptable. The bcs analysis, which runs only on selected images, uses an AI model to assess body condition. This currently is a visual metric based on analysis of the cow haunch, visible fat, muscle and skeletal structure.

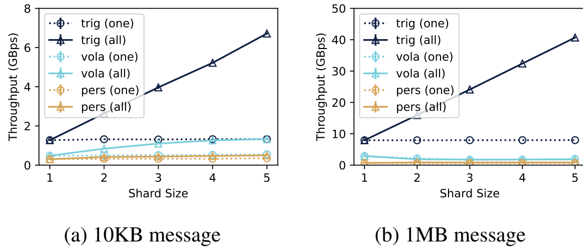
Both computer vision analytics are fairly basic, and unoptimized. The filter lambda is implemented as a convolutional neural network. Its output is a real number between one to zero: if the output is lower than a threshold, the image is dropped. In the production system, we will need to evolve it to run on a set of images of each animal, rejecting images that violate our goals and then selecting the best image among those that remain. The bcs lambda is based on CowNet [2], an open-source research framework, but in work still underway, is being extended to correlate the bcs score with other metric data associated with the cow (identified by RFID), such as milk quality and quantity, using LINQ to query electronic health records. Our full framework uploads these records and other metrics to Azure's CosmosDB, where we deploy AI logic to detect and clean up noisy records.

We found it easy to modify the existing filter and bcs logic to use the stored procedure interface defined in Table 1. We started by compiling the two lambdas against a C++ TensorFlow interface called CppFlow [3]. CppFlow turns out to be somewhat slow, but has the benefit of full compatibility with the Cascade DLL model. The resulting DLLs then needed to be configured with an object containing hyperparameters and a second object holding the trained mode; we stored these as K/V objects and the initializer in the DLL loads them using `get`, then passes the associated objects by reference when registering the lambdas. In the full system, the bcs lambda will also query CosmosDB, but for this experiment, we host all the needed objects in Cascade. Ultimately it will be important to leverage caching to avoid unnecessary critical-path delays, but that should not be difficult: milk metrics and electronic health records are uploaded at most once per day.

These lambdas can be instantiated multiple times per node. On our servers a concurrent load of four instances gives the best performance. Cascade allows us to impose such a limit by configuring the number of threads used for triggered upcalls in each server node, and Flink has a similar option.

Fig. 5 illustrates the three-staged logic that results. Associated with each stage is a folder within the K/V store: `/milk_farm/frontend/`, `/milk_farm/compute/`, and `/milk_farm/store/`. *Photo aggregator* nodes capture photos and metadata and forward each record using P2P trigger puts, selecting a target node associated with the frontend folder, where an upcall to the filter lambda will occur.

We deployed the solution in two forms: as a Cascade fast-path, and as an Apache Flink task. Each platform also requires a description of the two-step pipeline. For Cascade, we created a JSON file that describes the fast path topology in Fig. 5 using a data-flow graph notation. For Apache, we used Beam to describe the desired Flink stream-processing pipeline. The two deployments were then run on the identical hardware platform, configured with adequate resources for the highest possible performance. In both cases we arranged for node assignments that place the filter and bcs lambdas on different nodes, equipped with GPUS. Output is stored to a storage service, and we configured it to run on nodes with large high-speed storage options. Finally, we implemented an external client program. It runs as a stand-alone C++ application that



(a) 10KB message

(b) 1MB message

Figure 6: K/V Store Put Throughput

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

Table 2: Typical Put Operation Latencies

hosts the IoT camera, using our own RDMA-based RPC to access the Cascade API from Table 1. For performance evaluation, the client program was configured to replay a script of previously-captured images. Results are presented in Section 5.3, where we show that Cascade's low overheads translate to higher application performance. With faster filter and bcs logic, the gap would grow even wider.

5 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 the dairy image pipeline.

All experiments ran on a cluster of dedicated servers. Each server has a Mellanox ConnectX-4 VPI NIC card connecting to a Mellanox SB7700 InfiniBand switch, which provides an RDMA-capable 100Gbps network backbone, but the machines themselves have two configurations. The more powerful configuration has two Intel Xeon Gold 6242 processors, 192 gigabytes of memory, and an NVIDIA Tesla T4 GPU. The second setup 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 an Intel Optane P4800X NVMe card.

All servers run Linux kernel 5.0.0. We synchronize the server clocks with PTP [20] so that the skew among them is in the microseconds, allowing comparison of the timestamps from different servers with sub-millisecond precision. Appendix A provides additional clock-synchronization details.

5.1 Cascade K/V Store Performance

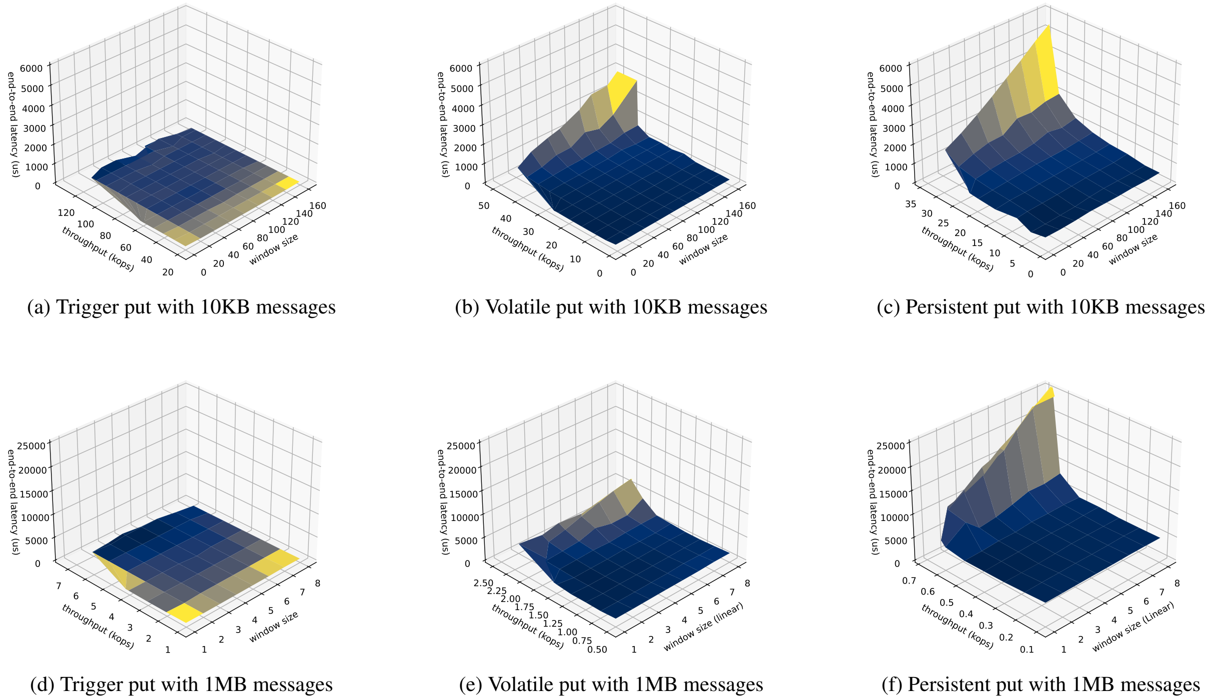
We started by evaluating the performance of Cascade's three types of put operations. A volatile put operation (`vola`) puts a K/V pair to a volatile Cascade subgroup, where the data replicated in a consistent, fault-tolerant manner. A persistent put operation (`pers`) is similar to a volatile put except for persisting data in a file system in the NVMe device. A trigger put operation (`trig`) is a point-to-point: it sends a K/V pair to a single node, and the data is not retained after the triggered action runs. In this microbenchmark, a set of clients running in type B servers issue requests to Cascade nodes running on type A servers, at a controlled rate. The nodes are all dedicated ones. 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`).

5.1.1 Throughput

Figure 6 shows the put throughput of Cascade K/V store with varying shard size, or 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, confirming the excellent scalability of the Derecho library. 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) and $\sim$ 800 MBps (800 ops), both for small and large messages. The bottleneck turns out to be a side-effect of the Paxos-style consistency model we support. Although our NVMe device can achieve sequential write bandwidth of 2.4 GBps, this requires an uninterrupted stream of 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. These delays prevent the system from leveraging the full DMA bandwidth. Trigger put operations scale best because these operations



(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 7: K/V Store Put Latency

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.

5.1.2 Latency

Table 2 shows typical latencies for put operations. 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 the time from the client first sent the request to when the last replica finishing persisting it; while for each trigger put operation, we measure until the request reaches a replica that upcalls to a developer-supplied lambda. Recall that although this entails comparing timestamps from distinct machines, their clocks are synchronized to a resolution of a few microseconds.

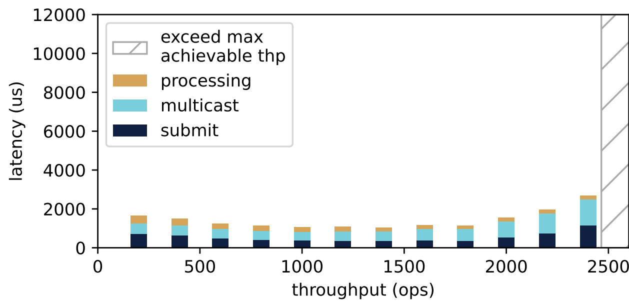
The persistent put latency is about four to five times higher than that of a volatile put. Below, we confirm that the bottleneck is the I/O to our storage devices. Trigger put is one order of magnitude faster than volatile and persistent put because it does not need to replicate or persist any data.

The data in Table 2 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 Figure 7. 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.

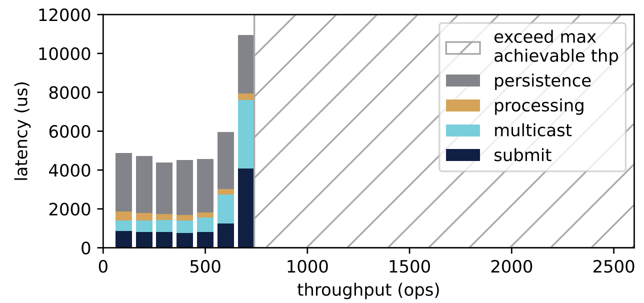
As shown in figures 7b, 7c, 7e, and 7f, before the system becomes saturated end-to-end latency is consistently low, corresponding to the flat part on the near right part of the curved surfaces. In this area, the messages haven't formed queuing backlogs in the transport windows, and are processed immediately. 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.

Figures 7a and 7d 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.

Figure 8 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

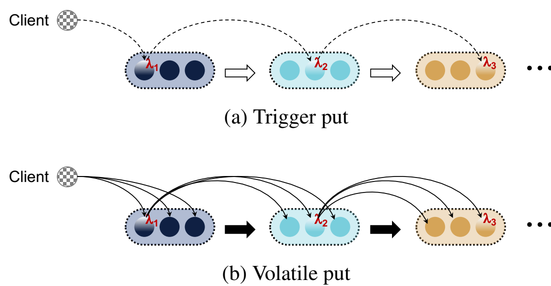


(a) Volatile put



(b) Persistent put

Figure 8: Latency Breakdown for Volatile and Persistent Put



(a) Trigger put

(b) Volatile put

Figure 9: Pipeline Illustration

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 from the replica state being updated to the corresponding update being persisted in all replica. Because trigger put has only the submitting component, we exclude it in this figure.

For volatile put, the multicast and submitting components account for most of the end-to-end latency. As the request rate increases, the overhead of those two components grows as well: messages pile up in the window slots. For persistent put, the persistence overhead dominates. However, as the request rate goes up close to the maximal achievable throughput, the overhead of submitting and multicast grows suddenly because the messages pile up. This is the issue noted previously: the underlying Derecho persistent multicast (a version of Paxos) sometimes syncs data to storage, delaying the pipeline.

5.2 Pipeline Performance

To evaluate 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 (Figure 9). As seen in the figure, the shards have three members each, consisting of 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 serve. The client program runs on a type B server node.

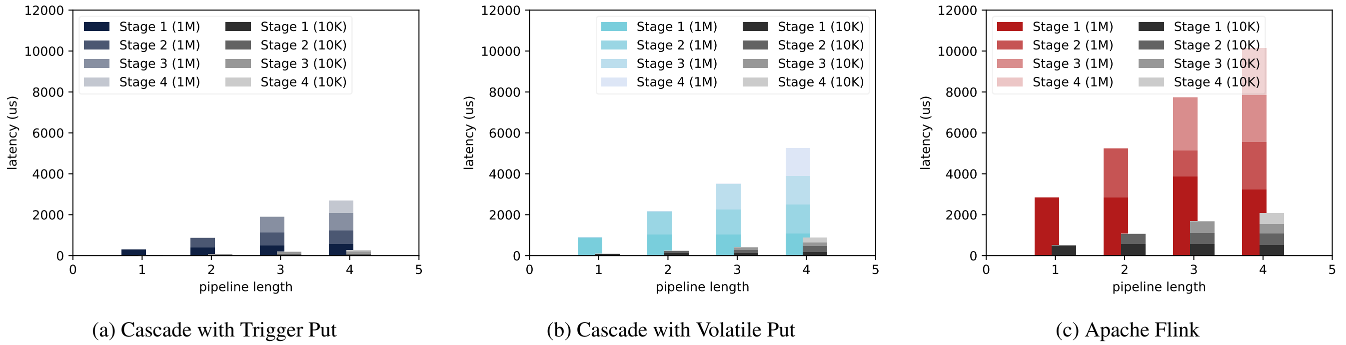
5.2.1 Pipeline Latency

For our first experiment, the client sends either 10 KB or 1 MB messages at steady but low request rate. We then measure the time from when the client sends each request to the time when the no-op logic is activated, stage by stage. Figure 10a and 10b show the average latency for all the messages during a representaive 5-second period, for varying pipeline lengths. The latencies for the first stage match the trigger and volatile put latencies shown in table 2. With longer pipelines, the latency for each stage is a linear function of the depth of that stage in the pipeline, showing that the overhead of the lambda upcall infrastructure is negligible. Of interest here is the comparison of trigger put, where the data is sent to just one shard member and not retained, with volatile put, where data is replicated using atomic multicast and is retained in memory. We see here that trigger is faster, but that the overhead of volatile put is surprisingly low.

To contrast these numbers with the state-of-the-art stream processing system, we then configured Apache Flink computations in an layout intended to mimic our trigger put scenario[1]. This required several steps[2]. First, we constrained Flink to use a single task-processing slot per server, ensuring that it would not try to run multiple tasks on any single node. Next, we disabled Flink's automatic operator chaining [9] to prevent it from combining all the no-op tasks into a single task (operator chaining is a form of inlined composition: useful, and equally

[1]Flink itself lacks data replication, but we could have mimicked a volatile put by storing objects into HDFS, configured for automatic data replication. We considered running such an experiment, but the performance would have been so poor that the comparison wouldn't have been useful.

[2]Tuning Flink proved to be challenging. In summary, we quickly discovered that although Flink is an event processing framework, it really isn't optimized for low latency. To enable a fair comparison we spent weeks adjusting parameters and testing, while simultaneously combating the Kinsing malware exploit, which attacks Flink through its web GUI API.



(a) Cascade with Trigger Put

(b) Cascade with Volatile Put

(c) Apache Flink

Figure 10: No-op Pipeline Latency

relevant to Cascade, but orthogonal to the topic of this paper). Two changes go beyond what most users might consider. We recompiled Flink to load 1MB at a time, rather than its standard 32KB block size, and also set its minibatch delay barrier to zero. This last change might be controversial: the Flink documentation emphatically specifies that the barrier delay not be reduced below 1ms [34], but when we used this guidance, Flink performance was terrible and the servers had very low CPU utilization levels.

As shown in Figure 10c, even with all of this tuning, Flink's pipeline latency is quite high. 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.

We see two reasons for Cascade's higher performance. First, the Cascade RDMA fast-path offloads communication to the RDMA hardware, whereas Flink uses TCP, incurring the standard overheads of the kernel network stack. We find it interesting that the RDMA advantage (at least with Derecho's multicast protocol) dominates even non-replicated P2P relaying. Cascade's fast path also benefits from in-place serialization. Flink uses the Java-based Kryo serializer, which requires copying from the Java-managed memory region into a network buffer. One implication is that even if Flink were ported to use RDMA (a non-trivial undertaking because the RDMA communication model is so different), marshalling would be a significant bottleneck.

5.2.2 Pipeline Throughput

We then stress the same pipelines by tuning our client to run at the maximum sustainable message rate for each setup, obtaining the first four throughput series shown in Figure 11. Again, the throughput of Cascade's one-stage pipeline matches the trigger and volatile put throughput in Figure 6, dropping slightly as we move to a pipeline with two or more stages. This reflects the extra costs associated with message relaying:

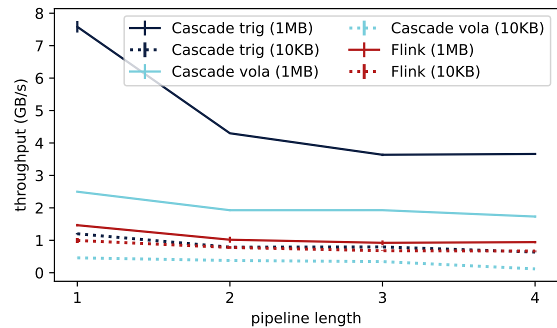


Figure 11: No-op Pipeline Bandwidth

the first pipeline stages each need to send as well as receiving. 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, although the error bars are still small.

5.3 Dairy Application Performance

We deployed the smart dairy application from Section 4. Recall that nodes in the frontend and compute stages perform image analysis: they benefit from GPUs and use servers of type A. The other nodes run in the type B servers. Like the previous experiments, 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. We configure the storage folder to run in Cascade's volatile mode, meaning that the most recent version of each object will be retained 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.

5.3.1 Latency Breakdown

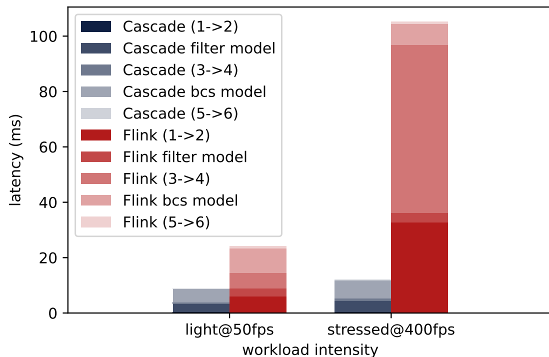


Figure 12: Smart Milk Farm Latency Breakdown

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 by the photo;
3. the filter model finishes processing the photo;
4. the bcs model is triggered by the photo;
5. the bcs model finishes processing the photo;
6. the result is written by the store folder.

By calculating the deltas between timestamps we obtain a latency breakdown along the pipeline.

The blue bars in Figure 12 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 nine milliseconds for the light workload at $\sim$ 50 fps and twelve milliseconds for the stressed workload at $\sim$ 400 fps. The time spent in model inference dominates end-to-end latencies: filter processing time represents about 36% of end-to-end latency; and bcs model processing is even greater, at nearly 53%. The aggregated data forwarding latency represents just 11%, reflecting the efficient fast-path.

The Flink latency breakdown is shown as the red bars in Figure 12. Although the filter and bcs models consume an identical amount of 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 12 *ms*, is about one-ninth that of Flink's. Even with light load at 50 fps, Flink's end-to-end latency is 25 *ms*, whereas Cascade is just 9 *ms*, a 64% reduction.

5.3.2 Throughput Scalability

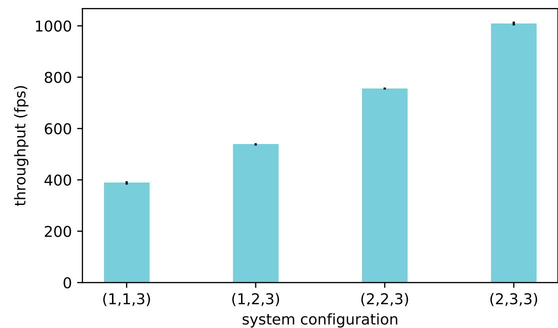


Figure 13: Dairy Pipeline Transactions/sec. The triples indicate the number of members comprising the front-end stage, compute stage, and storage stage, respectively.

We investigated scalability by varying the number of nodes while tracking throughput, but focused purely on Cascade. Here, the configuration of the shard responsible for each stage has a significant impact, so we use a three-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 bcs scoring lambda), and storage. For example, $(1,2,3)$ 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 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 Figure 13. For context we benchmarked both lambdas on a single server of type A: filter runs at $\sim$ 540 fps, while bcs runs at $\sim$ 400 fps.

The overall trend is easily understood. The bcs lambda is the bottleneck in the $(1,1,3)$ configuration. In the $(1,2,3)$ 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 $\sim$ 540. With configuration $(2,2,3)$ bcs is again the limit. With the $(2,3,3)$ configuration, the two lambdas are balanced and throughput exceeds one thousand fps. Broadly, these results support our view that Cascade has excellent scalability, and useful configuration flexibility.

6 Additional Related Work

The introduction and background sections discussed a number of widely used big-data platforms and the challenges of adapting them for use in event-driven edge settings. Although Apache Flink and Storm [15] aim at stream processing, these are not the only systems similar to ours. For example, Spark [38] achieves impressive performance for iterative tasks where in-memory RDD caching and smart scheduling can be leveraged. The core trade-off is the one we discussed in connection with Apache Flink: today's back-end systems are optimized for batch computing. Although they can be configured to use very small batch sizes (mini-batches), their efficiency degrades because we deprive their schedulers of the scheduling flexibility they require.

The tradeoff is interesting. By configuring Flink to disable its minibatching feature, we departed from the standard setup of that platform. We had no choice: in the normal configuration, Flink's per-event latencies would have been 50x higher. The natural question is whether Cascade obtains acceptable levels of resource utilization. In fact we do, via per-event scheduling, but for reasons of brevity we must leave that topic for a future paper.

Cascade was created from the ground up for event-driven settings where timely response is prioritized, and uses job collocation controlled by the developer (through the use of affinity keys) to keep resources busy. The goal of low per-event latency drives a series of design choices that wouldn't be ideal in a batched system where high throughput is the overarching goal. Additionally, the most common style of component-to-component interaction changes. Whereas all of the systems we described encourage a graphical style of application development, the subtasks (the nodes within these graphs) often are components of some single process. The style of ML federation on which our work focuses would often involve programs written in different languages and perhaps running on a variety of frameworks, communicating by sharing K/V objects. Optimization of the fast-path for this case posed new challenges.

Prior work on K/V stores includes RDMA-enabled systems such as FARM [19] and FASST [27] as well as commercial products, such as Amazon's DynamoDB [18], Snowflake [8], Microsoft Cosmos [21, 22], Databricks Datalake, Cassandra, RocksDB or even the Ceph object-oriented file system, which runs over a key-value store called RADOS [36]. The distinction here is that none of these solutions host the developer-supplied lambdas within the same address space as the storage system and the hosted GPU accelerator, forcing costly locking, copying and domain crossings. Our contribution centers on the Cascade fast-path, which hosts lambdas in the address space of Cascade itself, as well as our use of RDMA for lock-free and zero-copy hardware accelerated data movement. This permits multiple orders of magnitude improvements both in event-response latency and in overall throughput.

We do not have space for a systematic review of prior work on database platforms for AI and ML, but in fact view this topic as somewhat tangential. Whereas Cascade focuses on low-latency stream processing with AI/ML lambdas on the critical path, many AI and ML applications treat databases as an integral aspect of the artificial intelligence algorithm, which they express directly as database computations. This leads to a great deal of work on expressiveness of the query APIs offered to the AI/ML logic, database schemas optimized for the patterns of queries that arise in AI/ML settings, and optimization when such systems are running batched workloads over very large data sets. These are important questions and the work is of high quality. However, the focus is not on the style of per-event processing that our effort seeks to optimize. Conversely, Cascade's LINQ query package has limited functionality and would not be the best choice for ambitious big-data computations.

7 Conclusion

Cascade demonstrates that even with strong consistency guarantees, an edge IoT platform can host very low-latency, high bandwidth edge intelligence. With pipelines of developer-supplied lambda methods, stage-to-stage delays were as low as $33us$ in our experiments, and bandwidth as high as 4.5Gbps. In contrast, Apache Flink exhibited latencies of $560us$ or more per pipeline stage for the same workload, and bandwidth peaked at less than 2GBps. These benchmark performance figures carry over to full applications, as demonstrated using a smart-dairy application that has been deployed at a research dairy farm on our agriculture and life sciences campus.

The demand for AI/ML in edge IoT settings will surely continue to expand, with applications in many settings. The smart dairy experiment explored here already strains the limits of what standard platforms can do. As edge communication options expand to include 5G first-hop links and perhaps even edge RDMA (or similar options such as DPDK), intelligent applications will grow to encompass mobile vehicles and smart infrastructures, bringing steadily more demanding latency and throughput requirements.

Cascade is fully open and available under permissive 3-clause BSD licensing.

References

[1] AI platform. https://cloud.google.com/ai-platform.

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

[3] Cppflow. https://github.com/serizba/cppflow.

[4] Microsoft Azure Machine Learning Studio(classic). https://studio.azureml.net/.

[5] Nividia collective communications library (nccl). https://developer.nvidia.com/nccl.

[6] Torchserve. https://github.com/pytorch/serve.

[7] Triton inference server. https://github.com/triton-inference-server/server.

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

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

[10] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 265–283, USA, 2016. USENIX Association.

[11] Jonathan Behrens, Sagar Jha, Ken Birman, and Edward Tremel. Rdmc: A reliable rdma multicast for large objects. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 71–82. IEEE, 2018.

[12] Ken Birman, Bharath Hariharan, and Christopher De Sa. Cloud-hosted intelligence for real-time iot applications. *SIGOPS Oper. Syst. Rev.*, 53(1):7–13, July 2019.

[13] Rajarshi Biswas, Xiaoyi Lu, and Dhabaleswar K Panda. Accelerating tensorflow with adaptive rdma-based grpc. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, pages 2–11. IEEE, 2018.

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

[15] 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.

[16] Raphaël Champeimont. Making Queries 100x Faster With Snowflake. 2020.

[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] 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.

[20] 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.

[21] Seth Elliot. Microsoft Cosmos: Petabytes perfectly processed perfunctorily. https://blogs.msdn.microsoft.com/seliot/2010/11/05/microsoft-cosmos-petabytes-perfectly-processed-perfun Nov. 2015.

[22] Mary Jo Foley for All About Microsoft. The Bing back-end: More on Cosmos, Tiger and Scope. http://www.zdnet.com/article/the-bing-back-end-more-on-cosmos-tiger-and-scope/, Oct. 2011.

[23] Nusrat S Islam, Mohammad Wahidur Rahman, Jithin Jose, Raghunath Rajachandrasekar, Hao Wang, Hari Subramoni, Chet Murthy, and Dhabaleswar K Panda. High performance rdma-based design of hdfs over infiniband. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2012.

[24] 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.

[25] Sagar Jha, Lorenzo Rosa, and Ken Birman. Spindle: Techniques for optimizing atomic multicast on rdma, 2021.

[26] Ameet V Joshi. Amazon's machine learning toolkit: Sagemaker. In *Machine Learning and Artificial Intelligence*, pages 233–243. Springer, 2020.

[27] 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.

[28] Sandeep S Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. Logical Physical Clocks. In *Principles of Distributed Systems*, pages 17–32. Springer, 2014.

[29] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided {RDMA} reads to build a fast, cpu-efficient key-value store. In *2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, pages 103–114, 2013.

[30] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, October 2018. USENIX Association.

[31] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*, 2017.

[32] Weijia Song, Theo Gkountouvas, Ken Birman, Qi Chen, and Zhen Xiao. The freeze-frame file system. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, page 307–320, New York, NY, USA, 2016. Association for Computing Machinery.

[33] Edward Tremel, Sagar Jha, Weijia Song, David Chu, and Ken Birman. Reliable, efficient recovery for complex services with replicated subsystems. In *DSN*, pages 172–183. IEEE, 2020.

[34] Kostas Tzoumas. High-throughput, low-latency, and exactly-once stream processing with apache flink™. https://bit.ly/3pHLkaA, 2015.

[35] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. Apus: Fast and scalable paxos on rdma. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 94–107, 2017.

[36] 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.

[37] 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.

[38] 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.

A Time Synchronization with PTP

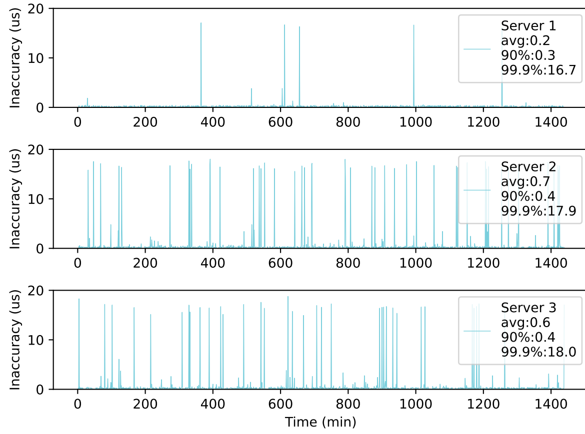


Figure 14: Time Synchronization with PTP

The server clocks are synchronized to a master clock in the directly connected switch with Precision Time Protocol (PTP) [20]. Figure 14 shows how skew between the server clocks and the master clock varies over time. The Y-axis (inaccuracy) is calculated from the output of Linux PTP daemon by adding the root mean square of the sampled offsets to the average offset in two seconds. Although some sporadic noises go up to more than ten microseconds, the average offsets are bounded in one microsecond. This is adequate for the open loop latency measurement in our evaluation.