frontiers | Frontiers in High Performance Computing

TYPE Original Research
PUBLISHED 17 September 2025
DOI 10.3389/fhpcp.2025.1638203

OPEN ACCESS

EDITED BY
Sarah Neuwirth,
Johannes Gutenberg University Mainz, Germany

REVIEWED BY
Francisco F. Rivera,
University of Santiago de Compostela, Spain
Eero Vainikko,
University of Tartu, Estonia
Hendrik Nolte,
Max Planck Society, Germany

*CORRESPONDENCE
Matthieu Dorier
✉ mdorier@anl.gov

RECEIVED 30 May 2025
ACCEPTED 07 August 2025
PUBLISHED 17 September 2025

CITATION
Dorier M, Gueroudji A, Hayot-Sasson V, Nguyen HD, Ockerman S, Souza R, Bicer T, Pan H, Carns P, Chard K, Chard R, Gonthier M, Huerta E, Lenard B, Nicolae B, Patel P, Wozniak J, Foster I, Rao NS and Ross RB (2025) Toward a persistent event-streaming system for high-performance computing applications.
Front. High Perform. Comput. 3:1638203.
doi: 10.3389/fhpcp.2025.1638203

Toward a persistent event-streaming system for high-performance computing applications

Matthieu Dorier[1]*, Amal Gueroudji[1], Valérie Hayot-Sasson[2], Hai Duc Nguyen[1,2], Seth Ockerman[3], Renan Souza[4], Tekin Bicer[1], Haochen Pan[2], Philip Carns[1], Kyle Chard[1,2], Ryan Chard[1], Maxime Gonthier[1,2], Eliu Huerta[1], Ben Lenard[1], Bogdan Nicolae[1], Parth Patel[1], Justin Wozniak[1], Ian Foster[1,2], Nageswara S. Rao[5] and Robert B. Ross[1]

[1]Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL, United States, [2]Department of Computer Science, University of Chicago, Chicago, IL, United States, [3]School of Computer, Data & Information Sciences, University of Wisconsin-Madison, Madison, WI, United States, [4]National Center for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge, TN, United States, [5]Computational Sciences and Engineering Division, Oak Ridge National Laboratory, Oak Ridge, TN, United States

High-performance computing (HPC) applications have traditionally relied on parallel file systems and file transfer services to manage data movement and storage. Alternative approaches have been proposed that use direct communications between application components, trading persistence and fault tolerance for speed. Event-driven architectures, as popularized in enterprise contexts, present a compelling middle ground, avoiding the performance cost and API constraints of parallel file systems while retaining persistence and offering impedance matching between application components. However, adapting streaming frameworks to HPC workloads requires addressing challenges unique to HPC systems. This paper investigates the potential for a streaming framework designed for HPC infrastructures and use cases. We introduce Mofka, a persistent event-streaming framework designed specifically for HPC environments. Mofka combines the capabilities of a traditional streaming service with optimizations tailored to the HPC context, such as support for massively multicore nodes, efficient scaling for large producer-consumer workflows, RDMA-enabled high-performance network communications, specialized network fabrics with multiple links per node, and efficient handling of large scientific data payloads. Built using the Mochi suite of HPC data service components, Mofka provides a lightweight, modular, and high-performance solution for persistent streaming in HPC systems. We present the architecture of Mofka and evaluate its performance against Kafka and Redpanda using benchmarks on diverse platforms, including Argonne's Polaris and Oak Ridge's Frontier supercomputers, showing up to 8× improvement in throughput in some scenarios. We then demonstrate its utility in several real-world applications: a tomographic reconstruction pipeline, a workflow for the discovery of metal-organic frameworks for carbon capture, and the instrumentation of Dask workflows for provenance tracking and performance analysis.

KEYWORDS

HPC, I/O, streaming, Mochi, Mofka, Kafka, Redpanda

1 Introduction

High-performance computing (HPC) applications and workflows are becoming increasingly complex as they integrate multimodal data streams from computations, simulations, and/or instruments. Their executions require the persistence of data and computing state in the presence of dynamic events such as arrival of new measurements and outputs of computing modules. HPC applications have traditionally used a parallel file system (PFS) such as Luster (Donovan et al., 2003) and GPFS (Schmuck and Haskin, 2002) to store and manage large amounts of scientific data. This approach simplifies use, as it provides a similar file system interface to those used on personal computers. The PFS also makes data portable across machines and easy to transfer using common file system tools. APIs and libraries such as MPI-IO (Corbett et al., 1996), HDF5 (Folk et al., 2011), ADIOS (Godoy et al., 2020), and pNetCDF (Li et al., 2003) provide scalable methods for reading and writing files at scale. However, the PFS approach has limitations that researchers have sought to overcome for several decades. The POSIX consistency model forces the use of locking or synchronization mechanisms, hindering performance (Wang et al., 2021). The need to serialize data into a flat file format introduces overheads, and the infamous computation-and-I/O gap causes a performance bottleneck at scale (Lockwood et al., 2017).

To overcome these limitations, HPC I/O research has moved in two directions in parallel, beyond simply trying to make file I/O more efficient. On the one hand, I/O that requires persistence, such as checkpointing, motivated the development of services such as FTI (Bautista-Gomez et al., 2011) and VeloC (Nicolae et al., 2021, 2019), which exploit the available storage hierarchy (RAM, local SSD, burst buffers, and parallel file systems) and non-blocking I/O and do not represent data as files. On the other hand, I/O meant for scientific discovery, carrying data that still requires postprocessing, led to *in situ* analysis approaches (Childs et al., 2020), which couple analysis and visualization software with HPC applications in a way that bypasses the file system altogether.

Meanwhile, new classes of I/O traffic have emerged. Scientific workflows are becoming more complex and more heterogeneous, requiring more elaborate strategies to manage data movement. Tools for provenance capture (Souza et al., 2023), telemetry (Adamson et al., 2023), and anomaly detection (Kelly et al., 2020) produce background I/O traffic from many sources. The coupling of supercomputing resources with scientific instruments such as light sources, telescope arrays, and gravitational wave detectors introduces new challenges for data management (Bicer et al., 2020). Machine learning and, in particular, federated learning require interactions across facilities (Ryu et al., 2022). Infrastructures such as Globus (Chard et al., 2016, 2023; Zheng et al., 2024) still make it possible to use the file model, but these new access patterns highlight the fact that neither parallel file systems nor current data services, let alone tight-coupling approaches such as *in situ* processing, are a good fit in all cases for HPC data management.

Complex HPC workflows used to run many scientific applications generate enormous quantities of data that can be abstracted as streams of "events" of various types, from small telemetry events to large scientific datasets. State-of-the-art distributed event-streaming services such as Kafka (Wang et al., 2015) and Redpanda[1] show how event-driven models can improve scalability by decoupling producers and consumers at scale in internet service environments. Achieving comparable impacts in scientific computing. however, requires more than simply deploying such services on HPC systems (Chantzialexiou et al., 2018). For example, scientific workloads have unique characteristics including large event sizes, specialized data formats, and diverse producer-consumer applications, while HPC systems offer high-performance network technologies (e.g., HPE Slingshot, Cornelis Omni-Path), communication methods [e.g., RDMA (Javed et al., 2017; Taranov et al., 2022; Lu et al., 2016)], and heterogeneous storage devices that can significantly improve streaming performance.

In this paper we explore the potential for an event-driven data management architecture designed specifically for HPC. We propose Mofka, a prototype of such a streaming framework, developed using the Mochi suite of components for HPC data services (Ross et al., 2020). Mofka natively leverages HPC networks and technologies such as RDMA, as well as efficient use of multithreading, to offer higher throughput than state-of-the-art systems such as Kafka and Redpanda. Experiments on Argonne's Polaris and Improv supercomputers and Oak Ridge's Frontier exascale supercomputer show up to 8× improvement in throughput in some scenarios.

The second contribution of this paper is the presentation of three use cases where an event-driven approach, supported by Mofka, has been leveraged in HPC applications. These use cases are 3D tomography reconstruction (TekApp), the discovery of novel metal–organic frameworks (MOFs) with AI-driven molecular simulations (MOFA), and the performance characterization and provenance of HPC workflows (Flowcept).

The rest of this paper is organized as follows. Section 2 presents motivation for using persistent streaming systems in an HPC context by drawing a parallel with their use in businesses. Section 3 introduces Mofka. Section 4 evaluates Mofka on multiple machines and compares its performance with that of Kafka and Redpanda. Section 5 presents three real-world use cases for HPC event streaming, showcasing the broader utility of the Mofka framework. Section 6 compares Mofka with related works. Section 7 presents a summary and a brief look at future work.

2 Motivation: event-driven architectures

The event-driven architecture is presented as a solution to these problems. A distributed, persistent, resilient event-streaming system is set up as the single source of truth and as the means for departments/applications to communicate, with each having a set of topics it produces information to or consumes from. A schema is still required for applications to understand each other, but these schemas are managed by a schema registry, making it easy for changes to be picked up automatically. A fault in an application no

1 Redpanda-Data. *Redpanda*. Available online at: https://www.redpanda.com/.

longer affects other applications, because the faulty application will simply restart and pick up where it left off in the stream of events it subscribed to.

Much as exploratory data analysis (EDA) solves data management challenges in large businesses, we argue that it could also overcome most of the challenges posed by existing HPC data management approaches such as parallel file systems, data services, and *in situ* processing. HPC applications are natural producers and consumers of immutable data. They do not need file consistency semantics, and they do not need file formats (*in situ* analysis is an evidence of that fact). EDA brings back persistence, resilience, and impedance matching, which are missing from the *in situ* approach. It naturally reconciles offline and inline processing and has the potential for being the backbone of complex workflows. EDA is also well suited to the needs of emerging HPC workflows, including provenance capture, anomaly detection, communications with scientific instruments, and cross-facility interactions.

Event-streaming software such as Kafka (Wang et al., 2015) and Redpanda (see footnote 1) provide an EDA solution for businesses. Cloud providers such as AWS and Confluent provide cloud-managed Kafka deployments. Azure Event Hubs accept the Kafka protocol. However, these software products have been developed for cloud environments (whether public or private). They do not make use of features specific to HPC systems such as massively multicore nodes, high-performance networks, remote direct memory access (RDMA), and multiple NICs; and they do not cater to application spanning hundreds of thousands of processes. In the following, we introduce Mofka, an event-streaming service specifically designed for HPC use cases.

Event stores are often described as **a database turned inside out**[2] (Stopford, 2018). The question this paper aims to answer is, in a sense, **What does a parallel file system turned inside out look like?**

3 The Mofka event-streaming framework

While heavily inspired by the likes of Apache Kafka, Mofka aims to better support HPC application by leveraging the hardware available on high-end supercomputers. This section starts by listing design goals, before diving into Mofka's architecture, its API, and its implementation.

3.1 Design goals

For Mofka to be suitable for an HPC context, we designed it with the following goals in mind.

1. **Efficient network usage.** HPC machines rely on high-performance networks such as Infiniband, Omni-Path, or Slingshot. While the TCP stack is available on top of these networks, it is not the best transport to maximize throughput and minimize latency. Instead, an event-driven system for HPC should natively use the high-performance transport, including RDMA capabilities, to transfer data directly from the application's memory to the target storage device with as few data copies as possible and as little kernel and CPU involvement as possible. Furthermore, we start to find compute nodes with multiple NICs. Argonne's Polaris features two NICs per node, and Aurora features eight. Making efficient use of all the NICs is also critical.
2. **Efficient threading.** Compute nodes on HPC machines feature an increasingly larger number of cores. Hence, an event-driven system may be used concurrently by a large number of threads or processes. Such a large number of cores is also an opportunity for the event-driven system to offload tasks such as serialization, validation, and batching. An event-driven system for HPC should make efficient use of this multicore environment.
3. **Handling of large data.** Event-driven systems in enterprises are typically used for small, structured messages. In an HPC context, users may want to attach large payloads (multiple megabytes to multiple gigabytes) to each event. An event-driven system for HPC should be able to handle a wider range of event sizes than traditional event-driven systems can handle.
4. **Support for scientific data.** HPC applications usually produce multidimensional arrays of numerical data. Consumers of such data may be interested in only some sections of such arrays. An event-driven system for HPC should allow consumers to specify such interest *before* the event is transferred, in order to improve transfer speeds.
5. **High modularity.** An event-driven system for HPC should adapt to vastly different supercomputer hardware and application workloads. This calls for a modular design where everything from the network stack, to internal scheduling policies, or to local storage usage can be changed and fine-tuned for each individual use case.

These design goals led to the architecture and implementation described hereafter.

3.2 Overview of Mofka's architecture

Mofka builds on the Mochi suite of technologies for developing HPC data services (Ross et al., 2020). The use of Mochi helps address the five design goals listed above. Mochi provides a collection of portable software components, together with a methodology to compose them together into specialized HPC services tailored to particular use cases. Such components include efficient RPC/RDMA targeting vendor-specific HPC networks such as Slingshot and Verbs; efficient threading using a mix of hardware- and user-level threads (co-routines) to leverage massively multicore nodes; and key/value storage and blob storage components, among other components, that have been designed to handle large, scientific data with a modular design.

Mochi has been used to develop numerous HPC data services, including distributed file systems (Vef et al., 2018; Brim et al., 2023; Tatebe et al., 2022), *in situ* analysis and visualization services (Ramesh et al., 2022; Dorier et al., 2022), and object stores (Hennecke, 2020). Mofka shares most of its components with these existing services, assembling them in a manner suitable to the

2 Example: https://www.kurrent.io/blog/turning-the-database-inside-out.

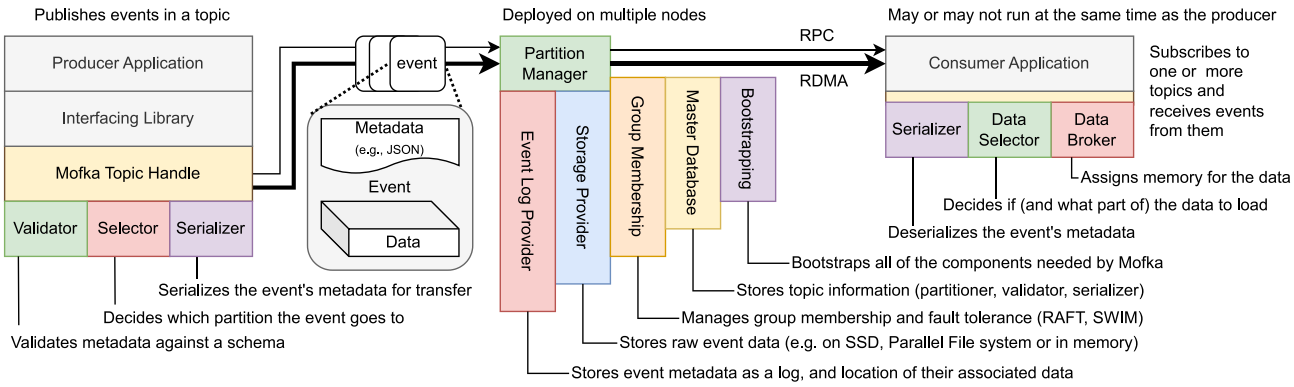

FIGURE 1
Overview of the Mofka architecture, including client libraries (producer and consumer). Mofka is an assemblage of Mochi components handling distinct aspects of the service, from bootstrapping, to storing the metadata and data that compose events, to managing group membership and global information.

task at hand and offering a streaming interface on top. This section dives into Mofka's architecture, including design decisions that we made to ensure its suitability to HPC use cases.

Figure 1 shows the Mofka architecture. A key difference between Mofka and other streaming services such as Kafka is that Mofka divides events into two parts: a *metadata* part, which is assumed to be small and structured, that is, interpretable by both clients and servers (such as a JSON document), and a *data* part, which can be much larger and would carry a scientific payload that is opaque to the server.

This division matches typical HPC use cases where large amounts of data may need to be transferred and persisted together with some metadata describing it. For example, in a distributed deep-learning application, the data part could contain an update to apply to a particular layer of a deep neural network, in the form of a list of tensors, while the metadata could contain information on how this update was produced, for provenance tracing. In a workflow coupling a light source with a supercomputer, the data could be an X-ray diffraction image, and the metadata could contain the settings used by the sensors and information about the sample being scanned.

This division of metadata and data allows Mofka to use different tools to transfer the two parts and independently optimize the control and data path in order to minimize overhead. Metadata parts are serialized into batches that are sent via RDMA to an intermediate buffer in the server, before making their way to a "log provider." Data parts, on the other hand, are transferred by using RDMA directly to and from user-provided memory and to and from their final storage destination. Structuring it in such a way helps avoid serialization overhead, reduce memory overhead, and leverage direct RDMA transfers for the highest volume portion of each event.

3.2.1 Producing events

Producer applications (on the left of Figure 1) interact with the service via a *topic handle*. This handle contains a *validator*, a *partition selector*, and a *serializer*: three user-provided objects that work on an event's metadata before it makes its way into a batch. The validator checks that the metadata satisfies a particular format expected for the topic (for instance, that it is valid JSON or that this JSON satisfies a particular schema). The partition selector decides which partition a particular event will be sent to. It can do so based on the metadata content or based on any algorithm (e.g., random, round-robin). The serializer takes the metadata and writes it into a batch. As an example, the validator could check that the metadata is a valid JSON object with a floating point "energy" attribute (e.g., `{"energy": 123.456}`); the partition selector could decide on the target partition based on this energy value, and the serializer could simply write the 8-byte double value rather than the full JSON string.

Once a batch is formed, it is sent to a *partition manager*, the component in charge of a single partition in a server. This partition manager redirects the batch of metadata to a log provider and the RDMA handles for the data to a *storage provider*. It then creates the association between each metadata and each data part and adds it to the log provider.

Similar to the concept of schema registry in event-driven systems, the validator, partition selector, and serializer are provided when creating a topic and are stored in a master database in Mofka. This ensures that all producer and consumer applications use the same objects for a given topic.

3.2.2 Mofka servers

Mofka itself consists of a set of server processes running the components shown in Figure 1. As we will show in Section 3.3, the user has full control over where these components live, where each partition is located, and how it is implemented.

One partition manager handles a single partition of a single topic; hence, creating a topic and its partitions means instantiating the necessary number of partition managers across multiple processes.

Mofka partition managers keep track of which consumer is subscribed to them and actively fetch batches of metadata from the log provider to send to the consumer. However, it does not proactively send them the data part of each event, as will be explained next.

3.2.3 Consuming events

Consumer applications (right part of the picture) also interact with Mofka using a *topic handle*, by subscribing to a given set of partitions from the topic. The topic's serializer, retrieved from the master database, allows deserializing the metadata part of the received events.

Mofka consumers must be initialized with user-provided *data selector* and *data broker* objects. A Mofka partition manager only sends the metadata part of events, in batches. Upon receiving and deserializing them, the consumer invokes its data selector to decide which part of the data (all, none, or any arbitrary subset) it is interested in. This is another place where the division of events into a metadata part and a data part allows transfer optimizations, since the consumer application can decide to transfer only a subset of the data part. This also means that multiple consumers could consume from the same topic but with a focus on different parts of the data. As an example, in a weather application where events contain a 3D "chunk" of atmosphere, a consumer may be selecting a 2D slice at ground level to display a color map, while another consumer could pull the full data for volume rendering.

Data selection could also be based on the metadata of an event. For instance, a producer could add statistics about the data in each event's metadata, such as the minimum, maximum, and average temperature values in the region represented by the event. The consumer could then decide to select only the data for which the average exceeds a certain threshold.

After the data selector has described which part of the data should be pulled, the data broker is executed. Its role is to tell the Mofka client library where to transfer the data. This gives consumer application control over where the data should ultimately be placed, thereby enabling efficient direct RDMA transfer with no intermediate buffering.

3.3 Mofka interface

Mofka provides a C++ interface and Python bindings, in addition to a command-line interface for creating and managing topics. Listing 1 shows the creation of a "collisions" topic and the addition of a partition to it. The created topic will use a validator, partition selector, and serializer implemented in C++ and compiled into dynamic libraries. These objects are also individually configured for the particular use case.

Listing 2 shows an example of producer in Python. Of note is the control over the number of background threads to use to carry out the validation, partition selection, and serialization, as well as the batch size for event metadata. The producer's `push` function is non-blocking, returning a future that can be awaited. Alternatively, the producer can be `flush`ed periodically.

Listing 3 shows an example of consumer in C++, including a data selector and a data broker. Just like the producer's `push` function, the consumer's `pull` function is non-blocking. Its returned future can be awaited, returning an event with a `metadata` and a `data` field.

```
mofkactl topic create collisions --groupfile mofka
     .json \
    --validator energy_validator:
     libenergy_validator.so --validator.energy_max
     100 \
    --partition-selector energy_selector:
     libenergy_selector.so --partition-selector.
     energy_max 100 \
    --serializer energy_serializer:
     libenergy_serializer.so --serializer.
     energy_max 100 \

mofkactl partition add collisions --groupfile
     mofka.json --rank 0 \
    --type default --metadata
     my_metadata_provider@local \
    --data my_data_provider@local
```

Listing 1. Topic and partition creation using the Mofka CLI.

```
from mochi.mofka.client import MofkaDriver, \
                               ThreadPool
# First, a MofkaDriver should be created
driver = MofkaDriver(group_file="mofka.flock")
# An existing topic can be opened
topic = driver.openTopic("my_topic")
# A producer is created for the topic,
# configured with a ThreadPool and a batch size
producer = topic.producer(
    name="app1"
    thread_pool=ThreadPool(4),
    batch_size=128)
for i in range(0, 100):
    # The metadata can be a string or a dictionary
    metadata = {"x": i*42, "y": [1, 2, 3]}
    # The data must satisfying the buffer protocol
    # (bytearray, memoryview, numpy, etc.)
    data = np.random.randint(100, size=(3, 5))
    # push is non-blocking and returns a Future
    future = producer.push(
        metadata=metadata,
        data=data)
    # the future can be waited on...
    event_id = future.wait()
    # ... or the producer can be flushed
    producer.flush()
```

Listing 2. Example of Mofka Python producer.

```
#include <mofka/MofkaDriver.hpp>
using namespace mofka;

MofkaDriver driver{"mofka.flock"};
auto topic = driver.openTopic("my_topic");
// the selector selects a part of the data
auto selector = [](const Metatadata& md,
                   const Descriptor& dd) {
    // only interested in a subset of the data
    return dd.makeSubView(2, dd.size());
};
// the broker decides where to place it
auto broker = [](const Metatadata& md,
                 const DataDescriptor& dd) {
    return Data{
        new int[dd.size()/sizeof(int)],
        dd.size()};
};
// the consumer is created
auto consumer = topic.consumer(
    "app2", selector, broker, ThreadPool{4});
// we can now pull events
auto future = consumer.pull();
auto event = future.wait();
// event.id(), event.metadata() and event.data()
// can be used to access the event's content
```

Listing 3. Example of Mofka C++ consumer.

We also implemented a "Kafka adapter" for Mofka. That is, the Mofka API can be used to interact with a deployment of Kafka, Redpanda, or any similar system satisfying the Kafka protocol. It does so by using librdkafka,[3] a widely used C implementation of the Kafka protocol, and by adding Argobots-based threading on top of it to provide some of the features from the Mofka client library that are missing from librdkafka. Hence, HPC applications that use the Mofka API can also be tested on top of Kafka or Redpanda. We use this feature in particular for benchmarking against these systems in Section 4.

3.4 Implementation and modularity

As stated earlier, Mofka is built using the Mochi suite of technologies, which advertises a composable approach to the design of HPC data services, with reusable components at the center of such design.

Figure 1 shows that Mofka servers are made of multiple components, only one of which is in fact exclusive to Mofka: the partition manager. The event log consists of a Yokan component,[4] which provides key/value and document collection storage. The storage provider is a Warabi component,[5] which provides blob storage functionality. Group membership is managed by Flock,[6] the master database is another Yokan component, and bootstrapping and configuration are handled by Bedrock.[7] Other components such as Poesie[8] and REMI[9] will eventually be added to the mix to provide in-server script execution and file migration, respectively.

Mofka also relies on Plumber,[10] a Mochi component for topology-aware network selection. We spawn multiple Mofka server processes on each node; and, should the node provide more than one NIC, Plumber will automatically assign processes to their closest NIC, ensuring maximum use of the network resources available.

This component-based design has multiple advantages (Ross et al., 2020). First, all the components listed above are used in other HPC data services, within and outside our group, constantly benefiting from bug fixes, extensions, and performance optimizations as a result. Second, compartmentalization of functionality enables rapid innovation without perturbing the overall design. For example, an innovative design for an event log using a specialized hardware could be implemented as a backend for Yokan and immediately be tested in Mofka, without any changes to Mofka's design or to other components.

3 https://github.com/confluentinc/librdkafka

4 https://github.com/mochi-hpc/mochi-yokan

5 https://github.com/mochi-hpc/mochi-warabi

6 https://github.com/mochi-hpc/mochi-flock

7 https://github.com/mochi-hpc/mochi-bedrock

8 https://github.com/mochi-hpc/mochi-poesie

9 https://github.com/mochi-hpc/mochi-remi

10 https://github.com/mochi-hpc/mochi-plumber

3.5 A parallel file system turned inside out

Distributed streaming frameworks such as Apache Kafka are often described as "a database turned inside out" because they invert the conventional architecture of data-centric systems. Traditional databases prioritize storage and subsequent querying of data, typically treating the change log as an internal implementation detail. In contrast, Kafka exposes the append-only commit log as a first-class abstraction, positioning it as the central system of record. Data is ingested as an immutable, ordered sequence of events, which can be consumed, processed, and replayed by multiple downstream systems. This design enables decoupled, real-time data processing pipelines and supports scalable, event-driven architectures that are fundamentally more transparent and flexible than traditional database systems.

In this sense, Mofka be viewed as a "parallel file system turned inside out." Traditional parallel file systems require users to reason carefully about data layout within files to achieve acceptable performance, often entangling application logic with low-level storage considerations. Mofka inverts this paradigm by replacing file-based I/O with persistent, ordered streams of events, allowing users to focus on the semantics of data production and consumption rather than its physical organization on disk. By decoupling data movement from file management, Mofka enables applications to exchange data efficiently and asynchronously across distributed nodes, without sacrificing durability or throughput. This abstraction simplifies the development of real-time, event-driven workflows in HPC systems, aligning with the need for low-latency communication and scalable I/O in modern scientific applications.

4 Evaluation

We begin our empirical evaluation using carefully controlled synthetic benchmarks to compare Mofka with two existing persistent messaging systems, Kafka and Redpanda, on Argonne's Improv and Polaris, described hereafter. Redpanda is an alternative to Kafka relying on the same protocol.

For this study we focus on scientific use cases (see Section 5) that use batch-scheduled ephemeral resources. We thus disable replication and focus on the upper-bound performance offered by each system. We will introduce replication and consensus and study these in greater detail in future work.

4.1 Platforms

We ran our benchmark on two production HPC systems at Argonne National Laboratory, described below. In this section we also describe Oak Ridge National Laboratory's Frontier, which we use in a later section.

Improv[11] is a 2.51 PFlop/s machine with 825 nodes. Each node is equipped with two AMD EPYC 7713 64C 2 GHz processors (128 cores per node), 256 GB DDR4 memory, and a 960 GB NVMe SSD. The nodes are interconnected via InfiniBand HDR200.

11 https://www.lcrc.anl.gov/systems/improv

Polaris[12] is a 34 PFlop/s machine with 560 nodes interconnected by a 200 Gbps HPE Slingshot 11 network in a Dragonfly topology with adaptive routing. Each node features an AMD EPYC "Milan" processor (32 cores per node), two Slingshot endpoints, and two 1.6 TB NVMe SSD each offering up to 3300 MB/s sequential write performance and 3,300 MB/s sequential read performance, and arranged in a RAID-1.

Frontier[13] will be used in Section 5 but is introduced here for completeness. It is a 1.6 exaflop/s machine with 9,402 nodes connected with 4x HPE Slingshot 200 Gbps NICs providing a node-injection bandwidth of 800 Gbps. Each compute node has two 1.92 TB NVMe SSDs, although we will deploy Mofka in memory in experiences on this platform.

4.2 Benchmark design

Properly benchmarking messaging systems is difficult. Although the Open Messaging Benchmark[14] is, to the best of our knowledge, the only open-source benchmark available to evaluate the performance of messaging systems, this benchmark is no longer maintained and is not adapted to the deployment of many producers and consumers as MPI jobs on a supercomputer. Hence, we had to design our own benchmark.

Our benchmark is written in C++ for two reasons. The first is to avoid any language binding overheads; the second is to streamline integration with librdkafka.

We are interested in aggregate throughput from a large number of producers or consumers. Hence our benchmark consists of an MPI application that starts N client processes on multiple cores of multiple nodes. This setup is representative of how HPC applications would use Mofka or any event-driven system. In producer mode, each process produces a desired number of messages of a specified size to a designated partition. We rely on a fixed assignment of producer to partition to avoid variations in performance caused by unbalanced load. While cloud-computing applications are more elastic, HPC workloads generally execute in fixed-size job, where the number of clients is known in advance.

In all our experiments we produce and consume 10 million events per partition. The messages are produced in batches of 1,000 messages for Mofka, and we let librkafka manage its own batching (librdkafka producers batch messages automatically and send the batches from a background thread). A `flush` operation is invoked every 10,000 events. One million events are produced as a warmup before we start measuring performance. In consumer mode, each process consumes messages from a specific partition, again with a fixed assignment to avoid the overhead of rebalancing operations. We consume events in batches of 1,000 events. We therefore consume in batches of 1,000 events both with Mofka and with librdkafka.

12 https://www.alcf.anl.gov/polaris

13 https://www.olcf.ornl.gov/frontier/

14 https://openmessaging.cloud/docs/benchmarks/

4.3 Tuning considerations

Mofka, Kafka, and Redpanda are complex systems with many tuning parameters that can dramatically affect performance. Months of evaluating them and exchanging with the developers and users of Kafka and Redpanda led us to what we believe is the best performance we could achieve for each of these systems. For Kafka for example, our investigations included varying the number of I/O threads and network threads. Redpanda requires root permissions to tune kernel parameters, which we cannot do on production HPC systems. We therefore warn the reader that better performance may be achievable from Redpanda if set up by a system administrator with escalated privileges.

Lesson 1: Configuration parameters are highly sensitive.

Our experiments taught us that slight changes to configuration parameters or to the client code can have a significant impact on performance. We advise authors of any future studies on such systems, as well as any users and practitioners, to spend enough time investigating these aspects.

We use `rd_kafka_producev` to produce messages and let librdkafka's background thread handle batching, as recommended by practitioners. However, we use `rd_kafka_consume_batch_queue` to explicitly consume messages in batches, since `rd_kafka_consume` does not do it automatically.

Lesson 2: APIs can facilitate aggregation.

There is a significant performance benefit in allowing clients to batch operations into larger units to improve protocol efficiency. This behavior is enabled by using explicit interfaces rather than by configuration tuning.

Profiling the benchmark using the HPCToolkit (Adhianto et al., 2010) showed that the overhead of small memory allocations and data copies when handling a large number of events can drive down the performance for all messaging systems. Using `jemalloc`[15] or `mimalloc`[16] instead of `glibc`'s implementation of `malloc` dramatically improves performance. In all our tests we use `jemalloc` as a preloaded library because it led to the best performance gains. We also made sure that the benchmark does not do unnecessary copies or allocations.For example, producing and consuming 10 million events of 1 KB from a single client to a single server with Mofka lead to an average of 605 MB/s of production throughput and 1,250 MB/s of consumption throughput when using `jemalloc`. Without `jemalloc`, this same setup leads to a production throughput of 555 MB/s and a consumption throughput of 514.60 MB/s. More than half the performance of the consumer is due to simply using a better allocator. This kind of performance improvement is also observed on top of Kafka and Redpanda.

15 https://jemalloc.net/

16 https://microsoft.github.io/mimalloc/

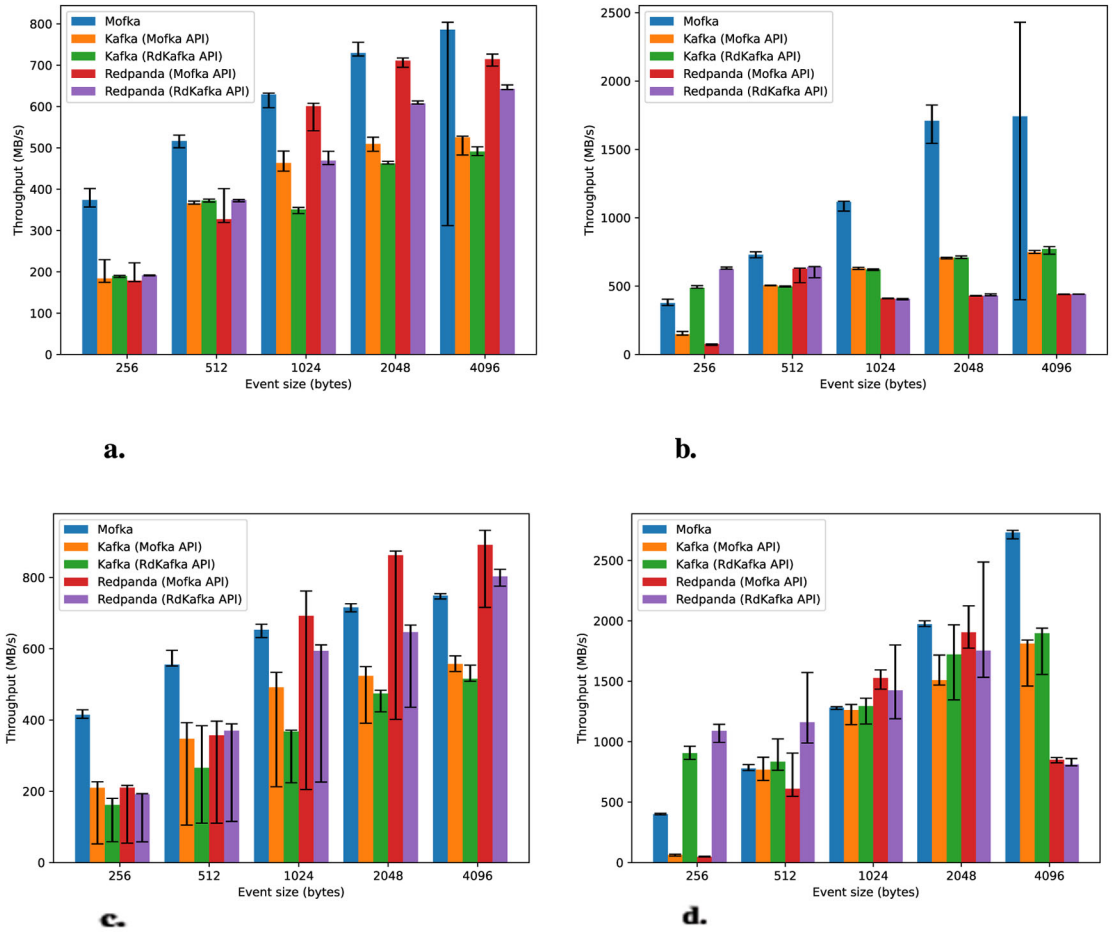

FIGURE 2
Production and consumption of 10 million events using a single server and a single client. Median result from 5 runs, with error bars representing the minimum and maximum values. Note that for readability, the figures use different scales. **(a)** Producers on Improv. **(b)** Consumers on Improv. **(c)** Producers on Polaris. **(d)** Consumers on Polaris.

Lesson 3: Memory allocations have a significant impact on performance.

Event-driven applications will often manipulate a very large number of small objects. Any strategy that avoids repeated allocation/deallocations, be it built-in or provided by a custom allocator, should be a first step to improve performance.

4.4 Single producer and consumer

We first run our benchmark on Polaris and Improv with one server node and one client node. The purpose of this experiment is to provide a baseline for production and consumption in conditions where no parallelism is involved and no interference can be expected. The client node is used to run a single producer, followed by a single consumer, accessing a single partition. We vary the size of events from 256 bytes to 4 kilobytes.

Figure 2 shows that Mofka outperforms Kafka and Redpanda in almost all scenarios on Improv, underperforming only when consuming messages of 256-bytes. The picture on Polaris is different. Whereas Mofka shows more stable performance, Redpanda performs better in the production of event 1 KB and up, and Mofka starts outperforming Kafka and Redpanda in consumption only for events 2 KB and up.

Note that the current version of the Slingshot software stack on Polaris causes the thread running the network progress loop to consume more CPU time than expected, which could be a possible factor in the discrepancy between Improv and Polaris under network-intensive workloads.[17]

These results also show the effect of using the Mofka API on top of librdkafka rather than using librdkafka directly. By relying on Argobots user-level threads, the Mofka API is often able to achieve better performance on top of Kafka and Redpanda.

17 https://github.com/ofiwg/libfabric/pull/10681

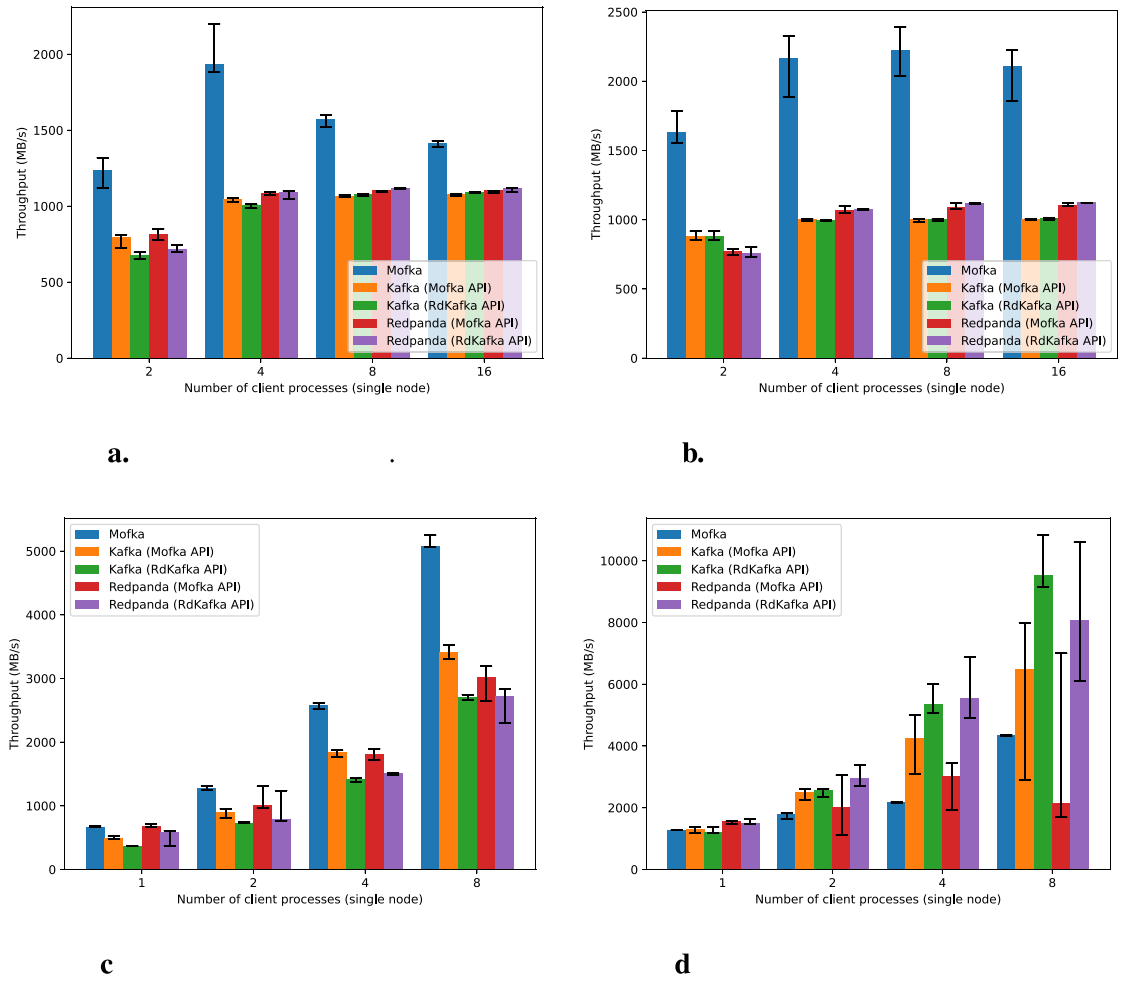

FIGURE 3
Production and consumption of 10 million events (of 1 KB each) per client using a single server node and a single client node and varying the number of client processes in this node. Median result from 5 runs, with error bars representing the minimum and maximum values. **(a)** Producers on Improv. **(b)** Consumers on Improv. **(c)** Producers on Polaris. **(d)** Consumers on Polaris.

Lesson 4: Careful multithreading is essential to performance on many-core HPC systems.

Differences in performance between the Mofka API and the direct use of librdkafka on top of Kafka or Redpanda highlight the importance of experimenting with client-side threading to make optimal use of available CPU resources.

4.5 Single server, multiple clients

With supercomputer networks providing extremely high bisection bandwidth, we can expect the streaming data between clients and servers to be an embarrassingly parallel task. Multiplying the number of both clients and servers by the same factor should increase throughput in proportion. Hence, two questions arise: (1) To what extent can we saturate a client node with concurrent processes, especially given the massively multicore nature of modern HPC nodes? and (2) To what extent can we saturate a single server by making it service an increasing number of client nodes, given the scale of today's machines? This section answers both questions.

We fix the event size to 1 KB and continue to produce and consume 10 millions of them, but this time *per client process*, varying the number of client nodes or the number of client processes within a single node. We keep a single server node (Mofka spawns multiple processes to take advantage of multiple NICs when relevant, e.g., on Polaris).

Figure 3 shows the performance of producers and consumers on top of Mofka, Kafka, and Redpanda, with one client node, varying the number of processes per node. Because both the Mofka and librdkafka client libraries rely on a number of background threads, we limit the number of

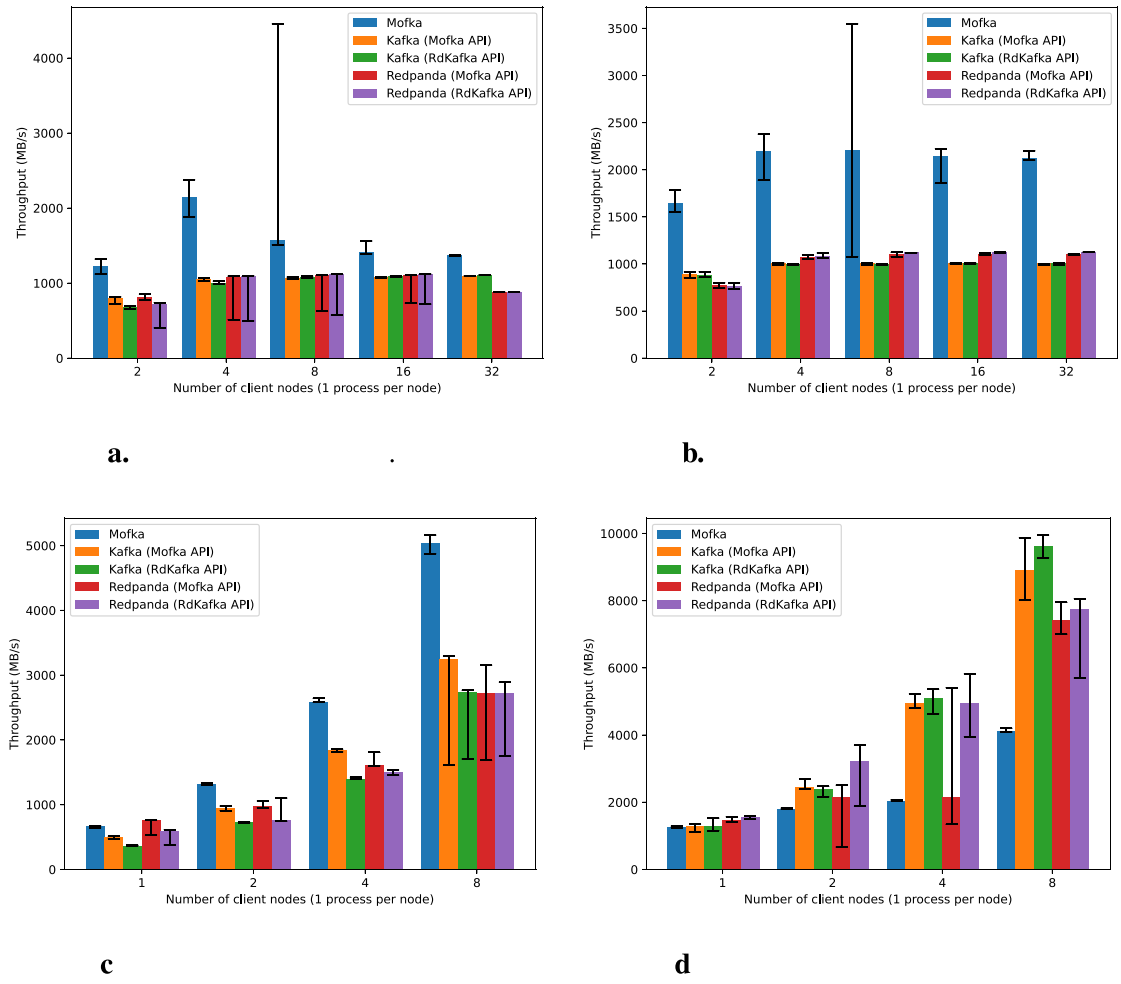

FIGURE 4
Production and consumption of 10 million events (of 1 KB each) per client using a single server node and varying numbers of client nodes (one process per client node). Median result from five runs, with error bars representing the minimum and maximum values. **(a)** Producers on Improv. **(b)** Consumers on Improv. **(c)** Producers on Polaris. **(d)** Consumers on Polaris.

processes so that we do not oversubscribe the cores on each platform (core binding is set such that each process is assigned exclusive access to a subset of cores). Oversubscribing cores led to operations frequently timing out with Kafka and Redpanda, causing experiments not to complete in their allocated time.

This figure shows that Mofka generally outperforms both Kafka and Redpanda on Improv, achieving up to 2× the throughput. On Polaris, however, while Mofka is better for production, it is slower for consumption.

On Polaris, the increase in throughput as we increase the number of processes (for a given system) shows that neither the network bandwidth nor the SSDs are the limiting factor in this scenario. Better performance can be obtained by producing concurrently to multiple partitions from the same client node. On Improv, however, four client processes achieve the maximum throughput, both in production and in consumption, after which performance stagnates for Kafka and Redpanda and decreases for Mofka. This indicates that either the network or the SSD becomes limiting.

Lesson 5: Higher throughput can be achieved from a single node through concurrent access.

Each of the messaging systems can handle multiple concurrent producers/consumers per node, making them well suited to HPC applications.

Consumption results on Polaris show that as the number of client processes increases, the use of the Mofka API becomes detrimental to performance compared with the direct use of librdkafka. We therefore suspect that the use of an Argobots thread, combined with librdkafka's background POSIX threads and more processes per node, causes interferences and generally poor performance, going back to Lesson 4: threading could often matter more than which system is used.

Next, we run our benchmark with one server serving a varying number of client nodes, using one process per client node. Figure 4 is similar to Figure 3, suggesting that (1) performance is limited by the server (either its network bandwidth or its storage), (2) more concurrent clients (whether from multiple cores in a node

or from multiple nodes) may be beneficial to performance, and (3) performance plateaus on Improv past a certain number of concurrent clients.

4.6 Multiple servers, single client node

Having shown the potential of concurrency on the application side, we look at the server side in this experiment. We run our benchmark on a single client node with either 8 or 16 processes, against multiple server nodes.

Results are shown in Figure 5. On Improv, Mofka throughput doubles as we double the number of server nodes, while that of Kafka and Redpanda stagnates. These results suggest that, on this machine, the use of TCP by Kafka and Redpanda is the limiting factor, here at the application level. This conclusion is also consistent with results from the previous experiments showing Kafka and Redpanda throughput being generally capped at 1 GB/s regardless of event size, number of client processes, or number of client nodes.

Lesson 6: More servers increase performance...

Concurrency within a single client node can benefit from using more servers, assuming the client network is not a bottleneck.

On Polaris we ran the same experiment with either 8 or 16 processes in a client node. In the latter case, because of background threads created by the Mofka client library and/or by librdkafka, the cores are oversubscribed. This dramatically changes the results. Eight processes in a node is not enough to take advantage of more servers for production and for consumption when it comes to Kafka and Redpanda. With 16 processes, we see more advantages to increasing the number of servers, especially when producing with Mofka. Kafka's and Redpanda's performance improves slightly when increasing the number of servers from 1 to 2 but then stagnates. This indicates a bottleneck at the application level. In consumption, Mofka generally underperforms Kafka and Redpanda and only benefits from using 2 servers instead of 1. Other systems do not benefit from more servers.

Lesson 7: ... or not.

Many factors may hinder the benefit of adding more servers, including network bottlenecks at the client level but also threading strategies, oversubscription, and API use, ultimately making it difficult to find a clear winner in the different systems.

4.7 Benchmarking conclusions

The conclusion from our benchmarking campaign (of which the above is only a representative subset) is that relying on HPC technologies such as RDMA and native high-performance networks often leads to better performance than using systems designed for a cloud environment. However, many factors also impact performance before the network even starts to play its part. Such factors include the number of processes in the application, the API used, the threading technique, process and thread placement (including oversubscription), and SSD speed.

A caveat not shown in the figures above is that the Kafka and Redpanda systems frequently timed out and failed to complete benchmarks as we increased concurrency. Considerable effort was needed to identify configuration permutations that would avoid this issue. This would make them difficult to use in an HPC environment without careful engagement from an experienced administrator. We believe that librdkafka was the common root cause of many of these issues for both systems, but we found it difficult to get community support for our particular problems.

Lesson 8: We should invest more in developing an HPC-ready event-driven system.

Comparing Mofka with Kafka (or Redpanda) for event streaming is like comparing Luster with NFS: the former is HPC-ready, uses high-performance networks, and offers the right set of features for file I/O in HPC applications. The latter is useful in an HPC context; and while the modern NFSv4 has made progress in handling parallel I/O, it will always rely on Ethernet and have performance limitations for HPC applications. Kafka's development started in 2011 and Redpanda's in 2019; each is production software backed by a company and used by large businesses. The fact that we were able, with limited resources and in a short time, to develop a competing prototype for the HPC space that outperforms them in many scenarios is an encouraging first step that we think justifies more investments in this direction.

4.8 Data vs. metadata

In the above experiments we used only the metadata part of events when using Mofka, since our comparison against Kafka and Redpanda required small events.

Table 1 shows the results of experiments where each event carries 1 KB of metadata and 10 MB of data. When using the metadata path only, performance suffers from copying large amounts of data into batches on the client side and intermediate buffers on the server side. When the data part of events is used to carry the 10 MB payload, Mofka can optimize transfer by using RDMA to and from the client's memory and to and from the final storage destination, bypassing any intermediate buffer. The performance improvement is most pronounced when using in-memory storage, since the bandwidth is not limited by that of an SSD.

Lesson 9: Separating metadata and data improves performance.

Since HPC applications typically generate large events, separating their payload into a metadata part and a data part improves performance by avoiding costly copies on the data path.

While Mofka allows consumers to select subsets of data from each event, improving performance by avoiding the transfer of data

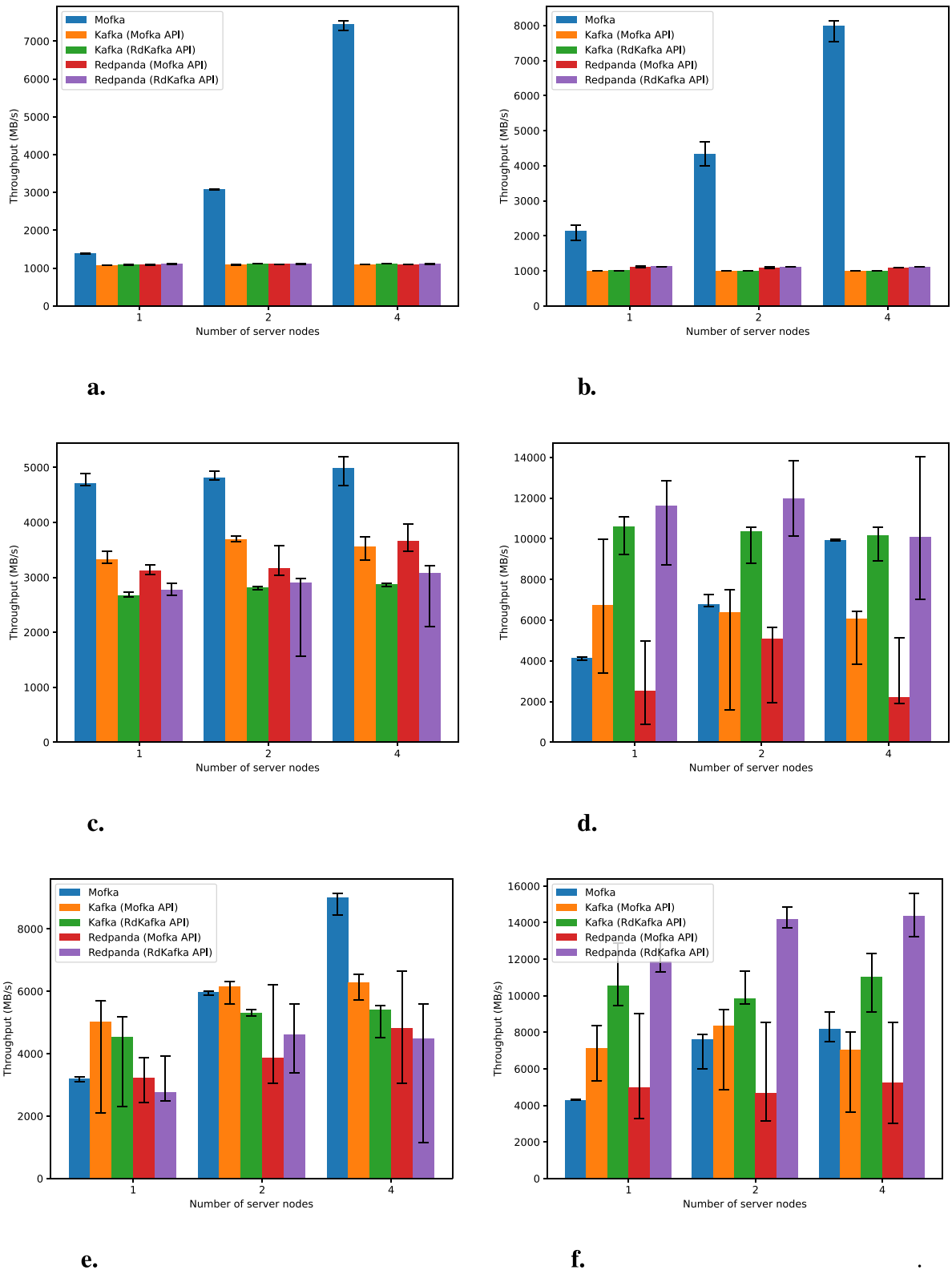

FIGURE 5
Production and consumption of 10 million events (of 1 KB each) per client process (8 or 16 processes on one node), varying the number of server nodes. Median result from 5 runs, with error bars representing the minimum and maximum values. **(a)** Producers on Improv (16 procs). **(b)** Consumers on Improv (16 procs). **(c)** Producers on Polaris (8 procs). **(d)** Consumers on Polaris (8 procs). **(e)** Producers on Polaris (16 procs). **(f)** Consumers on Polaris (16 procs).

TABLE 1 Performance of Mofka when producing/consuming events using either the metadata part only or separating their payload into a small (1 KB) metadata part and a large (10 MB) data part.

	SSD storage		In-memory storage	
	Metadata only	Metadata + Data	Metadata only	Metadata + Data
Production	794 MB/s	1,165 MB/s	955 MB/s	6,257 MB/s
Consumption	4,017 MB/s	7,763 MB/s	3,524 MB/s	8,529 MB/s

Performance of Mofka when producing/consuming events using either the metadata part only or separating their payload into a small (1 KB) metadata part and a large (10 MB) data part.

that is not needed, we do not evaluate this feature in the present paper.

5 Application use cases

We now present three use cases in which event-driven design benefits an HPC application or workflow.

5.1 Use case 1: TekApp

The reconstruction of 3D tomography datasets is a core application in X-ray imaging that provides valuable information about imaged samples and their morphologies. Here we work with a streaming tomographic reconstruction mini-app, "TekApp" (Bicer, 2024), derived from the Trace code (Bicer et al., 2017a), that performs real-time reconstruction on streams of tomographic data (Bicer et al., 2017b). This code provides a sliding window data structure to store incoming projections and a reconstruction process to update the object volume using the data in the window. The reconstruction algorithm is based on the Simultaneous Iterative Reconstruction Technique (SIRT) algorithm (Batenburg et al., 2009). It can continuously update reconstruction data as more data becomes available. The reconstruction quality can be improved with multiple configuration parameters (typically) at the cost of more computational demand and/or memory footprint (Bicer et al., 2020). The code is CPU-based and is optimized for shared- and distributed-memory parallelism. While this application employs X-ray imaging data generated at the Advanced Photon Source for tomographic tasks, its structure is applicable to a wide class of tomography applications, including scanning transmission electron microscopy (Al-Najjar et al., 2022).

TekApp comprises four components: (1) data acquisition (DAQ), (2) distribution (DIST), (3) reconstruction (SIRT), and (4) denoising (DEN). DAQ simulates the data acquisition step in the workflow; it reads the experimental data, X-ray projections, from an HDF5 file and streams them to the distributor component. DIST synchronizes with SIRT to retrieve the number of processes participating in the reconstruction, partitions the experimental data (sinograms), and distributes these to the participating SIRT task(s) accordingly. SIRT performs the reconstruction and then streams the reconstructed images to the DEN component. DEN reduces noise and improves image quality, particularly beneficial during the early stages of the reconstruction process (Liu et al., 2019, 2020).

The original TekApp mini-app was configured to use ZeroMQ for streaming. This approach requires components to perform a handshake at the start and synchronization throughout the workflow execution to avoid data transmission errors, for example, due to buffer overflows or intermittent component failures. This makes the intercomponent and task communication inherently synchronous and tightly coupled. With Mofka instrumentation, however, the workflow becomes more flexible, eliminating the need for components to coexist. They can operate at different times without any data loss. Additionally, the loosely coupled nature of the Mofka-instrumented version enables fault tolerance—if one component fails, the rest of the workflow remains unaffected.

Figure 6 shows how we adapted the TekApp workflow to employ Mofka for streaming. We augment the four existing components with Mofka server(s) (depicted at the bottom of the figure), which serve as intermediate storage between event producers and consumers, enabling asynchronous workflows where components can operate independently at different times. Currently, the system maintains synchronization between the DIST and SIRT components. During runtime, the number of reconstruction tasks is determined, and the DIST component is configured accordingly. However, this runtime dependency can be eliminated by using a shared configuration file or just by passing the needed configuration as parameters to the components, paving the way for a fully asynchronous pipeline. Mofka organizes events into topics, with each topic corresponding to a pair of communicating components in the workflow. For example, the `T_daq_dist` topic manages events transmitted from the DAQ component to the DIST component. Mofka clients are integrated with the components, enabling them to push events to or pull events from the Mofka servers. The two intermediate components, DIST and SIRT, act as both data consumers and producers. Each consumes data from its preceding component and generates new data for subsequent components. More components can be plugged into this architecture easily.

The tomographic reconstruction process, which takes the 2D (X-ray) projections as input, aims to generate a 3D (volumetric) object that represents the imaged sample. DAQ and Dist stream the 2D projections, and the 3D volumes are created in the SIRT component and denoised in the Den component.

5.1.1 TekApp experiments setup

We evaluated TekApp on the Polaris supercomputer. Table 2 shows the different settings for the four application components and the Mofka server. DAQ, DIST, and DEN are single Python process components, each deployed on one node. SIRT is a C++ MPI code deployed on 1 to 32 nodes (2 to 64 processes).

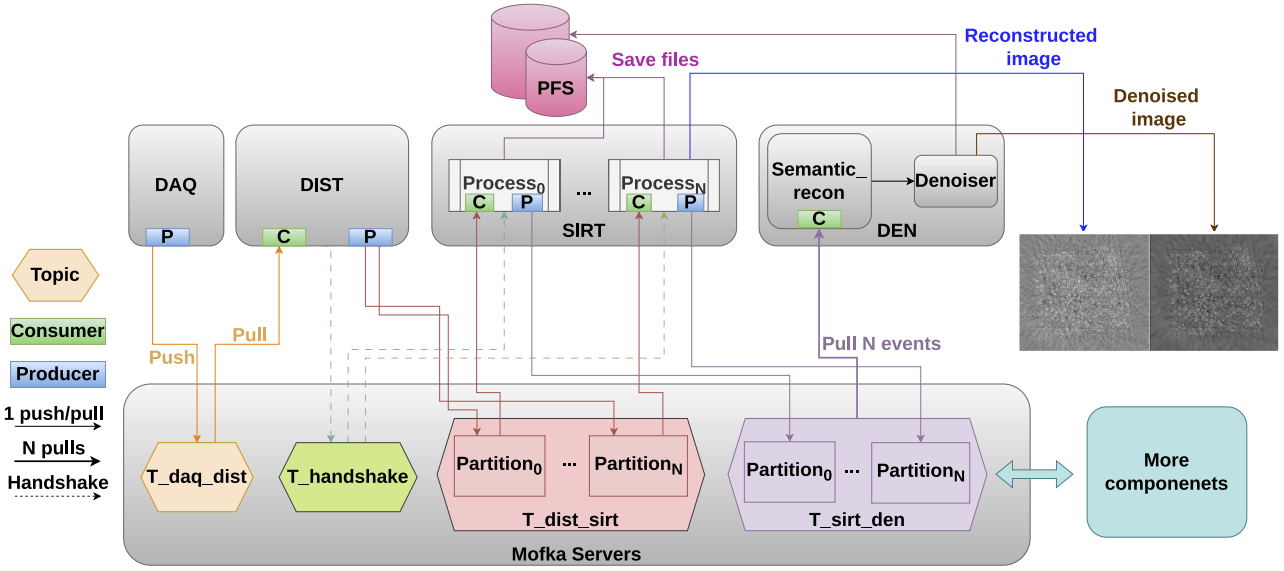

FIGURE 6
The four-stage TekApp mini-app—comprising DAQ, DIST, SIRT, and DEN components **(top)**—is connected to Mofka servers **(bottom)** to enable high-performance, reliable data streaming. Each Mofka topic (T_P_C) represents a communication channel between a producer (P) and a consumer (C). Reconstructed and denoised image examples are shown on the right.

TABLE 2 TekApp component configurations for DAQ, DIST, SIRT, and DEN. The [P] corresponds to data producers and [C] to data consumers.

Component	Nodes	Processes	Partitions	Data	Metadata	Total events
DAQ [P]	1	1	1	20 KB	30 B	1,500
DIST [C]	1	1	1	20 KB	30 B	1,500
DIST [P]	1	1	2–64	10 KB	148 B	3,000–96,000
SIRT [C]	1–32	2–64	2–64	10 KB	148 B	3,000–96,000
SIRT [P]	1–32	2–64	2–64	25 MB	290 B	188–6,016
DEN [C]	1	1	2–64	25 MB	290 B	188–6,016
Mofka server	1	2–64	9–129	4.64–47.8 GB	0.518–15.25 MB	7,867–103,516

The Mofka server is deployed on one node and uses 2 to 64 processes. DAQ pushes its data to a single partition in a `T_daq_dist` topic. DIST pushes its events to a `T_dist_sirt` topic with multiple partitions. The number of partitions is set to the number of SIRT processes (2 to 64); each SIRT process consumes from one partition and pushes new events to its dedicated target partition in the `T_sirt_den` topic (also set up with as many partitions as SIRT processes). The DEN component consumes data from all the partitions from the `T_sirt_den` topic.

Data and metadata sizes per event and the total number of events are shown in the table for each component. The last line (Mofka server) shows the total amount of data, metadata, and events transferred in this use case.

We evaluate Mofka's performance against the original ZeroMQ implementation as a first step, and then we conduct an in-depth analysis of Mofka's performance across a range of configurable parameters such as the batchsize, the number of partitions, and event sizes.

5.1.2 Mofka vs. ZeroMQ evaluation

To ensure a fair comparison with the ZeroMQ implementation, all components in the Mofka setup are executed concurrently. However, this concurrency may lead to conservative performance measurements for Mofka consumers, since the reported results can include idle time spent waiting for a batch of events to be produced. Table 3 presents a comparison of per-event overhead between Mofka and ZeroMQ. Both systems follow the configuration in Table 2, except that only two processes are used for SIRT[C] and SIRT[P]. Since ZeroMQ does not support batch size tuning, we compare it against Mofka using the best-performing batch sizes, as shown in the "Best Config" columns. The mean values of Mofka's `push` and `flush` combined and ZeroMQ's `send` are shown for the producers; and Mofka's `wait` and ZeroMQ's `receive` mean values are represented for the consumers. Mofka's `push` calls are blocking only if more than two complete batches have not been sent yet. We have added a call to `flush` that forces the producer to send all the buffered batches even if they are incomplete, after sending the last event in each component. The call

TABLE 3 TekApp producer/consumer per-event overhead: Mofka vs. ZeroMQ. The [P] corresponds to data producers and [C] to data consumers. Overhead is measured in microseconds.

Operator	Mofka (Memory)				Mofka (Default)				ZMQ		
	Best Config	Min	Med	Max	Best Config	Min	Med	Max	Min	Med	Max
DAQ [P]	64	2.73	3.77	1.2e6	64	3.32	3.71	4.6e6	40.1	46.7	260
DIST [C]	64	2.00	2.25	8.84	16	1.99	2.31	8.9e4	10.2	44.5	1.7e5
DIST [P]	64	3.57	7.06	1.1e6	16	3.65	7.24	1.5e7	103.4	232.6	1.2e6
SIRT [C]	1	0.02	0.10	5.2e4	1	0.02	0.11	3.5e5	7.31	384	6.5e5
SIRT [P]	64	7.39	8.24	2.6e5	32	7.15	7.97	2.3e6	3,373	4,062	4,108
DEN [C]	8	4.07	5.92	2.7e8	64	4.34	6.13	1.5e9	1,186	1,865	6,865

to `wait` on the consumer side is blocking only for the first event in the batch, because the Mofka events are transferred in batches rather than individually. ZeroMQ relies on direct point-to-point communication and requires frequent synchronization throughout the workflow to ensure reconstruction correctness. As a result, Mofka consistently outperforms ZeroMQ, achieving at least a 12× reduction in overhead across all components. The performance gap becomes even more significant for large events. For example, between the SIRT and DEN components, event sizes can reach 25 MB, causing ZeroMQ to incur over 1 ms of overhead per event. In contrast, Mofka reduces this overhead to under 10μs on average, delivering up to 500× better performance than ZeroMQ. Both Mofka and ZeroMQ exhibit significant variability in overhead, with maximum values often reaching up to six orders of magnitude above the minimum. For Mofka, however, this variability is due mainly to rare outliers (blocking `push` or `flush`), as the median overhead remains consistently close to the minimum, indicating more stable and predictable performance in typical cases.

Median-based performance metrics can be deceptively optimistic in configurations where the operational workload is heavily imbalanced. Consider, for example, the behavior of SIRT producers using a batch size of 64. Each process issues 94 non-blocking `push` operations—each incurring very low overhead—followed by a single blocking `flush`, which is considerably more expensive. In such cases, the median overhead is within the lightweight operations, effectively obscuring the cost of the rare but critical `flush`. As a result, batch size 64 may appear favorable based on median metrics, while in reality it introduces significant performance penalties that are hidden in the aggregate, which is the transfer of 94 25 MB-event at once, as we can see clearly in the max value. In the following sections we present a detailed component-level analysis demonstrating why this configuration is suboptimal and how a more nuanced evaluation reveals these hidden inefficiencies.

5.1.3 Mofka DAQ evaluation

For the rest of the TekApp experiments we define the Mofka overhead as the time spent on calls to `push` and `flush` in the producers and to `event.wait`, `event.metadata`, and `event.data` in the consumers.

DAQ generates 1,500 events of 20,548 B data and 30 B metadata. In this experiment we have varied the batch size of the producer from 1 to 64. By default, a Mofka producer maintains a maximum of two batches per partition: one is sent to the server by a background thread while the application fills up a second one. If the application produces too fast, a call to *push* may block as the second batch has filled up and the first batch has not completed its transfer yet. Tuning the batch size is therefore important to ensure best performance. A batch size that is too small would incur the overhead of sending more RPCs. A batch size that is too large delays the sending of events (and their processing by a consumer) and requires more memory.

Figure 7 shows that the overhead of Mofka decreases when increasing the batch size in DAQ; most of the overhead comes from the few blocking `push` calls that ensure that the batches are completely sent to Mofka. When the batch size is small, the number of blocking calls is higher; for instance, when the batch size is 1, we find a blocking call every 3 *push* calls. When we increase the batch size, not only is the number of blocking calls less frequent, but Mofka is more likely to complete the transfer of batches in the background before the third batch's event gets pushed.

5.1.4 Mofka DIST evaluation

The DIST component consumes the DAQ events and produces new ones for SIRT. A total of 1,500 events are received, each with 20,548 B data and 30 B metadata. Figure 8a shows the Mofka overhead while consuming events, again as a function of the batch size; error bars represent min and max values across six runs. The jump in overhead between a batch size of 4 and a batch size of 8 comes from a change in regime in the consumer. DIST consumes at a fast rate. With small batch sizes, batches are sent fast enough to be transferred in the background. The data selector and data broker also have time to execute in the background and fetch the required data before DIST needs to consume the event. Past 8 events per batch, however, DIST waits not only for the batches to be transferred but also for the selector and broker to execute. These costs become synchronous, causing the observed overhead.

DIST produces events with 10,240 KB data and 148 B metadata. Figure 8b shows the Mofka overhead while varying the batch size for different numbers of events and partitions; we keep the event/partition ratio fixed. DIST always runs on a single process; hence, this figure shows that the overhead is driven by the performance of a single partition and that parallelization across multiple partitions, even hosted by the same server node, is

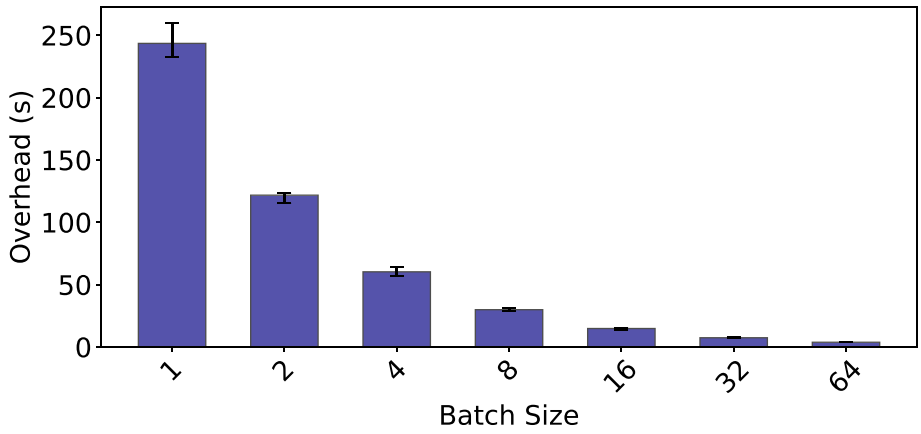

FIGURE 7
Mofka producer overhead in DAQ (median, min, and max from 6 runs), as a function of the batch size used by the producer. A total of 1,500 events are produced, each with 20,548 B data and 30 B metadata.

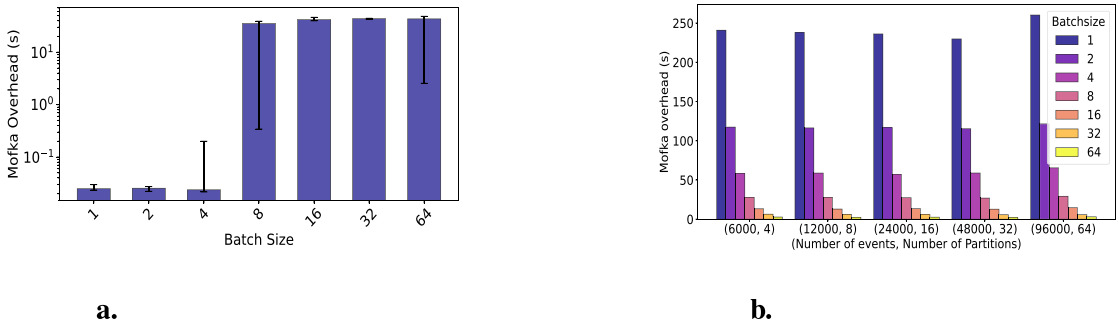

FIGURE 8
Mofka overhead in DIST for event production (left) and consumption (right). **(a)** DIST consumer overhead. **(b)** DIST producer overhead.

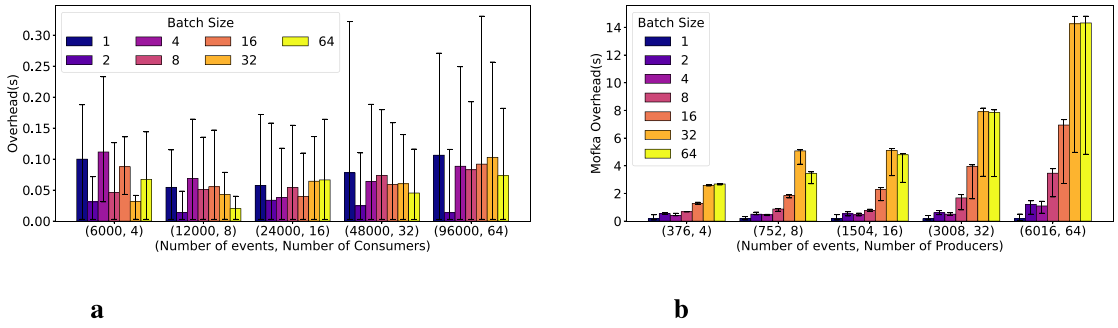

FIGURE 9
Mofka overhead for the SIRT component of the TekApp application for event production (left) and consumption (right), for varying numbers of partitions and batch sizes. **(a)** Mofka consumer overhead. **(b)** Mofka producer overhead.

effective. Note that Polaris nodes have 32 cores; and when we use 64 partitions (across 64 server processes hosted in a single node), we notice 7% higher overhead, likely due to oversubscription in the Mofka server.

5.1.5 Mofka SIRT evaluation

Figure 9a shows the Mofka consumer overhead while consuming the events in SIRT; median, min/max error bar values across all the processes are represented. The x-axis

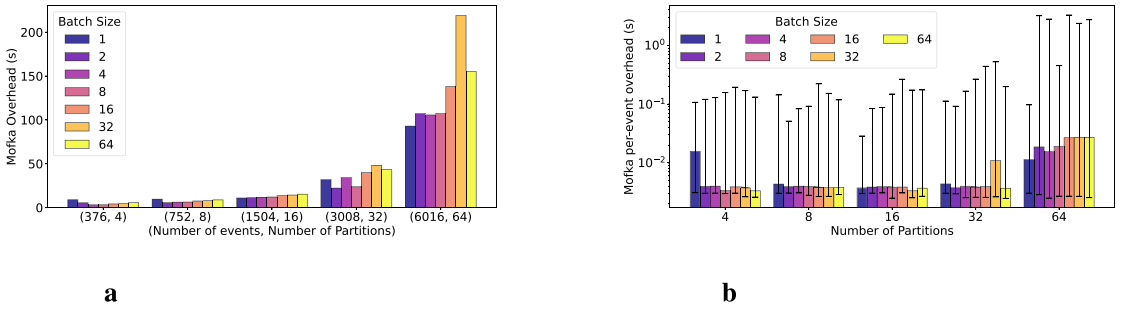

FIGURE 10
DEN consumption overall and per-event overhead while varying the number of partitions. **(a)** DEN consumption overhead. **(b)** DEN per-event overhead.

categories (E, C) represent the total number of events (E) and the total number of consumers (C). The event per consumer ratio is preserved. Every SIRT consumer pulls from a single partition. This overhead is very small and does not seem to be affected by the batch size. The reason is that the performance of SIRT is mostly driven by that of its computation and its data production, as shown hereafter. Hence, the consumer here has ample time to pull events in the background, regardless of the batch size.

Figure 9b shows the SIRT producer's overhead; median, min/max values across the processes are represented. The x-axis categories (E, P) represent the total number of events (E) and the total number of producers (P). The event per producer ratio is preserved. For each configuration, we vary the batch size from 1 to 64. Unlike the previous components, increasing the batch size increases the Mofka overhead, and this is visible in all configurations. This is mainly due to the size of the data the SIRT producers are sending, which is 25 MB per event. Such a data size per event removes any advantage of batching. Instead, batching delays sending the data to the server, causing a larger overhead. Overall, when the events are considerably large, as is the case for the SIRT component, it is recommended to use smaller batch sizes to avoid delaying transfers.

5.1.6 Mofka DEN evaluation

DEN consumes the data produced by the SIRT component. It is a single Python process, pulling data from 4 to 64 partitions. Figure 10a shows the overhead of pulling events from the servers. The x-axis represents the tested configurations (E, P), where E is the total number of events and P is the number of partitions. The ratio of event/partition is fixed. For each category, we vary the batch size from 1 to 64. Increasing the batch size does not significantly affect performance. The overhead increases as the total number of events increases. However, Figure 10b shows that the overhead per event remains the same until we use 64 partitions, at which point oversubscription in the server starts to affect performance.

5.1.7 Lessons learned from TekApp

Lesson 10: Batch size matters.

The batch size can have an important impact on throughput when producing and consuming events, especially when producing small events. However, a large batch size is not necessarily better, since it may delay production and consumption, increase latency, and prevent properly overlapping computation with data transfers.

Lesson 11: Don't batch large events.

Mofka separates the metadata and data part of an event so more efficient RDMA techniques can be used for the data part. When this data part is large, there is little benefit to batching more events together because the performance will be driven by the transfer of the data part, rather than the number of RPCs issued.

Lesson 12: Python binding has an overhead.

Some of the components in TekApp are written in Python. Not shown in our experiments above, but observed during our experiments using HPCToolkit, is the fact that Python can incur an important performance overhead. The conversion of a large number of Python objects to C++ and vice versa can often have more impact than the serialization of events or their transfer. Furthermore, Python's Global Interpreter Lock (GIL) makes it difficult to enable concurrency, especially in Mofka consumers, where Python-based data selectors and data brokers need to be invoked.

Lesson 13: More partitions equals more parallelism.

Increasing the number of partitions, even to produce (or consume) from a single process to a single server, allows for more parallelism and better performance.

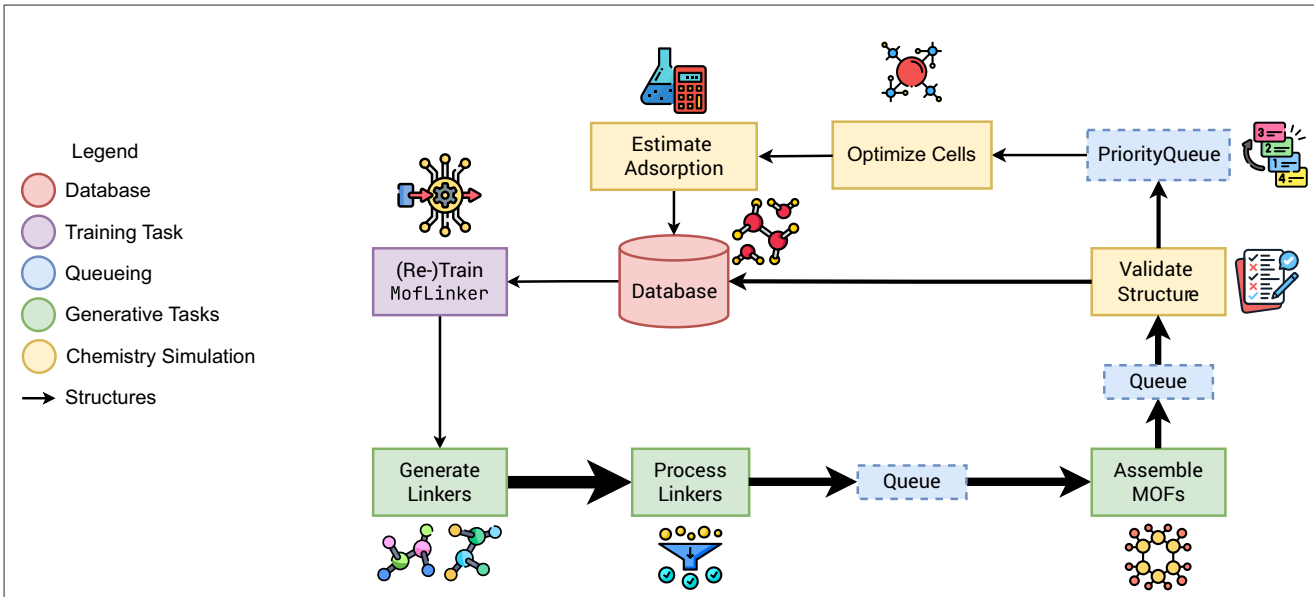

FIGURE 11
MOFA implements an online learning loop that refines a generative AI model using feedback from concurrently running physics-based evaluations. Validated linkers assemble MOFs, which are queued, screened for stability and gas capacity, and filtered to retain promising candidates. Resulting structures and properties are stored in a database for model retraining. Arrow widths indicate progressive filtering.

5.2 Use case 2: MOFA

MOFA (Yan et al., 2025), shown in Figure 11, is an AI driven discovery pipeline that explores the vast chemical design space of metal-organic frameworks (MOFs) for superior CO_2 capture materials. Designed to run on leadership HPC systems, the workflow links a diffusion-based generative model with a ladder of physics simulations in a closed online-learning loop that continually refines the generator with fresh simulation feedback.

The loop comprises seven asynchronous task types—generate, process, assemble, validate, optimize, estimate, and retrain—each backed by a pool of CPU and/or GPU workers:

1. **Generate linkers**: Propose candidate organic linkers.
2. **Process linkers**: Add hydrogen atoms, enforce net-zero charge and valid valence.
3. **Assemble MOFs**: Combine linkers with metal nodes, check bonds and atomic distances.
4. **Validate structure**: Check geometry and bonds, test stability and porosity.
5. **Optimize cells**: Perform DFT-level cell relaxation; compute partial charges.
6. **Estimate adsorption**: Simulate CO_2 absorption at 0.1 bar pressure and 300 K.
7. **Retrain**: Fine-tune the diffusion model with newly screened MOFs.

These heterogeneous tasks are orchestrated with Colmena (Ward et al., 2021, 2025), a Python framework designed for managing large simulation ensembles. Task execution is handled by Parsl worker pools (Task Servers), while the orchestration logic is centralized in a single Colmena Thinker process. Within the Thinker, each task type is managed by a lightweight agent thread that monitors available Task Server resources, submits task requests, and processes task results.

Communication between agents and their associated Task Servers occurs through Colmena Queues, implemented as Redis lists. Each agent–Task Server pair shares a dedicated queue for returning task results, while all agents share a common queue for submitting new task requests, with task types distinguished within the messages themselves.

Events flowing through Colmena queues are JSON objects of varying sizes. Many control and metadata events are smaller than 1 MB—well within efficient network limits—while larger stages such as MOF assembly can involve inputs of 10–40 MB and outputs of 1–2 MB. Intermediate tasks, such as linker processing and structure validation, typically produce messages between 100 KB and 600 KB.

Task durations also vary widely across the workflow. Fast tasks, such as linker generation, processing, and assembly, generally complete within 20 seconds. Structure validation, which involves preliminary molecular dynamics checks, averages around 4 min per structure. More computationally intensive steps, such as cell optimization and CO_2 adsorption estimation, take significantly longer—each requiring approximately 20–30 min to complete. Retraining the generative model depends on the size of the updated training set and typically takes between 30 and 300 seconds per cycle.

5.2.1 How Mofka transforms MOFA

In the original MOFA implementation, all task coordination and data exchange were handled through a single Redis instance co-located with the Colmena Thinker process on the login node. This Redis-backed queuing model introduced two critical limitations: it created a single point of failure, where a crash or stall in Redis could

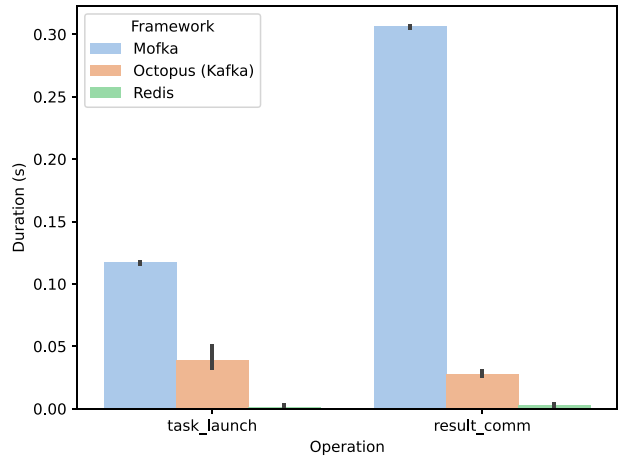

FIGURE 12
Time to communicate between producer and consumer for various operations in MOFA. `task_launch` represents the time to communicate the task to launch, and `result_comm` represents the time in takes to communicate candidate MOFs.

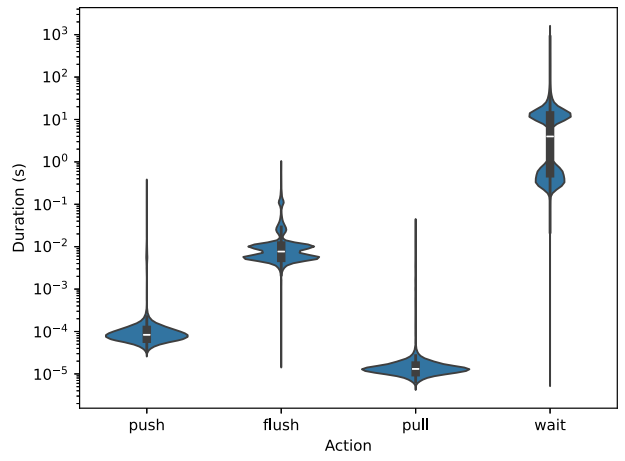

FIGURE 13
Times to perform various operations in MOFA using Mofka.

leave the workflow in an unrecoverable state, and it constrained throughput when event rates are high, as all data and control traffic were funneled through a single login node.

To address these challenges, we replaced Redis-based Colmena queues with Mofka-backed queues, leveraging Mofka's distributed, persistent event-streaming architecture. As in the original setup, each task type publishes results to a dedicated queue, now mapped to a Mofka topic, while task requests are consolidated into a shared request topic. Unlike Redis, however, Mofka topics are distributed across a cluster of brokers, allowing the workflow to scale more effectively and recover gracefully from failures.

5.2.2 MOFA experiments: generative task communication

MOFA generative tasks produce candidate MOFs at a high frequency, making it a good candidate for streaming frameworks that can rapidly communicate events. For this experiment we compare the rate at which we are able to create and communicate the generative task to run (`task_launch`) and the time to communicate the candidate MOFs from the Task Server back to the Thinker (`result_comm`) across the default approach (using a Redis queue) with a cloud-hosted Kafka solution (Octopus). To perform these experiments, a producer and consumer were initialized within the Thinker on a Polaris login node to produce the next generative tasks and consume its result. For the execution of the generative tasks, a consumer and producer were initialized on a Polaris compute node to receive the task to be executed and communicate results back to the Thinker. As the messages communicated were quite small in size (<1 MB), the execution was entirely latency bound. Timestamps were collected at each communication step to obtain information on latency overheads.

Contrary to other experiments, where we measured producer or consumer throughput, here we measure the latency between production and consumption. We do so by adding a timestamp when a message is produced, and we compare it with the current time when it is consumed. Figure 12 shows that Mofka exhibits a significant latency when communicating data, especially when compared with the default Redis and Octopus. There are a number of reasons for this latency. First, Mofka has not yet been optimized for low latency, preferring large batches and persistence first, as opposed to quickly getting a message to a consumer. Second, MOFA relies on Parsl, which mixes Python multithreading and multiprocessing. This forces the Mofka client to be set up without a background progress thread. Consequently, the network progress loop executes only when a Mofka API is being called, reducing opportunities for Mofka to overlap computation and communications. We are looking into solving this problem in the future, first by leveraging newer versions of Python where the GIL can be disabled, second by setting a side-process to handle communication when threading isn't an option.

Looking at the execution times for operations in Mofka (Figure 13) shows that the individual operations times are reasonable in Mofka, with the greatest overhead being in the `wait` operation. Since the progress thread is disabled, communication does not occur until `wait` is called, exacerbating wait durations.

5.2.3 Lessons learned from MOFA

Lesson 14: Synchronous execution leads to significant bottlenecks.

While synchronous executions are not recommended, they can sometimes not be avoided because of Python application multithreading/multiprocessing. Although there is currently no solution to address this limitation, solutions are being pursued.

5.3 Use case 3: Flowcept

Performance characterization and provenance of HPC workflows can be challenging, especially when dealing with hybrid and/or heterogeneous workloads. For example, large language model (LLM) training is a growing HPC workflow

that often utilizes distributed environments and stresses diverse system components (e.g., GPU, CPU, network fabric, PFS). With standard profiling tools, capturing the system performance and data provenance of such multi-GPU, multinode workloads remains challenging. Existing profiling tools often suffer from high overhead; and if multiple tools are used, correlating the performance of each individual component is difficult.

Flowcept (Souza et al., 2023) aims to address this challenge by capturing profiling data from massively parallel workloads and storing the results as a unified dataset. Flowcept currently provides adapters for Dask, MLFlow, and PyTorch, enabling multilevel data collection (e.g., task-level, epoch-level, layer-level, operation-level) for ML workflows. These features enable Flowcept to generate highly detailed profiling data from distributed HPC workloads including GPU power and utilization, CPU utilization, per-core CPU utilization, per-process CPU and memory utilization, disk and memory usage, network usage, tensor inspection, and task-level data. To reduce overhead during profiling, Flowcept streams provenance and profiling data to message queue servers for later processing. At runtime, the data is held in Flowcept buffers before being passed to the message queue producer, which streams the data to the server. Currently, Flowcept supports Redis and Kafka as message queue backends, and we extended it to support Mofka.

5.3.1 Flowcept experiments

We test each message queue using a multinode, multi-GPU LLM hyperparameter search workflow. During execution, Flowcept collects profiling data, which we refer to as a profiling unit, and bundles collections of profiling units into a batch of a user-defined size. Each batch is then passed to the message queue as a single message queue event, and the message queue flushes the event. Thus, each event (or message) varies in size depending on internal Flowcept settings that control what data to collect and the batch size.

We run tests with 24 concurrent producers, each corresponding to a single LLM hyperparameter configuration in a model-parallelism optimization process. To analyze consumer behavior independently from the producers, we launch a consumer after the workflow completes. However, Flowcept does not currently support a mechanism to allow Redis to store profiling data for later consumption (i.e., if the consumer is not subscribed when the data is published to a given topic, it is lost). Hence, we omit consumer analysis from this use case.

5.3.2 Evaluation setup

To perform LLM hyperparameter tuning, we employ multinode, multi-GPU Dask for model parallelism. We search 24 total hyperparameter configurations, with each Dask worker mapped to an individual GPU and a single hyperparameter configuration. Testing is carried out on the Polaris supercomputer using 6 compute nodes with 4 GPUs each, which enables all hyperparameter configurations to be evaluated concurrently. Additionally, we place the message service in its own compute node to isolate its effects and use one additional compute node for the Dask client script, totaling 8 compute nodes. To explore each message queue's performance in different profiling scenarios, we perform two styles of profiling:

1. Lightweight: this configuration produces small events that impose less pressure on the message queue's bandwidth but require low latency. A non-exhaustive list of the metrics collected is as follows:
   - Number of dask workers.
   - Task start and end time.
   - Input data paths.
   - Batch size.
   - Workflow epochs.
   - Number of hidden layers and their dimensions.
   - Number of tokens.
   - Dropout rate.
2. Heavyweight: this configuration produces larger events that place more pressure on the message queue's bandwidth and decrease the importance of network latency. The following is a non-exhaustive list of the metrics collected in addition to the data collected with the lightweight configuration:
   - CPU power, memory, and utilization.
   - Per-core CPU power, memory, and utilization.
   - GPU temperature, and utilization.
   - Per-layer dropout rate, embedding dimensions, and number of attention heads.
   - I/O counters (e.g., read bytes, write bytes).
   - Tensor dimensions and memory usage.

We employ the following batch sizes to test the MQs under different levels of stress: 10^1, 10^2, 10^3, and 10^4. As the batch size decreases, the message queue is forced to flush more frequently and experiences more computational load. Given that we do not observe any message queue interaction during workflow execution when using a batch size of 10^5, we limit profiling to a batch size of 10^4 or less (this is the result of the batch size exceeding the number of profiling units each worker produces, avoiding flushes until the worker closes).

Table 4 shows more workload details about lightweight and heavyweight profiling with a batch size of 10. As expected, heavyweight profiling generates more messages and a larger total amount of data and displays a higher median message size. These trends hold as the batch size grows and provide two different use cases to test the backend message queues. We record the overall workflow time and the hyperparameter search duration, which corresponds to the producer's runtime. Each profiling run

TABLE 4 Workload details about each style of profiling with a batch size of 10. Numbers are averaged across all MQs.

Profiling setting	Mean number of messages	Mean data transmitted	Median message size
Light-weight	54,452	334 MB	6 KB
Heavy-weight	162,636	26 GB	169 KB

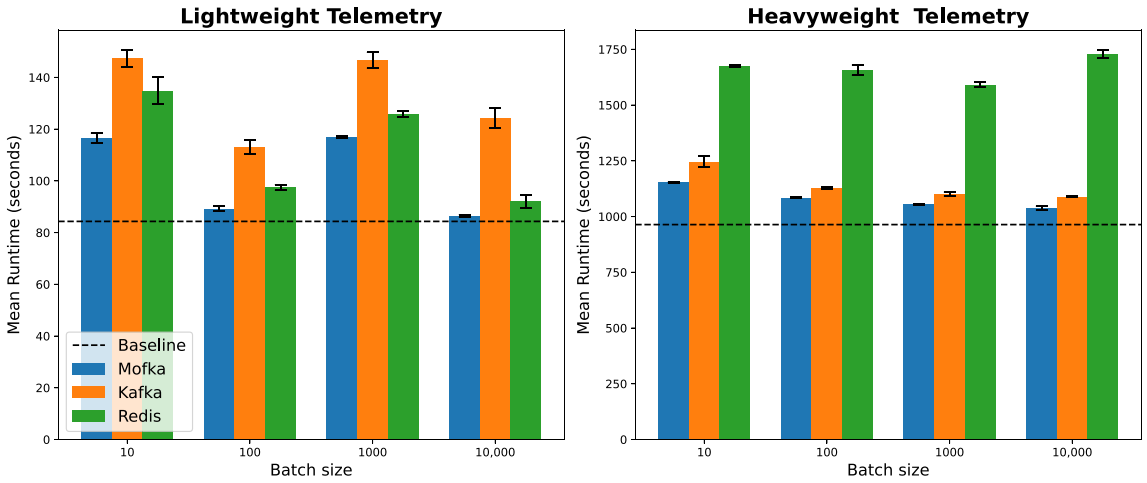

FIGURE 14
Flowcept workflow runtime by message queue with two different levels of profiling. The baseline corresponds to the runtime when there is no MQ interaction during execution.

TABLE 5 Percent of total workload time that each message queue spent flushing by batch size when performing lightweight and heavyweight profiling.

Implementation	Batch size			
	10^1	10^2	10^3	10^4
Mofka lightweight	0.35%	0.22%	0.26%	0.25%
Mofka heavyweight	1.33%	1.20%	0.95%	0.83%
Redis lightweight	18.58%	10.10%	9.39%	7.72%
Redis heavyweight	35.73%	36.96%	34.79%	27.84%
Kafka lightweight	26.09%	21.41%	19.39%	16.00%
Kafka heavyweight	9.86%	7.24%	6.79%	6.31%

and batch size are repeated 3 times to ensure that the results are representative.

5.3.3 Results on Polaris

Across both profiling levels and all four batch sizes, Mofka outperforms Redis and Kafka, generating less overhead and achieving up to a 1.66× and 1.44× speedup over Redis and Kafka, respectively (Figure 14). When performing lightweight profiling, Redis generates less overhead than Kafka; however, when performing heavyweight profiling, Kafka significantly outperforms Redis. These results suggest that Kafka is better suited for larger message sizes than Redis is but struggles with small streaming workloads (and conversely, Redis is well suited for small message sizes). Mofka outperforms in both contexts, taking advantage of its HPC optimizations. In particular, it is able to utilize the underlying CXI network protocol, while the other message queues utilize TCP-based transfer protocols. The performance gap between the two protocols is clearly highlighted by the amount of time each message queue spent performing blocking communication during execution; with a batch size of 1,000 while performing heavyweight profiling, Mofka blocked the workflow to flush a total of 7.30 seconds, while Kafka and Redis blocked for 36.19 and 265.81 seconds, respectively. During lightweight profiling, Mofka spends less time blocking for communication than does either of the other message queues; and, as highlighted previously, Redis outperforms Kafka with smaller message sizes, spending less time performing blocking communication (see Table 5 for a full comparison). The time spent on blocking communication directly correlates to performance; and since Mofka spent a significantly reduced percentage of overall workflow time blocking regardless of message size (see Table 5), Mofka reduces overhead in both cases. In summary, Mofka outperforms existing Flowcept message queue backends by fully utilizing the available HPC infrastructure, enabling efficient profiling and provenance.

5.3.4 Results on Frontier

We repeated a subset of our experiments on OLCF's Frontier. We utilized the same experimental settings as described above; however, each compute node has 8 GPUs rather than 4. Hence, only 3 compute nodes are used to evaluate the 24 hyperparameter configurations. Additionally, we limit testing to only the lightweight profiling workload. The results are shown in Figure 15.

The total runtime of the application is similar, whether we use Mofka or Redis as message broker. But contrary to Redis, Mofka provides persistence, allowing analysis of events to happen after the workflow is completed, rather than during its execution.

5.4 Summary of our experiments

Across the three use-cases and synthetic benchmark, Mofka was compared against ZeroMQ, Kafka, Redpanda, and Redis. This section briefly highlights the takeaways from these results.

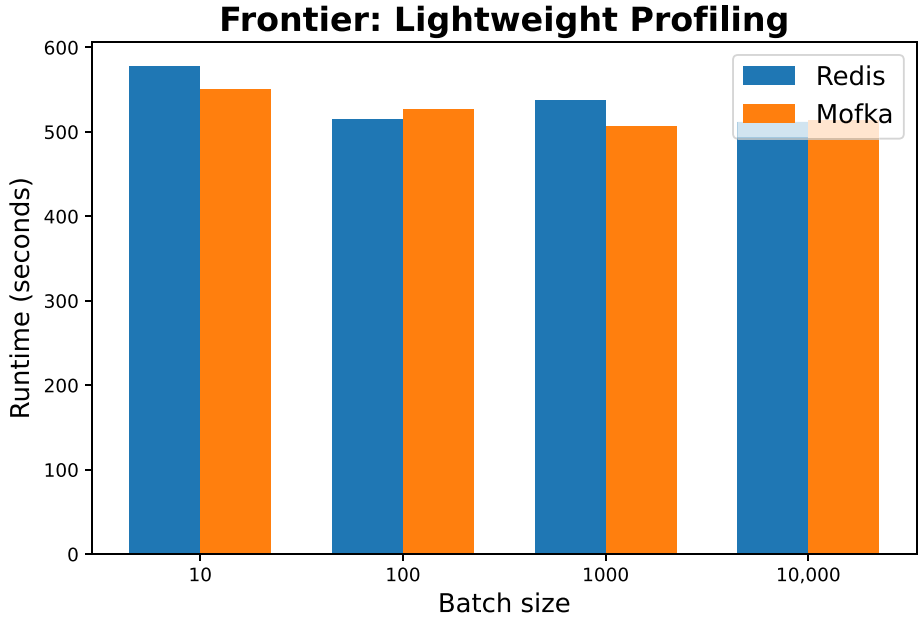

FIGURE 15
Flowcept workflow runtime by message queue on Frontier.

Mofka vs. ZeroMQ. Mofka was evaluated against ZeroMQ with TekApp. ZeroMQ does not provide persistence, forcing all components of the TekApp workflow to run at the same time. It also relies on TCP instead of taking advantage of high-performance networks. Our experiments comparing Mofka with ZeroMQ show that the former can achieve up to 500× better performance, while providing impedance matching and persistence.

Mofka vs. Kafka. Kafka was used in Section 4 with synthetic benchmarks, as well as in Sections 5.2 and 5.3 with MOFA and Flowcept respectively. In all our experiments, Kafka proved difficult to configure for best performance on HPC systems, experiencing frequent timeouts. Kafka is the inspiration for Mofka, but it does not support HPC networks, relying on TCP instead. Our synthetic benchmark showed that Mofka outperforms Kafka in many scenarios, achieving an 8× better throughput in some of them. In Flowcept, the use of Mofka incurred a lower overhead on the workflow than that of Kafka, with Mofka's overhead ranging from 0.22% to 1.33% of workflow runtime, against 6.31% to 26.09% for Kafka. In MOFA, Kafka (used in Octopus) outperformed Mofka. The overhead of Mofka was found to be caused by the limitations of Python when it comes to multithreading and multiprocessing, which force all the calls to Mofka to be blocking. This is an aspect we need to address in future versions of Mofka.

Mofka vs. Redpanda. Redpanda is a drop-in replacement for Kafka, answering to the same protocol. We could not fully take advantage of some of Redpanda's configuration parameters because they require privileges we do not have on HPC compute nodes. In benchmarks, Mofka outperformed Redpanda in most cases, just as it did Kafka.

Mofka vs. Redis. Redis was used in Flowcept and compared against Kafka and Mofka. The overhead of Redis on the workflow runtime ranged from 7.72% to 36.96%, significantly higher than that of Mofka across all tested configurations. However, Redis was also used in MOFA, where it outperformed both Mofka and Kafka by a large margin. Redis is an in-memory cache; hence, while it provides impedance matching and component decoupling, it does not provide persistence. Along with providing a better Python interface than Mofka, this lack of persistence may also explain its performance in the context of MOFA.

These results highlight the potential for an event-driven system tailored to HPC systems, such as Mofka.

6 Related work

Too many distributed event and data-streaming platforms exist to be listed exhaustively in this section. Hence we focus on three aspects. First, we put our work in perspective with other streaming systems specifically designed for HPC (whether they focus on computational streams or act as message brokers). Second, we reference other data management services in the HPC space that address producer/consumer dataflows. Third, we list three other streaming systems designed outside of HPC that could be used, or have been used, in an HPC context.

6.1 Streaming for HPC

SLoG (Matri et al., 2018) proposes streaming for HPC, with Kafka and CORFU (Balakrishnan et al., 2013) as inspiration. SLoG stores its logs directly in a parallel file system, unlike Mofka, which stores them in local SSDs. SloG uses proxy nodes to increase concurrency; in other words, rather than having all the nodes of an application read from a partition, proxy nodes read from them and redistribute the data across a larger set of nodes. To the best of our knowledge, SLog relies on TCP for communications, rather than on

native HPC network libraries. Its concept of proxy could easily be adapted to Mofka, especially thanks to Mofka's composable design.

Neon (Matri and Ross, 2021) was a first take on streaming for HPC in the context of the Mochi framework. While not persistent and not distributed, it provides a way for users to set up data streams between HPC and edge applications, and it has proved twice as fast as Apache Storm thanks to its use of HPC networks.

Chronolog (Kougkas et al., 2020) was proposed to handle logs from very large HPC applications, emphasizing the need to maintain time-based ordering of events, something that other messaging systems (including Mofka) do not guarantee. Chronolog relies on a distributed, multitiered key-value store to store events, with the event's timestamp used as the key. This design is radically different from the append-only partition-based design of the likes of Kafka and Mofka. While it can be suitable for some HPC use cases such as provenance tracking, it would not be suitable for other use cases attaching large amounts of data to events.

HStream (Cernuda et al., 2024) is another recent take on the HPC streaming challenge, with a focus on computations in streams. Its contributions include an adaptive parallelism controller that dynamically adjusts computational resources to match the variable I/O demands of scientific workloads and a hierarchical memory manager that leverages local and remote storage to alleviate memory pressure. HStream was implemented using Thallium, one of the components of the Mochi framework, and was evaluated against Neon.

Kafka is used in production at ORNL via STREAM (Adamson et al., 2023), a platform designed to collect telemetry from multiple of OLCF's machines. With such telemetry consisting of small events, and given that each machine has its own high-performance network, using Kafka to leverage high-speed Ethernet LAN in the facility makes sense and shows that systems designed for a cloud environment can be relevant in an HPC facility.

Pilot-Streaming (Chantzialexiou et al., 2018) provides a unified framework for deploying and managing streaming applications on HPC infrastructure. By leveraging technologies such as Kafka for message brokering, Spark Streaming and Dask for data processing, and the Pilot-Abstraction for dynamic resource management, Pilot-Streaming simplifies the integration of streaming workflows into HPC environments. This framework exemplifies the necessity of streaming in HPC, enabling timely data analysis and dynamic experiment steering, which are essential for modern scientific research.

6.2 Producer/consumer HPC services

As stated in Section 1, HPC applications are fundamentally producers and consumers of data, rather than modifiers. Scientific simulations produce periodic outputs that must be consumed by analysis and visualization tools. Scientific instruments are producers of sensor data that must be sent to post-processing tools on supercomputers. HPC workflows comprise tasks that produce and consume data. If a file needs to be *modified*, it is generally to update a header or metadata related to its structure, not to touch its scientific content. Hence, most HPC data services can be thought of as addressing this producer/consumer pattern.

ADIOS (Godoy et al., 2020), initially developed as an I/O library with modular backend plugins, provides the SST plugin for streaming (Eisenhauer et al., 2024). SST is not a data service, in the sense that it does not require extra servers running a specific software. Rather, it turns the producing application into a data service. `Put` operations queue application data locally, making it remotely available to the consumer application's `Get` operations via RDMA. SST does not provide persistence. However, it illustrates the need for efficient streaming in HPC applications.

Checkpointing systems are also frequently used together with producer/consumer patterns. They facilitate a decoupled interaction between producers and consumers by enabling the capture and reuse of key data structures. Multilevel checkpointing solutions such as VeloC (Nicolae et al., 2021, 2019) leverage both node-local (GPU memory, host memory, NVMe) and remote storage to cache and persist checkpoints asynchronously, which reduces the I/O overheads significantly by masking them in the background. These techniques can be combined with prefetching to co-optimize checkpointing initiated by the producers with the reuse patterns of the consumers, which can be especially effective if the consumers express their intent (hints about what checkpoints to read in what order) in advance (Maurya et al., 2023). The convenience of working with decoupled checkpoint files also can be leveraged at the system level by intercepting I/O calls and transparently implementing them as direct links between producers and consumers. This was demonstrated by LowFive (Peterka et al., 2023) for the popular HDF5 (Folk et al., 2011) format, with notable applications for AI workflows that need to continuously train learning models (producers) that need to be used at the same time for inferences (consumers) (Ye et al., 2024).

Services that may provide persistence include all the implementations of the data staging concept, one of which is DataSpaces (Docan et al., 2010). Such services rely on dedicated nodes that act as a buffer between a producing application (generally a simulation) and a consuming application (generally an analysis code). They may only provide data production and consumption capabilities or may enable in-service computation, as in Colza (Dorier et al., 2022), which provides *in situ* visualization capabilities. More generally, the entire field of *in situ* analysis and visualization, a good overview of which is provided by Childs et al. (2020), addresses the problem of coupling a data producer to a data consumer while avoiding the overhead of a parallel file systems. Many of the solutions in this space include staging with various degrees of persistence, and this area of HPC would benefit from a proper distributed event-driven service.

Aside from scientific data, metadata about the execution of workflows, including provenance data (as in Flowcept) or anomaly detection data, and performance monitoring data, can benefit from a streaming service. Chimbuko (Kelly et al., 2020) is an example of scalable performance trace analysis tool, focusing on anomaly detection from performance data captured using TAU. Its provenance database is built using Mochi and relies an append-only document storage component reminiscent of persistent append-only logs found in streaming services.

SciStream (Chung et al., 2022) was proposed to enable streaming data from scientific instruments to supercomputers or across supercomputers in a WAN. It uses Science DMZ gateway nodes to cross networks. Since WAN requires Ethernet-based

communications, it cannot take advantage of high-performance networks inside a supercomputer. SciStream also provides data persistence. In fact, its purpose is to avoid the traditional way of sharing data via parallel file systems.

All these producer/consumer services for HPC heavily rely on HPC technologies such as RDMA and high-performance networks, justifying the use of these technologies when developing an event-streaming service for HPC.

6.3 Other streaming services

Originally meant as a message queuing system, Apache Pulsar (Sharma and Atyab, 2021) is an alternative to Kafka that later added event-streaming capabilities. Contrary to the monolithic design of Kafka, Pulsar decouples message queueing from persistence, relying on Apache BookKeeper for the latter and allowing multitiered storage. Pulsar is, however, no more adapted to HPC than Kafka is. Confluent reports Kafka to be twice as fast as Pulsar and to provide lower latency,[18] although these results are debatable and may depend on use cases (Andström, 2024).

KerA (Marcu et al., 2018) is similar to Kafka but proposes a hierarchical partitioning of the data, splitting topics into a static number of *streamlet* corresponding to semantic partitioning, each split into a dynamic number of groups. KerA was built on RAMCloud (Ousterhout et al., 2015), which supports InfiniBand, and hence could be used in an HPC context. However, neither KerA nor RAMCloud is currently maintained.

DataStates (Nicolae, 2020) is a data model in which users do not interact with a data service directly to write datasets but rather tag datasets with properties expressing hints, constraints, and persistency semantics, which automatically adds snapshots (called data states) into the lineage, a history recording the evolution of all snapshots, using an optimal I/O plan. Later, these data states can be discovered and revisited based on the lineage. In effect, they can act as a stream between producers and consumers and can be pruned as needed, while retaining persistency for important data. This principle can be applied even for basic data structures, such as ordered key-value stores (Nicolae, 2022).

7 Conclusion

Event-driven systems have been massively adopted in internet businesses to provide a single source of truth while decoupling applications. In HPC, such a model could be adopted just as well to support workflows made of applications that produce and consume data.

In this paper we investigated the potential for an event-driven framework tailored to HPC systems by exemplifying three such workflows. While exceptions exist (e.g., AI-training applications requiring random accesses to a large number of sample data), many HPC applications can be thought of as a producer/consumer workflow. Traditional, bulk-synchronous scientific simulations are periodic producers of checkpoints, for which postprocessing tools are their consumers. With the emergence of more complex workflows, a distributed, persistent, event-driven framework for HPC becomes increasingly relevant, enabling decoupling workflow components and impedence matching between production and consumption rates.

Our Mofka framework, based on the Mochi suite of components for HPC data services, makes use of HPC technologies such as high-speed networks with RDMA and efficient multithreading to efficiently support HPC applications. In this study we showed that (1) there is an advantage to relying on such HPC-specific technology to design an event-driven system for HPC, rather than relying on technologies from the cloud computing and internet landscape, as motivated by performance evaluation comparing Mofka with Kafka and Redpanda, and (2) there is an actual need for such an event-driven system in HPC, as exemplified by three workflows in which Mofka is now used.

As future work, we plan to make Mofka resilient by enabling data replication as done in Kafka and Redpanda. We will do so using a custom, RDMA-enabled implementation of the RAFT protocol. We also plan to turn Mofka into a multiuser service by enabling authentication and access control, using MUNGE,[19] OpenSSL,[20] and a methodology we recently released for developing multiuser Mochi-based services.[21]

We also plan to explore bridging Mofka with other technologies such as Octopus (Pan et al., 2024) to make it part of a larger, scientific event fabric. In this context Mofka would form the HPC part of such an event fabric spanning scientific instruments, HPC facilities, edge devices, and cloud computing.

Data availability statement

The original contributions presented in the study are included in the article/supplementary material, further inquiries can be directed to the corresponding author.

Author contributions

MD: Visualization, Writing – review & editing, Supervision, Methodology, Writing – original draft, Conceptualization, Investigation, Software. AG: Visualization, Writing – review & editing, Software, Supervision, Writing – original draft, Investigation. VH-S: Writing – original draft, Writing – review & editing, Investigation, Supervision, Visualization, Software. HN: Writing – review & editing, Writing – original draft, Visualization, Software, Investigation. SO: Writing – original draft, Visualization, Software, Writing – review & editing, Investigation. RS: Writing – original draft, Investigation, Software, Visualization, Writing – review & editing. TB: Writing – original draft, Supervision, Software, Writing – review & editing. HP: Writing – original draft, Investigation, Writing – review & editing, Visualization, Software. PC: Writing – review & editing, Software, Conceptualization, Writing – original draft. KC: Writing – review & editing, Writing – original draft. RC: Writing – review & editing, Writing – original draft. MG: Writing – review & editing, Writing – original draft. EH: Writing – review & editing, Writing – original draft. BL:

18 https://www.confluent.io/kafka-vs-pulsar/

19 https://dun.github.io/munge/

20 https://www.openssl.org/

21 https://github.com/mochi-hpc/mochi-auth-examples

Writing – review & editing, Writing – original draft. BN: Writing – review & editing, Writing – original draft. PP: Software, Writing – review & editing, Writing – original draft. JW: Writing – original draft, Writing – review & editing. IF: Writing – review & editing, Funding acquisition, Writing – original draft. NR: Writing – review & editing, Writing – original draft, Funding acquisition. RR: Writing – review & editing, Writing – original draft.

Funding

The author(s) declare that financial support was received for the research and/or publication of this article. This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under contract number DE-AC02-06CH11357. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided by the Laboratory Computing Resource Center at Argonne National Laboratory. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Acknowledgments

The authors thank Gail Pieper for proofreading and editing this manuscript.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

The author(s) declared that they were an editorial board member of Frontiers, at the time of submission. This had no impact on the peer review process and the final decision.

Generative AI statement

The author(s) declare that no Gen AI was used in the creation of this manuscript.

Any alternative text (alt text) provided alongside figures in this article has been generated by Frontiers with the support of artificial intelligence and reasonable efforts have been made to ensure accuracy, including review by the authors wherever possible. If you identify any issues, please contact us.

Publisher's note

References

Adamson, R., Osborne, T., Lester, C., and Palumbo, R. (2023). *STREAM: a scalable federated HPC telemetry platform*. Technical report, Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States).

Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., et al. (2010). HPCToolkit: tools for performance analysis of optimized parallel programs. *Concurr. Comput.* 22, 685–701. doi: 10.1002/cpe.1553

Al-Najjar, A., Rao, N. S., Sankaran, R., Ziatdinov, M., Mukherjee, D., Ovchinnikova, O., et al. (2022). "Enabling autonomous electron microscopy for networked computation and steering," in *IEEE 18th International Conference on e-Science (e-Science)* (IEEE), 267–277. doi: 10.1109/eScience55777.2022.00040

Andström, V. (2024). *A comparative analysis of Apache Kafka and Apache Pulsar*. PhD thesis, Master's thesis, University of Helsinki, Faculty of Science.

Balakrishnan, M., Malkhi, D., Davis, J. D., Prabhakaran, V., Wei, M., and Wobber, T. (2013). CORFU: a distributed shared log. *ACM Trans. Comput. Syst.* 31, 1–24. doi: 10.1145/2535930

Batenburg, K. J., Bals, S., Sijbers, J., Kübel, C., Midgley, P. A., Hernandez, J., et al. (2009). 3D imaging of nanomaterials by discrete tomography. *Ultramicroscopy* 109, 730–740. doi: 10.1016/j.ultramic.2009.01.009

Bautista-Gomez, L., Tsuboi, S., Komatitsch, D., Cappello, F., Maruyama, N., and Matsuoka, S. (2011). "FTI: high performance fault tolerance interface for hybrid systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11* (New York, NY, USA: Association for Computing Machinery). doi: 10.1145/2063384.2063427

Bicer, T. (2024). *Streaming tomography mini-app*. Available online at: https://github.com/diaspora-project/aps-mini-apps

Bicer, T., Gürsoy, D., Andrade, V. D., Kettimuthu, R., Scullin, W., Carlo, F. D., et al. (2017a). Trace: a high-throughput tomographic reconstruction engine for large-scale datasets. *Adv. Struct. Chem. Imag.* 3, 1–10. doi: 10.1186/s40679-017-0040-7

Bicer, T., Gursoy, D., Kettimuthu, R., Foster, I. T., Ren, B., De Andrede, V., et al. (2017b). "Real-time data analysis and autonomous steering of synchrotron light source experiments," in *13th International Conference on e-Science* (IEEE), 59–68. doi: 10.1109/eScience.2017.53

Bicer, T., Nikitin, V., Aslan, S., Gürsoy, D., Kettimuthu, R., and Foster, I. T. (2020). "Tomographic reconstruction of dynamic features with streaming sliding subsets," in *IEEE/ACM 2nd Annual Workshop on Extreme-scale Experiment-in-the-Loop Computing (XLOOP)* (IEEE), 8–15. doi: 10.1109/XLOOP51963.2020.00007

Brim, M. J., Moody, A. T., Lim, S.-H., Miller, R., Boehm, S., Stanavige, C., et al. (2023). "UnifyFS: a user-level shared file system for unified access to distributed local storage," in *International Parallel and Distributed Processing Symposium* (IEEE), 290–300. doi: 10.1109/IPDPS54959.2023.00037

Cernuda, J., Ye, J., Kougkas, A., and Sun, X.-H. (2024). "HStream: a hierarchical data streaming engine for high-throughput scientific applications," in *53rd International Conference on Parallel Processing*, 231–240. doi: 10.1145/3673038.3673150

Chantzialexiou, G., Luckow, A., and Jha, S. (2018). "Pilot-streaming: a stream processing framework for high-performance computing," in *14th International Conference on e-Science* (IEEE), 177–188. doi: 10.1109/eScience.2018.00033

Chard, K., Tuecke, S., and Foster, I. (2016). "Globus: recent enhancements and future plans," in *XSEDE16*, 1–8. doi: 10.1145/2949550.2949554

Chard, R., Pruyne, J., McKee, K., Bryan, J., Raumann, B., Ananthakrishnan, R., et al. (2023). "Globus automation services: Research process automation

across the space-time continuum. *Fut. Gener. Comput. Syst.* 142, 393–409. doi: 10.1016/j.future.2023.01.010

Childs, H., Ahern, S. D., Ahrens, J., Bauer, A. C., Bennett, J., Bethel, E. W., et al. (2020). A terminology for *in situ* visualization and analysis systems. *Int. J. High Perform. Comput. Appl.* 34, 676–691. doi: 10.1177/1094342020935991

Chung, J., Zacherek, W., Wisniewski, A., Liu, Z., Bicer, T., Kettimuthu, R., et al. (2022). "SciStream: architecture and toolkit for data streaming between federated science instruments," in *31st International Symposium on High-Performance Parallel and Distributed Computing*, 185–198. doi: 10.1145/3502181.3531475

Corbett, P., Feitelson, D., Fineberg, S., Hsu, Y., Nitzberg, B., Prost, J.-P., et al. (1996). "Overview of the MPI-IO parallel I/O interface," in *Input/Output in Parallel and Distributed Computer Systems* (Boston, MA: Springer US), 127–146. doi: 10.1007/978-1-4613-1401-1_5

Docan, C., Parashar, M., and Klasky, S. (2010). "DataSpaces: an interaction and coordination framework for coupled simulation workflows," in *19th ACM International Symposium on High Performance Distributed Computing*, 25–36. doi: 10.1145/1851476.1851481

Donovan, S., Huizenga, G., Hutton, A. J., Ross, C. C., Petersen, M. K., and Schwan, P. (2003). "Lustre: building a file system for 1000-node clusters," in *Proceedings of the 2003 Linux Symposium*, 380–386.

Dorier, M., Wang, Z., Ayachit, U., Snyder, S., Ross, R., and Parashar, M. (2022). "Colza: enabling elastic *in situ* visualization for high-performance computing simulations," in *International Parallel and Distributed Processing Symposium* (IEEE), 538–548. doi: 10.1109/IPDPS53621.2022.00059

Eisenhauer, G., Podhorszki, N., Gainaru, A., Klasky, S., Davis, P. E., Parashar, M., et al. (2024). Streaming data in HPC workflows using ADIOS. *arXiv preprint arXiv:2410.00178*.

Folk, M., Heber, G., Koziol, Q., Pourmal, E., and Robinson, D. (2011). "An overview of the HDF5 technology suite and its applications," in *AD '11: Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases* (Uppsala, Sweden), 36–47. doi: 10.1145/1966895.1966900

Godoy, W. F., Podhorszki, N., Wang, R., Atkins, C., Eisenhauer, G., Gu, J., et al. (2020). ADIOS 2: The adaptable input output system. A framework for high-performance data management. *SoftwareX* 12:100561. doi: 10.1016/j.softx.2020.100561

Hennecke, M. (2020). "Daos: a scale-out high performance storage stack for storage class memory," in *Supercomputing Frontiers*, 40. doi: 10.1007/978-3-030-48842-0_3

Javed, M. H., Lu, X., and Panda, D. K. (2017). "Characterization of big data stream processing pipeline: a case study using Flink and Kafka," in *4th IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*, 1–10. doi: 10.1145/3148055.3148068

Kelly, C., Ha, S., Huck, K., Van Dam, H., Pouchard, L., Matyasfalvi, G., et al. (2020). "Chimbuko: a workflow-level scalable performance trace analysis tool," in *ISAV'20 In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization* (ACM), 15–19. doi: 10.1145/3426462.3426465

Kougkas, A., Devarajan, H., Bateman, K., Cernuda, J., Rajesh, N., and Sun, X.-H. (2020). "Chronolog: a distributed shared tiered log store with time-based data ordering," in *36th International Conference on Massive Storage Systems and Technology*, 80.

Li, J., Liao, W.-k., Choudhary, A., Ross, R., Thakur, R., Gropp, W., et al. (2003). "Parallel netCDF: A high-performance scientific I/O interface," in *ACM/IEEE Conference on Supercomputing*, 39. doi: 10.1145/1048935.1050189

Liu, Z., Bicer, T., Kettimuthu, R., and Foster, I. (2019). "Deep learning accelerated light source experiments," in *3rd Workshop on Deep Learning on Supercomputers* (IEEE), 20–28. doi: 10.1109/DLS49591.2019.00008

Liu, Z., Bicer, T., Kettimuthu, R., Gursoy, D., De Carlo, F., and Foster, I. (2020). TomoGAN: low-dose synchrotron x-ray tomography with generative adversarial networks. *J. Opt. Soc. Am. A* 37, 422–434. doi: 10.1364/JOSAA.375595

Lockwood, G. K., Hazen, D., Koziol, Q., Canon, R. S., Antypas, K., Balewski, J., et al. (2017). *Storage 2020: A vision for the future of HPC storage*. doi: 10.2172/1632124

Lu, X., Shankar, D., Gugnani, S., and Panda, D. K. (2016). "High-performance design of Apache Spark with RDMA and its benefits on various workloads," in *IEEE International Conference on Big Data* (IEEE), 253–262. doi: 10.1109/BigData.2016.7840611

Marcu, O.-C., Costan, A., Antoniu, G., Pérez-Hernández, M., Nicolae, B., Tudoran, R., et al. (2018). "Kera: scalable data ingestion for stream processing," in *38th International Conference on Distributed Computing Systems* (IEEE), 1480–1485. doi: 10.1109/ICDCS.2018.00152

Matri, P., Carns, P., Ross, R., Costan, A., Pérez, M. S., and Antoniu, G. (2018). "SLoG: large-scale logging middleware for HPC and big data convergence," in *38th International Conference on Distributed Computing Systems* (IEEE), 1507–1512. doi: 10.1109/ICDCS.2018.00156

Matri, P., and Ross, R. (2021). "Neon: low-latency streaming pipelines for HPC," in *14th International Conference on Cloud Computing* (IEEE), 698–707. doi: 10.1109/CLOUD53861.2021.00089

Maurya, A., Rafique, M., Tonellot, T., AlSalem, H., Cappello, F., and Nicolae, B. (2023). "GPU-enabled asynchronous multi-level checkpoint caching and prefetching," in *HPDC'23: The 32nd International Symposium on High-Performance Parallel and Distributed Computing* (Orlando, FL, USA), 73–85. doi: 10.1145/3588195.3592987

Nicolae, B. (2020). "DataStates: towards lightweight data models for deep learning," in *SMC'20: The 2020 Smoky Mountains Computational Sciences and Engineering Conference* (Nashville, United States), 117–129. doi: 10.1007/978-3-030-63393-6_8

Nicolae, B. (2022). "Scalable multi-versioning ordered key-value stores with persistent memory support," in *IPDPS 2022: The 36th IEEE International Parallel and Distributed Processing Symposium* (Lyon, France), 93–103. doi: 10.1109/IPDPS53621.2022.00018

Nicolae, B., Moody, A., Gonsiorowski, E., Mohror, K., and Cappello, F. (2019). "VeloC: towards high performance adaptive asynchronous checkpointing at large scale," in *IPDPS'19: The 2019 IEEE International Parallel and Distributed Processing Symposium* (Rio de Janeiro, Brazil), 911–920. doi: 10.1109/IPDPS.2019.00099

Nicolae, B., Moody, A., Kosinovsky, G., Mohror, K., and Cappello, F. (2021). "VeloC: very low overhead checkpointing in the age of exascale," in *SuperCheck'21: The First International Symposium on Checkpointing for Supercomputing, Virtual Event*.

Ousterhout, J., Gopalan, A., Gupta, A., Kejriwal, A., Lee, C., Montazeri, B., et al. (2015). The RAMCloud storage system. *ACM Trans. Comput. Syst.* 33, 1–55. doi: 10.1145/2806887

Pan, H., Chard, R., Zhou, S., Kamatar, A., Vescovi, R., Hayot-Sasson, V., et al. (2024). "Octopus: experiences with a hybrid event-driven architecture for distributed scientific computing," in *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis* (IEEE), 496–507. doi: 10.1109/SCW63240.2024.00071

Peterka, T., Morozov, D., Nigmetov, A., Yildiz, O., Nicolae, B., and Davis, P. E. (2023). "LowFive: *in situ* data transport for high-performance workflows," in *IPDPS'23: The 37th IEEE International Parallel and Distributed Processing Symposium* (St. Petersburg, FL, USA), 985–995. doi: 10.1109/IPDPS54959.2023.00102

Ramesh, S., Childs, H., and Malony, A. (2022). "Serviz: a shared *in situ* visualization service," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis* (IEEE), 1–14. doi: 10.1109/SC41404.2022.00026

Ross, R. B., Amvrosiadis, G., Carns, P., Cranor, C. D., Dorier, M., Harms, K., et al. (2020). Mochi: composing data services for high-performance computing environments. *J. Comput. Sci. Technol.* 35, 121–144. doi: 10.1007/s11390-020-9802-0

Ryu, M., Kim, Y., Kim, K., and Madduri, R. K. (2022). "APPFL: open-source software framework for privacy-preserving federated learning," in *International Parallel and Distributed Processing Symposium Workshops* (IEEE), 1074–1083. doi: 10.1109/IPDPSW55747.2022.00175

Schmuck, F., and Haskin, R. (2002). "GPFS: a shared-disk file system for large computing clusters," in *Conference on File and Storage Technologies*.

Sharma, R., and Atyab, M. (2021). "Introduction to apache pulsar," in *Cloud-Native Microservices with Apache Pulsar: Build Distributed Messaging Microservices* (Springer), 1–22. doi: 10.1007/978-1-4842-7839-0_1

Souza, R., Skluzacek, T. J., Wilkinson, S. R., Ziatdinov, M., and da Silva, R. F. (2023). "Towards lightweight data integration using multi-workflow provenance and data observability," in *IEEE International Conference on e-Science*. doi: 10.1109/e-Science58273.2023.10254822

Stopford, B. (2018). *Designing Event-driven Systems*. O'Reilly Media, Incorporated.

Taranov, K., Byan, S., Marathe, V., and Hoefler, T. (2022). "Kafkadirect: zero-copy data access for Apache Kafka over RDMA networks," in *International Conference on Management of Data*, 2191–2204. doi: 10.1145/3514221.3526056

Tatebe, O., Obata, K., Hiraga, K., and Ohtsuji, H. (2022). "CHFS: parallel consistent hashing file system for node-local persistent memory," in *International Conference on High Performance Computing in Asia-Pacific Region*, 115–124. doi: 10.1145/3492805.3492807

Vef, M.-A., Moti, N., Süß, T., Tocci, T., Nou, R., Miranda, A., et al. (2018). "GekkoFS-A temporary distributed file system for HPC applications," in *International Conference on Cluster Computing* (IEEE), 319–324. doi: 10.1109/CLUSTER.2018.00049

Wang, C., Mohror, K., and Snir, M. (2021). "File system semantics requirements of HPC applications," in *30th International Symposium on High-Performance Parallel and Distributed Computing*, 19–30. doi: 10.1145/3431379.3460637

Wang, G., Koshy, J., Subramanian, S., Paramasivam, K., Zadeh, M., Narkhede, N., et al. (2015). Building a replicated logging system with Apache Kafka. *Proc. VLDB Endowment* 8, 1654–1655. doi: 10.14778/2824032.2824063

Ward, L., Pauloski, J. G., Hayot-Sasson, V., Babuji, Y., Brace, A., Chard, R., et al. (2025). Employing artificial intelligence to steer exascale workflows with Colmena. *Int. J. High Perfor. Comput. Applic.* 39, 52–64. doi: 10.1177/10943420241288242

Ward, L., Sivaraman, G., Pauloski, J., Babuji, Y., Chard, R., Dandu, N., et al. (2021). "Colmena: scalable machine-learning-based steering of ensemble simulations for high performance computing," in *IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments* (Los Alamitos, CA, USA: IEEE Computer Society), 9–20. doi: 10.1109/MLHPC54614.2021.00007

Yan, X., Hudson, N., Park, H., Grzenda, D., Pauloski, J. G., Schwarting, M., et al. (2025). MOFA: discovering materials for carbon capture with a genai- and simulation-based workflow. *arXiv preprint arXiv:2501.10651*.

Ye, J., Cernuda, J., Rajesh, N., Bateman, K., Yildiz, O., Peterka, T., et al. (2024). "Viper: a high-performance I/O framework for transparently updating, storing, and transferring deep neural network models," in *ICPP'24: The 53nd International Conference on Parallel Processing* (Gotland, Sweden). doi: 10.1145/3673038.3673070

Zheng, W., Kordas, J., Skluzacek, T. J., Kettimuthu, R., and Foster, I. (2024). Globus service enhancements for exascale applications and facilities. *Int. J. High Perform. Comput. Appl.* 38, 658–670. doi: 10.1177/10943420241281744