Impeller: Stream Processing on Shared Logs

Zhiting Zhu*
zhitingz@lepton.ai
Lepton AI Inc.
Cupertino, California, United States

Zhipeng Jia
zhipengjia@google.com
Google LLC
Seattle, Washington, United States

Newton Ni
nwtnni@cs.utexas.edu
The University of Texas at Austin
Austin, Texas, United States

Dixin Tang
dixin@cs.utexas.edu
The University of Texas at Austin
Austin, Texas, United States

Emmett Witchel
witchel@cs.utexas.edu
The University of Texas at Austin
Austin, Texas, United States

Abstract

Current stream processing systems provide exactly-once semantics using checkpointing or a combination of logging and checkpointing. These approaches can introduce high overhead, significantly increasing the latency for normal stream processing because maintaining exactly-once semantics requires coordination across distributed nodes and streams to capture a globally consistent state. We observe that modern distributed shared logs offer a promising solution for maintaining exactly-once semantics with a small overhead. We propose Impeller, a stream processing system that uses a distributed shared log for data storage and exactly-once processing. To maintain exactly-once semantics, Impeller includes a novel and efficient progress marking protocol based on string tags and selective reads in a shared log. The key idea is to leverage the log's record-tagging feature to atomically mark progress across all streams. The experiments over the NEXMark benchmark show that Impeller achieves 1.3× to 5.4× lower p50 latency, or 1.3× to 5.0× higher saturation throughput than Kafka Streams.

ACM Reference Format:
Zhiting Zhu, Zhipeng Jia, Newton Ni, Dixin Tang, and Emmett Witchel. 2025. Impeller: Stream Processing on Shared Logs. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3689031.3717485

1 Introduction

Stream processing is a paradigm for continuously transforming and analyzing data as it arrives. To handle high data rates, stream processing systems distribute the workload across multiple nodes. However, unlike simple data-parallel batch jobs, streaming computations are long-running and stateful – they maintain and update intermediate results over time as new data arrives (which is called the dataflow model [3, 29, 30], and § 2). This stateful nature, combined with distributed processing, introduces challenges to ensuring correct and consistent results in the face of node failures. Stream processing systems must provide fault tolerance mechanisms that can recover the state of a streaming computation after a failure, while still achieving high throughput and low latency. The need to coordinate distributed state and reliably persist high volumes of state updates places significant demands on the storage layer.

Fault tolerance is crucial for stream processing systems to ensure results remain correct in the presence of failures. The key challenge is providing *exactly-once* semantics – ensuring each input record is reflected in the processing results exactly one time, even if failures occur. Exactly-once semantics are difficult because a failure can happen at any point during processing an input record. For example, an operator may have updated its internal state but not yet produced its output record when a failure strikes. Simply reprocessing the input record after recovery would produce incorrect results. Distributed processing further complicates the situation as each node can fail independently. Recovering the entire streaming computation to a globally consistent state requires careful coordination.

Most modern streaming systems use a combination of checkpointing and logging to achieve fault tolerance [5, 37, 43]. Streaming systems mostly use simple, dedicated logging systems; in our work, we want to use the capabilities of modern fault-tolerant, distributed, shared, and scalable logs [8–10, 16]. Scalable shared log systems have nodes dedicated to storage and ordering which provide fault-tolerant storage whose bandwidth capacity scales with increasing resources. Some of these shared logs support string tags for selective reads [22]. String tags are a set of strings provided by the user of the log, stored as metadata associated with a data record in the log, and the log builds an index based on the tags. A key observation for this work is that *string tags can implement an efficient atomic multi-stream append by a specific encoding of metadata on a single data record append (§3.2).* Writing one data record with two string tags (e.g., {"A", "B"}) allows that record to be read by clients interested in the "A" stream and

*Work down while at The University of Texas at Austin.

EuroSys '25, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/2025/03
https://doi.org/10.1145/3689031.3717485

the "B" stream. Compared to current systems that send messages to coordinate the atomic logging, using this advanced features of modern shared logs significantly reduces the cost of maintaining exactly-once semantics and subsequently the median and tail latencies for a range of workloads (§5.3).

In this paper, we propose Impeller[1], a stream processing engine that uses a distributed shared log for both data storage and providing exactly-once processing. Impeller achieves exactly-once processing via a novel progress tracking protocol built on the shared log. The key idea is to use the log itself to coordinate progress across multiple streams and operators, avoiding the need for a separate atomic append protocol.

We emphasize that while Impeller gets scalability, fault-tolerance, and tagging from the underlying log, the contribution of Impeller is how to use the log to provide exactly-once semantics to streaming queries. Preserving exactly-once processing requires several complex protocols that maintain their invariants during arbitrary failures, and it forms the bulk of our design (§3) and the complexity of our implementation (16,895 lines of Go code (§4)).

In Impeller's progress tracking protocol, each operator writes special progress marker records to the log, indicating which input records it has successfully processed, which output records it has produced, and the state changes. By including the log sequence number of the last processed input, progress markers act as a frontier of processing state. Crucially, progress markers leverage the log's record tagging feature to atomically mark progress across all relevant input and output streams. When a progress marker is written, it is tagged with the identifiers of all streams involved in the operator's processing. This effectively creates a consistent cut across all these streams, without needing additional coordination (§3). That one data record appears in all of the identified streams (each of which is totally ordered), identifying a unique log location across multiple streams. To support stateful operators, Impeller also writes periodic checkpoints of operator state to the shared log, tagged with the operator's state stream. On failure recovery, operators resume processing from the point indicated by the last progress marker, and stateful operators restore their state from the most recent snapshot and replay the remaining changelog to the last progress marker.

Impeller's comprehensive support of both stateless and stateful operators allows us to port the complete NEXMark benchmark suite [32, 34]. We compare the performance of Impeller with Kafka Streams [23], which is an industry-leading distributed stream processing engine that uses logging for fault tolerance. Evaluation results show that Impeller achieves 1.3× to 5.4× lower p50 latency, or 1.3× to 5.0× higher saturation throughput than Kafka Streams on NEXMark workloads. We also compare different mechanisms for achieving exactly-once semantics all within our framework: Impeller progress markers, Kafka Streams transactions [23], and checkpointing [35]. We show that progress markers achieve a maximum of 1.4× lower p50 latency and 3.1× lower p99 latency compared to Kafka Streams transactions. Impeller progress marking achieves a maximum of 4.5× lower p50 latency and 5.8× lower p99 latency compared to checkpointing.

[1]An impeller turns kinetic energy into pressure energy to promote the flow of a fluid.

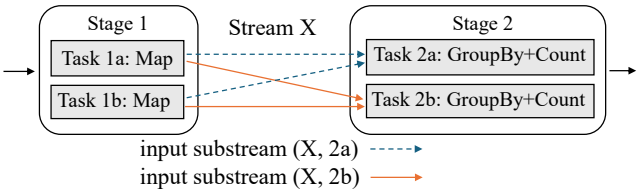

Figure 1. Word count: an example of distributed query processing.

This paper makes the following contributions.

- We propose a novel progress marking protocol that leverages log tags and the total order provided by the shared log to achieve exactly-once processing with small overhead compared to existing approaches (§3).
- We implement a stream processing system, Impeller, over the shared log, which supports exactly-once processing using a wide range of stateless and stateful operators, such as map, window aggregate, and stream-to-stream join (§4).
- We evaluate Impeller using complex queries from NEXMark [32, 34], and compare it with Kafka Streams and two baselines implemented in Impeller. Our results show that Impeller can significantly reduce median and tail latencies while supporting much higher saturation throughput compared to the existing approaches (§5).

2 Background

In this section, we discuss the background of distributed stream processing, fault tolerance for stream processing, and shared logs.

2.1 Distributed Stream Processing

Stream processing frameworks, such as Kafka Streams [23] and Flink [5], process an unbounded and continuous input stream of data records, and generate a similarly unbounded and continuous output stream. A stream processing framework provides users a set of *stream operators*, such as map, join, groupby, and aggregate, to build a *stream query* that analyzes input data records. A stream query is a directed acyclic graph (DAG) of stream operators, where each node is an operator and each edge is a flow of data records from the output of one operator to the input of another. An operator can be stateless (e.g., filter) or stateful (e.g., join).

Because the input data of a stream query is unbounded, it can be useful for users to define window semantics to determine when to output a result. Two options are a tumbling window, which computes a result for every fixed interval of input data (e.g., every 5 seconds), or a sliding window, which updates the result at regular intervals but considers a broader

range of input data (e.g., updating every second based on data from the past 5 seconds).

To execute a DAG of operators in a distributed environment, the DAG is broken into multiple *stages*, where each stage includes a sequence of operators [23, 46] executed in parallel. The data between any two consecutive stages needs to be exchanged such that the output data of the upstream stage is reorganized to fit the input requirement of the downstream stage (e.g., partitioned by a key). The data between any two operators in a stage is pipelined since the output data of the former operator fits the input requirement of the latter operator (e.g., a scan operator followed by a filter).

We define a *task* as a unit of execution for processing a partition of the input data within a stage. Thus, the execution of a stage is parallelized by multiple tasks. In addition, we define a *stream* as the input/output data of a stage, with a *substream* as the input/output data for an individual task within a stage.

Figure 1 shows an example of a stream query that counts the number of appearances of each distinct word in a stream of text sentences. Stage 1 of the stream query includes a map operator (executed by two tasks) to tokenize each sentence into words and send the words to Stage 2. Between the two stages, the words are repartitioned such that identical words will always be sent to the same task. For example, all the "Hello" words from Task 1a and Task 1b will be sent to Task 2a. Finally, Stage 2 uses the groupby and count operators to count the number of appearances of each word within a window (e.g., every 5 seconds). Here, the data sent from Stage 1 to Stage 2 is a stream (i.e., "Stream X") and the input data for Task 2a (X, 2a) and for Task 2b (X, 2b) are input substreams.

2.2 Fault Tolerance for Stream Processing

Fault tolerance in stream processing requires the system to deliver correct query results in spite of system failures and delays. One important and widely adopted criterion for delivering correct results is maintaining *exactly-once semantics* [4, 12, 26, 31, 40, 43]. That is, for each record from the query input data streams, its processing result will be reflected exactly once in the query output data stream even under failures. This paper assumes a failure model with server failures [17], network partitions, and delayed or reordered network packets, which present the following challenges for maintaining exactly-once semantics.

- *Recovering state.* In case of a server failure, a stateful operator will lose its in-memory state. The system needs to reconstruct the state correctly and efficiently to maintain exactly-once semantics.
- *Tracking progress.* To ensure each task recovers to a consistent point after a server failure, the system needs to track the progress of each task, such as the input data records it has processed and the output data records it has generated. In addition, tasks must use this progress information to identify duplicate input data records and avoid generating duplicate output data records.
- *Neutralizing zombies.* A stream processing system uses a task manager to monitor the status of each task [5, 23]. Network failures or delays may cause the task manager to erroneously identify a running task as failed. In this case, the task manager will start a new task and kill the old task if it exists. We call the old task a "zombie task". The system must prevent the zombie from creating duplicate output while detecting and killing it in a timely fashion (§3.4).

Impeller creates multiple streams to allow parallel processing of inputs. Without multiple streams, computation becomes the bottleneck factor. When a computation has fanout, outputs must be atomic with respect to failure, so if for example one input record creates an output record on 4 different streams (which is a property of the streaming query, not a feature of Impeller), then any fault tolerant streaming system needs some way to coordinate the output. Impeller uses progress markers, Kafka streams uses a two-phase commit protocol.

Existing systems maintain exactly-once semantics via checkpointing or a combination of logging and checkpointing. Checkpointing is easy to implement – it is a consistent snapshot of the computational state written to durable storage. However, checkpoints can be large and therefore slow to write. For example, Flink started with checkpointing [5] and applied many optimizations, but migrated to using checkpoints and logging [37] due to unacceptable delays. Kafka Streams [43] also adopts logging plus checkpointing, logging state changes and stream processing progresses while performing full state checkpointing asynchronously. The challenge is that the system needs to atomically log state changes and stream processing progress to get a consistent snapshot. Kafka Streams models these operations as a transaction and uses a separate transaction coordinator to atomically commit this transaction, which introduces performance overheads as we show in our evaluation.

2.3 Fault-tolerant, Distributed, Shared Logs

Fault-tolerant, distributed, shared logs [9] represent a system design where flash drives are attached to multiple machines to create a distributed storage service optimized for appends. Initially, shared log systems were only optimized for write-heavy workloads, but techniques for selective log reads have significantly improved read-heavy workloads [22, 44].

Impeller relies on four key features provides by shared log systems: scalable consensus via the shared log abstraction [8, 22], high-throughput appends with global total order [16], selective reads that are not limited by physical log placement, and set-of-strings tag metadata for log entries [22] (see §6 for details). Even though Impeller uses a log, the read and write bandwidth of that are scalable.

The recent innovation of log tags [22] plays a key role in Impeller's design. Log tags are the mechanism for selective reads: each log entry has a set of string tags as metadata, and

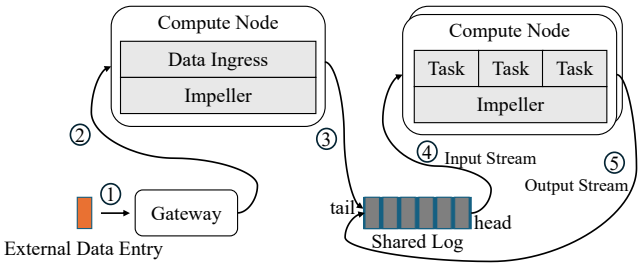

Figure 2. Stream processing in Impeller. Stream processing applications execute on different hosts and use Impeller to compose processing logic and communicate via a shared log.

log reads can specify a specific log tag. Tag format is not defined by the log, any number of any length string is acceptable (though of course large string tags consume log storage). The purpose of tags is to provide users with a flexible mechanism to filter log entries on reads. Impeller uses log tags to logically partition a data stream into substreams, enabling parallel processing using multiple tasks. More importantly, Impeller employs sets of log tags to implement an efficient alternative to Kafka Streams transactions. Impeller implements a multi-stream atomic append using a single log entry with multiple log tags, avoiding the use of a transaction coordinator and the exchange of messages.

Log tags are supported directly by the log, so they are superior to user-defined typed records (like those in DARQ [27]). The log indexes entries based on tags, allowing high-bandwidth reads of records with a specific tag even as the number of tags increases. Impeller is the first stream processing system to incorporate tagged logs into its design.

3 Impeller design

We now discuss the design of Impeller, a high-throughput stream processing system that provides exactly-once semantics [4, 12, 26, 31, 40, 43] based on modern shared logs [8–10, 16, 44]. Conceptually, Impeller stores intermediate stream data in shared logs and achieves high throughput by leveraging the partitioning of shared logs; that is, a shared log can be logically split and processed in parallel to enable efficient execution. At the same time, Impeller maintains exactly-once semantics by leveraging the flexible tag metadata supported by some shared logs, which means that Impeller can atomically mark the progress of stream processing by appending one log record (i.e., the **progress marker**) that is read by multiple logically partitioned logs.

We present definitions and assumptions of shared logs in §3.1 and discuss the modern features that will enable their use in a stream processing system. Then, we discuss supporting stream processing over shared logs (§3.2) and our methods for maintaining exactly-once semantics under failures (§3.3-3.4) in Impeller. Finally, we will discuss supporting checkpointing, supporting window semantics, and compare with Kafka Streams (§3.5).

3.1 Definitions and Assumptions

We assume a distributed and globally ordered shared log, where clients can concurrently append log records. Each log record has metadata that can be provided during the append operation, which includes a list of string tags. The log supports efficient selective reads of a sub-sequence of log records in order based on their tag value [22]. For example, a tag might include a logical stream name and the name of a destination task. We further assume that each stream query is backed by a separate shared log instance, for simplicity (otherwise the performance of different queries would interfere with each other in possibly chaotic ways). The important terms in Impeller are summarized in Table 1.

3.2 Stream Processing over Shared Logs

Figure 2 gives an overview of Impeller. External applications like IoT devices or network monitors send data to Impeller through the gateway's network interface (①). The gateway forwards input records to the data ingress component, which materializes each input record as a log entry in the shared log (②-③). Finally, a stream query pulls records tagged with its input stream from the shared log, processes them, and appends output records tagged with its output stream to the shared log. (④-⑤).

As mentioned in §2.1, a stream query is composed of multiple stages and each stage is executed by multiple tasks. In Impeller, we use a *task manager* for scheduling tasks and monitoring the status of each task (e.g., deciding if a task has failed). The task manager assigns each task a unique id (i.e., *task id*) and an *instance number*. After a task fails and is restarted by the task manager, the restarted task has the same id as the failed one but has an incremented instance number. Each running task repeatedly i) reads a data record from its input substreams, ii) processes the data record and modifies its internal state if this task includes stateful operators (e.g., join or aggregate operators), iii) writes the output records to

Table 1. Impeller terminology.

Term	Explanation
task	Unit of execution. A task processes a partition of data of an input stream for a stage, which could include multiple operators.(§3.2)
stream	Named sequence of data records.
substream	Totally ordered sub-sequence of a stream. A stream is partitioned into substreams to allow parallel processing by concurrent tasks.
progress marker	In-log marker for a computation's progress that supports exactly once semantics under failures (§3.3)
task log	Contains progress markers to optimize recovery (§3.2).
change log	Contains updates to stateful operators to optimize recovery (§3.2).

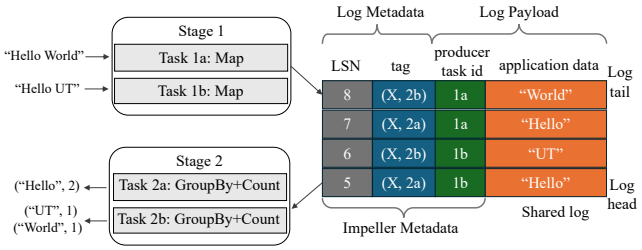

Figure 3. Stream X is stored in the shared log and buffers data between stages. Each log record has a tag that identifies its substream.

all output substreams, which will be consumed by tasks of the next stage, and iv) potentially writes additional log records to the shared log for efficient recovery.

Impeller stores the input and output streams of all stages and information for maintaining exactly-once semantics in the shared log to enable the following benefits: 1) the shared log provides high throughput for communicating data, and it is scalable in these dimensions: read bandwidth, write bandwidth, storage capacity, and ordering capacity (where ordering capacity is the maximum throughput for ordering log appends, which is decoupled from persisting them [16]); 2) the shared log is fault-tolerant, providing a reliable infrastructure; 3) the shared log is durable and supports high write bandwidth, so there is no need for Impeller to handle stream backpressure created by bursty data streams; 4) the shared log enables exactly-once semantics with small overhead through mechanisms discussed in the next subsection.

Representing streams in a shared log. Data flows between two consecutive stages via the shared log. Log records are comprised of metadata and a payload. The metadata includes a log sequence number (*LSN*) assigned by the underlying shared log system, and one or more tags assigned by Impeller. As a matter of encoding, the log payload includes Impeller metadata, like the task id for the task that produced the log record, and application data (see Figure 3).

Figure 3 shows an example of word counting. Stage 1 takes a line of text as input, breaks it into single words, and writes each word along with the producer task id and tag to the shared log. The tag determines the task that consumes the corresponding log record and is represented as a pair of values (Stream Name, Input Substream Name). Tasks selectively read their input substream based on the tag, for example, Task 2a reads records with the tag (X, 2a) only. The tag is known to the fault-tolerant distributed log, which optimizes the placement and transfer of the data read by Task 2a. The tags for log records from Stage 1 guarantee that i) Stage 1's output log records belong to the logical stream called "X", which will be consumed by Stage 2 only and ii) identical words will have the same substream identifier, so they will be processed by the same task (e.g., all instances of the word, "Hello" have tag (X, 2a) and are processed by Task 2a). In Stage 2, each task selectively reads its input substream based on its tag and builds a hash table to count the number of appearances for each distinct word.

Reading from multiple inputs. Tasks in Impeller can have more than one input substream, such as the join operator, which requires two (or sometimes more) input substreams. A stage implementing a join operator can have two upstream stages (e.g., the data from stream Y and stream Z), where each stage places log records with identical join keys in the same substream. Each task for the join operator will read the log records whose tags include the names of the two input streams (e.g., streams Y and Z) and the same substream name. This way, the upstream data records that have the same join key are sent to the same task and the join operation is performed locally. Other operators, such as union, can be supported similarly.

Atomically appending to multiple outputs using tags. If Impeller appends a log record with the tags (X,2a) and (X,2b), then that record will be read by *both* Task 2a and Task 2b. The log record appears once in the log, but it appears in both logical substreams when those substreams are read by a task. This will be important in §3.3.1 when we discuss atomically appending progress markers to multiple substreams.

Supporting fault tolerance. Impeller stores additional record types in the shared log to maintain exactly-once semantics. It uses *progress markers* to atomically mark the progress of stream processing. Progress markers are appended by tasks into regular data streams. Progress markers are also appended to two additional streams, the *task log* and the *change log*, which are used for recovery. The task log stream stores progress markers and the largest instance number of each task for fast recovery. The task log stream is split into substreams, one per task. Each substream is tagged with (T, task id) (see § 3.3 and Figure 4 for more details). The change log stream records changes to the state of each task that includes stateful operators (e.g., hash tables for join or aggregate operators). The change log stream has a substream for each task, tagged with (C, task id). Each task can recover the state of its operators by reading the substream for its task id. We next discuss how Impeller uses these two streams to maintain exactly-once semantics and achieve fast recovery.

3.3 Maintaining Exactly-Once Semantics

We begin with some definitions. Consider a stream operator *Op* with initial state S_0. *Op processes* input record i_n with state S_n to produce a new state S_{n+1} and zero or more output records $[o_n^0, o_n^1, \ldots]$. *Op* processes a sequence of inputs inductively in the obvious way, beginning with state S_0 and processing each record using the previous state.

An execution of *Op* with input sequence $I = [i_0, i_1, \ldots]$ has the: (1) *exactly-once* property if *Op* processes exactly sequence I; (2) *at-least once* property if *Op* processes a supersequence of I; (3) *at-most once* property if *Op* processes a subsequence of I. We say that *Op* has a property if every execution of *Op* has the property. We now argue briefly that if each operator

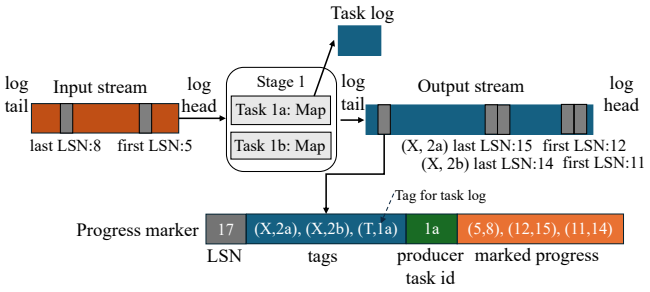

Figure 4. An example of a progress marker for a stateless stage

has the *at-least once* and *at-most once* properties, then every stream query has the *exactly-once* property.

First, note that at-least once is a liveness property and at-most once is a safety property, and together they imply exactly-once. Then, note that operators can be composed by merging their state and piping the output of one to the input of the other, and furthermore that if both operators have the exactly-once property, then so does their composition. Finally, note that a stage can be modeled as a sequential composition of operators, and a stream query is a DAG of stages, which we can again inductively compose.

At-least once. This property is easier to satisfy. Impeller uses a shared log for durability, so no input records can be lost. The task manager ensures that a slow or dead operator will eventually be restarted to recover and continue processing.

At-most once. This property is normally difficult to satisfy, as it requires an operator to atomically record not only what inputs it has processed, but also the resulting state changes and outputs. Kafka Streams's transaction approach is outlined in § 3.6, for reference. Impeller uses progress markers, which are consistent cuts of operator input, state, and output. More concretely, progress markers contain LSN ranges for (1) input records that have been processed, (2) change log records for operator state updates, and (3) output records generated by processing. We call a record *committed* by a task if that task writes a progress marker referencing the record's LSN. Then two simple invariants are enough to establish the *at-most once* property: an operator may only process input records committed by an upstream operator, and it may commit an input record only once. The atomicity of a progress marker append ensures that a downstream operator sees its input as committed exactly when the current operator commits its input as processed.

In the sections that follow, we elaborate on progress marker construction and failure recovery. There are two cases: (1) a stage that only includes stateless operators and (2) a stage that includes at least one stateful operator.

3.3.1 Stateless stage: progress marker Figure 4 depicts a stage with a scan operator followed by other stateless operators, such as map or filter. Each task for this stage processes an input substream and writes the data to one or more output substreams—which, in turn, are input substreams of the downstream stage. For now, we assume all input records are committed, and will discuss uncommitted input records in Section 3.3.3. Periodically, each task writes a progress marker to the shared log. This progress marker stores metadata for the input log records the task has processed, and metadata for the corresponding output log records it has generated since the last progress marker. Specifically, the LSNs of the first and last input log records the task has processed along with the LSNs of the first and the last output log records are recorded in the progress marker. Since a task will write output records to multiple downstream substreams (e.g., for grouping data tuples based on a key), one challenge is to ensure that a progress marker is atomically appended to the shared log and can be read by all the downstream substreams. To address this challenge, Impeller sets *multiple tags* on the log record containing the progress marker–one tag for each downstream substream. When a downstream task selectively reads its substream via the tag, it will read all the progress markers of upstream tasks (see the discussion in §3.2 on log records with multiple tags). The progress marker is additionally appended to the task log with the tag (T, task id) so a recovered task can quickly find the last progress marker by reading the tail of the substream (T, task id). Figure 4 shows an example of a progress marker for a stateless stage. The input and output streams are physically stored in the shared log. The progress marker is appended by Task 1a. It marks progress by storing the information that the output log records for substream (X, 2a) between LSNs 12 and 15 and substream (X, 2b) between LSNs 11 and 14 are computed from the input log records between LSNs 5 and 8. This progress marker includes three tags. Two of them correspond to the two downstream tasks and the third one (i.e., (T,1a)) corresponds to the task log substream. Note that since other tasks can concurrently write log records to the shared log, a log record whose LSN is within the recorded range may be the output of a different task, for example Task 1b. These records will be skipped by downstream tasks based on their tags and task id.

3.3.2 Stateless stage: handling failures If a task fails, it might have written some output records, but not yet written the progress marker for those records. Such records are uncommitted. After Impeller restarts the task, the task finds its most recent progress marker by reading its task log, gets the LSN for the last input record in this progress marker, and processes its substream starting from the next log record after the last input LSN.

3.3.3 Stateful stage: progress marker Tasks in a stage with at least one stateful operator, like groupby or aggregate, must additionally store some representation of their state in order to recover after failure. In Impeller, state is represented as a sequence of change log records. These records are stored in the shared log, and are tagged with (C, task id) as part of the change log stream. We will now discuss how a stateful

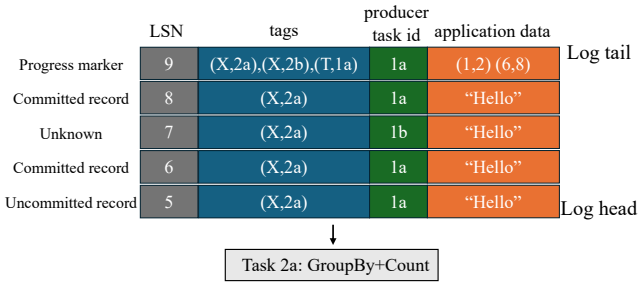

Figure 5. An example of a stateful task processing input records

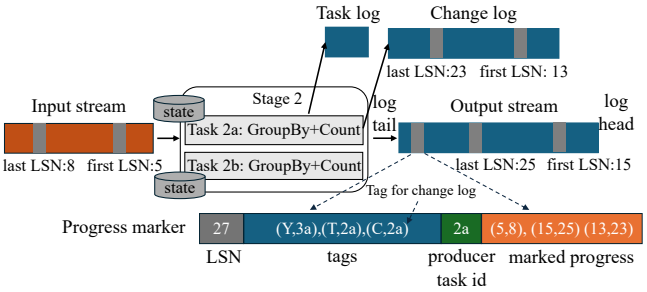

Figure 6. An example of a progress marker for a stateful stage

task processes its input substreams, and then how it generates progress markers. For simplicity, we will focus on a stateful task that takes one substream as input.

A stateful task may read uncommitted log records (i.e., log records not marked by progress markers) due to task failures in its upstream stage. These uncommitted log records are identified based on progress markers from the upstream stage and should be discarded to maintain exactly-once semantics. Conceptually, the stateful task buffers input log records until it sees a progress marker. Then it uses the committed LSN range in each upstream task's progress marker (e.g., the range for recording the output log records) to decide whether a log record is uncommitted.

We use the example in Figure 5 to explain the algorithm. In this example, we assume the task has buffered records whose LSNs are between 5 and 8 and is processing the progress marker from Task 1a. Tasks maintain an in-memory queue of buffered log records and a mapping from producer task id to committed LSN ranges. When a task enqueues a progress marker, it first updates this mapping with the progress marker's producer task id and output LSN range, and then begins processing by repeatedly examining the log record at the head of the queue. This record's producer task id (which may be different from the progress marker's producer task id) is mapped and compared to a set of committed LSN ranges; then this record's LSN falls into one of three cases.

- LSN is before the earliest committed range. This record is `uncommitted` and can be dequeued and discarded. It is not covered in the current committed range and cannot be covered by future committed ranges. In Figure 5, the committed LSN range for Task 1a is [6,8], so log record 5 will be discarded. Its LSN is smaller than the first LSN of the current committed range (i.e., 6), and future committed ranges will always be larger than [6,8].
- LSN is within a committed range. This record is `committed` and can be dequeued and processed by task operators. In Figure 5, log record 6 will be processed by the groupby and count operators.
- LSN is after the latest committed range. This record is `unknown`, as a later progress marker may transition this record to either committed or uncommitted; a record from a producer that has not committed anything also falls in this case. The task stops processing and returns to buffering until it sees a new progress marker. In Figure 5, log record 7 is from Task 1b. Its state is unknown because Task 1b has not yet generated a progress marker that commits an LSN range.

As the stateful task processes the input substream, updates the state, and writes data to the output stream, it needs to periodically record its progress. Similar to a stateless task, it needs to record the range of input/output log records the task has read/written and write a progress marker committing these records with the proper tags such that downstream substreams and the task log substream will read this progress marker. It will additionally record the range of the log records for the change log tag the record with (C, task id). One example is shown in Figure 6. We see that the progress marker needs to record three pairs of LSNs, where the first two correspond to input/output LSN ranges and the third corresponds to the change log LSN range. The record contains multiple tags, including the tags for downstream substreams (e.g., (Y,3a)), the task log substream (i.e., (T,2a)), and the change log substream (i.e., (C,2a)).

3.3.4 Stateful stage: handling failures

A stateful task must recover to a consistent state before it can process new inputs. In Impeller, each progress marker is a consistent cut of state changes and inputs, so a task can simply replay its change log up to and including the latest progress marker. If a previous progress marker has a checkpoint (§ 3.5), the task can replay from after that progress marker instead of the beginning.

To replay the change log, the task reads and buffers log records until it sees a progress marker. Then it takes this progress marker's change log LSN range (e.g., (13, 23) in Figure 6) and replays change log records in the range. It repeats this procedure until it has processed the most recent progress marker, after which its state is synchronized with all committed input records. At this point, it is safe to resume normal processing at the LSN immediately after the most recent progress marker, just like a stateless task.

3.4 Zombies

Network partitions and packet reordering create an important corner case: duplicate task instances. Consider a task instance

that temporarily loses its network connection to the task manager. The manager may assume it failed, and start another instance to replace it. However, the original instance, which we now call a *zombie*, may still be running and generating duplicate outputs.

Impeller solves the zombie problem using two features of the shared log: global metadata and conditional appends. As part of its configuration state, the shared log itself has key-value metadata. The task manager uses this metadata store to associate each task id with an instance number, atomically incrementing it every time a new task instance is started. Impeller invokes the shared log's conditional append with the instance number to exclude zombies: only records matching the current instance number will be appended. It is sufficient to protect only progress markers this way; even if zombies successfully append output or state change records, these records cannot be consumed without a progress marker to commit them. Downstream consumers detect and discard zombie inputs when they receive a progress marker for the same task id but a higher instance number. The instance number is part of the individual log records. Because the instance number is incremented atomically, it is impossible for two progress markers to be committed for the same outputs.

3.5 Discussion

Shrinking progress markers. We have thus far depicted progress markers with two LSN ranges for input and output streams (e.g., Figure 4), and optionally a third range for the change log stream (e.g., Figure 6).

We can reduce this overhead with two observations. First, the start of the input range is not used in Impeller; only the end of the input range is used in recovery. Because the progress marker represents progress up to the end of the input range. Second, Impeller can use the LSN of the progress marker itself as a conservative estimate for the end of the output and change log ranges. The progress marker is the log record that logically follows the last output log record and the last state change log record, so its LSN is a valid upper bound. Therefore, one LSN in each range is unnecessary and can be omitted.

Accelerating state recovery. To reduce the time for restoring the state of a stateful task, Impeller supports checkpointing the state of an operator up to a progress marker. Checkpointing is performed asynchronously to avoid impacting the performance of normal data processing. Impeller builds a checkpoint by replaying the change log of a task up to and including a progress marker, skipping uncommitted records. All the log records before this progress marker can be deleted. Each checkpoint is incrementally built on the previous one by replaying new log records. The checkpoint is for local operator state only, the input and output streams do not need checkpointing because they are persistent. In Impeller, the checkpoint is stored in an external database, such as Kvrocks.

Garbage collection (GC) Records marked by a task as consumed can be garbage collected. Records from the beginning of a substream to the most recently committed consumed records by a task processing that substream can be collected. Shared logs provide a trim API which trims a prefix of the log, so Impeller provides support to determine the proper trim command based on records consumed by tasks. For each substream, there is a GC task that accepts the most recently committed consumed record LSN from all tasks that consume that substream and computes the minimum of the LSNs. There is a master GC task for each shared log that accepts LSN from GC tasks of the substreams stored in that shared log, computes the minimum, and issues the trim API using this global minimum value.

Supporting window semantics. The design of Impeller naturally supports any window semantics, such as different window types (e.g., sliding windows vs. tumbling windows [3]) and different time domains a window is applied to (e.g., event time vs. arrival time [3]). The metadata for maintaining these semantics is stored in the payload of a log record, so it is orthogonal to Impeller's fault-tolerant design.

Log ordering. The input stream and output streams for each stage are independent. They have no ordering constraints and could be placed in different physical logs. The output and changelog stream share a physical log. The advantage of a totally ordered log is that it allows Impeller to use scalars to indicate progress, and therefore recover from faults. (A scalar is sufficient to indicate progress in a totally ordered log, while in general a vector is needed for multiple logs). Impeller progress records are compact, which makes them more efficient than current logging alternatives. Using a single physical log has become more reasonable recently, as systems like Scalog [16] have shown admirable scaling by decoupling ordering from persistence. As a future optimization, it might be possible to require less ordering from the log for some operators, but add a barrier/flush operation before writing a progress marker.

Duplicate appends to a single substream. A producer to a substream might generate the same record multiple times due to various failure scenarios such as time-outs due to network jitter. Impeller must ensure that only one record is consumed despite that multiple records could appear in the input stream.

Each record sent by the producer has a monotonically increasing sequence number. The consumer uses this sequence number with the producer's task id to ignore the duplicate entries appended by the producer. The duplicate entry can be garbage collected in the background.

3.6 Comparing Impeller to Kafka Streams

Kafka Streams leverages Kafka, a partitioned log system, to transfer data and maintain exactly-once semantics [23]. Kafka Streams uses similar terminology as Impeller, except for Kafka *topic*s, *partition*s, and *offset*s, which correspond to Impeller streams, substreams, and LSNs, respectively; this section will use Impeller terms for consistency. Both systems need to synchronize reading input records and writing state change

and output records across multiple streams. The key difference is that Kafka does not support a single-operation atomic appends to multiple streams, so Kafka Streams must implement its own complex transactional protocol on top of Kafka, including a separate transaction *coordinator* with its own *transaction stream*, and a per-task, per-stream *LSN stream* to record the latest input record a task has processed.

Kafka Streams transaction. Consider a task that has processed some input records and buffered the resulting state change and output records in memory. The task must atomically append its (1) latest processed input record to its LSN stream, (2) buffered state change records to its change log stream, and (3) buffered output records to its output streams. In Kafka Streams, the task executes the following two-phase transaction protocol. While Kafka Streams calls it a transaction protocol [43], it implements a multi-stream atomic append, without isolation.

In the first phase, before a task can append to any stream, it must register the stream name and substream identifier with the coordinator, which appends these values to its transaction stream. The task can then (non-atomically) append (1), (2), and (3) above. Afterward, the task asks the coordinator to commit the transaction, and the coordinator appends a *pre-commit* record to its transaction stream. Upon receiving the pre-commit response from the coordinator, the task can resume processing input records, but needs to buffer state changes and output records until the current transaction is committed. If its buffer fills up, it must pause processing until this transaction commits.

In the second phase, the coordinator appends *commit* records to all relevant streams. If all appends succeed, the coordinator appends a *commit* record to its transaction stream; at this point the transaction is committed. Whenever the task tries to start a new transaction (by registering the stream name and substream identifier with the coordinator), it may need to wait for an in-progress transaction to commit.

This protocol introduces several additional appends to implement transactional semantics, whereas Impeller only requires one append per progress marker. The first phase of the protocol is synchronous, which could cause higher tail latency compared to Impeller if the network connection to the transaction coordinator is unreliable. While the second phase is mostly asynchronous, its latency cannot always be hidden by pipelining if buffers reach capacity or if the commit interval is too short (see §5.3.2 for a performance evaluation).

4 Implementation

Impeller is implemented on top of Boki, a state-of-the-art shared log [22] that aims to enable stream processing with high performance and strong consistency. The codebase for Impeller consists of 16,895 lines of Go (all of which is independent from Boki). Impeller implements a set of stateless stream operators: scan, stream/table filter, and map, and a set of stateful stream operators: groupby, stream/table aggregate, stream window aggregate, stream-stream inner join, stream-table inner join, and table-table inner join. The operators are implemented with algorithms from Kafka Streams [23].

Impeller stores state in memory for low access latency and high bandwidth. Updates to the local state store are appended to a change log stream for fault tolerance [33, 43]. When checkpointing is enabled, Impeller checkpoints the state store every 10 seconds as a progress marker is written. Auxiliary data in the progress marker indicates the presence of a checkpoint. In recovery, Impeller will read the latest checkpoint (if it exists) from the checkpoint store and replay the rest of the records in the change log (§3.5).

5 Evaluation

We compare Impeller with Kafka Streams and two baselines implemented in Impeller that maintain exactly-once semantics. Our evaluation addresses the following questions:

1. Does using Boki in Impeller as a shared log provide an inherent performance advantage over Kafka Streams? *§5.2 shows that Kafka's produce-to-consume latency is (in almost every case) lower than Boki's, indicating that Impeller's end-to-end performance advantages are not because it uses Boki as a shared log.*
2. How does Impeller perform relative to Kafka Streams? *§5.3.1 shows that Impeller has significantly lower p50 and p99 latencies than Kafka Streams when processing the same throughput of input events.*
3. How does Impeller's progress marking compare to Kafka Streams's transaction protocol? *We implement Kafka Streams' transaction protocol in Impeller. §5.3.2 shows that Impeller's progress marking protocol yields better performance than the one from Kafka Streams.*
4. How does Impeller's progress marking compare to Flink's checkpointing? *We implement Flink's checkpointing approach [35] in Impeller. §5.3.3 shows that Impeller's progress marking protocol has much lower p50 and p99 latency.*
5. How much performance does Impeller give up to obtain exactly once semantics? *§5.3.4 shows that Impeller's p50 latency is* 1.2×–2.0× *compared to unsafe Impeller, which disables the progress marking protocol. Impeller's p99 latency is* 1.0×–1.8× *unsafe Impeller's p99 latency (Figure 9).*
6. Is Impeller's failure recovery efficient? *§5.3.5 shows that asynchronous checkpointing greatly reduces the recovery time by limiting the size of the log that needs to be replayed.*

5.1 Experiment Setup and Baselines

We conduct all our experiments on Amazon EC2 c5d.2xlarge instances in the us-east-2 region. Each instance has 8 vCPUs, 16 GiB DRAM, 1x200GiB NVMe SSD, and runs Ubuntu 20.04 VMs with Linux 5.10.39 and hyper-threading enabled. Every experiment uses 13 nodes total: 4 storage nodes, 4 input generation nodes, 4 compute nodes, and 1 control plane node.

Impeller's storage nodes run Boki stores for the shared log and Kvrocks 2.7.0 for the checkpoint store. Boki also has

sequencers, which are replicated on three storage nodes. The compute nodes use the Impeller API. The control plane node runs ZooKeeper (for Boki configuration) and a Boki gateway.

Kafka Streams' storage nodes run Kafka (cp-kafka:7.1.0) for the shared log. The compute nodes use the Kafka Streams API (7.1.4-ccs streams and clients). The control plane node runs ZooKeeper (cp-zookeeper:7.1.0).

Impeller uses an in memory state store for its operator state, so we also configure Kafka Streams to use its in memory state store (instead of the default RocksDB). Impeller uses Kvrocks as a checkpoint store, and we configure it to synchronously flush appends to its write-ahead log to avoid losing state checkpoints. Unless otherwise stated, experiments use the following settings:

- Replication factor: 3. Topic replicas in Kafka, sequencer and store replicas in Boki.
- Commit interval: 100ms. This is the interval between progress checkpoints, transaction interval in Kafka Streams, and progress marking interval in Impeller.
- Snapshot interval: 10s. Operator state checkpointing interval in Impeller.

In addition to Kafka Streams, we implement two baselines in Impeller that maintain exactly-once semantics. We implement the baselines in Impeller because we could not fully explain the performance differences by simply comparing end-to-end performance of the systems. There are thousands of configuration settings for these systems.

Kafka Streams transaction. We implement Kafka Streams' transaction protocol [43] in Impeller, which is described in § 3.6. For this protocol, we place one transaction coordinator on each storage node and use gRPC to implement the transaction API. Kafka topics and partitions are emulated by shared log tags in Impeller.

Aligned checkpoint. We implement Flink's *aligned checkpoint* protocol [12] in Impeller. For this protocol, we store checkpoint metadata, state checkpoints and substream offsets in Kvrocks instances. Flink calls these checkpoints "aligned" because they are created by having a checkpoint marker flow through the data stream. This approach creates a logical snapshot, but can only be done as fast as data flows through the system. We allow one in-progress checkpoint in the system.

5.2 Impeller's log vs. Kafka Latency

Table 2 shows the latency between appending a 16 KiB log entry and consuming it from another node at various throughputs. Impeller appends to its shared log, while Kafka appends to a topic with a single partition. Both Impeller's log and Kafka disable batching. Impeller's log's p50 latency is about 1ms higher than Kafka's at all input throughput levels, which is a relative slowdown of 1.3×–1.8×. While Kafka's p99 latency at 10 log entries per second is higher than Impeller's log, it is lower for 50 and 100 log entries per second. We can conclude that Impeller's log does not have a latency advantage over Kafka, and therefore the performance difference between Impeller and Kafka Streams cannot be attributed to using the Boki log instead of Kafka.

Table 2. p50 and p99 latencies to read an appended 16 KiB record for Impeller's log (Boki) and Kafka at different append rates. *aps* is appends per second and the number in the parentheses is the slowdown of Impeller's log relative to Kafka.

	Impeller's log		Kafka	
	p50 (μs)	p99 (μs)	p50 (μs)	p99 (μs)
10 aps	(1.30×) 2714	(0.83×) 3711	2074	4448
50 aps	(1.63×) 2604	(1.10×) 3832	1596	3463
100 aps	(1.76×) 2546	(1.22×) 3596	1449	2942

Table 3. NEXMark queries and their operators. For each query, we include its semantics and the stateless and stateful operators they include.

	Stateless Operators	**Stateful Operators**
Q1: Transforms bids from USD to Euro.		
	Stream map and filter	
Q2: Filters bids by their auction identifiers.		
	Stream filter	
Q3: Joins auctions and people to find the person selling in particular US states.		
	Branch, stream filter	Table-table inner join, stream groupby
Q4: Selects the average of the wining bids for all auctions in each category.		
	Branch, table map values	Stream-stream inner join, stream/table groupby stream aggregate, table aggregate
Q5: Reports, every 2 seconds, the auctions with the highest number of bids taken over the previous 10 seconds.		
	Stream filter	Stream aggregate, stream groupby stream-table inner join
Q6: Reports the average selling price per seller for their last 10 closed auctions.		
	Branch, table map values	Stream-stream inner join, stream/table groupby stream aggregate, table aggregate
Q7: Reports the highest bid each minute.		
	Groupby, stream filter	Stream aggregate, stream-stream inner join
Q8: Reports a 10 second windowed join between new people and new auction sellers.		
	Branch, stream filter	Stream-stream inner join, stream groupby

5.3 NEXMark workloads

We select the NEXMark benchmark suite [32, 34] to experimentally compare Impeller with the three baselines. The NEXMark suite simulates an auction site whose input is a high-volume stream of new users, auctions, and bids. The suite consists of eight queries that contain stateless (e.g., projection and filtering) and stateful (e.g., join) operators; these queries are summarized in Table 3.

We implement the input generator following Apache Flink's NEXMark reference implementation [32]. The average size for bid, auction and new user events are 100, 500 and 200 bytes respectively. The input stream contains 92% bids, 6% auctions and 2% new user events. All experiments run four

input generators, which provide sufficient throughput to saturate all systems under test. All input generators output events in batches, flushing every 10 ms for Q1-2 and every 100 ms for Q3-8. Both Impeller and Kafka Streams has an in-memory output buffer to batch log appends for greater efficiency. We set the buffer size to 128 KiB based on a small sensitivity study that shows making the buffer 64KiB or 256KiB reduces the performance by less than 5% compared to using the size of 128 KiB. All experiments run each query for 3 minutes, which is long enough to show stable performance.

We use the default configuration for Nexmark, which includes skewed key popularity. Impeller has some mechanisms to tolerate skew, like supporting reconfiguration and it can create more substreams to enable higher processing parallelism. Boki has a storage cache on function nodes that reduces IO traffic. But a thorough treatment of data skew is beyond the scope of this paper.

We report the end-to-end event-time latency for a given throughput of input events. This latency is the interval between the record's event-time, the time the event was generated, and its emission time from the output operator [25]. Since each query has different performance characteristics, we start with different input throughputs for each query and run them until their p99 event-time latency exceeds 60 ms for Q1 and Q2 and one second for Q3-Q8.

5.3.1 Impeller vs. Kafka Streams Figure 7 shows the event-time latency as a function of input throughput for the NEXMark queries. Comparing Kafka Streams to Impeller shows that Impeller is either very close in performance to Kafka Streams, or clearly superior.

Q1 and Q2 are single-stage stateless queries representing the simplest cases. Both systems have similar p50 latencies, but the p99 latency of Q1 for Kafka Streams exceeds 100ms at 256,000 events/s while Impeller's p99 latency remains stable until 320,000 events/s. The event-time latency is calculated before a record is pushed to the output stream. In most cases, this latency includes the time the record stays in the input generator's batch buffer, the time to push to and read from the input stream, and then a very small amount of processing time for the stateless operator. The sum of these small latencies roughly corresponds to the p50 latency. The overheads of progress marking is reflected on the record after the progress marking finishes, and therefore it shows up in the p99 latency, where Impeller is superior to Kafka Streams at higher input rates.

For all stateful queries (Q3-Q8, Figure 7(c) to 7(h)) Impeller has much lower p50 and p99 event-time latencies. Kafka Streams' p50 latency is 1.3× to 5.4× Impeller's p50 latency and its p99 latency is 1.2× to 5.7× Impeller's p99 latency. When we limit the p99 latency to 1 second, Impeller can handle 1.3× to 5.0× higher input throughput than Kafka Streams.

5.3.2 Progress marking vs. Kafka Streams transactions in Impeller Figure 7 shows the performance of the Kafka Streams transaction protocol implemented in Impeller. We see that progress marking in Impeller has smaller p50 latencies for Q1 and Q3 and smaller p99 latencies for Q1, Q2, Q3, Q5, and Q7 compared to Kafka Streams transactions. The maximal speedup of Impeller over Kafka Streams transaction for the p50 latency is 3.0×.

We have also observed many cases where the performance difference between the two protocols is not evident. The Kafka Streams transaction protocol has two phases, with the first being synchronous and the second is asynchronous (§3.6). When the commit interval is 100ms, the second phase often overlaps with the normal stream processing work, which hides the latency.

Therefore, we further evaluate the performance impact of four different commit intervals: 100, 50, 25 and 10 ms. We examine commit intervals down to 10ms because the designers of Kafka Streams also evaluate a 10ms commit interval [43]. For each query, we choose the largest input throughput that meets two criteria: (1) it does not increase Impeller's latency for more than 10% compared to the smallest input throughput and, (2) the latency difference between progress marking and Kafka Streams transactions in Figure 7 is within 10%. The intuition is that we want to choose an input throughput that gives both protocols similar performance at the commit interval of 100 ms, and is large enough to stress the system, but is not too large to overwhelm the system.

Figure 8 shows that the progress marking protocol has a larger performance benefit compared to Kafka Streams transaction for all queries when the commit interval decreases. This is because when the commit interval decreases, Kafka Streams transactions need to append more log entries compared to Impeller's progress marking protocol and its second phase cannot fully overlap with normal stream processing, increasing its latency. Looking at Q4 as an example, the p50 latency of Kafka Streams transactions at 10ms is 1.4× Impeller's latency, and its p99 is 3.1× Impeller's.

5.3.3 Impeller progress marking vs. aligned checkpoint Aligned checkpoints are efficient when the amount of state being checkpointed is small, so the p50 latency of Q1 and Q2 is about the same as Impeller shown in Figure 7. However, as the input throughput grows, the need to persist the checkpoint to storage can cause the system to bottleneck on writing checkpoints, which we start to see for the p99 of Q1 and Q2. For the stateful queries (Q3-Q8), checkpointing shows its weakness in inferior p50 and p99 latencies relative to Impeller almost everywhere. There are a few regions, like low input throughput for Q5 and Q6, where checkpointing is the most efficient alternative because the checkpoint sizes are very small. Aligned checkpoint achieves 1.9× lower p50 latency for Q5 at 96,000 events/s. But our results validate the widely perceived weaknesses of checkpoints that they create performance problems as soon as their size is non-trivial. Impeller progress

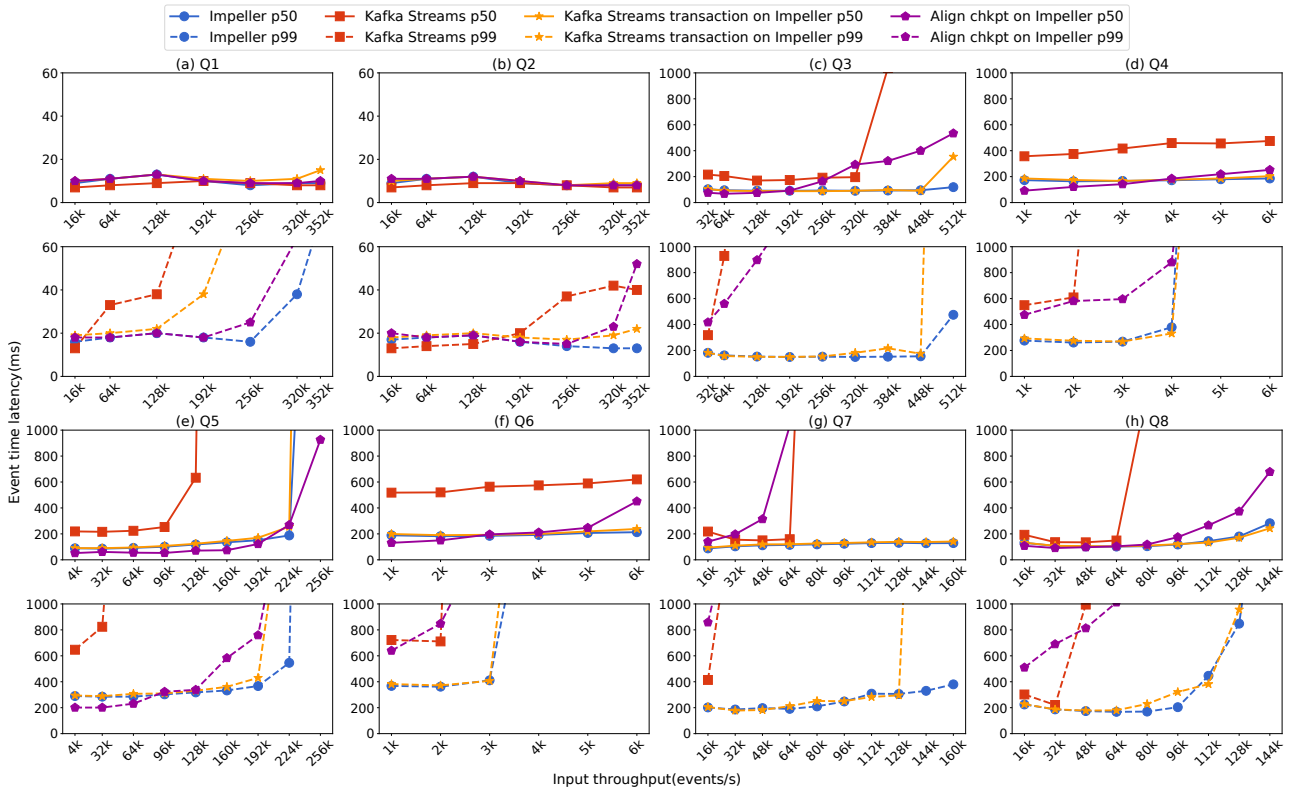

Figure 7. NEXMark results. For a given input throughput, charts show the event-time latency, both median (p50) and tail (p99) for Impeller and Kafka Streams.

marking achieves a maximum of 4.5× lower p50 latency and 5.8× lower p99 latency compared to aligned checkpoint.

5.3.4 Cost of progress marking While Impeller's progress marking protocol enables exactly-once semantics, it introduces overhead to normal stream processing. In this experiment, we want to understand the performance cost of maintaining the strong semantics in Impeller. We implement a variant, called *unsafe*, in Impeller, which disables the progress marking protocol of Impeller, and compare its performance with the other approaches using Q5. The variant is unsafe because without progress marking, it cannot guarantee exactly-once processing if there are server failures. The results in Figure 9 show that Impeller's p50 latency is 1.2×–2.0× the unsafe Impeller's p50 latency. Impeller's p99 latency is 1.0×–1.8× unsafe Impeller's p99 latency. The progress marking protocol adds 15–96ms to the p50 event-time latency of unsafe Impeller and 13–250ms to the p99 event-time latency. Both Impeller and unsafe Impeller's p50 and p99 event-time latency exceed 1s at 256,000 events/s.

5.3.5 Failure recovery. We now evaluate the performance of recovering from failures. We use Q8 for our failure recovery experiment because it contains many stateful operators. The experiment runs 4 tasks for each stage for 330 seconds with input throughput 80,000, 96,000 and 112,000 events per second. The query fails at 300s then recovers, and we measure the recovery time. We include a baseline that does not do checkpointing. Recall that for Impeller, we asynchronously generate a checkpoint every 10 seconds. Table 4 shows that with checkpointing, the recovery process is 14×–16× faster—going from 3.8s–4.7s to below 300ms. The number of log entries and changes needed to read and apply are reduced by a factor of 27×–30×.

6 Related work

Exactly-once processing. Many stream processing engines strive to achieve a consistency guarantee of exactly-once processing, meaning the system can recover to a consistent processing state after server failures, as if stream records are processed exactly once without failures [2, 5, 6, 23, 29, 30, 35]. Taking a consistent checkpoint across different components within a distributed system is challenging. Some systems adopt different checkpointing methods to achieve exactly-once semantics, including lazy checkpointing [29, 30, 35, 36],

Table 4. Recovery performance with and without checkpointing.

		Input throughput (events/s)		
		80,000	96,000	112,000
Recovery time (s)	baseline	3.858	3.920	4.758
	+checkpoint	0.273	0.270	0.297

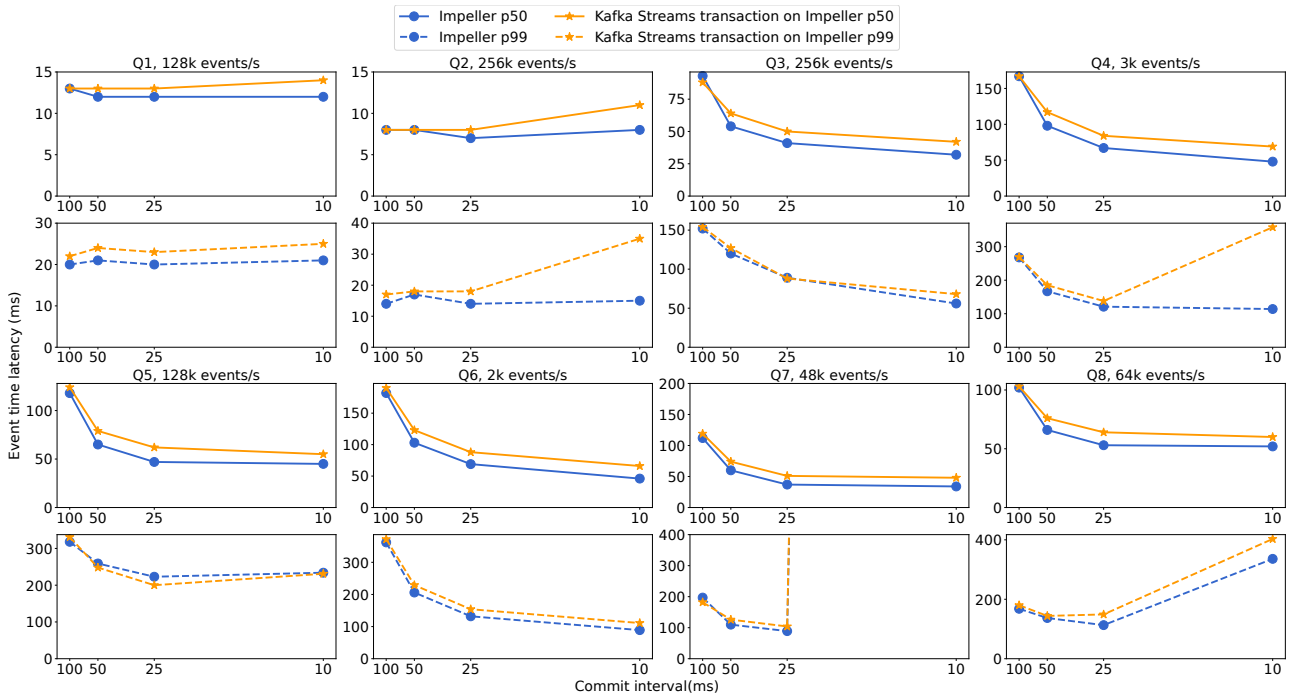

Figure 8. Event-time latencies for different commit intervals at a fixed input rate.

eager checkpointing [1, 2, 6], or a hybrid approach [18]. Checkpointing is known to be costly when the checkpointed state is large, which is confirmed by our experiments using a representative checkpointing method from Apache Flink [35]. Millwheel [2] achieves exactly-once semantics by materializing the IDs for each processed record. To ensure at-least once semantics, Millwheel keeps sending a generated record to its downstream operator until it gets ACKed. To ensure at-most once semantics, each generated record is associated with a unique ID and each operator materializes the IDs of the records received and processed in external storage (e.g., BigTable) and checks the materialized IDs for deduplication [2]. This approach is fundamentally different from Impeller, which does not materialize record IDs for each operator. Instead, it uses a progress marker to record the progress and ensure exactly once semantics.

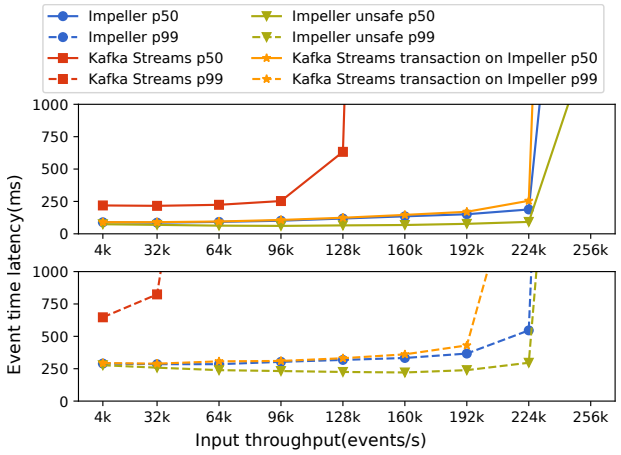

Figure 9. NEXMark Q5: For a given throughput show the median (p50) and tail (p99) processing latency for Impeller and Kafka Streams. This figure includes data for a configuration of Impeller that is unsafe (but faster) because it does not write progress markers.

Structured Streaming [7] and Kafka Streams [31, 43] are both stream processing systems designed for high throughput with exactly-once semantics. Both systems use a combination of logging and asynchronous checkpointing to provide exactly-once semantics. Structured Streaming requires output sinks to support idempotent writes, which is not required in Impeller. Kafka Streams involves a complex two-phase transaction protocol as discussed in the previous section. Our experiments (§5.3.1) show the high median and tail latencies of the Kafka Streams' transaction protocol.

DARQ [27], Ambrosia [19], Netherite [11] and Temporal [39] focus on providing a generic programmer framework for exactly-once processing. While both DARQ and Ambrosia use log-structured storage to facilitate fault tolerance, their designs do not explore improvements on log-structured protocols for scalable stream processing. Impeller introduces a novel atomic append protocol across multiple partitioned logs, which is largely orthogonal to DARQ's innovations on programming model. DARQ and Impeller rely on a single physical log, but Impeller's log is physically partitioned and hence has scalable read and write bandwidth. Recovery in Impeller can be performed independently by workers, without requiring a centrally coordinated Distributed Prefix Recovery

algorithm [28]. While we would like to compare directly with DARQ, its source code is not publicly available.

Portals [38] uses Flink's aligned checkpoint protocol and two-phase commit to provide exactly-once semantics. Exoflow [50] provides exactly-once semantics for workflows that are composed of different data processing systems. Impeller is an efficient stream processing system with fast recovery that can be composed by Exoflow.

Distributed, replicated, shared, fault-tolerant logs. Impeller's storage layer relies on recent advances on distributed, replicated, shared, fault-tolerant logs. Corfu [9] pioneered the abstraction of a shared log, and Scalog [16] further improves shard logs' ordering protocol for scalable, high-throughput log appends. A shared log that is scalable and fault-tolerant enables Impeller to store data records of streams, task logs, and change logs all in the same log. Other shared log works, including Tango [10], vCorfu [44], Delos [8], and Boki [22], improve the selective read functionality of shared logs. Impeller uses a fault-tolerant, distributed, scalable, shared log, but Impeller's contribution is in how it uses the performance and tagging of the log to achieve efficient stream queries.

Boki's main contribution is to introduce shared logs to the serverless paradigm, with three use cases studied in the Boki paper: workflows, a key-value store, and message queues. Workflows and key-value stores use log tags for a consistency protocol and transactions. The message queue only uses tags for sharding, but not for a consistency protocol. In contrast, Impeller is a system for stream processing, which represents an application distinct from all of Boki's use cases. Impeller uses log tags for both data sharding and consistency protocols, but in a way that is specialized for streaming (e.g., Section 3 shows how to provide exactly-once semantics for streaming queries). None of these mechanisms are needed or presented in Boki and Boki's use cases.

Distributed streaming systems. Stream processing is an important data-intensive workload that requires distributed processing for high throughput. Early work [2, 3, 13, 29, 30, 33, 46] provides a flexible abstraction for stream computation while achieving desired properties such as high throughput and fault tolerance. More recent work [14, 20, 21, 24, 41, 42, 45, 47–49] focuses on designing new scaling and scheduling algorithms for distributed stream processing systems to better utilize resources, or develops new algorithms for optimizing stream operators. In contrast, Impeller designs a new and light-weight protocol for maintaining exactly-once semantics to improve the performance of stream processing. While Samza [33] and Facebook Stylus [15] use Kafka, a partitioned log, to provide fault tolerance and communication, Impeller uses the new features in distributed, replicated, shared, fault-tolerant logs to provide exactly-once semantics using a novel, high-performance progress marking protocol.

7 Conclusion

Impeller is the first streaming system that leverages a distributed shared log for communicating data and maintaining exactly-once semantics efficiently at the same time. Through a novel progress tracking protocol that uses log tags for atomic multi-stream append, Impeller minimizes overhead and reduces latency compared to existing approaches.

8 Acknowledgment

We thank our shepherd, Aishwarya Ganesan. Our work is supported in part by PRISM, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] [n. d.]. Trident Tutorial. https://storm.apache.org/releases/current/Trident-tutorial.html (Accessed: May 2024).

[2] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *Proc. VLDB Endow.* 6, 11 (aug 2013), 1033–1044. https://doi.org/10.14778/2536222.2536229

[3] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, out-of-Order Data Processing. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[4] Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das, Ashish Gupta, Haifeng Jiang, Tianhao Qiu, Alexey Reznichenko, Deomid Ryabkov, Manpreet Singh, and Shivakumar Venkataraman. 2013. Photon: Fault-Tolerant and Scalable Joining of Continuous Data Streams. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 577–588. https://doi.org/10.1145/2463676.2465272

[5] Apache Software Foundation. 2024. Apache Flink — Stateful Computations over Data Streams. https://flink.apache.org (Accessed: May 2024).

[6] Apache Software Foundation. 2024. Trident State. https://storm.apache.org/releases/current/Trident-state.html (Accessed: May 2024).

[7] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 601–613. https://doi.org/10.1145/3183713.3190664

[8] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. 2020. Virtual Consensus in Delos. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 617–632. https://www.usenix.org/conference/osdi20/presentation/balakrishnan

[9] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D. Davis. 2012. CORFU: A Shared Log Design for Flash Clusters. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 1–14. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/balakrishnan

[10] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. 2013. Tango: Distributed Data Structures over a Shared Log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 325–340. https://doi.org/10.1145/2517349.2522732

[11] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S. Meiklejohn, and Xiangfeng Zhu. 2022. Netherite: efficient execution of serverless workflows. *Proc. VLDB Endow.* 15, 8 (apr 2022), 1591–1604. https://doi.org/10.14778/3529337.3529344

[12] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *Proc. VLDB Endow.* 10, 12 (aug 2017), 1718–1729. https://doi.org/10.14778/3137765.3137777

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

[14] Ugur Çetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, John Meehan, Andrew Pavlo, Michael Stonebraker, Erik Sutherland, Nesime Tatbul, Kristin Tufte, Hao Wang, and Stanley B. Zdonik. 2014. S-Store: A Streaming NewSQL System for Big Velocity Applications. *Proc. VLDB Endow.* 7, 13 (2014), 1633–1636. https://doi.org/10.14778/2733004.2733048

[15] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. 2016. Realtime Data Processing at Facebook. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1087–1098. https://doi.org/10.1145/2882903.2904441

[16] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. 2020. Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 325–338. https://www.usenix.org/conference/nsdi20/presentation/ding

[17] Michael J. Fischer. 1983. The consensus problem in unreliable distributed systems (a brief survey). In *Foundations of Computation Theory*, Marek Karpinski (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 127–140.

[18] Ionel Gog, Michael Isard, and Martín Abadi. 2021. Falkirk Wheel: Rollback Recovery for Dataflow Systems. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 373–387. https://doi.org/10.1145/3472883.3487011

[19] Jonathan Goldstein, Ahmed Abdelhamid, Mike Barnett, Sebastian Burckhardt, Badrish Chandramouli, Darren Gehring, Niel Lebeck, Christopher Meiklejohn, Umar Farooq Minhas, Ryan Newton, Rahee Ghosh Peshawaria, Tal Zaccai, and Irene Zhang. 2020. A.M.B.R.O.S.I.A: providing performant virtual resiliency for distributed applications. *Proc. VLDB Endow.* 13, 5 (jan 2020), 588–601. https://doi.org/10.14778/3377369.3377370

[20] Philipp M. Grulich, Sebastian Breß, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. 2020. Grizzly: Efficient Stream Processing Through Adaptive Query Compilation. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2487–2503. https://doi.org/10.1145/3318464.3389739

[21] Rong Gu, Han Yin, Weichang Zhong, Chunfeng Yuan, and Yihua Huang. 2022. Meces: Latency-efficient Rescaling via Prioritized State Migration for Stateful Distributed Stream Processing Systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 539–556. https://www.usenix.org/conference/atc22/presentation/gu-rong

[22] Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful Serverless Computing with Shared Logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 691–707. https://doi.org/10.1145/3477132.3483541

[23] Kafka Streams Authors. [n. d.]. Kafka Streams. https://kafka.apache.org/documentation/streams/ (Accessed: March 2024).

[24] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 783–798. https://www.usenix.org/conference/osdi18/presentation/kalavri

[25] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking Distributed Stream Data Processing Systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 1507–1518. https://doi.org/10.1109/ICDE.2018.00169

[26] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. 2019. AStream: Ad-Hoc Shared Stream Processing. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 607–622. https://doi.org/10.1145/3299869.3319884

[27] Tianyu Li, Badrish Chandramouli, Sebastian Burckhardt, and Samuel Madden. 2023. DARQ Matter Binds Everything: Performant and Composable Cloud Programming via Resilient Steps. *Proc. ACM Manag. Data* 1, 2, Article 117 (jun 2023), 27 pages. https://doi.org/10.1145/3589262

[28] Tianyu Li, Badrish Chandramouli, Jose M. Faleiro, Samuel Madden, and Donald Kossmann. 2021. Asynchronous Prefix Recoverability for Fast Distributed Stores. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1090–1102. https://doi.org/10.1145/3448016.3458454

[29] Wei Lin, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. 2016. StreamScope: Continuous Reliable Distributed Processing of Big Data Streams. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 439–453. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/lin

[30] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[31] Neha Narkhede. 2017. Exactly-Once Semantics Are Possible: Here's How Kafka Does It. https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/ (Accessed: May 2024).

[32] Nexmark Benchmark Authors. 2024. Nexmark Benchmark. https://github.com/nexmark/nexmark (Accessed: May 2024).

[33] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. 2017. Samza: Stateful Scalable Stream Processing at LinkedIn. *Proc. VLDB Endow.* 10, 12 (aug 2017), 1634–1645. https://doi.org/10.14778/3137765.3137770

[34] Pete Tucker, Kristin Tufte, Vassilis Papadimos and David Maier. 2010. NEXMark – A benchmark for queries over data streams. https://web.archive.org/web/20100620010601/http://datalab.cs.pdx.edu/niagaraST/NEXMark/ (Accessed: May 2024).

[35] Piotr Nowojski and Mike Winters. 2018. An Overview of End-to-End Exactly-Once Processing in Apache Flink. https://flink.apache.org/features/2018/03/01/end-to-end-exactly-once-apache-flink.html (Accessed: May 2024).

[36] Hang Qu, Omid Mashayekhi, Chinmayee Shah, and Philip Levis. 2018. Decoupling the control plane from program control flow for flexibility and performance in cloud computing. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 1, 13 pages. https://doi.org/10.1145/3190508.3190516

[37] Yuan Mei Roman Khachatryan. 2022. Improving speed and stability of checkpointing with generic log-based incremental checkpoints. https://flink.apache.org/2022/05/30/improving-speed-and-stability-of-checkpointing-with-generic-log-based-incremental-checkpoints/ (Accessed: May 2024).

[38] Jonas Spenger, Paris Carbone, and Philipp Haller. 2022. Portals: An Extension of Dataflow Streaming for Stateful Serverless. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Auckland, New Zealand) *(Onward! 2022)*. Association for Computing Machinery, New York, NY, USA, 153–171. https://doi.org/10.1145/3563835.3567664

[39] Temporal Technologies. 2024. Temporal. https://temporal.io (Accessed: Jul 2024).

[40] Georgios Theodorakis, Fotios Kounelis, Peter Pietzuch, and Holger Pirk. 2021. Scabbard: Single-Node Fault-Tolerant Stream Processing. *Proc. VLDB Endow.* 15, 2 (oct 2021), 361–374. https://doi.org/10.14778/3489496.3489515

[41] Jonas Traub, Philipp M. Grulich, Alejandro Rodriguez Cuellar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2019. Efficient Window Aggregation with General Stream Slicing. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, Melanie Herschel, Helena Galhardas, Berthold Reinwald, Irini Fundulaki, Carsten Binnig, and Zoi Kaoudi (Eds.). OpenProceedings.org, 97–108. https://doi.org/10.5441/002/EDBT.2019.10

[42] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. 2017. Drizzle: Fast and Adaptable Stream Processing at Scale. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 374–389. https://doi.org/10.1145/3132747.3132750

[43] Guozhang Wang, Lei Chen, Ayusman Dikshit, Jason Gustafson, Boyang Chen, Matthias J. Sax, John Roesler, Sophie Blee-Goldman, Bruno Cadonna, Apurva Mehta, Varun Madan, and Jun Rao. 2021. Consistency and Completeness: Rethinking Distributed Stream Processing in Apache Kafka. In *Proceedings of the 2021 International Conference on Management of Data*. Association for Computing Machinery, New York, NY, USA, 2602–2613. https://doi.org/10.1145/3448016.3457556

[44] Michael Wei, Amy Tai, Christopher J. Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, Michael J. Freedman, and Dahlia Malkhi. 2017. vCorfu: A Cloud-Scale Object Store on a Shared Log. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 35–49. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/wei-michael

[45] Le Xu, Shivaram Venkataraman, Indranil Gupta, Luo Mai, and Rahul Potharaju. 2021. Move Fast and Meet Deadlines: Fine-grained Real-time Stream Processing with Cameo. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 389–405. https://www.usenix.org/conference/nsdi21/presentation/xu

[46] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 423–438. https://doi.org/10.1145/2517349.2522737

[47] Xianzhi Zeng and Shuhao Zhang. 2023. Parallelizing Stream Compression for IoT Applications on Asymmetric Multicores. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 950–964. https://doi.org/10.1109/ICDE55515.2023.00078

[48] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M. Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. 2020. The NebulaStream Platform for Data and Application Management in the Internet of Things. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p7-zeuch-cidr20.pdf

[49] Shuhao Zhang, Jiong He, Amelie Chi Zhou, and Bingsheng He. 2019. BriskStream: Scaling Data Stream Processing on Shared-Memory Multicore Architectures. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 705–722. https://doi.org/10.1145/3299869.3300067

[50] Siyuan Zhuang, Stephanie Wang, Eric Liang, Yi Cheng, and Ion Stoica. 2023. ExoFlow: A Universal Workflow System for Exactly-Once DAGs. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 269–286. https://www.usenix.org/conference/osdi23/presentation/zhuang

A Artifact Appendix

A.1 Abstract

The artifact includes source code of Impeller and scripts to recreate the experiments.

A.2 Description & Requirements

A.2.1 How to access The public source code repository is in https://github.com/ut-osa/impeller-artifact and it is archived in zenodo with DOI: *10.5281/zenodo.14877808*.

A.2.2 Hardware dependencies Our evaluation workloads run on AWS EC2 instances in us-east-2 region.

A.2.3 Software dependencies EC2 VMs for running experiments use a public AMI (ami-0c6de836734de3280) from Boki, which is based on Ubuntu 20.04 with necessary dependencies installed. Install instructions for software dependencies to compile the artifact is documented in the Readme of the artifact.

A.2.4 Benchmarks This artifact uses nexmark [34] stream processing benchmark.

A.3 Set-up

Please follow the Readme of the repository to setup the controller machine, EC2 security group and placement group, and compile the source code.

A.4 Evaluation workflow

For artifact functional evaluation, run query 1 for 60 seconds with 1 iterations. Run *./run_q1_quick.sh* in `impeller-artifact/impeller-experiments/nexmark_impeller`.

A.4.1 Major Claims

- (C1): Impeller's progress marking protocol yields better performance than the one from Kafka Streams. This is proven by the experiment (E1) described in §5.3.2.
- (C2): Impeller's process marking protocol has much lower p50 and p99 latency than Flink's checkpointing approach [35]. This is proven by the experiment (E2) described in §5.3.3.

A.4.2 Experiments

- Experiment (E1): [6300 mins compute]: Run *run_q<1-9>_commit_interval.sh* script in `impeller-artifact/impeller-experiments/nexmark_impeller`.
- Experiment (E2): [1600 mins compute]:
  - Run *run_q<1-9>.sh* script in `impeller-artifact/impeller-experiments/nexmark_impeller`.
  - Then run *run_q<1-9>.sh* script in `impeller-artifact/impeller-experiments/nexmark_kafka-streams`