

AgileLog: A Forkable Shared Log for Agents on Data Streams

Shreesha G. Bhat, Tony Hong, Michael Noguera, Aishwarya Ganesan, Ramnatthan Alagappan
University of Illinois Urbana-Champaign

Abstract

In modern data-streaming systems, alongside traditional programs, a new type of entity has emerged that can interact with streaming data: *AI agents*. Unlike traditional programs, AI agents use LLM reasoning to accomplish high-level tasks specified in natural language over streaming data. Unfortunately, current streaming systems cannot fully support agents: they lack the fundamental mechanisms to avoid the performance interference caused by agentic tasks and to safely handle agentic writes. We argue that the *shared log*, the core abstraction underlying streaming data, must support creating *forks of itself*, and that such a *forkable shared log* serves as a great substrate for agents acting on streaming data. We propose AgileLog, a new shared log abstraction that provides novel forking primitives for agentic use cases. We design Bolt, a system that implements the AgileLog abstraction. Bolt uses many novel techniques to make forks cheap and to provide logical and performance isolation.

CCS Concepts: • Information systems → Data streaming; Stream management; Cloud based storage.

Keywords: Shared logs, data streaming, AI agents

ACM Reference Format:

Shreesha G. Bhat, Tony Hong, Michael Noguera, Aishwarya Ganesan, Ramnatthan Alagappan. 2026. AgileLog: A Forkable Shared Log for Agents on Data Streams. In *Symposium on Operating Systems Principles (SOSP '26)*, September 29–October 2, 2026, Prague, Czech Republic. ACM, New York, NY, USA, 18 pages. <https://doi.org/10.1145/3830418.3843875>

1 Introduction

Shared logs [35, 51, 70, 86] are the fundamental building blocks that underpin data-streaming systems [8, 19, 75, 106]. Typically, upstream sources ingest data into the shared log, which orders and makes the data durable. Downstream applications then consume the data from the shared log and process it for many purposes like real-time analytics [100], ML inference [130], search [126], and others [33, 60].

With the advances in reasoning capabilities of large language models (LLMs), a new kind of entity has emerged that can interact with streaming data: *AI agents*. Agents differ

from “traditional” entities in how they operate. The behavior of a traditional application interacting with streams is defined in advance by programmers: it follows fixed logic to produce and consume records, process the consumed data, and generate outputs. In contrast, AI agents can accomplish high-level tasks on streaming data, specified in natural language. Rather than following fixed instructions, the agent uses LLM reasoning to determine on-the-fly what sequence of steps will help achieve its goal. The agent then exercises its ability to invoke external tools [84, 112] to execute the LLM-informed steps on streams. Today, many practical streaming systems allow AI agents to read and write streaming data [13, 14, 41, 103].

Agents interacting with streaming data unlock a variety of use cases. For example, agents allow users who do not write code to run ad-hoc data-analysis tasks on streaming data specified in natural language [79, 104, 109]. Similarly, agents can use LLM intelligence for ambiguous tasks on unstructured streaming data [43–45]. Agents can also use LLM capabilities to generate contextually relevant test events and inject them into streams for testing purposes [31, 79].

Agents, however, introduce new problems for streaming systems. First, agents, unlike traditional tasks, are exploratory in nature: they explore many steps and paths to accomplish a task [59, 84]. Further, with the ease of building agents, many agents would operate on the system simultaneously [49, 84]. Combined, these two aspects cause agents to place significant load, creating performance interference for other workloads, which the system must mitigate. Second, writes performed by agents can be incorrect because LLM-informed actions can be inaccurate [78, 84, 116]; this causes correctness issues for applications consuming the data. Thus, the system must isolate and validate agentic writes before allowing them to take effect. Further, agents may explore many write paths and choose one. The system must thus safely allow such exploration. Finally, to enable coding and testing agents to build and test stream-processing applications, the system must provide sandboxes with realistic data. Unfortunately, today’s streaming systems lack mechanisms to address these problems.

The above problems are not specific to streaming systems; agents create similar issues for any data system they interact with [84]. To address some of these problems, a few recent databases [6, 12, 48], object stores [11], and lakehouses [122] provide *forking* as a core primitive that can create isolated copies of the data system on which agents can safely operate.

Inspired by these systems, we argue that *shared logs*, the core storage abstraction underlying streaming data, must



This work is licensed under a Creative Commons Attribution 4.0 International License.

SOSP '26, Prague, Czech Republic

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2585-2/26/09

<https://doi.org/10.1145/3830418.3843875>

also support creating *isolated forks of itself* to better support AI agents operating on streaming data. With such a forkable shared log, each agent can operate on its own fork, while traditional applications operate on the main shared log, enabling performance isolation. Forks also enable safe handling of agentic writes: an agent’s writes can be isolated on a fork, and then integrated after validation into the main log. Similarly, an agent can explore many write paths in parallel on different forks, isolated from each other and the main log. Finally, each fork naturally acts as a coding and testing sandbox that contains real (production) data.

AgileLog. Based on this, we propose AgileLog, a new shared log abstraction that offers forking as a core primitive. A fork in AgileLog is a *cheap, logically separate, and performance-isolated* child copy of the main shared log that can be read and appended to (just like its parent), and in turn, itself be forked. Besides being the first forkable shared log, a key novelty in AgileLog is a new form of forks that we call *continuous forks*.

The usual notion of forks in prior forkable data systems is that the parent and the child are disconnected from each other at the fork point: while the parent and child share the state up to the fork point, after the fork, the writes on the parent are not visible to the child and vice versa. However, this is insufficient for agents acting on real-time, streaming data. Consider an agent performing streaming analytics on a fork. For the agent to keep producing up-to-date results, the fork must continue to see the real-time data from the parent log that is ingested *even after the fork point*. For this purpose, AgileLog offers *continuous forks* (or cForks). A cFork shares the history with its parent up to the fork point but, importantly, *continuously inherits all further appends* on the parent as well.

A cFork can also be appended to, but those records remain private to the fork. cForks thus provide a unique form of isolation that we call *unidirectional write isolation*: records ingested into the parent are visible to the child but *not* the other way around. Thus, agents performing writes can do so on a cFork, safely isolated from the parent, and consumers of the parent will not see those records. A key property of cForks is that the records appended on a cFork are linearizably interleaved with the inherited parent records. This property is critical for interleaving agentic writes on a fork with non-agentic writes on the parent in the correct order.

In use cases like testing, agentic writes are synthetic and thus remain private to the cFork forever. However, in use cases where agents produce actual data, the writes need to be integrated into the main log. AgileLog thus provides a way to *promote* a cFork (after validation) as the parent. Promote is also useful with exploration: agents can explore many write paths on different cForks and finally promote one. In addition to cForks, AgileLog also offers a (regular) *severed fork*, where the child and parent are disconnected after the fork point. Overall, these primitives provided by AgileLog help satisfy

the needs of various agentic use cases over streaming data.

Bolt System. We build Bolt, an implementation of the AgileLog abstraction. Bolt must satisfy two requirements. First, creating forks must be cheap and quick. Second, forks must be isolated in performance: agentic tasks on child forks should not interfere with workloads on the parent. Since forks must be cheap, any design that copies data to create a fork is a non-starter. To avoid copies, one could host forks on the same set of storage servers that host the main shared log so that the fork and the parent can share the same underlying data. This design, however, will suffer from performance interference. This points to a fundamental tension: copying data provides performance isolation but is slow, while sharing data enables fast fork creation but leads to interference.

Our architectural insight to resolve this tension is to leverage the recent “diskless” shared-log architecture [47, 72, 134] as the foundation for Bolt. Diskless shared logs use *stateless brokers* that make records durable on *cloud object stores* (e.g., S3) instead of local disks. A fault-tolerant *metadata layer* then sequences the durable records, and maintains the log metadata: an index that maps log positions to shared-storage objects. Bolt adopts this diskless architecture to make forks cheap and performance-isolated. Specifically, Bolt creates a fork with *zero-data-copy* by instantiating a fork’s metadata to be the same as that of the parent log, making the fork point to the same objects on shared storage as the parent. Bolt then serves reads and appends to the fork on a separate broker; because the parent and forks are hosted on separate brokers and because cloud object stores scale well, Bolt cleanly isolates the performance of the parent and the forks.

While the diskless architecture provides a good base, Bolt must address important challenges. First, even just copying metadata to create a fork can be slow. Bolt makes fork creations truly low-latency by avoiding even metadata copies via a new *hierarchical log index* technique. Another major challenge is to implement cForks correctly and efficiently. A cFork must continuously inherit records from the parent and must linearizably interleave them with its own appends. This must be efficient: even with many cForks, there should be little impact on the parent’s performance. Bolt implements cForks by carefully sequencing the *metadata of parent’s new records* into the *child’s metadata*, without data copies. Bolt optimizes this idea with new techniques like *tail-only updates* and *lazy-tail propagation*. The end result is that a parent log can have many cForks with little to no performance impact.

Results. Our experiments show that Bolt can create forks quickly. Bolt also effectively isolates workloads on the parent log from agentic tasks. Bolt’s techniques help maintain the parent log’s high performance even with 100s of cForks. We build three real agents powered by LLMs: an ad-hoc analytics agent for IoT data, a stream-processor testing agent, and a supply-chain-management agent. Bolt provides performance isolation and enables safe handling of agentic writes in these

applications. In contrast, Kafka, the de facto shared log for data streaming, suffers from interference (e.g., 14× and 130× higher mean and p99 latencies), and consumer failures when agents produce problematic records.

Contributions. This paper makes four contributions.

- We articulate how a forkable shared log serves as a better substrate for agents interacting with streaming data.
- We present AgileLog, the first shared log abstraction to offer forking as a core primitive. AgileLog offers a novel type of fork, called a continuous fork, as well as primitives for integrating and exploring agentic writes.
- We build Bolt, an AgileLog implementation that realizes cheap and performance-isolated forks by adopting a diskless architecture. It uses novel techniques like hierarchical log indexes, tail-only updates, and lazy-tail propagation.
- We show that AgileLog benefits real agentic applications.

2 Background

We describe the role of shared logs in data streaming, and explain the trend of AI agents interacting with such systems.

2.1 Shared Logs and Data Streaming

Shared logs [34–36, 51, 70, 86, 93] offer an abstraction of a durable and linearizably [67] ordered sequence of records. Shared logs expose a simple API. Clients can *append* records, upon which they are ordered and made durable. Clients can *read* records at given positions in the sequence.

Shared logs are the core storage abstraction underlying data-streaming systems [36, 74, 75, 133, 144]. These systems support many applications like fraud monitoring [33, 96], real-time analytics [100] and search [126], and live inference [83, 130]. In all these applications, upstream sources ingest data into the shared log that downstream applications consume and process. Shared logs have gained significant attention in research and many research implementations [35, 51, 70, 85, 86] exist today. Many practical shared logs exist as well. Kafka [27] is perhaps the most widely used, but others [28, 30, 61, 99] are also popular.

2.2 AI Agents Interacting with Streaming Data

Recent improvement in LLMs’ reasoning abilities [65, 135] has enabled the widespread adoption of *AI agents*. AI agents are systems that couple LLMs’ reasoning with the agency to take actions, such as invoking external tools, maintaining state, and coordinating with other agents [112, 140].

The ability to invoke external tools, in particular, has enabled AI agents to interact with data systems. With this ability, AI agents today can interact with databases [48, 84, 90, 117], object stores [10, 11], lakehouses [122], and warehouses [32, 125, 129]. They can perform a variety of data tasks like retrieval, analysis, and manipulation [38, 49, 54, 84, 98]. Adoption of agents for data tasks is rapidly growing; for example, a recent survey shows that many companies now use agents for data tasks and that a vast majority of

databases are created and operated upon by AI agents [49].

Unsurprisingly, agents have also started interacting with streaming data systems. Today, many practical systems support agentic access to streaming data [14, 41, 103]. They do so via the Model Context Protocol (MCP) [26, 68]: each system provides an MCP server through which agents can read from and write to data streams [1, 13, 20, 79, 80, 107].

3 Agents Acting on Data Streams

3.1 Real Use Cases

AI agents operating on streaming data enable an array of use cases. We now present these use cases, which we distilled from real scenarios that practical systems aim to support.

Agentic Data Analytics. Agents can analyze streaming data, enabling non-programmers (e.g., sales personnel) to “talk” to data via natural language [38, 48, 54]. Agents can construct on-the-fly dashboards, answer windowed streaming queries, compute metrics over time [104, 109], and handle ad-hoc questions [79] like “*what are the trending products in the last 24 hours?*” or “*show revenue impact of the latest feature deployment?*” [104]. In these scenarios, an LLM helps the agent decompose tasks and iteratively determine next steps. For example, based on LLM interaction, an agent may first probe streams to identify contents, sample records to infer schema, then read records for analysis from the target streams. Dashboards and streaming-query tasks require continuously fetching new records, while point-in-time queries require access only up to a specific point in the stream.

Real-Time Context Access. As many applications become agentic, they require the latest events happening in an organization to get real-time context [50, 55, 71, 111] to make accurate decisions [42, 123]. For example, an agentic order-fulfillment application requires access to real-time inventory status [55]. As in the previous use case, agents may iteratively probe and read streams to construct the context.

Agentic Stream Processing. Stream processors are long-running programs that consume records from streams, act on them, and write outputs to other streams. Traditionally, stream processors are hand-coded (e.g., Flink jobs [2] or Pulsar functions [3]). However, LLM-powered agents can also act as stream processors, particularly for tasks over unstructured, heterogeneous data. For example, an agent can act as a content-moderation processor to identify inappropriate images [45]. Here, the agent would invoke the LLM on each image from the input stream and write its reasoning and decision to an output stream. Downstream consumers (either traditional or agentic) then act on the output stream. Similar examples of agentic stream processing for customer support [44], supply chain [56], and fraud-risk analysis [43] exist as well. Unlike prior use cases, where agents only read streams, here, they both read and write to streams.

Agentic Coding & Testing of Stream Applications. Instead of having agents themselves operate on streams, one

could now use AI coding agents to develop (regular) applications that interact with streams [13, 80, 121]. Agents can synthesize stream processors or streaming SQL from prompts [80]. To build the application, the coding agent interacts with streams: it repeatedly reads the streams (to infer schema and understand the data) and iteratively fixes the code. Such iterative development existed even with hand-coded stream applications [16], but the rate of such iteration will grow with agents.

Agents can also be used to test streaming applications. LLMs excel at test generation [23, 39, 97] and thus agents with access to code can generate contextually relevant test cases [17]. Agentic testing is already applied to semantic testing of stream processors: the agent generates and injects test events into the stream, and checks if a stream processor behaves correctly [31, 79, 108]. For example, agents can inject synthetic transactions to test if a fraud-detection model [33, 96] catches evolving fraud patterns [37, 69]. Such testing can be applied to hand-coded programs or during iterative agentic development. Agents also enable schema testing, where events with new schemas are injected to check if processors can handle them. Further, agents can also perform “what-if” counterfactual testing [84]. With LLM aid [82], agents can do what-if tests on real-time data by simulating and injecting different event sequences and observing how downstream outputs change. Testing agents both read and write streams, but the written data is synthetic.

3.2 Agents on Streams: Problems

Need for Performance Isolation. On the surface, AI agents may appear to be yet another set of clients interacting with streaming data. However, agents have characteristics that distinguish them from usual applications. As identified by recent work [59, 84], agentic tasks are exploratory and speculative in nature: agents generate multiple courses of action to accomplish a task. This exploration may occur within a single agent or via an orchestrator spawning sub-agents, each exploring a distinct hypothesis or path. In either case, these paths can be partial attempts at the task, probes to understand the data, suboptimal solutions, or validation of prior steps. While such exploration helps improve the overall accuracy of the agent, it leads to a large number of requests being issued, potentially with suboptimal access patterns [119, 142], which places significant load on the underlying system [62, 63, 84]. Given the ease of building and deploying agents, many agents will operate on a system simultaneously, multiplying this load. This will negatively interfere with traditional production applications, especially latency-critical ones. However, current streaming systems do not provide mechanisms to isolate the performance of traditional workloads from agents.

Need for Safely Handling Agentic Writes. Allowing agents to directly write to a data system can be risky [78, 84, 116, 117]. This is because LLMs can generate inaccurate

actions, causing agents to write incorrect data. This, in turn, can lead to incorrect downstream computations based on those writes. Thus, the system must isolate agentic writes from the original stream and validate them (e.g., by a human in the loop or a “reviewer” agent [117, 118]). Only after such validation must the agentic writes take effect on the stream, and consumers must be able to see them. However, current streaming systems do not provide mechanisms to do this.

Further, an agent may explore many write paths either because of its inherent exploratory nature or because it uses sub-agents to explicitly explore many paths. For example, in content moderation, many sub-agents, each powered by a different LLM [64], may explore different ways to analyze the input and produce the output. It must thus be possible to explore these different paths in an isolated manner. Eventually, when a path is chosen as the desired one (e.g., based on result accuracy), it must be possible for the writes in that path to become part of the actual stream. However, today’s streaming systems do not offer such support for exploration.

Need for Realistic Sandboxes. Coding and testing agents require sandboxes. Testing agents inject test events, which must remain isolated from the production stream. Today, agents, for this purpose, create separate streams. While this provides isolation, testing is done purely over synthetic data. However, practitioners often desire testing using real data [15, 18, 57, 124]. With streaming data, testing is most effective when executed against realistic production data which carries inherent temporal context that helps uncover corner cases. For instance, when testing a fraud detector, the agent must be able to interleave the generated fraudulent test-events within a realistic transaction stream. Similarly, coding agents can produce more accurate code with realistic data. However, today, creating such isolated yet realistic sandboxes with production streaming data is not possible.

4 A Forkable Shared Log Abstraction

To better support agents operating on streaming data, we argue that the shared log, the core storage abstraction underlying streaming data, must support creating *isolated forks of itself*. AgileLog is a shared log abstraction that offers such forking capability. A fork in AgileLog is a *cheap, logically separate, performance-isolated* child copy of the main log.

With AgileLog, each agent can operate on its own fork, while traditional workloads operate on the main or root log. This provides performance isolation for workloads on the main log. AgileLog also enables safe handling of agentic writes. Writes performed by an agent on a fork are isolated from the root log, which can be validated before allowing them to take effect on the root log. Further, AgileLog enables simultaneous exploration of different write paths on separate forks. Finally, each fork acts as a sandbox with real data.

This section first describes the AgileLog abstraction (§4.1) and then how AgileLog satisfies the needs of use cases from

```

interface AgileLog:
  // Traditional shared-log API calls.
  Position append(Record r);
  List<Record> read(Position from, Position to);
  // Continuous fork; indicate if this fork is promotable.
  AgileLog cFork(promotable = false);
  // Severed fork; optionally can fork from a past offset.
  AgileLog sFork(optional Position past);
  bool promote(); // Promote this (promotable) cFork.
  void squash(); // Delete this fork.

```

Figure 1. AgileLog Interface.

§3.1 (§4.2). The next section (§5) describes the implementation of the AgileLog abstraction that enables cheap fork creation and ensures logical and performance isolation.

4.1 AgileLog Abstraction, Interface, Fork Semantics

An AgileLog instance has a root log, which can be forked to create child copies. Each child can be appended to and read (just like its parent), and can itself be forked. Figure 1 shows AgileLog’s interface. AgileLog augments the traditional shared-log API (read and append) with two new calls for forking: cFork and sFork; these calls return another AgileLog instance, which can be further forked to create deeper forks. Promote enables writes on a fork to take effect on its parent and squash deletes a fork.

Continuous Forks. Typically, a fork means that the parent and the child will be disconnected from each other at the fork point, and data modifications that happen either on the parent or the child will remain private to them. Forks provided by prior databases [4, 5, 12, 48] offer this regular fork semantics.

With streaming systems, live data keeps flowing in and agents need to operate on the live stream. Thus, it is insufficient to provide the regular fork, where the child stops seeing new data ingested on the parent after the fork point. Take the real-time streaming query or dashboards examples from the agentic data-analytics use case. To keep producing up-to-date results, the fork where the agent runs must continue to see the appends on the parent *even after the fork point*. Thus, AgileLog supports a novel form of fork that we call *continuous forks* or cFork. A cFork shares the history with the parent log up to the fork point, but it also *continuously inherits* new appends at the parent after the fork point.

While appends in the parent are visible to children, children may want to insert their own records into their forks. Take the testing use case; here, the agent simulates fraud scenarios by injecting fraudulent transactions. By operating on a cFork, the agent can see the original transaction stream and also inject fraud events. However, these events must remain private to the child. A cFork provides this required isolation: writes on child forks remain private to them and consumers of the parent will not see them. Thus, cForks enable a unique form of isolation that we call *unidirectional write isolation*: writes on the parent are visible to the children,

but not the other way around. This is in contrast to regular forks provided by prior data systems, where the isolation is bidirectional.

Since a cFork inherits records from its parent and also has its own records, how should these records be ordered on the cFork? AgileLog guarantees that appends on a cFork are linearizably [67] interleaved with those on the parent. That is, if an append *A* finishes on the parent and then *B* is appended to a cFork, then *B* is guaranteed to appear after *A* in the cFork.

In some use cases, appends on a cFork remain private to it forever (e.g., a cFork created for testing). However, in some cases, the agentic writes must be reflected on the main log; to enable this, AgileLog allows a cFork to be promoted as the parent (as we discuss below). If such promotion is desired, then the cFork must be created with the *promotable* flag set.

Figure 2(a) and (b) show cFork semantics. In 2(a), the cFork acts as a live read-only copy of the parent. In 2(b), in addition, records are inserted in the live copy (e.g., *Z* on green fork) that are linearizably interleaved with the parent records.

Severed Forks. AgileLog also offers regular forks via a sFork call, where the child is “severed” from the parent after the fork point. By default, a severed fork is created from the current log tail of the parent (at the instant sFork is called); however, it can also be created from a past offset of the parent. In either case, a severed fork shares the parent’s history up to the fork point and it stops seeing further parent updates. Some use cases may not perform writes on a severed fork as shown in 2(c), where the fork acts as a read-only snapshot of the parent. Some use cases write to a severed fork as shown in 2(d). The new records on the fork could be entirely unrelated to the parent’s data (like the yellow fork in 2(d)); or, they could be some perturbations of the parent values (like the green fork).

Squash. Squash deletes a fork and no further operations can be performed on it. For example, it can be used to dispose of unused sandboxes. AgileLog disallows squashing root logs.

Promote. To support use cases where agentic writes must be reflected on the parent, AgileLog offers a promote API. A promote call on a cFork (created with the *promotable* flag) of a parent makes that fork become the parent itself. Future operations that refer to the parent will now occur on the promoted child and ones that refer to the child are disallowed. sForks or non-promotable cForks cannot be promoted.

cFork and promote together enable safe integration of agentic writes. First, agentic writes can be isolated on a cFork. The cFork can then be validated and promoted. Since AgileLog guarantees linearizable interleaving of agentic writes on the cFork with the parent log’s records, promotion preserves the correct record order. If validation fails, the cFork is squashed without affecting the main log or its consumers.

One option instead of using a cFork would be to have the agents write to a temporary log, validate the writes there, and then append to the main log. However, this allows only

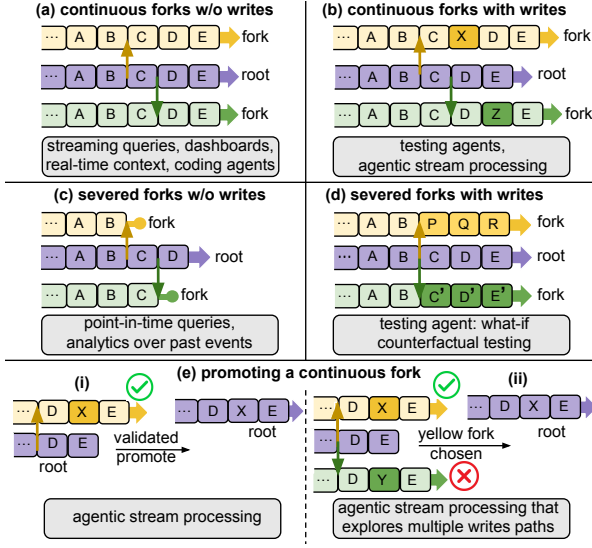


Figure 2. AgileLog Primitives. The yellow (upward) and green (downward) arrows are the fork points at which the yellow and green forks were created from the (purple) root log. Writes on the forks are shown in a darker shade. Gray boxes show use cases where a particular primitive is useful.

non-contextual, stateless checks of the current agentic writes (e.g., if the schema of the records is correct). In many cases, however, validation requires state: the current agentic writes must be validated *along with* the records in the main log before the fork and other records on the main log after the fork (ingested by non-agentic producers). cForks and promote enable such stateful validation. Specifically, the cFork will contain the previous records and the writes from non-agentic producers linearizably interleaved with the current agentic writes. Thus, the cFork has the required temporal context for validation. Validation can be done, for instance, by running a copy of the stateful consumer on the cFork to check if it produces desired results. If so, the cFork is promoted; otherwise it is squashed. Even if only stateless validation is required, cFork and promote provide a neat API to validate and reflect agentic writes on the main log, instead of having to manage temporary logs. Figure 2(e)(i) shows how cFork and promote safely handle agentic writes. The agent performs its writes on the yellow fork, which is promoted as the parent after validation.

Creating a promotable cFork introduces some restrictions on its parent. These restrictions arise because, after promotion, the parent will have additional records: ones that were inserted into the cFork. Thus, allowing reads on the parent beyond the fork point (before a promotion) could make those reads invalid (after a promotion). For this reason, AgileLog does not allow reads on the parent beyond a promotable cFork’s fork point until the fork is promoted or squashed. For example, in 2(e)(i), consumers on the parent cannot read entries *D* and beyond. However, once the cFork is promoted, those entries can be read. Further, while appends to the parent can continue, AgileLog cannot return indexes for those

appends beyond the promotable cFork’s fork point. This is because, after promotion, the indexes would change (e.g., in 2(e)(i), *E*’s position in root will change after promotion). Not returning indexes works in practice as many applications anyway do not use them [86]. Lastly, AgileLog also stops operations beyond the promotable cFork’s fork point on other non-promotable cForks of the parent until a decision is made. Note that these restrictions apply only when there are active promotable cForks. Despite these restrictions, agentic applications can keep this blocking interval short by delaying creation of the promotable cFork until the proposed writes have been computed. Potentially long-running analysis, such as log reads needed to determine those writes, can be done on separate non-promotable cForks, which have no restrictions. The agent can then create a promotable cFork, append the proposed records, and perform stateful validation. Furthermore, by copying the consumer state at the fork point, validation replays only the post-fork suffix rather than the entire log, making it quick in practice.

cFork and promote also enable agentic exploration with writes. The agent can simultaneously explore different paths on different promotable cForks and finally promote one. With many promotable cForks, only the first promote will succeed and AgileLog internally squashes others. As shown in 2(e)(ii), the agent explores two paths and promotes the yellow fork.

4.2 How Different Agentic Use Cases use AgileLog

Agentic Data Analytics. Agents for streaming queries and dashboards can see the most up-to-date data by operating on a cFork of the main log as shown in Figure 2(a). Agents for ad-hoc queries don’t need the latest data and so can work on a sFork (2(c)). By running these exploratory tasks on forks, the root log’s performance would remain unaffected.

Real-Time Context Access. Agents that require fresh real-time context can obtain it via a continuous fork (2(a)).

Agentic Stream Processing. Agentic stream processors (e.g., for content moderation) use cFork and promote to safely integrate agentic writes as shown in Figure 2(e)(i). The agent can periodically create and write to a new cFork, validate and promote it, and repeat. If the agent explores many paths (e.g., content-moderation with different models), it can explore each path on a cFork and promote one (2(e)(ii)).

AgileLog can also be used within conventional stream-processing pipelines. For example, an agent can test a candidate stream processor by creating cForks of its input streams and running the candidate against these live, isolated copies. To modify or replace a deployed processor, the application can stop the existing processor, create promotable cForks of its output streams, and direct the replacement processor’s outputs to those forks. The candidate outputs and their downstream effects can then be validated before the forks are either promoted or squashed.

Coding and Testing Agents. Coding agents can operate on a (read-only) cFork (2(a)). A testing agent can operate on a

non-promotable cFork, where it can safely inject test events linearizably interleaved with the real data (2(b)). It can explore many test cases in parallel on different cForks. For what-if testing, the agent can create many sForks to simulate alternate worlds (2(d)). The yellow fork in 2(d) explores a completely different event sequence after the fork point. The green fork, in contrast, explores values based off of the parent’s data (e.g., sensor values that are off by 5% in an IoT stream). To do this, the agent creates a cFork, reads from it, perturbs the read values, and writes them into an sFork.

4.3 Non-Agentive Use Cases

While the exploratory nature of agentic applications makes the need for AgileLog’s forking primitives especially urgent, AgileLog also enables non-agentive use cases. In particular, AgileLog provides a natural substrate for testing stream processors against live production data in CI/CD pipelines. Today, developers often maintain expensive shared staging or shadow clusters that are periodically synchronized with production, or develop sandboxing and multi-tenancy mechanisms to safely share infrastructure between testing and production [127, 143]. With AgileLog, each CI/CD test can spin off its own fork of the production stream, providing access to realistic data without requiring dedicated infrastructure. Furthermore, cForks can receive data at production rates while preserving production-workload characteristics.

5 Bolt Design

Bolt is an implementation of the AgileLog abstraction. Bolt aims to satisfy a few high-level requirements. First, creating a fork must be a *cheap* and *low-latency* operation. Second, a fork must be *isolated in performance* from the parent; this ensures that resource-intensive agentic workloads on forks do not degrade the performance of workloads on the parent log. Finally, Bolt must *efficiently scale* to many forks.

5.1 Design Rationale and Techniques Overview

One option to realize forks is to use mirroring tools [29, 53, 81, 110] that can continuously mirror a shared log. To create a fork, one can use these tools to mirror the parent log to the child log. Typically, these tools mirror data to a separate cluster with dedicated storage and compute. Thus, forks would be isolated in performance from the parent. However, this approach requires copying data, which makes creating and maintaining forks prohibitively slow and expensive. Another option is to build forks atop traditional shared-log designs [27, 36, 51, 86, 105] that use a set of replicated servers (or brokers) that store data on local disks. Here, a forked child can be hosted on the same servers as the parent and thus can share the underlying data on disk without copying. However, this would fail to provide performance isolation due to resource contention at the servers.

Diskless as a Substrate. We instead realize that the recent *diskless* architecture in shared logs [22, 72, 89, 134] provides

a good substrate for Bolt. In contrast to traditional designs that replicate data on the local disks on a set of brokers, diskless shared logs make brokers stateless and delegate storage to scalable shared object storage (e.g., AWS S3 [120], Azure Blobs [91]). The shared-storage layer provides the required data durability. A fault-tolerant metadata layer sequences durable records[†] and maintains an index to map log positions to shared-storage objects. The diskless architecture is increasingly popular for its two benefits [131, 141]. First, it avoids inter-fault-domain replication traffic between brokers, which is expensive in clouds. Second, it frees operators from managing disks and helps elastically scale the system.

Bolt adopts the diskless architecture to make forks zero-data-copy and quick, and isolate performance (§5.3). At a high level, Bolt creates a fork by having the forked child’s metadata point to the same objects on shared storage as the parent, without copying data. Further, Bolt serves operations to the created fork on a separate broker from that of its parent, avoiding interference to the parent at the broker layer. Since modern cloud object stores can scale reads exceptionally well without becoming a bottleneck (e.g., with 4MB objects and 10 object-name prefixes, one can get 220 GB/s read throughput with S3 [25]), Bolt can offer performance isolation between workloads on the main log and the agents on forks.

Challenges. While diskless architectures provide a good foundation, Bolt must solve some critical challenges. First, how to make fork creations truly low-latency? Copying even the metadata can incur high latency. Second, how to implement cForks that continuously inherit new records ingested on the parent while linearizably interleaving appends on the child with the ones on the parent? Third, even with many cForks, how to inherit updates with little to no impact on the parent’s performance? Finally, how to realize promotes? **Techniques Overview.** Bolt addresses these challenges via various techniques. Bolt avoids metadata copy using a *hierarchical log index* that enables swift fork creations (§5.4). Bolt realizes cForks as a metadata-layer operation (without data copies) by sequencing the metadata of new records on the parent into the child’s metadata (§5.5). To scale to many cForks, Bolt proposes *tail updates* and *lazy-tail propagation* (§5.5.1, §5.5.2). Finally, Bolt also realizes promotes as a metadata-layer operation without data copies (§5.6).

5.2 Diskless Shared Logs: A Primer

Before describing Bolt, we provide a primer on diskless designs. Figure 3 shows a typical diskless shared log [47, 72, 89, 134]. The system consists of stateless brokers, a metadata layer, and cloud object storage. A single diskless instance can host *multiple logs* (a unit of total order), each with a unique identifier. The metadata layer is a fault-tolerant group that implements state-machine replication (SMR) [113] using

[†]While some shared logs [51, 86] also separate durability and ordering, they aren’t diskless: they use local disks, not scalable cloud stores.

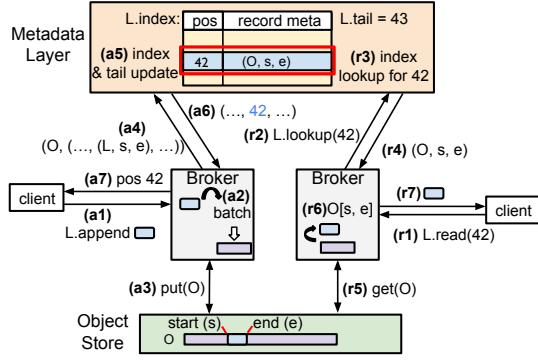


Figure 3. Diskless Primer. The figure shows how appends and reads work (for entries in log L). Only L 's metadata is shown in the metadata layer.

Paxos [77] or Raft [94]. It maintains an *index* and *tail* for each log. The index maps log positions to locations of records in the object store; the tail is the next free position in the log.

Appends. Appends to a log can be routed to any broker (step a1 in Figure 3). The broker batches several records from many clients (a2) and writes a large object to the object store (a3). Note that an object can contain records from different logs. Once the object write completes, the broker informs the metadata layer of the object identifier (O) and the metadata for each record within O (i.e., the log identifier and byte ranges within O) (a4). The metadata layer then sequences these records by logging this metadata to its consensus log and executing them as SMR operations. Specifically, for each record in O for a log L , it assigns positions from L 's current tail, and updates L 's index and tail (a5). It returns these positions (a6), and the broker acknowledges the clients (a7).

Reads. Reads to a log can be routed to any broker (r1). The broker contacts the metadata layer (r2), which then looks up the index to fetch the object identifier and byte range for the requested position (r3, r4). The broker uses this metadata to retrieve the record from the store (r5, r6) and return it (r7).

5.3 Forks in Bolt: Overview

We now describe how Bolt implements the basic forking functionality atop the diskless architecture.

Creating a Fork. Conceptually, Bolt treats a fork as an independent log with its own total order and metadata. Upon a fork of a parent log P , Bolt's metadata layer creates a new child log C and initializes C 's metadata to be the same as that of P . Thus, C 's metadata points to the same data objects on the shared storage as P , resulting in zero-data-copy fork creation. Concretely, when an agentic client needs a fork to operate on, it issues a fork call to any broker, and the call gets routed to the metadata layer. The fork operation is sequenced in the consensus log of the metadata layer as an SMR operation. When the operation executes, the metadata layer creates a new child log C , and initializes C 's index and tail to be the same as P 's index and tail at this instant. §5.4 discusses how Bolt does this initial fork creation efficiently

to reduce latencies.

To track the continuous-inheritance relationships between a cFork and its parent, Bolt maintains an *inheritance forest* (a collection of trees) at the metadata layer. Initially, there could be a number of *root* logs; these are the base logs. For each such root log, Bolt maintains an *inheritance tree*, which captures the continuous forks starting from that root. When a cFork C is created from a root log P , a new tree node is created for C as a child of P . If subsequent cForks are created from C , those will be added as children of C in that inheritance tree. Since a severed fork does not continuously inherit updates from its parent, when a severed fork S is created, a new tree is created with S as the root. If cForks are created from this severed fork, those forks will be added as children of this root.

Bolt finally assigns a broker distinct from those hosting the root logs to serve the created fork to avoid performance interference. Bolt maintains a small pool of brokers to avoid spinning up new ones upon fork creation. The log identifier of the created fork, i.e., C , is returned to the client. Clients can invoke future operations on the fork using this identifier.

After a Fork. After a fork is created, appends and reads to the fork C are handled by the separate broker. At the metadata layer, appends to C are sequenced at C 's current tail and added to C 's index and remain logically isolated from its parent; reads are handled by reading C 's index.

For severed forks, any append on the parent or the child gets sequenced only within that corresponding log index, logically isolated from each other. This is because severed forks provide bidirectional write isolation. However, for cForks, appends must not only be sequenced and recorded within the index for the log being appended to, but for all logs in its inheritance subtree. §5.5 describes how Bolt achieves this.

5.4 Zero-Metadata-Copy Fork Creation

To initialize a fork's metadata, one could copy the parent's metadata. While faster than copying data, copying even the metadata incurs high latency and memory overhead in the metadata layer. Further, during the metadata copy, metadata operations on the parent's index would have to wait behind it, impacting the performance of appends on the parent.

Bolt thus avoids physically copying even metadata to create forks. Bolt can do so because indexes of the fork and the parent up to the fork point would never change subsequently as logs are fundamentally *append-only*. Thus, Bolt just makes the child's index point to the parent's index for positions up to the fork point, making forks zero-metadata-copy. To realize this, Bolt uses a data structure that we call *hierarchical log index* (or HLI). With HLI, upon a fork, Bolt initializes the child C 's index as an empty map, but C 's tail is initialized as the parent P 's tail (or as the specified offset +1 if a past offset is given when creating a sFork). Future appends to C will be sequenced at C 's tail. During reads, index lookups for positions lower than the smallest position in C 's index are

looked up in P . If P is also a fork created from another log Q , it will result in a recursive lookup in Q 's index. Thus, HLI avoids duplicating metadata for log prefixes shared between parents and their children.

5.5 Supporting Continuous Forks

We now explain how Bolt implements cForks, where the child must see new records appended to the parent even after the fork point. Bolt realizes that continuous inheritance can be implemented as a *metadata-layer-only operation*, without any data copies. Specifically, it is sufficient to propagate the parent's metadata updates to the child's metadata. This metadata propagation indicates the presence of new parent records to the child. Since operations on the child also flow through the metadata layer, those operations will see the effect of the new parent records, helping establish a linearizable order across records on the child and those in the parent.

BoltNaiveCF: Naive Metadata-based cFork. An approach to realizing cFork is that whenever the metadata layer sequences a record in log P , it synchronously updates the indexes and tails of each descendant $D \in \mathbb{D}_P$, where $\mathbb{D}_P$ is the set of all descendants in P 's inheritance subtree. Thus, any further append to D will be linearized after P 's append. We refer to this version of Bolt as BoltNaiveCF. Figure 4(a) shows how BoltNaiveCF works. Here, log G is cFork-ed to create R when G 's tail was 1. With zero-metadata-copy fork creation, only R 's tail is set to 1, without copying the index entries to R . After the fork, record r_0 is inserted in R . Later, record g_1 is inserted in G . This results in inserting g_1 's index entry into both G 's and R 's indexes. Thus, when r_1 is subsequently appended to R , it gets correctly linearized after g_1 . Although records in G appear at different positions in R , by referring to the same objects on the shared storage, continuous inheritance can be realized without any data copies.

BoltNaiveCF, unfortunately, has two drawbacks. First, when an index entry m is sequenced into a log P , it inserts m into the index of every $D \in \mathbb{D}_P$. This exacerbates the memory overhead in the metadata layer ($n \times$ overhead with n forks). Second, updating the index of all descendants in the critical path would impact the parent's append latency and throughput. This can be especially detrimental when agents create many cForks of a root log (e.g., during agentic exploration).

5.5.1 Continuous Inheritance via Tail Updates. To alleviate the above problems, Bolt makes a key observation. Upon the append of a record to a parent, the effect of the append can be reflected on the descendants by *updating only their log tails and not their log indexes*. Concretely, upon an append of r at any P , Bolt inserts record metadata for r only into P 's index. However, instead of also doing so for all $D \in \mathbb{D}_P$, Bolt only updates their tails. This tail update ensures that any future append to a descendant D will reflect the presence of r , thereby achieving the effect of linearizably inserting r into D . Bolt thus completely avoids any metadata

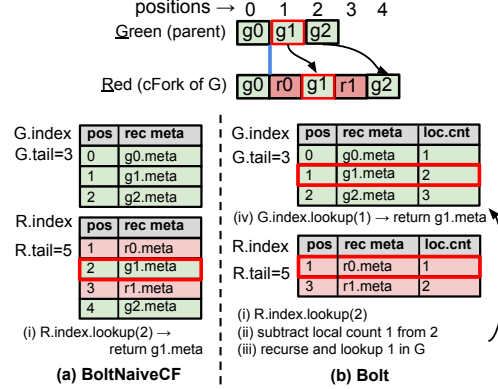


Figure 4. Continuous Inheritance. G is cForked to get R (blue line is the fork point). (a) and (b) show how BoltNaiveCF and Bolt implement cForks, respectively. Lookup of position 2 in R is also shown (red boxes).

duplication, i.e., each record r 's metadata is only inserted into one log index, one where it was originally appended.

Bolt realizes the tail-update technique by augmenting HLI. Figure 4(b) shows how the augmented HLI avoids duplicating metadata even when continuously inheriting records. After the fork and insertion of r_0 in R , when record g_1 is inserted in G , Bolt avoids copying g_1 's index entry into R 's index, but instead only updates R 's tail to 3, reflecting g_1 's presence. Thus, when r_1 is appended to R , it gets correctly linearized after g_1 at position 3 in R ; R 's tail is now 4. g_2 is then appended to G and again only R 's tail is updated (to 5).

Lookups. With tail updates, when looking up a position in a child, if the corresponding record was inherited, then it won't have an entry in the child's index (e.g., lookup for position 2 in R 's index in 4(b) since g_1 was inherited). Despite this, Bolt must be able to retrieve the metadata for such positions.

At a high level, the metadata for inherited records must be looked up in the parent's index. However, the challenge is that an inherited record may appear at different positions in the parent and children. In 4(b), g_1 is at position 2 in R but at 1 in G . How should Bolt determine which position in the parent a requested position in the child maps to? Observe that g_1 's position in G (i.e., 1) can be obtained by subtracting from g_1 's position in R (i.e., 2), the number of *locally appended records* to R prior to g_1 (1 record). Thus, to look up inherited records, HLI augments the entry for a position x in a log L 's index with another field: a cumulative count of records locally appended to L up to and including x .

The lookup works as follows. If a requested position i is in a log L 's index, then Bolt simply returns that metadata entry. If not, Bolt retrieves the locally appended entries in L before i : it finds the largest position smaller than i in L 's index and gets the corresponding local count l . It then looks up $i - l$ in L 's parent, recursing further if $i - l$ is also an inherited position in L 's parent. For example, in 4(b), position 2 is not in R 's index. Thus, Bolt finds the largest position < 2 in R , i.e., the 1st record. By subtracting local count of 1

from the requested position 2, Bolt looks up R 's parent G 's index at position 1 which correctly returns $g1$'s metadata. In comparison, BoltNaiveCF (4(a)) performs a single lookup on R 's full index, but does so at the cost of metadata duplication.

5.5.2 Lazy Tail Propagation. Even with tail updates, *eagerly* iterating over each descendant and updating its tail in the critical path of an append to the parent would affect the performance of the parent. Instead, Bolt uses a *lazy* mechanism. Here, Bolt updates a descendant D 's tail only when there is an operation on D that requires querying its current tail (i.e., an append to D or a read to positions beyond the currently reflected tail in D).

Efficient Tail Updates. Bolt realizes tail updates and lazy propagation efficiently even with wide or deep forks using a data structure that we call Lazy Tail Tree (LTT). Logically, LTT is an inheritance tree of tails, i.e., each log with its current tail forms a node, with edges between parents and inherited children. In this logical representation, an insert into a parent p must not only update the parent tail node, but must also recursively update the tails of each descendant node that lies in p 's inheritance subtree. This requires a data structure that can efficiently encode and update subtrees. To do this, physically, we maintain LTT as an Euler Tour of this logical tree stored in a balanced binary search tree (BST) [66]. An Euler Tour E of a tree T enumerates the nodes in the order they would be seen during a depth first traversal of T [138]. In the Euler tour E , every subtree rooted at a node of T appears as a contiguous range within E . Thus, a subtree update in T converts to a range update on the BST. Bolt then implements range updates lazily on the BST via standard lazy-propagation techniques in logarithmic time [88]. Similarly, point queries for tails on the BST take logarithmic time. Thus, forks could be arbitrarily deep or wide, but the complexity remains logarithmic in the number of logs.

5.6 Promotes and Squash

Promote. In Bolt, promoting a cFork can change the parent log beyond the fork point. Thus, parent reads cannot be allowed beyond it. Similarly, no appends or reads beyond this point can be allowed on any non-promotable descendant. To achieve this, Bolt maintains the earliest such position *earliest-fp* (fork point of the earliest promotable cFork that still exists) for each log. Both direct log reads and recursive reads from non-promotable descendants that go beyond *earliest-fp* are blocked. To block appends on non-promotable descendants, Bolt augments the lazy tail tree with the ability to block or unblock a subtree through the same lazy-propagation techniques as earlier. Further, since promotes can change the positions on the parent log, Bolt stops returning the append indexes after *earliest-fp*.

Bolt implements promotes via metadata copies, without any data copy. To promote a child forked at position fp , metadata entries from the child after fp are copied over to

replace entries beyond fp in the parent. Unlike fork creation, where the history will be long and thus metadata copy can be slow, copying metadata on a promote is a reasonable design choice since only metadata entries after fp are copied.

Squash. Squashing a log clears the metadata-layer state for it and all its children recursively. Apart from deallocating the indexes, Bolt also removes the corresponding nodes from the lazy tail tree. Lastly, squashing a promotable cFork also updates the parent's *earliest-fp*, and unblocks it if the squashed log is the only existing promotable cFork of its parent.

Lease-based Internal Squash. A promotable cFork imposes several restrictions on its parent, such as preventing reads beyond *earliest-fp*. Therefore, a rogue agent that creates a promotable cFork but neither squashes nor promotes it for an extended period can significantly degrade parent performance. To address this, Bolt optionally allows an authenticated entity to bound the lifetime of promotable cForks by configuring a lease duration. When leasing is enabled, Bolt's metadata layer internally squashes any promotable cFork whose lifetime exceeds the configured duration. This mechanism limits the impact of rogue or misbehaving agents.

5.7 Alternative: Tagged Global Address Space Design

In Bolt, we model each root and fork as an independent log with its own local address space. An alternative design could place all roots and forks in a single global address space, using record *tags* to identify the logs to which each record belongs. Prior systems have similarly partitioned a global address space among multiple logs [70, 136]. Because global positions remain stable when records are reassigned during promotion, this design could return append positions even while promotable cForks exist, removing one restriction of our promote implementation (see §4.1). It would not, however, eliminate the restriction on parent reads beyond the fork point. Moreover, efficient reads would still require per-log indexes, while a global address space would expose non-contiguous offsets to applications and impose an order over otherwise unrelated records. These tradeoffs illustrate that the AgileLog abstraction is independent of any particular address-space organization: Bolt implements one point in the design space, while alternative implementations can optimize for different application requirements.

5.8 Implementation

We implement Bolt in C++ (~21K LOC). We use eRPC [73] for communication and MinIO [92] as the shared-storage layer, but any S3-compatible store works. The metadata layer uses a Raft implementation [139]. We equip brokers with a local object cache to improve read performance. While Bolt hosts forks on separate brokers from those of their parents for performance isolation, it co-locates many forks of a parent on the same broker for two reasons. First, these forks often read the same inherited objects (e.g., across many analytics agents); co-location allows them to reuse broker-local

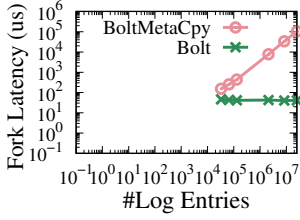


Figure 5. Fork Latency.

cached copies and reduce read traffic to the shared-storage layer. Second, grouping many forks to a broker reduces the overhead on the metadata layer. That said, applications can configure Bolt to host forks on separate brokers.

6 Evaluation

Our evaluation answers the following questions:

- Does Bolt enable low-latency fork creation? (§6.1)
- Does Bolt isolate the performance of workloads on root logs from those running on their forks? (§6.2)
- Can Bolt support many cForks without impacting the performance of workloads running on the parents? (§6.3)
- How do the techniques to implement cForks help? (§6.4)
- What is the metadata memory overhead in Bolt? (§6.5)
- What is the overhead of HLI’s recursive lookups? (§6.6)
- How does complex nesting impact parent inserts? (§6.7)
- Are promotes of cForks fast in Bolt? (§6.8)
- Does Bolt benefit real agentic applications? (§6.9)
- How does Bolt deal with rogue agents? (§6.10)

Setup. We experiment on an xl170 [7] CloudLab [52] cluster. We use nine nodes for MinIO and three replicas in the metadata layer. Each broker runs on its own node. We use 4KB records. Since no existing shared log offers forks, we mostly compare Bolt to its many variants. In some experiments, we compare to Kafka, a popular shared log for data-streaming, and Inkless [22], Aiven’s fork of Kafka that implements the diskless shared-log architecture.

6.1 Fork Creation Latencies

We first measure fork latency. We compare Bolt against BoltMetaCpy, a variant that creates forks by copying the metadata of the parent to instantiate the child index. Figure 5 shows fork latency with varying lengths of the parent log. In both systems, the created forks are assigned to a pre-existing broker different than that of the parent. By avoiding any data or metadata copy, Bolt has low latencies ($\sim 50\mu s$), regardless of the log length. Although BoltMetaCpy avoids data copies, it incurs high latency to copy metadata (e.g., 100ms with 25M entries, three orders of magnitude slower than Bolt).

Copying metadata not only hurts fork latency but also the parent’s performance. This is because record appends to the parent must sequence the metadata into the parent’s index, which is affected by the copy to create the fork. Figure 6 shows this: during the creation of 100 forks (such as at the start of an agentic exploration), with BoltMetaCpy, the

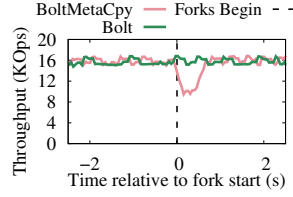


Figure 6. Parent Throughput during Fork Creation.

parent’s throughput suffers during fork creation; in contrast, with Bolt, the parent’s performance remains unaffected.

6.2 Performance Isolation

We next show that Bolt provides performance isolation for a latency-critical (lc) workload from a read-heavy workload resembling agentic data analysis. The lc-workload appends and reads records at the tail. The analysis workload reads records in bulk from the same log (up to an offset). We run these workloads on three systems, disk-based Kafka, Inkless [22] (a diskless Kafka implementation), and Bolt. Inkless does not offer any notion of isolated forks, but we can *simulate an isolated, read-only fork* by creating a new diskless broker that accesses the production topic-partition’s data through the shared object storage layer. Disk-based Kafka neither provides any notion of forks nor a means to create diskless broker instances and hence the analysis task must share data by running on the same storage brokers as the lc-workload. In Bolt, the analysis task creates and runs on a sFork that uses its own broker but shares the parent’s data via shared storage. Figure 7(a) shows the end-to-end (e2e) latency of the lc-workload with and without the analysis task running alongside it. In both Inkless and Bolt, the lc-workload experiences no interference from the analysis task. This demonstrates that the diskless architecture provides a strong substrate for implementing the AgileLog abstraction. In contrast, with Disk-based Kafka, the lc-workload’s latency degrades by $2\times$ due to resource contention at the shared broker. We note that this behavior is not specific to Kafka but to any shared log with stateful brokers or storage servers that store data on local disks [36, 51, 86].

We next show that the lc-workload is performance-isolated even with appends to the forks. We do not compare to Kafka or Inkless, since they have no notion of forks that allow appends that are logically isolated from the root log. The lc-workload appends to the root at 13Kops/s. We create a cFork that also appends its own records at 13Kops/s. Figure 7(b) shows the lc-workload’s e2e latency with and without the cFork. Even with appends on the fork, the mean and p99 e2e latencies are unaffected.

6.3 Performance of Parent with Many cForks

We now examine if the root log remains performance-isolated even when it has many cForks. We run an append-only workload on the root with 0, 10, and 100 cForks, and plot the root log’s append throughput vs. latency. As shown in Figure 8(a), even with many cForks which must continuously inherit appends from the root, the root log shows no performance degradation relative to the case with no forks. Figure 8(b) shows the case where the system hosts 32 root logs (mimicking real deployments where many topic-partitions are hosted on a diskless instance). We then create 0, 10, and 100 cForks per root log. Even with many root logs, each with many forks, root logs’ performance does not degrade in Bolt.

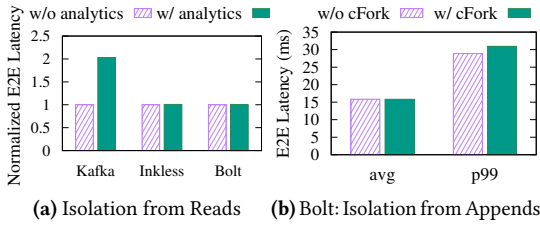


Figure 7. Performance Isolation with Bolt.

We next evaluate whether root logs remain performance-isolated even when many of their cForks are also being appended to. We append to 8 root logs at 20Kops/s in aggregate. We then vary the number of cForks per root log with a load of 40 ops/s per cFork (resembling agentic workloads, where appends are interspersed with slow reasoning steps). Figure 9 shows that root log append latency remains constant as the total cFork load per root log increases from 2Kops/s with 50 cForks to 10Kops/s with 250 cForks.

6.4 cFork Techniques Ablation

We now evaluate the different techniques that Bolt uses in the metadata layer to implement cForks. We compare Bolt with (i) BoltNaiveCF, which copies metadata entries from the parent’s index to the child indexes on every parent append; (ii) Bolt-ET (eager-tail), which improves over BoltNaiveCF by updating only the tails but does so eagerly and without the lazy tail tree (LTT). The difference in these variants emerges only when the data-layer throughput is high enough to stress the metadata layer. However, producing this metadata load would require more brokers and MinIO storage nodes than we have machines. Thus, we measure the metadata-layer throughput in isolation. We emulate two sets of brokers sending requests to the metadata layer: one set hosting the root log, which append records to the log; another set that hosts the cForks, which perform reads on the cForks.

Figure 10 shows the root log’s metadata performance with 10, 100, and 1000 cForks. First, BoltNaiveCF performs well with 10 cForks, but its approach of copying the index entries to all children becomes a bottleneck with 100 and 1000 forks, resulting in low metadata-layer performance. Once the metadata layer bottlenecks, the overall system’s performance would not scale further. Second, Bolt-ET’s tail-only updates help it closely match the performance of Bolt for 10 and 100 forks. However, with 1000 forks, eagerly updating even just the tail becomes the bottleneck. In contrast, Bolt’s metadata layer continues to scale well even with 1000 forks. This is because, unlike Bolt-ET, Bolt updates tails lazily (only upon a tail read on the cFork) and efficiently (via LTT).

6.5 Metadata Memory Overhead

We now examine the metadata memory overhead of Bolt and BoltNaiveCF. We measure the memory occupied to support 1000 cForks of a root log to which 1M records are appended.

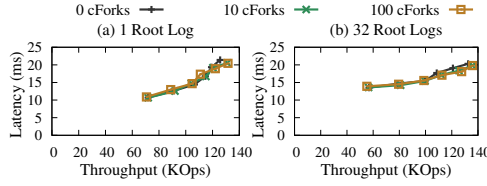


Figure 8. Performance Impact on Parent with many cForks.

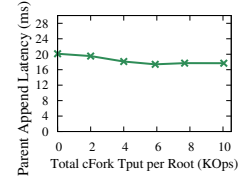


Figure 9. Parent Latency Impact with cFork Append Load.

BoltNaiveCF requires 4.4GB to maintain the cForks. In contrast, Bolt avoids metadata duplication, requiring only 8MB. Bolt’s techniques thus are not only critical for high metadata-layer performance but also to reduce memory overhead.

6.6 Recursive Metadata Lookup Overhead

With HLI augmented with tail updates, Bolt may require recursive lookups to find the metadata for a given log position (Figure 4(b)). We measure this overhead and compare it to BoltNaiveCF that doesn’t require recursive lookups. We create nested cForks with 1M records per level and query the deepest cFork, forcing recursion to the root. Figure 11 plots the metadata lookup latency against nesting depth. Bolt has minimal degradation: only 5.2% slower at depth 7 ($\leq 2\mu$ s overhead). Since this occurs in the metadata layer, which constitutes only a small fraction of the client-perceived latencies (which are in *ms*), the end impact is negligible.

6.7 Impact of Complex Nested cForks

We now evaluate parent performance under complex nesting. We vary the total number of logs from 100 to 1M and, at each scale, organize the logs into four types of cFork hierarchies. The *star* arrangement places every cFork directly below the root, producing a wide, single-level hierarchy. The *balanced* hierarchy uses a fan-out of four, increasing both breadth and depth as it grows. The *chain* represents the deepest extreme, with every cFork created from the preceding cFork. Finally, the *irregular* hierarchy selects parents at random, producing uneven branching and depth. We measure the hierarchical log index data structure with tail-only updates and lazy-tail propagation in isolation. Each measured root insert is interspersed with an insert and a read on distinct, randomly selected cForks. Figure 12 shows that hierarchy shape has little effect: for a fixed number of logs, mean insert latency on the parent differs by less than 0.5μ s across all structures. Thus, performance impact on root logs is minimal even with extreme nesting and many cForks. Further, as the number of logs increases, latency grows only logarithmically.

6.8 Performance of Promote

We now evaluate promotes. As figure 13 shows, metadata-copy-based promote incurs only 10-100s of μ s, even with many records. In contrast, a temporary-log-based approach that copies data to promote (described in §4.1) with Kafka (“DC (Kafka)”), is much slower than Bolt. Thus, apart from

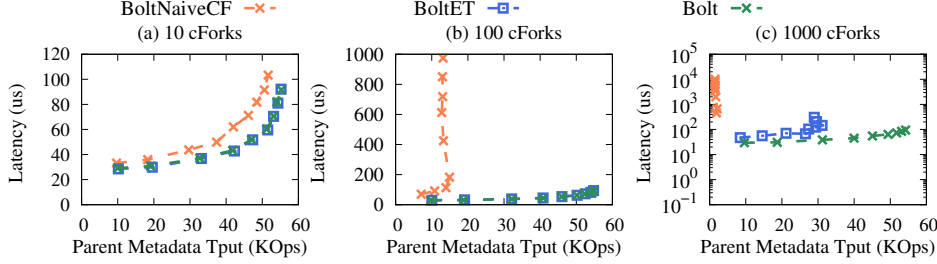


Figure 10. cFork Techniques Ablation.

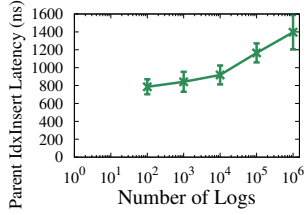


Figure 12. Impact of Complex Nested cForks.

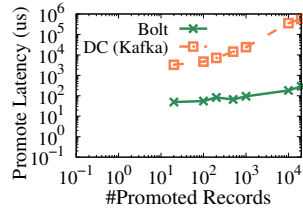


Figure 13. Promote Latency.

functionality limitations (i.e., no stateful validation, no linearizable interleaving of agentic and non-agentic records) the data-copy-based approach also suffers from high latencies.

6.9 Evaluation with Real Agentic Applications

We demonstrate Bolt’s benefits for real agentic tasks by building three agentic applications: an ad-hoc analytics agent for IoT sensor data, a stream processor testing agent, and a supply-chain restocking agent. Together, these applications exercise all AgileLog interface calls. All applications follow a single-agent architecture, and are built atop OpenCode [95] and use the Gemini-2.5-Pro LLM [40] for reasoning.

Analytics Agent (sFork). We build an agent to perform ad-hoc analysis of an IoT stream that has records with different metrics (temperature, humidity) and sensor status. The agent is given an open-ended task: “look for anomalies in the first 1M records”. The agent is provided a tool to read records from the shared log and a SQL tool [102] to analyze the read data. The agent can issue multiple parallel tool calls, each investigating a different hypothesis. We pre-populate 1M records in the stream with some anomalous values. The agent runs alongside a latency-critical (lc) workload that monitors newly ingested sensor readings. With Bolt, the agent creates an sFork to read records from. With Kafka, the agent directly queries the data stream. We find that the agent can form complex hypotheses, discover injected spikes, and correlate them with metrics. For determinism, we capture one run of the agent and replay the trace on both systems.

Figure 14(a) and (b) show the lc-workload’s e2e latencies. The shaded regions are the periods where the agent reads the stream, with annotations (nQ) indicating n parallel investigations; the non-shaded regions are the thinking phases, where the agent interacts with the LLM to evaluate results and plan next steps. With Kafka, the lc-workload suffers from latency spikes during query execution due to contention. In

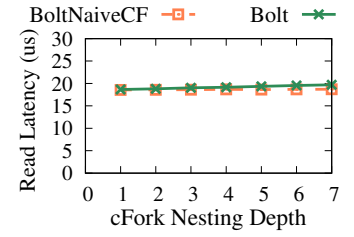


Figure 11. Metadata Lookup vs. cFork Depth.

Bolt, lc-workload experiences no interference. Figure 14(c) shows the latencies during the two phases. In Kafka, in the query-execution phase, the lc-workload experiences 14× and 130× higher mean and p99 latencies, respectively, compared to the thinking phase. Bolt has no interference.

Stream Processor Testing Agent (non-promotable cFork).

We build a stream processor that performs aggregates in tumbling windows [101] of 5ms. We task an agent with testing this stream processor under corner cases such as late, malformed, and duplicate records, while running alongside a real-time workload. The agent is provided a tool that can create non-promotable cForks of the root log, inject agent-generated test records into them, run the stream processor on the cFork, squash the cFork, and output execution traces (errors/failures). The agent can invoke this tool in parallel to run multiple tests concurrently. We note that this agent successfully triggers many buggy scenarios. Figure 15 plots e2e latency of the real-time workload with agentic tests running alongside. Shaded regions are test-execution phases (with annotation showing number of parallel tests); non-shaded regions are thinking phases. Despite tests running alongside, the real-time workload experiences no latency degradation. This shows that cForks provide a sandboxed environment with real data without impacting workloads on the root log.

Supply-Chain Agent (cFork + Promote/Squash).

We build a supply-chain agent [56]. The supply-chain stream contains order events for items and restock events that replenish supply. The order events are ingested by non-agentic producers. Based on historical trends, the agent proactively issues restock events for high-demand items. This stream is processed by a non-agentic downstream application. In Bolt, the agent is given a tool that creates a *promotable* cFork, adds restock events into it, and validates it by running a stateful copy of the downstream application on it, before promoting or squashing the cFork. In Kafka, the agent directly writes restock events to the main stream. To simulate an agent mistake, we inject schema errors into an agent-created event.

Figure 16 plots the downstream processor’s throughput. In Kafka, an agent mistake crashes the consumer application. Bolt avoids crashes through the promotable cFork, which helps safely validate agentic writes before incorporating them into the main stream. Since Bolt prevents reads from the main stream when a promotable cFork exists, there are throughput dips. However, once the cFork is either promoted

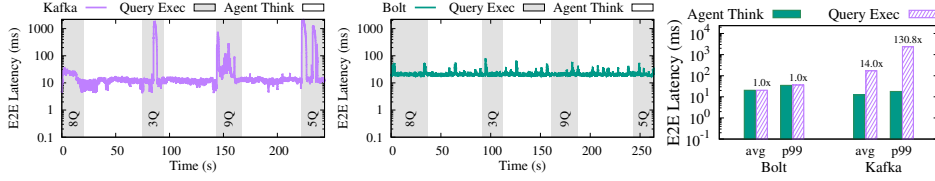


Figure 14. Ad-hoc Analytics Agent: Performance Isolation for a Latency-Critical Workload

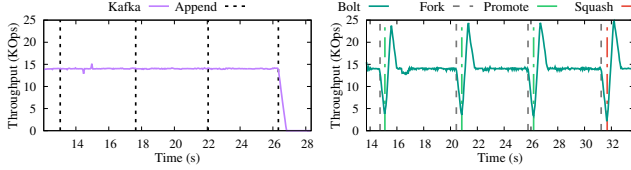


Figure 16. Supply-Chain Agent: Consumer Throughput.

or squashed, it unblocks the downstream consumer, allowing it to quickly catch up. Optionally, once the agent is thoroughly vetted, it can be allowed to directly write to the main stream and avoid the throughput dips. Promotable cForks enable the safe deployment of agents that write to production streams without worrying about application failures.

6.10 Rogue Agents

We now evaluate how Bolt behaves with rogue agents. In this experiment, we simulate a rogue agent that blocks forever after creating a promotable cFork. Figure 17(a) shows that without Bolt’s lease-based internal squash mechanism (see §5.6), such an agent can permanently affect the throughput of a consumer on the root log since reads beyond the fork point are blocked. In figure 17(b), we see that while consumer throughput initially drops, Bolt’s lease-based mechanism squashes promotable cForks internally after a configurable timeout (in this case 500ms) and parent consumer throughput heals. This prevents permanent harm to production workloads from misbehaving agents.

7 Related Work

Shared Logs. Prior shared logs change the interface for virtualization [34], low-latency [36, 86, 87], and flexible ordering [58, 85]. However, none provide the forking capability. While FuzzyLog [85] *implicitly* allows forked histories during network partitions, forking is not a first-class primitive and is not designed for agentic use. To our knowledge, AgileLog is the first shared log abstraction with forking as a core primitive to support agents. Like diskless designs, prior shared logs like Scalog [51] and Boki [70] also separate data and metadata. However, their storage servers are stateful (use local disks), which makes them prone to storage contention like Kafka (§6.2). Bolt avoids this via its diskless design.

Fork as a Core Primitive. Beginning with OS process forks, a fork as a first-class operation has been well explored in

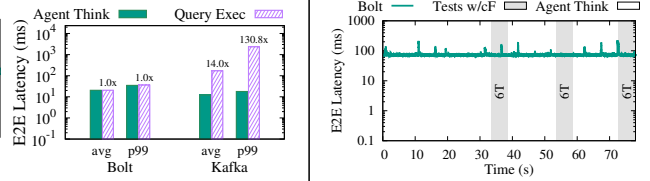
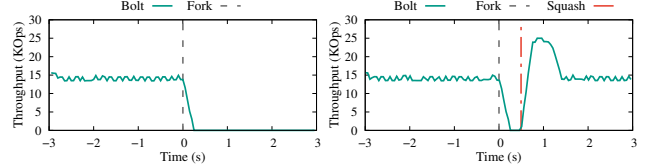


Figure 15. Real-time E2E Latency w/ Testing Agent.



(a) Without Lease-based Squashing (b) With Lease-based Squashing

Figure 17. Behaviour with Rogue Agent.

many contexts [24, 76, 128, 137]. Prior storage systems have also proposed forking (sometimes referred to as branching) primitives [21, 46]. More recently, to support AI agents, a few databases [6, 48, 84], object stores [10, 11], lakehouses [122], and file-systems [132] have realized the need for and adopted forking/branching as a first-class primitive. However, no shared log today provides forks. Further, unlike these systems, AgileLog provides a novel form of continuous forks.

Fork vs. Branch. Branching, unlike forking, is desired when divergent paths are intended to be merged [114]. A few databases such as Dolt [9] offer mergeable branches, but most do not [115]. Tardis [46] is an earlier system that uses a branch-on-conflict mechanism with merge capability for weakly-consistent systems. AgileLog’s promote is a restricted merge where updates from a single forked child are effectively merged into the parent. A more general notion of merge that can merge updates from multiple forks into the parent would not preserve linearizable ordering of appends. Further, our motivating use cases did not require it.

8 Conclusion

AgileLog is a new shared log that offers forking as a core primitive, enabling agents to operate over forks of a shared-log-based stream in a safe and isolated manner. As AI agents increasingly operate over data systems, it is imperative to build systems abstractions that enable safe agent interactions without interference to traditional applications. Our work takes a step in this direction for streaming data systems.

Acknowledgments. We thank our shepherd and SOSP’26 reviewers for their insightful comments. We thank Madhav Jivrajani for helping us conduct early experiments. We thank the other DASSL members and Xudong Sun for their feedback. This material was supported by funding from NSF grants CNS-2340218, CNS-2339784, IIS 2551556, IIDAI grants, as well as gifts from NetApp and TigerBeetle.

References

- [1] A Model Context Protocol (MCP) server for Apache Kafka. <https://github.com/tuannvm/kafka-mcp-server>.
- [2] Apache Flink: Stateful Computations over Data Streams. <https://flink.apache.org/>. Apache Software Foundation.
- [3] Apache Pulsar Functions Overview. <https://pulsar.apache.org/docs/next/functions-overview/>. Apache Software Foundation.
- [4] Branching in Supabase to Test and Preview Changes. <https://supabase.com/docs/guides/deployment/branching>.
- [5] Branching Workflows to Test Queries in Neon. <https://neon.com/docs/guides/branching-test-queries>.
- [6] Build Versioning / Checkpoints for your Agent. <https://neon.com/branching/branching-for-agents>.
- [7] Cloudlab Hardware. <https://docs.cloudlab.us/hardware.html>.
- [8] Confluent, Inc. <https://en.wikipedia.org/wiki/Confluent>. Accessed: 2025-08-28.
- [9] Dolt is Git for Data! <https://github.com/dolthub/dolt>.
- [10] Fluid Storage: Forkable, Ephemeral, Durable Infrastructure for the Age of Agents. <https://www.tigerdata.com/blog/fluid-storage-forkable-ephemeral-durable-infrastructure-age-of-agents>.
- [11] Fork Buckets Like You Fork Code | Tigris Object Storage. <https://www.tigrisdata.com/blog/fork-buckets-like-code/>.
- [12] Forks of a Database with Dolt. <https://docs.dolthub.com/concepts/dolthub/forks>.
- [13] Introducing the StreamNative MCP Server: Connecting Streaming Data to AI Agents. <https://streamnative.io/blog/introducing-the-streamnative-mcp-server-connecting-streaming-data-to-ai-agents>.
- [14] Lenses IO. <https://lenses.io/>.
- [15] Marble is Shipping Faster with Neon Branching. <https://neon.com/blog/marble-is-shipping-faster-with-neon-branching>.
- [16] Netflix Data Mesh: Streaming SQL in Data Mesh. <https://netflixtechblog.com/streaming-sql-in-data-mesh-0d83f5a00d08>.
- [17] NVIDIA Hephaestus: Building AI Agents to Automate Software Test Case Creation. <https://developer.nvidia.com/blog/building-ai-agents-to-automate-software-test-case-creation/>.
- [18] Production Testing with Dark Canaries. <https://www.linkedin.com/blog/engineering/infrastructure/production-testing-with-dark-canaries>.
- [19] StreamNative, Inc. <https://streamnative.io/>.
- [20] pulsar-mcp-server: A Model Context Protocol Server for Apache Pulsar. <https://github.com/nodece/pulsar-mcp-server>, 2025.
- [21] Marcos Kawazoe Aguilera, Susan Spence, and Alistair C Veitch. Olive: Distributed Point-in-Time Branching Storage for Real Systems. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI '06)*, San Jose, CA, May 2006.
- [22] Aiven. Diskless topics for Apache Kafka. <https://aiven.io/docs/products/kafka/diskless/concepts/diskless-topic-overview>.
- [23] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated Unit Test Improvement using Large Language Models at Meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024.
- [24] Chloe Alverti, Stratos Psomadakis, Burak Ocalan, Shashwat Jaiswal, Tianyin Xu, and Josep Torrellas. CXLfork: Fast Remote Fork over CXL Fabrics. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS '25*, 2025.
- [25] Amazon Web Services, Inc. Best practices design patterns: optimizing Amazon S3 performance. <https://docs.aws.amazon.com/AmazonS3/latest/userguide/optimizing-performance.html>.
- [26] Anthropic and Community Contributors. Model Context Protocol Specification. <https://modelcontextprotocol.io/specification>.
- [27] Apache. Kafka. <http://kafka.apache.org/>.
- [28] Apache. Pulsar. <https://pulsar.apache.org/>.
- [29] Apache Software Foundation. Kafka mirroring (mirror-maker). <https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=27846330>, 2017.
- [30] AWS. Amazon Kinesis. <https://aws.amazon.com/kinesis/>.
- [31] Ahmed Azraq and Moises Dominguez Garcia. Building an Event-Driven Agentic AI System with Apache Kafka on Confluent Cloud and watsonx Orchestrate. <https://developer.ibm.com/tutorials/event-driven-agentic-ai-system-confluent-watsonx-orchestrate/>, 2026.
- [32] Nooshin Bahador. Transparent, Evaluable, and Accessible Data Agents: A Proof-of-Concept Framework. *arXiv preprint arXiv:2509.24127*, 2025.
- [33] Gourav Singh Bais. How to detect fraudulent clicks in a real-time ad system. <https://www.redpanda.com/blog/detect-fraudulent-clicks-real-time-ads>.
- [34] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, et al. Virtual Consensus in Delos. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI '20)*, Virtual, November 2020.
- [35] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI '12)*, San Jose, CA, April 2012.
- [36] Shreesha G. Bhat, Tony Hong, Xuhao Luo, Jiyu Hu, Aishwarya Ganesan, and Ramnathan Alagappan. Low End-to-End Latency atop a Speculative Shared Log with Fix-Ante Ordering. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation (OSDI '25)*, Boston, MA, July 2025.
- [37] Caf.io. How Often Should Fraud Detection Rules be Reviewed to Maintain Accuracy? <https://www.caf.io/post/how-often-should-fraud-detection-rules-be-reviewed-to-maintain-accuracy>.
- [38] Calliope. AI-Powered Data Exploration on SQL, and NoSQL DBs. <https://calliope.ai/>.
- [39] Arda Celik and Qusay H. Mahmoud. A review of large language models for automated test case generation. *Machine Learning and Knowledge Extraction*, 7(3), 2025.
- [40] Gheorghe Comanici, Eric Bieber, Mike Schaeckermann, Ice Pasupat, Naveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.
- [41] Confluent. Confluent Streaming Agents. <https://www.confluent.io/product/streaming-agents/>.

- [42] Confluent, Inc. The Confluent Real-Time Context Engine: Power AI Agents and Apps With Instant Intelligence. <https://www.confluent.io/press-release/real-time-context-engine/>, 2025.
- [43] Confluent, Inc. Streaming agents examples and tutorials with confluent cloud. <https://docs.confluent.io/cloud/current/ai/streaming-agents/streaming-agents-examples.html#financial-fraud-detection>, 2026.
- [44] Confluent, Inc. Streaming agents examples with confluent cloud — customer support agent. <https://docs.confluent.io/cloud/current/ai/streaming-agents/streaming-agents-examples.html#customer-support-agent>, 2026.
- [45] Confluent, Inc. Streaming Agents Examples — Step 1: Create Content Analysis Agents. <https://docs.confluent.io/cloud/current/ai/streaming-agents/streaming-agents-examples.html#step-1-create-content-analysis-agents>, 2026.
- [46] Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement. Tardis: A branch-and-merge approach to weak consistency. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1615–1628, New York, NY, USA, 2016. Association for Computing Machinery.
- [47] Yashwanth Dasari. New in Confluent Cloud: Tableflow, Freight Clusters, Apache Flink AI Enhancements, and More. <https://www.confluent.io/blog/2025-q1-confluent-cloud-launch/#freight-clusters-q1>, 2025.
- [48] Tiger Data. Native Search and Retrieval on Databases with Tiger Data. <https://www.tigerdata.com/blog/postgres-for-agents>.
- [49] Databricks. State of AI Agents 2026. <https://www.databricks.com/sites/default/files/2026-01/State-of-AI-Agents-2026-Final.pdf>, 2026.
- [50] DeltaStream, Inc. Two Paths to Context: When GenAI Agents Need a Real-Time Context Engine like DeltaStream — and When They Don't. <https://www.deltastream.io/blog/two-paths-to-context-when-genai-agents-need-a-real-time-context-engine-like-deltastream-and-when-they-dont/>, 2025.
- [51] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log. In *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI '20)*, Santa Clara, CA, February 2020.
- [52] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.
- [53] Uber Engineering. uReplicator: Improvement of apache kafka mirrormaker. <https://github.com/uber/uReplicator>, 2016.
- [54] Fabi. Fabi: AI Data Analysts. <https://www.fabi.ai/product/analyst-agent>.
- [55] Sean Falconer. Introducing Real-Time Context Engine: Simplified Context Engineering With Real-Time, Processed Data for AI. <https://www.confluent.io/blog/introducing-real-time-context-engine-ai/>.
- [56] Matthew Finio and Amanda Downie. AI Agents in Supply Chain. <https://www.ibm.com/think/topics/ai-agents-supply-chain>, 2024.
- [57] Flagsmith. How to Test Safely in Production with Feature Flags? <https://www.flagsmith.com/blog/testing-in-production>.
- [58] Dimitra Giantsidi, Emmanouil Giortamis, Nathaniel Tornow, Florin Dinu, and Pramod Bhatotia. FlexLog: A Shared Log for Stateful Serverless Computing. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, pages 195–209, 2023.
- [59] Ioana Giurgiu and Michael E. Nidd. Supporting Dynamic Agentic Workloads: How Data and Agents Interact. *arXiv preprint arXiv:2512.09548*, 2025.
- [60] Ken Goodhope, Joel Koshy, Jay Kreps, Neha Narkhede, Richard Park, Jun Rao, and Victor Yang Ye. Building LinkedIn's Real-time Activity Data Pipeline. *IEEE Data Eng. Bull.*, 35(2):33–45, 2012.
- [61] Google. Pub/Sub. <https://cloud.google.com/pubsub>.
- [62] Thibaut Gourdel. Agents meet databases: The future of agentic architectures. <https://thenewstack.io/agents-meet-databases-the-future-of-agentic-architectures/>, 2025.
- [63] Confluent Technology Strategy Group. 2026 Confluent Predictions Report. <https://assets.confluent.io/m/62a1ef550db9e6a9/original/2026-Predictions-Program-DPF-EB.pdf>, 2026.
- [64] Jelani Harper. Confluent's Real-Time Agents Build on Kafka Streaming Data. <https://thenewstack.io/confluents-real-time-agents-build-on-kafka-streaming-data/>, 2025.
- [65] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring Massive Multitask Language Understanding. *arXiv preprint arXiv:2009.03300*, 2020.
- [66] Monika Rauch Henzinger and Valerie King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '95, page 519–527, New York, NY, USA, 1995. Association for Computing Machinery.
- [67] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3), July 1990.
- [68] Xinyi Hou, Yanjie Zhao, Shenao Wang, and Haoyu Wang. Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions. *arXiv preprint arXiv:2503.23278*, 2025.
- [69] Inscribe.ai. How we continuously update our fraud detectors. <https://www.inscribe.ai/blog/how-we-continuously-update-our-fraud-detectors>.
- [70] Zhipeng Jia and Emmett Witchel. Boki: Stateful Serverless Computing with Shared Logs. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP '21)*, Virtual, October 2021.
- [71] Jared Jones and Yves Laurent. Unleash real-time agentic ai with streaming agents on confluent cloud and couchbase. <https://www.couchbase.com/blog/real-time-agentic-ai-confluent-cloud>.
- [72] Kafka. KIP-1150: Diskless Topics for Kafka. <https://cwiki.apache.org/confluence/display/KAFKA/KIP-1150%3A+Diskless+Topics>.
- [73] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI '19)*, Boston, MA, February 2019.
- [74] Martin Kleppmann and Jay Kreps. Kafka, samza and the unix philosophy of distributed data. *IEEE Data Eng. Bull.*, 38:4–14, 2015.
- [75] Jay Kreps. The log: What every software engineer should know about real-time data's unifying abstraction. <https://www.linkedin.com/blog/engineering/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying>, 2013.

- [76] John Alistair Kressel, Hugo Lefevre, and Pierre Olivier. μ Fork: Supporting POSIX fork Within a Single-Address-Space OS. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pages 18–35, 2025.
- [77] Leslie Lamport. Paxos Made Simple. *ACM Sigact News*, 32(4):51–58, 2001.
- [78] James Leng. Safe Agent Writes in Workbench Demo. <https://www.dolthub.com/blog/2025-11-25-safe-agent-writes-video/>, 2025.
- [79] Lenses.io. AI-Assisted Engineering for Data Streaming with Lenses MCP. <https://lenses.io/kafka-ai>.
- [80] lenses.io. Lenses MCP: A New Era in AI Enablement for Streaming Application Development. <https://lenses.io/blog/2025/10/lenses-mcp-new-era-in-ai-enablement-for-streaming-app-dev/>.
- [81] Lenses.io. Kafka to kafka replication - lenses k2k. <https://lenses.io/kafka-replication/>, 2024.
- [82] Beibin Li, Konstantina Mellou, Bo Zhang, Jeevan Pathuri, and Ishai Menache. Large language models for supply chain optimization. *arXiv preprint arXiv:2307.03875*, 2023.
- [83] Christina Lin. Real-time predictions for ML apps with Redpanda Data Transforms (powered by WASM). <https://www.redpanda.com/blog/real-time-predictions-machine-learning-applications-wasm>, April 2024.
- [84] Shu Liu, Soujanya Ponnappalli, Shreya Shankar, Sepanta Zeighami, Alan Zhu, Shubham Agarwal, Ruiqi Chen, Samion Suwito, Shuo Yuan, Ion Stoica, Matei Zaharia, Alvin Cheung, Natacha Crooks, Joseph Gonzalez, and Aditya G. Parameswaran. Supporting Our AI Overlords: Redesigning Data Systems to be Agent-First. In *16th Conference on Innovative Data Systems Research (CIDR '26)*, 2026.
- [85] Joshua Lockerman, Jose M Faleiro, Juno Kim, Soham Sankaran, Daniel J Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. The fuzzylog: A partially ordered shared log. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*, Carlsbad, CA, October 2018.
- [86] Xuhao Luo, Shreesha G. Bhat, Jiyu Hu, Ramnatthan Alagappan, and Aishwarya Ganesan. LazyLog: A New Shared Log Abstraction for Low-Latency Applications. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP '24)*, Austin, TX, November 2024.
- [87] Xuhao Luo, Shreesha G. Bhat, Jiyu Hu, Ramnatthan Alagappan, and Aishwarya Ganesan. LazyLog: A New Shared Log Abstraction and Design for Modern Low-Latency Applications. *ACM Transactions on Computer Systems*, 2025.
- [88] Maintainers of cp-algorithms. Segment Trees: Range Updates (Lazy Propagation). https://cp-algorithms.com/data_structures/segment_tree.html#range-updates-lazy-propagation.
- [89] Matteo Merli, Sijie Guo, Penghui Li, Hang Chen, and Neng Lu. Ursa: A lakehouse-native data streaming engine for kafka. *Proc. VLDB Endow.*, 18(12):5184–5196, August 2025.
- [90] Microsoft. Build AI Agents with Azure Database for PostgreSQL and Azure AI Agent Service. <https://techcommunity.microsoft.com/blog/adforpostgresql/build-ai-agents-with-azure-database-for-postgresql-and-azure-ai-agent-service/4392616>, 2025.
- [91] Microsoft. Azure Blob Storage documentation. <https://learn.microsoft.com/azure/storage/blobs/>, 2026.
- [92] MinIO. MinIO: A high-performance, S3-compatible object storage. <https://github.com/minio/minio>.
- [93] Micah Murray, Wen Zhang, Aisha Mushtaq, Natacha Crooks, Aurojit Panda, and Scott Shenker. Designing a Datacenter-wide Distributed Shared Log. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*, pages 195–201, 2025.
- [94] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA, June 2014.
- [95] OpenCode. OpenCode: The open source AI coding agent. <https://opencode.ai/>.
- [96] Artem Oppermann. Detecting fraud in real time using Redpanda and Pinecone. <https://www.redpanda.com/blog/fraud-detection-pipeline-redpanda-pinecone>.
- [97] Juan Altmayer Pizzorno and Emery D Berger. Coverup: Effective high coverage test generation for python. *arXiv preprint arXiv:2403.16218*, 2024.
- [98] PowerDrill. Enterprise use-cases for Agentic AI. <https://powerdrill.ai/blog/ai-agents-for-data-analysis-and-visualization>.
- [99] Pravega. Pravega – A Reliable Stream Storage System. <https://pravega.io/>.
- [100] Shyam Purkayastha. Building a real-time data processing pipeline for IoT. <https://www.redpanda.com/blog/analyzing-iot-telemetry-data-apache-spark>.
- [101] Quix. A guide to windowing in stream processing. <https://quix.io/docs/blog/2024/03/13/windowing-stream-processing-guide.html>.
- [102] Mark Raasveldt and Hannes Mühleisen. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD '19*, page 1981–1984, New York, NY, USA, 2019. Association for Computing Machinery.
- [103] RedPanda. Redpanda: The Agentic Data Plane. <https://www.redpanda.com/>.
- [104] RedPanda. Remote MCP Server Overview. <https://web.archive.org/web/20251115205905/https://docs.redpanda.com/redpanda-cloud/ai-agents/mcp/remote/overview/#use-cases>.
- [105] Redpanda Data. Kafka Broker: Architecture, Functions, and Concepts. <https://www.redpanda.com/guides/kafka-architecture-kafka-broker>, 2025.
- [106] Redpanda Data, Inc. Redpanda Data – streaming data platform compatible with Apache Kafka. <https://www.redpanda.com/>.
- [107] Redpanda Documentation. Model context protocol (mcp). <https://docs.redpanda.com/redpanda-cloud/ai-agents/mcp/>.
- [108] Redpanda Documentation. Mcp tool patterns: Generate test data. <https://web.archive.org/web/20260209025737/https://docs.redpanda.com/redpanda-cloud/ai-agents/mcp/remote/tool-patterns/#data-generators>, 2026.
- [109] Redpanda, Inc. MCP Tool Patterns & Stream Processing in Redpanda Cloud Remote MCP. <https://web.archive.org/web/20260209025737/https://docs.redpanda.com/redpanda-cloud/ai-agents/mcp/remote/tool-patterns/#stream-processing>, 2026.
- [110] Salesforce. Mirus: Cross-data-center replication for apache kafka. <https://github.com/salesforce/mirus>, 2018.
- [111] Craig Sanchez. Agentic ai meets real-time data with streaming agents. <https://www.vectara.com/blog/agentic-ai-meets-real-time-data-with-streaming-agents>.
- [112] Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language Models Can Teach Themselves to Use Tools. In *Advances in Neural Information Processing Systems (NeurIPS '23)*, volume 36, 2023.
- [113] Fred B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.
- [114] Tim Sehn. So you want Database Forks? <https://www.dolthub.com/blog/2022-07-29-database-forks/>, 2022.

- [115] Tim Sehn. So you Want Database Branches? <https://www.dolthub.com/blog/2024-09-18-database-branches/>, 2024.
- [116] Tim Sehn. Agent Mode in the Dolt Workbench. <https://www.dolthub.com/blog/2025-11-12-agentic-dolt-workbench>, 2025.
- [117] Tim Sehn. Dolt for Agentic Workflows. <https://www.dolthub.com/blog/2025-03-17-dolt-agentic-workflows/>, 2025.
- [118] Tim Sehn. So you want an AI Database? <https://www.dolthub.com/blog/2025-12-09-ai-database/>, 2025.
- [119] Tim Sehn. So you want an AI Database? Isolation. <https://www.dolthub.com/blog/2025-12-09-ai-database/#isolation>, 2025.
- [120] Amazon Web Services. Amazon s3. <https://aws.amazon.com/s3/>.
- [121] StreamNative. StreamNative MCP Server: Create, Deploy, and Test Python Pulsar Functions Using an LLM. <https://www.youtube.com/watch?v=9JDHL-WaCXs>.
- [122] Jacopo Tagliabue and Ciro Greco. Safe, Untrusted, “Proof-Carrying” AI Agents: Toward the Agentic Lakehouse. *arXiv preprint arXiv:2510.09567*, 2025.
- [123] Midhat Tilawat. What are Context-Aware Agents in AI? <https://www.allaboutai.com/ai-glossary/context-aware-agents/>, 2025.
- [124] Tricentis. Testing in production. <https://www.tricentis.com/learn/production-testing>, 2026.
- [125] Typedef Team. Agentic Analytics on BigQuery: Turning SQL Warehouses into AI Analysts. <https://www.typedef.ai/resources/agentic-analytics-bigquery>, December 2025.
- [126] Rajkumar Venkatasamy. Building a real-time search application with Redpanda and ZincSearch. <https://www.redpanda.com/blog/real-time-data-search-redpanda-zincsearch>.
- [127] Daniel De Vera. Using Sandboxes in Kafka-based Asynchronous Applications. <https://www.signadot.com/blog/using-sandboxes-in-kafka-based-asynchronous-applications/>, 2024.
- [128] Manuel Vögele, Christopher Thomas, and Timo Hönig. Spork: A posix_spawn you can use as a fork. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*, HotOS ’25, page 96–102, 2025.
- [129] Ostap Vykhopen, Viktoria Skorik, Maxim Tereschenko, and Veronika Solopova. Beyond Text-to-SQL: Autonomous Research-Driven Database Exploration with DAR. *arXiv preprint arXiv:2512.14622*, 2025.
- [130] Kai Waehner. Real-Time Model Inference with Apache Kafka and Flink for Predictive AI and GenAI. <https://www.kai-waehner.de/blog/2024/10/01/real-time-model-inference-with-apache-kafka-and-flink-for-predictive-ai-and-genai/>, 2024.
- [131] Kai Waehner. The Rise of Diskless Kafka: Rethinking Brokers, Storage, and the Kafka Protocol. <https://www.kai-waehner.de/blog/2025/08/11/the-rise-of-diskless-kafka-rethinking-brokers-storage-and-the-kafka-protocol/>, 2025.
- [132] Cong Wang and Yusheng Zheng. Fork, Explore, Commit: OS Primitives for Agentic Exploration. *arXiv preprint arXiv:2602.08199*, 2026.
- [133] 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. Consistency and completeness: Rethinking distributed stream processing in apache kafka. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD ’21, page 2602–2613, New York, NY, USA, 2021. Association for Computing Machinery.
- [134] WarpStream. The Diskless, Kafka Compatible Data Streaming Platform. <https://www.warpstream.com/>.
- [135] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems (NeurIPS ’22)*, volume 35, 2022.
- [136] 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. vCorfu: A Cloud-Scale object store on a shared log. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 35–49, Boston, MA, March 2017. USENIX Association.
- [137] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. No Provisioned Concurrency: Fast RDMA-coded Remote Fork for Serverless Computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 497–517, Boston, MA, July 2023. USENIX Association.
- [138] Wikipedia contributors. Euler Tour Technique. https://en.wikipedia.org/wiki/Euler_tour_technique, 2025.
- [139] Willem Thiart, and other open-source contributors. willem/raft. <https://github.com/willem/raft>.
- [140] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations (ICLR ’23)*, 2023.
- [141] Filip Yonov. The 3 Es of Diskless Kafka BYOC. <https://aiven.io/blog/diskless-apache-kafka-guide-byoc>, 2025.
- [142] Yazhuo Zhang, Jinqing Cai, Avani Wildani, and Ana Klimovic. Rethinking Web Cache Design for the AI Era. In *Proceedings of the 2025 ACM Symposium on Cloud Computing (SoCC ’25)*, 2025.
- [143] Yunji Zhong, Amit Gud, and Carlos Herrera. Setting up Kafka Multi-Tenancy. <https://careersatdoordash.com/blog/setting-up-kafka-multi-tenancy/>, 2024.
- [144] Zhiting Zhu, Zhipeng Jia, Newton Ni, Dixin Tang, and Emmett Witchel. Impeller: Stream Processing on Shared Logs. In *Proceedings of the EuroSys Conference (EuroSys ’25)*, Rotterdam, Netherlands, April 2025.