# Agents for Data Streaming Tasks: The Missing Pieces

Shreesha G. Bhat*
UIUC

Landon Johnson*
UIUC

Michael Noguera*
UIUC

Aishwarya Ganesan
UIUC

Ramnatthan Alagappan
UIUC

## Abstract

Agents are emerging as the primary users of data systems. Yet, an important class of data systems has not evolved for agents: *real-time data-streaming systems*. To make streaming systems agent-first, the real-time data stack requires new infrastructure abstractions that let agents safely experiment on production dataflows via forking and merging. There is also a need for new benchmarks to evaluate agents acting on streaming data and agent-first streaming systems. The agent-first infrastructure and benchmarking needs of streaming systems have unique requirements and challenges compared to other agentic data systems. We outline our initial steps toward addressing these challenges and discuss open problems.

## Keywords

stream processing, real-time analytics, benchmarks, agentic systems

## 1 Introduction

AI agents are increasingly operating on data systems alongside traditional programs. Agents use the reasoning capabilities of LLMs and interact with external systems through tool calling [54] to perform complex data tasks specified in natural language.

Agents, as shown by prior work [34, 42], operate in ways that differ markedly from that of traditional workloads. Rather than following optimized access patterns, agents issue highly speculative and exploratory queries, resulting in sub-optimal system interactions. Agents can also make mistakes because the LLM-informed actions can be wrong. Supporting these agents thus requires rethinking the underlying systems from an agent-first perspective.

An orthogonal but equally important aspect is the development of rich, domain-specific benchmarks that evaluate agents on domain-specific data tasks. Such benchmarks play a dual role: they aid application developers to assess and improve their agents, while also offering system designers insight into how infrastructure must evolve to effectively support agent-driven workloads.

Many databases [1, 5, 30], object stores [3, 4], lakehouses [56], and filesystems [41, 57] have started embracing agents as first-class citizens and proposing agent-first primitives. Similarly, there are many expressive benchmarks that evaluate agents on data tasks such as Text-to-SQL [37, 39], general data tasks across many data systems [40, 46, 59, 62] and filesystem usage [43].

However, an important class of systems that are indispensable for modern, real-time applications have not fully evolved for the agentic era: *data-streaming systems*. Streaming systems [2, 11, 53] enable the management and processing of streams of real-time data. These data streams can be composed with stream processors [7, 12, 33]—entities that process one or more input streams to produce output streams—to create complex data pipelines [8] that enable organizations to glean insights and take actions in real-time. Data streaming plays a vital role in many domains like finance [17, 27, 28, 31, 50], supply chain management, and IoT [26, 51].

While many data-streaming companies today *aim to support* agentic AI use cases [6, 29, 38, 52], we foresee challenges on two fronts: deficiencies within core streaming infrastructure and the lack of domain-specific benchmarks for data-streaming tasks.

The streaming infrastructure consists of two core components: *shared logs* [18, 20, 32, 45], which ingest, durably store, and serve streams; and *stream processors*, which consume data streams, update state or perform computations, and output data to other streams. While streaming systems today allow agents to read/write streams, and to create/deploy stream processors, this is *insufficient*. Today's streaming systems lack the fundamental mechanisms to support agents. For instance, they cannot avoid the performance interference on streams and handle bursty spin-up/tear-down of stream processors, which arise from the exploratory nature of agents. Likewise, today's systems cannot stage and statefully validate agentic writes, nor support tasks like testing or counterfactual "what-if" analysis that inject synthetic events and observe their propagation through the pipeline without affecting production workloads.

Second, benchmarks for agentic tasks over streaming data are lacking. Existing benchmarks assess an agent's ability to write SQL and interact with data at rest, but none evaluate writing *real-time queries* over streams. Such queries fundamentally require different constructs as time-based windowing and state maintenance, skills which do not necessarily translate from other tasks. Beyond generating processors or simple analyses, benchmarks must test an agent's ability to reason about entire pipelines, for example, answering counterfactuals like: "How would a 5% increase in failure rates across the supply chain affect shipping throughput under current operating conditions?" Finally, the benchmark must also include metrics tailored to streaming tasks (e.g., robustness to late data).

This position paper describes our initial directions in bridging the gap in streaming infrastructure to support agents and lays out the open challenges (§3). We also highlight the missing pieces for benchmarking agents on streaming tasks and our initial directions towards such a benchmark (§4).

## 2 Background and Use Cases

Streaming systems are widely deployed to store and process huge amounts of real-time data. Thus, there is much to gain from connecting agents to the streaming ecosystem. We envision three modalities for the use of agents (Figure 1). First, developers may use coding agents to create stream processors (§2.1). Second, non-technical users may use chat-based agents to perform tasks on real-time data (§2.2). Third, an autonomous inline agent may itself be used as a stream processor, invoking LLMs to process events (§2.3).

*equal contribution

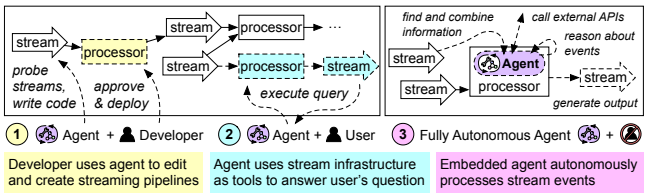


**Figure 1: Agents and Streams: Use Cases.**

### 2.1 Developer Use of Agents

Developers may use existing coding agents to write stream processors. Typically this is an iterative process with the agent editing code with developer guidance [19]. Coding agents heavily use tools [19] and probe data sources with exploratory queries [34, 42]. Multiple versions of the code are tested, then once the developer is satisfied they deploy the final version.

**Requirements:** To support this use case, streaming systems must provide agents with the ability to probe data and test code without negative consequences. During iteration, coding agents' reads and writes must be isolated from production. After the code is deployed, it can be treated as traditional (trusted) stream processors.

### 2.2 Non-expert Use of Streaming Agents

Enabling agents to interact with streams unlocks a variety of use cases for non-expert users who are not developers. Rather than writing and deploying stream processor code themselves, these users can instead chat with conversational agents capable of using the interfaces presented by stream infrastructure as tools to query streams (e.g. reading events from the underlying shared log like Kafka) and performing stream computations (e.g. running a Flink SQL query). Thus with the help of agents, even non-expert users can perform intricate computations that compose streams and stream processors.

**Stream reading.** Many use cases are enabled simply by allowing agents to read from streams. Agents might *read events directly* to answer ad-hoc queries, e.g., checking the latest temperature reading. Agents may also use stream processors to run *streaming queries with continuous output*. For instance, a marketing team could build a live dashboard for real-time insight into the performance of their advertising campaign. Also, agents could aid in *root-cause analysis* by searching stream data (e.g., SREs using an agent to search for anomalous metrics and correlate them with events).

**Stream creation.** Agents can also create their own streams. They might use these *streams as output channels* for computations or to provide a filtered view of an input stream. This could be used for *continuous rule-based monitoring*, e.g., a stream processor to issue a frost warning if the temperature drops below a specified threshold.

Agent-created streams also enable testing scenarios with *what-if analysis*: agents can copy and change an input stream to observe the resulting change in a computation. Consider a distributed sensor network that is producing an inaccurate output. A user could suspect a faulty sensor, and ask the agent what the output would look like excluding that sensor. The agent could simulate this by copying and filtering the sensor stream and re-running the calculation.

**Stream editing.** Agents can be asked to manipulate existing streams and stream processors. They could *insert events to a stream* on a user's behalf. For example, an airline representative might ask an agent to re-book passengers at risk of missing connecting flights due to delays. The agent would insert events to remove the passenger from the manifest of the missed flight and book them a seat on a different connection. *Editing a running stream processor* would allow an agent to alter the logic of a data pipeline in real time. For example, in detecting fraudulent transactions, a user might ask the agent to insert new rules to an existing stream processor.

**Requirements:** Agents acting on behalf of everyday users should not be able to negatively impact production data pipelines: agents must not impact performance of production workloads and agentic writes must be isolated. At times, it *is* intended that agents modify production data (as in "stream editing" uses); even in these cases it is important to review the changes to avoid negative consequences.

### 2.3 Autonomous Agents as Stream Processors

Distinct from agents using stream infrastructure as tools, agents may themselves be used to process stream events. New frameworks like Apache Flink Agents [15] and Confluent's Streaming Agents [29] embed an LLM-based agent within a stream processor. The agent is invoked for each input event in the stream, and can perform various actions including accessing data from other streams, calling outside APIs, and writing to output streams. Such an *embedded agent* can be used to classify stream events or assign labels. It can also *take advantage of particular models' strengths*. For instance, a model fine-tuned on financial transaction data may be used to detect suspicious transactions for fraud prevention. It can also be used to *transform or enrich stream events* using information from multiple sources (e.g., for real-time search result personalization).

**Requirements:** Because embedded agents run autonomously without per-event oversight, their actions need to be staged, validated, and only then be allowed to take effect.

## 3 Streaming Infrastructure

### 3.1 Problems with Streaming Infrastructure

Today's streaming systems enable agents to read, write, and create streams and stream processors [6, 29, 38, 52], but run into many problems when supporting different agentic use cases.

**Exploratory reads cause performance interference.** Agents are exploratory in nature: they repeatedly probe the data, format, and configurations to ground themselves and create their own context, and as a means of test-time scaling [55], often explore many parallel trajectories [42] to accomplish a task. This leads to a high volume of stream reads, which can contend with production workloads on the streams. For instance, an agent analyzing anomalies could issue many exploratory reads and thus contend with a real-time alert system, affecting the end-to-end latency for alerts. With many agents operating on a streaming system simultaneously, this interference will only worsen. Current streaming systems, however, lack mechanisms to isolate production workloads from agents.

**Stream processing engines are not designed for bursty spin-up and tear-down.** Agents tasked with building streaming pipelines might try multiple pipeline architectures and evaluate them to achieve a given task. To do so, they quickly spin-up and orchestrate many stream processors, and subsequently tear-down the ones that lead to bad solutions. Today's stream processors however are not designed for such bursty workloads but are instead architected for

a few carefully-crafted, long-lived pipelines. Provisioning stream-processing compute for peak usage would also be expensive. Thus, there is a need for stream-processing engines that quickly scale up or down as needed to support these bursty workloads.

**No means to stage agentic writes.** Agents need to write to streams in many use cases (e.g., an order-fulfillment agent writes events to an order confirmation stream). Allowing agents to write directly to a production stream, however, is risky because LLM-informed actions can be incorrect. The system must therefore allow agents to write to streams in *isolation*, validate them (e.g., using a reviewer agent), and only then *merge* them back into the production stream. Further, agents may need to explore multiple write paths (e.g., different LLMs producing different outputs), pick the best one, and merge only that path. Today's streaming systems offer no such explore-validate-merge primitives for stream writes.

**No means for stateful validation.** While in some cases agentic writes can be validated by inspecting the generated records in isolation, in most cases there is a need to *statefully validate* the record along with the history in the stream. Specifically, stream writes would benefit from quick validation from stateful downstream processors. Therefore, there is a need for streaming systems to not only allow isolated writes, but enable them to flow through the state being computed by downstream processors for validation without impacting the production processors themselves.

**No isolation for synthetic writes.** Use cases like testing and what-if analysis require injecting *synthetic* data into streams or perturbing streaming data to observe how it affects downstream computations. Performing such tasks with realistic production data has immense value. However, such synthetic events must remain isolated from production streams and pipelines, but must still flow through realistic processor state derived from real data. Today's systems force agents to spin up a separate cluster with synthetic data, which loses temporal realism and is operationally expensive.

### 3.2 Initial Directions and Open Challenges

We now describe a few initial directions in our recent work, AgileLog [21], to address these problems, and outline open challenges.

#### 3.2.1 Initial Steps: AgileLog

AgileLog introduces two new primitives to the shared log abstraction to address the deficiencies of data streams: *fork* and *promote*. Together these primitives provide agentic tasks the ability to cheaply create an isolated fork of a given data stream and merge appends back into it. Our implementation, Bolt, utilizes a diskless shared log architecture [13, 35, 49, 58] and incorporates several techniques to ensure that forks are zero-data-copy, low-latency, and performance-isolated even with many such forks in the system.

**Continuous Fork (cFork).** A fork in AgileLog is a cheap, logically separate, performance-isolated child copy of an existing parent instance that an agent can read from and write to just like any shared log without affecting the parent. With forks, exploratory agentic reads happen on the child rather than production streams, avoiding performance interference. Similarly, agentic writes can be staged on a fork, isolated from the main stream and statefully validated with the history on the main stream.

A key property that distinguishes streams from prior forkable data systems is that streams are *real-time*: data continues to be ingested into the parent after the fork is taken. For an agent answering a streaming query, the fork must therefore continue to see new appends on the parent even after the fork point, rather than freezing at a snapshot as forks in prior data systems do [1, 3–5, 30]. AgileLog thus offers a novel form of a *continuous fork* that continuously inherits records appended to the parent even after the fork is created, and interleaves them linearizably with private writes on the fork. This *arms agents with the ability to create cheap, isolated copies of any data stream that can be appended to while also remaining "in sync" with the production state.*

**Promoting a cFork to incorporate writes.** Forks alone are insufficient: when an agent's writes on a fork are validated and need to take effect, the system must provide a way to *merge* the fork into the parent (production) stream. AgileLog introduces a promote call that makes a cFork essentially *become* its parent. Thus, an agent's writes can not only be validated statefully in a continuously updating fork, but can also be incorporated back into the parent's stream.

#### 3.2.2 Open Challenges

AgileLog demonstrates the value in forkable streams for agentic tasks. However, AgileLog is only an initial step towards agent-first streaming systems. Specifically, while AgileLog addresses most of the challenges within the stream-storage layer, i.e, shared logs, it does *not* address challenges in the compute layer: *stream processors and stream processing engines*. We outline the open challenges.

**Challenge 1: Forking stream processors and pipelines.** A streaming system is not just a collection of streams; it is a graph of streams connected by stream processors with their own state. To support counterfactual analysis, testing, and to enable stateful validation, the unit that must be forked is the *entire pipeline*: streams, processors, and processor state. For instance, a testing agent that verifies whether a fraud-detection processor catches a new pattern must fork both the payments stream and the processor (with its state) so that the synthetic events are evaluated in a realistic context. Similarly, a "what-if" query requires forking the input stream along with the entire pipeline that the input touches in order to replay perturbed events on it, and observe outputs at the forked processors several hops downstream, all while the production pipeline runs unaffected. Such forks are also useful for stateful validation of agentic writes. Achieving this requires solving many problems.

*How to cheaply fork stream processors?* Stream processors store their state in-memory or within embedded databases. Cheap forking requires the ability to quickly clone this state. Today's stream processors do not provide a way to create such forks. However, an initial approach could rely on the fault-tolerance mechanisms that stream processors already employ. Specifically, stream processors often back their state to remote storage (e.g., S3) to recover from failures. Thus, a fork can be created by cloning the checkpointed state and catching it up to the latest offset. We believe recent work in disaggregated state management approaches for stream processors [48] can provide a good substrate for cheap forking.

*How to realize pipeline forks efficiently and correctly?* Pipelines can be deep and complex, and at any instant a record may have been processed by some operators, be in-flight at others, and unseen by the rest. A forked pipeline must resume from a state consistent with the parent's: it should neither re-process records the parent already acted upon nor miss in-flight records. We believe an initial

| | Feature | | | | Evaluation | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Benchmark | Explor-atory | Conversa-tional | Enterprise Data | Real-Time Queries | Accuracy | Time-to-Plan | Plan Performance | Tokens | Resource Utilization |
| Spider [61] | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ |
| KaggleDBQA [36] | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ |
| BIRD [39] | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ |
| Spider 2.0 [37] | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ |
| BEAVER [24] | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ |
| CoSQL [60] | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ |
| DAB [46] | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ |
| AgentFuel [47] | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ |

**Table 1: Features and Evaluation Metrics in Existing Benchmarks**

approach can be informed by the consistent global snapshot techniques already used within some stream processing engines [23] for fault-tolerance, software patches and testing. Turning them from a fault-tolerance tool into a cheap, low-latency, agent-initiated forking primitive is, in our view, an interesting open problem.

**Challenge 2: Efficient resource management within stream processing engines.** Today's stream processing engines are optimized for a few long-lived pipelines. However, for agentic exploration, they must support quick creation and destruction of many stream processors: some of which may be created anew while others created by forking existing processors. Additionally, they must be resource efficient and scale compute gracefully based on the workload. This requires fundamentally rethinking the design assumptions within these engines. An optimization that proves particularly useful for agents is reusing computation across explorations whenever possible [42]. For instance, if many explorations perform the same initial filter and join step, and each branch does not intend to perturb *the inputs* to these operators, then forking the stream processors that perform these operations is wasteful: instead, they can simply fork the output stream. Similarly, stream processing steps that can benefit from sharing read caches can be co-hosted on the same nodes. Importantly, stream processing engines must perform such optimizations *transparently* whenever possible.

**Challenge 3: General multi-fork merge beyond promote.** AgileLog's promote intentionally acts as a restricted merge: it only allows a singular continuous fork to *replace* its parent and destroys the other promotable forks. AgileLog is designed this way to explicitly support agentic patterns where a single write path wins. Agents sometimes might want to merge *more than one* fork while providing meaningful semantics. The challenge is that these forks can have diverging histories with no way to determine the relative order of records across forks. However, merges can be meaningful in certain contexts. For instance, in a stream of financial transactions, refund records written on two forks by two different agents to two different users' accounts can safely appear in either relative order on the merged stream. Merges must therefore generalize beyond first-promote-wins and allow applications to *specify merge policies*.

## 4 Benchmarks for Agents on Streams

Table 1 summarizes past work in benchmarks for evaluating LLM and LLM-based agents' usage of data systems. Initial benchmarks used for evaluating the capabilities of LLMs (e.g. Spider, KaggleDBQA, BIRD) to interact with data systems were focused on achieving high execution accuracy across a wide number of SQL queries, covering a large space of expressible SQL. These early benchmarks had a number of limitations that following works aimed to address. BEAVER demonstrated that even agents that performed well across public databases struggled to manage complexity when operating on private *enterprise* databases. Spider 2.0 and DAB evaluate the ability of a multi-turn agent to build its own context by searching database documentation or probing the database with queries.

**New benchmarks are needed for streaming systems.** The existing benchmarks do not exercise features unique to streaming settings, such as time-based windowing or state maintenance. AgentFuel, which focuses on LLMs querying time-series data, comes closest. It proposes a method to generate time-series data, but does not evaluate agents interacting with it *as it arrives*; instead agents are tasked with wrangling the data after it is collected in a database.

Properly evaluating agent use of streaming systems will require new benchmarks designed specifically for the stream processing setting. The types of queries relevant to stream processing are naturally different, as stream processors only have access to a small amount of in-memory state that they maintain. Likewise, in stream processing, data arrives gradually and is not available all at once. This leads to unique challenges such as handling events that arrive late or out of order, as well as the need to compute efficiently so as to not fall behind the stream. Agents' ability to write queries that handle these considerations should be considered part of their performance on the benchmark. Fundamentally, the time that events occur is as relevant to a streaming system as the event contents. A benchmark built for agent use of data streaming infrastructure should consist of queries defined over replayable event *traces* that specify arrival times rather than a list of tasks to execute on a static database.

**Benchmarks should focus on metrics important to real-time stream workloads.** Agents interacting with real-time data will use streams and stream processors and must be evaluated over dimensions that are more important in this setting, such as *time-to-plan*, *plan performance*, and *resource utilization*. *Time-to-Plan*, or the time for an agent to implement its final plan, is a crucial metric in the stream processing setting where a long time spent planning may mean missing important events. Additionally, the *performance* of an agent's plan can be equally as important as its accuracy, because poor performance leads to delayed downstream processing. Because streams constitute an unbounded amount of data, an agent's plan must also be evaluated for its *resource utilization*. As argued in BranchBench [14], any evaluation of performance should also quantify the impact of providing new agent-first abstractions for the underlying data system. And, of course, the cost (*tokens*) of agentic reasoning must also be considered during evaluation.

**Benchmarks should evaluate agent's ability to do complex tasks.** In addition to stream-analytics tasks, the benchmark must evaluate the agent's ability to perform complex tasks like what-if analysis. It must also evaluate the agent's ability to perform tasks that edit and create streams. In these complex tasks, the agent not only reads but also modifies the underlying data system.

**Benchmarks should be grounded in enterprise data and use cases.** While we anticipate that our new streaming abstractions will make it easier to evaluate agents on real production data streams, we also call for the release of enterprise-quality time-series datasets as existing stream processing benchmarks [9, 10, 16, 22, 25, 44] do not include suitable semantically rich data or queries. In this capacity, we have already begun discussions with industry partners.

## References


[1] Build Versioning / Checkpoints for your Agent. https://neon.com/branching/branching-for-agents.

[2] Confluent, Inc. https://en.wikipedia.org/wiki/Confluent. Accessed: 2025-08-28.

[3] 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.

[4] Fork Buckets Like You Fork Code | Tigris Object Storage. https://www.tigrisdata.com/blog/fork-buckets-like-code/.

[5] Forks of a Database with Dolt. https://docs.dolthub.com/concepts/dolthub/forks.

[6] 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.

[7] Kafka Streams. https://kafka.apache.org/42/streams/.

[8] Netflix Data Mesh: Streaming SQL in Data Mesh. https://netflixtechblog.com/streaming-sql-in-data-mesh-0d83f5a00d08.

[9] Nexmark: A Benchmark Suite for Queries over Continuous Data Streams. https://github.com/nexmark/nexmark.

[10] OpenMessaging Benchmarks. https://openmessaging.cloud/docs/benchmarks/.

[11] StreamNative, Inc. https://streamnative.io/.

[12] Structured Streaming Programming Guide in Apache Spark. https://spark.apache.org/docs/latest/streaming/index.html.

[13] Aiven. Aiven Inkless: Diskless Topics for Apache Kafka. https://aiven.io/inkless.

[14] Elaine Ang, Sam Weldon, In Keun Kim, Kevin Durand, Kostis Kaffes, and Eugene Wu. Branchbench: Aligning database branching with agentic demands. 2026.

[15] Apache Flink. Apache Flink Agents Documentation. https://nightlies.apache.org/flink/flink-agents-docs-release-0.2.

[16] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, page 480–491. VLDB Endowment, 2004.

[17] 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.

[18] 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.

[19] Joachim Baumann, Vishakh Padmakumar, Xiang Li, John Yang, Diyi Yang, and Sanmi Koyejo. Swe-chat: Coding agent interactions from real users in the wild, 2026.

[20] Shreesha G. Bhat, Tony Hong, Xuhao Luo, Jiyu Hu, Aishwarya Ganesan, and Ramnatthan 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.

[21] Shreesha G Bhat, Tony Hong, Michael Noguera, Ramnatthan Alagappan, and Aishwarya Ganesan. Agilelog: A forkable shared log for agents on data streams. *arXiv preprint arXiv:2604.14590*, 2026.

[22] Maycon Viana Bordin, Dalvan Griebler, Gabriele Mencagli, Cláudio F. R. Geyer, and Luiz Gustavo L. Fernandes. Dspbench: A suite of benchmark applications for distributed data stream processing systems. *IEEE Access*, 8:222900–222917, 2020.

[23] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in apache flink®: consistent stateful distributed stream processing. 10(12):1718–1729, August 2017.

[24] Peter Baile Chen, Fabian Wenz, Yi Zhang, Devin Yang, Justin Choi, Nesime Tatbul, Michael Cafarella, Çağatay Demiralp, and Michael Stonebraker. Beaver: an enterprise benchmark for text-to-sql. *arXiv preprint arXiv:2409.02038*, 2024.

[25] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, and Paul Poulosky. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, page 1789–1792, May 2016.

[26] Conduktor. IoT Data Streaming Architectures. https://www.conduktor.io/glossary/iot-data-streaming-architectures.

[27] Conduktor. Streaming Data in Financial Services. https://www.conduktor.io/glossary/streaming-data-in-financial-services.

[28] Confluent. 5 Ways Data Streaming is Fueling Financial Services Transformation. https://www.confluent.io/resources/ebook/5-data-streaming-use-cases-in-financial-services/.

[29] Confluent. Confluent Streaming Agents. https://www.confluent.io/product/streaming-agents/.

[30] Tiger Data. Native Search and Retrieval on Databases with Tiger Data. https://www.tigerdata.com/blog/postgres-for-agents.

[31] Alex Davies. Jump Trading Drives Faster Insights at Scale with Redpanda. https://thenewstack.io/jump-trading-drives-faster-insights-at-scale-with-redpanda/.

[32] 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.

[33] Apache Flink. Apache Flink. https://flink.apache.org/.

[34] Ioana Giurgiu and Michael E. Nidd. Supporting Dynamic Agentic Workloads: How Data and Agents Interact. *arXiv preprint arXiv:2512.09548*, 2025.

[35] Kafka. KIP-1150: Diskless Topics for Kafka. https://cwiki.apache.org/confluence/display/KAFKA/KIP-1150%3A+Diskless+Topics.

[36] Chia-Hsuan Lee, Oleksandr Polozov, and Matthew Richardson. Kaggledbqa: Realistic evaluation of text-to-sql parsers. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, page 2261–2273, Online, August 2021. Association for Computational Linguistics.

[37] Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, et al. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. *arXiv preprint arXiv:2411.07763*, 2024.

[38] 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/.

[39] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36, 2024.

[40] Jinyang Li, Xiaolong Li, Ge Qu, Per Jacobsson, Bowen Qin, Binyuan Hui, Shuzheng Si, Nan Huo, Xiaohan Xu, Yue Zhang, et al. Swe-sql: Illuminating llm pathways to solve user sql issues in real-world applications. *arXiv preprint arXiv:2506.18951*, 2025.

[41] Junxuan Liao, Jing Liu, Mai Zheng, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, et al. Don't let ai agents yolo your files: Shifting information and control to filesystems for agent safety and autonomy. *arXiv preprint arXiv:2604.13536*, 2026.

[42] Shu Liu, Soujanya Ponnapalli, Shreya Shankar, Sepanta Zeighami, Alan Zhu, Shubham Agarwal, Ruiqi Chen, Samion Suwito, Shuo Yuan, Ion Stoica, et al. Supporting Our AI Overlords: Redesigning Data Systems to be Agent-First. *arXiv preprint arXiv:2509.00997*, 2025.

[43] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023.

[44] Ruirui Lu, Gang Wu, Bin Xie, and Jingtong Hu. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, page 69–78, December 2014.

[45] 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, October 2024.

[46] Ruiying Ma, Shreya Shankar, Ruiqi Chen, Yiming Lin, Sepanta Zeighami, Rajoshi Ghosh, Abhinav Gupta, Anushrut Gupta, Tanmai Gopal, and Aditya G Parameswaran. Can ai agents answer your data questions? a benchmark for data agents. *arXiv preprint arXiv:2603.20576*, 2026.

[47] Aadyaa Maddi, Prakhar Naval, Deepti Mande, Shane Duan, Muckai Girish, and Vyas Sekar. Generating expressive and customizable evals for timeseries data analysis agents with agentfuel. (arXiv:2603.12483), March 2026. arXiv:2603.12483 [cs].

[48] Yuan Mei, Rui Xia, Zhaoqian Lan, Kaitian Hu, Lei Huang, Paris Carbone, Yanfei Lei, Vasiliki Kalavri, Han Yin, and Feng Wang. Disaggregated state management in apache flink® 2.0. *Proc. VLDB Endow.*, 18(12):4846–4859, August 2025.

[49] 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.

[50] Artem Oppermann. Detecting fraud in real time using Redpanda and Pinecone. https://www.redpanda.com/blog/fraud-detection-pipeline-redpanda-pinecone.

[51] Shyam Purkayastha. Building a real-time data processing pipeline for IoT. https://www.redpanda.com/blog/analyzing-iot-telemetry-data-apache-spark.

[52] RedPanda. Redpanda: The Agentic Data Plane. https://www.redpanda.com/.

[53] Redpanda Data, Inc. Redpanda Data – streaming data platform compatible with Apache Kafka. https://www.redpanda.com/.

[54] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language Models Can Teach Themselves to Use Tools. *arXiv preprint arXiv:2302.04761*, 2023.

[55] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

[56] Jacopo Tagliabue and Ciro Greco. Safe, Untrusted, "Proof-Carrying" AI Agents: Toward the Agentic Lakehouse. *arXiv preprint arXiv:2510.09567*, 2025.

[57] Cong Wang and Yusheng Zheng. Fork, Explore, Commit: OS Primitives for Agentic Exploration. *arXiv preprint arXiv:2602.08199*, 2026.

[58] WarpStream. The Diskless, Kafka Compatible Data Streaming Platform. https://www.warpstream.com/.

[59] Junjie Xing, Yeye He, Mengyu Zhou, Haoyu Dong, Shi Han, Lingjiao Chen, Dongmei Zhang, Surajit Chaudhuri, and HV Jagadish. Mmtu: A massive multi-task table understanding and reasoning benchmark. *arXiv preprint arXiv:2506.05587*, 2025.

[60] Tao Yu, Rui Zhang, Heyang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, Youxuan Jiang, Michihiro Yasunaga, Sungrok Shim, Tao Chen, Alexander Fabbri, Zifan Li, Luyao Chen, Yuwen Zhang, Shreya Dixit, Vincent Zhang, Caiming Xiong, Richard Socher, Walter Lasecki, and Dragomir Radev. Cosql: A conversational text-to-sql challenge towards cross-domain natural language interfaces to databases. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, page 1962–1979, Hong Kong, China, November 2019. Association for Computational Linguistics.

[61] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii, editors, *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, page 3911–3921, Brussels, Belgium, October 2018. Association for Computational Linguistics.

[62] Yihang Zheng, Bo Li, Zhenghao Lin, Yi Luo, Xuanhe Zhou, Chen Lin, Guoliang Li, and Jinsong Su. Revolutionizing database q&a with large language models: Comprehensive benchmark and evaluation. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pages 5960–5971, 2025.