Auditable Data Provenance in Streaming Data Processing

AFONSO BATE*, Instituto Superior Técnico, Portugal

Stream processing has gained prominence in Big Data analysis due to the demand for real-time analysis of unbounded data. Errors in data processing systems can lead to incorrect results, necessitating thorough examination of data flow and transformations. Data provenance is crucial for understanding errors and justifying results in stream processing, but it presents challenges due to the dynamic nature of the process. Existing solutions are often incomplete, lacking fine-grained provenance. In this work, a survey of stream processing and data provenance is conducted, proposing a solution involving three interconnected pipelines with Python modules. Testing in controlled environments emphasizes performance metrics, demonstrating the system's ability to preserve data provenance and offer insights for real-world scenarios. The results signify a significant advancement in reliable data tracing and management for stream processing systems.

Additional Key Words and Phrases: Stream Processing, Data Provenance

1 INTRODUCTION

With the rise of information-gathering devices like IoT sensors and smartphones, the demand for real-time data analysis has grown significantly. Batch processing, the conventional approach, falls short in dealing with this continuous influx of data. Stream processing, on the other hand, offers timely insights from unbounded data streams. Its advantages include the ability to handle never-ending data streams, reduced resource usage, and real-time processing, as opposed to storing data for batch processing. Common applications of stream processing include real-time fraud detection, IoT edge analytics, and personalized marketing.

This work addresses the need for data provenance in real-time streaming data processing. In a world flooded with streaming data, ensuring its reliability, integrity, and accountability is paramount. Traditional data lineage techniques aren't sufficient for the challenges posed by streaming data [8]. The focus is on ensuring auditable data provenance in real-time streams to track data's origin and history accurately. This is essential in domains like finance, healthcare, and security, where erroneous real-time data can have profound consequences [7, 12, 16, 17].

Auditable data provenance safeguards data from corruption, tampering, and unauthorized changes, reinstating confidence in its accuracy. It also enables the reproduction and verification of real-time results, crucial for various tasks like forensics and compliance with data protection laws [4].

In complex real-time systems, debugging is challenging. Auditable data provenance serves as a diagnostic tool, helping engineers and analysts trace data processing steps and identify discrepancies. This research aims to develop innovative solutions for auditable data provenance tailored to the dynamic nature of streaming data. By addressing these challenges, this work enhances the capabilities of data stream processing and extends its applicability across various domains.

*Supervisors: Luís Veiga, Paulo Carreira

Author's address: Afonso Bate, afonso.bate@tecnico.ulisboa.pt, Instituto Superior Técnico, Lisbon, Portugal.

Existing data provenance solutions for streaming data processing face several limitations. These challenges include:

- Latency and overhead;
- Scalability;
- Handling of complex events;
- Privacy and security;
- Dynamic semantics;
- Resource utilization.

Addressing these limitations is essential for the advancement of auditable data provenance in streaming data processing. Recognizing these shortcomings allows for the development of innovative solutions to address them effectively, ensuring data trustworthiness and transparency in the age of streaming data. The proposed framework, inspired by existing solutions, is designed to overcome these challenges and offers acceptable performance.

The framework is based on a sophisticated multi-pipeline system for continuous video stream processing, with a focus on data integrity, transparency, and result accuracy. Each pipeline comprises four modules (Module 0, Module 1, Module 2, and Module 3) with similar executions and functionalities. This modular structure enhances scalability and simplifies the addition of new functionalities.

Module 0 plays a fundamental role in ensuring data integrity by hashing both data sources and subsequent code files, guaranteeing that results align with the underlying codebase.

Module 1 is responsible for proactive anomaly detection, actively scanning video frames for irregularities and automatically triggering the second pipeline if an anomaly is detected. The second pipeline leverages periodic checkpoints and stored frames to recreate and validate results from the first pipeline, prioritizing, if needed, accuracy over performance.

For auditability and transparency, the third pipeline can be manually initiated by the user. It meticulously examines outcomes produced in the second pipeline, offering a comprehensive view of data lineage and processing steps, valuable for auditing and in-depth result analysis.

Apache Kafka serves as the communication backbone, facilitating seamless data exchange and tracking between pipelines and modules, emphasizing transparency and data provenance.

In summary, the proposed solution addresses existing system limitations by prioritizing data integrity, real-time processing, proactive anomaly detection, and precise, verifiable results. This comprehensive approach ensures trustworthiness and transparency in complex streaming data processing scenarios.

This work aims to contribute both theoretically and practically to the field of data provenance in data stream processing:

State-of-the-Art Survey A comprehensive study and analysis of the state of the art in stream processing and data provenance was conducted. This research spans the areas of Stream Processing and Data Audit, Lineage, and Provenance. By examining existing work and solutions, we identify their strengths and weaknesses. This analysis informs the design

of an architectural framework for complete and accurate auditable data provenance in a stream processing environment, ensuring acceptable performance and resource utilization.

Real-World Implementation The practical implementation and deployment of the proposed architecture within a real-life use case validates the effectiveness and feasibility of the auditable data provenance framework in addressing complex challenges in streaming data processing. The chosen use case involves processing continuous video streams in dynamic environments where individuals move within a physical space. This scenario presents numerous challenges, including real-time detection, tracking, and anomaly identification. By successfully implementing the proposed architecture in such a context, this work demonstrates its adaptability and problem-solving capabilities in dynamic scenarios.

Integration of Diverse Technologies The implementation integrates diverse technologies, including computer vision for object detection, deep learning models for tracking, Apache Kafka for communication, and state management for auditability. Additionally, it empowers users to configure the system for improved performance and provides tools for analyzing results through data and visual evidence.

Test Environment Design A test environment was designed to validate the correctness and assess the performance of our implementation. Relevant stream processing metrics, such as latency, throughput, and resource utilization, are rigorously analyzed. The accuracy and completeness of the solution are verified through this evaluation.

The rest of the paper is structured as follows. Section 2 describes briefly the fundamental and state of the art works in the Stream Processing and Data Provenance areas. Section 3 presents the architecture, the requirements for its implementation and the technological stack used. Section 4 presents the quantitative and qualitative evaluation to our solution. Finally, Section 5 wraps up the paper with our main conclusions.

2 RELATED WORK

This section represents all the work we found to be relevant to the creation of our proposed solution. We will divide this section into the two main topics of our work: **Stream Processing** and **Data Audit, Lineage, and Provenace**.

2.1 Stream Processing

The typical framework of a Stream Processing System is comprised of five disctinct layers:

- Data Stream Ingestion Layer
- Data Stream Processing Layer
- Storage Layer
- Resource Management Layer
- Output Layer

Nowadays, a wide variety of Stream Processing engines are available to handle the rising needs for processing streaming data. Based on some works [1, 6, 9, 13] we proposed a taxonomy, to evaluate the existing solutions, that takes into consideration the classification and main characteristics of stream processing engines, which are, **System Openness**, **Type of System**, **Architecture**, **Programming Model**, **Data Partitioning Strategy**, **State Management**, **Execution Semantics**, **Fault Tolerance** and **Deployment**. We also included some relevant metrics, such as, **Scalability**, **Performance** and **Resource Utilization**.

From the observed works [1–3, 6, 9–11, 13, 15, 20], we conclude that the most notable open-source stream processing engines are **Apache Spark**, **Apache Flink**, **Apache Storm** and **Apache Kafka Streams**.

Apache Spark is a versatile stream processing engine used for batch processing, interactive queries, and large-scale data stream processing. It supports multiple programming languages and has a large community. While it's mature and fault-tolerant, it's not a true stream processing engine and may have a few seconds of latency and high memory usage. To handle stream processing, Apache Spark introduced Spark Streaming, a micro-batch processing framework. Later, Structured Streaming, an upgraded module, used micro-batches and expressed streaming computation similarly to batch processing using Dataset/DataFrame APIs. Apache Spark and its modules provide end-to-end exactly-once processing guarantees.

Apache Flink is a native stream processing framework used for both batch and stream processing. It's capable of handling unbounded or bounded data streams from various sources. Flink provides exactly-once processing guarantees, offers high throughput with low latency, dynamically optimizes tasks, and has a user-friendly interface. However, it may have scaling limitations, supports only Scala and Java, and has limited community support.

Apache Storm is a stream processing engine known for its efficient real-time processing capabilities. It handles large data volumes with low latency. Its architecture is based on spouts and bolts, creating a directed acyclic graph (DAG) to process data. Storm offers a simple API, supports multiple languages, and is highly flexible and extensible. However, it lacks some features found in other stream processing engines, such as built-in windowing and state management. It provides at-least-once processing semantics but doesn't guarantee data ordering.

Apache Kafka Streams is a Java API for stream processing associated with Apache Kafka. It simplifies the creation of real-time data pipelines and applications, enabling operations like filtering, joining, aggregating, and grouping without extensive coding. Kafka Streams is easily integrable with existing applications, ensuring low latency and eliminating the need for conventional message brokers. However, it has limitations like reduced analytics capabilities, the absence of point-to-point queuing and other messaging paradigms, and challenges in managing numerous queues in a Kafka cluster.

2.2 Data Audit, Lineage, and Provenace

Data provenance is important to guarantee data quality by understanding the data's lifecycle, but that is not its only benefit. It can also be used to check the data's integrity, and help understand and justify the occurrence of some errors by providing an audit trail.

However, as we can see from several works [5, 14, 24, 27] on the topic of data provenance, this is not an easy feature to implement or to add to a stream processing system, as it poses numerous challenges. To trace data's lineage is an inherently heavy process

and ties the efficiency of the whole system to its own, since it can implicate an increase in latency, a reduction of throughput and complications in memory storage. We can conclude the main challenges are:

- Storage
- Latency and Throughput
- Determinism

Through extensive research we were able to find works that propose solutions to the previously mentioned challenges while still providing correct data provenance in stream processing systems.

GeneaLog [18], as presented by Palyvos-Giannas, focuses on fine-grained data provenance in deterministic stream processing engines. Its key contributions involve reducing storage costs. It achieves this by utilizing a small, fixed set of meta-attributes shared across all data streaming operators, minimizing the memory overhead per tuple, which is common in data provenance systems. Additionally, GeneaLog optimizes memory management to identify source tuples that contribute to the application output, discarding irrelevant data and conserving temporary storage. GeneaLog was prototyped on Liebre and Flink stream processing engines, with evaluations confirming its correctness and performance. The results showed that GeneaLog provides accurate data provenance while minimizing throughput and latency overheads compared to other state-of-the-art provenance systems.

Ananke [19] introduces a unique approach to address the lack of streaming-based tools for forward lineage tracing. It extends fine-grained backward provenance tools and creates a live bipartite graph of fine-grained forward provenance. Ananke not only identifies the source tuples contributing to each output but also determines which of these source tuples can still generate future distinct outputs, thus preventing duplication and reducing memory costs. This is achieved by leveraging native operators of the underlying stream processing engine and implementing specialized-operator-based and modular solutions. The authors implemented two variations of Ananke in Flink, one focusing on parallelization for handling larger amounts of provenance data and the other optimizing the labeling of expired source data for efficiency. The results confirm Ananke's correctness and demonstrate its efficiency in terms of rate, latency, throughput, memory usage, and CPU utilization. It offers live forward lineage tracing with minimal overhead, comparable to state-of-the-art backward lineage tracing tools, and it outperforms the Genealog system, which Ananke extends for backward lineage tracing.

The **s2p** [27] provenance solution combines both fine-grained and coarse-grained provenance in stream processing systems. It follows a philosophy inspired by lambda architecture. Online provenance traces and maps lineage from source data to result data, providing coarse-grained provenance. Offline provenance offers detailed information about intermediate results or transformation processes, offering fine-grained provenance. The approach of s2p is to focus on detailed lineage for a limited set of data that is considered relevant, rather than tracking the transformation and lineage for all input data. This selective approach minimizes overhead and costs. The system also takes into account operator states and considers the semantics of each operator when analyzing data relationships. One unique aspect is that s2p manages data locally and aggregates selected data only if a lineage query occurs, reducing data transformation costs. The authors implemented a prototype of s2p on Flink and conducted three experiments, showing that it increases end-to-end cost, reduces throughput, and has memory storage limitations. However, when compared to other existing provenance solutions, the runtime overhead is acceptable, considering that s2p targets more provenance-related data. It's important to note that s2p is limited to stream processing engines consisting entirely of deterministic operators.

Ariadne [5] is an early system designed for fine-grained provenance in stream processing. It achieves this by modifying operator behavior through a technique called Reduced-Eager operator instrumentation. This method propagates lineage information during query execution and then reconstructs it independently of the original network execution. Ariadne's approach, while incurring a higher cost for storing and reconstructing tuples, utilizes compressed representations to offset this cost with improved runtime and latency performance. It offers users the flexibility to request lineage tracing for specific results and can handle non-deterministic operators accurately. Additionally, Ariadne employs techniques like Replay-Lazy and Lazy-Retrieval to optimize lineage computation separately from stream processing. The authors conducted experimental evaluations that considered various parameters and workloads, assessing computational cost and latency. The results validate Ariadne's correctness and effectiveness while showing that, despite some minor overhead, it outperforms the state-of-the-art technique at the time, query rewrite.

Spark-Atlas-Connector (SAC) [23] is a system that facilitates interactive data provenance in stream processing systems, and it extends the functionality of Apache Atlas. One notable feature of SAC is its seamless integration with Apache Spark, requiring no modifications to the stream processing engine or additional user inputs. SAC efficiently provides a query interface for managing captured data lineage. It supports various data storage systems and stream processing paradigms. Moreover, it offers a visual representation of data lineage, allowing users to trace the flow of data from its source to the output stream. The system has demonstrated the capability to efficiently track data lineage for over 100GB per day in real-world deployments. However, it's important to note that SAC primarily focuses on coarse-grained provenance, which distinguishes it from other systems discussed here that mainly emphasize fine-grained provenance.

Yazici and Aktas have introduced a **real-time data lineage visualization method** [26] that employs graphs to represent the lineage of data. Their work provides forward and backward tracing, allowing users to identify relationships of data tuples both before and after a given data point. It offers the ability to condense extensive provenance data to focus on the most relevant aspects, which can be particularly helpful since such data can be quite large. Users can also compare lineage graphs, which is valuable for identifying anomalies and understanding data relationships. The authors have implemented a prototype visualization tool and evaluated its performance using two distinct datasets. The results show that the proposed visualization methods introduce minimal processing overhead and are scalable.

Zvara et al. [28, 29] have introduced a **lineage tracing framework** designed for both batch and streaming data processing systems. This framework aims to enhance performance and reduce overhead by identifying inefficiencies in lineage tracing. In this approach, lineage is determined by wrapping each record and capturing causality on a record-by-record basis. Additionally, incoming records are randomly sampled to decrease overhead, which contributes to efficiency optimization. The main advantages of this solution are its applicability to both batch and streaming data processing, as well as its ability to trace lineage in multiple systems using the same framework. The authors implemented two prototypes on Apache Spark, one for batch processing and another for stream processing, to perform experimental evaluations. The results demonstrate that this solution indeed enhances efficiency and reduces tail-latency. However, it also highlights that tracing lineage for all data can be quite expensive and lead to substantial overhead.

Huq et al. [8] have devised a solution to address the storage challenges associated with fine-grained provenance by introducing a **provenance inference algorithm** that combines a temporal data model with coarse-grained provenance. The temporal data model involves adding a temporal attribute, such as a timestamp, to each data item, enabling the retrieval of the complete state of a database at any specific time. When combined with the capacity to reconstruct the original processing window, which is facilitated by coarse-grained provenance, this approach ensures reproducibility and enables the algorithm to infer fine-grained provenance. This method is not tied to any specific dataset and becomes more storage-efficient as the processing windows overlap. Nonetheless, it comes with limitations. It can only provide precise lineage information when processing windows consistently produce the same number of output tuples. Furthermore, it offers highly accurate lineage information in a system that operates nearly instantaneously, which is often unattainable in practical settings. In real-world systems, where processing delays are common, the likelihood of inaccuracies in the inferred fine-grained provenance is considerably high.

Sansrimahachai et al. [21, 22] propose a solution for fine-grained provenance in stream processing, addressing the challenge of storage consumption. Their approach relies on a reverse mapping function known as the **Stream Ancestor Function**, which identifies dependency relationships for any data tuple within the stream, thus offering fine-grained provenance and ensuring the reproducibility of data processing. To resolve the storage issue associated with this method, the authors optimized the function. This optimization involved eliminating the necessity to store every intermediate stream element and enabling dynamic performance of provenance queries. Through an experimental evaluation, the authors confirmed the correctness of their solution and evaluated its impact on storage consumption and throughput. The results indicated that the solution not only reduced storage consumption but also introduced acceptable processing overheads.

Yamada et al. [25] introduced the concept of **augmented lineage**, a technique designed to ensure traceability of complex data analysis, particularly involving User Defined Functions for processing, within fields like Artificial Intelligence and Machine Learning. While their work initially focused on more static data analysis, the authors suggest that this framework can be extended to stream processing environments. Given its efficiency and potential, this framework could contribute to lineage tracing in stream processing in the future.

3 SOLUTION

Starting by the chosen use case, it involves user interaction through a video stream as input. The primary pipeline comprises real-time modules for people detection, tracking, area drawing, and area counting. As frames are processed, the system identifies individuals, tracks their movements, and calculates the number of people in specified areas. Continuous monitoring for anomalies occurs during real-time processing. If an anomaly, such as a person unexpectedly disappearing, is detected, the system generates an alert. This alert activates the secondary pipeline, which replays video frames starting from the initial detection of the suspect person. It replicates the normal processing steps to confirm if the same alert is triggered. If confirmed, the system permanently stores the relevant video frames and state data. Verified alerts and their associated information are registered, and the data gathered during alert validation empowers the system to reproduce the data stream deterministically through a third similar pipeline at any future time for audit or investigative purposes.

The architectural design of a stream processing system is guided by key requirements aimed at ensuring comprehensive data provenance and accountability. These requirements in our design are as follows:

- **Data Provenance and Lineage:** The core objective is to establish and maintain data provenance by capturing lineage information for all processed data. Essential data is stored to enable segment replays, promoting transparency and reducing resource usage.
- **Code Integrity:** The architecture ensures code and algorithm integrity by employing hash-based verification processes, guaranteeing that results align precisely with the employed code.
- **Dependency Management:** Effectively managing dependencies, especially in continuous streaming, is crucial. State information is meticulously and periodically preserved to facilitate replication.
- **Replicability for Validation:** Replicability is crucial for result validation, achieved through secondary and tertiary pipelines that recreate results for verification, made possible by storing critical data.
- **User Involvement:** While not the primary focus, user engagement allows users to review and validate results, enhancing the system's transparency and precision.
- **Scalability:** The architecture accommodates increasing data volumes while upholding data lineage tracking. Modularity within pipelines allows users to adapt the system by adding or removing modules without compromising data provenance.
- **Alerting:** The system's alerting mechanism detects anomalies in real-time, enhancing anomaly detection while efficiently managing resource usage through customized alerts.
- **Manual Auditing:** Specific auditing processes can be initiated manually, enabling comprehensive audits and data

stream analysis at a later time, providing users with flexibility for verification requirements.

Having these requirements in mind, an architectural framework was designed, which comprises three pipelines with similar modules and purposes but distinct roles. This section is organized into subsections for each pipeline, addressing communication, module interactions, and storage. Our work draws inspiration from s2p [27] framework, structured into an online phase with the first two pipelines and an offline phase represented by the third pipeline.

3.1 Primary Pipeline or Continuous Video Processing Pipeline

The continuous video processing pipeline is designed for real-time video analysis with a focus on integrity and provenance. It includes four modules:

- **Module 0 (Init):** This module ensures code integrity by hashing module code files and the video source. These hashes are stored for verification, ensuring consistency and detecting alterations in code.
- **Module 1 (Detection):** Responsible for real-time detection and tracking of individuals within video frames. It detects persons, tracks their movement, and alerts if individuals disappear unexpectedly. It maintains a state store for potential replays and generates a register of detection times.
- **Module 2 (Visual representations):** This module adds visual cues to the video. It draws bounding boxes around detected individuals, aiding transparency and result verification.
- **Module 3 (Counting):** Manages people counting and area analysis. It defines areas within frames for counting, and continuously updates counts as individuals move. This module provides analysis results to users, including entry/exit registers and annotated videos.

The primary pipeline includes a final Apache Spark module responsible for aggregating results from previous modules. It generates an updated list of detected person activities, tracking their entries and exits from the store and specific areas. This data is continuously updated using Apache Spark Structured Streaming.

The architectural design emphasizes object detection, tracking, and counting, along with modules for anomaly detection, visualization, and user interaction. This approach ensures result consistency with code, timely anomaly detection, and user engagement, enhancing the system's reliability and transparency in real-time video processing. Data flow within the pipeline, as well as communication between modules and pipelines, is managed by Apache Kafka.

Module 0 initiates the system by hashing the video source and code files of the pipeline's remaining modules. It sends these hash values and the video source to Module 1.

Module 1 receives the video source, processes frames for object detection and tracking, and stores states and alerts for replay in the State Store. Alerts trigger the secondary pipeline and, along with entry/exit times and coordinates, are sent to this pipeline. Results, frame count, and hash value are forwarded to the next module.

Module 2 adds bounding boxes around detected people and sends annotated frames, detections, frame count, and hash value to the following module.

Module 3 draws designated areas, counts people, and sends results and hash value to the Spark Module. Annotated frames are used to create a video with drawn areas and detected individuals.

3.2 Secondary Pipeline or Validation Pipeline

The secondary pipeline serves as a critical component, activated in response to primary pipeline alerts. Its primary function is to validate primary pipeline results, particularly anomaly detection, ensuring their accuracy and reliability. It comprises interconnected modules that replicate primary pipeline functions using stored state information and images. Continuous video processing poses challenges, with real-time demands and dependencies. The primary pipeline provides real-time insights but may encounter variations in processing conditions due to unbroken video streams. The secondary pipeline acknowledges these challenges. Its uniqueness lies in its commitment to accuracy. Even though it may generate slightly different results due to inherent variations, if both the primary and secondary pipelines trigger an alert for the same situation, it strongly validates the alert's genuineness. In summary, the secondary pipeline acts as an effective filter to confirm the validity of primary pipeline alerts by leveraging these inherent variations in continuous video processing.

The secondary pipeline is activated by an alert from the primary pipeline. Module 0, upon receiving the alert, hashes video source and code files to ensure system integrity. If the hash value matches the primary pipeline's stored value, the system proceeds; otherwise, it's interrupted. Module 0 sends alert information, such as suspect person ID and entry/exit details, to Module 1. It also retrieves stored states, frames, and alert details from the primary pipeline, facilitating data processing. Module 1 cross-references entry coordinates to identify the person to track and if the results of the replay confirm the alert, it stores this alert in permanent memory. It also saves video frames and necessary model states for replay. Module 2 processes the data, drawing bounding boxes and sending results to Module 3. Module 3, similar to the primary pipeline, draws designated areas and calculates people per area by frame, storing results in a file.

3.3 Third Pipeline or Replay Pipeline

The third pipeline ensures result replication and verification, a key aspect of data provenance. It replicates processes performed in the secondary pipeline. Unlike the primary and secondary pipelines, the third pipeline doesn't trigger automatically. Instead, it remains dormant until manually activated, typically for auditing purposes. This means that in case of an anomaly confirmation, the associated video section and state information are securely stored, ready to be revisited and re-validated later. Essentially, it enables the user to re-validate a previously detected alert by the two other pipelines without disrupting them.

To activate the third pipeline, the user selects a stored alert and a suspect's ID for data stream replay. Module 0 then checks the alert's existence in memory. If confirmed, it retrieves the suspect's entry and exit times, the video source, and hashes it along with the code files of the following modules. These resulting hash values are sent to the next module, along with each video frame.

Module 1 receives this data and obtains the Object Detection Model and Predictor associated with the alert from memory. After processing the frames, it produces detailed detection results with coordinates, storing them in a file for later analysis and sending them, along with the hash value, to the following module.

Module 2 performs operations similar to previous pipelines and sends frames with drawn areas, along with hash values and object detections from Module 1, to Module 3.

Module 3 processes the data stream, yielding results of the suspect per area by frame, which are stored in a file. Unlike the prior pipelines, Module 3 retains video frames with annotated areas and bounding boxes identifying the suspect person in memory. This enables the user to visually analyze results confirmed by the secondary pipeline by watching the resulting video or individual frames.

3.4 Data Structures

Data structures play a crucial role in our system for ensuring data lineage, maintaining data quality, and enabling efficient querying and replaying. A common aspect across all three pipelines is the use of hash values. Module 0, in all pipelines, hashes the video source and the files of subsequent modules using the SHA-256 algorithm and saves these values in a designated file. These values are used at the end of each pipeline to correlate incoming data with produced results, guaranteeing data integrity.

Another reason for data structures being fundamental components of our system, is the crucial role they play by representing events, like entries and exits of individuals in specific areas of a store or in the store itself, and alerts.

In our system, data structures are essential for storing data in the state store, enabling replay functionality for the second and third pipelines. The primary pipeline periodically saves data in the state store, and this periodicity is user-defined. The key elements stored in the state store include:

- Object Detection Model
- Object Detection Predictor
- Alerts

Additionally, a user-defined quantity of video frames is retained in the state store. This frame retention ensures that a history of video frames is continuously available, allowing the secondary pipeline to replay from a chosen checkpoint, enabling vital functions like result validation, auditing, and anomaly investigation. Some of these frames are also stored in the system's permanent memory during alert validation, optimizing resource management.

As for the object detection models, even though they are deep learning models with fixed neural network parameters, the checkpoint is implemented to ensure the correct object detection model and predictor are consistently used. This prevents a situation where a change in the object detection module could lead to inconsistent results when the second and third pipelines process queued alerts. Furthermore, it's important to store this data in a serialized format like Torchscript, JSON, or Pickle for python-to-python cases like in our system.

This usage of data structures is fundamental for ensuring provenance and traceability in our system by ensuring:

- Granular Data Representation
- Temporal Traceability
- Spatial Traceability
- Action Classification
- Provenance Verification
- Replay and Audit
- Cross-Referencing
- Data Integrity

To summarize, the data structures responsible for recording entry and exit events, including their associated details, combined with the use of a state store to enable precise data stream replays, are the foundational elements of our system's provenance tracking and auditability. These structures are crucial for maintaining a dependable record of events, detecting anomalies, and confirming the accuracy of processing steps. Their role in providing granular data and traceability is essential for ensuring transparency, accountability, and trustworthiness in our stream processing system.

4 EVALUATION

This chapter focuses on the comprehensive evaluation of the auditable data provenance system in stream processing. The evaluation process involves several phases, including local testing and networked scenarios, to assess performance and efficiency.

The evaluation begins with local testing to understand latency, throughput, and resource utilization. Apache Spark Structured Streaming benchmarks are used to simulate real-world workloads with different complexities. Prometheus and Grafana integration provides a detailed analysis of latency and throughput metrics within the online phase of the system. This analysis covers individual modules in the first two pipelines and evaluates the overall pipeline performance. Node Exporter is used to monitor CPU and memory utilization, offering insights into resource requirements.

The testing strategy includes both local and networked scenarios. Local testing isolates the system to examine its core performance factors without network-related variables. Networked testing introduces real-world network conditions, challenging the system in a more complex environment.

This thorough evaluation process offers a comprehensive view of the system's performance, enabling meaningful conclusions and insights for the practical implementation of the designed architecture.

In the remainder of this section we will present the obtained results for the two phases of testing, **Local Testing** and **Distributed Pipeline Testing**.

4.1 Local Testing

The initial phase of the evaluation aims to assess the system's core performance in an isolated environment. This phase serves as a foundational step to understand the system's behavior without the influence of potential confounding variables like network latency.

Local testing offers the advantage of isolating the system from network-related factors that can distort performance metrics. It provides a clean baseline of the system's capabilities, free from external influences. This phase helps establish a fundamental understanding of the system's strengths and weaknesses. The insights gained from this phase serve as the basis for further evaluations, including those conducted in more complex networked scenarios.

Two Apache Spark Structured Streaming benchmarks are used to establish a performance benchmark for the system. These benchmarks are designed to mimic real-world workloads, introducing randomness to simulate unpredictable data, which is essential for data provenance.

Benchmark A: is relatively simple but introduces randomness to test the system's ability to handle unpredictable data variations. It measures response time and resource utilization.

Benchmark B: is more complex and includes advanced operations and randomness to stress-test the system's processing capabilities, resembling scenarios with intricate data transformations.

Two datasets with 1000 events each are used in the benchmarks, delivered via a Kafka stream. The results are crucial for comprehending the system's performance under varying conditions and external factors in the testing environment.

However, it's crucial to note that the benchmarks and our implementation have key distinctions. The benchmarks use Apache Spark Structured Streaming with a micro-batching model, while our implementation handles data frame by frame. Additionally, our implemented system deals with more complex image data, impacting resource utilization and performance differently.

These distinctions must be considered when interpreting the results, especially when comparing resource utilization metrics. Despite these differences, analyzing these metrics is valuable for understanding how alternative data processing solutions perform under controlled conditions.

Table 1. Benchmark testing results

	Average CPU Usage	Average RAM Usage
Benchmark A	85.5%	98.33%
Benchmark B	87.9%	98.83%
Normal Usage	4.5%	50.5%

The next step in the local testing phase was to analyse the **latency**, **throughput** and **resource utilization** of our implementation's online phase pipelines.

The data for this analysis was collected from processing a 5-minute video stream and was meticulously acquired through Prometheus, Grafana, and Node Exporter, enabling a multifaceted evaluation of the system's capabilities.

The data collected allowed us to calculate the following measurements:

- Average Latency
- Total Latency through 15 Seconds
- Percentiles for Latency (25th, 50th, 75th, 90th, 95th)
- Throughput Per Second
- CPU Usage
- RAM Usage

These analyses were conducted individually for each of the three modules within the two pipelines that compose the online phase, allowing for a fine-grained examination of how each module contributes to the overall system performance. Furthermore, we assessed the pipelines as a whole to gain insights into the end-to-end performance of the system.

As for the results regarding resource usage during testing, they were on average:

CPU Usage: 92.1%
RAM Usage: 70.7%

The results for latency testing are presented on the tables 2, 3, 6 and 7.

The results for throughput testing are presented on the tables 4 and 5.

Table 4. Throughput (frames per second) results for local testing on Primary Pipeline

	Average	Max Average	Min Average
Module 1	4.48	5.07	3.99
Module 2	4.44	5.13	3.60
Module 3	4.46	5.00	4.07
Pipeline	4.46	5.00	4.07

Table 5. Throughput (frames per second) results for local testing on Secondary Pipeline

	Average	Max Average	Min Average
Module 1	4.70	5.21	4.13
Module 2	4.70	5.20	4.27
Module 3	4.71	5.27	4.27
Pipeline	4.71	5.27	4.27

Table 6. Latency Percentiles for Primary Pipeline and its Modules in local testing

Percentile	Module 1 (s)	Module 2 (s)	Module 3 (s)	Pipeline (s)
25th	0.020	0.023	0.007	0.221
50th	0.031	0.038	0.011	0.256
75th	0.041	0.054	0.030	0.287
90th	0.047	0.080	0.048	0.329
95th	0.055	0.092	0.059	0.371

Table 7. Latency Percentiles for Secondary Pipeline and its Modules in local testing

Percentile	Module 1 (s)	Module 2 (s)	Module 3 (s)	Pipeline (s)
25th	0.249	0.035	0.021	0.448
50th	0.291	0.057	0.032	0.537
75th	0.338	0.079	0.042	0.589
90th	0.404	0.092	0.052	0.641
95th	0.442	0.098	0.064	0.706

4.2 Distributed Pipeline Testing

In the second phase of our evaluation, known as Distributed Pipeline Testing, we deploy both pipelines of our online system on separate computing nodes within the same local network. This controlled yet more realistic environment mimics real-world scenarios, providing

Table 2. Latency results for local testing on Primary Pipeline

	Avg Lat	Max Avg Lat	Min Avg Lat	Avg Lat 15s	Max Lat 15s	Min Lat 15s
Module 1	0.061 s	0.139 s	0.024 s	2.95 s	4.18 s	1.24 s
Module 2	0.041 s	0.047 s	0.037 s	1.89 s	2.34 s	1.49 s
Module 3	0.017 s	0.024 s	0.012 s	0.72 s	0.94 s	0.50 s
Pipeline	0.292 s	0.359 s	0.228 s	12.72 s	13.57 s	11.63 s

Table 3. Latency results for local testing on Secondary Pipeline

	Avg Lat	Max Avg Lat	Min Avg Lat	Avg Lat 15s	Max Lat 15s	Min Lat 15s
Module 1	0.213 s	0.240 s	0.191 s	14.96 s	15.27 s	14.68 s
Module 2	0.038 s	0.043 s	0.035 s	2.77 s	2.96 s	2.48 s
Module 3	0.021 s	0.025 s	0.015 s	1.45 s	1.77 s	1.15 s
Pipeline	0.363 s	0.394 s	0.330 s	25.59 s	26.54 s	24.15 s

insights into the system's performance in distributed, networked conditions. The objective is to evaluate how the system handles collaborative processing and distributed workloads, reflecting operational setups commonly encountered in practical applications.

During this testing phase, running each pipeline on different machines within the same local network added a layer of complexity due to the need for network communication between them. Kafka remained the primary messaging system for transmitting alert messages from the primary pipeline to the secondary pipeline. However, the secondary pipeline required access to more than just alert messages; it needed the state information stored by Module 1 in the primary pipeline. These files contained critical data for replicating frame processing and validating alerts.

To facilitate this data transfer, the SSH File Transfer Protocol (SFTP) was employed because Kafka is not suitable for file transfers. Machine B, responsible for the primary pipeline, transferred these files to Machine A, which hosted the secondary pipeline. Machine A played a crucial role in managing the state store essential for replicating and thoroughly assessing data processing. This meticulous data transfer process aimed to create a testing environment more closely resembling real-world scenarios, enhancing the system's effectiveness and efficiency validation.

The metrics and measurements for performance analysis in this phase remain the same as in the previous stage, acquired through the same process and tools.

The results for latency testing are presented on the tables 9, 10, 13 and 14.

The results for throughput testing are presented on the tables 11 and 12.

Table 11. Throughput (frames per second) results for distributed pipeline testing on Primary Pipeline

	Average	Max Average	Min Average
Module 1	3.53	4.40	2.33
Module 2	3.63	4.93	2.40
Module 3	3.42	4.73	1.87
Pipeline	3.42	4.73	1.87

Table 12. Throughput (frames per second) results for distributed pipeline testing on Secondary Pipeline

	Average	Max Average	Min Average
Module 1	7.63	8.07	5.87
Module 2	7.64	8.08	6.07
Module 3	7.65	8.11	5.93
Pipeline	7.65	8.11	5.93

Table 13. Latency Percentiles for Primary Pipeline and its Modules in distributed pipeline testing

Percentile	Module 1 (s)	Module 2 (s)	Module 3 (s)	Pipeline (s)
25th	0.020	0.021	0.011	0.401
50th	0.030	0.034	0.025	0.499
75th	0.041	0.046	0.046	0.618
90th	0.047	0.067	0.071	0.680
95th	0.051	0.079	0.082	0.755

Table 14. Latency Percentiles for Secondary Pipeline and its Modules in distributed pipeline testing

Percentile	Module 1 (s)	Module 2 (s)	Module 3 (s)	Pipeline (s)
25th	0.113	0.020	0.018	0.226
50th	0.128	0.030	0.028	0.252
75th	0.143	0.040	0.039	0.278
90th	0.152	0.046	0.046	0.296
95th	0.159	0.048	0.048	0.323

4.3 Analysis of Experimental Results

Before analysing the results obtained from our two phases of testing, it's important to highlight a limitation. This limitation is the usage of a single video data source for testing, which, due to the video's nature, resulted in a higher frequency of alerts than typical real-world situations, impacting system performance but allowing for worst-case scenario testing.

In the Local Testing phase, resource usage is compared with benchmark testing, showing slightly higher CPU usage but more efficient RAM utilization. In the Distributed Pipeline Testing phase, CPU usage is lower, indicating optimized utilization when distributing the workload. RAM usage improves on one machine and marginally on the other.

Table 8. Resource Usage results during distributed pipeline testing

	Machine A normal	Machine A testing	Machine B normal	Machine B testing
CPU Usage	4.5%	60.6%	0.5%	58.7%
RAM Usage	50.5%	68.5%	10.2%	29.7%

Table 9. Latency results for distributed pipeline testing on Primary Pipeline

	Avg Lat	Max Avg Lat	Min Avg Lat	Avg Lat 15s	Max Lat 15s	Min Lat 15s
Module 1	0.061 s	0.118 s	0.024 s	3.14 s	4.74 s	1.23 s
Module 2	0.032 s	0.058 s	0.021 s	1.71 s	2.78 s	0.95 s
Module 3	0.023 s	0.038 s	0.014 s	1.186 s	1.60 s	0.65 s
Pipeline	1.064 s	2.460 s	0.353 s	51.89 s	143.00 s	18.61 s

Table 10. Latency results for distributed pipeline testing on Secondary Pipeline

	Avg Lat	Max Avg Lat	Min Avg Lat	Avg Lat 15s	Max Lat 15s	Min Lat 15s
Module 1	0.131 s	0.165 s	0.124 s	14.93 s	15.11 s	14.56 s
Module 2	0.022 s	0.025 s	0.020 s	2.69 s	2.71 s	2.19 s
Module 3	0.015 s	0.016 s	0.013 s	1.67 s	1.88 s	1.42 s
Pipeline	0.244 s	0.298 s	0.233 s	27.952 s	28.97 s	26.61 s

Latency in the Primary Pipeline remains similar in both phases, but the overall pipeline latency increases in the Distributed Pipeline Testing due to network latency and concurrent file transfers affecting performance. The Secondary Pipeline shows improved performance in the Distributed Pipeline Testing due to the reduced competition for resources.

As for the qualitative analysis, it confirms that the Third Pipeline consistently produces identical results to the Secondary Pipeline, emphasizing result accuracy.

In summary, the system demonstrates efficient resource utilization and reliable performance under various testing scenarios, with an emphasis on result accuracy in data provenance-sensitive situations.

5 CONCLUSION

In our work we aimed to explore, design, and implement auditable data provenance solutions tailored to the dynamic nature of streaming data. By understanding the advantages and shortcomings of the existing solutions, we designed an architecture framework for complete and correct auditable data provenance in a stream processing environment which was applied to a real-life use case. By processing continuous video streams in a dynamic setting, we showed our system can handle multiple complex challenges and be integrated with diverse technologies, like computer vision for object detection, deep learning models for tracking, Apache Kafka for communication and state management for auditability. A test environment was also designed to validate the correctness of our solution and evaluate its performance. Through this test environment a quantitative analysis was performed which allowed to evaluate the performance of our solution in different conditions. A qualitative analysis was also made, which allowed us to guarantee that our system offers dependable and accurate data provenance in the context of a demanding stream processing environment.

REFERENCES

[1] Fuad Bajaber, Radwa Elshawi, Omar Batarfi, Abdulrahman Altalhi, Ahmed Barnawi, and Sherif Sakr. 2016. Big data 2.0 processing systems: Taxonomy and open challenges. *Journal of Grid Computing* 14 (2016), 379–405.

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

[3] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. 2016. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 1789–1792.

[4] Robert Eiss. 2020. Confusion over Europe's data-protection law is stalling scientific progress. *Nature* 584 (08 2020), 498–498. https://doi.org/10.1038/d41586-020-02454-7

[5] Boris Glavic, Kyumars Sheykh Esmaili, Peter M Fischer, and Nesime Tatbul. 2014. Efficient stream provenance via operator instrumentation. *ACM Transactions on Internet Technology (TOIT)* 14, 1 (2014), 1–26.

[6] Darshankumar Vinubhai Gorasiya. 2019. Comparison of open-source data stream processing engines: spark streaming, flink and storm. (2019).

[7] Fatih Gürcan and Muhammet Berigel. 2018. Real-Time Processing of Big Data Streams: Lifecycle, Tools, Tasks, and Challenges. In *2018 2nd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*. 1–6. https://doi.org/10.1109/ISMSIT.2018.8567061

[8] Mohammad Rezwanul Huq, Andreas Wombacher, and Peter MG Apers. 2011. Inferring fine-grained data provenance in stream data processing: reduced storage cost, high accuracy. In *Database and Expert Systems Applications: 22nd International Conference, DEXA 2011, Toulouse, France, August 29-September 2, 2011, Proceedings, Part II 22*. Springer, 118–127.

[9] Haruna Isah, Tariq Abughofa, Sazia Mahfuz, Dharmitha Ajerla, Farhana Zulkernine, and Shahzad Khan. 2019. A survey of distributed data stream processing frameworks. *IEEE Access* 7 (2019), 154300–154316.

[10] Ziya Karakaya, Ali Yazici, and Mohammed Alayyoub. 2017. A comparison of stream processing frameworks. In *2017 International Conference on Computer and Applications (ICCA)*. IEEE, 1–12.

[11] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking distributed stream data processing systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1507–1518.

[12] Taiwo Kolajo, Olawande Daramola, and Ayodele Adebiyi. 2019. Big data stream analysis: a systematic literature review. *Journal of Big Data* 6 (06 2019), 47. https://doi.org/10.1186/s40537-019-0210-7

[13] Devesh Kumar Lal and Ugrasen Suman. 2019. Towards comparison of real time stream processing engines. In *2019 IEEE Conference on Information and Communication Technology*. IEEE, 1–5.

[14] Hyo-Sang Lim, Yang-Sae Moon, and Elisa Bertino. 2009. Research issues in data provenance for streaming environments. In *Proceedings of the 2nd SIGSPATIAL ACM GIS 2009 International Workshop on Security and Privacy in GIS and LBS*.

58–62.

[15] Xiufeng Liu, Nadeem Iftikhar, and Xike Xie. 2014. Survey of real-time processing systems for big data. In *Proceedings of the 18th International Database Engineering & Applications Symposium*. 356–361.

[16] Erum Mehmood and Tayyaba Anees. 2020. Challenges and Solutions for Processing Real-Time Big Data Stream: A Systematic Literature Review. *IEEE Access* 8 (2020), 119123–119143. https://doi.org/10.1109/ACCESS.2020.3005268

[17] Alramzana Nujum Navaz, Saad Harous, Mohamed Adel Serhani, and Ikbal Taleb. 2019. Real-Time Data Streaming Algorithms and Processing Technologies: A Survey. In *2019 International Conference on Computational Intelligence and Knowledge Economy (ICCIKE)*. 246–250. https://doi.org/10.1109/ICCIKE47802.2019.9004318

[18] Dimitris Palyvos-Giannas, Vincenzo Gulisano, and Marina Papatriantafilou. 2019. GeneaLog: Fine-grained data streaming provenance in cyber-physical systems. *Parallel Comput.* 89 (2019), 102552.

[19] Dimitris Palyvos-Giannas, Bastian Havers, Marina Papatriantafilou, and Vincenzo Gulisano. 2020. Ananke: a streaming framework for live forward provenance. *Proceedings of the VLDB Endowment* 14, 3 (2020), 391–403.

[20] Henriette Röger and Ruben Mayer. 2019. A comprehensive survey on parallelization and elasticity in stream processing. *ACM Computing Surveys (CSUR)* 52, 2 (2019), 1–37.

[21] Watsawee Sansrimahachai, Luc Moreau, and Mark J Weal. 2013. An on-the-fly provenance tracking mechanism for stream processing systems. In *2013 IEEE/ACIS 12th International Conference on Computer and Information Science (ICIS)*. IEEE, 475–481.

[22] Watsawee Sansrimahachai, Mark J Weal, and Luc Moreau. 2012. Stream ancestor function: A mechanism for fine-grained provenance in stream processing systems. In *2012 Sixth International Conference on Research Challenges in Information Science (RCIS)*. IEEE, 1–12.

[23] MingJie Tang, Saisai Shao, Weiqing Yang, Yanbo Liang, Yongyang Yu, Bikas Saha, and Dongjoon Hyun. 2019. Sac: A system for big data lineage tracking. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1964–1967.

[24] Jianwu Wang, Daniel Crawl, Shweta Purawat, Mai Nguyen, and Ilkay Altintas. 2015. Big data provenance: Challenges, state of the art and opportunities. In *2015 IEEE international conference on big data (Big Data)*. IEEE, 2509–2516.

[25] Masaya Yamada, Hiroyuki Kitagawa, Toshiyuki Amagasa, and Akiyoshi Matono. 2022. Augmented lineage: traceability of data analysis including complex UDF processing. *The VLDB Journal* (2022), 1–21.

[26] Ilkay Melek Yazici and Mehmet S Aktas. 2022. A novel visualization approach for data provenance. *Concurrency and Computation: Practice and Experience* 34, 9 (2022), e6523.

[27] Qian Ye and Minyan Lu. 2021. s2p: provenance research for stream processing system. *Applied Sciences* 11, 12 (2021), 5523.

[28] Zoltán Zvara, Péter GN Szabó, Barnabás Balázs, and András Benczúr. 2019. Optimizing distributed data stream processing by tracing. *Future Generation Computer Systems* 90 (2019), 578–591.

[29] Zoltán Zvara, Péter GN Szabó, Gábor Hermann, and András Benczúr. 2017. Tracing distributed data stream processing systems. In *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE, 235–242.