Chaos Engineering for Streaming Process Mining

Florian Schlösser
stu240349@mail.uni-kiel.de
Kiel University
Kiel, Germany

Abstract

This paper investigates the resilience of distributed streaming process mining algorithms against failures, specifically for the Heuristics Miner with Lossy Counting. The algorithm is tested using chaos engineering to introduce failure scenarios into the streaming environment. The findings offer insights into the fault tolerance of the HM-LC algorithm for various failure conditions. In the experiments made, clear failure patterns for specific faults become apparent, which could prove valuable for detecting them accurately. Additionally, I provide a framework, applicable to any Kubernetes cluster, to quickly deploy, monitor, and analyze chaos engineering tests.

1 Introduction

Process mining involves discovering, monitoring, and improving processes by extracting information from event logs. It is relevant in many fields (e.g. business process modeling, anomaly detection). Process mining algorithms make it possible to create process models from the event data [8]. Traditionally applied to completed event logs, recent algorithms process streaming data, where analysis occurs in real-time rather than after all data is collected [2].

Distributed systems are prone to failure and complex errors. For these systems, testing their fault tolerance to maintain stability in productive environments is crucial [1, 5].

Chaos engineering is a recent software testing approach Netflix developed for making their streaming service resilient to failures [1]. In the following, I will refer to tests made using chaos engineering principles as chaos tests.

Traditional software testing typically defines a specific state (e.g. in a unit or end-to-end test) [11]. In contrast, chaos tests are made against a hypothesis, for instance: "The system will still work if 40% of all packets do not arrive or are corrupted." Given the common failures in distributed systems, such as server outages, and high network latencies chaos engineering has become a testing approach used in many professional environments [5].

Chaos engineering has not been applied to process mining yet, according to my literature review. This paper aims to bridge the gap by benchmarking the failure resilience of streaming process mining algorithms, using chaos tests. To limit the scope, I focus on the Heuristics Miner with Lossy Counting [3], refereed to as HM-LC in the following. Specifically, I tested how system failures and network delays affect the message processing of the HM-LC algorithm.

Structure: Section 1 explains the state of related research and defines the research questions. Section 2 describes process mining and explains the HM-LC algorithm in detail. Additionally Chaos Engineering and stream processing environments are introduced. Section 3 outlines the contributions of this paper and defines used metrics. Section 4 lists conducted experiments in detail and presents the results. Finally, Section 5 suggests applications for the conducted research and outlines questions to be explored further.

1.1 Problem Description

Testing how well distributed systems respond to failures is crucial for ensuring availability [1]. While the failure recovery of distributed algorithms is well understood, stream processing in distributed environments presents more challenges [4]. In a previous study chaos engineering has been used to benchmark the performance of stream processing frameworks, when failures are injected into them [9]. However, streaming process mining algorithms have not yet been tested.

1.2 State of Research

Process Mining: Burattin et al. proposed an extension of the Heuristics Miner using lossy counting [3]. They suggested that the HM-LC has similar characteristics to common data streaming tasks [2]. Reiter tested the scalability of the HM-LC and made suggestions for improving the algorithm [6].

Chaos engineering originated in the industry; nevertheless, researchers have begun developing benchmarks for chaos testing systems [5]. The fault tolerance of commonly used data streaming frameworks has been investigated and benchmarked by Vogel et al. in a recent study. They injected system failures into the Kubernetes [1] (refered to as K8s) pods[2]executing the benchmarking algorithm using Chaos Mesh[3] [9].

1.3 Research Questions

This paper seeks to answer the following research questions:

- **RQ1:** How quickly does the HM-LC recover from system failures in the environment, compared to previous studies?
- **RQ2:** How does network latency impact the recovery of the HM-LC after a previous pod failure?

Additionally, this paper introduces a framework utilizing Chaos Mesh for efficiently deploying, monitoring, and analyzing chaos test executions. The framework is flexible and configurable, allowing chaos tests to be deployed in any Kubernetes environment.

2 Foundation

Process mining is a research field about improving processes and gathering information about them based on event data. An event typically consist of a case identifier c_{id} for the process, an activity name or description A, and a timestamp t.

Although an event may have more attributes as explained in [8] this paper focuses on these three key ones. Hence, a single event is

[1] Open-source container orchestration management system (https://kubernetes.io).

[2] A pod is the smallest containerized application unit in Kubernetes

[3] Open-source chaos testing framework (https://chaos-mesh.org).

defined as the tuple $e := (c_{id} \times A \times t)$. The **Heuristics miner** (HM) is a classic process mining algorithm, applied to historic event data. It counts occurrences of events to infer which events are causally related to each other. Let $|a > b|$ be the number of times event b is observed directly after event a. The dependency measure indicating the strength of the relation between a and b is defined as:

$$a \implies b := \frac{|a > b| - |b > a|}{|a > b| + |b > a| + 1} \in [-1, 1]$$

The algorithm considers only those relations with a dependency measure above a defined threshold. These strong dependencies are used to construct a dependency graph, with nodes representing activities and edges representing the directly follows relations between two activities. The algorithm also works in scenarios where activities have multiple incoming or outgoing connections [2].

Burattin et al. introduced the concept of **streaming process mining** where events arrive in a stream of data. "An event stream is a stream where each observable unit contains information related to the execution of an event and the corresponding process instance" [2]. Let O be the universe of observable units. An event stream S is defined as an infinite sequence of observable units: $S := \mathbb{N}_{\geq 0} \rightarrow O$" (analogously to [2]). When observing an event stream, the part of the stream that a system can store is finite.

The **Heuristics Miner with Lossy Counting** (refer to Figure 1) collects incoming data from a stream and computes the dependency measure of the directly follows relations, deriving a process model from them.

In detail, the algorithm takes the event stream S and an approximation error ϵ as input. It tracks buckets of finite size. The bucket size w is determined based on ϵ. The variable N is used as an index for switching between buckets. The algorithm manages three sets $\mathcal{D}_A, \mathcal{D}_C, \mathcal{D}_R$. At the start of each iteration, an event e is taken from the stream S, and the current bucket is identified. Incoming events S are tracked by $\mathcal{D}_A$, which contains tuples (a, f, Δ) where a is the activity name, f the number of observations of this event in the current bucket, and Δ is the bucket number (0-indexed) of the first observation of that event.

The set $\mathcal{D}_C$ holds the last occurrence of an event, while $\mathcal{D}_R$ holds directly follows relations r, with $r \in A \times A$ and stores tuples (r, f, Δ). All three sets are updated similarly: if the relevant entry already exists in a set, it is modified; otherwise it is added to the set. At the end of each iteration, if a bucket is full, the least frequent relations and corresponding events are deleted from the sets to create space for new entries. Finally the algorithm moves to the next bucket and updates the process model based on dependency measures calculated from the relations found in $\mathcal{D}_R$ [3]. This process is repeated forever. For detailed information, refer to [2, 3, 6].

Chaos Engineering tests the resilience of distributed systems by injecting controlled failures into the system. It can help to determine system performance under stress by simulating disruptions such as pod failures, increased network latency, and limiting available resources. This approach can help identify vulnerabilities and weaknesses in system design and algorithms that might not be exposed when testing traditionally.[5]

Stream processing deals with programs processing continuous data streams. Each stream is an infinite sequence of items. Workers executing a stream processing algorithm receive an input stream

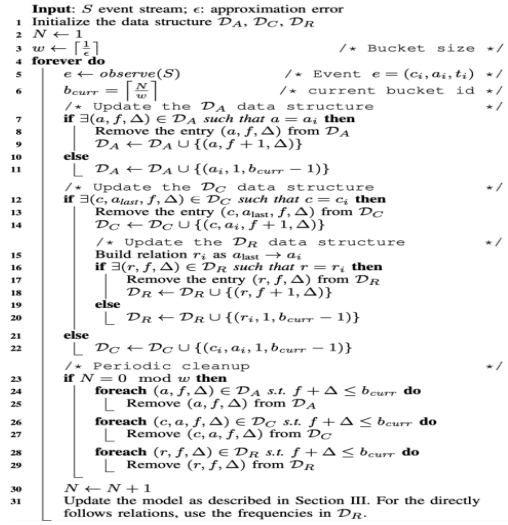


Figure 1: HM-LC (Burattin et al., 2022)[3]

and produce an output stream [4]. *Shufflebench*, a benchmarking framework for stream processing, mainly re-distributes aggregated data that the workers hold. In a recent study, *Shufflebench* was used with Chaos Mesh [9]. Given the comparable approach of the HM-LC, which stores relations and events in buckets until full and then redistributes them, I use similar metrics and methodology.

3 Chaos Engineering Setup and Performance Metrics

This paper presents a framework[4] based on Chaos-Mesh to automate the application and monitoring of chaos tests on K8s resources. It offers configurable templates for quickly deploying chaos tests, introducing network delays, corrupted packets, time skews, and pod failures. These templates can be adjusted using jsonnet[5] and converted to YAML for deploying in K8s. The main application provides a command-line interface to run and monitor selected tests automatically. Tests can also be defined in the Chaos Dashboard[6] and be integrated by loading the YAML into the main application. Additionally, the framework can do post-processing, data analysis, and generate plots. The code is highly modular and can be adjusted and extended easily. The framework is environment-independent as long as K8s is used. Tested applications and algorithms can vary, but the data source must be configured accordingly.

3.1 Metrics and Benchmarking

This paper benchmarks the HM-LC for different chaos events. To determine if the algorithm is operating as expected, the implementation measures if the amount of messages that the HM-LC cannot process increases. Similar metrics are provided by stream processing frameworks. For Apache Kafka[7], which is used in this case, the

[4] https://github.com/fsch-ppi/ChaosMeshWizard

[5] Configuration language for JSON (https://jsonnet.org).

[6] Web interface of Chaos Mesh.

[7] Open-source distributed event streaming framework (https://kafka.apache.org).

metric observed is the consumer group lag. It is obtained by counting the messages produced, but not consumed by the consumer group. In the following, consumer group lag will be referred to as lag.

To benchmark the HM-LC similar to [9], the conducted experiments inject pod failures, which simulate an operating system failure in the pod. When a pod becomes unresponsive, the K8s cluster will provide a new instance eventually. For pod kills, similar observations to [9] were made in early tests; the K8s cluster is directly informed of the pod kill and provides a new pod in 2-3 seconds. This paper focuses on pod failures for the experiments, since for pod kills, the lag is never reasonably high, as pods are quickly replaced. Also, pod failures are more likely to occur.

For failure recovery in streaming frameworks, [7] finds that 5 minutes is a suitable time to create new pods. This paper deviates from this suggestion, because the used HM-LC implementation only has a single worker. A shorter time offset has shown to be sufficient.

The data observed contains a timestamp and lag measurements. Let d_c be the duration of a chaos event and d_r be the recovery time, which is the duration from when the lag increases until it stabilizes again. It holds that $d_c \leq d_r$, since the system needs time to respond. To determine d_r from the data, find a start timestamp and end timestamp using the Prominence[8] (for a detailed definition refer to [10]) and the median of the series. Let $S = \{s_1, ...s_n\}$ with $s_n \in \mathbb{N}$ be the observed series of data points. Each index s_i has a timestamp $t(s_i)$, and a lag $l(s_i)$. For the used metric define $P := \{i \mid \text{Promincence}(l(s_i)) > 50\}$ as the set containing points of maximal lag. For a $s_i \in P$, determine the interval $C = [s_l, s_r] \subseteq S$, where s_l is the smallest index with $\text{median}(\text{lag}(S)) \leq \text{lag}(s_l)$ and $s_l < s_i$. Similarly, s_r is the largest index with $\text{median}(\text{lag}(S)) \leq \text{lag}(s_r)$ and $s_n \geq s_r > s_i$. The failure recovery time is calculated using $d_{r_{s_i}} = t(s_r) - t(s_l)$. Using this metric, recovery times are comparable for different chaos testing scenarios.

4 Experiments

The conducted experiments include:

- **Ex1:** Injecting Pod failures into the worker pod
- **Ex2:** Injecting Pod failures into the worker pod with varying general network latency

Ex1 aims to address **RQ1**, while **Ex2** provides insight for **RQ2**. In both experiments, pod failures are injected using a cron schedule[9]. Experiments are not re-created during a monitored run as this would introduce a methodical error dependent on the current system load. Additionally, for **Ex2** the latency for all messages being sent in the K8s namespace[10]is increased. The namespace contains all pods related to Kafka as well as the HM-LC worker.

Each experiment was conducted three times for each variable parameter to obtain statistically reliable results, similar to [9]). In each of the three runs, pod failures are injected twice, waiting two minutes in between. Since the used HM-LC implementation has only one worker, the architecture tested is not distributed. Consequently, the preconditions, compared to [9], are different. However, waiting two minutes in between failure injections has proven sufficient.

4.1 Setup

All tests were conducted in a K8s environment hosted on a Virtual Machine deployed in Microsoft Azure[11]. The used VM was a Standard_D8_v3[12]. K8s was given 24GB of RAM and all 8 CPUs. The chaos tests were executed using the Chaos Mesh library. Apache Kafka was used as the stream processing framework. The HM-LC was deployed using Apache Flink[13]. Data was monitored with Prometheus[14], which was queried by the monitoring scripts of the contributed framework.

For each run, the heuristic miner, data stream, and stream processing framework were re-deployed to ensure independent results between runs. The installation guide, including all configurations can be found here[15].

4.2 Results

For **Ex1** the pod failures can be clearly observed in the data (refer to Figure 2). Before a chaos event, the lag is constant, with minor fluctuations. When a failure occurs, the lag initially increases linearly, then spikes to very high values until a new pod is created. Afterwards, the lag becomes constant again. **The average failure recovery time d_r in Ex1 is approximately 56.594 seconds**.

In **Ex2** the lag is also constant initially. In contrast to **Ex1**, the observed pattern changes with increasing network latency. Additionally, the observed lag values are much higher. When a chaos event occurs, the lag spikes instantly, then decays linearly with steps, like a stairway. For increasing network latency, the failure recovery time increases slightly, but not significant (refer to Figure 3).

4.3 Discussion

Ex1 yielded results similar to [9]. Since the implementation used in this paper, does not use more than one worker, the message overhead for failure recovery observed in [9] for distributed architectures is not observable in the data.

For **Ex2** the lag does not differ to **Ex1** in base-load regardless of the latency. This is because the latency affects the entire system. Consequently, message production is also delayed equally as the message consumption. Therefore, the lag does not increase. However, the time required to recover from the pod failure very slightly. To assess significance, more data is required. If significant, this increase is likely due to latency impacting the messages K8s uses to detect and replace the failed pod.

In both experiments, the lag fluctuates, which is probably caused by the system load. To provide accurate insights for industry-grade systems, the experiments from this paper would need to be conducted on such a system.

[8] From signal processing: How much a peak stands out relative to its higher neighboring local minima

[9] Time job scheduler on Unix systems (https://en.wikipedia.org/wiki/Cron).

[10] Kubernetes concept: A namespace can hold multiple pods and services.

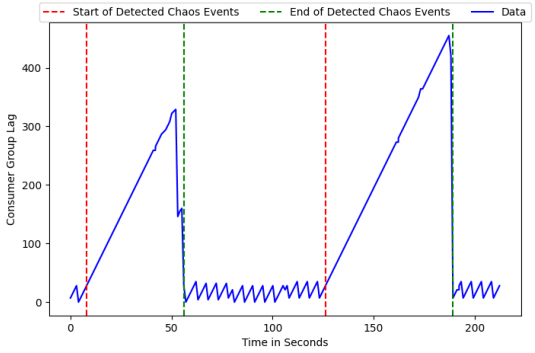
[11] Could computing platform and software hosting service by Microsoft (https://portal.azure.com).

[12] https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/general-purpose/dv3-series

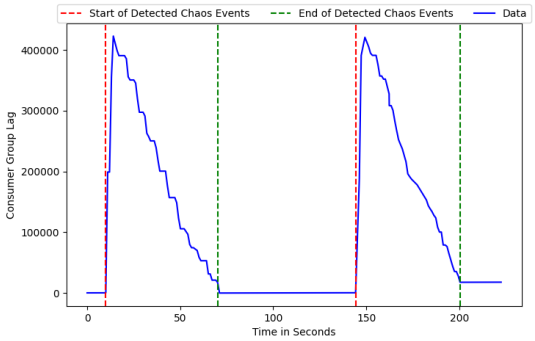
[13] Open-source stream processing framework (https://flink.apache.org).

[14] Open-source monitoring and alerting framework (https://prometheus.io).

[15] https://github.com/fsch-ppi/HMLC-benchmark



(a) Single Run of a Pod Failure



(b) Single Run of a Pod Failure with 400ms latency

Figure 2: Comparative Analysis of Pod Failures

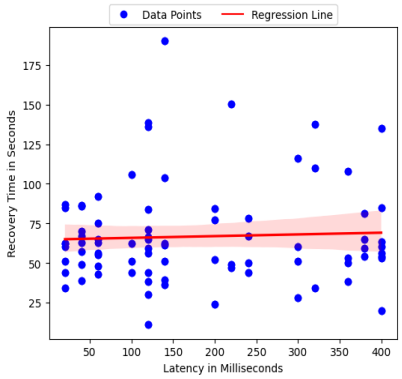


Figure 3: Failure Recovery Times for varying Network Latencies (Ex2)

When a distributed implementation for the HM-LC is available all experiments should be re conducted. Failure recovery times should vary in a distributed architecture [9].

In some runs, the fluctuation of the lag follows a clear pattern. This could be linked to the limited variance in messages and their processing order, given by the implementation. Despite that, this could also be a desired property, indicating that the HM-LC is working correctly, since it observes directly follows relations. When process mining is applied to real-world data, similar patterns may also be observed, given that events are not entirely random.

Generally, pod failures are always indicated by a linear increase in lag, before spiking, similar to [9]. This information could be useful for monitoring systems and real-time failure detection.

5 Outlook

To provide accurate results for industry applications, the HM-LC needs to be tested with a fully distributed implementation on an industry-grade system. Additionally, injecting time skews into the workers might change the process model derived from the HM-LC. To determine if results are still plausible, appropriate chaos tests would need to be conducted. The origin of the lag fluctuation should be investigated in a further study. If there really is a pattern linked to the messages being processed, training an algorithm to detect anomaly patterns based on the lag of a streaming process mining algorithm is possible.

Training an algorithm to detect pod failures in real-time is an exciting prospect. This study outlines patterns of pod failures, which could be applied to algorithms for all kinds of stream processing tasks, since [9] yielded similar results.

Acknowledgments

Hendrik Reiter has provided the implementation of the HM-LC algorithm.

References

[1] Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. 2016. Chaos Engineering. *IEEE Software* 33, 3 (2016), 35–41. https://doi.org/10.1109/MS.2016.60

[2] Andrea Burattin. 2022. *Streaming Process Mining*. Springer International Publishing, Cham, 349–372. https://doi.org/10.1007/978-3-031-08848-3_11

[3] Andrea Burattin, Alessandro Sperduti, and Wil M. P. van der Aalst. 2014. Control-flow discovery from event streams. In *2014 IEEE Congress on Evolutionary Computation (CEC)*. 2420–2427. https://doi.org/10.1109/CEC.2014.6900341

[4] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A catalog of stream processing optimizations. *ACM Comput. Surv.* 46, 4, Article 46 (March 2014), 34 pages. https://doi.org/10.1145/2528412

[5] Joshua Owotogbe, Indika Kumara, Willem-Jan Van Den Heuvel, and Damian Andrew Tamburri. 2024. Chaos Engineering: A Multi-Vocal Literature Review. arXiv:2412.01416 [cs.SE] https://arxiv.org/abs/2412.01416

[6] Hendrik K Reiter. 2024. *Scalability Benchmarking of the Realtime Heuristics Miner implemented as a Microservice Architecture*. Master's thesis. Kiel University.

[7] Li Su and Yongluan Zhou. 2021. Fast recovery of correlated failures in distributed stream processing engines. In *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems* (Virtual Event, Italy) *(DEBS '21)*. Association for Computing Machinery, New York, NY, USA, 66–77. https://doi.org/10.1145/3465480.3466923

[8] Wil M. P. van der Aalst. 2022. *Process Mining: A 360 Degree Overview*. Springer International Publishing, Cham, 3–34. https://doi.org/10.1007/978-3-031-08848-3_1

[9] Adriano Vogel, Sören Henning, Esteban Pérez-Wohlfeil, Otmar Ertl, and Rick Rabiser. 2024. A Comprehensive Benchmarking Analysis of Fault Recovery in Stream Processing Frameworks. 171–182. https://doi.org/10.1145/3629104.3666040

[10] Jianwei Zheng, Huimin Chu, Daniele Struppa, Jianming Zhang, Magdi Yacoub, Hesham El-Askary, Anthony Chang, Louis Ehwerhemuepha, Islam Abudayyeh, Alexander Barrett, Guohua Fu, Hai Yao, Dongbo Li, Hangyuan Guo, and Cyril Rakovski. 2020. Optimal Multi-Stage Arrhythmia Classification Approach. *Scientific Reports* 10 (02 2020). https://doi.org/10.1038/s41598-020-59821-7

[11] Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software unit test coverage and adequacy. *ACM Comput. Surv.* 29, 4 (Dec. 1997), 366–427. https://doi.org/10.1145/267580.267590