TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Aleksandr Madisson 221249IAPM

STATEFUL STREAM PROCESSING: A COMPARATIVE ANALYSIS OF APACHE FLINK AND KAFKA STREAMS FRAMEWORKS

Master's Thesis

Supervisor: Radu Irbe
M.Sc.

Tallinn 2024

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Aleksandr Madisson 221249IAPM

OLEKUPÕHINE VOOGTÖÖTLUS: APACHE FLINK JA KAFKA STREAMS RAAMISTIKE VOOGTÖÖTLUSE VÕRDLUS

Magistritöö

Juhendaja: Radu Irbe
M.Sc.

Tallinn 2024

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Aleksandr Madisson

17.05.2024

Abstract

The rapid growth of data volumes that must be processed in real-time introduces challenges for architectural decisions regarding the selection of appropriate streaming technology. Stream processing is a paradigm that is focused on handling continuous and unbounded data streams in real-time. Frameworks such as Kafka Streams, Apache Flink, and Apache Spark are built for fault-tolerant scalable data processing. Their key part is graph-based data shuffling to guarantee resilent and distributed data processing.

In this study, two prototypes were developed that use rule-based matching service to demonstrate the proposed approach's effectiveness in a distributed environment. Rule based matching service produces labeled data records where each labeled record is unique state. The main is to figure out how efficiently frameworks recover a state under a high load in case of unexpected faults. The study involves a series of four experiments, focusing on the systems' performance under different failure scenarios using automatic failure simulator called Chaos Mesh.

Each experiment includes the automatic replicas fault simulation: first experiment, 2 out of 8 replicas, and second experiment, all worker replicas. The experimental setup processes an input of 200000 1-kilobyte records per second, where records are sourced from a Kafka cluster. All experiments are executed within an Amazon Elastic Kubernetes Service (EKS) environment in the AWS cloud. Each experiment comes with a set of metrics such as input throughput, output through, lag trend, and CPU utilization to analyze rebalancing processes in case of automatic faults.

Experiment results show that the Apache Flink-based prototype performs with lower latency and faster state recovery. The study references related work results that demonstrate better performance for Apache Flink.

The thesis is written in English and is 67 pages long, including 6 chapters, and 38 figures.

Annotatsioon

Olekupõhine voogtöötlus: Apache Flink ja Kafka Streams raamistike voogtöötluse võrdlus

Reaalajas töödeldavate andmemahtude kiire kasv toob kaasa väljakutseid arhitektuursete otsuste tegemiseks seoses sobiva voogedastustehnoloogia valikuga. Voogtöötlus on paradigma, mis keskendub pidevate ja piiritlemata andmevoogude töötlemisele reaalajas. Sellised raamistikud nagu Kafka Streams, Apache Flink ja Apache Spark on loodud veatolerantseks skaleeritavaks andmetöötluseks. Need raamistikud pakuvad graafipõhist andmete segamist, et tagada paindlik ja hajutatud andmetöötlus.

Selles uuringus on välja töötatud kaks prototüüpi, mis kasutavad reeglipõhist sobitamisteenust, et demonstreerida väljapakutud lähenemisviisi tõhusust hajutatud keskkonnas. Reeglipõhine sobitamisteenus toodab märgistatud andmekirjeid, kus iga märgistatud kirje on unikaalne olek. Peamine eesmärk on välja selgitada, kui tõhusalt raamistikud taastavad oleku ootamatute rikete korral kõrge koormuse all. Uuring hõlmab nelja eksperimendi seeriat, keskendudes süsteemide jõudlusele erinevate rikete stsenaariumide korral, kasutades automaatset rikete simulaatorit Chaos Mesh.

Iga eksperiment hõlmab replikaatide rikete automaatset simuleerimist: esimeses eksperimendis 2 kaheksast replikast ja teises eksperimendis kõik töötajate replikad. Katse ülesehitus töötleb 200 000 1-kilobaidist kirjet sekundis, kus kirjed pärinevad Kafka klastrist. Kõik eksperimendid viiakse läbi Amazon Elastic Kubernetes Service (EKS) keskkonnas AWSi pilves. Iga eksperimendi juurde kuulub hulk mõõdikuid, nagu sisendi läbilaskevõime, väljundi läbilaskevõime, mahajäämus ja protsessori kasutamine, et analüüsida automaatsete rikete korral toimuvaid tasakaalustamisprotsesse.

Katsetulemused näitavad, et Apache Flinkil põhinev prototüüp töötab madalama latentsusega ja kiirema oleku taastamisega. Uuringus viidatakse varasematele töödele, mis näitavad Apache Flinki paremat jõudlust.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 67 leheküljel, 6 peatükki, 38 joonist.

List of Abbreviations and Terms

EKS	Elastic Kubernetes Service
ECR	Elastic Container Registry
EC2	Amazon Elastic Computer Cloud
MSK	Amazon Managed Streaming
AWS	Amazon Web Services
HA	High Availability
YARN	Yet Another Resource Manager
DAG	Directed Acyclic Graph
JVM	Java Virtual Machine
K8S	Kubernetes
MP	MapReduce
TB	Terabyte
UC	Use Case
JMX	Java Management Extension
API	Application Programming Interface
CPU	Central Processing Unit
IDE	Integrated Development Environment
IOT	Internet Of Things
VM	Virtual Machine
EBS	Elastic Block Store
EFS	Elastic File System
DAG	Directed Acyclic Graph
RAM	Random Access Memory
ABS	Asynchronous Barrier Snapshotting
RDD	Resilient Distributed Dataset

Table of Contents

List of Figures

1. Introduction

1.1 Background and Motivation

Software engineering is a diverse field that addresses various business domain problems. These domain-specific issues and their corresponding use cases necessitate in-depth analysis to determine the most suitable technologies for achieving optimal problem-solving outcomes. One rapidly growing area in software engineering is big data. Although big data is sometimes regarded as a marketing term, it encompasses complex data processing frameworks and datasets.

As the volume of produced data has dramatically increased over the years, modern technical solutions capable of processing massive amounts of data are required. The open-source community offers decent frameworks and tools, making it challenging to choose the best option. This research focuses on comparing the most suitable frameworks for specific use cases, particularly those that are designed for stateful real-time stream processing.

Stream processing use cases are relatively rare compared to typical problems that can often be solved with traditional batch processing or simple programs that don't require the MapReduce model. However, stateful stream processing assumes that real-time processing relies on previous states and unbounded data flow, which may indicate abnormal system behavior, such as fraud alerts in financial transactions within a specific time frame. Quick response can facilitate necessary actions and save time. Therefore, it is crucial to provide a technical analysis of the most suitable frameworks, assess their advantages and disadvantages, and examine the complexities involved in their application for stateful stream processing based on a given use case. This master's thesis offers a technical overview, benchmarks, and comparisons for appropriate use cases. The use case considers that incoming events trigger state aggregation and re-computation and evaluates how efficiently Kafka Streams [1] and Apache Flink [2] manage the state under heavy loads, represented by a stream of Kafka [3] messages. The ideal framework should address existing problems while offering excellent scalability, state recovery, and cost efficiency.

1.2 Problem Statement

The problem comes with business demands that are based on current needs for a more scalable and efficient solution. Before getting into the problem details, the current model will be provided to better understand why this research brings value and how streaming frameworks are comparable with the batch processing model 1.

1.2.1 Batch Processing Model Overview

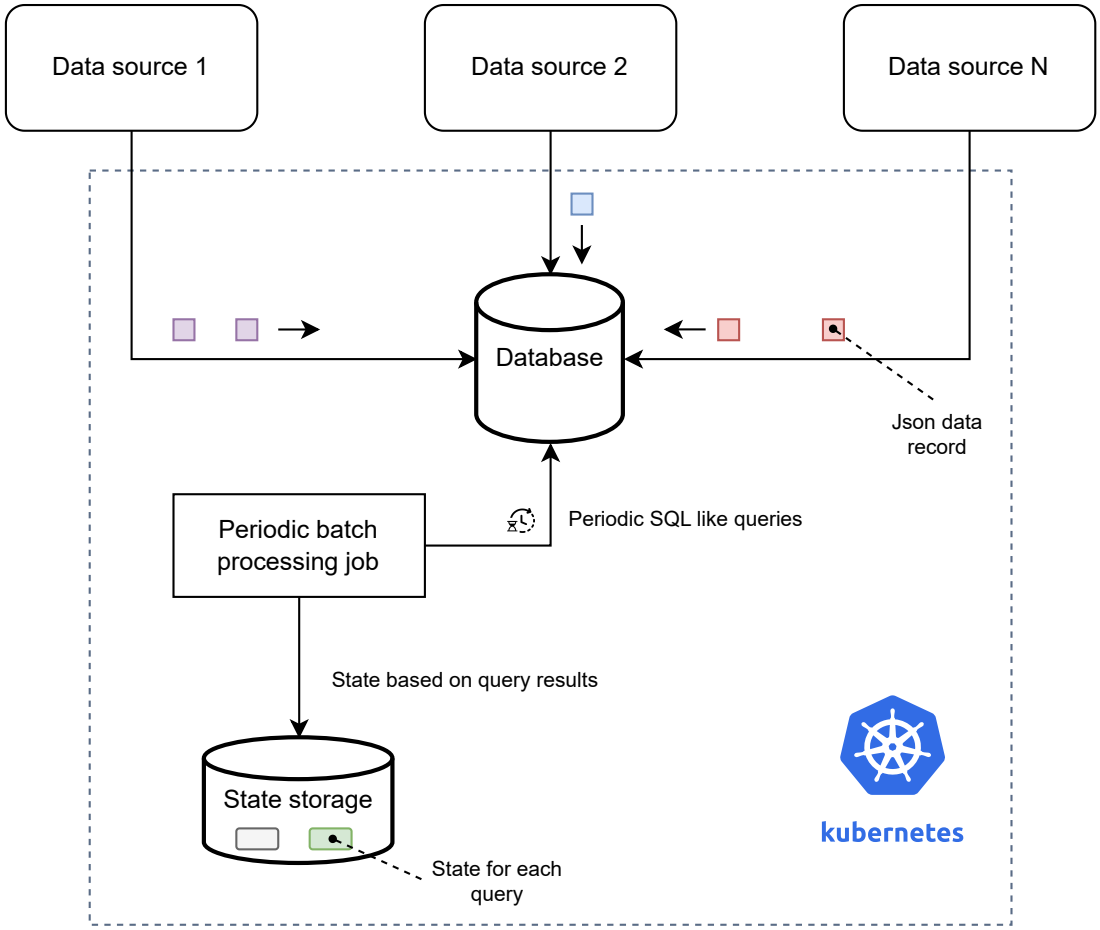

Figure 1. *Generic model of a batch processing job that should replaced by stateful stream processing.*

Kubernetes Open source platform for managing containerized workloads and services [4].

Data course N It is an external data source component that sends a json records as unbounded stream.

Batch processing job Represents a worker that executes queries in the databsase and saves a query result as a state to the state database. It runs with a predefined interval.

State storage Is a storage for states. Each query has its own state.

Database Stores records from data sources.

Some of the main disadvantages of the batch proccessing illustrated on Figure 1l. New stateful stream processing with Kafka Stream and Apache Flink should overcome these disadvantages, and provide a brand-new solution to process unbound data streams.

- The system is not scalable enough.
- The state gets lost it the job gets killed.
- Periodic batch processing has higher latency compared to stream proccesig that is designed to proccess data in real time with a minimal latency.

1.3 Research Question and Objectives

The main question is to figure out how different frameworks behave in case of fault tolerance, for example, if some processing replicas or workers get killed. How would load rebalancing work, how much delay would it create, and how long would it take for the system to restore a state? The research consists of several tasks.

- Create prototypes with Apache Flink and Kafka Streams.
- Prepare cloud setup to make experiments repeatable.
- Define metrics and benchmarks.
- Execute experiments.
- Collect results for analysis.
- Plot the data and make a conclusion.

1.3.1 Selecting a Suitable Framework

They're lots different frameworks out there available for solving different data streaming problems. Most of the frameworks were born on top of each other as a new generation solution [57]. Each new framework is trying to bring better performance and scalability. Down below is an evolution of streaming frameworks [5] with a brief description:

Apache S4 Apache S4 (Simple Scalable Streaming System) was an early stream processing engine developed at Yahoo! Labs. It was designed for unbounded data stream

processing which uses simple programming model based on a publish/subscribe pattern. Apache S4 became an open source project under Apache Software Foundation but eventually became inactive due to limited community adoption [6].

Apache Storm Apache Storm was a popular distributed stream processing framework. It was designed for real-time data processing and provided guarantees such as at-least-once and exactly-once processing semantics. Apache Storm was more popular comparing to Apache S4, but it provides less performance and flexibility comparing to next generation frameworks [7].

Apache Flink Originally was developed at the Technical University of Berlin. Apache Flink is one of the most powerful stream processing framework that unified batch and stream processing with focusing on streaming. Flink provided low-latency, high-throughput, and exactly-once processing semantics. With its advanced features, such as event time processing, watermarks, and savepoints, Flink has become the one of the most popular stream processing framework which is well-designed for highly loaded complex data stream processing use cases [2].

Kafka Streams Introduced as part of Apache Kafka, Kafka Streams is a rather a stream processing library than a framework which allows developers to build real-time applications and microservices using the Kafka platform. Kafka Streams provides a simple, functional programming model and is tightly integrated with the Kafka ecosystem. It's well-suited for use cases like real-time analytics, data transformation and event-driven architectures. Kafka Stream can a replacement for Apache Flink for cases where heavy integration and complex computation is not needed [1].

Apache Spark Is well known alternative for Apache Flink and Kafka Streams, is a popular solution for most of the cases, but does a micro-batching [8].

However, for this study Apache Flink and Kafka streams were chosen as most popular real-time processing framework which do not use micro-batching.

1.3.2 Deployment Environment

In 2024, the most popular and advanced application container manager is Kubernetes. First, stream processing frameworks were designed to be run on YARN clusters. YARN is no longer considered to be the preferred deployment manager. It means that a framework must be able to run in Kubernetes using containers. All experiments for this study were conducted with an AWS cloud provider.

1.3.3 Requirements Summary

Apache Flink and Kafka Streams are two leading stream processing frameworks that are used in prototypes.

As a summary, these important bullet points provided down below which will be considered during the evaluation for the new solution.

Scalability The current is not scalable enough.

Fault tolerance A state must not be lost if a pod where a running job gets killed. The system should be able to proceed stream processing with previous state once job is restarted.

Simple integration A solution, weather it is a programming language, or framework, should be compatible with the current technology stack, such as JVM and Kubernetes. It also means that a solution won't require hiring an entire engineering department or a team.

Cost effecienty It's quite important that a highly loaded solution doesn't cost too much.

Cloud independence A solution must not depend on a cloud provider.

Functionality A solution has to have an API that allows to implement and test a considered use case.

2. Theory Introduction

This chapter provides the theory behind Apache Flink and Kafka Streams, key concepts, illustrative examples, and main challenges. A reader should be able to understand the context and why such frameworks are needed. Both frameworks are based on principles of distributed systems design, which are all covered in the context of a given problem.

2.1 Scalability Problems

Let's define a simple task that is given by a business. For example, the business wants to know what songs get more streams in different countries. A straightforward solution is the code below 2.1.

```
top_charts = db.select("top_charts")
ordered_bands = top_charts
    .group_by(chart -> chart.country)
    .agreggate_by(chart -> chart.streams)
    .order_by(chart -> chart.streams)
    .map(chart -> {chart.streams, chart.song, chart.country})
```

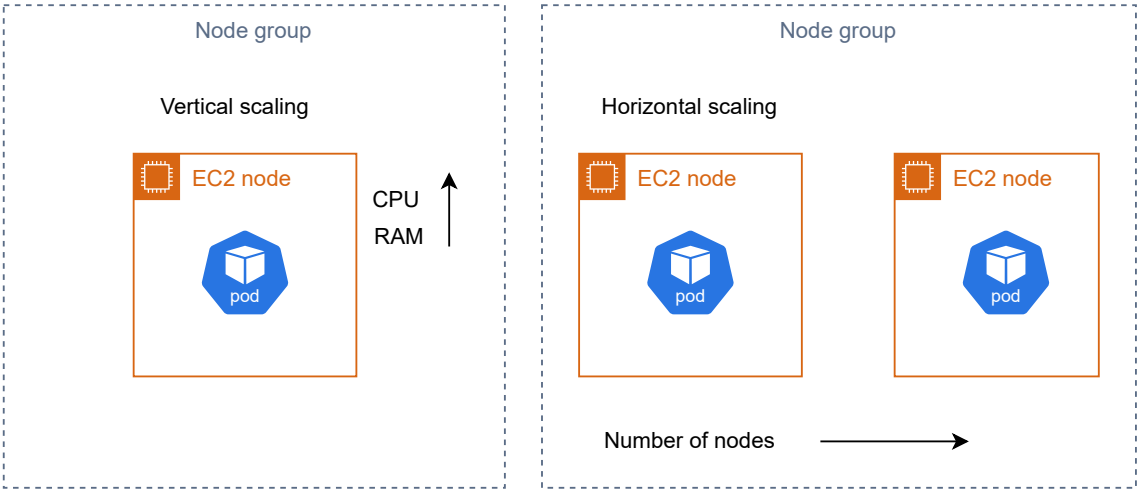

Figure 2. *Example of vertical and horizontal scaling where vertical is about using more powerful node and horizontal about adding more nodes.*

The first straightforward solution is to use a more advanced node [9] with more CPU, RAM, and Cores. Such a solution works until there's no such powerful node that can process high volumes of data. Even if such a node still exists, it might lead to processing

downtime if the node is down for some reason. Adding additional resources such as CPU and RAM refers to vertical scaling that scales less than horizontal scaling [10]. Example on Figure 2.

Even having a scaling setup, big data processing is not that straightforward to scale without having in an efficient distributed processing model. These models are MapReduce [11, 12, 13] and DAG [14, 15]. There are open-source solutions which implement one of model and successfully being used in industry [16, 17]. Moreover, they provide an api which look almost the same as map, filter, reduce functions.

Here are key points about frameworks for data pipelines such as Apache Flink, Kafka Streams and Apache Flink, that use complex DAG model under the hood.

Scalablity These frameworks are designed to be scalable by default. The main difference in scalability is that frameworks use models such as MapReduce, DAG. It means that frameworks know how to be scalable under the hood, using multiple nodes. Framework knows how to distribute sub-problems across multiple parallel workers and combines multiple sub-results into a single result. Execution performance is achieved by having multiple parallel workers that shuffle a data stream by key and distribute a load between each other.

Fault Tolerance Frameworks are designed to be tolerant to faults. The Common approach is replication, rebalancing, and a state reprocessing after recovery. However, they're different depending on a framework. For example, Kafka Streams is based on changelog topics [18], Apache Spark uses RDD or resident distributed datasets [19] , Apache Flink uses distributed snapshots, the algorithm is well covered in [20].

Kubernets operator Modern frameworks are designed to be run in the Kubernetes environment. Kubernetes operators are used to manage such complex deployments in production [21].

Built-in integrations For example, it can be machine learning pipelines, external data sources, graph processing algorithms [22, 23, 24].

2.2 Stream Processing

Stream processing is a concept that is dedicated to systems that need to process an infinite message or event, often called an unbounded dataset [25].

A stream has a quite broad meaning, it often gets referred to stdin and stdout of Unix programming languages [26]. For some developers, stream could mean file stream API and for an average person it would mean video content stream, for example on YouTube or on Twitch. In this case study, the stream refers to infinite dataset that represents a infinite dataset.

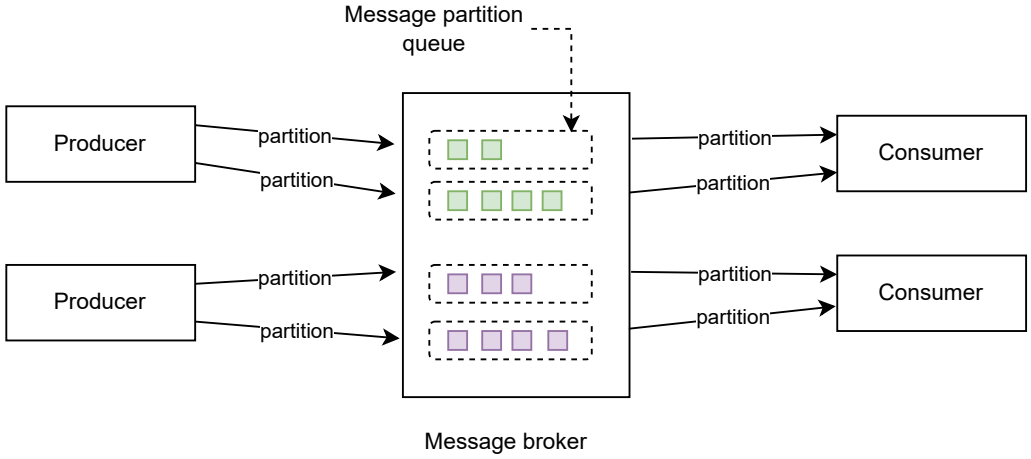

Figure 3. *This diagram illustrates a message queue system optimized for handling unbounded datasets. Each partition has its own queue.*

2.3 Stream Processing Challenges

Since this research is focusing on stateful streaming, a variety of challenges must be handled by a new solution. Detailed challenges are described in [8].

State Management In stateful stream processing systems, a state represents a unit of information stored, accessed, and updated by a key. The simplest example of a key-based state is a counter. Setting up state storage in distributed systems is challenging due to different state sizes, update frequencies, and storage types. Stream processing frameworks must ensure consistency and minimal latency while managing state across multiple workers. Frameworks come with built-in state backends and metrics exporters that expose information about the system performance. Example of state metrics: represent state size, memory usage, latency, and read-write metrics.

Fault tolerance It includes snapshot configurations, message commit time interval, storage configurations, state monitoring.

Scalability Since the load in data-intensive production systems may vary over time, it's crucial to design the system as auto-scalable. Auto-scalable systems automatically adjust the number of workers needed for incoming data in real-time. Having redundant workers might lead to additional expenses, while insufficient worker replicas lead to system crashes and slowness..

Rebalancing Repartition is a core feature that deals with fault tolerance and scalability in case of fault. It's part of the rebalancing process. The controller coordinator is responsible for repartition if a consumer is no longer responsive. An example is Kafka's consumer group [27].

Processing latency Apache Flink and Kafka Stream are able to handle a load in a reasonable time range, but might have different latency in case of stateful stream processing with different state sizes and input throughout. An example of such benchamarks [28].

Delivery guarantee The Most common types of semantics in stream processing are at least once semantics, and exactly once semantics that deal with a complex commit process. These processes are described here [29, 30].

Deployment At this time, streaming frameworks have already Kubernetes integration which allows managing deployment and resources. However, complex cloud configurations require deep expertise.

2.4 Directed Acyclic Graph Model in Stream Processing

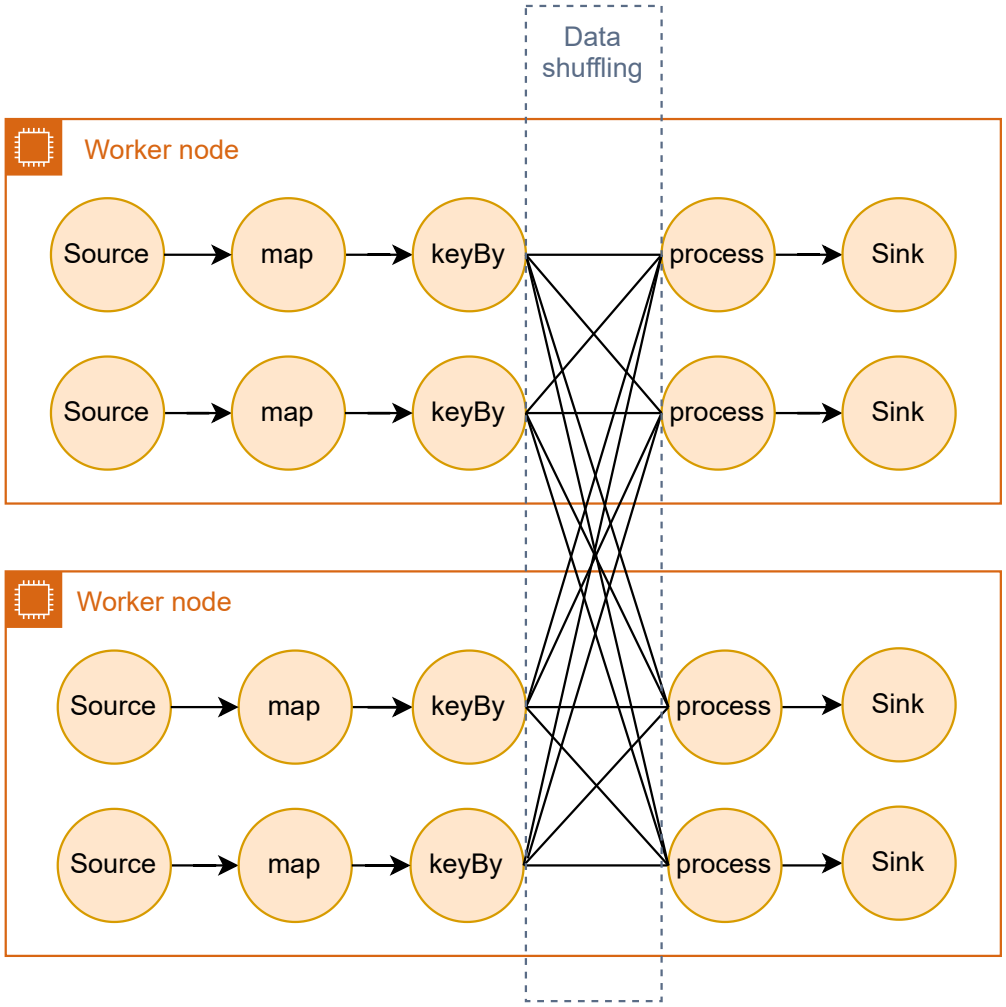

Figure 4. *This diagram illustrates how data flows from Source to Sink passing through various shuffling and processing stages using directed acyclic graphs. The most popular frameworks that use DAG model are: Kafka Streams, Apache Flink, Apache Spark, Hazelcast Jet, Apache Storm.*

In big data and real-time stream processing frameworks, processing and analyzing data as it flows through systems is crucial. Data streaming frameworks have emerged as powerful tools to handle such tasks, and at the heart of these frameworks lies a fundamental concept: the Directed Acyclic Graph (DAG) [20, 25, 14, 15]. DAG model is illustrated on Figure 4. Stream processing based on DAG consists of the following components [15, 26, 8].

Source A source is the component in a data streaming framework that ingests partitioned

streams of data from external systems, initiating the data processing pipeline.

map A map is a data transformation component that applies a specified function to each element in the input data stream, producing a transformed output stream. It is commonly used to transform data records from one form to another as they pass through the processing pipeline.

keyBy A keyBy component in a data streaming framework shuffles and partitions the data stream based on a key extracted from each data record, ensuring that records with the same key are sent to the same downstream task for processing.

process A process component in a data streaming framework is responsible for stateful processing, maintaining and updating the state as it processes each data record. This allows for complex event processing and aggregation based on the changing state.

Sink A sink is the component in a data streaming framework that sends processed and partitioned data records to an external system for storage, analysis, or further processing.

Data shuffling Data shuffling between the keyBy and process components in a data streaming framework involves redistributing the data across different partitions or nodes based on the key extracted from each data record. This is a crucial step for ensuring that records with the same key are processed together, enabling stateful operations and accurate aggregation.

2.5 Kafka Cluster

The use case in this research uses Kafka cluster of 3 brokers as a data source and data sink. Kafka cluster plays a crucial role in evaluation fault tolerance performance since both Apache Flink and Kafka Streams based prototypes represent a cluster of one Kafka consumer group.

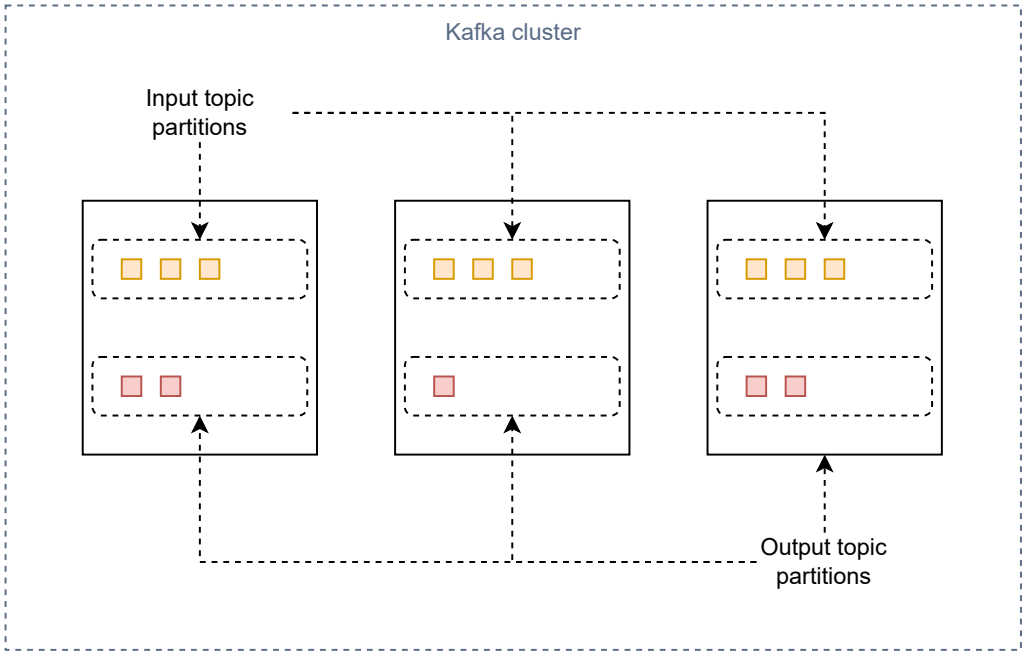

Figure 5. *Illustration of simplified Kafka cluster model that is uses in this case study. In the case study the cluster consists of 3 brokers and 2 topics with 50 partitions for each topic.*

While the model in Figure 5 illustrates a simple diagram, a real Kafka cluster is composed of advanced components such as leader election, a control plane for cluster management, and robust mechanisms for data replication and fault tolerance. This case study does not evaluate the fault tolerance of the Kafka cluster, but it highlights Kafka's extensive use as a core component for data processing. The Full description of Kafka cluster leader election and control is described in the official Kafka documentation [31]. Down below are the core components of the Kafka cluster that are described in details [27].

Kafka Consumer Group A Kafka consumer group is a collection of consumers that work together to read records from one or more Kafka topics. Each consumer in the group reads data from a different subset of the partitions, allowing for parallel processing and load balancing.

Kafka Consumer A Kafka consumer is a client application that subscribes to one or more Kafka topics and processes the data records published to those topics. Consumers read data in real-time and can be part of a consumer group for distributed processing.

Kafka Producer A Kafka producer is a client application that publishes data records to Kafka topics. Producers send records to specific topics and can partition data across multiple partitions within those topics for scalability and load balancing.

Kafka Broker A Kafka broker is a server that hosts Kafka topics and manages the storage, retrieval, and replication of data records. Brokers handle the incoming data from producers, serve data to consumers, and ensure data durability and fault tolerance through replication.

Leader Election Leader election in Kafka is the process by which one broker is designated as the leader for each partition. The leader is responsible for all reads and writes for the partition, ensuring consistent and orderly data management. Other brokers that store replicas of the partition data act as followers.

2.6 State Recovery

State recovery is crucial in distributed stream processing systems to ensure fault tolerance and consistency. When a system encounters failures, it must recover its state and resume processing without data loss or inconsistency. This section delves into the state recovery mechanisms that are used in Apache Flink and Kafka Streams, focusing on distributed snapshots and change logs.

2.6.1 Distributed Snapshots

Distributed snapshots are essential for capturing a consistent global state of a distributed system. The Chandy-Lamport algorithm, introduced by K. Mani Chandy and Leslie Lamport in their 1985 paper "Distributed Snapshots: Determining Global States of a Distributed System," provides a robust method to achieve this [32]. Distributed snapshots are foundational in stream processing frameworks like Apache Flink, enabling them to maintain accurate states across distributed nodes by consistent state recovery and fault tolerance. Later, the ideas behind the Chandy-Lamport algorithm were used to implement Asynchronous Barrier Snapshotting [20] algorithm for Apache Flink. ABS algorithms quarantine consistence snapshots and are especially efficient in case of frequent snapshotting starting from 2 seconds period. These benchmarks are provided in [20]. Since the case study uses Kafka as a data source, snapshots keep track of the latest committed

message to ensure state consistency. This allows for rerunning state updates by processing uncommitted Kafka records during recovery.

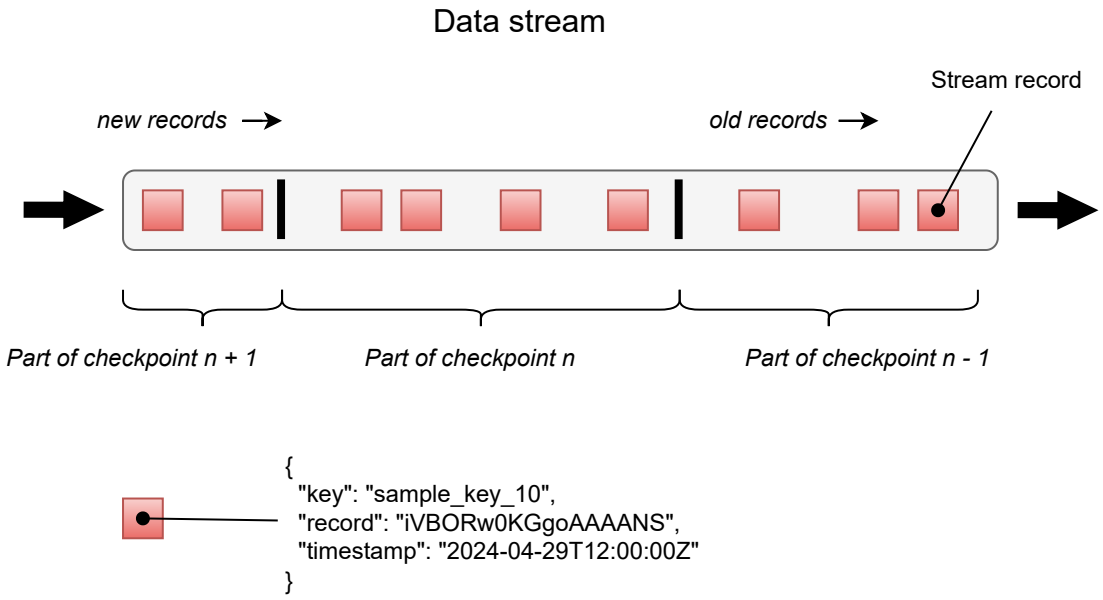

Figure 6. *Illustration of snapshots applied for data stream.*

Simplified stream snapshotting period is illustrated on Figure 6. The ABS algorithm performs asynchronous micro-pauses to capture records currently being processed by a worker. This ensures a proper snapshot is taken and saved in the configured storage, maintaining data consistency and enabling efficient state recovery. The storage and snapshot interval for the case study is illustrated on Figure 7.

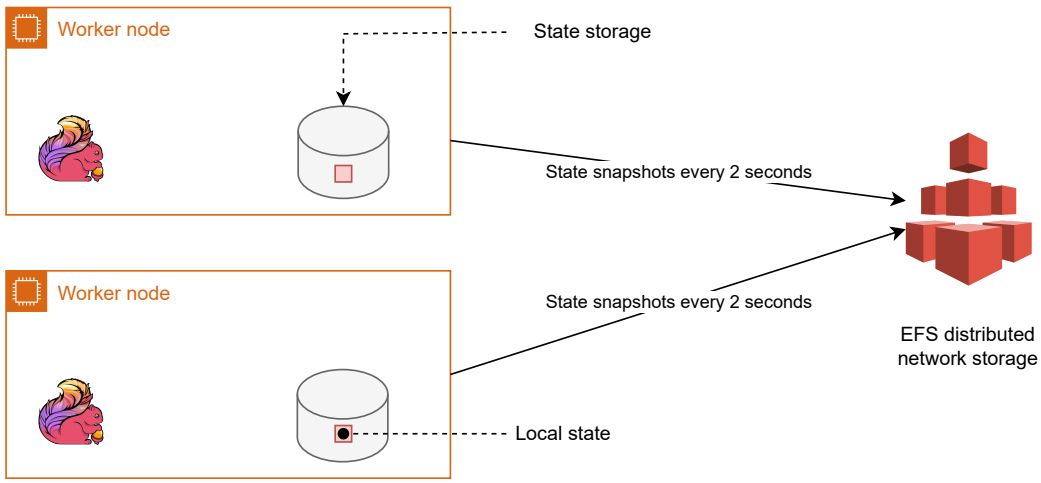

Figure 7. *Illustration of snapshots for an Apache Flink-based prototype in this case study. Snapshots get stored to AWS EFS network storage.*

2.6.2 Change Logs

In Kafka Streams, state stores are used to keep track of accumulated state, such as counts, sums, or other aggregations. These state stores are backed by change logs, which are special Kafka topics that record every update to the state store [1].

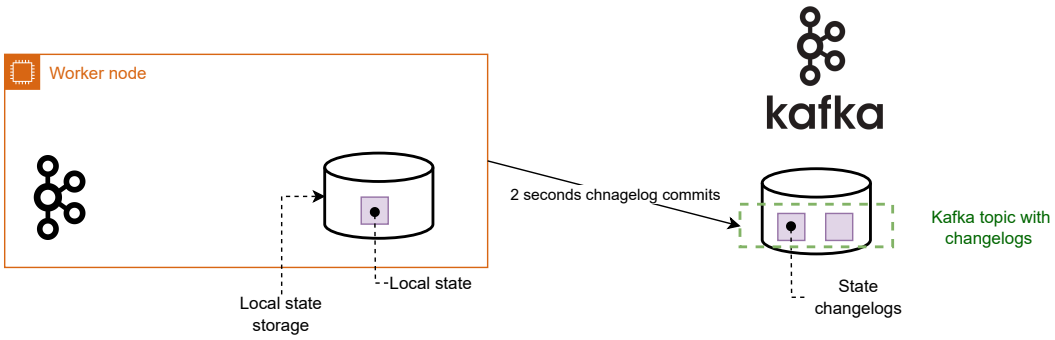

Figure 8. *Illustration of changelogs for Kafka Streams-based prototype in this case study. Changelogs get stored to Kafka cluster.*

Illustration on Figure 8 demonstrates Kafka Streams stage management model. Change log topics record every update to the state store. It includes any inserts, updates, or deletes within the state store. Each state change is logged as a new record in the change log topic. If the application crashes, it can replay the change log topic from the beginning to reconstruct the state store with all user sessions up to the point of the crash. By capturing every state update as an ordered log of key-value pairs, Kafka Streams algorithms ensure that stateful stream processing applications can recover from failures and continue processing without data loss. Using 2 seconds commit interval makes the case study comparison more accurate since Apache Flink based prototype uses 2 seconds snapshot interval.

2.7 Rule Based Matching Service

The core component of prototypes in this case study is the rule based matching service. The goal of such a service is an emulation of rules based logs matchers. Each rule represents a logical consumer that has a certain propability matching with an incoming data record. The core component of prototypes in this case study is the rule-based matching service. The goal of such a service is to emulate rule-based log matches. Each rule represents a logical consumer with a specific matching probability with an incoming data record.

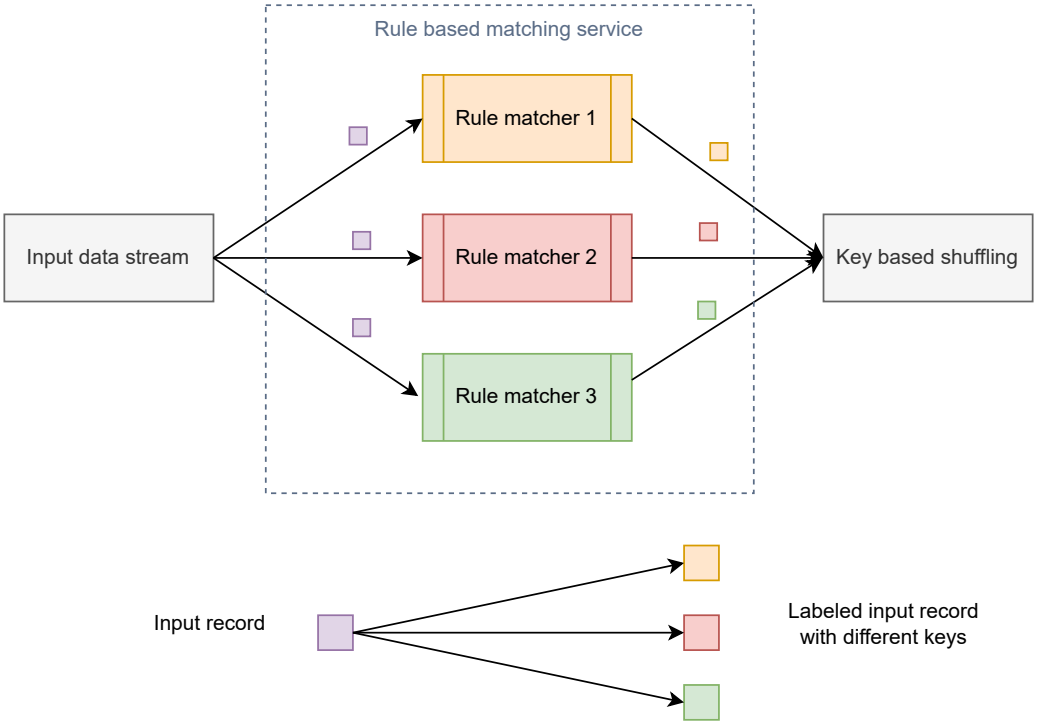

Figure 9. *Illustration of the rule based matching service.*

Rule-based matcher in illustrated on Figure 9. Each incoming record is processed through all the rule matchers. For each match, a new keyed record is generated, where the key represents the record's label. All rule matchers have a fixed matching probability in this use case simulation. For example, if there are 10 rules and the matching probability is 0.1, then it is expected that at least 1 rule will match per record, resulting in at least one labeled record for further shuffling and processing. If the matching probability is 1, then 10 labeled records get shuffled and processed for 1 incoming record. All rule matchers are based on the seed that guarantees experiment replication. The real use case might be implemented with Intel Hyperscan (see Figure [33]), but this case study does not focus on that.

3. Methodology

This chapter describes experiment setup and performance evolution methods that are used to demonstrate performance difference.

3.1 Introduction

This chapter describes the methods that are used in this research. The main goal of the thesis is to figure out how Apache Flink and Kafka Streams behave and perform, what configurations are needed, and how to solve fault tolerance problems in case of a stateful streaming. They both are designed for use in a streaming domain, but there must be some difference in how they perform and get configured to run in the Kubernetes cluster. The key moments for both systems are scalability, resilience, fault tolerance, cost efficiency, setup complexity, documentation, and learning curve. To get experiments running, it is crucial to get an execution environment, especially for big data solutions, which are not possible to run on a laptop. Big data solutions and benchmarks require a cluster of hardware machines that represent nodes. For this reason, all the benchmarks and execution environments are done with AWS and EKS in particular. AWS provides services such as Amazon Managed Streaming. It solves a lot of configuration problems, but it adds additional cost. It should be suitable for less-loaded streaming solutions where load demand is not that big and there's no dedicated team. Heavy-loaded services where latency is also essential rely on their own Kafka cluster setups and streaming setup, such as Spark Streaming or Apache Flink. These experiments are based on the execution configuration provided by the Theodolite framework. Theodolite provides an essential customizable Kafka cluster setup, which is easy to reconfigure and has more control over running experiments, making it easy to deploy infrastructure in EKS [34]. Theodolite [35], a robust framework, plays a pivotal role in acquiring measurable benchmarks. It offers a suite of tools, including Grafana [36] and Prometheus Operator [37]. These tools are equipped with ready-to-use and configurable Prometheus [38] pod monitoring agents, enabling the collection of various metrics from running pods within the EC2 node [9] [39]. The Theodolite operator, in conjunction with Prometheus and Grafana, is specifically designed for easy deployment in a Kubernetes cluster environment, providing super low latency real-time metrics.

One of the most important step is a data analysis, which is also possible using Theodolite framework since it provides metrics export interface. All the recorded metrics will be analysed using Python libraries and tools [40] [41] [35].

3.2 Research Technical Tasks

Two frameworks, Apache Flink and Kafka Streams, are written in Java and designed to run in a cloud environment. Both prototypes are written in Java. For experiments, AWS was chosen since it is one of the most advanced cloud providers. AWS provides services for running scalable streaming solutions such as EBS [42], EFS [43], EC2 [39], EKS [34]. The research is split into following steps:

Prototypes development Development of two Java based prototypes with Apache Flink and Kafka Streams which process the same input and send result to the same output. Both prototypes are based on Java 11. Input and output are Kafka topics with 50 partitions.

Tools selection AWS as cloud provider, EKS as kubernetes cluster, Theodolite [35] [44] that is one of the most advanced frameworks for setting up streaming benchmarks in cloud environment. Theodolite includes out of the box Grafana [36], Prometheus [38] [37], Kafka [3], Apache Flink and Kafka Streams cloud benchmarking cloud configurations provides as Helm charts [45]. Chaos Mesh [46] is chosen as it provides scheduled job failures in a cloud environment.

Kubernetes cluster design The cluster must have different node groups for different components. For this study were used 3 main node groups, worker node group, infra node groups, generator node groups, depicted on Figure 10.

Metrics definition Metrics include node network traffic, CPU consumptions, consumer group lag, lag for each partition, state recovery time. All metrics are implemented with PromQL [47].

Experiments execution All experiments are defined with Theodolite execution files [48] that allows to rerun experiments with the same execution conditions.

Results collection Results from PromQL execution are saved in Kubernetes by Theodolite as scv files.

Results analysis CSV files get analysed with Matplotlib Python library.

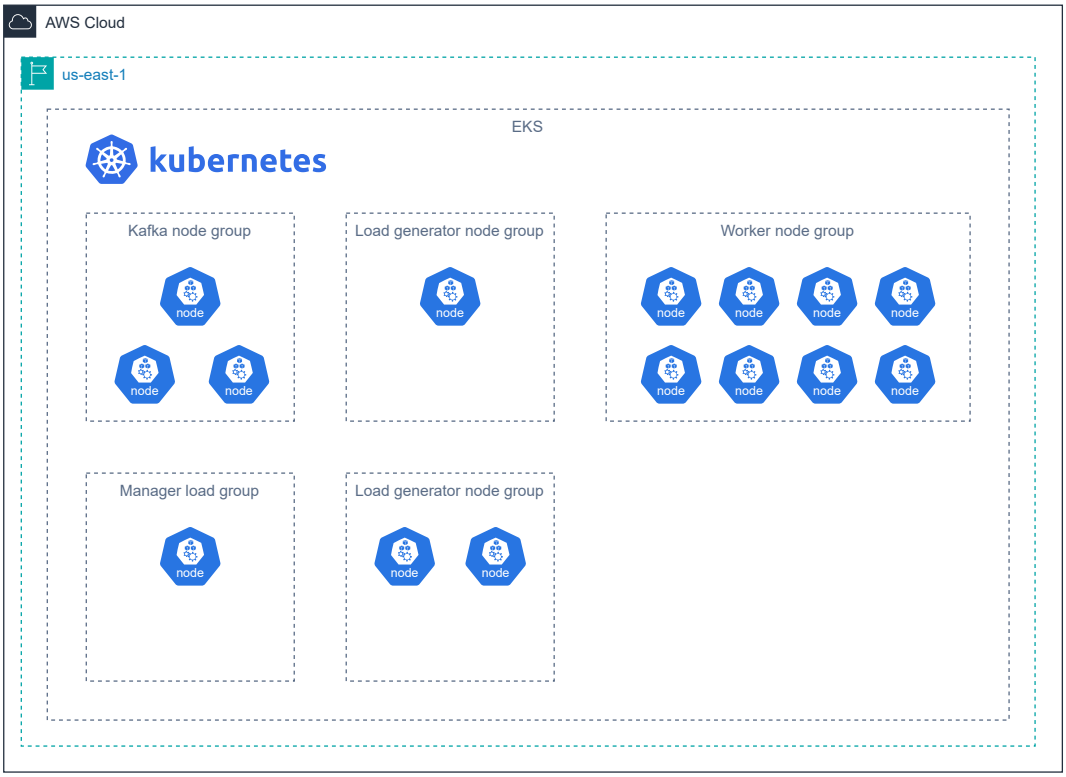

Figure 10. *Node groups for the case study experiments.*

3.3 Kubernetes Cluster Setup

This section is describing Kubernetes cluster and key components which are running with during experiments execution to collect metrics.

3.3.1 EKS Node Groups

As shown on Figure 10, the setup is based on 5 node groups and EC2 nodes [9].

Kafka node gorup 3 Kafka brokers, where each broker gets deployed to a separate node. Such that several brokers never run on the same node. Node types for this group is m6i.2xlarge [9].

Load generator node group Is Java based application which generates Kafka messages of 1Kb size to Kafka input topic with 50 partitions. In this study two load generator instances are used where each generates 100000 messages per second. Node types for this group is m6i.xlarge.

Manager node group The following node groups is intended only for Apache Flink task

manager [49] instance of 1 replica and is not used for experiments with Kafka Streams.

Infra node group This node group is used for benchmarking tools deployment such as EBS, EFS controllers, Grafana, Theodolite operator, Prometheus, Chaos Mesh and additional operators which are used only for data collection and visualisations. Should not affect workers and load generators. Node types for this group is m6i.xlarge.

Worker node group This node group is used for Kafka Streams setup and Apache Flink task managers [49]. Such that stream processing replicas are running on a separate nodes, where each node has only one worker replica running in the node. Detailed worked groups is shown on Figure 11. For this case study were chosen 8 nodes with 8 workers replicas. Node types for this group is c5.large.

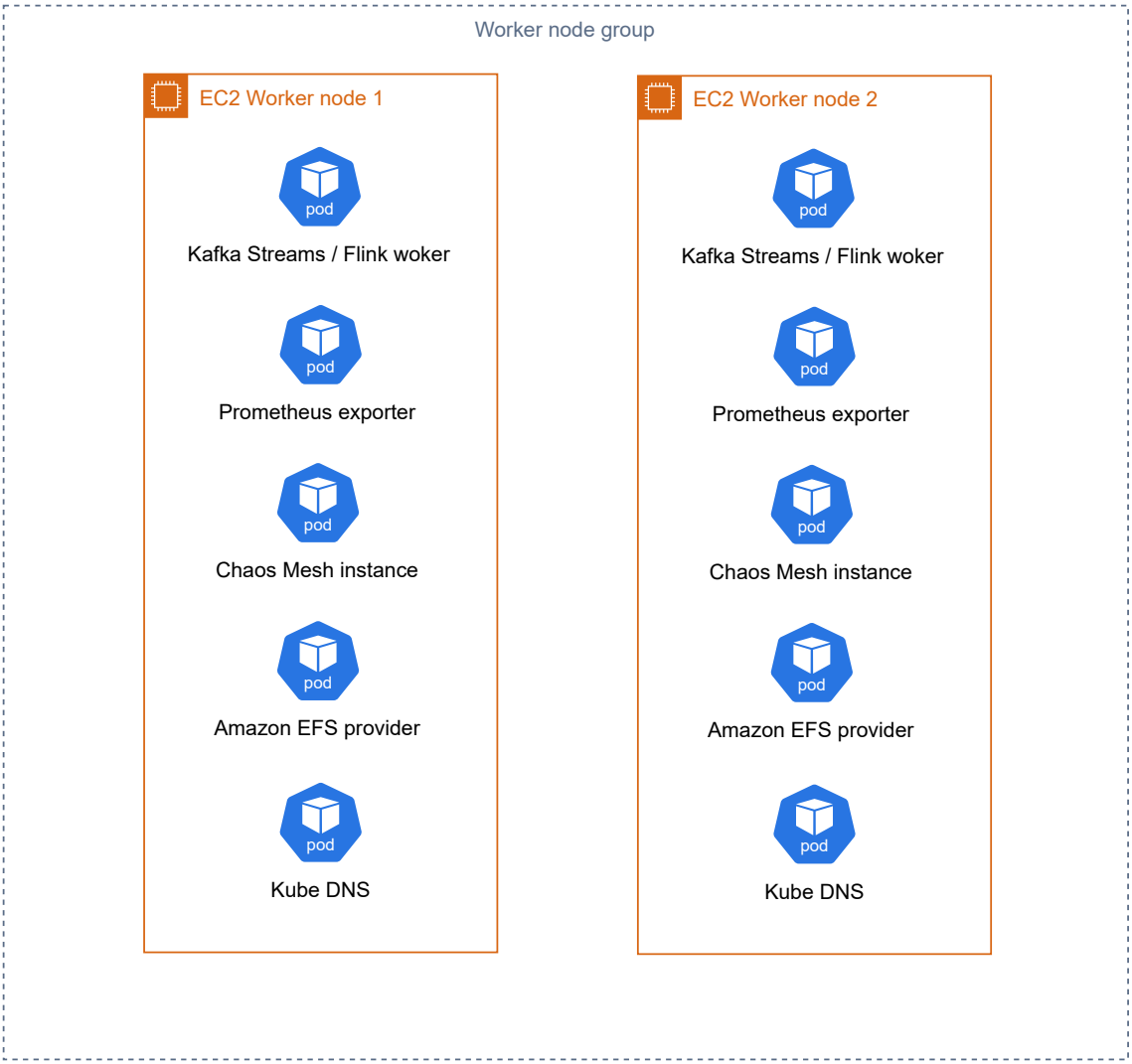

Figure 11. *Worker node group with 2 nodes and pods in each node.*

3.3.2 EFS and EBS Storage Services

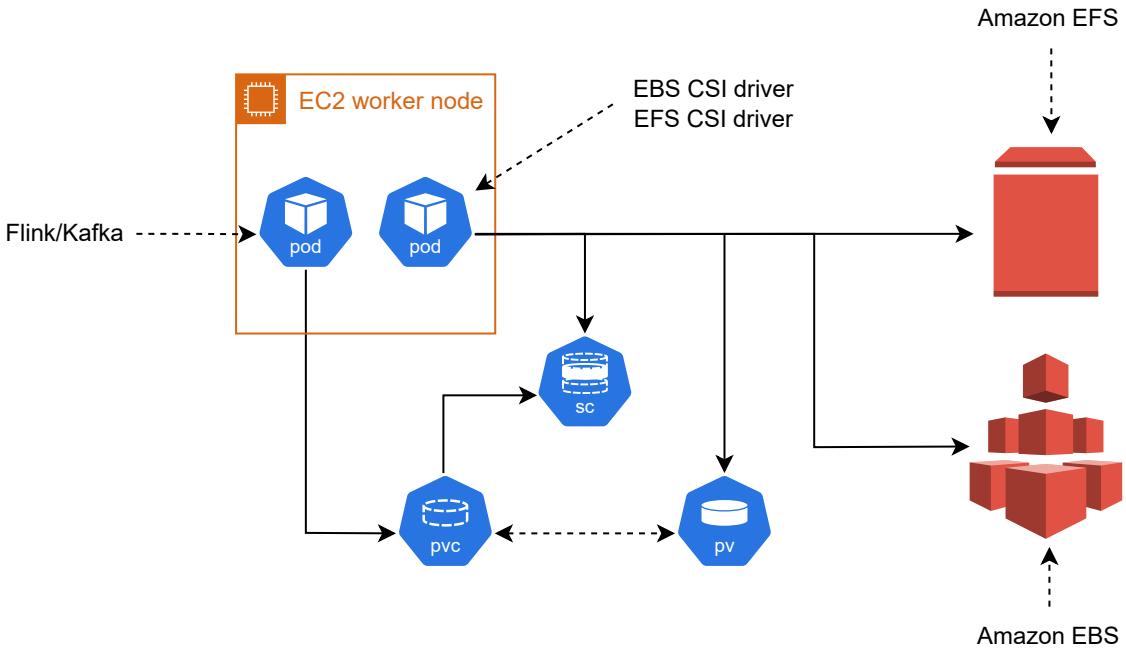

Figure 12. *EBS and EFS network storage diagram.*

Stream processing uce case is data intensive in this study. To be able to handle data loads during experiments were chosen two storage services for storing the data. These storage are EFS [43], and EBS [42]. AWS provides networks storage types for EC2 nodes [39] which are used for storing a big amount of data. The crucial part of this network storage is that if worker on node goes down for some time, then saved state won't be lost. A real physical node is connected to the network storage via a network configuration as it shown on Figure 12 which is also described in details in the following documentation [50]. Such a complex model gives pod a network connection to a physical storage such that huge amount of data gets written and read fast enough for production systems. The diagram on Figure 12 includes several the following components.

Flink and Kafka pods These are Java application which use networks storage. For thi study Flink workers use EFS storage for storing stages and Kafka brokers use EBS for Kafka topics.

EBS and EFS CSI drivers Cloud tool provided by AWS to manage network connection between nodes and networks storage. Driver runs on the same node as Kafka brokers and Flink worker to get network storage access.

PV Persistent volume gives networks access to a physical storage, where the state gets saved.

PVC Persistent volume claim is Kubernetes abstraction that connects to PV.

3.4 Metrics Exporters

This section cover how a Theodolite framework [35] is used to get metrics and to run experiments in cloud environment.

3.4.1 Kafka Metrics Exporters

Theodolite comes with a set of tools which are used to export metrics from running pods.

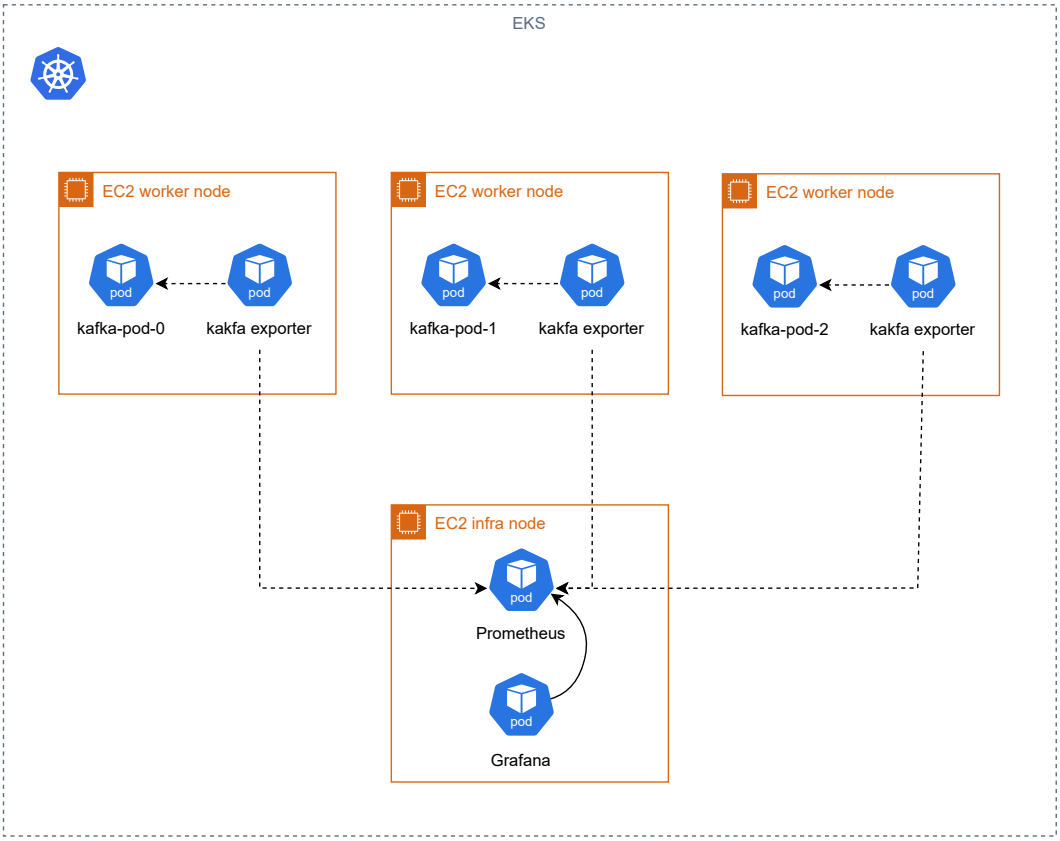

Figure 13. *Kafka metrics exporter that sends metrics from JMX exporter to Prometheus.*

On Figure 13 is an example of Kafka metrics exporter. It reads metrics from Kafka's JMX exporter about consumer groups, commit lag, topics, topic partitions, topic offsets.

3.4.2 Kubernetes and Worker Metrics Exporters

Kubernetes setup in this study is coming with Kubernetes metrics services [51] which are configured by Theodolite. Metrics diagram is depicted on Figure 14.

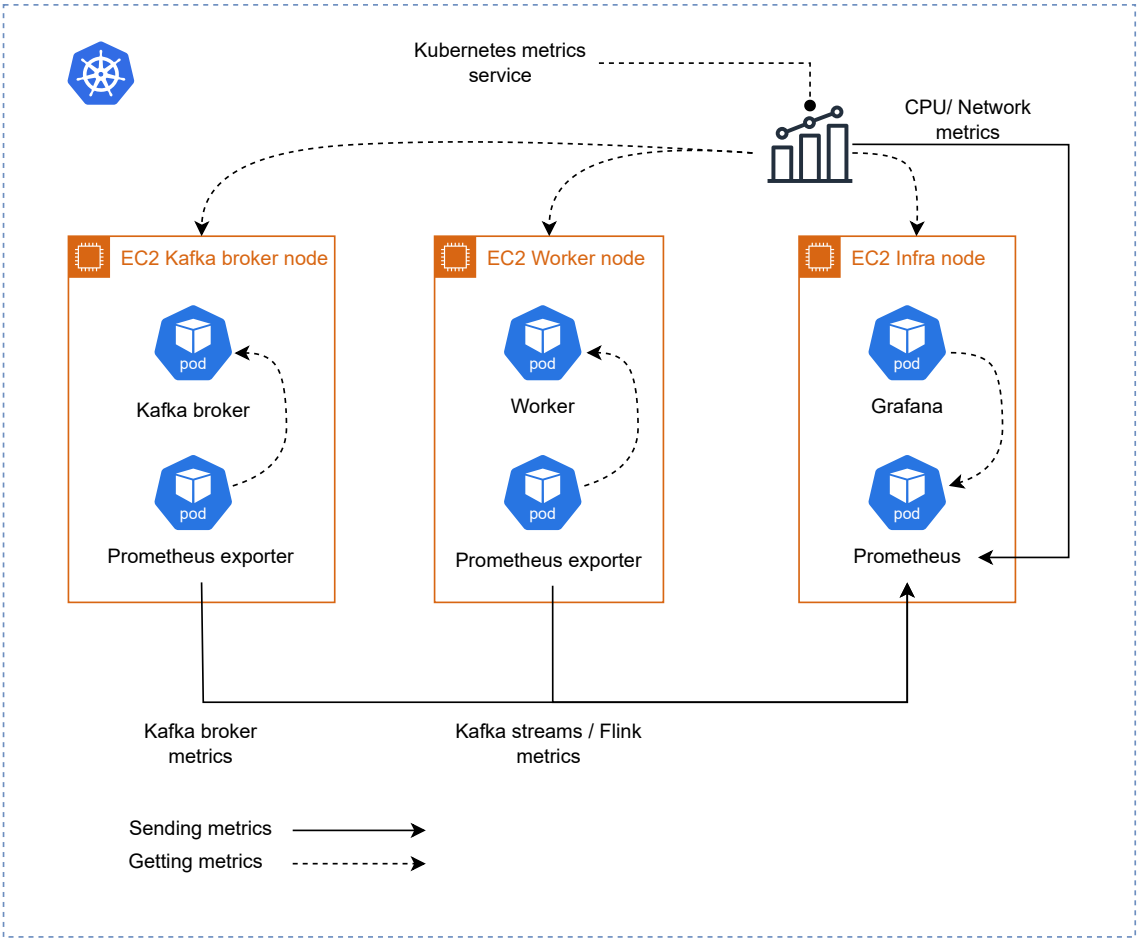

Figure 14. *Kubernetes and worker metrics exporters.*

Kubernetes metrics services expose metrics about node network utilization and CPU consumption. All exposed metrics get saved to Prometheus that makes them accessible by executing PromQL. For example, Grafana queries PromQL queries to plot real time charts with used cluster resources.

3.4.3 Latency Exporter

The Latency exporter measures the latency between when a message was generated by load generator and when it was committed to a Kafka log. Provides positive and negative latencies as metrics that are available in Prometheus as p50, p90, p95. These latencies get calculated by micrometer [52].

Positive Latency If the latency is positive (a message commit time is later than event time).

Negative Latency If the latency is negative (a message commit time is earlier than event time)

3.5 Benchmarks Setup

Theodolite is responsible for a full lifecycle of benchmarks execution with a set of Kubernetes deployment configurations. The General diagram is depicted on Figure 15.

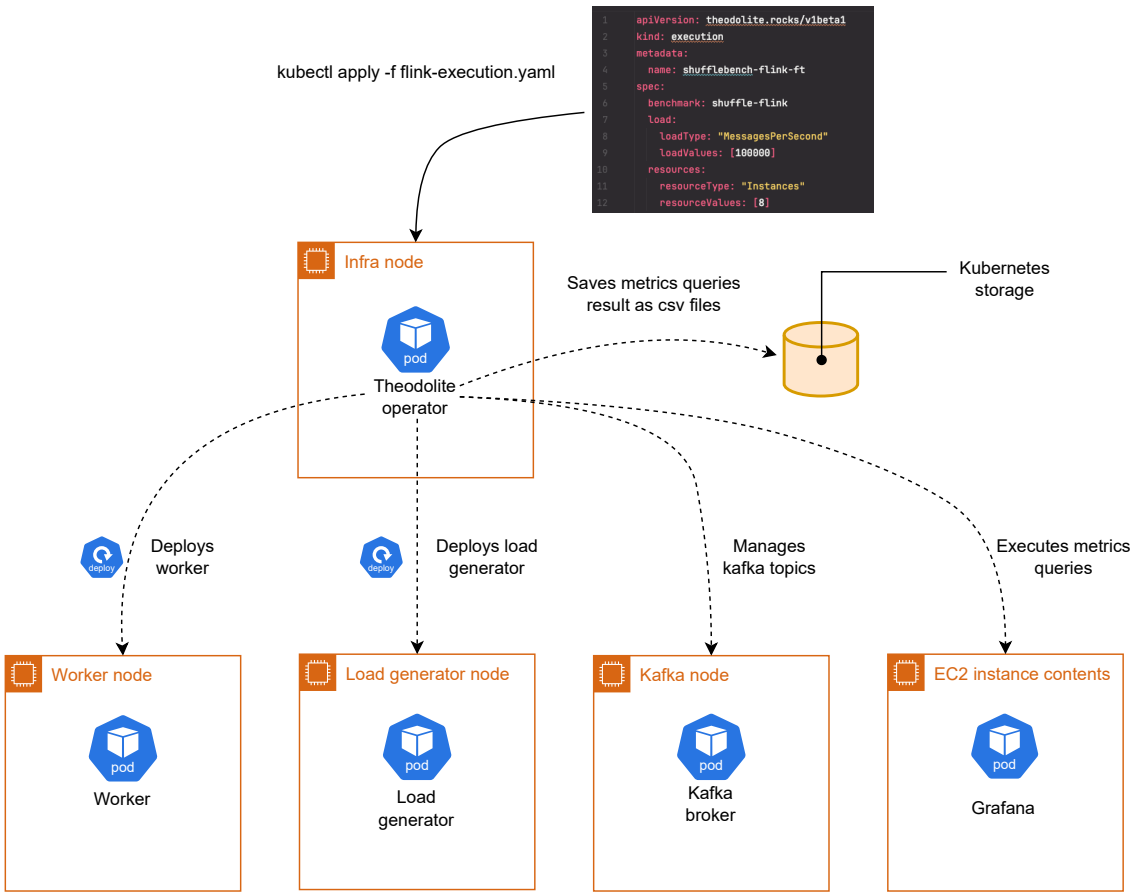

Figure 15. *Theodolite manages deployment and undeployment of load generator and workers during experiment execution.*

To run execution with Theodolite, some preconfiguration has to be made. Theodolite needs to get access to Kubernetes deployment files for Kafka Streams, Apache Flink. Deployment for load generator and monitoring tools comes with Theodolite our of the box. For custom usage, it is possible to override some deployment configurations. Apache Flink and Kafka Streams deployment files have to be stored in Kubernetes as config maps [53]. After each benchmark execution, Theodolite automatically undeploys load generator and workers.

3.6 Chaos Engineering with Chaos Mesh

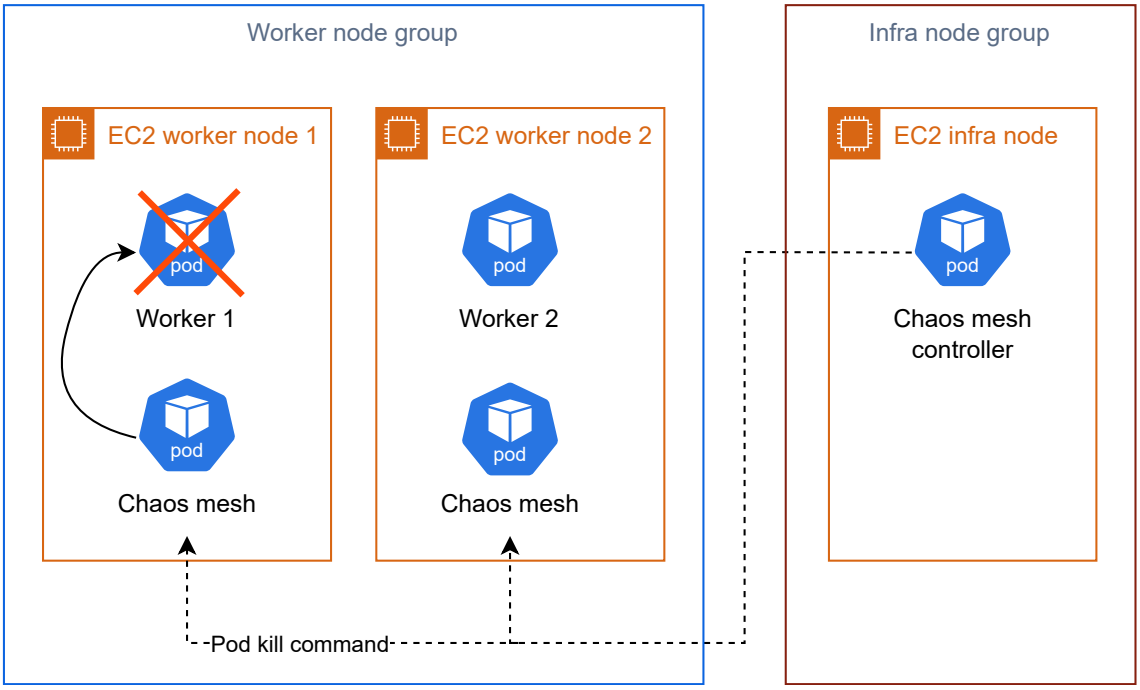

Figure 16. *Example of Chaps Mesh periodic Pod kill. Chaos Mesh controller selects Chas Mesh daemon to and sends a message to kill the worker.*

Chaos Mesh [46] is a tool used in the Kubernetes environment to simulate different kinds of failures. In this case study it is used to kill worker node by using pod selector. Each deployed worked has its own label, for example, type:worker. For all experiments, Chaos Mesh is configured to kill worker pods every 3 minutes. An example use of Chaos Mesh in this case study is depicted on Figure 16. Chaos Mesh installs Chaos daemon on each worker node, such that has Chas Mesh contoller has access to worker pods for failure simulation. However, Kubernetes quite fast redeploys killed worker replicas, but this time is more than enough to find valuable metrics about how the system would behave. For example, Kafka Streams and Apache Flink use different re-partitions algorithms to handle load balancing. Depending on re-partitions strategies, this case study can show this difference by measurement a stare recovery time.

3.7 Experiment Setup

This section describing experiment execution steps to show how to deploy benchmarks setup and get benchmarks as csv files from scratch.

3.7.1 Prerequisite

To get started with benchmarks execution, some steps need to be finished first.

Prototype compilation Build prototypes source code.

Docker images Once code is compiled, Docker images need to be built and deployed to ECR [54] service which is AWS image storage.

Kubernetes Deployment configuration Configure deployment configuration files for Apache Flink and Kafka Streams prototypes.

Built Docker images and configured Kubernetes deployment files are prerequisite to get started with Theodolite and EKS deployment.

3.7.2 EKS Cluster

Execute command with eksctl tool EKS cluster configuration in cluster.yaml 3.7.2.

```
eksctl create cluster -f cluster.yaml
```

Once the cluster is created, EBS and EFS network storage have to be configured. For EFS the following installation guide needs to be done first [55]. The Next step is applying EBS and EFS storage configuration 3.7.2.

```
kubectl apply -f kafka-storage-class.yaml
kubectl apply -f flink-storage-class.yaml
```

3.7.3 Theodolite Configuration

Prerequisite is values.yaml with defined node selector for EKS node groups, it should not be installed to worker and load generator nodes.

```
helm install theodolite theodolite/theodolite -f values.yaml
```

These Kubernetes config maps load deployment files to Kubernetes config such that Theodolite has access to files within Kubernetes.

```
kubectl create configmap --from-file ./shuffle-load-generator/
kubectl create configmap --from-file ./shuffle-latency-exporter/
kubectl create configmap --from-file ./shuffle-kstreams/
kubectl create configmap --from-file ./shuffle-flink/
```

These commands tell Theodolite how to access config maps created in a previous step 3.7.3.

```
kubectl apply -f theodolite-benchmark-kstreams.yaml
kubectl apply -f theodolite-benchmark-flink.yaml
```

At this step, deployment configurations should be ready to be used during experiments.

3.7.4 Running Experiments

To run experiments, Thedolite is listening for execution files to be applied but kubectl. Theodolite Kubernetes execution file represents instruction for Theodolite operator about how to run experiments. For example, what services to deploy, how log should experiment be running, environmental variables, how many repeats have to be. Experiment execution could be described in the following steps:

Appy execution config Once Theodolite operator found the execution config, it's starting deploying components which are need to be running only during the experiment execution. In this case study, these are: load generator, latency exporter, Kafka Streams or Apache Flink workers replicas.

Running experiment Based on the execution config, the experiment is running for a certain time period, in this case study it's 18 minutes.

Finishing experiment Once the execution time has passed, Theodolite operator undeploy, deployed services at execution start, executes PromQL queries to get benchmarks from Grafana, and saves them as CSV files.

Examples of csv files with recorded benchmarks produced by Theodolite operator. Each experiment has its own set of benchmarks.

```
exp3_managerNodesDiskReadMB_2s_1.csv
exp3_generic_latency_p90_30s_1.csv
exp3_kafkaBrokerNodesCPUsPercentageUtilization_2s_1.csv
exp3_workerNodesCPUsPercentageUtilization_2s_1.csv
```

4. Experiments Results and Findings

4.1 Introduction

This section is covering experiments results that were gotten in this case study. The main idea behind benchmarks and metrics in the case study is to understand the following details:

- How fast two systems recover from unexpected faults.
- Partition balancing behavior in case of unexpected faults.
- How much network traffic do systems use.
- How much CPU is used to find most effective nodes for workers, production cost depends on the type of nodes.

All experiments were executed in Kubernetes environments, which means all replicas that get periodically killed by Chaos Mesh automatically restarted by Kubernetes. Both prototypes based on Kafka Streams and Apache Flink use the same Kafka records commit time which is 2 seconds for all experiments. Such record commit time is supposed to achieve similar stateful streaming behavior for both prototypes. For this case study were chosen two main scenarios, when 2 of 8 cluster workers are killed for a short period and when all workers 8 of 8 are killed. Such scenarios should give an overview about how two systems behave if part of workers get killed and a whole worker cluster.

- Kafka Streams with having 8 worker replicas killed every 3 minutes
- Kafka Streams with having 2 worker replicas killed every 3 minutes
- Apache Flink with having 2 worker replicas killed every 3 minutes
- Apache Flink with having 8 worker replicas killed every 3 minutes

For each experiment were chosen the following parameters:

Workers 8 replicas

Semantics At least once semantics

Experiment execution time 18 minutes

Records per second 200000

Record size 1KB

Number of Kafka topic partitions 50

Chaos Mesh failure period 3 minutes

Number of states 1000

Kafka Streams commit time 2 seconds

Apache Flink checkpoint save period 2 seconds

4.1.1 Benchmarks

Input Records per Second records that are coming from a Kafka input topic.

Output Records per Second processed records that get sent to a Kafka output topic.

Lag Trend denotes the difference between the last record produced by the producer and the offset committed by the consumer group. Defines how fast a cluster of workers is able to process an incoming load from a Kafka input topic. Such a benchmark combines all partitions that are assigned to a workers consumer group.

Lag Trend per Partition denotes Lag trend for each partition separately.

CPU Utilization denotes how much CPU worker node consumes during the experiment in percentages.

Network Recevied denotes how much network traffic in MB worker nodes received during the experiment.

Network Transmitted denote how much network traffic in MB worker nodes transmitted during the experiment.

Definitions of vertical red and green lines.

Red vertical line moment when workers get killed by Chaos Mesh.

Green vertical line moment when the system produces the expected output record rate that is 100k records per second.

4.2 Benchmarking Kafka Streams Fault Tolerance

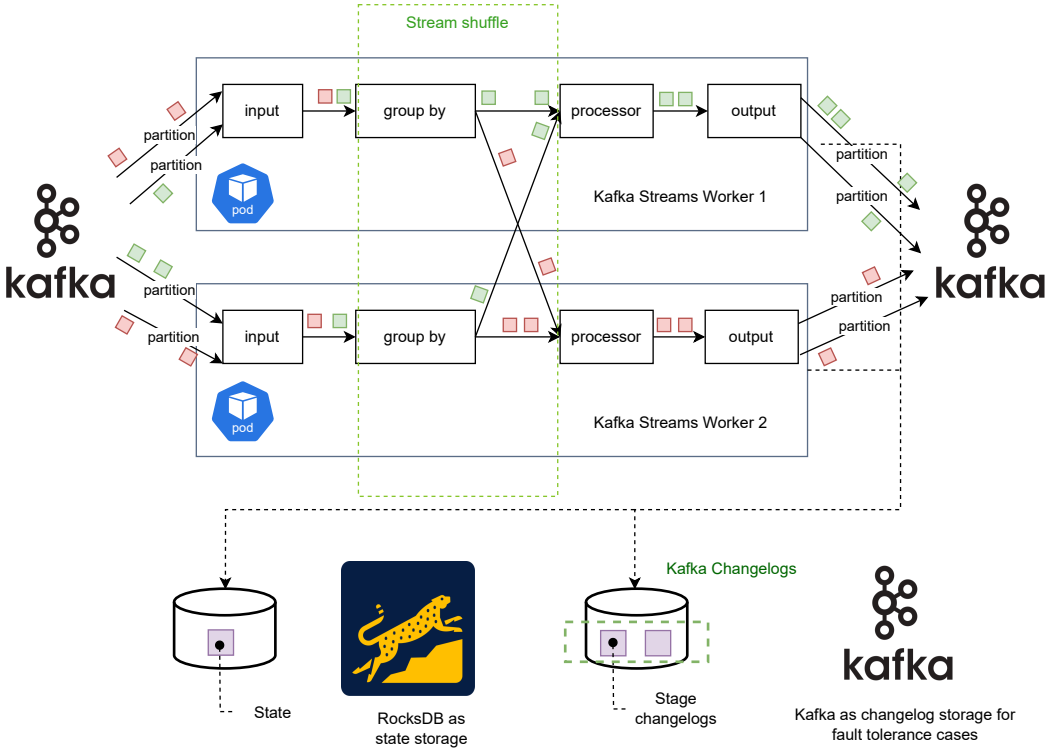

Figure 17. *Illustrative example of Kafka Streams workers for stateful stream processing. implemented model also includes record match service between input and grouping blocks.*

The model on Figure 17 shows fault tolerance model which is used for experiments with Kafka Streams. Each Kafka Streams worker has its own local state. In case of a fault tolerance, the state gets restored using a Kafka changelog topic. Kafka Streams workers are intended to run only with Kafka cluster.

4.2.1 Analyzing 2-Pod Failures in an 8-Pod Cluster

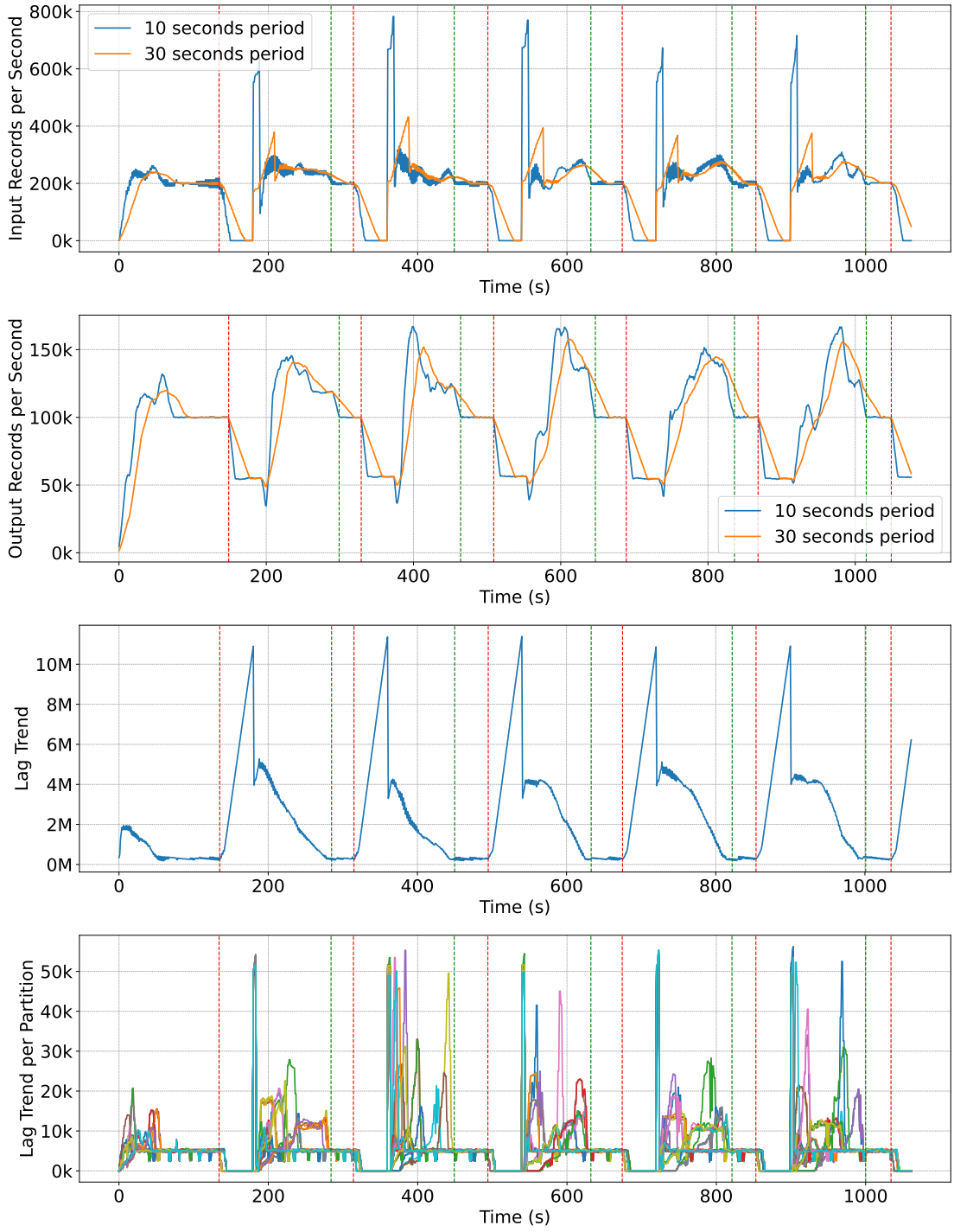

Figure 18. *Benchmarks for Kafka Streams experiment in case of 2 workers failure. Red vertical line is a start of the failure green vertical line is a moment when the system is back to a normal state and producing expected load of records.*

This set of benchmarks on Figure 18 contains four different benchmarks which describe system behavior for several repetitive failures.

The fault tolerance process based on benchmarks could be described in the following way.

Workers consumer group gets stopped During the failure, the input load and output load get stopped for a short period of time. At this moment, the consumer group is busy with rebalancing cluster workers and repartition Kafka input topic. Rebalancing process also includes state recalculation based on last commited records.

Consumer group lag increasing While a consumer group is not polling new records, the number of uncommited records in the input topic is actively growing. The input topic is still getting populated by the Load generator while the consumer group is being repartitioned. In can be seen on the Lag Trend chart. On the Lag Trend per Partition repartition can be seen for many partitions.

Repartition is finished Workers in the consumer group start actively polling new records such that the number of records in the output topic is actively growing to process records that have come while the consumer group wasn't polling.

The system is getting back to normal processing state The cluster of workers is able to process the previously uncommited records and new incomming records. Delayed records processing is finished once the output is at 100k records per second.

From the green vertical line to the next red line, the system is processing the load in the normal state. At the next red line, the rebalancing process repeats. Benchmarks show that rebalacing looks relevantly identically for all recorded failures. However, such frequent failures are not expected to happen in production. Such a rebalancing process might look the same way in case of cluster worker redeployment with a new worker version which is working under a high load. The process is also known as draining or draining in Kubernetes [4].

Another set of benchmarks on Figure 19 show consumed resources for all workers during the experiment. These benchmarks show how a network traffic and CPU utilization change during a state recovery. Network traffic keeps being stable while the system is in a normal processing state. The average time from a fault to a normal execution state is about 143 seconds.

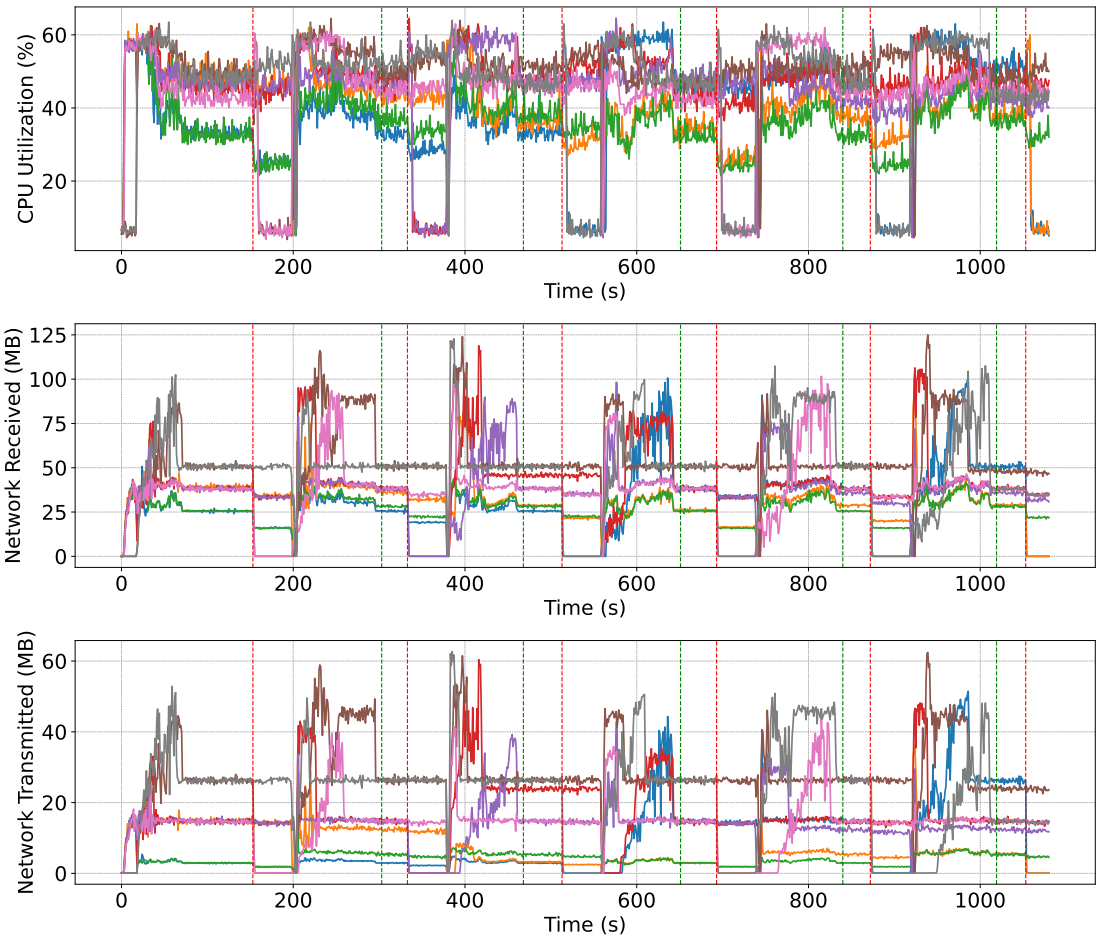

Figure 19. *Resources consumption during rebalancing a state recovery in case of 2 worker failures. Red vertical line denote a start of a fault, and the green vertical line is when the system gets back to normal state.*

4.2.2 Analyzing 8-Pod Failures in an 8-Pod Cluster

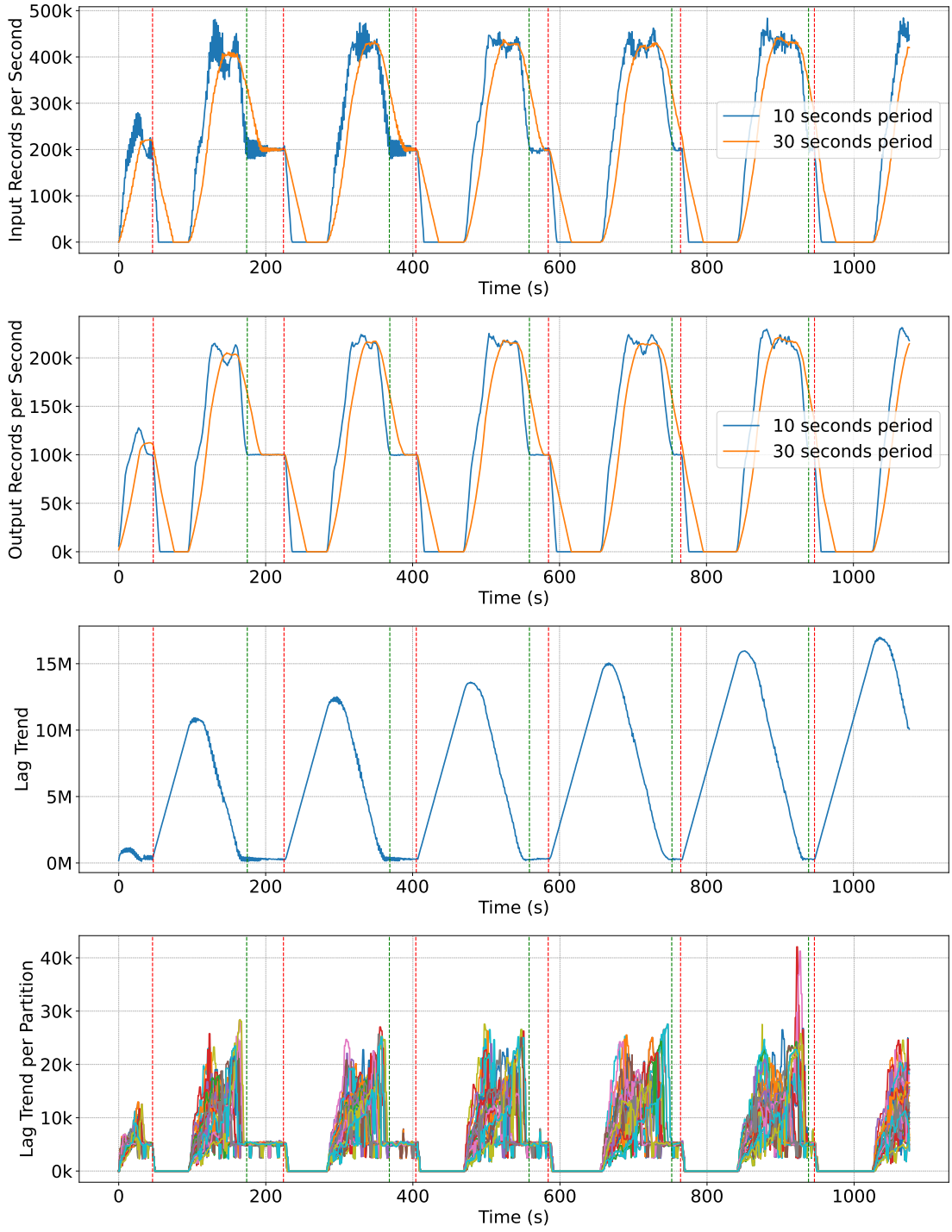

Figure 20. *Benchmarks for Kafka Streams experiment in case of 8 workers failure. Worker cluster is fully killed. Red vertical line is a start of the failure green vertical line is a moment when the system is back to a normal state and producing expected load of records.*

This experiment is trying to get metrics for the case when all workers get killed. It has the same rebalacing life cycle as described in the first experiment. 4.2.1 According to

benchmarks on Figure 20 The average time from a fault to a normal execution state is about 153 seconds, that is only 10 seconds longer than in experiment 4.2.1. However, network resources consumption is more extensive since all workers are involved Figure 21. Also, all partitions are involved into repartition, which can be seen on Figure 20 for Lag Trend and Lag Trend per Partition.

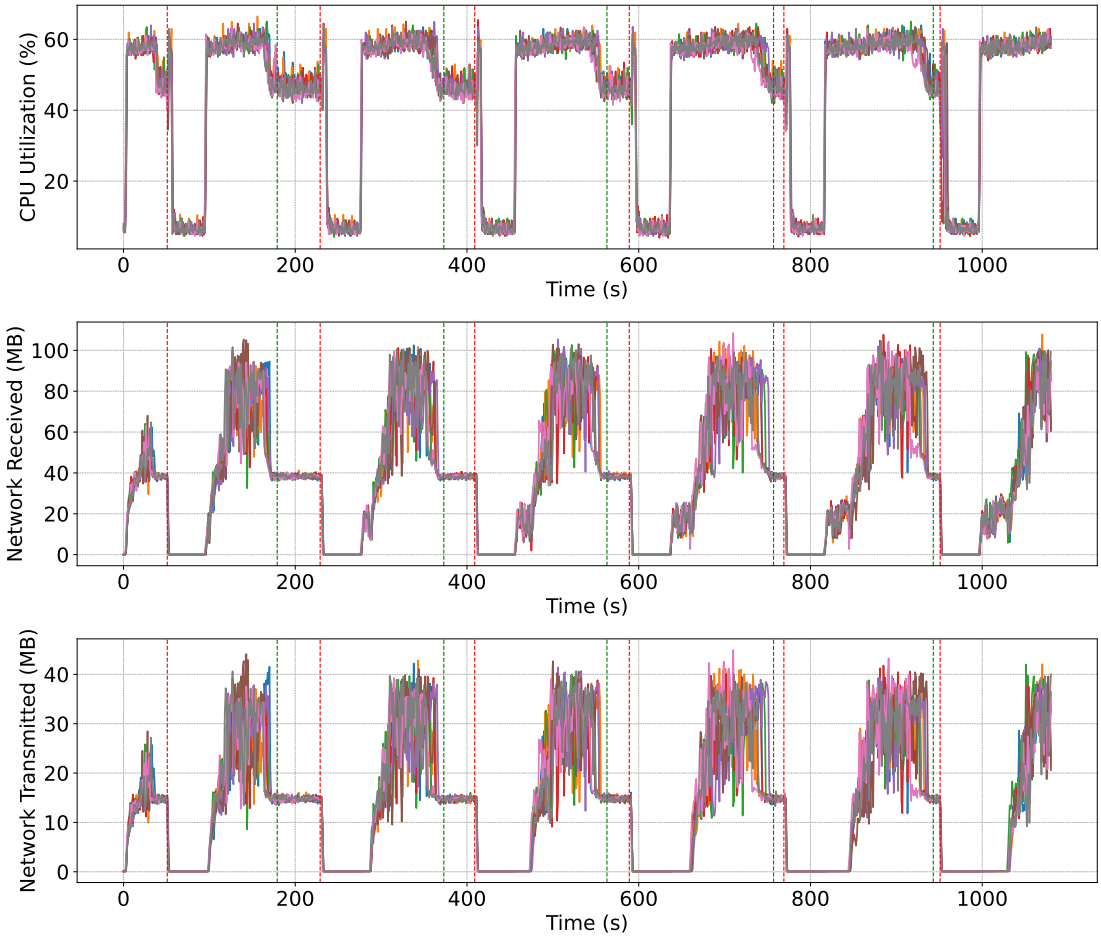

Figure 21. *Resources consumption during rebalancing and a state recovery in case of all workers failure. Red vertical line denote a start of a fault, and the green vertical line is when the system gets back to normal state.*

4.3 Benchmarking Apache Flink Fault Tolerance

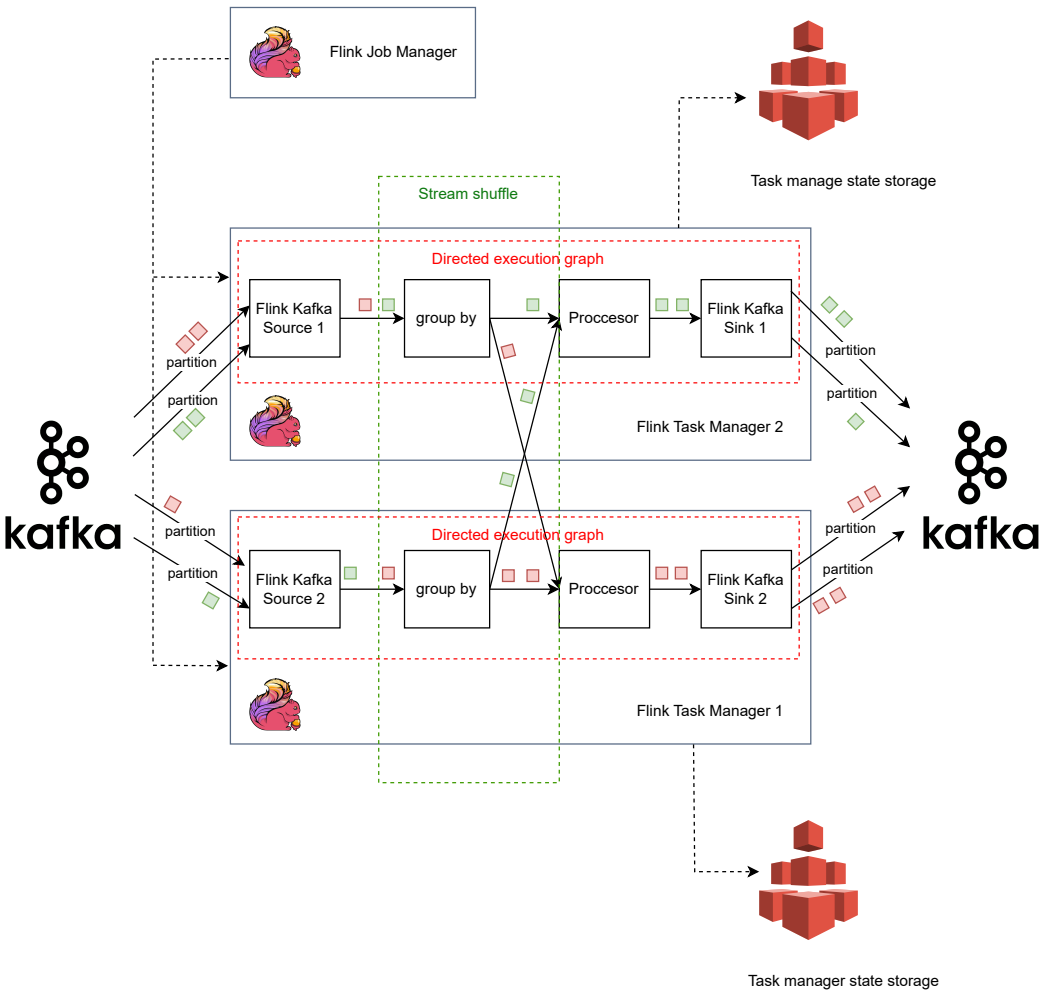

Figure 22. *Illustrative example of Apache Flink workers for stateful stream processing. implemented prototype also includes record match service between input and grouping blocks.*

The prototype model on Figure 22 is based on Apache Flink. It uses the same Kafka input and Kafka output topics as Kafka Streams prototype on Figure 17. Important difference for Kafka Streams implementation is that Flink uses checkpoints for state recovery. In this prototype, the state checkpoints are stored in EFS network storage which is mounted and connected to Flink workers using Kubernetes and AWS storage configuration.

4.3.1 Analyzing 2-Pod Failures in an 8-Pod Cluster

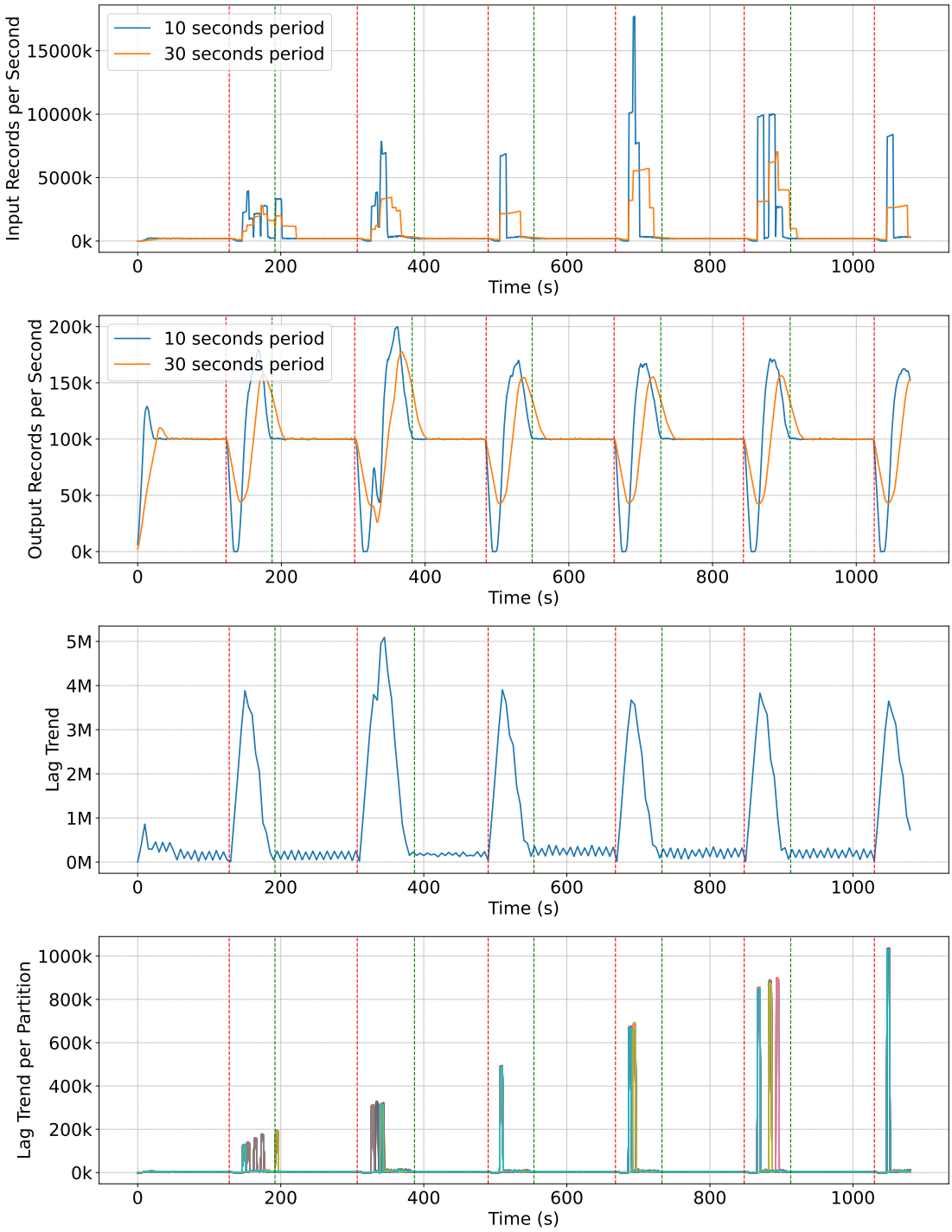

Figure 23. *Benchmarks for Apache Flink experiment in case of 2 workers failure. Worker cluster is fully killed. Red vertical line is a start of the failure green vertical line is a moment when the system is back to a normal state and producing expected load of records.*

Balancing process is also the same as described in 4.2.1, Since Flink worker cluster is also a Kafka consumer group that's polling records from a Kafka input topic. On the Figure 23

depicted the same benchmarks as on Figure 18. Benchmarks on Figure 24 denote resource consumption. The average time from a fault to a normal execution state is about 113 seconds.

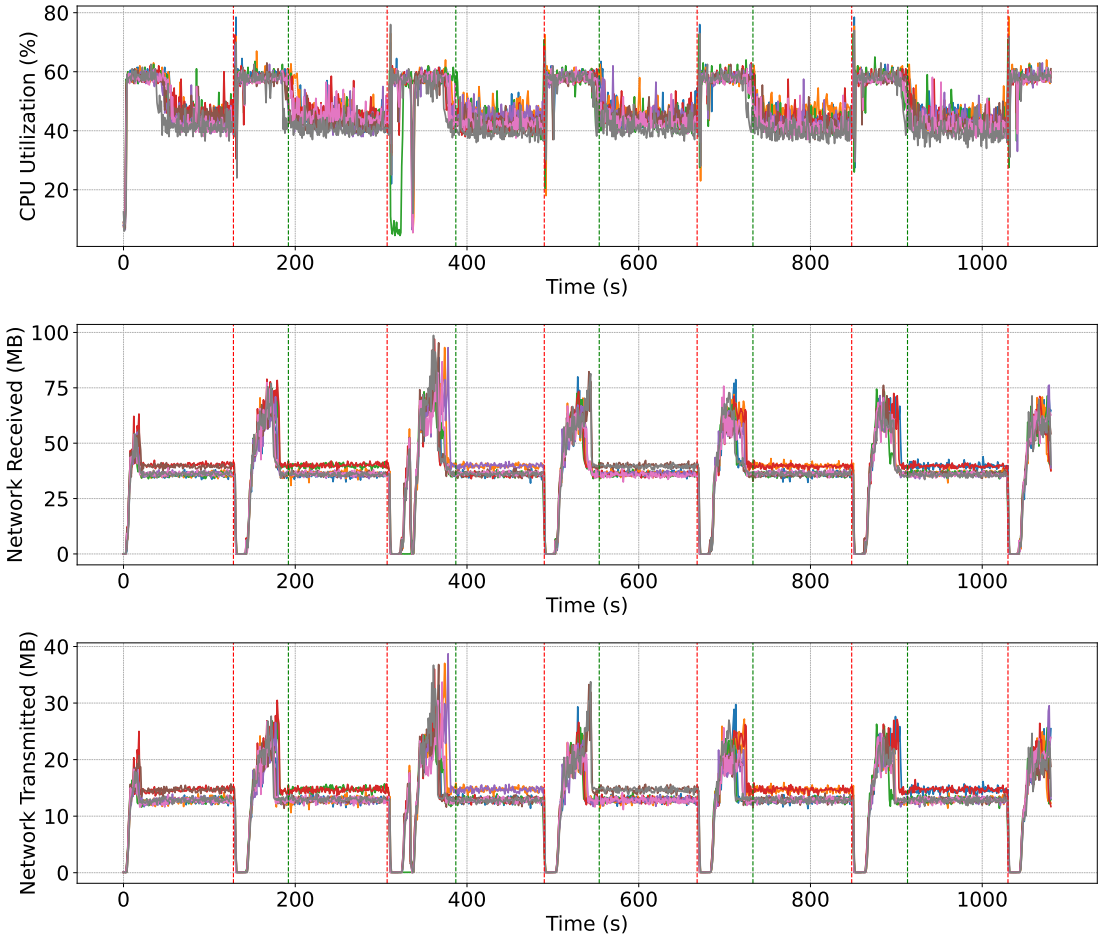

Figure 24. *Resources consumption during balancing and a state recovery in case of 2 worker failures. Red vertical line denote a start of a fault, and the green vertical line is when the system gets back to normal state.*

4.3.2 Analyzing 8-Pod Failures in an 8-Pod Cluster

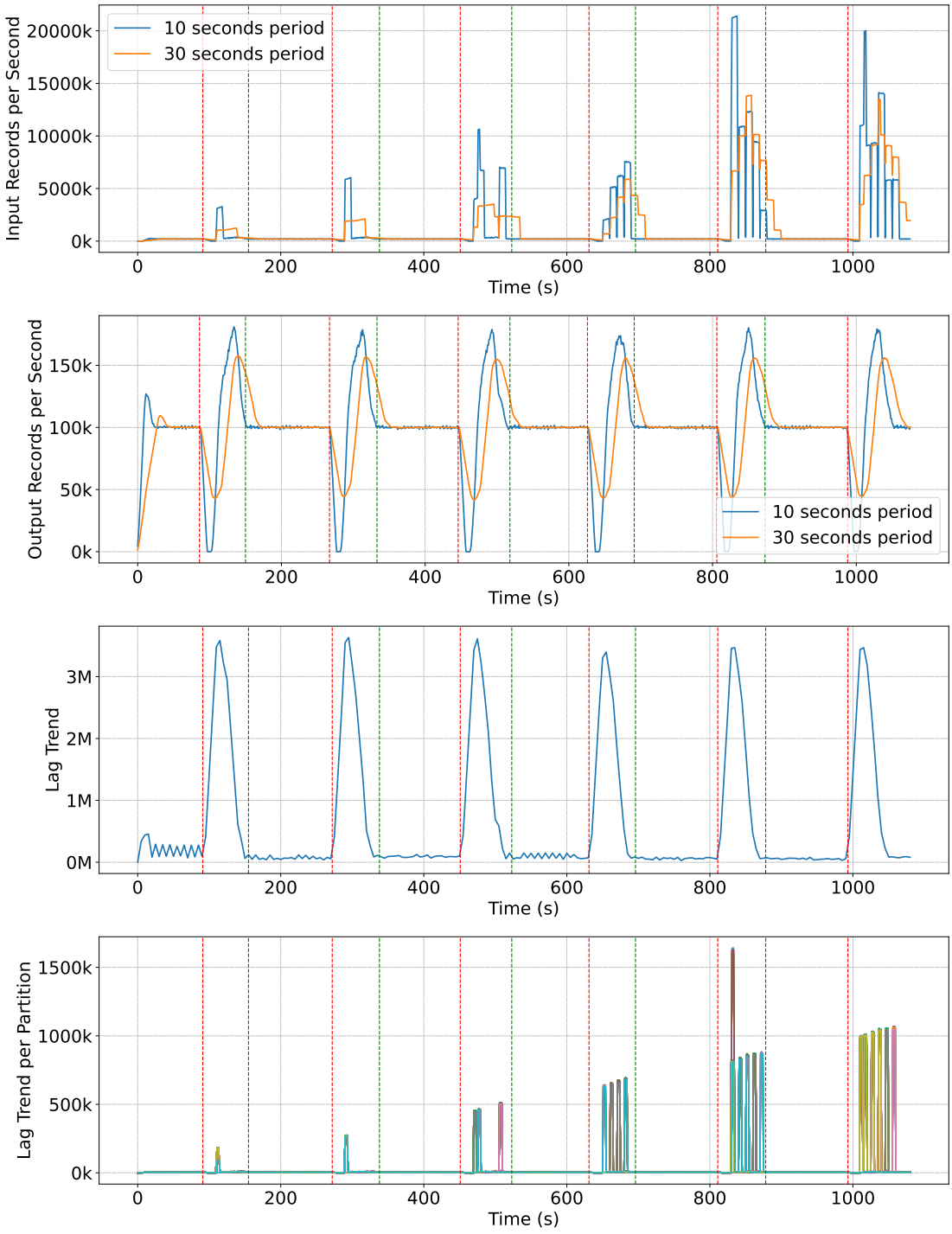

Figure 25. *Benchmarks for Apache Flink experiment in case of 8 workers failure. Worker cluster is fully killed. Red vertical line is a start of the failure green vertical line is a moment when the system is back to a normal state and producing expected load of records.*

These benchmarks on Figure 25 and resources consumption benchmarks on Figure 26 denote Apache Flink performance in case of all workers in worker cluster get killed. The

average time from a fault to a normal execution state is about 114 seconds. That it's quite an impressive result, 1 second difference comparing to benchmarks with 2 workers on Figure 23.

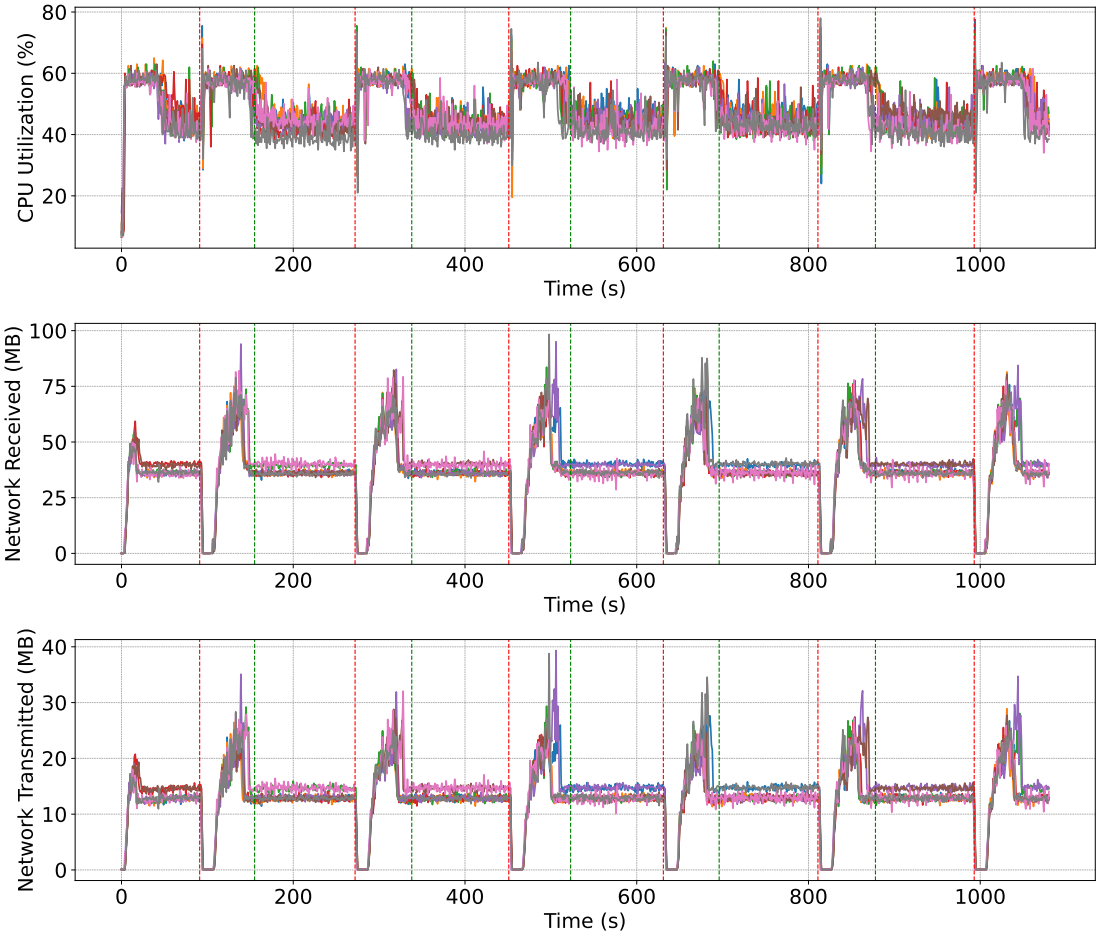

Figure 26. *Resources consumption during balancing and a state recovery in case of 8 worker failures. Red vertical line denote a start of a fault, and the green vertical line is when the system gets back to normal state.*

4.4 Comparative Analysis

This section is comparing Apache Flink and Kafka Streams benchmarks on the same chart to observe a behavior difference in two systems. Important notice, workers failures happened in different moments since for Kafka Streams and Flink, due to Chaos Mesh cron jobs.

4.4.1 Input Throughput

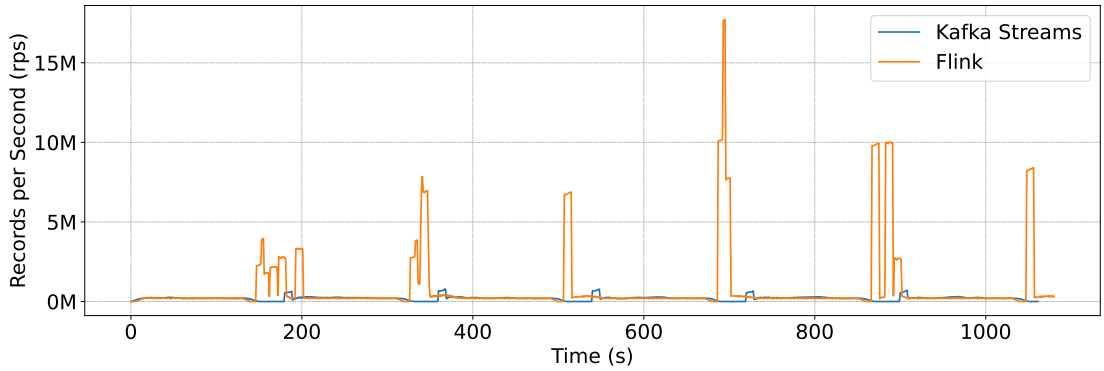

Figure 27. *Input throughput for Kafka records in case of 2 workers. Failure period for kafka Stream and Apache Flink is not synced but happens within the same period.*

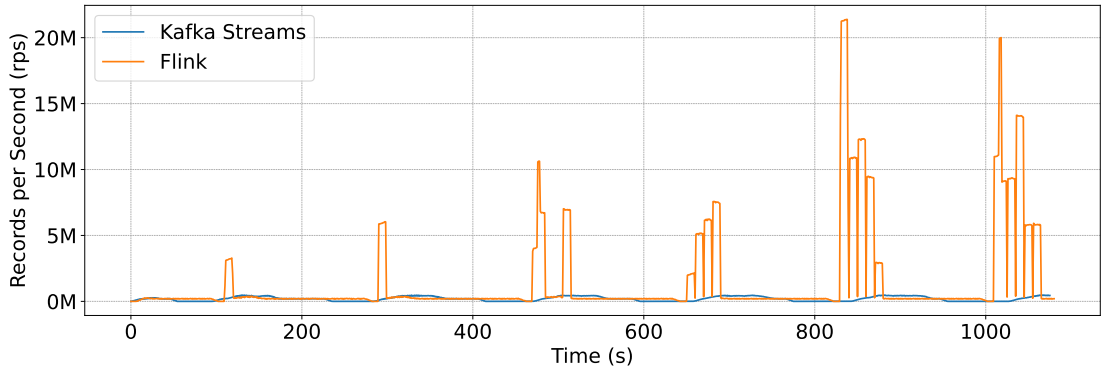

Figure 28. *Input throughput for Kafka records in case of 8 workers. Failure period for kafka Stream and Apache Flink is not synced but happens within the same period.*

In both cases on Figure 27 and Figure 28 Flink is showing extreme high input rate comparing to Kafka Streams. Flink uses its own offset commit mechanism in case of fault tolerance [56]. Also, such behavior is due to faster startup time for Flink workers, CPU metrics on Figures 26 24 for Flink and on Figures 19 21 for Kafka Streams show that Flink workers get started faster.

4.4.2 Output Throughput

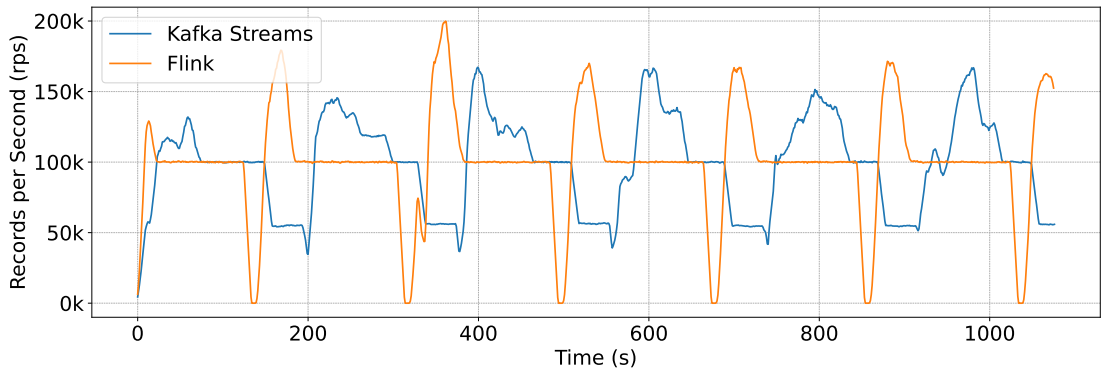

Figure 29. *Output throughput for Kafka records in case of 2 workers failures. Failure period for kafka Stream and Apache Flink is not synced but happens within the same period.*

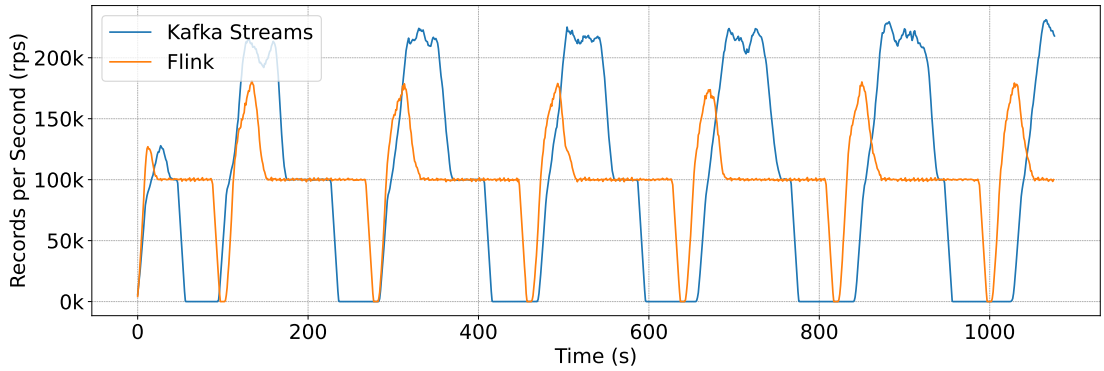

Figure 30. *Output throughput for Kafka records in case of 8 worker failures. Failure period for kafka Stream and Apache Flink is not synced but happens within the same period.*

Interesting behavior can be seen on Figure 29 and 30. On the Figure 29 Flink workers produce higher output rate rather than Apache Flink while on Figure 30 Kafka Streams produce higher rate. This behavioral could be described in the following way. Since CPU metrics show that Kafka Streams workers need more startup time, especially if a whole worker cluster was down for a short period. During a downtime time, the load generator has produced a lot of uncommited records, that Kafka Streams workers start processing later comparing to Flink. To catch up with real time and process delayed records, Kafka Streams has to process more records that lead to a higher output. In the case of 2 workers downtime, Kafka Streams need to wait only for 2 worker. Flink is able to start processing records sooner, for this reason, Flink needs to process less uncommited records.

4.4.3 Lag Trend

Lag trend metric is described in the benchmarks section 4.1.1.

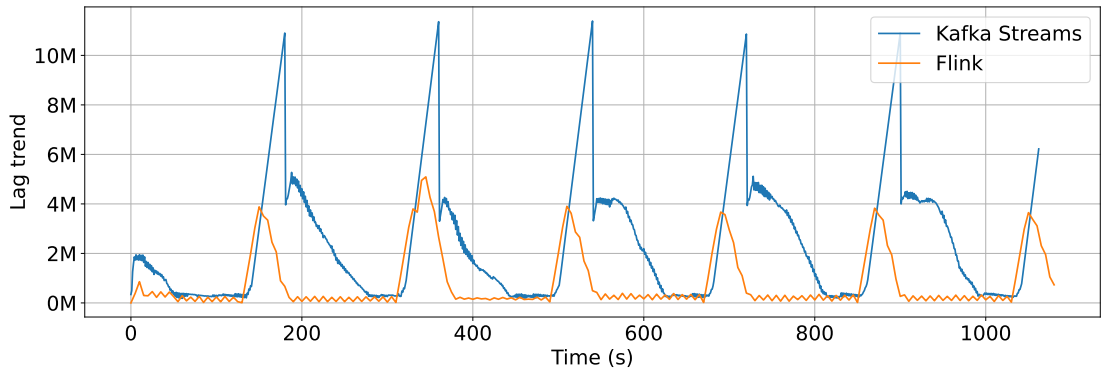

Figure 31. *Lag trend for worker cluster consumer group in case of 2 worker failures. Failure period for kafka Stream and Apache Flink is not synced but happens within the same period.*

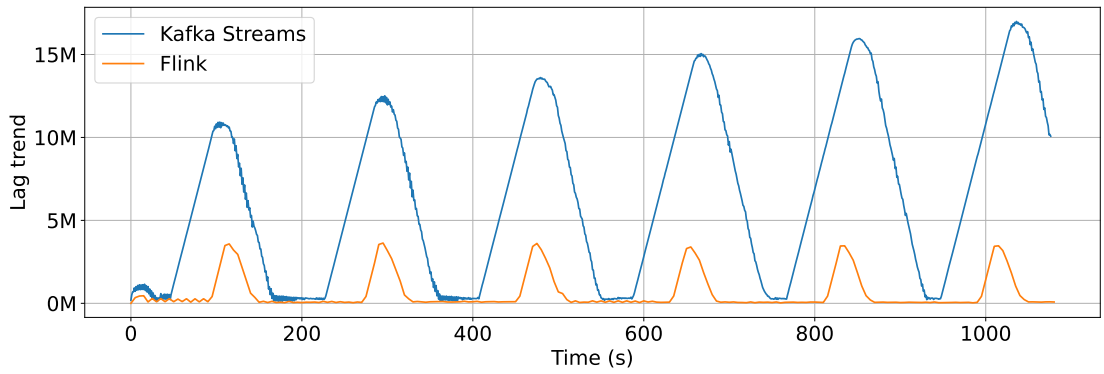

Figure 32. *Lag trend for worker cluster consumer group in case of 8 worker failures. Failure period for kafka Stream and Apache Flink is not synced but happens within the same period.*

As described in 4.4.2, due to faster startup time and different records offset processing, Flink is able to process input records with less delay. For this reason, there is such a lag difference on Figure 31 and Figure 32.

4.4.4 CPU utilization

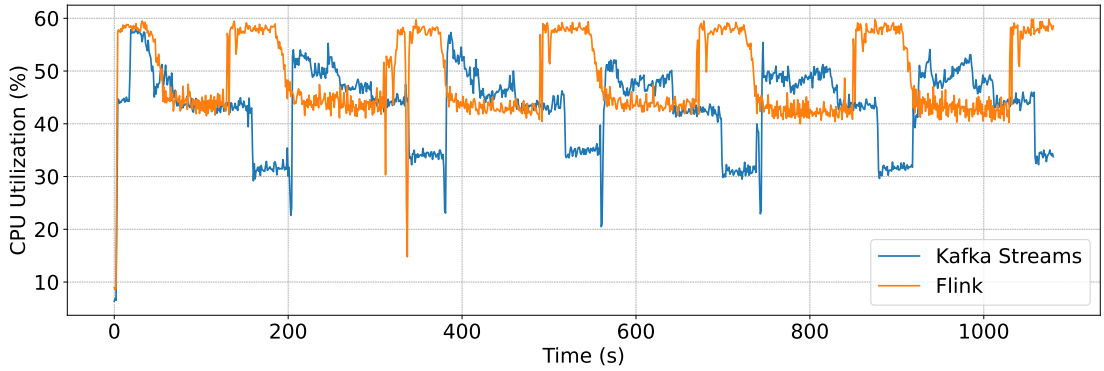

Figure 33. *Average CPU utilization for all workers in case of 2 worker failures. Failure period for kafka Stream and Apache Flink is not synced but happens within the same period.*

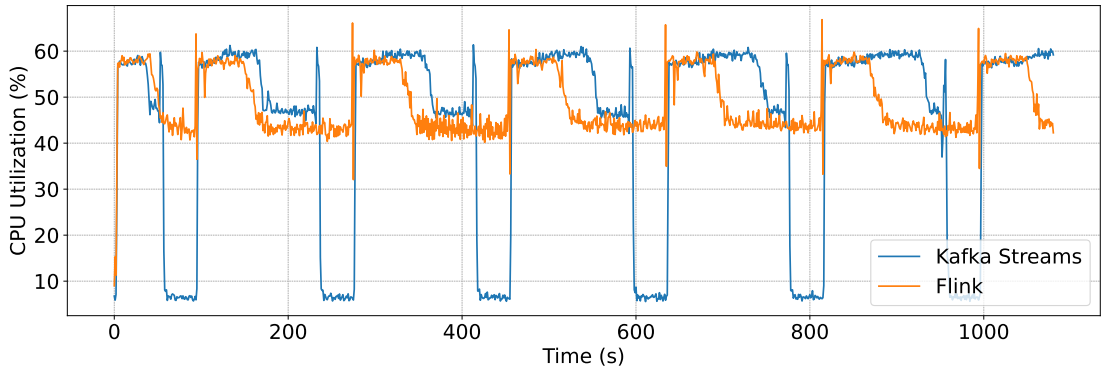

Figure 34. *Average CPU utilization for all workers in case of 8 worker failures. Failure period for kafka Stream and Apache Flink is not synced but happens within the same period.*

CPU metrics on Figure 33 and 34 show faster Flink workers startup for both cases. Kafka Streams workers show higher downtime.

4.4.5 Network Traffic

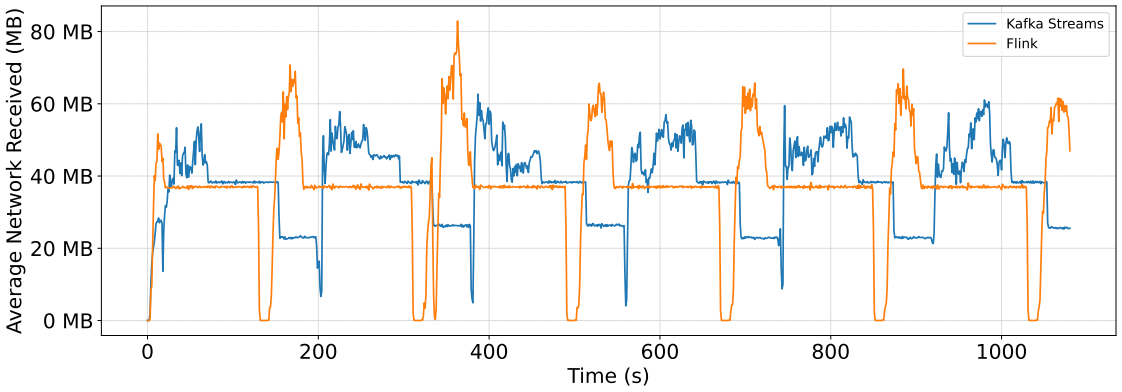

Figure 35. *Average inbound network traffic for all workers in case of 2 worker failures. Failure period for kafka Stream and Apache Flink is not synced but happens within the same period.*

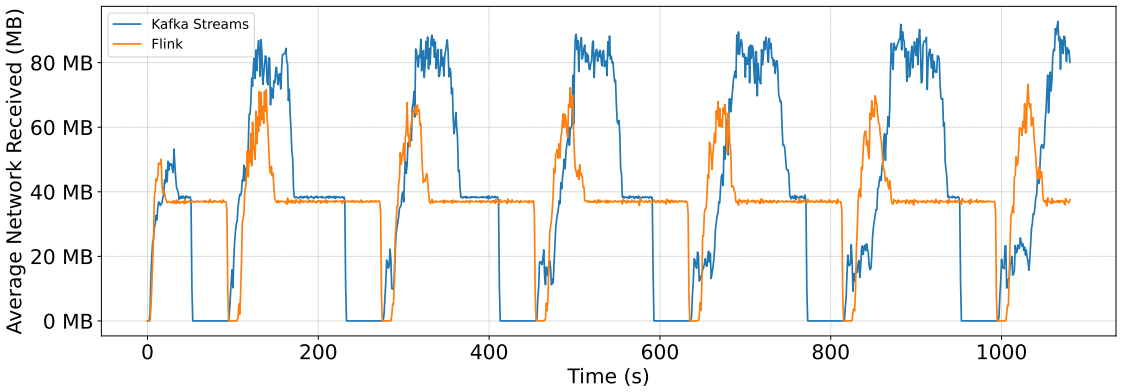

Figure 36. *Average inbound network traffic for all workers in case of 8 worker failures. Failure period for kafka Stream and Apache Flink is not synced but happens within the same period.*

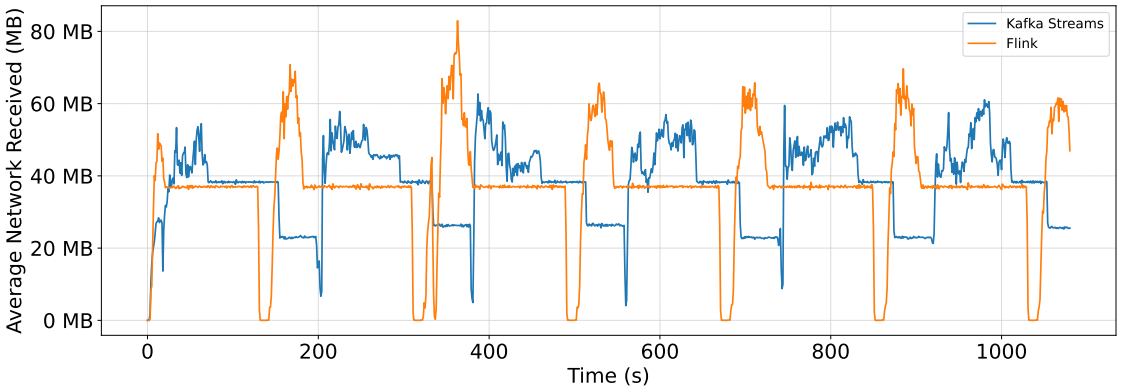

Figure 37. *Average outbound network traffic for all workers in case of 2 worker failures. Failure period for kafka Stream and Apache Flink is not synced but happens within the same period.*

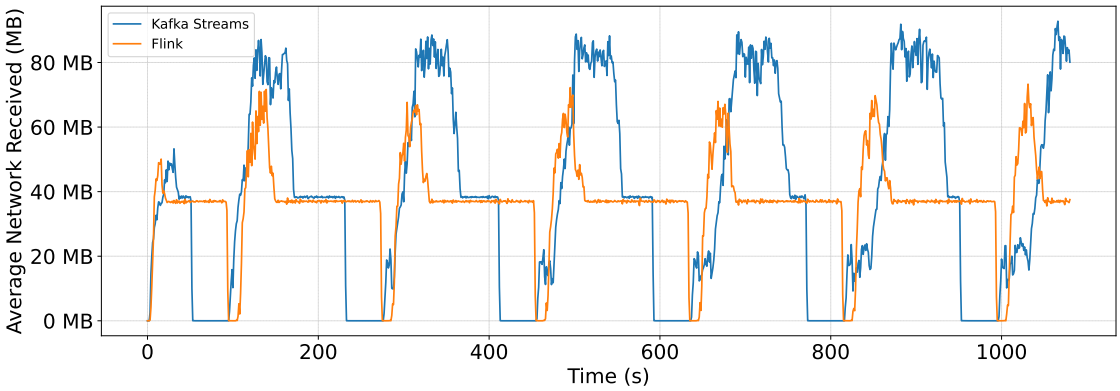

Figure 38. *Average outbound network traffic for all workers in case of 8 worker failures. Failure period for kafka Stream and Apache Flink is not synced but happens within the same period.*

Inbound and outbound network traffic is used for a state recovery and incoming load processing. In case of 8 worker failures Figure 36 and Figure 38 Kafka Streams uses higher network to process faster uncommited records during the downtime. Output record rate on Figure 20 is higher to due to longer downtime for Kafka Streams workers. However, Flink workers show high spikes in case of 2 worker failures, but for a shorter period.

5. Related Work

Stream processing is a domain that focuses on continuously processing and analyzing data streams in real-time or near real-time. Compared to traditional batch processing, where data is collected, stored, and then processed, stream processing deals with unbounded data sets that are continuously generated and need to be processed on the fly [26, 25, 19, 23]. This approach is essential for applications requiring real-time analytics, such as fraud detection, network monitoring, and real-time recommendation systems.

One of the first mentions of the stream processing concept appeared in the early 1990s when databases evolved into data stream management systems (DSMS). Early research focused on developing algorithms and systems that could process data incrementally as it arrived rather than relying on the conventional batch processing model. One of the pioneering projects in this field was the Continuous Query Language (CQL) developed at Stanford University, which allowed users to express continuous queries over data streams. The first Complex Event Processing (CEP) systems appeared in the 2000s. Early CEP systems like Esper and Apache Storm provided the groundwork for modern stream processing frameworks by introducing concepts such as event windows, stateful processing, and real-time analytics. The rise of big data in the late 2000s and early 2010s brought new challenges and opportunities for stream processing. Traditional CEP systems struggled to handle the massive scale and high velocity of big data, leading to the development of distributed stream processing frameworks capable of scaling horizontally across many clusters of nodes. Modern systems from this era include Apache Flink [14], Apache Spark [8], Kafka Streams [1]. The history of stream processing is well described in A Survey on the Evolution of Stream Processing Systems [57].

Choosing the most suitable stream processing system for a specific use case can be challenging. In such cases, benchmarks provide a clearer understanding of system performance under various conditions. The Theodolite framework [35] was developed to evaluate the performance of big data systems, particularly in cloud environments based on Kubernetes. This framework includes scalability benchmarks for Apache Flink and Kafka Streams. One use case involves processing data from thousands of sensors in real-time.

Results show how Apache Flink can be efficient in case of increasing data volumes, such that Flink requires fewer replicas to handle the same amount of data. Another study covers benchmarks for windowing aggregations [58]. Apache Flink, Apache Storm, and

Apache Spark are considered for SQL-based windowing, where Apache Flink shows higher throughput. The recent release of ShuffleBench [28] that is focusing on scalability benchmarks evaluation of Apache Flink, Hazelcast Jet, Apache Spark, and Kafka Streams.

The following studies [59] [60] focus primarily on checkpoints and state recovery use cases. The foundation covered in these studies gives a full overview of Flink's performance in case of a state recovery, especially in case of extensive data streams. Research conducted by colleagues from Tartu University offers additional insights into performance [61], demonstrating that Apache Flink perform having a low latency even under high input throughput compared to other frameworks.

6. Conclusion

This section provides results and future work that should provide deep knowledge about systems performance.

6.1 Summary

Both prototypes based on the Apache Kafka and Apache Flink frameworks demonstrated robust capabilities in state restoration during short-term replica failures. However, several performance insights appeared in the distributed environment while processing the same input data for all experiments. In an experiment involving a cluster of 8 workers, the average recovery time to return to normal processing after 2 workers failed was 143 seconds for Kafka Streams and 113 seconds for Apache Flink. When the entire cluster of 8 workers gets killed, the average recovery time to return to normal processing was 153 seconds for Kafka Streams and 114 seconds for Apache Flink. For both experiments, Apache Flink has shown better performance, for the first is about 30 seconds and for the second is 40 seconds. The main reason behind this behavior is Apache Flink's ability to complete rebalancing faster and start processing of incoming records sooner. CPU metrics from the case study show that the Kafka Streams cluster experiences higher downtime. The lag trend metric in this case study showed better performance for Apache Flink. This means that, despite latency after rebalancing and state recovery, Apache Flink processes incoming records with less latency for all experiments. A significant difference was recorded when the entire cluster gets killed.

6.2 Future work

Such results just depict one of use cases, however, benchmarks could be significantly extended with having set of additional configurations. This research could be extended with additional configurations.

JVM Different Java versions should be able to show a performance difference especially with new Java 21 garbage collector release. Both Apache Flink and Kafka Streams are JVM based frameworks.

State Backend The main state backend for Kafka Streams is RocksDB. However, Apache Flink provides different state backend setups and RocksDB as well.

State Size Different state size in a case of fault tolerance should bring more insights about frameworks for real production systems that use large states.

Another direction of benchmarking involves testing various failure scenarios that are common in production systems, such as slow networking, disk backpressure, and running out of disk space. That would help extend chaos engineering knowledge for the critical real-time systems.

References

[1] *Kafka Streams*. URL: https://kafka.apache.org/documentation/streams/.

[2] *Apache Flink*. URL: https://flink.apache.org/.

[3] *Apache Kafka*. URL: https://kafka.apache.org/.

[4] *Kubernetes*. URL: https://kubernetes.io/.

[5] Marios Fragkoulis, Paris Carbone, and Vasiliki Kalavri Asterios Katsifodimos. "A survey on the evolution of stream processing systems". In: *arXiv preprint arXiv:2103.10169* (2023). URL: https://link.springer.com/article/10.1007/s00778-023-00819-8.

[6] *Apache S4*. URL: https://incubator.apache.org/projects/s4.html.

[7] Jeyhun Karimov et al. "Benchmarking Distributed Stream Data Processing Systems". In: *https://arxiv.org/abs/1802.08496* (2018).

[8] Matei Zaharia et al. "Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark". In: *Proceedings of the 2018 International Conference on Management of Data*. 2018. URL: https://people.eecs.berkeley.edu/~matei/papers/2018/sigmod_structured_streaming.pdf.

[9] Amazon Web Services. *Amazon EC2 Instance Types*. Accessed: 2024-05-05. 2024. URL: https://aws.amazon.com/ec2/instance-types/.

[10] Xiangqun Meng and Zhiying Wang. "Cloud Computing Research and Development Trend". In: *2014 Second International Conference on Future Generation Communication Technology (FGCT)*. IEEE, 2014, pp. 89–92. DOI: 10.1109/FGCT.2014.7066349. URL: https://ieeexplore.ieee.org/document/6847479.

[11] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, pp. 137–150.

[12] Ralf Lämmel. "Google's MapReduce programming model — Revisited". In: *Science of Computer Programming* 70.1 (2008), pp. 1–30. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2007.07.001. URL: https://www.sciencedirect.com/science/article/pii/S0167642307001281.

[13] Rabi Prasad Padhy and Rabi Prasad Padhy. “Big Data Processing with Hadoop-MapReduce in Cloud Systems”. In: *International Journal of Cloud Computing and Services Science (IJ-CLOSER)* 2.1 (Nov. 2012). DOI: 10.11591/closer.v2i1.1508. URL: https://journal.iaescore.com/index.php/IJ-CLOSER/article/view/1508.

[14] Paris Carbone et al. “Apache Flink: Stream and Batch Processing in a Single Engine”. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015), pp. 28–38. URL: https://research.tudelft.nl/en/publications/apache-flink-stream-and-batch-processing-in-a-single-engine.

[15] Confluent. *Kafka Streams Architecture*. 2024. URL: https://docs.confluent.io/platform/current/streams/architecture.html.

[16] Apache Flink. *Powered by Apache Flink*. Accessed: Your Access Date. 2023. URL: https://flink.apache.org/what-is-flink/powered-by/.

[17] Netflix Technology Blog. *Streaming SQL in Data Mesh*. Accessed: Your Access Date. Apr. 2023. URL: https://netflixtechblog.com/streaming-sql-in-data-mesh-0d83f5a00d08.

[18] Confluent. *Stateful Fault Tolerance*. Accessed: Your Access Date. 2023. URL: https://developer.confluent.io/courses/kafka-streams/stateful-fault-tolerance/#stateful-fault-tolerance.

[19] Matei Zaharia et al. “Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing”. In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. Accessed: Your Access Date. San Jose, CA: USENIX Association, Apr. 2012, p. 2. DOI: OptionalDOIifavailable. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[20] Paris Carbone et al. “Lightweight Asynchronous Snapshots for Distributed Dataflows”. In: 2015. URL: https://arxiv.org/abs/1506.08603.

[21] *Flink Kubernetes Operator*. Accessed on: 01/02/2024. URL: https://nightlies.apache.org/flink/flink-kubernetes-operator-docs-stable/.

[22] Xiangrui Meng et al. “MLlib: Machine Learning in Apache Spark”. In: *Journal of Machine Learning Research* 17.34 (2016), pp. 1–7. URL: http://jmlr.org/papers/v17/15-237.html.

[23] Asterios Katsifodimos and Sebastian Schelter. “Apache Flink: Stream Analytics at Scale”. In: *2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW)*. 2016, pp. 193–193. DOI: 10.1109/IC2EW.2016.56.

[24] *Flink ML*. Accessed on: 01/02/2024. URL: https://nightlies.apache.org/flink/flink-ml-docs-release-2.3/docs/try-flink-ml/java/quick-start/.

[25] Tyler Akidau et al. "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing". In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1792–1803. DOI: 10.14778/2824032.2824076.

[26] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 2017. ISBN: 978-1449373320.

[27] Gwen Shapira, Neha Narkhede, and Todd Palino. *Kafka: The Definitive Guide*. 2nd ed. O'Reilly Media, 2020. ISBN: 978-1492043072.

[28] Sören Henning et al. "ShuffleBench: A Benchmark for Large-Scale Data Shuffling Operations with Distributed Stream Processing Frameworks". In: *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*. ICPE '24. ACM, May 2024. DOI: 10.1145/3629526.3645036. URL: http://dx.doi.org/10.1145/3629526.3645036.

[29] Confluent. *Enabling Exactly-Once in Kafka Streams*. Accessed: Your Access Date. 2023. URL: https://www.confluent.io/blog/enabling-exactly-once-kafka-streams/.

[30] Apache Flink. *An Overview of End-to-End Exactly-Once Processing in Apache Flink, with Apache Kafka Too*. Accessed: Your Access Date. Feb. 2018. URL: https://flink.apache.org/2018/02/28/an-overview-of-end-to-end-exactly-once-processing-in-apache-flink-with-apache-kafka-too/.

[31] Confluent. *Control Plane*. Accessed: Your Access Date. 2023. URL: https://developer.confluent.io/courses/architecture/control-plane/.

[32] K. Mani Chandy and Leslie Lamport. "Distributed Snapshots: Determining Global States of a Distributed System". In: *ACM Transactions on Computer Systems* (Feb. 1985), pp. 63–75. URL: https://www.microsoft.com/en-us/research/publication/distributed-snapshots-determining-global-states-distributed-system/.

[33] Intel. *Introduction to Hyperscan*. Accessed: Your Access Date. 2023. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-hyperscan.html.

[34] Amazon Web Services. *Amazon EKS: Managed Kubernetes Service*. Accessed: 2024-05-05. 2024. URL: https://aws.amazon.com/eks/.

[35] Sören Henning and Wilhelm Hasselbring. "Theodolite: Scalability Benchmarking of Distributed Stream Processing Engines in Microservice Architectures". In: *ResearchGate* 208 (2021). DOI: 10.1016/j.bdr.2021.100209.

[36] Grafana Labs. *Grafana: The open observability platform*. Accessed: 2024-05-05. 2024. URL: https://grafana.com/grafana/.

[37] Prometheus Operator. *Prometheus Operator: Easy monitoring definitions for Kubernetes services*. Accessed: 2024-05-05. 2024. URL: https://prometheus-operator.dev/.

[38] Prometheus. *Prometheus: Open-Source Systems Monitoring and Alerting Toolkit*. Accessed: 2024-05-05. 2024. URL: https://prometheus.io/.

[39] Amazon Web Services. *Amazon EC2: Secure and resizable compute capacity in the cloud*. Accessed: 2024-05-05. 2024. URL: https://aws.amazon.com/ec2/.

[40] Kelsey Hightower, Brendan Burns, and Joe Beda. *Kubernetes: Up and Running*. O'Reilly Media, Inc., 2019. ISBN: 9781492046530.

[41] Julien Pivotto and Brian Brazil. *Prometheus: Up & Running*. O'Reilly Media, Inc., 2023. ISBN: 9781098131142.

[42] Amazon Web Services. *Amazon Elastic Block Store (EBS)*. 2024. URL: https://aws.amazon.com/ebs/.

[43] Amazon Web Services. *Amazon Elastic File System (EFS)*. 2024. URL: https://aws.amazon.com/efs/.

[44] Sören Henning and Wilhelm Hasselbring. "A configurable method for benchmarking scalability of cloud-native applications". In: *Empirical Software Engineering* 27 (2022). DOI: 10.1007/s10664-022-10162-1.

[45] Helm Project. *Helm: The Package Manager for Kubernetes*. Accessed: Your Access Date. 2024. URL: https://helm.sh/.

[46] Chaos Mesh Project. *Chaos Mesh: A Powerful Chaos Engineering Platform for Kubernetes*. 2024. URL: https://chaos-mesh.org/.

[47] Prometheus Project. *Querying Basics*. 2024. URL: https://prometheus.io/docs/prometheus/latest/querying/basics/.

[48] Theodolite Project. *Creating an Execution*. 2024. URL: https://www.theodolite.rocks/creating-an-execution.html.

[49] Apache Flink Project. *Flink Architecture*. 2024. URL: https://nightlies.apache.org/flink/flink-docs-master/docs/concepts/flink-architecture/.

[50] Amazon Web Services. *Persistent Storage for Kubernetes*. 2024. URL: https://aws.amazon.com/blogs/storage/persistent-storage-for-kubernetes/.

[51] Kubernetes. *Resource Metrics Pipeline*. 2024. URL: https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-metrics-pipeline/.

[52] Micrometer Project. *Micrometer: Application Monitoring for JVM-based Systems*. 2024. URL: https://micrometer.io/.

[53] Kubernetes. *ConfigMaps*. 2024. URL: https://kubernetes.io/docs/concepts/configuration/configmap/.

[54] Amazon Web Services. *Amazon Elastic Container Registry (ECR)*. 2024. URL: https://aws.amazon.com/ecr/.

[55] Amazon Web Services. *Amazon EFS CSI Driver*. 2024. URL: https://docs.aws.amazon.com/eks/latest/userguide/efs-csi.html.

[56] *Flink Kafka Offset*. 2024. URL: https://nightlies.apache.org/flink/flink-docs-release-1.19/docs/connectors/datastream/kafka/#consumer-offset-committing.

[57] Marios Fragkoulis et al. *A Survey on the Evolution of Stream Processing Systems*. 2023. arXiv: 2008.00842 [cs.DC].

[58] Jeyhun Karimov et al. "Benchmarking Distributed Stream Data Processing Systems". In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 2018, pp. 1507–1518. DOI: 10.1109/ICDE.2018.00169.

[59] Paris Carbone et al. *Lightweight Asynchronous Snapshots for Distributed Dataflows*. 2015. arXiv: 1506.08603 [cs.DC].

[60] George Siachamis et al. *CheckMate: Evaluating Checkpointing Protocols for Streaming Dataflows*. 2024. arXiv: 2403.13629 [cs.DC].

[61] Elkhan Shahverdi, Ahmed Awad, and Sherif Sakr. "Big Stream Processing Systems: An Experimental Evaluation". In: *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*. 2019, pp. 53–60. DOI: 10.1109/ICDEW.2019.00-35.

Appendix 1 – Non-Exclusive License for Reproduction and Publication of a Graduation Thesis[1]

I Aleksandr Madisson

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "Stateful Stream Processing: A Comparative Analysis of Apache Flink and Kafka Streams frameworks", supervised by Radu Irbe
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

17.05.2024

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.