Insights into Distributed System Failures: Locating Anomalies in Log Data

by

Anis Ben Saada

Matriculation Number XXXXXX

A thesis submitted to

Technische Universität Berlin
Fakultät IV - Electrical Engineering and Computer Science
Department of Telecommunication Systems
Distributed and Operating Systems

Bachelor's Thesis

November 14, 2024

Supervised by:
Prof. Dr. Odej Kao
Prof. Dr. Volker Markl

Abstract

This thesis addresses the challenges of locating anomalies in application logs within distributed systems, focusing on Apache Kafka and Apache Flink. Distributed systems, due to their complex, interconnected nature, demand robust methods for detecting faults across components and their dependencies to ensure reliable performance. This research employs a fault injection approach to simulate real-world failures, systematically targeting vital components in the implemented Environment. These controlled experiments generate detailed log data that serves as the basis for anomaly analysis. The collected logs undergo a structured process of generalization and differentiation to standardize message patterns and highlight deviations specific to fault scenarios. Following this, an anomaly determination framework based on a sequential decision tree evaluates relevant log attributes. This approach enables the flagging of logs into potential anomalies without requiring labeled data. The results reveal significant interdependencies between components, with the Flink TaskManager displaying heightened sensitivity, frequently producing collateral anomalies due to faults in related components. This finding underscores the complexity of root cause analysis in distributed systems, where failures in one area can cascade and obscure the initial fault's origin. This study analyses the systems' behavior and inter-component connections. Such advancements are essentiel because they lay a foundation for future anomaly detection analysis work. They also help create resilient systems capable of enduring faults and accurately diagnosing them in real-world interconnected environments.

The implementation can be found on GitHub [1].

[1] https://github.com/anisbs96/ba-implementation

Zusammenfassung

Diese Thesis befasst sich mit den Herausforderungen, Anomalien in Anwendungsprotokollen innerhalb verteilter Systeme zu lokalisieren. Der Schwerpunkt liegt dabei auf Apache Kafka und Apache Flink. Aufgrund ihres komplexen und vernetzten Charakters erfordern verteilte Systeme robuste Methoden zur Erkennung von Fehlern in deren Komponenten und Abhängigkeiten, damit sie eine zuverlässige Leistung gewährleisten können. Diese Forschung verwendet einen Fehlerinjektionsansatz, um reale Ausfälle zu simulieren und dabei systematisch auf wichtige Komponenten in der implementierten Umgebung abzuzielen. Diese kontrollierten Experimente erzeugen detaillierte Protokolldaten, welche als Grundlage für die Analyse von Anomalien dienen. Die gesammelten Protokolle werden einem Prozess der Generalisierung und Differenzierung unterzogen, um Meldungsmuster zur standardisieren und störungsspezifische Abweichungen hervorzuheben. Anschließend werden durch ein auf einem sequentiellen Entscheidungsbaum basierendes Anomaliebestimmungs-Framework relevante Protokollattribute ausgewertet. Durch diesen Ansatz wird die Kennzeichnung von Protokollen als potenzielle Anomalien ermöglicht, ohne dass gekennzeichnete Daten erforderlich sind. Die Ergebnisse zeigen signifikante Abhängigkeiten zwischen den Komponenten, wobei der Flink TaskManager eine erhöhte Sensitivität erkennen lässt und häufig kollaterale Anomalien produziert, aufgrund von Fehlern in verwandten Komponenten. Diese Erkenntnis unterstreicht die Komplexität der Anomalieerkennungsanalyse in verteilten Systemen, in denen sich Fehler in einem Bereich kaskadenartig ausbreiten und die Ursache des ursprünglichen Fehlers verdecken können. Diese Studie analysiert das Verhalten des Systems und die Verbindung zwischen den Komponenten. Solche Fortschritte sind von entscheidender Bedeutung, da sie die Grundlage für das künftige Erkennen von Anomalien und die Ursachenanalyse bilden. Außerdem tragen sie dazu bei, widerstandsfähige Systeme zu schaffen, die Störungen in realen, vernetzten Umgebungen standhalten und genau diagnostizieren können.

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Problem Description 1
1.3 Outline 2

2 Background **3**
2.1 Distributed Systems 3
2.2 Apache Kafka 4
2.3 Apache Flink 5
2.4 Containerization 6
2.5 Google Cloud Platform 7
2.6 Log Templating 8

3 Related Work **11**
3.1 System Failures 11
3.2 Anomaly Detection 11

4 Method **13**
4.1 System Overview 13
4.2 Components 14
4.3 Fault Injection 15
4.4 Log Generalization and Difference 18
4.5 Anomaly Determination 20

5 Evaluation **29**
5.1 Generated Log Data 29
5.2 Fault Injection 31
5.3 Results 34

6 Conclusion **41**
6.1 Future Work 41

List of Tables **43**

List of Figures **45**

Bibliography **47**

1 Introduction

1.1 Motivation

In the world of distributed systems, like Apache Kafka and Apache Flink, lies the ability to manage real-time data streams at a scale. These systems provide applications effective data processing while meeting tight performance demands [1] [2]. As these systems grow in sophistication and scope, ensuring reliability and fault tolerance is difficult. Within settings, spotting irregularities that may hint at system breakdowns or declining performance is essential. Logs, from applications, play a role in comprehending how distributed systems operate since they document system activities and communications in detail to spot any unusual occurrences effectively. Unlike system logs, application logs are more disorganized. Therefore, examining them poses an obstacle [3]. Their lack of organization creates variety and complexity rendering it more difficult to recognize issues and spot irregularities. Most studies, on identifying anomalies have mainly focused on using machine learning methods with system logs as the area of interest [4]. These techniques often need labeled datasets. However application logs haven't received research scrutiny despite being crucial in understanding how different parts of distributed systems interact. The scarcity of data and the diverse nature of application logs make traditional machine learning approaches less effective, for analysis and detection tasks.

This study identify irregularities in application logs designed to determine potential anomalies to address these challenges. This framework enhances existing techniques by providing a method to identify logs that may need investigation. It operates by analyzing logs generated during fault injection experiments considering aspects such as log severity, frequency of occurrence, associated faults and specific terminology used. The unusual findings can be inputted into machine learning models to improve the effectiveness of detecting anomalies in distributed systems.

1.2 Problem Description

Working with machine learning methods to detect irregularities using application logs can pose a challenge due to a limitation; the requirement for labeled datasets poses difficulties as they are not always easy to establish for logs [5]. Furthermore the deep learning models employed in anomaly detection might operate in a different manner when it comes to their decision making process. This concern becomes evident in situations where precise anomaly identification's crucial, for troubleshooting issues [6]. One more drawback of these methods is their focus is solely placed on system logs, which are typically more organized and simpler to examine compared to application logs, which tend to be quite diverse in nature, with a notable amount of irrelevant information and substantial variations seen across various components and systems [7]. These variations contribute to the complexity involved in anomaly detection tasks, resulting in a research gap concerning techniques for evaluating application logs. Log-based methods can be used to detect anomalies effectively to find irregularities and issues that could lead to a decrease in system performance. This system is not designed to substitute machine learning models; instead, it enhances ongoing research efforts by identifying possible anomalies that could be valuable for training and improving machine learning models. It reviews logs gathered from fault injection tests, where purposeful failures are induced to study how the system responds. The system assists in recognizing logs that may suggest anomalous behavior by emphasizing characteristics like log severity level and frequency, as well as fault specificity.

This method offers an organized approach to spotting irregularities in application logs by concentrating on examining logs after failure events occur, rather than in real-time detection, the system offers insight into system performance and behavior. It bridges the divide between manual log evaluation and automated anomaly detection systems.

1.3 Outline

- **Chapter 2** covers distributed systems, like Apache Kafka and Apache Flink. It also outlines the architecture and value of platforms of them. This spotlights the important role of log analysis in assuring system reliability.
- **Chapter 3** reviews related work on anomaly detection and system failures then highlights the limitations of current machine learning-based approaches in handling application logs.
- **Chapter 4** describes the methodology, and detailes the system architecture, the fault injection experiments, and the process used to flag potential anomalies in application logs. The implementation can be found on Github [1].
- **Chapter 5** presents the evaluation of the system, presents results from the fault injection experiments, and analyzes the flagged anomalies.
- **Chapter 6** summarizes the discoveries, suggests future steps such as integrating machine learning models and expanding the system for better anomaly detection capabilities.

[1] https://github.com/anisbs96/ba-implementation

2 Background

2.1 Distributed Systems

A distributed system is an assembly of several loosely coupled elements that carry out "collective" tasks under a common goal; distributed across several nodes (which could be present in physically different environments or clouds), this architecture enables enhanced scalability and fault tolerance, two important requirements for online data processing for real-time applications [1] [2].

In distributed systems each node has its role to play. The components interact using message passing. The communication happens independently leading to challenges, in coordination, such as data consistency, network delays and system errors. These aspects are crucial for system functioning. The reliability of the code hinges on how these challenges can be managed efficiently.

One significant advantage is the effective management of faults. Distributing data and tasks across the system ensures the functionality without disruption when specific components fail. Platforms like Apache Kafka and Apache Flink, can tolerate faults by copying data and creating checkpoints. These strategies allow the system to recover lost data or processes from duplicates or saved points. This reduces the impact of failures on the system's performance [1] [2].

Logs are also a key component for monitoring the consistency and health of a distributed system. They provide detailed factoring of events from a system (e.g., system calls, errors, warnings) as well as user-system interactions. Logs are essentiel for diagnosing root causes of system failures and developing a “gut instinct” for underlying behavioral patterns that may foreshadow future failures. Logs are also a primary source of “abnormal” data, from which anomalies in the form of behavioral outliers can be detected. Logs often include timestamps and other properties important for evaluating data collected at different times [8].

Even though distributed systems come with benefits, they face challenges. They have to manage components while keeping things stable and reliable across the system. If one component malfunctions it could affect the system and lead to disturbances. Therefore quickly identifying irregularities and faults is crucial to minimize system downtime and ensure operations [9].

Logs can also be used to spot occurrences. By gathering logs from various parts along with a simple anomaly model, irregular behavior, like extended caching delays, can be identified. Detective failures are important in real-time to prevent any slowdown or data loss during processing.

2.2 Apache Kafka

Apache Kafka is a distributed messaging system utilized for real-time data streaming, within distributed systems. The structure revolves around three main elements. Producers send data to Kafka brokers for storage and management before consumers pick up the data for processing tasks. This design enables systems to separate creators and users so that they can function autonomously. This is crucial, in distributed settings where elements need to manage data streams [1].

Data is often written to Kafka in high volumes and low latency, and low latency ensures that data is not dropped. Also, the partitions are broken up into topics. Each of those topics are broken up into partitions. The partitions are essentially what different machines get to process. When data is distributed across these partitions, more builds can be done in parallel, allowing for faster system performance and being able to deliver across multiple machines. This concept is what allows Kafka to scale beyond what Amazon Aurora has suited it for and allows it to handle any volume of data requested over real-time with no problem [1].

A fundamental property of Kafka's design is fault tolerance. Each partition of a topic is replicated on any number of brokers to guarantee durability of data and to prevent data loss in case of system failures. This leads to the leader-follower model in which we have one broker denoted as a leader for each partition with other brokers being followers that only replicate this data. When the leader fails, one of the followers advances to the status of the new leader. This way, an additional copy of the data will be up and running without any manual intervention and therefore, data loss is minimized [10].

A key part of how Kafka accomplishes reliable message delivery and data persistence is its log-based architecture. Every single message that gets written to a topic of topics in a Kafka cluster goes to that log: think of it as being stored in a journal for the designated retention period of the topic they were sent to. That's why consumers can read and re-read messages, even after they've already been consumed, and make fault recovery really easy as Kafka can simply replay messages for recovery without losing any context. Of course, this becomes particularly important in systems where, for the most part, the integrity of the data while recovering from a failure are essential [1].

The records produced by Kafka does not function to store data They also offer useful insights, for monitoring and diagnosing system performance issues. Kafka logs document system activities like sending and receiving messages and identifying errors that occur during data transfers. These records are vital for pinpointing problems such as delays, in message processing or lost and undelivered messages. Studying Kafka logs enables system administrators to spot irregularities and address issues effectively to enhance the reliability of the system [1].

In the scope of distributed systems, landscape comparisons between Kafka and other messaging platforms such as RedPanda are practices. RedPanda is recognized for its performance enhancement strategies that aim to reduce resource consumption while offering functionality features as Kafka does, but with a twist; it minimizes reliance on external components like Zookeeper for a more streamlined architecture and better performance, especially in high throughput scenarios. Nonetheless, despite RedPanda's allure and advantages, it is worth noting that Kafka's extensive usage across industries with its established ecosystem and its remarkable fault tolerance capabilities make it the preferred choice for live data flow needs in distributed systems seeking scalable messaging solutions [10].

In architectures of distributed systems Kafka plays a role due to its efficient handling of real-time data streams along with its strong fault tolerance and scalability. Using logs for message retention and system diagnostics also boosts Kafkas usefulness allowing for monitoring system health and recovering from failures without losing data. These features make Kafka especially valuable, in environments that prioritize reliability and seamless data processing [1].

2.3 Apache Flink

Apache Flink is a system that processes data streams across sources in immediate scenarios on a large-scale platform. It stands out due to its ability to manage both streaming and batch data effectively, making it an asset for distributed systems needing real-time analytics and batch-processing functionalities. Apache Flink finds applications, in environments where data flows persistently and quick decisions or analyses need to be made as the data flows in [2].

At the core of Flink's streaming capability is its ability to handle unbounded data streams that keep sliding at a constant pace overtime, with elements being generated never ending. However, it's the right abstraction for distributed systems, processing data streamed by multiple producers and processed in real-time. On top of its streaming capabilities, Flink also has support for bounded data processing in batch homogeneous, serialised and loading data, allowing for a unified approach to real-time and offline data processing [11].

One more characteristic of Flink is the ability to do event-time processing, where records are processed based on the time they occurred, regardless of if records are arriving out of order due to network delay or some other reason. This is important in distributed systems for ensuring the correctness of the results when they depend on the order or time that things happened. Flink handles outside lateness with watermarks based on timestamps to help discover where the watermark has reached in the stream of events, and the ability of the system to process events based on the time they occurred, even though in the distributed system [2]. Real-time applications that perform distributed monitoring in environments where the timing in the data matters are great use cases for Flink's ability to do event-time processing.

Flink is built with fault tolerance as an aspect, in mind. To ensure this reliability factor the system uses state checkpointing to save the processing jobs status to storage. In case of a breakdown Flink can recover its status from the successful checkpoint enabling the system to continue processing without losing any data. This feature of fault tolerance is particularly vital in distributed settings, where issues, like network disruptions, node failures or hardware faults can unexpectedly arise [11]. Flink's capability to bounce back, from setbacks guarantees its ability to uphold availability and dependability, in implementations.

Flink's logs created while running offer details about how the system behaves and performs. These logs contain data about task execution progress and errors occurring during checkpoint processes. They are crucial for troubleshooting failures. Grasping the efficiency of the system. In complex distributed setups, with processing nodes involved, these logs allow administrators to track individual task advancements and spot any obstacles or challenges that might come up during operation [11]. Flink logs prove handy when pinpoint performance problems or irregularities, in data pipelines arise.

In anomaly detection scenarios, within Flink's framework, Flink's logs play a role. Given Flink's function of handling real-time data, it is crucial to identify and rectify any delays, mistakes, or deviations from the usual operation to uphold system efficiency. Detecting anomalies involves investigating the logs for deviations from the patterns, such, as unusual processing durations unsuccessful checkpoints or unanticipated task malfunctions [2]. Timely identification of irregularities aids in resolving issues and reducing their effect on the system's overall performance.

Flink stands out from stream processing engines due, to its capability for processing a feature that sets it apart in distributed systems where handling state across nodes can pose challenges during failures or task redistribution between nodes while maintaining consistency is paramount [11]. Flink tackles this obstacle by locally managing the state within each processing node and regularly backing it up to durable storage as a fail-safe measure for quick recovery in case of system breakdowns, without compromising key data integrity. In scenarios involving event analysis demands, like event processing applications, need to uphold the system's status throughout various data flow phases.

Flink's versatility is evident. It can manage real-time streaming and batch processing within a framework. This dual functionality enables companies to use Flink for an array of tasks from real-time analytics and notifications to batch operations handling extensive fixed datasets. By offering a solution for streaming and batch data processing Flink streamlines the data processing setup. Minimizes the necessity, for distinct tools tailored to different workload types [2].

To sum up the points of Apache Flink's capabilities, it provides a solution for processing real time data in distributed systems effectively and reliably. With features like event time processing and fault tolerance via state checkpointing and stateful processing, Apache Flink is well equipped to manage data processing operations. Additionally, the logs produced by Flink offer information on system performance serving as a resource for monitoring, troubleshooting problems and identifying irregularities. Flinks flexibility, in accommodating both time and batch processing expand its usefulness across scenarios, in contemporary distributed settings.

2.4 Containerization

Containerization is now popular for deploying and managing systems. It packages applications and their parts into isolated containers. These compartments integrate with the operating system kernel of the host system they are on. This setup saves on resources compared to virtual machines. It also keeps different applications nicely separated from one another [12].

Conflicts arising from software dependencies can be avoided by placing each application within its container. This separation enables distributed systems to expand effectively and to manage growing workloads without alterations to the existing infrastructure [13]. Moreover, container coordination tools like Kubernetes oversee containerized applications' deployment, scaling, and functioning, ensuring system availability remains intact in cases of container failures.

Containerization also enhances the robustness of distributed systems by providing fault tolerance advantages; if a containerized application or component fails the container can be replaced without impacting components negatively. This isolation minimizes the impact of failures. This also Strengthens the systems resilience. In distributed environments swiftly swapping or scaling containers is crucial, for maintaining performance and availability [12].

However, with containerization, this poses a certain difficulty: it's harder to manage stateful applications in containers since any state information they might contain has to outlive either a container restart or a container migration. It can be tough to guarantee consistent performance across containers, especially at a broader distributed scale. Ensuring consistent performance in containers at timescales relevant to the end user is a major active direction in research in this area. Logs runtime provides observations of system behavior and performance anomalies [13].

Understanding the functionality of containers in distributed systems is crucial. Logs play a role in that aspect. Regularly monitoring them is essential to ensure operations. During dynamic adjustments or movements of containers they provide informations on resource usage and network activities among details. Moreover they assist in failures or performance issues by detecting any unusual behaviors [12].

2.5 Google Cloud Platform

Cloud services, like Google Cloud play a role in supporting distributed systems by offering solutions for managing logs and processing real-time data effectively across various network nodes with a focus, on resource flexibility and consistent performance [14].

Another significant benefit is that using sets of machines (eg, Google Cloud) can allow for scalable log management. Many, if not most, distributed systems produce high volumes of logs, created by the machines and modules that make up the functioning system. All these logs must be stored, analyzed, and acted on as they are generated in real-time. Google Cloud supplies the necessary infrastructure that can handle this scale: its systems give the capability of making the storage of the logs dynamic relative to the current needs, which means that logs are still available for analysis even when traffic is high. Using distributed systems in this way provides for scalable storage and immediate processing of logs [15].

Furthermore the platform can offer an overview of the systems health by merging log data with system metrics. With the help of tools that seamlessly connect with cloud platforms, such as Google Cloud logs can be merged with monitoring methods to identify irregular patterns or deviations from the system's expected behavior. Using a log based strategy for identifying anomalies is crucial in catching irregularities (incidents) that could lead to a decrease in system performance [14]. Google Cloud gives effective tools to analyze logs systematically for ongoing system monitoring support to address emerging problems, in distributed systems promptly.

Google Cloud also supports fault injection tests commonly employed in distributed systems to assess system resilience by causing controlled failures to observe system reactions effectively. Logs created during experiments play a role, in diagnosing issues and pinpointing areas for enhancement. Google Clouds' capability to efficiently manage and analyze log data during these assessments ensures that fault tolerance mechanisms can be thoroughly assessed and fine-tuned for performance [15]. This procedure plays a role, in enhancing the system's robustness by identifying and rectifying vulnerabilities before they lead to major disturbances.

Additionally, Google Cloud offers support, for automated log analysis by enabling the integration of machine learning models to identify patterns and irregularities in logs. Tools such as LogAI can be used to process and examine log data without requiring involvement enhancing the effectiveness of anomaly detection [16]. This feature is particularly beneficial, in distributed settings, where the high volume of logs produced would make analysis unfeasible. By utilizing the potential of cloud services businesses can enhance the accuracy and effectiveness of analyzing logs. This helps in identifying and addressing problems before they affect the system functionality.

Overall, google Cloud plays a role in managing and processing logs for distributed systems. By providing a cost solution for collecting and analyzing logs from regular operations as well as fault injection scenarios, It enhances resilience and reliability. Its robust infrastructure also enables integration with fault-detection tools.

2.6 Log Templating

Analyzing log data is key for grasping how systems function and pinpointing problems in distributed systems. The disorganized format in logs, especially application logs, presents challenges for thorough examination and assessment purposes. Log templating is a strategy employed to systematize log data by extracting organized details from log entries; this method establishes templates that streamline log analysis procedures and facilitate the extraction of pertinent details while recognizing trends and anomalies effectively [17].

In distributed settings, logs are produced by parts, each of which may create logs in different styles. This diversity can lengthen log analysis. To tackle this problem, log templating is used to spot shared patterns in log entries and arrange them into templates. These templates simplify fixed aspects of the log (like text) and separate the changing elements (like timestamps, error codes, or system statuses). Standardizing log entry formats through templating streamlines the process of analyzing logs effectively across components and systems [7].

Creating log templates typically requires parsing log data to extract information and eliminate details efficiently; distinguishing between fixed and variable components in logs is necessary for retaining only key data for analysis purposes. After parsing, logs are categorized into templates representing various system events. This templating approach proves especially beneficial in complex distributed systems where the sheer volume of log data can easily become overwhelming [17] [7].

Log templating is a practice widely used in detecting anomalies in distributed systems. By transforming log data into organized templates. This makes it simpler to spot irregularities from the usual pattern of operation, such as out-of-the-ordinary event sequences or atypical error structures. It might signal underlying problems or potential breakdowns. Employing templating techniques in this context allows for automating the comparison of generated logs with templates. This helps in swiftly spotting anomalies without requiring manual examination [17].

Templating proves its worth in settings with a flood of logs that require real-time analysis to be effective. Tools for automated log parsing have been created to meet this demand. For instance, online log parsing systems create templates as new log entries come in, allowing utilization of real-time analysis and speedier anomaly detection [17]. This becomes important for distributed systems since action against anomalies helps avoid interruptions in the system's operation.

Recent progress in self-taught log parsing methods has opened up possibilities for developing solutions on a scale. Machine learning techniques are used to analyze logs autonomously and generate templates without the need for labeled data. These approaches rely on identifying patterns within the log data to improve template accuracy significantly. This reduces the necessity for intervention, Broadens the application of log templating in intricate distributed setups where log formats may differ significantly among components.

To sum it up, using log templates is crucial for organizing log information in distributed systems. The utility of templating for anomaly detection is derived from ordering logs into a fixed log schema that can be processed efficiently at scale, making log templating an even more important aspect of operating scalable distributed systems with constantly increasing quantities of log data. The utility of log templating can be further enhanced through automation of an improved process for log parsing, such as the one described by Zhang et al., and self-supervised approaches to deriving log signatures. Both techniques are up-to-date approaches to normalizing log data in distributed systems.

3 Related Work

3.1 System Failures

Researchers have shown an increased interest, in investigating system failures in distributed systems because of the challenges involved in sustaining reliability and performance over time. Distributed systems are particularly prone to failures triggered by hardware issues, software glitches or disruptions, in network connectivity. When these failures happen within systems they can spread across components. Impact the systems overall functionality. To address and minimize these system failures scientists have developed techniques to simulate and analyze the behavior of systems during malfunctions.

Injecting faults is a technique utilized to replicate scenarios of failure in a controlled lab environment where detailed records of system activities are accessible both pre and post the onset of a failure incident [18] [19]. These experiments offer insights, into how distributed systems react when subjected to stress with the aim of vulnerabilities in the systems, for efficient mitigation or resolution. Logs produced during fault injection tests are a resource for pinpointing failures and understanding their causes better by documenting the events before and after a failure occurs in a chronological manner. In distributed systems, where different parts interact closely, analyzing logs becomes key to unraveling how failures spread throughout the system [20]. System logs typically have a format that offers an account of process-wide activities, which simplifies the analysis process. However, records within the application itself can pose challenges when trying to understand them, as they document interactions occurring at the application level.

Though fault injection using system logs has been implemented, application logs have received hardly any attention, even though they contain essential aspects of the internal state of applications, including interactions with other components—an information-rich source for failure diagnosis. This is another area in which much more research is needed to learn how to best utilize these logs [9]. One broader challenge that should be tackled is applying these advanced analysis techniques in various complex systems, which exhibit a significant degree of variability. Diagnosing failures based on event logs, particularly application logs, entail challenging tasks, requiring more advanced analysis approaches.

3.2 Anomaly Detection

Making a distributed system reliable so it can be run efficiently and not fail in unplanned scenarios is a major challenge for getting things done. Detection of anomalies is one of the basic approaches to accomplishing this. "Anomaly" is generically used to describe the misbehavior of a system: anomaly detection identifies any event or behavior that deviates from the normal scheduling of resources or services and indicates a problem when it occurs. Timely detection of an anomaly offers a chance of avoiding failure. Through the years different approaches have been investigated to identify irregularities by employing methods, like machine learning and rule based systems.

Machine learning models are widely used for detecting anomalies. They can analyze large amounts of data and spot irregularities automatically based on learned patterns from the datasets provided. In particular, deep learning models recognize nonlinear connections within the data. It makes them well-suited for spotting anomalies in organized data, like system logs [21]. These models, however usually require labeled datasets to function and which acquiring can be challenging when dealing with application logs. They lack structure, are noisy, and differ between systems. This makes the process of organizing labeled data for training models quite complicated [5].

By contrast, interpretability poses a challenge. Conventional complex algorithms in machine learning are famous for being difficult to explain. Deep learning models are often seen as enigmatic meaning explaining why an anomaly was detected can be a difficult. This lack of clarity can present difficulties trying to pinpoint the root cause of a problem for rectification. Sometimes it is crucial to know the reasons behind labeling a behavior as abnormal and identifying the root issue [6]. This in essence restricts the use of machine learning methods to settings. Rule based or threshold based approaches that do not rely on machine learning have been widely used to identify anomalies for a while [22]. These methods use predefined rules to recognize deviations, from expected patterns and offer an explanation for flagging an event as an anomaly. However human crafted techniques face challenges in accommodating the characteristics of logs. Additionally detecting anomalies that do not align with rules or thresholds may pose limitations in such methodologies.

Currently, there is another issue in anomaly detection research that requires attention. The distinction, between studying system logs and application logs remains a focus area for investigation and development efforts in the field of anomaly detection research. While system logs are known for their nature and compatibility with existing anomaly detection methods; application logs pose a challenge due to their lack of structure and consistency [4]. Application logs typically capture events at the application layer but necessitate sophisticated techniques, for log preprocessing and parsing before analysis can be effectively carried out [7]. In order to bridge this discrepancy in knowledge analysis methods, this thesis introduces an anomaly detection system that zeros in on uncovering anomalies in application logs following log analysis. Post-events are examined carefully for any irregularities that may surface. By shifting through logs obtained from experiments involving fault injection, these anomalies are brought to light as a result of factors such as log severity, frequency, and fault-related characteristics. Unlike machine learning systems, it does not rely on labeled data, making it a good option for analyzing unstructured application logs. Furthermore, the identified irregularities could be utilized in the training of machine learning algorithms to enhance their precision and facilitate the transition from log inspection to fully automated anomaly detection [23].

This approach enhances research by offering an approach to pinpoint irregularities, in application logs without the need for extensive and time-consuming data labeling efforts. In essence anomaly detection is made easier to understand through a method for determining anomalies in distributed systems, which could be beneficial even in scenarios where real time anomaly detection is unattainable.

4 Method

4.1 System Overview

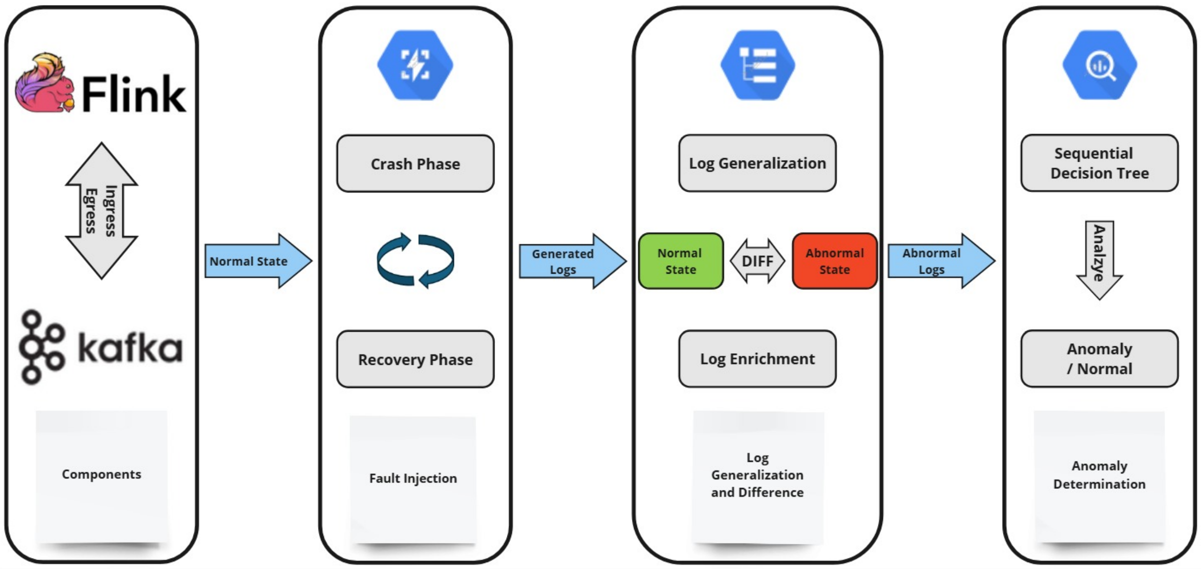

Figure 4.1: System Overview

This section introduces the process flow of the anomaly determination methodology. It offers a high-level view of each stage, setting the context for the following sections, which detail the specific components, fault injection process, log processing and anomaly determination approach. **Figure 4.1** illustrates the entire methodology, showing how each part contributes to identifying potential anomalies in the application logs, generated by the implemented distributed system.

The process begins with the **Components** of the distributed system, which include Apache Kafka and Apache Flink. These components manage data ingestion, processing and task execution, as explained in section **4.2 Components**. Logs are initially collected under normal operating conditions to establish a baseline for standard behavior.

The next step is the **4.3 Fault Injection**, where controlled disruptions are introduced to simulate real-world failures. As illustrated in **Figure 4.1**, these interruptions include both crash and recovery phases, capturing system responses under stress. Logs generated during this process record both direct responses from the fault-targeted component and indirect effects on other components.

After collecting fault-induced logs, they proceed to **4.4 Log Generalization and Difference**.

This stage standarizes logs into a consistent format. This sorts abnormal logs from normal ones and adds extra context. This process helps spot patterns that may signal faults, as shown in **Figure 4.1**.

Finally, the processed logs are analyzed in the **4.5 Anomaly Determination** section. A sequential decision tree evaluates each log entry based on multiple attributes like severity and rarity, flagging logs with potential anomalies. This structured approach, shown on the right side of **Figure 4.1**, provides Intuitions into system behavior under fault conditions.

To sum up, this system overview outlines each step in the methodology, from data collection through to anomaly determination. The next section, **Components**, will detail the roles and configurations of each distributed system element involved in this process. The implementation can be accessed on Github [1].

4.2 Components

This section describes the main components of the distributed system—**Apache Kafka (Redpanda), Apache Flink, and the Functions Service**. Each component's function, resilience configurations, and network setup are outlined, focusing on their roles in generating logs for anomaly determination. These configurations are essential for simulating fault conditions, observing system responses, and preparing the system for the **Fault Injection** phase.

Apache Kafka

Apache Kafka acts as the messaging system for the platform. It is in charge of handling the flow of real-time data among producers, processors and consumers. Kafka is set up with replication, partitioning to ensure data redundancy and seamless operation in case of failures. Replication enables messages to be retrieved from brokers while partitioning spreads data across nodes for processing.

The Docker setup for Kafka includes limited memory (1 GB) and single core processing to replicate an environment for testing purposes. By imposing these restrictions it becomes possible to observe how Kafka functions under stress and assess its performance in managing message processing and data integrity when resources are limited. Various tests are conducted to evaluate Kafkas behavior when faced with memory limitations and message handling challenges. The logs produced during these tests capture Kafkas reactions to resource strain and pressure drawing attention to its resilience in demanding situations.

Kafka's configuration thus supports testing for data availability and stability under various conditions, generating logs that reflect its performance under load. The next subsection focuses on Flink's configurations for distributed processing and state management.

Apache Flink

Apache Flink handles distributed data processing within the system by coordinating tasks and ensuring their execution efficiently. It involves two roles, The JobManager that distributes tasks and the TaskManager that executes them with the added benefit of incremental checkpointing using RocksDB configured in the JobManager for efficient state management and quick recovery in case of faults.

[1] https://github.com/anisbs96/ba-implementation

Similarly, TaskManagers use incremental checkpointing to retain task state, which enables seamless task resumption after recovery. The fixed-delay restart strategy permits automatic retries after failures, maintaining operational continuity. Flink's heartbeat settings, with a 1-second interval and a 5-second timeout, support timely detection of task failures and prompt recovery. Both JobManager and TaskManager have memory limits of 1 GB, simulating constrained resources to test stability under limited memory.

Flink's StateFun Runtime further extends its functionality, handling stateful functions asynchronously and using exactly-once delivery semantics with Kafka to ensure reliable message processing without duplication or loss. Overall, Flink's configurations support reliable data processing and task recovery, generating logs that capture fault responses. The following subsection covers the Functions Service, which manages custom application logic in collaboration with Kafka and Flink.

Functions service

The Functions Service manages application logic, and event-driven processing activities colaborating with Kafka and Flink to handle live data streams efficiently in real-time scenarios. The communication with Flink's StateFun is asynchronous, allowing for handling numerous events without disrupting other operations. This setup supports testing the service's response to concurrent requests under varying conditions.

The Functions Service operates on a dedicated network port, which facilitates targeted fault injections by isolating network-related disruptions. Simulated network issues, such as latency and disconnections, generate logs on the service's responses, providing insight into its resilience under stress. Configured for asynchronous event handling and isolated network interactions, the Functions Service supports testing of network and resource-related faults. Logs from these tests capture the service's behavior under high-load or disrupted network conditions. With each component configured for resilience, the next section, **Fault Injection**, will detail the controlled disruptions applied to assess system stability.

Containerization and GCP Logging

The Docker containerization and network setup enable fault injection and monitoring across the distributed system. Each component runs within a Docker container and connects through a bridge network, allowing isolated fault injections and supporting network-based tests, including latency, packet loss, and disconnections.

GCP logging and monitoring are used to track system behavior during fault conditions, capturing data on how each component and the network respond to various disruptions. These tools reveal details into the system's resilience and performance under fault scenarios. The containerized network and GCP monitoring enable precise fault testing and performance tracking, contributing to an understanding of component resilience. With the network and components configured, the next section, **Fault Injection**, will describe the fault scenarios applied to evaluate system stability.

4.3 Fault Injection

This section outlines the fault injection methodology used to simulate controlled disruptions across the system's core components: Flink Master, Flink Worker, Kafka Broker, and the Functions Service. Each fault scenario was designed to test system resilience by targeting one component at a time with multiple fault types, generating structured log data that captures both immediate responses and recovery patterns. This data will be essential for anomaly determination in later phases. Fault injection is applied to evaluate each system component's resilience under realistic failure conditions. By focusing on a single component and subjecting it to various fault types—such as memory limits, network delays, and crashes—the methodology captures nuanced component-specific behaviors. Each component operates within a dedicated Docker container connected by a shared bridge network. This isolated setup ensures that disruptions affect only the targeted component, enabling precise log generation without unintended interference across the system.

Logs are timestamped and structured to record component behavior during both the active fault phase (crash phase) and the recovery phase, where the system stabilizes after the fault is removed. This setup produces a detailed dataset that tracks each fault's full impact, supporting comprehensive analysis in subsequent anomaly determination.

Fault Types per Component

- **Flink Master**:
  - **Container Crashes**: Simulates unexpected crashes by terminating the JobManager container, logging impacts on task scheduling and coordination.
  - **Memory Constraints**: Limits memory allocation for the JobManager, testing job scheduling and task management under reduced resources.
  - **Heartbeat Timeout Reductions**: Shortens the heartbeat interval to simulate network latency, assessing the JobManager's ability to detect and recover from node isolation.
- **Flink Worker**:
  - **Memory Constraints**: Reduces available memory for the TaskManager, testing processing capability under constrained resources.
  - **Task Saturation**: Increases task assignments to simulate high load, observing TaskManager handling of overload conditions.
  - **Job Cancellations and Terminations**: Abruptly cancels tasks mid-execution, analyzing propagation of task cancellations.
  - **Crashes During Checkpointing**: Forces a crash during checkpointing to observe how the TaskManager manages interrupted task states.
- **Kafka Broker**:
  - **Broker Crashes**: Terminates the broker container, requiring Kafka to recover using replication.
  - **Memory Constraints**: Limits memory, testing Kafka's ability to maintain message flow under reduced resources.
  - **Partition Failures**: Makes partitions unavailable to assess message redistribution capabilities.
  - **Consumer Group Disruptions**: Disrupts consumer connections, observing Kafka's ability to maintain consistent delivery.
  - **Transaction Timeouts**: Introduces transaction timeouts, testing exactly-once deliv-

    ery under delay.
  - **Network Disruptions**: Injects latency and packet loss, assessing Kafka's handling of delayed and dropped packets.
- **Functions Service**:
  - **Service Crashes**: Terminates the Functions Service container, capturing its task handling and recovery during high load.
  - **Network Disruptions**: Introduces network instability to analyze impact on service communication with other components.
  - **Memory Constraints**: Limits available memory, testing processing capabilities under resource shortages.

Every unique fault situation for each part gives an understanding of how they respond when faced with faults and provides insights into their individual effects, as well as how they interact with each other. The subsection that follows details the stages of system failure and subsequent recovery phases to enrich the log data by recording not only the consequences of faults but also the solutions applied to address them.

Crash and Recovery Phases

Each fault scenario is split into two stages : the **crash phase** and the **recovery phase**. These phases provide a complete perspective on system resilience by capturing both initial impacts and the stabilization process. During the **crash phase**, the fault is activated, generating logs that document real-time responses within the affected component. Logs from this phase highlight immediate failures, adjustments in task scheduling, and any interruptions in data flow. In the **recovery phase**, the fault is removed, and the system begins stabilizing. Logs capture the return to normal operation, providing insights into resilience and recovery time. Together, these phases create a structured dataset documenting the full lifecycle of each fault, helping understand component resilience under different failure scenarios.

By dividing each fault scenario into crash and recovery phases, the methodology captures both the immediate disruptions and recovery patterns, generating data essential for analyzing resilience. The next subsection describes how this log data is structured and enriched to support effective anomaly determination.

Data Preparation for Anomaly Determination

The fault injection process produces enriched log data with attributes that link each entry to specific fault scenarios. Each log entry includes details about the fault type, source component, and timestamp, which supports tracking both direct fault impacts and indirect effects across components.

This structured dataset is essential for the **Log Generalization and Difference** phase, where patterns unique to fault scenarios are identified, enabling potential root cause analysis. By associating logs with fault scenarios, deviations from normal behavior become easier to detect, providing a foundation for anomaly determination.

By associating log data with fault scenarios and component, this approach highlights variations and patterns related to fault scenarios, facilitating focused anomaly identification. The final subsection discusses challenges and observations from the fault injection process.

Challenges and Observations

The fault injection process reveals complexities in analyzing dependencies and interactions between components. Inter-component correlations become apparent when faults in one component indirectly affect others. For example, Kafka disruptions often impact Flink Workers, providing insights into resilience gaps and potential weaknesses within system connectivity.

Due to the high volume of logs generated by components like Kafka and Flink Worker, it was not feasible to collect logs from all fault types simultaneously. As a result, multiple experiments were conducted, with each experiment focusing on diverse fault types. The most representative logs, which combined container crashes and memory constraints under high message load were selected. While other fault types were also injected while the system was active, only logs from the most diverse scenarios are included in this section. Further details on the results of these fault injections are provided in **Section 5.1 Fault Injection**.

Finally, the structured dataset created through fault injection allows for future root cause analysis by linking faults to specific system behaviors. Detailed logs of each fault and response make it possible to detect patterns that may be useful in advanced anomaly detection.

These observations highlight the complexities of fault interactions within the system, providing valuable interpretations of dependencies and resilience gaps. The fault injection process generates a robust dataset for analyzing system responses, preparing data for the **Log Generalization and Difference** phase.

4.4 Log Generalization and Difference

This section describes the process used to transform raw logs from fault injection scenarios into a structured and enriched dataset, preparing them for anomaly determination. This process involves four stages—**Log Preparation, Log Generalization, Log Difference, and Log Enrichment**—that collectively filter, template, and enhance logs. **Log Generalization and Difference** reflects the section's main objectives: to generalize logs through consistent formatting and to identify unique, fault-specific logs by differentiating them from normal logs. This systematic approach ensures that only relevant and meaningful logs are retained, allowing for a better anomaly determination later.

Log Preparation

In the first stage, logs are cleaned and organized to ensure consistency and quality. Logs are collected from both normal and fault scenarios, with each entry labeled by a `fault-target` attribute to differentiate normal from fault-specific logs. Only essential attributes, including `component` and `message` are retained. Entries with missing values are removed to maintain data quality.

An example of a log entry after preparation is shown in **Table 4.1**, with the following attributes:

- **fault-target**: Identifies the component targeted by the fault (e.g., `flink-master`), even though this log was generated by another component, `flink-worker`. This label supports later differentiation of fault-induced logs from normal logs. This attribute will be later used to determine whether the failure was a direct one caused by the same component targeted or not.

- **component**: Specifies the component that generated the log (e.g., `flink-worker`).
- **message**: Contains the detailed log message including timestamps, severity, and key descriptors of the event which will be extracted in later stages.
- **timestamp**: Records the exact time of the log entry, formatted consistently for easier sorting and comparison. It is extracted from the `message` later and is going to be used in identifying the crash or recovery phase. It is also needed to calculate the log frequency change in the log enrichment phase.

This step ensures that the dataset is clean and focused, containing only relevant information for further processing.

fault-target	component	jsonPayload.message
flink-master	flink-worker	2024-11-08 23:19:24,705 ERROR org.apache.flink.runtime.taskexecutor. TaskManagerRunner [] - Fatal error occurred while executing the TaskManager. Shutting it down...

Table 4.1: Sample Log after Preparation

Log Generalization

In the log generalization stage, log messages are templated by removing dynamic elements, such as IP addresses and file paths, to create a standardized format. This reduces log volume by grouping nearly identical messages. The process includes extracting severity levels (e.g., ERROR, WARN) and replacing variable data, like timestamps and unique IDs, with placeholders. This consistent templating focuses on the core message content. For example, in the log from **Table 4.1**, the `jsonPayload.message` attribute is generalized by stripping out variable elements and identifiers, while the severity level `ERROR` and `Timestamp` are retained. This transformation yields a standardized log, simplifying the detection of unique patterns associated with faults.

Log Difference

The log difference stage separates fault-specific logs from normal logs to identify events unique to fault conditions. Normal logs are stored separately, creating a reference for expected patterns under non-fault conditions. Fault logs are then filtered to exclude entries that match normal logs, leaving only fault-specific logs, which highlight events that deviate from routine system behavior.

In this stage, the example log remains in the fault-specific set, as it is unique to the fault injection scenario targeting `flink-master`. This exclusivity suggests that it may be relevant as a potential anomaly.

Log Enrichment

In the final stage, logs are improved by adding details to each entry. This stage enriches the dataset with information that helps in understanding the significance of logs in relation of faults. The essential attributes added, as shown in **Table 4.2**, include:

- **Phase Labeling**: Logs are tagged as `crash` or `recovery` based on their timestamps, helping to analyze logs in the context of the system's operational phase.
- **Frequency and Frequency Change**: Analyzing logs over time intervals following crashes allows for monitoring the frequency and patterns of changes between phases. This metric

helps spot trends like: Logs becoming less frequent during recovery (indicating stabilization) or a sudden increase in activity after a crash, suggesting intense system activity or adjustments.

- **Component-Specific and Fault-Specific Labels**: Logs get flagged if they seem related to a specific component or fault scenario, pinpointing distinct occurrences to certain conditions.
- **Rarity**: Logs are labeled as "rare" if they occur only once, are unique to one component, and one fault type, helping to highlight unusual rare events.
- **Direct Cause**: Logs are marked as "direct cause" if the fault target matches the component generating the log, helping to distinguish primary effects from secondary effects.

Attribute	**Value**
severity_level	ERROR
templated_message	TaskManagerRunner [] - Fatal error occurred while executing the TaskManager. Shutting it down
direct_cause	False
rare	False
fault-target	flink-master
fault-specific	True
component	flink-worker
component-specific	True
phase	crash

Table 4.2: Sample Log after Enrichment

The example log, as shown in **Table 4.2** after enrichment, illustrates how these attributes provide essential context. It is flagged as fault-specific to `flink-master` but component-specific to `flink-worker`, suggesting a possible collateral effect. Although not rare, its high severity and relevant keywords make it worthy for further investigation in anomaly determination.

This particular log collected during a fault injection targeting the `flink-master` captures an unexpected crash of the `flink-worker` shortly afterward. With attributes like high severity, fault-specificity to `flink-master`, and component-specificity to `flink-worker`, this log stands out for its potential collateral effect, even though it is not rare. This example will be revisited to determine if it might qualify as a potential anomaly in later analysis.

4.5 Anomaly Determination

This section presents a structured approach to determine whether logs are normal or abnormal, using a sequential decision tree illustrated in **Figure 4.2**. The decision tree evaluates key attributes of each log, such as severity, keywords, component specificity, and frequency change. It identifies deviations indicative of anomalies under fault conditions. This process captures a multi-dimensional view of each log, incorporating content, context, behavioral, and timing-based assessments for comprehensive anomaly determination.

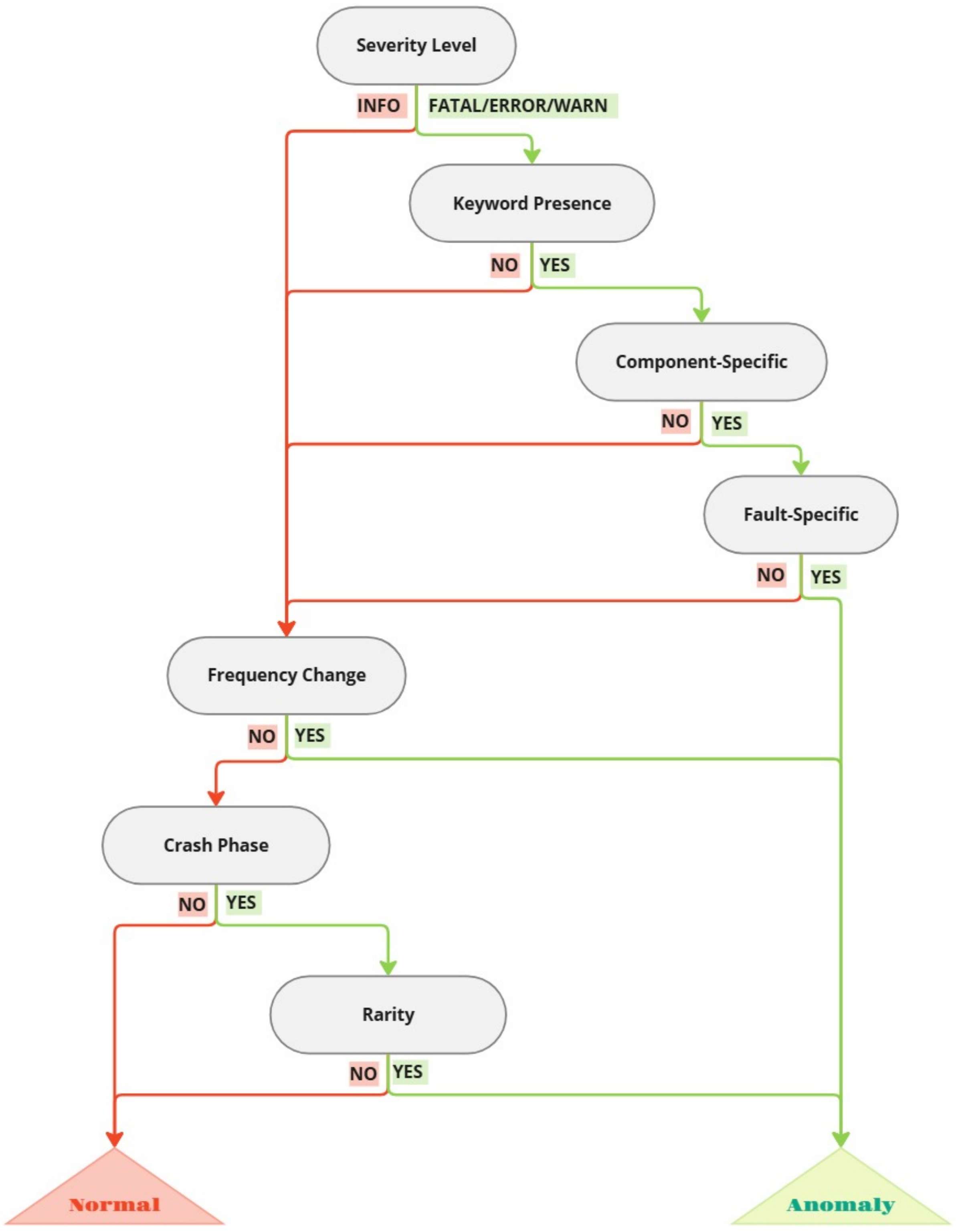

Figure 4.2: Sequential Decision Tree

The sequence of processes in the decision tree signals a priority-based approach:

1. **Severity and Keywords**: Serve as initial filters to identify high-severity, failure-indicating logs.
2. **Component and Fault Specificity**: Add important context, focusing on logs that directly involve critical components or specific fault scenarios.
3. **Frequency Change, Crash Phase, and Rarity**: Capture recurring patterns and unique, high-impact events, ensuring precise identification of abnormal behavior.

---

1. **Severity Check**
   - **Type**: **Content-Based** — Severity directly highlights the importance of the log message, signaling critical issues.
   - **Content**: Logs with high severity levels (`ERROR`, `FATAL`, or `WARN`) proceed to the keyword check. Logs with `INFO` severity, generally informational, bypass keyword analysis and move directly to frequency change checks.
   - **Sufficiency for Anomaly Determination**:While severity alone cannot determine an anomaly, it acts as a critical filter. High-severity logs typically require additional context or behavioral evidence to confirm their significance.
   - **Rationale**: Severity is a primary filter, prioritizing logs likely to reflect urgent issues while minimizing false positives from non-critical events.
   - **Decision and Transition**:
     - If the log's severity level is `ERROR`, `FATAL`, or `WARN`, it transitions to the **Keyword Presence Check**.
     - If the severity is `INFO`, the log skips keyword checking and moves to **Frequency Change Check** for behavioral assessment.

---

2. **Keyword Presence Check**
   - **Type**: **Content-Based** — Keywords indicate specific fault-related events within the log content.
   - **Content**: The log message is scanned for failure-related keywords (e.g., "error," "timeout," "exception", "fail", "critical", "fatal", "crash", "lost"), which are commonly associated with faults. Logs containing these keywords proceed to the component-specific check, while others go to frequency change analysis.
   - **Sufficiency for Anomaly Determination**: Keywords alone do not determine an anomaly, but their presence increases the likelihood of fault relevance. Keywords paired with high severity create a stronger case for a potential anomaly.
   - **Rationale**: Keywords help refine the scope by isolating logs that explicitly reference failure conditions, narrowing down high-severity logs to those with more precise fault associations.
   - **Decision and Transition**:
     - If keywords are detected, the log proceeds to the **Component-Specific Check**.

    ◦ If no keywords are present, the log transitions to **Frequency Change Check**.

---

3. **Component-Specific Check**
    - **Type**: **Context-Based** — This step evaluates the relationship between the log and specific system components.
    - **Content**: Logs specific to a single component (e.g., Flink worker or Kafka broker) move to the fault-specific check. Logs not tied to any specific component proceed to the frequency change check.
    - **Sufficiency for Anomaly Determination**: While component source alone may not determine an anomaly, it adds vital context, as logs from key components often warrant further analysis. When combined with other factors, component-specificity strengthens anomaly determination.
    - **Rationale**: Component specificity helps identify where the issue originates, providing perspectives into localized impacts within vital system parts, especially if they are involved in fault scenarios.
    - **Decision and Transition**:
        - If the log is component-specific, it advances to the **Fault-Specific Check**.
        - If not component-specific, the log transitions to **Frequency Change Check**.

---

4. **Fault-Specific Check**
    - **Type**: **Context-Based** — Fault-specificity assesses whether the log is unique to a specific fault scenario.
    - **Content**: Fault-specific logs are determined as anomalies (*Anomaly-FS*) due to their close alignment with specific fault scenarios. Logs that do not match a fault scenario proceed to frequency change analysis.
    - **Sufficiency for Anomaly Determination**: Logs unique to a specific fault scenario, especially those with high severity and failure keywords, are generally sufficient to identify as anomalies. This sufficiency increases when the log is also component-specific, meaning it originates from only one component. Together, fault-specific (appearing only in one fault) and component-specific (originating from a single component) attributes strengthen the log's determination as an anomaly.

      Additionally, the combined checks up to this point—**Severity Check**, **Keyword Presence Check**, **Component-Specific Check**, and **Fault-Specific Check**—create a layered filter that integrates **content**, **context**, and **specificity** attributes. Each previous check progressively narrows the focus to logs that are not only high-severity but are contextually tied to a single component and a unique fault scenario. This multi-check approach minimizes the risk of false positives by confirming that the log is both contextually relevant to the fault and unique in its occurrence, contributing a robust basis for determining it as an anomaly.
    - **Rationale**: Fault-specificity combined with component-specificity provides a strong indication of an anomaly, as logs appearing in a single fault scenario and from a

single component are likely to be directly related to the injected fault, revealing more about the source and scope of the issue.

- **Decision and Transition**:
  - If the log is both fault-specific and component-specific, it is determined to be an anomaly (*Anomaly-FS*).
  - If it is not fault-specific, it moves to **Frequency Change Check**.

---

5. **Frequency Change Check**
   - **Type**: **Behavioral** — This check analyzes the recurrence pattern of log messages, identifying persistent issues or bursts in frequency.
   - **Content**: The frequency change is calculated in time windows between consecutive crash phases, observing log behavior between the crash and recovery phases. A burst in log frequency during the crash phase and a noticeable decrease during recovery may indicate a potential failure. Logs with such a significant increase in frequency are determined as anomalies (Anomaly-FC). Logs without frequency change continue to the crash phase check.
   - **Sufficiency for Anomaly Determination**: A significant frequency change, particularly an increase during the crash followed by a drop in recovery, often signals a recurring or unresolved issue that points to a potential anomaly. The change in frequency across these phases provides observations on how the log pattern responds to system disruptions, making it sufficient in many cases to determine an anomaly.
   - **Rationale**: This attribute captures persistent anomalies that occur repeatedly during faults, indicating sustained system stress or failures. Monitoring behavior across both crash and recovery phases adds depth to the analysis, as fluctuations in log frequency may reveal stability issues.
   - **Decision and Transition**:
     - If there is a positive frequency change (e.g., a burst during the crash and a decrease in recovery), the log is determined to be an anomaly (*Anomaly-FC*).
     - If no significant frequency change is detected, the log transitions to the **Crash Phase Check**.

---

6. **Crash Phase Check**
   - **Type**: **Timing-Based** — Timing-Based — This check evaluates the timing of log entries concerning the crash or recovery phases.
   - **Content**: Logs occurring during the crash phase are directed to the rarity check for further examination. Logs from the recovery phase are generally considered normal (Normal-CP), often reflecting stabilization processes.
   - **Sufficiency for Anomaly Determination**: Logs in the crash phase alone may not be anomalies, but timing information during crashes highlights critical events, especially when paired with other indicators.

- **Rationale**: Distinguishing between crash and recovery phases reduces the likelihood of falsely determining recovery logs as anomalies.
- **Decision and Transition**:
  - If the log is in the crash phase, it proceeds to the **Rarity Check**.
  - If it is in the recovery phase, it is determined as normal (*Normal-CP*).

---

7. **Rarity Check**
   - **Type**: **Behavioral** — This step assesses the uniqueness of the log entry by evaluating its frequency and occurrence phase.
   - **Content**: The rarity check is triggered when there is no significant frequency change. However the log occurs during the crash phase, this indicates that the log may not be frequently recurring but might represent an unusual event during a pivotal time. The check identifies rare events based on the frequency of occurrence, focusing on low-frequency logs that appear during the crash phase, which can indicate an anomaly if other checks do not determine it as one.
   - **Sufficiency for Anomaly Determination**: The rarity of a log, particularly one that appears during a crash without showing a frequency burst, may be sufficient to indicate a potential anomaly. Since reaching this step means the log hasn't met the conditions of the first four checks, the rarity check captures rare but contextually significant logs, ensuring that unusual, low-frequency events in the crash phase are still flagged.
   - **Rationale**: The rarity check serves as a final filter, isolating unique logs that appear only occasionally, often capturing one-off issues that might still indicate an underlying problem due to their presence in the crash phase.
   - **Decision and Transition**:
     - If the log is rare (appears infrequently and only during the crash phase), it is determined to be an anomaly (*Anomaly-RC*).
     - If it does not meet rarity criteria, it is determined as normal (*Normal-RC*).

The decision tree process is illustrated through two distinct log examples: **Example 1** (Table 4.3) and **Example 2** (Table 4.4).

- **Example 1 (Table 4.3)**: This log from the Flink worker captures an indirect anomaly during the crash phase of a fault targeting the Flink master. Although the Flink-worker was not the faulty target, it experienced an indirect impact from the Flink-master's failure. The high severity (ERROR), fault and component-specific attributes, and critical keywords (e.g.,"fatal") suggest that this anomaly likely resulted from sequential effects due to the targeted component's crash.
- **Example 2 (Table 4.4)**: In contrast, this log from Flink-master reflects a direct anomaly encountered during the recovery phase. As the Flink-master attempted to tear the cover, it logged a warning (WARN) related to a connection failure. This issue may be connected to the Flink worker's unavailability following its crash in Example 1. The direct cause, high fault specificity, and recovery phase context identify this as a unique anomaly.

Attribute	Value
determination	`Anomaly-FS`
severity_level	`ERROR`
templated_message	`TaskManagerRunner [] - Fatal error occurred while executing the TaskManager. Shutting it down`
direct_cause	`False`
rare	`False`
fault-target	`flink-master`
fault-specific	`True`
component	`flink-worker`
component-specific	`True`
phase	`crash`
frequency_change	`1.0`
frequency	`4`
frequency_phase	`4`

Table 4.3: Example 1 of Indirect Anomaly Log during Crash Phase

Attribute	Value
determination	`Anomaly-FS`
severity_level	`WARN`
templated_message	`NettyTransport [] - Remote connection to [null] failed with java.net.ConnectException: Connection refused`
direct_cause	`True`
rare	`True`
fault-target	`flink-master`
fault-specific	`True`
component	`flink-master`
component-specific	`True`
phase	`recover`
frequency_change	`0.0`
frequency	`1`
frequency_phase	`1`

Table 4.4: Example 2 of Direct Anomaly Log during Recovery Phase

Combined Insights

- **Propagation of Fault Effects**: The examples illustrate distinct anomaly types—*indirect (collateral) anomalies* affecting non-target components and *direct anomalies* within the fault-target component.
- **Timing and Recovery Challenges**: These examples underline how anomalies may show during both the crash and recovery phases. This demonstres the importance of phase-based timing in detecting fault persistence or collateral impacts.
- **Determination Process Effectiveness**: These logs highlight the decision tree's ability to distinguish between direct and indirect impacts, utilizing attributes like severity, component and fault specificity, and phase timing to achieve a nuanced understanding of fault effects.

This section established a systematic process for identifying anomalies in distributed system logs through a sequential decision tree. By examining each log's attributes—ranging from severity and keywords to component and fault specificity, frequency changes, phase timing, and rarity, the decision tree provides a multi-faceted approach to anomaly determination. This method efficiently narrows down logs to those most likely associated with system faults, distinguishing between routine events and significant deviations that might signal underlying failures.

The layered checks in the decision tree enable a nuanced view, capturing both direct impacts, as seen in fault-targeted components, and indirect or collateral effects propagating through the system. The chosen attributes and their sequence contribute to a robust process that limits false positives while emphasizing contextually relevant anomalies.

This following section uses a decision tree to analyze fault injection results. By observing events under fault conditions, we understand resilience, fault spread, and areas needing recovery.

5 Evaluation

The Evaluation chapter examines the system's resilience. It reviews, under fault conditions, log data produced during fault injection experiments. This analysis seeks to grasp each component reaction to faults, identify dependencies, and detect patterns of fault propagation.

The experiments were carried out in a controlled environment. Each component was set up on a separate virtual machine (VM) in Google Compute Engine, configured as an e2 medium instance with 2 vCPUs, 1 core, and 4 GB of memory. This layout isolated each component for fault testing, allowing thorough examination and monitoring system-wide effects.

Fault injections followed a structured and sequential process. Each involved a five-minute fault period followed by a five-minute recovery period for 30 minutes. This ensured that each experiment tested only one component's resilience at a time.

Continuous log collection was performed using Google Cloud Logging. It gathered data from each component throughout each fault scenario. Logs were filtered and labeled by fault scenario and operational phase (crash or recovery) to map faults and system responses accurately.

The chapter follows an organized approach. It examines system behavior in steps. In that way it can assess component sensitivity and overall robustness through several key stages:

- **5.1 Generated Log Data**: Initial analysis of logs across processing stages to establish baseline behaviors under fault conditions.
- **5.2 Fault Injection**: Examination of component responses to faults, with a focus on dependencies and cascading effects.
- **5.3 Results**: Identification and analysis of anomalies to differentiate direct from collateral impacts, assessing component sensitivity.

5.1 Generated Log Data

This section examines log counts for both fault targets and components across four processing stages: Preparation, Generalization, Difference, and Enrichment. It Trackes how log counts change at each stage provides findings on logging activity and the effectiveness of data filtering, which sets up a baseline for understanding system behavior under fault conditions.

Tables 5.1 and 5.2 display the number of occurrences for each type of fault and component in all four stages :

- **Preparation**: Initial filtering of relevant log entries.
- **Generalization**: Standardizes entries and reduces the log count by removing variable patterns.

- **Difference**: Filters out routine logs, narrowing down to fault-specific entries.
- **Enrichment**: Adds attributes like phase labeling to maintain relevance to fault scenarios.

fault-target	Preparation	Generalization	Difference	Enrichment
flink-master	3812	1028	46.0	47.0
kafka-broker	14500	1022	38.0	43.0
flink-worker	19961	998	15.0	22.0
functions	6939	1005	21.0	22.0
normal	1471	986	0.0	0.0

Table 5.1: Log Count per Fault-Target across Processing Stages

Index	Component	Preparation	Generalization	Difference	Enrichment
1	flink-worker	20506	255.0	52.0	72.0
3	kafka-broker	8406	709.0	32.0	32.0
0	flink-master	17699	174.0	20.0	30.0
2	functions	22	0.0	0.0	0.0
4	kafka-console	50	10.0	0.0	0.0

Table 5.2: Log Count per Component across Processing Stages

In **Table 5.1**, flink-worker and kafka-broker fault targets have high initial log counts in the Preparation stage, with 19,961 and 14,500 logs respectively. This high activity suggests that these components generate more logs under fault conditions. As data progresses through Generalization, log counts decrease across all targets, especially for flink-worker and kafka-broker, which drop to 998 and 1,022 logs. The Difference stage further narrows the data, with log counts dropping significantly to focus on fault-specific entries—15 logs for flink-worker and 38 for kafka-broker. Enrichment adds minimal changes, tagging logs by phase.

This analysis shows that the Generalization and Difference stages effectively filter the data to retain critical fault-specific entries. The high initial log counts for flink-worker and kafka-broker suggest these components respond actively to fault conditions.

In **Table 5.2**, flink-worker and flink-master show the highest initial log counts in the Preparation stage, with 20,506 and 17,699 logs, indicating high logging activity. Log counts decrease significantly for all components through the Generalization and Difference stages, showing effective filtering process. Minimal changes occur in Enrichment, as this stage preserves fault-relevant entries while adding phase information.

This component-level analysis highlights Flink-worker and Flink-master as the most active components in logging during fault conditions. The filtering stages progressively reduce log counts, focusing on fault-relevant data.

Overall, the Generalization and Difference stages effectively refine the data, isolating fault specific entries for both fault targets and components. The high initial counts for flink-worker and kafka-broker suggest that these components produce extensive logs when faulted. As explored in the next section, Fault Injection Results, these findings set a foundation for analyzing how components respond in specific fault scenarios.

5.2 Fault Injection

This section analyzes system behavior under specific fault scenarios. It focuses on how different components respond to targeted faults. By examining log counts per fault target and component and the percentages of enriched attributes. This analysis provides an understanding of component dependencies and the impact of faults on system activity. Tracking these responses across fault scenarios highlights patterns of fault propagation and resilience, preparing the data for detailed anomaly analysis in the following sections.

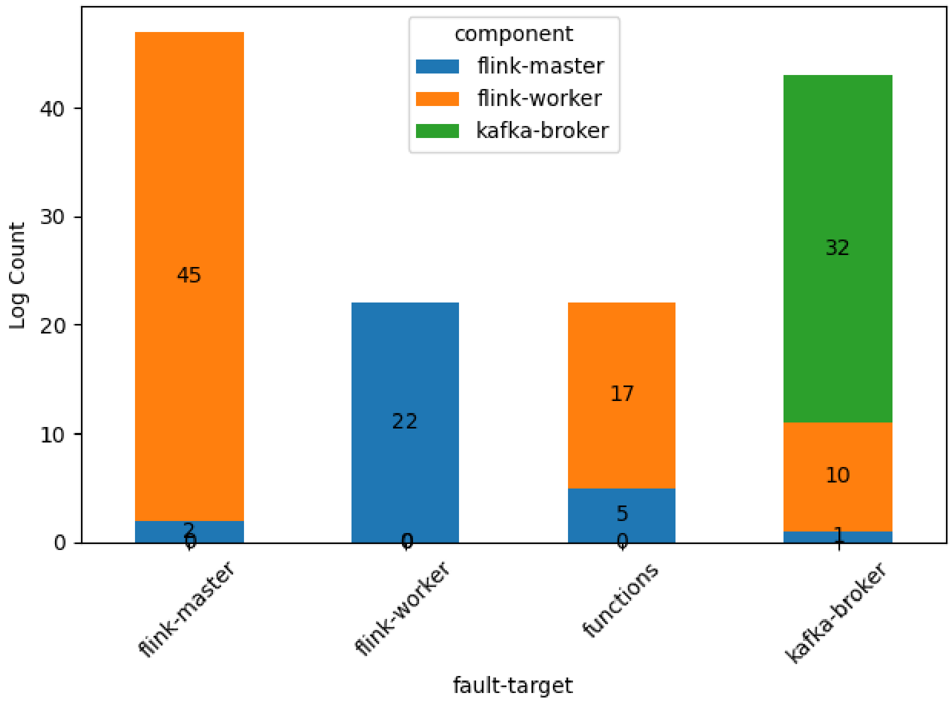

Figure 5.1: Log Count per Fault-Target per Component

Figure 5.1 shows the log counts generated by each component in response to faults in specific targets. Shedding light on inter dependencies and component sensitivity to faults. The x-axis represents fault targets (Flink-master, flink-worker, functions, Kafka-broker), while the y-axis shows the log counts. Each color indicates a different component (flink-master, flink worker, kafka-broker), allowing for a direct comparison of component responses to each fault target.

The data shows that faults in flink-master generate the highest log count in flink-worker (45 logs). and only a small number of logs recorded for flink-master itself (2 logs). This suggests that flink-worker is particularly sensitive to faults in flink-master, indicating a possible dependency. In contrast, flink-worker faults result in significant logging activity in flink-master (22 logs), suggesting that flink-master reacts strongly when flink-worker experiences faults.

For faults in functions, 17 logs are generated in flink-worker and 5 logs in flink-master, showing moderate responses across these components. Meanwhile, kafka-broker faults primarily result in high logging within kafka-broker itself (32 logs), with some additional logs in flink worker (10 logs). This indicates that kafka-broker's response is mostly contained within its own logging, with a lesser effect on flink-worker.

Overall, **Figure 5.1** highlights that flink-worker shows extensive logging when flink-master is faulted, This suggests a dependency, while kafka-broker has a strong internal response to its own faults.

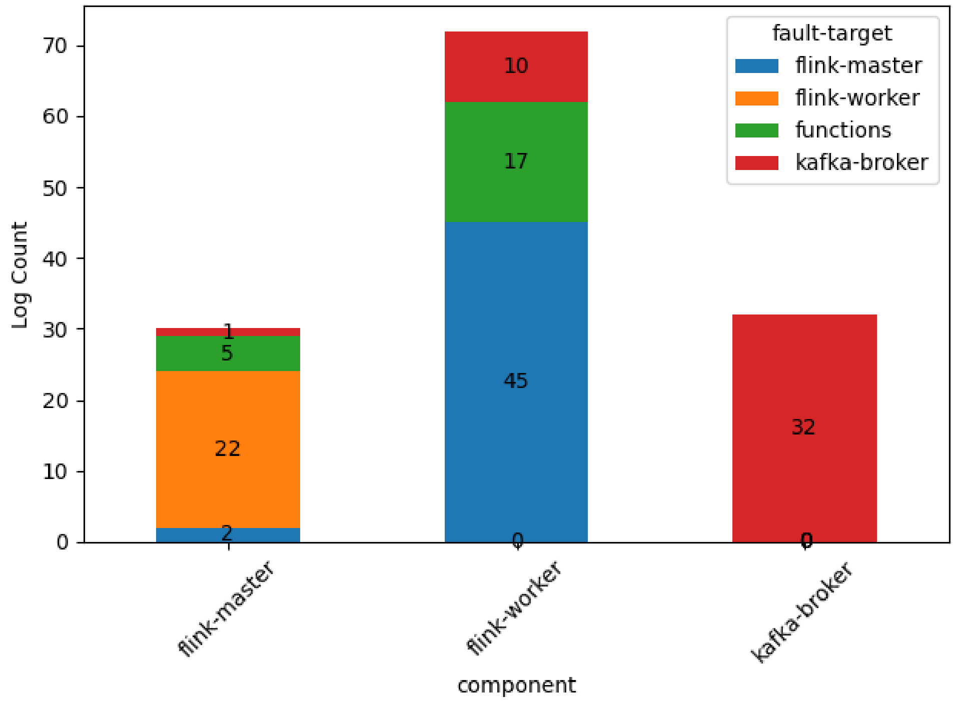

Figure 5.2: Log Count per Component per Fault-Target

Figure 5.2 shows the log counts generated by each component in response to faults across different scenarios. This highlights the behavior and responsiveness of each component under specific fault conditions.

The x-axis represents the components (flink-master, flink-worker, kafka-broker), while the y-axis shows the log counts. Each color represents a different fault target (flink-master, flink worker, functions, kafka-broker), allowing for a direct comparison of each component's re sponse across fault scenarios.

The data shows that flink-worker has the highest overall log count across fault scenarios, particularly in response to faults in flink-master, with 45 logs generated. This indicates that flink-worker is highly sensitive to issues in flink-master. Additionally, flink-worker also logs 17 times in response to functions faults and 10 times for kafka-broker faults, reflecting its responsiveness across multiple fault sources.

flink-master logs significantly when faults occur in flink-worker, with 22 logs recorded. This suggests a dependency or substantial reaction to faults in flink-worker. Smaller log counts are observed for faults in functions (5 logs) and kafka-broker (1 log), indicating a more limited response to these fault scenarios.

Kafka-broker shows most of its logging activity in response to its own faults, generating 32 logs. With minimal logging activity recorded for faults in other components, this indicates that kafka-broker's response is largely self-contained.

Overall, **Figure 5.2** highlights flink worker's extensive logging across fault scenarios. This suggests high responsiveness, while Kafka broker primarily logs in response to its own faults, which indicates relative isolation.

Figures 5.1 and 5.2 reveal key component behavior patterns under fault conditions : Flink-worker consistently shows high logging activity across various fault targets and scenarios. This suggests that it is highly responsive and potentially sensitive to interruptions in other components, particularly flink-master. However Kafka-broker demonstrates a largely self contained response, with most of its logs generated in response to its own faults, indicating relative isolation from other components.

These observations on component interactions and logging behavior provide a foundation for the next section, which will analyze the percentages of enriched attributes across fault scenarios and components to further understand fault-specific behaviors and dependencies within the system.

The next step is to analyze enriched attribute percentages for each fault scenario and component, focusing on component-specific, fault-specific, crash, rare, and direct cause attributes. Examining these percentages allows us to observe characteristic patterns in response to faults and within specific components.

Tables 5.3 and 5.4 provide enriched attribute percentages for each fault scenario and component:

- **total_count**: Total log count after filtering and enrichment.
- **component-specific**: Percentage of logs specific to each fault target or component.
- **fault-specific**: Percentage of logs unique to the fault condition or component.
- **crash**: Percentage of logs related to the crash phase.
- **rare**: Percentage of logs marked as rare.
- **direct_cause**: Percentage of logs identified as having a direct cause.

fault-target	total_count	component-specific	fault-specific	crash	rare	direct_cause
flink-master	47	100.0	72.3	93.6	27.7	4.3
kafka-broker	43	100.0	100.0	11.6	2.3	74.4
flink-worker	22	100.0	72.7	68.2	0.0	0.0
functions	22	100.0	27.3	9.1	0.0	0.0

Table 5.3: Enriched Attribute Percentages per Fault-Target

In **Table 5.3**, flink-master faults show a high component-specific percentage (100%) and a strong presence of crash-related logs (93.6%),substantial fault-specific logs (72.3%) and a notable rare percentage (27.7%). This indicates a strong, unique response pattern when flink master is faulted.

Kafka-broker faults also have 100% component-specific and fault-specific logs, with a high percentage attributed to direct cause (74.4%), suggesting an intense internal response when faults target kafka-broker. Flink-worker shows 72.7% fault-specific logs and moderate crash-related activity (68.2%), displaying a strong but typical response pattern, while functions has lower fault-specific and crash percentages (27.3% and 9.1%), reflecting a limited response.

component	total_count	component-specific	fault-specific	crash	rare	direct_cause
flink-worker	72	100.0	63.9	70.8	15.3	0.0
kafka-broker	32	100.0	100.0	0.0	0.0	100.0
flink-master	30	100.0	70.0	50.0	10.0	6.7

Table 5.4: Enriched Attribute Percentages per Component

In **Table 5.4**, flink-master and flink-worker exhibit high component-specific and fault-specific percentages, indicating broad responses, particularly when faults affect other components. Kafka-broker shows 100% component-specific and fault-specific logs, with a significant portion marked as direct cause, which reflects an isolated, internally focused response pattern. However, functions shows lower fault-specific and crash-related log percentages, indicating reduced sensitivity to faults in other components.

The combined analysis from **Tables 5.3 and 5.4** reveals patterns in component-specific and fault-specific responses, with flink-master and flink-worker displaying extensive responses across scenarios and kafka-broker primarily showing isolated, internally driven reactions. These insights establish a basis for the next section, Anomaly Determination Results, where enriched attributes will be further analyzed to understand patterns in direct and collateral anomalies.

5.3 Results

This section aims to provide a deeper understanding of system behavior by analyzing the flagged anomalies across different dimensions: by fault-target, component, and the interaction between specific components and fault-targets. This evaluation shows how the distributed system responds to faults, revealing dependencies, cascading effects, and potential sensitivities within components.

Since we lack a labeled dataset, we cannot verify with certainty that all anomalies were flagged. However, manual checks across multiple experiments confirmed that all flagged anomalies have the potential to be anomalous. The decision tree and assessment order was refined to produce the current anomaly determination method. This represents the most effective configuration observed. Further progress in machine learning-based anomaly detection could provide greater assurance, but this method still offers helpful details on system behaviors and dependencies.

We will further explore the flagged anomalies. Examining their distributions across fault targets, components, and improved attribute percentages, will reveal essential aspects of the system's responses to faults.

In **Table 5.5** the number of flagged anomalies with each fault target is displayed. This information helps understand different fault scenarios impact the system components.

- **Fault-Target**: Component subjected to fault injection.
- **Anomaly Count**: Total number of flagged anomalies for each fault-target scenario.

The table shows that flink-master has the highest anomaly count with 21 flagged anomalies, followed by kafka-broker with 9 anomalies. Flink-worker and functions have lower counts,

fault-target	anomaly_count
flink-master	21
kafka-broker	9
flink-worker	6
functions	4

Table 5.5: Anomaly Count per Fault-Target

with 6 and 4 flagged anomalies, respectively.

The elevated anomaly count for flink-master when targeted by faults suggests that this component is more sensitive to faults or produces logs more likely to meet anomaly criteria under fault conditions. This result aligns with previous findings from **Section 5.1**. Flink-master generated higher log counts during fault scenarios. This can potentially indicate a higher fault response activity. The moderate anomaly count for kafka-broker also supports the observed pattern of sensitivity, although less significantly.

On the other hand, the lower anomaly counts for flink-worker and functions as fault-targets imply these components may either be less affected by faults or generate fewer conditions that meet anomaly criteria when targeted directly. These findings suggest that while flink-master and kafka-broker are more vulnerable to direct fault impacts. Flink-worker and functions may experience faults indirectly, likely due to dependencies or interactions with other components.

These results point out flink-master's susceptibility to faults. They also explains how different components react when directly targeted by faults. The next section will examine anomaly counts for each component within specific fault-target scenarios. Moreover, It will help better understand dependency patterns and spreading effects in the system.

Table 5.6 presents the total number of flagged anomalies for each component, identifying which components exhibit higher frequencies of flagged anomalies

- **Component**: Specifies each component within the system analyzed for anomalies.
- **Anomaly Count**: Indicates the total number of flagged anomalies associated with each component.

component	anomaly_count
flink-worker	30
flink-master	10

Table 5.6: Anomaly Count per Component

The table reveals that flink-worker has the highest anomaly count with 30 flagged instances, while flink-master shows a lower count of 10 anomalies. Notably, functions and kafka-broker do not appear in this table, indicating no flagged anomalies for these components independently.

The higher anomaly count for flink-worker suggests that it may be more sensitive to faults. It also propose that its role in processing leads to more frequent conditions flagged as anomalies. This is consistent with findings from the fault-target analysis in **Table 5.5**. Flink-master had a higher anomaly count when targeted by faults. But, on a component level, flink-worker comes up as the most commonly involved component with flagged anomalies. This suggests a broader fault sensitivity across different scenarios.

The absence of flagged anomalies for functions and Kafka as stand-alone components may indicate two things : Either greater resilience to faults or a lower likelihood of generating logs meeting anomaly criteria independently. In the fault-target results (**Table 5.5**), Kafka-broker showed flagged anomalies when faults were directly targeted. This indicate reliances or indirect fault impacts that may not emerge under normal operating conditions.

This component-level anomaly count highlights flink-worker′s heightened fault sensitivity. With a more frequent association with flagged anomalies than flink-master. On the flip side, functions and kafka-broker exhibit fewer direct flagged anomalies. This points toward potential resilience or only indirect fault impacts. The following section will investigate the combined effects of specific fault-targets on each component to further understand interactions within the system.

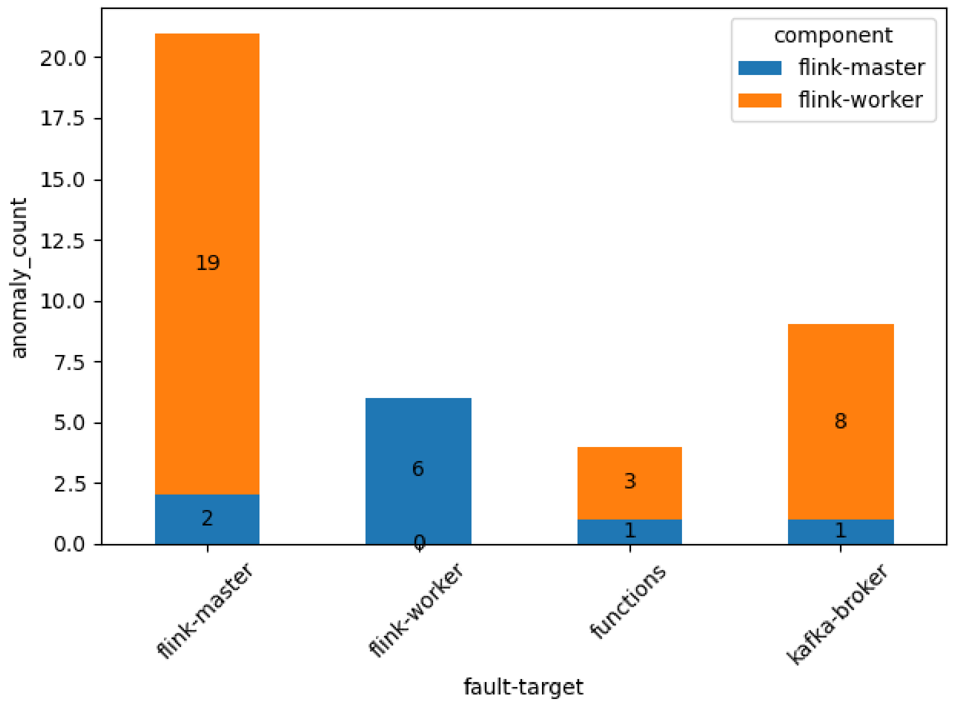

Figure 5.3: Anomaly Count by Fault-Target and Component

Figure 5.3 illustrates the relationship between fault targets and the affected components. This explors fault propagation and progressive effects within the system. Analyzing anomalies according to fault targets and the specific components they impact reveals connections and weak spots.

- **fault-target**: Indicates the origin of each fault scenario.
- **component**: Shows the component where anomalies are recorded.
- **anomaly_count**: Lists the total count of anomalies observed in each component per fault-target.

Faults originating in the flink-master component lead to 19 anomalies in flink-worker and 2 within flink-master itself. Faults in kafka-broker result in 8 anomalies in flink-worker and 1 anomaly within flink-master. Additionally, faults in flink-worker contribute to 6 anomalies in flink-master. Lastly, faults originating in functions lead to fewer anomalies, with 3 occurring in flink-worker.

The data highlights that flink-worker is significantly affected by faults originating in other components This shows high sensitivity to external disruptions. Anomalies in flink-worker due to faults in both flink-master and kafka-broker suggest dependency in the system, where disturbances in certain components propagate and impact others. This recurring impact on the flink-worker shows it as a key point for failure spread, making it important to watch.

In line with previous section results from **Table 5.1** and **Figure 5.2**. Flink-worker's heightened anomaly count across fault scenarios. Flink-worker frequently experience anomalies due to indirect effects from faults in other components. Therefore flink worker serves as a conduit for fault propagation within the system.

These findings show chain reactions and connections, especially in the way flink-worker often face indirect impacts. This shows vulnerability to faults from other components. This shows the need for resilience strategies focused on flink-worker.

Anomaly Determination Categories

In this section, we present three suggested ways to categorize anomalies. Using the decision tree previously illustrated in **Figure 4.2**. The determination decision given to each node in the tree signals an anomaly. It detects forms of behavior depending on how the system reacts to malfunctions.

- **Anomaly-FC (Frequency Change)**: Anomalies flagged due to a significant log burst between phases. This category captures logs with a high frequency change, detected early in the process based on their immediate and unusual increase in occurrence.
- **Anomaly-RC (Rarity Check)**: Anomalies identified at the final decision step. These logs are rare, isolated events occurring in the crash phase. They do not show a high frequency change but are still flagged due to their scarcity and phase association, indicating potential anomalies.
- **Anomaly-FS (Fault-Specific)**: These anomalies have passed all prior checks (including severity, keywords, component-specificity, and fault-specificity). They are flagged at the final fault-specific step. Anomaly-FS instances are highly linked to specific fault scenarios, indicating strong fault-target relevance.

category	total_count	component-specific	fault-specific	crash	rare	direct_cause
Anomaly-FS	26	100.0	100.0	53.8	11.5	7.7
Anomaly-RC	11	100.0	100.0	100.0	100.0	0.0
Anomaly-FC	3	100.0	100.0	100.0	0.0	0.0

Table 5.7: Anomaly Count and Attributes Percentages by Category

From the **Table 5.7** following observations can be made:

- **Anomaly-FS (Fault-Specific)**:
  - We observe that 53.8% of Anomaly-FS entries occur in the crash phase. This means that a substantial portion of these anomalies aligns closely with fault-induced disruptions, even though not all Anomaly-FS entries are crash-phase specific. So even outside the immediate crash phase, faults can produce component specific and fault-specific anomalies that exhibit high severity and include relevant keywords.
  - Notably, Anomaly-FS is the only category to include direct-cause anomalies(7.7%). This emphasizes its role in identifying faults directly impacting the system's components. This presence of direct-cause attributes hints that Anomaly-FS is highly effective in capturing both immediate and indirect fault interactions, as it considers instances where faults are explicitly causing disruptions.
  - The presence of 11.5% rare entries in this category. Some occurrences are infrequent even if they are strongly tied to fault conditions. This adds value to identifying recurring yet unusual fault scenarios that may need further investigation for long-term stability.
- **Anomaly-RC (Rarity Check)**:
  - 100% of Anomaly-RC entries are rare and occur only during the crash phase. This shows that Anomaly-RC effectively captures uncommon but critical events. These events, like frequency spikes, may not have apparent signs, but they are still important because they align with crash conditions. This category reveals rare, high-impact faults that other checks might overlook, particularly in stable components.
  - The fact that 100% of Anomaly-RC entries are rare and occur in the crash phase reinforces that Anomaly-RC is particularly effective at isolating rare, contextually significant events that may not display other anomaly indicators, such as frequency surges, but are still notable due to their specific alignment with crash conditions. This category thus provides insights into infrequent but high-impact fault responses that could be overlooked in other checks, especially in components less prone to frequent fluctuations.
- **Anomaly-FC (Frequency Change)**:
  - Anomalies in this category occur exclusively in component- and fault-specific contexts and are identified based on increased frequency in the crash phase. The presence of high frequency indicates heightened system responses or bursts of activity during fault conditions, capturing the system's direct response to immediate faults.
  - Anomaly-FC flags times when components show chain reactions to faults, focusing on increased activity rather than rare patterns. This helps identify bursts of logs that signal indirect effects or connections triggered by faults.

Through these categories, the decision tree (**Figure 4.2**) highlights distinct aspects of fault responses:

- **Anomaly-FS** captures anomalies with direct and indirect fault impacts, highlighting faults with distinct, recurring impacts even outside the crash phase.
- **Anomaly-RC** isolates anomalies with rare but notable patterns tied exclusively to crash conditions, ideal for identifying unique fault impacts.

- **Anomaly-FC** pinpoints high-activity responses to faults, useful for identifying cascading or amplified reactions during fault events.

The examination indicates that identified irregularities are unique to the component and fault types in systems such as Apache Kafka and Flink.

- **Component-Specific**: These irregularities arise exclusively within a component. Which makes them unique to the behavior and interactions of that component.
- **Fault-Specific**: These anomalies are linked to a fault situation. This shows their importance.

This approach recognize exceptions through fault patterns. It improves our perspective of components' resilience and fault effects.

Both examples in **Section 4.5** are categorized as Anomaly-FS, showcasing how anomalies appear as direct or collateral responses:

- **Example 1** (shown in **Table 4.3**) shows a collateral anomaly in the flink-worker component caused by a fault in the flink-master, illustrating fault impacts on connected components.
- **Example 2** (illustrated in **Table 4.4**) shows a direct-cause anomaly in the flink-master due to its interaction with an unavailable flink-worker. It shows the challenges of linked faults.

The findings reveal that the flink-worker, despite generating fewer logs, accounts for most flagged anomalies, underscoring its critical role in the system. This dependency suggests that many components rely on the flink-worker, underscoring its centrality in overall system operations. This illustrates the challenge of identifying root causes in interconnected systems.

Although the anomaly determination process helps. Further improvements are needed to handle the challenges of fault spread due to system connections. This will be discussed in the concluding section.

6 Conclusion

This study explains a way to find unusual patterns in application logs for systems like Apache Kafka and Flink. We focused on key log details like fault type, component involved, crash phase, and rare events. Using these, we built a decision tree to spot possible issues based on log patterns. We found that connected systems like these have special challenges in spotting and understanding system problems. Based on our findings from the study we conducted, certain elements, even if they are not the point of failure such as the Flink Worker, often play a crucial role in the spread of errors. This shows how some components can make problems worse. It is important to see the difference between direct issues and ripple effects. Figuring out which problems are main faults versus side effects can be tricky, as faults can spread and cause loops that hide the original issue. With testing, we improved the decision tree to spot unusual patterns in the logs. It flagged possible anomalies well, but it had limits because we didn't have labeled data to test it fully. This study shows the need for better ways to find direct and spreading faults, possibly using tools like machine learning.

6.1 Future Work

Following these discoveries, future studies could focus on enhanced methods for detecting irregularities. Using dependency modeling and machine learning techniques, this can enhance issue comprehension, classification and distinguish faults from effects. One approach to adress the observed limitations is creating a labeled dataset for training machine learning algorithms. This could capture the complex behavior of faults and their propagation across components. Supervised models have the potentiel to improve anomaly identification accuracy. They can also contribute to developing predictive fault-detection systems. Incorporating real-time anomaly detection and root cause analysis could also add significant value. These would allow for faster response times and a deeper understanding of fault dynamics in live systems. Also, integrating dependency analysis would help map the interconnected nature of components. This enables anticipating likely paths of fault propagation. Finally, expanding this framework to analyze root causes and feedback effects in more detail could reveal how component interactions can spread issues. This approach would go beyond simply flagging anomalies. It could fully understand system resilience and fault interactions in such environments.

List of Tables

4.1 Sample Log after Preparation 19
4.2 Sample Log after Enrichment 20
4.3 Example 1 of Indirect Anomaly Log during Crash Phase 26
4.4 Example 2 of Direct Anomaly Log during Recovery Phase 26

5.1 Log Count per Fault-Target across Processing Stages 30
5.2 Log Count per Component across Processing Stages 30
5.3 Enriched Attribute Percentages per Fault-Target 33
5.4 Enriched Attribute Percentages per Component 34
5.5 Anomaly Count per Fault-Target 35
5.6 Anomaly Count per Component 35
5.7 Anomaly Count and Attributes Percentages by Category 37

List of Figures

4.1 System Overview 13
4.2 Sequential Decision Tree 21

5.1 Log Count per Fault-Target per Component 31
5.2 Log Count per Component per Fault-Target 32
5.3 Anomaly Count by Fault-Target and Component 36

Bibliography

[1] K. M. M. Thein, “Apache kafka: Next generation distributed messaging system,” *International Journal of Scientific Engineering and Technology Research*, vol. 3, no. 47, pp. 9478–9483, 2014.

[2] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, “Apache flink: Stream and batch processing in a single engine,” *The Bulletin of the Technical Committee on Data Engineering*, vol. 38, no. 4, 2015.

[3] T. Wittkopp, P. Wiesner, D. Scheinert, and O. Kao, “A taxonomy of anomalies in log data,” in *International Conference on Service-Oriented Computing*. Springer, 2021, pp. 153–164.

[4] X. Wei, J. Wang, C.-a. Sun, D. Towey, S. Zhang, W. Zuo, Y. Yu, R. Ruan, and G. Song, “Log-based anomaly detection for distributed systems: State of the art, industry experience, and open issues,” *Journal of Software: Evolution and Process*, 2024.

[5] Y. Yang, Y. Wu, K. Pattabiraman, L. Wang, and Y. Li, “How far have we come in detecting anomalies in distributed systems? an empirical study with a statement-level fault injection method,” in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2020, pp. 59–69.

[6] J. Bogatinovski, S. Nedelkoski, L. Wu, J. Cardoso, and O. Kao, “Failure identification from unstable log data using deep learning,” in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2022, pp. 346–355.

[7] S. Nedelkoski, J. Bogatinovski, A. Acker, J. Cardoso, and O. Kao, “Self-supervised log parsing,” in *Machine Learning and Knowledge Discovery in Databases: Applied Data Science Track: European Conference, ECML PKDD 2020, Ghent, Belgium, September 14–18, 2020, Proceedings, Part IV*. Springer, 2021, pp. 122–138.

[8] J. Bogatinovski and S. Nedelkoski, “Multi-source anomaly detection in distributed it systems,” in *International Conference on Service-Oriented Computing*. Springer, 2020, pp. 201–213.

[9] E. Gelenbe, D. Finkel, and S. K. Tripathi, “Availability of a distributed computer system with failures,” *Acta Informatica*, vol. 23, no. 6, pp. 643–655, 1986.

[10] Redpanda, “Kafka.” [Online]. Available: https://docs.redpanda.com/current/get-started/intro-to-events/

[11] Flink, “Statefun.” [Online]. Available: https://nightlies.apache.org/flink/flink-statefun-docs-stable/

[12] O. Bentaleb, A. S. Belloum, A. Sebaa, and A. El-Maouhab, “Containerization technologies: Taxonomies, applications and challenges,” *The Journal of Supercomputing*, vol. 78, no. 1, pp. 1144–1181, 2022.

[13] J. Watada, A. Roy, R. Kadikar, H. Pham, and B. Xu, “Emerging trends, techniques and open issues of containerization: A review,” *IEEE Access*, vol. 7, pp. 152 443–152 472, 2019.

[14] M. Farshchi, J.-G. Schneider, I. Weber, and J. Grundy, “Experience report: Anomaly detection of cloud application operations using log and cloud metric correlation analysis,” in *2015 IEEE 26th international symposium on software reliability engineering (ISSRE)*. IEEE, 2015, pp. 24–34.

[15] H. Zhou, W. Qian, X. Zhou, Q. Dong, A. Zhou, and W. Tan, “Scalable and adaptive log manager in distributed systems,” *Frontiers of Computer Science*, vol. 17, no. 2, p. 172205, 2023.

[16] Q. Cheng, A. Saha, W. Yang, C. Liu, D. Sahoo, and S. Hoi, “Logai: A library for log analytics and intelligence,” *arXiv preprint arXiv:2301.13415*, 2023.

[17] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, “Drain: An online log parsing approach with fixed depth tree,” in *2017 IEEE international conference on web services (ICWS)*. IEEE, 2017, pp. 33–40.

[18] C. Pham, L. Wang, B. C. Tak, S. Baset, C. Tang, Z. Kalbarczyk, and R. K. Iyer, “Failure diagnosis for distributed systems using targeted fault injection,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 2, pp. 503–516, 2016.

[19] S. Dawson and F. Jahanian, “Deterministic fault injection of distributed systems,” in *Theory and Practice in Distributed Systems: International Workshop Dagstuhl Castle, Germany, September 5–9, 1994 Selected Papers*. Springer, 2005, pp. 178–196.

[20] X. Fu, R. Ren, S. A. McKee, J. Zhan, and N. Sun, “Digging deeper into cluster system logs for failure prediction and root cause diagnosis,” in *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2014, pp. 103–112.

[21] M. Du, F. Li, G. Zheng, and V. Srikumar, “Deeplog: Anomaly detection and diagnosis from system logs through deep learning,” in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 1285–1298.

[22] T. Wittkopp, D. Scheinert, P. Wiesner, A. Acker, and O. Kao, “Pull: Reactive log anomaly detection based on iterative pu learning,” *arXiv preprint arXiv:2301.10681*, 2023.

[23] H. Ott, J. Bogatinovski, A. Acker, S. Nedelkoski, and O. Kao, “Robust and transferable anomaly detection in log data using pre-trained language models,” in *2021 IEEE/ACM international workshop on cloud intelligence (CloudIntelligence)*. IEEE, 2021, pp. 19–24.