Self Adjusting Log Observability for Cloud Native Applications

Divya Pathak*
Indian Institute of Technology Hyderabad
cs21mtech12009@iith.ac.in

Mudit Verma*, Aishwariya Chakraborty, Harshit Kumar
IBM Research, India
mudiverm@in.ibm.com, aishwariya.chakraborty1@ibm.com, harshitk@in.ibm.com

***Abstract*—With the increasing complexity of modern applications, particularly those relying on microservices architectures, the volume of observability data, encompassing logs, metrics, traces, etc., has surged significantly. This is further exacerbated by extensive cloud deployments, where observability is crucial for comprehending the health and performance of these systems, leading operations teams to collect as much data as possible for the "fear of missing out". However, the collection, storage, and analysis of observability data entail significant costs, both in terms of resources and finances. Specifically, logs comprise the most substantial portion of observability data volume, thus exerting the greatest impact on observability cost. Moreover, logs also exhibit unstructured and noisy characteristics, where the efficacy of downstream AI for IT operations (AIOps tasks or Day-2 operations), such as fault classification, fault diagnosis, log anomaly detection etc., can be negatively impacted by log data volume. Hence, striking a balance between the verbosity of log observability and its impact on day-2 operations and debuggability is essential. In this paper, we introduce an autonomous system named SALO, which stands for Self-Adjusting Log Observability. SALO selectively collects logs based on real-time necessity, location, and granularity, as opposed to the conventional practice of collecting indiscriminately from all the components continuously. Our experiments show that SALO drastically decreases the log volume, by as much as 95%, while still maintaining data quality for downstream AIOps usage, especially for post-hoc diagnosis tasks. Operating on a reduced volume of log data not only decreases storage, transfer, and retention costs but also streamlines observability pipelines, making them leaner, more efficient, and less resource-hungry.**

***Index Terms*—Observability, Log Volume, AIOps**

I. Introduction

As the cloud-native applications continue to grow in complexity and size, effective observability [1]–[3] becomes paramount for managing the application's performance and health in real-time. Despite all precautionary measures, such as through end-to-end testing and in-built reliability, faults are inevitable in these complex applications.

Organizations invest significant resources, amounting to as much as 30% of overall cloud infrastructure cost for deploying observability pipelines[1]. Typically, a closed-loop observability pipeline consists of several tasks [4]–[6], such as collection, filtering, aggregation, pattern mining and learning, prediction and resolution retrieval etc. However, the ever-expanding volume of observability data poses significant challenges.

- **Storage and retention** of observability data entails substantial expenses. Additionally, in large-scale distributed applications with advanced observability pipelines, the data is typically moved from local or edge sites to remote or central locations for storage and analysis, consuming a significant portion of **WAN bandwidth**.
- Certain types of observability data, particularly logs, exhibit a direct correlation with the application's workload. As the workload increases, so does the volume of logs generated, and this **interferes with application's performance and infrastructure resources** [2].
- The escalated data volume places significant strain on closed-loop autonomous observability pipelines, necessitating additional resources and resulting in prolonged **mean time to detect** (MTTD) and **mean time to resolve** (MTTR) . AIOps models for downstream tasks such as root cause analysis, failure prediction, outage analysis, and fault classification are susceptible to the problem of **"garbage in, garbage out**", wherein the presence of noisy data diminishes their effectiveness [7].

To overcome the aforementioned challenges, large-scale applications employ various strategies. In one classical approach, particularly prevalent in High Performance Computing (HPC) and performance-critical applications [8]–[10], logging is either disabled or maintained at minimal levels in production environments. This cautious approach is due to the significant cost associated with even slight interference with application performance or infrastructure resources caused by logging. However, with limited logging data, when an issue is raised, debugging is often activated to enable the collection of log data at a higher granularity, or faults are reproduced in test environments to collect necessary data for issue diagnosis. Both of these methods are highly manual, cumbersome, and may fail to precisely replicate the context of the production environment. Also, this approach is reactive in nature, as data is collected only after a fault event [11], potentially missing crucial data leading up to the fault. It not only hampers the ability to proactively address issues, but also may result in incomplete or insufficient information for further analysis.

At the opposite end of the spectrum, autonomous observability pipelines represent a paradigm where data acquisition and analysis occur in real-time. This approach is extensively

*Equal contributors.

[1]https://devops.com/observability-costs-are-too-damn-high

[2]https://outshift.cisco.com/blog/logging-impact-on-application-performance

implemented in large-scale modern microservices-based applications [12] and infrastructures, particularly for their self-healing capabilities. Advanced AIOps models [13]–[15] are employed for various proactive and reactive downstream tasks. These models scrutinize incoming observability log data in real-time, aiming for expedited MTTD and MTTR. However, this methodology frequently results in indiscriminate log data collection, where all available logs are gathered continuously, regardless of their immediate relevance or significance. Consequently, this approach incurs substantial observability costs, as the storage and analysis of vast amounts of data strain resources and infrastructure, potentially outweighing the benefits of real-time analysis [16].

In this paper, we propose an autonomous system SALO, which stands for self-adjusting log observability, that facilitates selective and strategic collection of observability log data in real-time guided by the following questions:

When to collect*?* The system should autonomously determines the optimal timing for gathering log data at finer granularity based on specific intelligence criteria.

From where to collect? The system should selectively collect log data from specific components of an application being observed, recognizing that not all components hold equal importance in a given context.

At what granularity? The system should accurately discern the pertinent log levels (verbosity) for individual components.

To the best of our knowledge, there is currently no log collection system like SALO that enables dynamic adjustment of log granularity with automated intelligence, without human intervention. Furthermore, SALO maintains high responsiveness to support real-time data collection and utilization. Additionally, it is imperative that SALO gathers both adequate and essential data to prevent any adverse effects on downstream tasks resulting from reduced log collection. In summary, the key contributions of this work are:

1) Introduction of a system SALO capable of dynamically adjusting log collection granularities within observability pipelines, along with the development of a prototype for self-adjusting log observability, featuring a lightweight health detector and a central controller.
2) Creation of 3 datasets for evaluating SALO, using two benchmarking microservices-based applications of varying complexity. These datasets incorporate log data containing injected atomic and cascading faults of both transient and persistent nature, within a controlled operational setting of the application, serving as a foundational benchmark for future research.
3) SALO's effectiveness was demonstrated through extensive experiments, showing reduction of up to 95% in log volume and outperforming critical downstream tasks by up to 20% with minimal resource overhead.

II. Background and Related Works

Observability refers to inferring a system's state based on its telemetry data [1]. An effectively observed system offers deep visibility into its operations, facilitating critical insights into performance and resources [17]. In microservices-based applications, the growing number of components further accentuate the importance of observability [18]. While the collection of telemetry data falls under the scope of monitoring, the analysis of the relevant data to gather useful information is strictly part of observability [19]. This renders the existing monitoring solutions insufficient for ensuring observability. Hence, over the years, the domain of observability has received considerable attention, from academia and industry alike. In Cloud deployments, typically the telemetry data is sourced from local sites and brought to a central location for further analysis [2]. Researchers and practitioners proposed several frameworks and protocols for data migration to a central location. For example, Marie-Magdelaine *et al.* [20] demonstrated the implementation of such a framework for a cloud-native application built on AWS. IntelligentMonitor [21] and 5GC-Observer [22] are few other examples of such frameworks which employ real time data collection and intelligent analytics to ensure observability. Additionally, tools such as Prometheus, Jaegar, Zipkin, Grafana, and Thanos[3] are widely used for observability pipelines.

Observability data has four modalities - Metrics, Events, Logs, and Traces, often referred to as *MELT* [23]. These modalities vary significantly in terms of storage, computation, and communication requirements. Among these, logs are of utmost importance since it provide deeper information with rich context [23]. Logs are easy to instrument as all programming languages provide numerous logging libraries. However, there are several challenges associated with logs: (a) the volume of logs generated by a system can be massive, leading to complexities in storage and processing, and (b) the unstructured nature of the logs makes it challenging to extract meaningful information. To mitigate these issues and to reap the benefits of logs, researchers have proposed several approaches such as sampling of useful logs [24], parsing of unstructured logs into structured format [25]–[27], and compression of logs based on commonality and variability [28]–[30]. Some of these approaches are heavily dependent on the downstream AIOps tasks such as anomaly detection [31]–[34].

As applications grow, manual debugging and AIOps analysis sift through large data volumes for relevant information, akin to finding a 'needle in a haystack'. To meet the SLOs of majority of modern applications, closed-loop fault management involving data collection, processing, fault detection, and remediation, needs to be done in real-time and with minimal human intervention. Trained AIOps models are deployed to accomplish the aforementioned tasks, however, they consume significant resources and deliver sub-optimal results when fed with large amounts of unnecessary data [35]. Hence, it is not only costly but also harmful to aimlessly collect data about anything and everything under the sun in the name of observability. This necessitates the design of low-

[3]https://prometheus.io, https://www.jaegertracing.io, https://zipkin.io, https://grafana.com, https://thanos.io

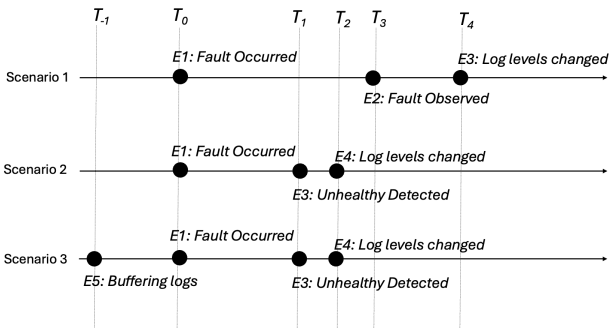

Fig. 1: Event timelines and scenarios

cost, intelligent, and adaptive logging schemes for modern cloud native applications. Therefore, this paper proposes an autonomous system that collects log data only when it is required, from where it is required and at the right granularity, while still maintaining the data quality for downstream tasks.

III. System Design

This section addresses the questions posed in Section I and outlines the algorithms and approaches utilized in designing the proposed system SALO.

A. *When to collect?*

As can be recalled, in many classical performance-sensitive applications, logging is generally turned off or kept at a minimal level until an issue is observed, at which point operators manually enable or increase logging levels. This creates a temporal gap, resulting in the loss of significant information pertaining to the moment of the issue's actual occurrence and the subsequent manual elevation of log levels, as shown in Scenario 1 of Fig 1. *E1:Fault Occurred* at time *T0* followed by *E2:Fault observed* at time *T3*. However, only at time *T4*, the logging levels were raised. Further, this approach is not applicable to modern closed-loop real-time observability pipelines for downstream day-2 AIOps models. These models are designed to process live streaming logs in real-time. Consequently, log data is collected at the highest granularity in these systems all the time, exacerbating the problem of data overload.

Hence, the critical question revolves around identifying the optimal timing for initiating adjustments in log levels. SALO addresses this by designing an intelligence-based health detection technique, incorporated in a lightweight sidecar, called GateKeeper, attached to each component of an application. The GateKeeper consists of a health detector and a filtering agent. This system filters logs based on the component's health status, allowing minimal or no logs through when the component is deemed healthy. Conversely, if the component is identified as unhealthy, higher log levels are permitted through the filtering mechanism. This can be implemented in different ways, with two primary objectives:

- ***Objective 1***: The health detector should be lightweight, responsive, and must not introduce latency into the pipeline.
- ***Objective 2***: The intention behind the health detector is not to replace downstream comprehensive log analysis via manual debugging or the utilization of AIOps models. Rather, its role is to offer an early indication of a potential issue with the component. Therefore, false positives are acceptable, but false negatives are not.

As illustrated in Scenario 2 of Fig. 1), the health detector detects a component of the application as unhealthy at time *T1* (Event $E1$), triggering log levels changes to higher granularity at time *T2* (Event *E3*), thereby relaxing the filter to allow more logs to pass through. For SALO, the Event $E4$ in scenario 2 executes automatically, unlike Event $E3$ in Scenario 1 that requires manual intervention, thereby reducing the risk of losing important events in logs.

One obvious approach to implement health detection is to identify any ERROR level log messages, interpret them as potential health issues for the corresponding component. However, it might result in false negatives as the issue may not be present in ERROR message alone. The health detection in SALO uses a more sophisticated technique, that involves training a lightweight model on the healthy log data of the component, detecting any deviations from normal behavior as potential health issues. Health detection could also integrate additional inputs from the system, such as extracting health endpoints from the component or leveraging data from other sources such as metrics or traces. However, we have not explored this aspect in our current work, and it will be considered as a topic for future research.

B. *Where to collect from?*

Having established the scenarios under which log data collection is warranted, the subsequent task involves pinpointing the specific components within the application from which data should be gathered.

In a complex distributed application, faults in one component can propagate to other components, leading to cascading failures. Research on major outages from the past decade suggests that, on an average, these cascades involve around four components of an application [36]. This phenomenon is commonly referred to as the "Blast Radius" [37] in the AIOps literature, denoting the extent to which a fault can inflict damage to other components of the application.

From a logging perspective, it's crucial not only to elevate log levels for the unhealthy component but also for its neighboring components as a precautionary measure. This ensures that downstream tasks also receive relevant logs immediately from potentially impacted components in the event of cascading faults. There are two ways in which it can be addressed.

- **Static**: A predefined policy for an application can dictate how data collection granularity should increase log levels based on the originating fault component. This can involve having a predefined mapping of a component to potentially impacted neighboring components. Alternatively, static configuration policies can be designed based on the application's topology, which can act as a guidance

to adjust data collection strategies around component interactions.

- **Dynamic**: This approach utilizes real-time insights into the application's interaction topology. It is a dynamic approach that can determine the blast radius of potentially affected components during runtime. This dynamic assessment can be supplemented with static configurations policies and rules, such as limiting involvement to nearby neighbors up to k hops.

Our system is designed to configure both the static policy as well as dynamic policy, along with the interaction topology, to determine the blast radius for identifying the effected components.

C. *At what granularity?*

Following from the above discussion on blast radius calculation, the fault in a component may also impact a subset of other components in the application. This subset is determined based on the application topology and interaction among various components. Specifically, an application is said to have a blast radius of k if a fault in any of its components impacts other components within a maximum distance of k hops in the application topology. Here, it is noteworthy that not all of these other components are impacted to the same degree. For example, the components that are in closer proximity to or have frequent interactions with the faulty component are more prone to have an early onset of faulty behavior as opposed to other components within the application.

Thus, from the viewpoint of fault localization, an undesired behavior in a component can result from a fault occurring either within the component or from any of its neighbouring components. Hence, it is imperative to elevate log levels, not only from the component exhibiting the faulty behavior, but also from its neighboring components within a given blast radius. Additionally, to reduce the volume of unwanted log data, it is imperative to determine the log collection granularity of the neighboring components while considering the degree of impact caused by cascading faults.

To determine the suitable log level for neighbouring components during a component's health deterioration event, SALO employs the fan-out edge weights to estimate the impact on neighboring components. SALO considers following two approaches to calculate the degree of impact on neighboring components:

- **Static Equal Weightage:** Each fan-out edge is assigned an equal weightage. For instance, if an impacted component has 3 neighbors in the topology, the probability of each of them being affected in a cascade of faults is the same. This process is applied for up to k hops within the blast radius.
- **Dynamic Interaction-based Weightage:** Fan-out edges are assigned weights proportional to the real-time interaction ratio between fan-out components. This approach is based on the observation that the likelihood of a component being affected is directly proportional to the interaction it has with the faulty component, compared to another component with comparatively lower interaction.

TABLE I: Log-level – Edge weight spectrum map

	Log Level	Edge Weight
L1	CRITICAL	$0-20$
L2	ERROR	$20-40$
L3	WARN	$40-60$
L4	INFO	$60-80$
L5	DEBUG	$80-100$

Once a fault and the potentially impacted components are identified, the log-level of the impacted components are updated based on the corresponding edge weights. Towards this aim, SALO analyzes the spectrum of edge weights and maps them to the different log-levels, an example of such a mapping is shown in Table I. For this work, the maximum weight that an edge can have is 100. Please note that, there are 5 log levels — DEBUG (L5), INFO (L4), WARN(L3), ERROR(L2), and CRITICAL(L1), in the increasing order of severity. That is, if the log level of a component is set to WARN(L3), then the component emits logs at WARN, ERROR, and CRITICAL levels, i.e., L3 and below. SALO collects logs at L1 level by default, unless an unhealthy microservice is detected by the GateKeeper. The detailed algorithm for log level determination is outlined in Algorithm 1.

Algorithm 1 Log Level Determination

Input
  K Blast Radius
  M Faulty Component
  $\mathcal{L}$ Log levels of all components
  $\texttt{Map}(L, V)$ Log-level – Edge weight spectrum map
Output
  $\mathcal{L}^n$ New log levels of all components
Find $\mathcal{S}_M$ using Depth First Search algorithm
  ▷ $\mathcal{S}_M$: Upstream components $\leq K$ hops from M
Sort $\mathcal{S}_M$ in increasing order of K
for $s \in \mathcal{S}_M$ **do**
  for upstream component s' of s where $s' \in \mathcal{S}_M$ **do**
    $V_s^n \leftarrow V_s + \frac{w_{s's}}{\sum_{s''} w_{s's''}}$
      ▷ V_s, V_s^n: old and new values of component s
      ▷ s'': upstream component of s'
      ▷ w: edge weight
Obtain $\mathcal{L}^n$ corresponding to V^n from $\texttt{Map}(V, L)$
Return $\mathcal{L}^n$

D. *Architecture*

Figure 2 illustrates the architecture of SALO that consists of two building blocks: **GateKeeper** has a local view, with one gatekeeper instance for each component, primarily responsible for detecting the health of the component to determine the appropriate log level for log filtering; **Central Controller** has a real-time global view of the application, discovers unhealthy component by probing, thereafter compute blast radius to

identify other affected components, and instruct them to raise their log levels accordingly. The following section provide details of each component.

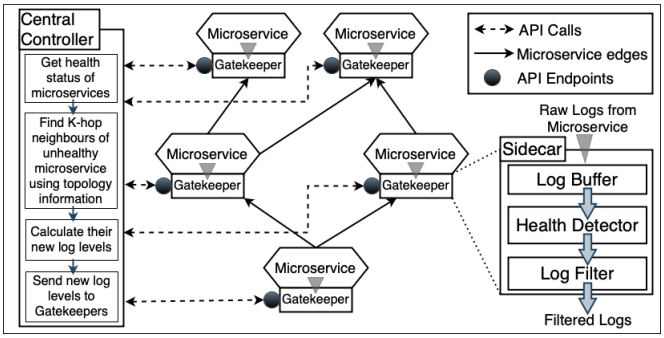

Fig. 2: SALO Architecture

*1) **GateKeeper**:* GateKeeper is a lightweight sidecar[4] that attaches to every component within the application. It intercepts all logs generated by the component and serves three core functions: *Health Detection*, *Log Filtering*, and *Buffering*.

Health Detector module, illustrated in Fig. 3, is designed to identify the health of the component based on the log emanating from it. It uses Drain3 [38], a log template miner, that mines log data to learn a log template model [39]–[41] from healthy logs. Typically, the number of templates are very few in numbers compared to the total volume of logs; this is because log lines repeat frequently, with only the variable values changing while the static strings remain constant, thus grouping them into a single template with a template-id assigned to it. Below is an example of a template along with two matching log lines:

```
template: --TIMESTAMP-- INFO User --NAME-- is
successfully authenticated on VM --NAME--

2024-01-01 00:00:00 INFO User Bob is successfully
authenticated on VM vm1.domain.com

2024-02-02 11:10:20 INFO User Root is successfully
authenticated on VM vm10.domain.com
```

During an offline phase, a Template Model is learned using healthy logs specific to each component. This model comprehends all patterns of healthy logs for the component. In the online phase, a Drain template matcher is deployed with the learned Template Model to process incoming logs from the component, generating a stream of template IDs. Any newly detected template, not previously learned, is assigned a special ID (e.g., -1). These templates pass through a temporal windowing component with a configurable duration. The sequence of windowed templates is then analyzed to identify presence of unknown template id (-1). Presence of at-least one -1 template id in a window of log template-ids would label it as unhealthy and vice versa. Based on this analysis, an application component is determined to be Healthy or Unhealthy.

Log Filter module within the GateKeeper intercepts logs from its associated component, allowing only the intended log levels to pass through. Log filter maintains a record of the current log level of the associated component. There are two ways in which the log levels can be altered:

a) When the Health Detector identifies a component as unhealthy, the Log Filter adjusts the log level to L5, allowing all logs to pass through. Conversely, when the component is healthy, only logs at L1 level (or the configured level) are permitted.

b) The Log Filter may also receive instructions from the Central Controller to increase or decrease the log level. This occurs, if the associated component is potentially affected by its neighboring faulty components as determined by the Central Controller. If the requested log level is lower than the current level, the request is ignored. For instance, a component is already unhealthy and the current log level is L5. However, there is also a fault in its neighbouring component, which causes Central controller to instruct this component to raise its log level to L3, the new request is disregarded because the component's log collection is already at a higher level.

It's worth highlighting that adjusting log levels at runtime can be accomplished using the functionality built into the logging library, provided it offers such capabilities. While Python3's logging library [5] supports this feature, it's not commonly available in many other logging libraries across various programming languages.

Buffering: In the event that a component is identified as being in an unhealthy state, it is typically imperative to analyse the logs produced before the fault occurrence, either for conducting root cause analysis or for addressing other downstream tasks. Therefore, to ensure the availability of such important logs leading up to unhealhty state of the component, *GateKeeper* employs a buffering mechanism. It stores logs at maximum log level, defined by the configured duration, in a temporary circular temporal buffer. As shown in Scenario 3 of Fig. 1, while $E1$ Fault occurred at time $T0$, SALO would have had buffered logs at L5 level from an earlier time T_{-1}. SALO first flushes buffered logs from time T_{-1} to time T_2, before allowing through the logs at higher level from time T_2 (E4: log levels changed), thereby ensuring that important logs are not missed out before and after the faulty event E1.

*2) **Central Controller**:* The Central Controller is a shared module that periodically polls all *GateKeeper* instances, associated with each component, to assess the component's health status and log levels. Upon detecting that a particular component is unhealthy, it calculates the blast radius based on the application's interaction-based topology and determines the appropriate log levels for potentially impacted neighboring components. Subsequently, it communicates the elevated log levels to the attached *GateKeeper* of the potentially affected components. The process for identifying potentially affected neighbors and their corresponding log levels is detailed in Algorithm 1. Once the fault is rectified, the controller is also

[4]https://learn.microsoft.com/en-us/azure/architecture/patterns/sidecar

[5]https://docs.python.org/3/howto/logging.html

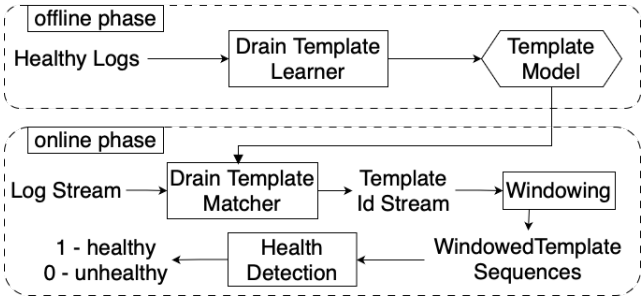

Fig. 3: Health Detector Module

responsible for restoring the log levels of affected components to their default log levels.

IV. Implementation

A prototype of the proposed system, SALO, is implemented for container orchestration platforms like Kubernetes or OpenShift and emphasis was placed on aligning with cloud-native design principles, with a focus on intent-driven functionality and performance prioritization.

GateKeeper is deployed as a sidecar container, co-located with the primary container of the application component (referred to as a microservice hereafter) within the same pod. Implemented in Python 3, GateKeeper hosts a REST server that offers endpoints for adjusting the log level and retrieving the corresponding microservice log level and health status. The Central Controller is also deployed as a container pod in a separate namespace. Also written in Python 3, it can be replicated as needed to accommodate scalability demands. GateKeeper encompasses 297 LOC, while the Central Controller comprises 272 LOC.

For the brevity of space, not all the details are provided, however, the following are the key implementation details:

1) GateKeeper processes are encompassed in a docker image, deployed alongside the microservices, either during initial deployment or dynamically via ISTIO service mesh [6] at runtime.
2) When a microservice generates logs on stdout or stderr, the corresonding sidecar container intercepts them by accessing the Kubernetes node's filesystem where the microservice is hosted. If the logs are stored within the container's file system, GateKeeper is configured to read from those files.
3) The log filter of GateKeeper outputs filtered logs on its stdout or stderr, which are then stored by the Container Orchestration platform on the file system of the node where the microservice pod is hosted. Moreover, the ELK[7] (Elasticsearch, Beat, Logstash and Kibana) logging stack is configured to automatically scrape microservice's filtered logs from the sidecar container rather than the main microservice container. Consequently, any downstream task retrieves filtered logs from SALO through Elasticsearch.
4) GateKeeper provides knobs to configure Drain matcher window duration, buffered log duration and default log filter level when the microservice is healthy. These parameters can be adjusted to strike a balance between performance and latency.
5) GateKeeper maintains records of the *current log level* and *told log level* for every microservice. The *current log level* denotes the present verbosity level of filtered logs, while the *told log level* represents the log level directed by the central controller.
6) The central controller offers an API endpoint for GateKeepers to register themselves, serving as the primary mechanism to identify all existing microservices. This information is centrally stored by the controller and utilized for communication with the GateKeepers.
7) The Central Controller periodically queries the endpoints provided by the GateKeepers in a round-robin fashion to ascertain the health status of microservices.
8) The central controller interfaces with Kiali[8] to obtain real-time interaction topology of the application.
9) Fig. 4 shows the implemented interaction flow between the GateKeepers and Central Controller.
10) Fig. 5 shows a working example of *GET /fetch* and *PUT /update* API endpoint calls between the Central controller and the GateKeeper.

V. Experiment

In this section, we assess SALO in terms of: a) its effectiveness in reducing log volume, b) its impact on downstream tasks, and c) the overheads associated with its implementation. To ground the aforementioned goals, we define the following research questions.

- ***Q1:*** To what degree does the decrease in log volume achieved through SALO impact day-2 AIOps tasks?
- ***Q2:*** Can SALO effectively detect the cascading fault sequences and associated microservices?
- ***Q3:*** What extent of log volume reduction can be achieved by SALO?
- ***Q4:*** What are the additional resource overheads associated with SALO?

The remainder of this section is structured as follows: We begin by discussing the environment setup, including details about middlewares and the selected applications used for this study. Next, we elaborate on the preparation of the dataset, including the injection of faults. Finally, we evaluate SALO across several dimensions and provide analysis of results.

A. Setup

All experiments were conducted on a cloud-hosted 6-node OpenShift cluster [42], which serves as an enterprise Kubernetes container orchestration platform. Each node in the cluster was equipped with 16 cores and 32 GB of RAM. This

[6]https://istio.io/latest/about/service-mesh/
[7]https://www.elastic.co/elastic-stack
[8]https://kiali.io/

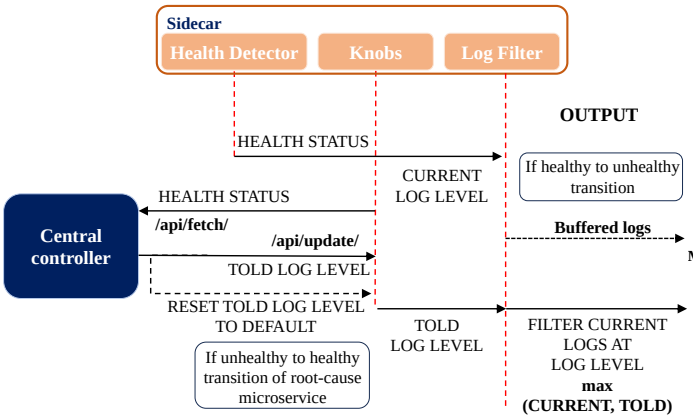

Fig. 4: Sequence of interaction events between GateKeeper and Central Controller

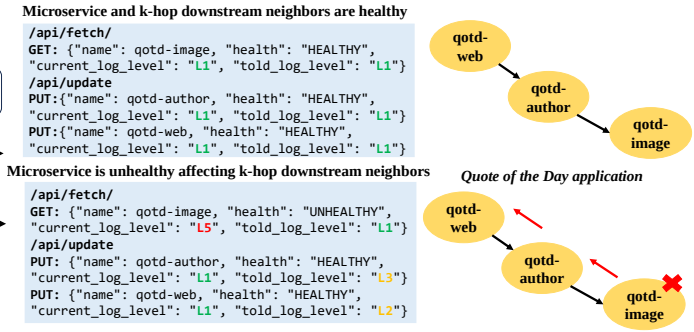

Fig. 5: API calls between GateKeeper and Central Controller for QoTD application

dedicated cluster was exclusively utilized for the experiments, with no other software or applications co-hosted on it, except for essential supporting tooling and middleware.

Applications: To prepare the dataset for the task of log volume reduction, we selected two applications based on levels of complexity in terms of number of microservices: Quote of the Day (QoTD)[9] and Train Ticket (*TT*)[10] [43]. QoTD is a small-scale application comprising 8 microservices and a dedicated database. *TT*, on the other hand, is a medium-scale application consisting of 41 microservices and multiple databases. Both applications utilize REST calls for communication between microservices.

B. Dataset Preparation

To evaluate SALO, a substantial-duration log dataset is desired, documenting instances where the application experienced diverse fault conditions, evident through non-healthy or faulty log entries. Logs were collected by inducing a variety of faults, while SALO is deployed under different configurations. One of the key contributions of this paper is to build a log dataset that contains unhealthy log lines, injected randomly across one or more components of the application. We expect that this dataset will serve as a benchmark for log volume reduction task, addressed in this paper, and other day-2 tasks in AIOps such as fault localization, incident detection, incident prioritization [44], etc.

Faults: In a large scale complex application, failures are inevitable. These failures, termed *faults*, can affect either a single microservice or cascade through interconnected microservices. For this paper, faults are broadly classified into two types:

- **Isolated Atomic** faults arise from malfunctioning within a single microservice, exclusively affecting the particular service.
- **Cascading** fault originates from a single microservice failure but propagates to other microservices through a chain reaction, impacting a set of dependent microservices and triggering a cascade of faults.

Faults can also be further sub-classified based on their duration of impact.

- **Persistent** faults persist in the system for an extended period until resolved, such as a continuous memory leak leading to system instability.
- **Transient** faults, conversely, are sporadic and temporary, occurring briefly under specific conditions, like intermittent network timeouts causing occasional service disruptions.

Fault Injection: We incorporated varieties of the aforementioned fault types with different permutations and combinations into both applications. We ensured that for atomic faults, the fault would manifest through the presence of erroneous or unhealthy log lines in the affected microservice, and for cascading faults, in multiple components as dictated by the application's topology. Transient faults were introduced randomly, lasting between 500 milliseconds to 1 second, with corresponding fault log lines appearing in affected microservices at 100 ms intervals. Whereas, the duration of persistent faults varied randomly between 1 to 5 minutes, with fault log entries occurring at 100 ms intervals. To generate log data for cascading faults, after injecting faults into the originating component, cascades were observed in subsequent components, following a randomly determined duration ranging from 1 to 5 seconds, and so forth. For example, a log line indicative of an atomic fault injected into a component of the Train Ticket application might appear as follows:

```
[{'service': 'ts-consign-service','log': '[Consign
Service] Error processing consignment request for
consignment ID XXX'}]
```

An illustration of a cascading fault affecting two components of the Train Ticket application would show the following unhealthy log lines in the respective component:

[9]https://gitlab.com/quote-of-the-day/quote-of-the-day
[10]https://github.com/FudanSELab/train-ticket

```
[{"service": "ts-basic-service", "log": "[Basic
Service] Unexpected error occurred for basic
operation with ID XXX"},
{"service": "ts-price-service", "log": "[Price
Service] Price calculation error for item ID XXX"}]
```


Load Generation. Both applications were subjected to a simulated workload using the specialized load generation tool integrated within the framework of each respective application. The workload was designed to mimic real-world scenarios, that varied in terms of user interactions, such as requests for data retrieval, processing, and updates, as well as the frequency and intensity of these interactions. The workload encompassed a diverse range of operations, including read and write, data querying, and computations, to comprehensively cover log generation under different usage patterns.

Edge-weights determination: We derive weights for microservices interaction using two approaches: *static equal edge weightage* and *dynamic interaction-based edge weightage*, as discussed in Section III. In the interaction-based method, we utilize Kiali to generate a weighted topology. In the event, where there is no interaction between any two microservices, an edge weight of 0 is assigned. It is important to highlight that in the equal edge weightage approach, all edges originating from a microservice are assigned the same weight. Please note that for the QoTD application, the maximum hop (k) considered in a blast radius is 2. And, for the *TT*, same is 3.

Dataset: Given, QoTD is a small application, we only injected atomic faults. Whereas, to create a more complex dataset we injected both atomic and cascading faults into the *TT*. For the QoTD, we gathered a total of 6 hours worth of logs. Throughout this period, we randomly introduced a set of 7 distinct faults, repeated across multiple microservices, resulting in a total of 14 combinations. The log dataset comprises a mixture of healthy and faulty logs, amounting to $\sim 260K$ log lines. For the *TT*, we collected the log data for 52 hours, and we introduced a set of 26 distinct faults, resulting in a total of 208 combinations with cascading faults impacting 1 to 8 microservices in the chain. The dataset obtained from these injections consists of $3.7M$ and $3.5M$ log lines for persistent and transient faults, respectively, named as TT-Persistent and TT-Transient.

C. Results

In this section, we list the different variants of SALO for comparison with the baseline method.

- **Baseline: (A)** This is the default scenario where logs are collected continuously and comprehensively from all microservices, capturing data at the highest granularity level (L5).
- **Collect From All When One Is Faulty (B):** SALO is configured to gather logs at the highest granularity (L5) from all microservices **when** at least one microservice is detected unhealthy by the GateKeeper. Conversely, when all the microservices are healthy, logs are collected at lowest granularity (L1).
- **Collect From Faulty Only (C):** SALO is configured to gather logs at highest granularity (L5) only from those microservices that are detected unhealthy by the GateKeeper. For all the other microservices, logs are collected at the lowest granularity (L1).
- **Collect From Potentially-Impacted when One Is Faulty:** As discussed earlier, a faulty microservice may impact its neighboring microservice(s), leading to fault cascade. The cascade can be up to k hops as defined in static configuration policy (SubSection III-B). There are two ways in which the impact on neighbouring microservices and associated appropriate log levels can be determined.
  1) **Static Equal Edge Weightage (D):** SALO is configured to collect logs at the highest granularity (L5) L5 for unhealthy microservices. And, for the neighbouring microservices, determine the log level based on the equal edge weightage method discussed in the SubSection III-C.
  2) **Dynamic Interaction-based Edge Weightage (E):** Same as above, however, the log levels for the neighboring microservices are determined based on the interaction degree that the neighboring microservice has with the faulty microservice, refer SubSection III-C.

TABLE II: Overview of Various SALO Scenarios

Scenarios	Logging Strategy during Fault Occurrence
A	Collect all logs continuously from every microservice
B	Collect logs from all microservices when one becomes unhealthy
C	Collect logs only from the unhealthy microservice
$D_{k=1}$	Collect logs from potentially affected neighbors within 1 hop, with equal weightage
$E_{k=1}$	Collect logs from potentially affected neighbors within 1 hop, using interaction-based weightage
$D_{k=2}$	Collect logs from potentially affected neighbors within 2 hops, with equal weightage
$E_{k=2}$	Collect logs from potentially affected neighbors within 2 hops, using interaction-based weightage
$D_{k=3}$	Collect logs from potentially affected neighbors within 3 hops, with equal weightage
$E_{k=3}$	Collect logs from potentially affected neighbors within 1 hops, using interaction-based weightage

For strategies D and E, we evaluated the system for different values of k, the number of hops in blast radius, ranging from 1 to 3. For instance, $k = 3$, all the microservices within 3 hops shall be considered as potentially impacted neighbours, marked for raising the log levels. All the variations of SALO are summarised in Table II.

*1) **Q1 - Impact of SALO on downstream tasks**:* There are trade-off between collecting less logs using a system like SALO, and the impact that it could have on the existing AIOps downstream tasks due to unavailability of all the logs. For this experiment, we have chosen Fault Classification as one of the downstream tasks, and we would test the accuracy of the fault classification with and without SALO in place.

The task of Fault classification is designed to predict each

TABLE III: The fault classification for different SALO scenarios
NA = Results not applicable since the maximum hops possible for QoTD is 2

Scenario	QoTD				TT Transient				TT Persistent			
	Ar	**P**	**R**	**F1**	**Ar**	**P**	**R**	**F1**	**Ar**	**P**	**R**	**F1**
A	0.884	0.881	0.875	0.867	0.978	0.945	0.865	0.905	0.997	0.995	1	0.995
B	0.930	0.951	0.898	0.917	0.989	0.940	0.950	0.970	0.991	1	0.995	0.995
C	0.873	0.873	0.916	0.894	0.973	0.935	0.835	0.880	0.996	0.995	0.980	0.985
$D_{k=1}$	**1**	**1**	**1**	**1**	**0.991**	**0.945**	**0.985**	**0.960**	**0.999**	**1**	**1**	**1**
$E_{k=1}$	**1**	**1**	**1**	**1**	**0.991**	**0.945**	**0.985**	**0.960**	**0.999**	**1**	**1**	**1**
$D_{k=2}$	**1**	**1**	**1**	**1**	**0.991**	**0.945**	**0.985**	**0.960**	**0.999**	**1**	**1**	**1**
$E_{k=2}$	**1**	**1**	**1**	**1**	**0.991**	**0.945**	**0.985**	**0.960**	**0.999**	**1**	**1**	**1**
$D_{k=3}$	NA	NA	NA	NA	**0.991**	**0.945**	**0.985**	**0.960**	**0.999**	**1**	**1**	**1**
$E_{k=3}$	NA	NA	NA	NA	**0.991**	**0.945**	**0.985**	**0.960**	**0.999**	**1**	**1**	**1**

30 sec window of log data as faulty or non-faulty. Log data in each dataset was segmented into 30 sec windows, resulting in 756 and 6240 windows for QoTD and *TT*, respectively. Since fault classification is a binary classification task, 80-20 split was used to prepare train and test datasets. Each window of log data in the train set and test set is labeled as Faulty(1) or Non-faulty(0), depending upon if they contain faulty logs or not. For each scenario, a state-of-the-art Fault Classification model was trained using *Bert-For-Sequence-Classification*[11].

To compare all the SALO scenarios in their ability to detect faulty windows, we use Accuracy(Ar), Precision(P), Recall(R) and F1-Score(F1) [45] as evaluation metrics. Precision measures the ratio of number of correct faulty windows detected to the total number of faulty windows detected. Whereas, Recall measures the ratio of the number of faulty windows detected to the actual number of faulty windows in the dataset. Accuracy measures the ratio of total number of correctly identified faulty windows and non-faulty windows to the total number of windows present in the test dataset. F1-Score computes the harmonic mean of precision and recall, such that it symmetrically represents both precision and recall in one metric. Although, Accuracy is easy to interpret, however it is not the preferred metric when the data is unevenly distributed, i.e. class imbalance. F1-score is a preferred metric to report over Accuracy when there is a class imbalance (uneven distribution). It is obvious that the number of faulty windows are much lesser than the number of non-faulty windows in our datasets, therefore we use F-Score to report the results. For example, the number of faulty windows and non-faulty windows in the Tansient-TT transient are 208 and 6032, respectively.

As observed in the results presented in Table III, for all datasets, all scenarios with SALO, except C, outperform the baseline scenario A across all metrics. While the initial expectation from this experiment was to demonstrate that reduced log volume should not adversely affect downstream tasks, **it is noteworthy that SALO's outperform the baseline in most cases**. This can be attributed to the fact that log lines are selectively collected at higher granularity only from the impacted or potentially impacted microservices during the fault occurrence. This approach contributes to noise reduction by minimizing the inclusion of irrelevant log lines that could potentially confuse the fault classification model.

As depicted in Table III, scenarios D and E yield the best results when compared to scenario A, showcasing improvement across all metrics. This is because **scenarios D and E strike a fine balance in log data collection, guided by the blast radius to identify potentially impacted microservices, where log data is collected at the highest granularity only from the impacted microservices and their potentially affected neighbors**.

Please be aware that the values of evaluation metrics for scenarios D and E remain identical due to the behavior of the Fault Classification model. According to this model, a window is considered unhealthy if it detects the log signature of at-least one fault related to a microservice, although there is a fault cascade involving multiple faults across different microservices. Ideally, for cascading faults with up to k hops, a window should only be detected by the model as faulty if all the log signatures associated with each fault in the cascade are present within the window. To tackle this issue, we have designed an additional experiment as follows.

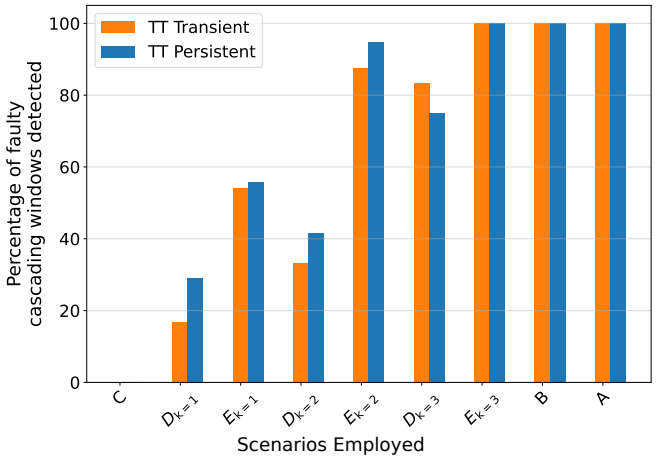

Fig. 6: Percentage of Cascading faulty windows detection under various scenarios

2) Q2: Detection of cascading faulty sequences and microservices: As we know, when a fault occurs in a microservice, it can potentially affect neighboring microservices in

[11]https://huggingface.co/docs/transformers/v4.39.2/en/model_doc/bert#transformers.BertForSequenceClassification

a cascade. Hence, it's crucial to assess SALO's ability to collect logs at the appropriate levels from all microservices involved in the fault cascade sequence. We verify this by examining the percentage of identified faulty windows in fault classification results that contain the signature faulty log lines from the affected microservices in the cascade chain. For instance, in scenario C, SALO raises the log granularity only for the original faulty microservice but doesn't do the same for the neighboring microservices impacted by the fault cascade. In this scenario, fault classification might still detect the window as faulty, with only one originating microservice appearing in the faulty logs. However, the other affected microservices in the cascade of faults should also have had their signature faulty logs included in the faulty window. This explains why for scenario C, in Fig. 6, the percentage of faulty cascading windows detected is zero. However, for scenarios D and E with increasing blast radius (hops), the percentage of faulty cascading windows detected improves with maximum at $k = 3$. It is also noteworthy that using dynamic interaction-based weightage ($E_k = 3$) for increasing log granularity proves to be a better method as opposed to assigning equal weightage for all neighboring microservices ($D_k = 3$), although that both exhibit identical performance across all metrics (refer Table III). Scenario B and A are the upper bound, as in both cases, data from all the microservices is collected at the highest granularity during the faulty event.

In summary, **the results illustrate that $E_{k=3}$ performs the best in terms of detecting all fault sequences in both TT-Transient and TT-Persistent datasets.**. It is important to note that these results are specific to the *TT* application, where the complexity of the application clearly allows for clear visibility of cascading fault interactions.

3) Q3: Log Volume Reduction: One of the primary goals of SALO is to minimize the log data volume gathered from large-scale applications. Therefore, we calculate and compare the number of log lines collected under various SALO scenarios with the baseline. Results are presented in Fig. 7. The x-axis has the baseline (A) and the various SALO scenarios ordered by the volume of log data. Note that, scenario C collects the least amount of log data ($94K$ for TT-Persistent), however, it is not the desired method because it cannot identify all the faulty windows and the constituent faults in the cascading fault chain, as is evident from the results in Table III and Fig. 6. The baseline A collects the largest amount of logs ($3.7M$ for TT-Persistent), because it continuously collects all the log data, from all the microservices. The SALO scenarios D and E with varying values of k would collect different amount of data, mostly depending on the value k. For example, for the TT-Persistent dataset, the SALO scenarios $E_{k=3}$ collected $180K$ log lines, a reduction of 95% compared to the amount of logs collected by baseline A. **Combining these insights with results in Table III and Fig. 6, one can deduce that scenario $E_{k=3}$ has F1-Score of 1 and 100% faults were identified, while only 5% of log data was collected.**

4) Q4: System Overheads: We must ensure that all the building blocks of SALO are lightweight and have minimal overhead on the resources. Experiment was performed to measure the CPU and Memory utilisation of GateKeeper and Central Controller for scenario $E_{k=3}$, because this is the best performing scenario under which SALO collects the optimal amount of data while not impacting the downstream task. The findings in Table IV indicate that both the Gatekeeper and Central Controller exhibit negligible CPU and memory overhead. **This minimal resource utilization underscores the practicality of SALO, as it imposes minimal computational and memory burdens on the underlying infrastructure.**

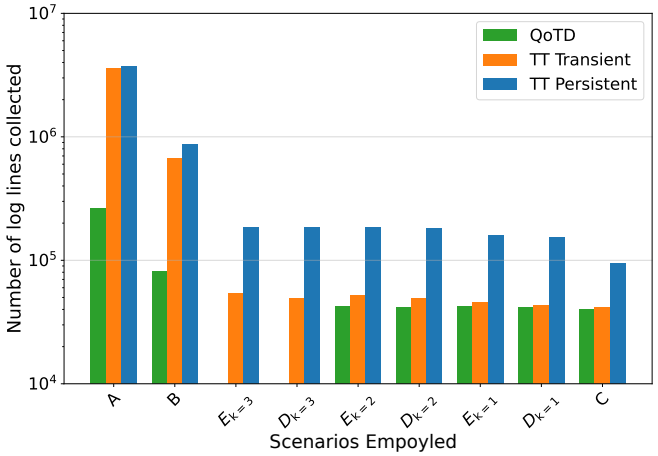

Fig. 7: Log volume reduced using various scenarios

TABLE IV: System Overheads after deploying SALO

Overheads	GateKeeper	Central Controller
CPU	2 millicores	~9 millicores
Memory	~110 MB	59 MB

VI. Conclusion and Future Work

In this paper, we introduce SALO, an autonomous system designed to efficiently reduce volume of logs originating from large-scale microservices-based applications deployed on the cloud, while simultaneously enhancing downstream task performance. We demonstrated that SALO not only mitigates the considerable costs associated with log observability in terms of resources and finances, but also streamlines the log observability pipelines to be more resource-efficient and leaner. As a future endeavor, one area of exploration entails eliminating the central controller from SALO. In this setup, each GateKeeper would regularly broadcast its health status to neighboring microservices. If a GateKeeper detects an unhealthy neighbor, it would adjust its log levels accordingly based on the intensity of interaction with that microservice. Additionally, we intend to expand the scope of the proposed SALO system to encompass multi-cloud multi-cluster environments, transitioning from deploying one GateKeeper per microservice to one per cluster. In this setup, each GateKeeper would be responsible for managing log levels for all workloads running in the cluster.

References

[1] J. Kosińska, B. Baliś, M. Konieczny, M. Malawski, and S. Zieliński, "Toward the observability of cloud-native applications: The overview of the state-of-the-art," *IEEE Access*, vol. 11, pp. 73 036–73 052, 2023.

[2] N. Marie-Magdelaine, T. Ahmed, and G. Astruc-Amato, "Demonstration of an observability framework for cloud native microservices," in *IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2019, pp. 722–724.

[3] M. M. Alshammari, A. A. Alwan, A. Nordin, and I. F. Al-Shaikhli, "Disaster recovery in single-cloud and multi-cloud environments: Issues and challenges," in *2017 4th IEEE International Conference on Engineering Technologies and Applied Sciences (ICETAS)*, 2017, pp. 1–7.

[4] X. Zhang, C. Du, Y. Li, Y. Xu, H. Zhang, S. Qin, Z. Li, Q. Lin, Y. Dang, A. Zhou, S. Rajmohan, and D. Zhang, "Halo: Hierarchy-aware fault localization for cloud systems," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, ser. KDD '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 3948–3958. [Online]. Available: https://doi.org/10.1145/3447548.3467190

[5] A. Saha and S. C. H. Hoi, "Mining root cause knowledge from cloud service incident investigations for aiops," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 197–206. [Online]. Available: https://doi.org/10.1145/3510457.3513030

[6] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang, "Capturing and enhancing in situ system observability for failure detection," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 1–16. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/huang

[7] B. Hanson, S. Stall, J. Cutcher-Gershenfeld, K. Vrouwenvelder, C. Wirz, Y. Rao, and G. Peng, "Garbage in, garbage out: mitigating risks and maximizing benefits of ai in research," *Nature*, vol. 623, no. 7985, pp. 28–31, 2023.

[8] D. Yokelson, O. Lappi, S. Ramesh, M. Väisälä, K. Huck, T. Puro, B. Norris, M. Korpi-Lagg, K. Heljanko, and A. Malony, "Observability, monitoring, and in situ analytics in exascale applications," in *Cray User Group 2023*, May 2023, cray User Group, CUG ; Conference date: 07-05-2023 Through 11-05-2023.

[9] R. Souza, T. J. Skluzacek, S. R. Wilkinson, M. Ziatdinov, and R. F. da Silva, "Towards lightweight data integration using multi-workflow provenance and data observability," in *IEEE 19th International Conference on e-Science (e-Science)*, 2023, pp. 1–10.

[10] G. Bosilca, A. Bouteiller, A. Guermouche, T. Herault, Y. Robert, P. Sens, and J. Dongarra, "A failure detector for hpc platforms," *Int. J. High Perform. Comput. Appl.*, vol. 32, no. 1, p. 139–158, jan 2018. [Online]. Available: https://doi.org/10.1177/1094342017711505

[11] M. Scrocca, R. Tommasini, A. Margara, E. D. Valle, and S. Sakr, "The kaiju project: enabling event-driven observability," in *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 85–96. [Online]. Available: https://doi.org/10.1145/3401025.3401740

[12] G. Yu, P. Chen, Y. Li, H. Chen, X. Li, and Z. Zheng, "Nezha: Interpretable fine-grained root causes analysis for microservices on multi-modal observability data," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 553–565. [Online]. Available: https://doi.org/10.1145/3611643.3616249

[13] P. Notaro, J. Cardoso, and M. Gerndt, "A survey of aiops methods for failure management," *ACM Trans. Intell. Syst. Technol.*, vol. 12, no. 6, nov 2021. [Online]. Available: https://doi.org/10.1145/3483424

[14] J. Bogatinovski, S. Nedelkoski, J. Cardoso, and O. Kao, "Self-supervised anomaly detection from distributed traces," in *IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, 2020, pp. 342–347.

[15] Y. Hua, "A systems approach to effective aiops implementation," Ph.D. dissertation, Massachusetts Institute of Technology, 2021.

[16] S. Karumuri, F. Solleza, S. Zdonik, and N. Tatbul, "Towards observability data management at scale," *SIGMOD Rec.*, vol. 49, no. 4, p. 18–23, mar 2021. [Online]. Available: https://doi.org/10.1145/3456859.3456863

[17] R. Picoreti, A. Pereira do Carmo, F. Mendonça de Queiroz, A. Salles Garcia, R. Frizera Vassallo, and D. Simeonidou, "Multi-level observability in cloud orchestration," in *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, 2018, pp. 776–784.

[18] M. Usman, S. Ferlin, A. Brunstrom, and J. Taheri, "A survey on observability of distributed edge container-based microservices," *IEEE Access*, vol. 10, pp. 86 904–86 919, 2022.

[19] J. Kosińska, B. Baliś, M. Konieczny, M. Malawski, and S. Zieliński, "Toward the observability of cloud-native applications: The overview of the state-of-the-art," *IEEE Access*, vol. 11, pp. 73 036–73 052, 2023.

[20] N. Marie-Magdelaine, T. Ahmed, and G. Astruc-Amato, "Demonstration of an observability framework for cloud native microservices," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2019, pp. 722–724.

[21] P. Thantharate, "Intelligentmonitor: Empowering devops environments with advanced monitoring and observability," in *2023 International Conference on Information Technology (ICIT)*, 2023, pp. 800–805.

[22] A. Khichane, I. Fajjari, N. Aitsaadi, and M. Gueroui, "5gc-observer: a non-intrusive observability framework for cloud native 5g system," in *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, 2023, pp. 1–10.

[23] S. Karumuri, F. Solleza, S. Zdonik, and N. Tatbul, "Cloud observability: A melting pot for petabytes of heterogenous time series." in *CIDR*, 2021.

[24] L. Yu, Z. Zheng, Z. Lan, T. Jones, J. M. Brandt, and A. C. Gentile, "Filtering log data: Finding the needles in the haystack," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, 2012, pp. 1–12.

[25] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, "Towards automated log parsing for large-scale log data analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 6, pp. 931–944, 2018.

[26] Y. Liu, X. Zhang, S. He, H. Zhang, L. Li, Y. Kang, Y. Xu, M. Ma, Q. Lin, Y. Dang, S. Rajmohan, and D. Zhang, "Uniparser: A unified log parser for heterogeneous log data," in *Proceedings of the ACM Web Conference 2022*, ser. WWW '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1893–1901. [Online]. Available: https://doi.org/10.1145/3485447.3511993

[27] T. Kimura, K. Ishibashi, T. Mori, H. Sawada, T. Toyono, K. Nishimatsu, A. Watanabe, A. Shimoda, and K. Shiomoto, "Spatio-temporal factorization of log data for understanding network events," in *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, 2014, pp. 610–618.

[28] S. DeCelles, M. Stamm, and N. Kandasamy, "Data reduction, compression, and recovery for online performance monitoring," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019, pp. 256–263.

[29] X. Li, H. Zhang, V.-H. Le, and P. Chen, "Logshrink: Effective log compression by leveraging commonality and variability of log data," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3608129

[30] R. Christensen and F. Li, "Adaptive log compression for massive log data," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 1283–1284. [Online]. Available: https://doi.org/10.1145/2463676.2465341

[31] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, J. Chen, X. He, R. Yao, J.-G. Lou, M. Chintalapati, F. Shen, and D. Zhang, "Robust log-based anomaly detection on unstable log data," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 807–817. [Online]. Available: https://doi.org/10.1145/3338906.3338931

[32] S. Son, M.-S. Gil, and Y.-S. Moon, "Anomaly detection for big log data using a hadoop ecosystem," in *2017 IEEE International Conference on Big Data and Smart Computing (BigComp)*, 2017, pp. 377–380.

[33] L. N. Valli, S. N., and V. Geetha, "Importance of aiops for turn metrics

and log data: A survey," in *2023 2nd International Conference on Edge Computing and Applications (ICECAA)*, 2023, pp. 799–802.

[34] J. Zhu, S. He, P. He, J. Liu, and M. R. Lyu, "Loghub: A large collection of system log datasets for ai-driven log analytics," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. Los Alamitos, CA, USA: IEEE Computer Society, oct 2023, pp. 355–366. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ISSRE59848.2023.00071

[35] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, J. Chen, X. He, R. Yao, J.-G. Lou, M. Chintalapati, F. Shen, and D. Zhang, "Robust log-based anomaly detection on unstable log data," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 807–817. [Online]. Available: https://doi.org/10.1145/3338906.3338931

[36] O. Sharma, M. Verma, S. Bhadauria, and P. Jayachandran, "A guided approach towards complex chaos selection, prioritisation and injection," in *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, 2022, pp. 91–93.

[37] J. Hwang, L. Shwartz, Q. Wang, R. Batta, H. Kumar, and M. Nidd, "Fixme: Enhance software reliability with hybrid approaches in cloud," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2021, pp. 228–237.

[38] 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.

[39] A. Vervaet, "Monilog: An automated log-based anomaly detection system for cloud computing infrastructures," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 2739–2743.

[40] R. Mahindru, H. Kumar, and S. Bansal, "Log anomaly to resolution: Ai based proactive incident remediation," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1353–1357.

[41] P. Gupta, H. Kumar, D. Kar, K. Bhukar, P. Aggarwal, and P. Mohapatra, "Learning representations on logs for aiops," in *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. IEEE, 2023, pp. 155–166.

[42] "Red hat openshift enterprise kubernetes container platform," https://www.redhat.com/en/technologies/cloud-computing/openshift, (Accessed on 03/29/2024).

[43] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, "Benchmarking microservice systems for software engineering research," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2018, pp. 323–324.

[44] K. Bhukar, H. Kumar, R. Mahindru, R. Arora, S. Nagar, P. Aggarwal, and A. Paradkar, "Dynamic alert suppression policy for noise reduction in aiops," in *ACM/IEEE International Conference on Software Engineering*, 2024.

[45] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks," *Information processing & management*, vol. 45, no. 4, pp. 427–437, 2009.