Toward Context-Aware Anomaly Detection for AIOps in Microservices Using Dynamic Knowledge Graphs

Pieter Moens, Bram Steenwinckel, Femke Ongenae, Bruno Volckaert, *Senior Member, IEEE*, and Sofie Van Hoecke

***Abstract*—Microservice applications are omnipresent due to their advantages, such as scalability, flexibility and consequentially resource cost efficiency. The loosely-coupled microservices can be easily added, replicated, updated and/or removed to address the changing workload. However, the distributed and dynamic nature of microservice architectures introduces a complexity with regard to monitoring and observability, which is paramount to ensure reliability, especially in critical domains. Anomaly detection has become an important tool to automate microservice monitoring and detect system failures. Nevertheless, state-of-the-art solutions assume the topology of the monitored application to remain static over time and fail to account for the dynamic changes the application, and the infrastructure it is deployed on, undergoes. This paper tackles these shortcomings by introducing a context-aware anomaly detection methodology using dynamic knowledge graphs to capture contextual features which describe the evolving state of the monitored system. Our methodology leverages resource and network monitoring to capture dependencies between microservices, and the infrastructure they are running on. In addition to the methodology for anomaly detection, this paper presents an open-source benchmark framework for context-aware anomaly detection that includes monitoring, fault injection and data collection. The evaluation on this benchmark shows that our methodology consistently outperforms the non-contextual baselines. These results underscore the importance of contextual awareness for robust anomaly detection in complex, topology-driven systems. Beyond these achieved improvements, our benchmark establishes a reproducible and extensible foundation for future research, facilitating the experimentation with broader ranges of models and a continued advancement in context-aware anomaly detection.**

***Index Terms*—Anomaly detection, microservices, knowledge graph, dynamic graphs, knowledge graph embedding, AIOps, context.**

Received 23 September 2024; revised 1 April 2025, 10 October 2025, and 17 December 2025; accepted 4 January 2026. Date of publication 12 January 2026; date of current version 16 January 2026. This research was funded by the FWO Junior Research project HEROI2C (G085920N) that investigates hybrid machine learning for improved infection management in critically ill patients. The associate editor coordinating the review of this article and approving it for publication was D. G. Berbecaru. *(Corresponding author: Bram Steenwinckel.)*

The authors are with the IDLab, Ghent University–imec, 9052 Gent, Belgium (e-mail: Pieter.Moens@ugent.be; Bram.Steenwinckel@ugent.be; Femke.Ongenae@ugent.be; Bruno.Volckaert@ugent.be; Sofie.VanHoecke@ugent.be).

Digital Object Identifier 10.1109/TNSM.2026.3652304

I. Introduction

IN THE digital landscape that has characterized the last decade, businesses ranging from e-commerce to finance up to healthcare rely on software systems to deliver seamless experiences to their end-users. For instance, an online retail platform handles thousands, or even millions, of transactions daily [1], while eHeatlh platforms are processing a large amount of (temporal) data from a plethora of monitoring equipment that requires rapid and accurate analysis and interpretation [2]. To ensure smooth operations and meet stakeholder demands, many companies have transitioned from a traditional monolithic architecture to a Microservice Architecture (MSA) [3]. Instead of one large, unwieldy application handling everything, e.g., from inventory management to payment processing in e-commerce or patient monitoring and prescription handling in eHealth, the system is decoupled into smaller microservices. Each microservice focuses on a specific task, such as managing product listings, ingesting patient monitoring data or processing payments, and communicates over the network with other dependent services through well-defined Application Programming Interfaces (APIs). These loosely coupled, fine-grained microservices can be individually developed and deployed across distributed machines. This improves scalability and maintenance, but also allows for more frequent and granular updates, introducing new versions and features without disrupting the entire system. For example, during peak shopping seasons, the payment processing microservice can be scaled independently to handle increased traffic without affecting other parts of the system. Similarly, the microservices for ingestion and processing of real-time monitoring data in an eHeatlh system can be scaled individually depending on the number of hospital rooms currently occupied. This avoids horizontal scaling of the entire application, and thus minimizes the operational cost while ensuring availability of the overall system.

However, these scalability and resource efficiency benefits obtained through adoption of the MSA come at a cost of increased operational complexity in terms of monitoring and observability [4]. As each microservice encapsulates a single business functionality, a request to the application often traverses across multiple interconnected components before serving the end-user. A fault occurring in one component can therefore propagate throughout the system. For example, the checkout microservice for a customer of an e-commerce

platform or a patient being discharged in an eHealth platform could rely on the payment and e-mail microservices. A fault in one of these services, e.g., network latency, can affect in these cases the response time of the checkout service, which in turn could lead to a violation of timeouts toleration, resulting in an invalid request observed at the front-end of the application.

System faults observed in microservice applications include, but are not limited to, network latency, anomalous resource consumption and unavailability of system components [5]. These faults can cause inaccurate autoscaling of services, violation of Service Level Agreements (SLAs) or even temporary outage of the application. This in turn can result in increased operational costs, costs associated with downtime, or even more drastic effects due to delayed patient care in eHealth applications.

Classic monitoring solutions, e.g., using Prometheus [6] and Grafana, or using the ElasticSearch, Logstash and Kibana (ELK) stack [7], are capable of notifying IT and DevOps engineers in case of failures in the system through threshold-based alerting. These solutions are based on fine-grained alerting and require multi-faceted thresholds for Key Performance Indicators (KPIs) in different layers of the application [8]. Determining and updating these thresholds, as well as deriving the root cause of a fault that propagated through the system based on numerous alerts, is a complex and tedious task for IT and DevOps engineers.

Artificial Intelligence for IT Operations (AIOps) [9] aims to move from monitoring towards observability and address this issue through Machine Learning (ML). Instead of monitoring microservices using static thresholds, ML-based solutions, such as Dynatrace [10], adopt Anomaly Detection (AD) and Root Cause Analysis (RCA) techniques to automate the prediction and detection of faults. It thus allows engineers to respond promptly to potential failures by improving both failure management as well as resource provisioning (e.g., resource consolidation, service composition, and workload estimation).

The biggest challenge that arises when applying state-of-the-art AD techniques to microservice-based applications, however, is their inherent dynamic behavior. To optimize resource consumption and minimize operational costs while ensuring maximum reliability of the overall system when handling varying workload, its components are continuously added, updated or removed through autoscaling and rescheduling of resources. While AD for microservice-based applications has received increased attention in literature over recent years, this paper highlights shortcomings and open challenges when focusing on the temporal characteristics of the monitored system.

Most related work infers the application topology. It does this by using Control Flow Graphs (CFG) extracted from logs or distributed traces to capture how microservices depend on each other [11]. The relations that can be obtained from these distributed traces are, however, often limited to the dependencies between different microservices, and do not include their relation with the infrastructure they are running on, for example, the specific hardware node the microservice is deployed on. Taking into account these relations can be paramount for detecting anomalies stemming from node-level faults, e.g., node unavailability and resource shortage. Furthermore, integration of log-based and distributed tracing in MSA is intrusive and requires changes to the application itself [12]. Developers are tasked with implementing descriptive and informative logging at critical points in the code. The resulting AD solutions are therefore strongly dependent on the quality and uniformity of these logs. In addition to this, a commonly observed shortcoming in related work is that the overall topology or dependency graph is considered to be static over time, meaning that the dependencies between components remains the same throughout the entire dataset. The dynamicity of the monitored system can lead to temporal drift and affect the performance of the deployed AD models over time. Accurately capturing these types of dynamic changes in benchmark datasets is paramount to evaluate the applicability of the proposed solutions to real-world scenarios.

We define concrete requirements for the AD solution proposed in this paper based on the these aforementioned open challenges:

- R1 It should take into account both the application layer as well as the infrastructure layer, and therefore be able to detect faults in both.
- R2 It should be non-invasive and introduce no overhead for the developers.
- R3 It should be able to accurately detect system faults under dynamic conditions. Temporal changes in the topology or state of the monitored system should have minimal impact on the AD model.
- R4 It should be deployable for real-time detection of system faults.

This work makes two main complementary contributions addressing the identified requirements. First, we introduce a benchmark dataset that captures the multi-layer and non-invasive aspects of microservice environments (R1, R2). Second, we propose an anomaly detection approach leveraging heterogeneous knowledge graphs to model system dynamics and support real-time fault detection (R3, R4).

The presented research can be subdivided in four (sub)contributions in chronological order of the paper:

- C1 An open, reproducible deployment bundle that integrates non-intrusive, production-grade monitoring and fault-injection tooling to produce literal-enriched, multi-layer evaluations for microservice anomaly detection.
- C2 A schema or ontology to construct these Knowledge Graphs (KGs) and express the dependencies between heterogeneous components in both the application and infrastructure layers of a MSA.
- C3 A benchmark dataset for AD in microservices containing both data features and contextual features to capture the dynamic state of the system over time.
- C4 A novel methodology for real-time context-aware AD in microservices based on KGs.

The remainder of this paper is structured as follows. First, Section II summarizes related work in the domain of anomaly detection in microservices, and the benchmark datasets used for evaluation. It also highlights the contributions of this work with respect to this related work. Before diving into the

specifics of our methodology, we first describe the microservice environment (C1), the designed schema used to construct the KGs (C2) and the resulting benchmark dataset (C3) in Sections III and IV respectively, which provide the necessary context for understanding our approach. Section V then introduces the proposed methodology towards dynamic KG-based context-aware anomaly detection in microservices (C4). Section VI provides evaluation results of the proposed solution and a comparison with the state-of-the-art. Section VII reflects on the proposed solution and discusses its limitations and proposes directions for future work. Finally, Section VIII concludes this paper and highlights the main insights of the conducted research.

II. Related Work

This section provides a summary and discussion of related literature in the domain of anomaly detection in microservices. We discuss methodologies proposed in recent years as well as the benchmark datasets on which they have been evaluated, focusing on the type of faults that are being considered and the conditions under which the dataset has been collected.

Related work in the domain of microservice-based monitoring and anomaly detection can be roughly divided based on the type of data used, i.e., log-based solutions, distributed tracing-based solutions and resource metric-based solutions [5].

A. Log-Based Solutions

In most nodes, containers and devices, logs are captured at critical points for debugging and root cause analysis purposes. These textual messages are often unstructured and require parsing strategies and systems [13]. When correctly parsed, logs provide valuable information about the current state of the individual services, which has been showcased in the related work discussed in the following paragraphs.

Nandi et al. [11] introduce a technique to automatically mine different log message types, or templates, from unstructured execution logs. They construct a CFG from the logs to capture the execution flow within a single machine. LogSed [14] extends this method and introduces a temporal dimension to the collected CFG by calculating the time between logs within a single flow, resulting in a time-weighted CFG. They used these time-weighted graphs to detect deviations from normal application behavior by matching captured interval times with the previously established time weights in the learned normal CFGs. LogSed [14] uses a benchmark microservice application, namely Acme Air [15]. They simulate a load of several hundreds of concurrent requests to several microservices in the application and inject artificial faults pertaining three types of anomalies, namely sequence anomalies, redundancy anomalies and latency anomalies. These anomalies directly relate to the CFGs they construct such as deviations from known sequences, previously unseen logs that can not be mapped on the known graph, and logs exceeding the expected interval time.

DeepLog [16] uses a deep learning model, i.e., a Long Short-Term Memory (LSTM) model, to model a system log as a natural language sequence. They then learn log patterns from normal behavior data and detect anomalies based on deviations from these patterns. In addition, they construct CFGs from these system logs for diagnosis and RCA of the detected anomalies. DeepLog focuses on cybersecurity threats and captures anomalies such as Denial of Service (DoS) and detected port scans. The dataset considered stems from the 2011 IEEE VAST Mini Challenge 2 [17] and contains log messages from a firewall and intrusion detection system.

AutoLog [18] captures batches of logs from a given node, container or microservice and calculates a numerical entropy vector based on the frequency of the occurrence of certain terms in the batch of log messages. They then train a deep learning model, i.e., an Auto Encoder (AE) model, to learn this normal behavior and use this model to detect future anomalies. This unsupervised method does not require examples of anomalies in the training data and is able to detect previously unseen anomalies. AutoLog [18] captures its own log data using five different scenarios, each representing a single system fault. Similarly to DeepLog, the considered scenarios are focused on cybersecurity and include the detection of DoS, bruteforce, tampering with data and/or logs.

A second category of log-based solutions for anomaly detection in microservice-based applications focuses on system call (syscall) log monitoring which captures and tracks low-level system interactions made by applications (e.g., file access, network connections, and process execution). In contrast to application log-based solutions, syscall log monitoring is less invasive to the application under inspection, making it more suitable for monitoring third-party or legacy applications. Due to the nature of syscalls, related work in this field is primarily focused on the detection of anomalous behavior and container-based security threats, such as privilege escalation, unauthorized file modifications, or container breakouts.

Related work, such as ContainerGuard [19], addresses the problem of imperfect resource isolation and shared kernel between containers and uses a Variational AE trained on syscall logs to detect meltdown and spectre attacks. Guo et al. [20] utilizes the information captured by syscall logs to detect zero-day data manipulation and DoS attacks, meaning that their solution is able to detect previously unseen threats.

Karn et al. [21], on the other hand, identified cryptocurrency mining as one of the most serious malware threats in container-based applications and proposed a solution using sycall log monitoring and eXplainable AI (XAI) to detect these malicious cryptocurrency miners and prevent server resources from being hijacked.

All of these log-based solutions capture detailed information about events that occur in a single machine, or service. However, logs lack the full context of the transaction flow across system components, or microservices, making it difficult to link correlated events as faults propagate through the distributed system, and to derive the impact on the overall system. Information about the dependencies between different system components is essential to capture the state of the monitored system and detect anomalies that might be considered normal in different contexts, e.g., the expected resource usage of a hardware machine is directly correlated to the number and type of microservices currently running on it.

B. Distributed Tracing-Based Solutions

Distributed tracing describes the execution process of a request across service instances, i.e., invocation chain, by attaching a trace ID to the captured log messages, making it possible to easily capture dependencies between services.

Nedelkoski et al. [22] use these distributed traces for anomaly detection and classification by training a deep learning model, namely a Variational AE in combination with a Recurrent Neural Network (RNN). They demonstrate their model on a lab environment for four different injected anomalies: (i) increase in latency caused by the service, (ii) network latency, (iii) network packet loss, and (iv) server process kill. They implemented their own microservice application consisting of only three microservices for evaluation of their proposed method.

DeepTraLog [23] investigates a graph-based approach using a Gated Graph Neural Network (GGNN) model for graph embedding of logging data of distributed traces. They then train a Support Vector Data Description (SVDD) classifier on these embeddings for supervised AD. BertHTLG [24] extends on this approach and improves the calculation of embeddings from logging data using a heterogeneous graph representation enhanced by Sentence-Bert before utilizing a Graph Convolutional Network (GCN) and SVDD. Similarly to DeepTraLog, they then train a SVDD model for supervised AD. DeepTraLog [23] and BertHTLG [24] both evaluate their approaches on the TrainTicket benchmark dataset [25], which contains 14 different types of faults. These faults, however, are limited to service-level anomalies, such as incorrect configuration of SSL, JVM or Docker, incorrect calculation of ticket prices, a wrong column name included in the constructed SQL statement and incorrect synchronization of service states among services.

Distributed tracing-based solutions, just like logging-based methods, require custom code within each deployed service. The collected logs from distributed tracing solutions are thus limited to the application layer, making it difficult to capture the impact of an individual trace on the infrastructure the microservices are running on. Furthermore, monitoring systems relying on logs and traces are considered invasive as the adoption of these systems affects the underlying software application itself. The performance of these approaches is also heavily reliant on the quality of the available logs.

C. Resource Metric-Based Solutions

As resource metrics (e.g., CPU and memory) and network metrics (e.g., throughput and latency) can be captured without altering the application, resource metric-based solutions are less intrusive and do not require altering the underlying software application itself, nor are they reliant on the qualitative log data.

Du et al. [26] leverage resource and network metrics for supervised classification of four types of anomalies, namely CPU hog, memory hog, network latency and network package loss. They evaluate four common classification techniques, i.e., k-Nearest Neighbors (k-NN), Random Forest (RF), Naive Bayes (NB) and Support Vector Machines (SVM). Du et al. capture a dataset using a custom lab environment. They monitor resource consumption within Virtual Machines (VMs), such as CPU, memory, cache, network usage (i.e., bytes sent and received). They implemented a custom script to inject four targeted system faults, namely high CPU consumption, memory leak, network package loss and network latency increase.

Nobre et al. [27] investigates a supervised deep learning approach to anomaly detection in microservice-based systems. They collected and labeled network resource data (e.g., network latency and response status codes) through fault injection, and compared the performance of a Multi-Layer Perceptron (MLP) classification model trained on the service-level monitoring data captured during the load test versus a model trained on resource-level monitoring data (e.g., CPU and memory usage). Nobre et al. evaluate their approach using the SockShop benchmark application [28]. They injected two faults in two specific microservices, namely the *orders* and *carts* services. The anomalies are on the one hand a functional anomaly, which throws a runtime exception in the respective services, and on the other hand a performance anomaly, which pauses the execution of the program for 5 seconds.

ServiceAnomaly [29] uses a combination of distributed tracing and network resource metrics to characterize the normal behavior of the microservice application. They model the normal behavior of the application as a CFG by extracting inter-service dependencies from the collected traces. They then annotate the edges of this CFG using network resource metrics. During prediction, they calculate the differences between collected graphs from the healthy state against the newly received graph. As the authors mention, their combination of distributed traces with network resource metrics yields high performance results, but their proposed approach is not capable of capturing dynamic changes in the application and suffers when the normal behavior changes over time. ServiceAnomaly evaluates its proposed solution on two benchmark datasets, namely Train Ticket [25] and TeaStore [30]. They injected two additional faults on these benchmark environment under varying load, namely a fixed service latency of 15 seconds for 80% of the requests and a service hangup anomaly simulated by temporarily downscaling the targeted microservice to zero replicas.

Resource metric-based solutions are able to collect high-level resource and network metrics at the application layer and the infrastructure layer without altering the monitored system. While these quantitative metrics are informative of the health of the system components, they do not include any context information. ServiceAnomaly [29] tackles this by combining distributed traces and resource metric monitoring data into a CFG. This method, however, is once again intrusive. Furthermore, the constructed CFGs are limited to the application layer dependencies and do not encompass metrics captured at the infrastructure layer.

D. Positioning

AD for microservice applications has become an active research domain, and many solutions have been proposed and

evaluated on a plethora of benchmark microservice applications. This related work shows high performance when detecting a variety of faults, such as spikes in resource and network usage, and application failures. As explicitly pointed out by ServiceAnomaly [29], all related work in the domain evaluates their methodologies under varying user load, but the topology of the application is always assumed to be static. This poses a strong limitation on the applicability of the aforementioned state-of-the-art research solutions for real-life usage.

Distributed software applications, such as microservices, are characterized by their dynamic nature in order to enable scalability. This means that microservice instances can be added, rescheduled onto different hardware nodes, and removed to optimize resource efficiency and robustness of the overall application. These dynamic changes to the application affect the current state of the system and influence the normal behavior seen in monitoring data.

There is thus a need for methodologies capable of detecting anomalies as the topology of the application changes over time. In order to achieve this, a more advanced benchmark dataset is required that captures these dynamic characteristics and includes contextual features about the monitored application, and the infrastructure it is deployed on.

While some state-of-the-art techniques have leveraged graph structures, such as CFGs to represent relations between different services and components, they were limited to the application layer and contain simple dependency graphs between microservices. In contrast to homogeneous CFGs, a KG is defined as a set of triples $\mathcal{G} = \{(h, r, t)\}$. Each triple is composed of a head entity $h \in V$, a tail entity $t \in V$, and a relationship between them $r \in E$. V denotes the set of vertices, or nodes, in the graph and E denotes the set of edges, or relationships. KGs are high dimensional data structures which can be used to incorporate rich semantic metadata about the heterogeneous entities and their relations.

As previously stated, none of the aforementioned techniques have been applied to dynamic graphs. A *dynamic* graph can be defined as a set of *static* graphs $\mathcal{G}_t(V_t, E_t)$ where each graph $\mathcal{G}_{t_i}(V_{t_i}, E_{t_i})$ represents a snapshot of the dynamic graph at timestamp t_i. This graph format is closely related to the group of temporal KGs and recent temporal KG-based anomaly-detection efforts provide powerful tools to model and detect changes in these time-varying relational data [31], [32], [33]. However, those works are typically developed for symbolic temporal facts and evaluated on offline link prediction or KGs-quality tasks. They do not directly address the practical requirements of microservice monitoring, namely (i) native support for dense numeric monitoring literals (e.g., CPU, memory, latency), (ii) incremental/online updating for low-latency, streaming detection, and (iii) combined application and infrastructure multi-layer semantics. Adapting these temporal KGs and KG-AD techniques to meet these constraints therefore requires nontrivial extensions. To the best of our knowledge, we are the first to consider the dynamic character of microservices and exploit heterogeneous KGs to capture the contextual features about the changing topology in both the application as well as the infrastructure layer.

TABLE I
COMPARISON OF PUBLIC BENCHMARKS AND OUR DATASET. "+" = FULLY SUPPORTED, "–" = NOT SUPPORTED, AND "±" = PARTIALLY SUPPORTED

	Static	Multi-layer	Numeric	Streaming
TrainTicket [25]	+	-	±	-
TeaStore [30]	+	-	±	-
SockShop [28]	+	-	±	-
Acme Air [15]	+	-	-	-
VAST [17]	+	-	-	-
Custom lab [26]	±	±	±	±
This work (ours)	**-**	**+**	**+**	**+**

Contrary to related work, this paper therefore approaches the problem of AD for microservices from a different perspective and emphasizes the impact and complexity of the inherent dynamic nature of microservice applications.

To make the differences explicit, Table I summarizes a set of features that are relevant for microservice anomaly-detection research. Here, Static refers to whether the benchmark assumes a fixed service topology. Multi-layer indicates whether both application- and infrastructure-level data are captured. Numeric denotes the inclusion of time-aligned quantitative monitoring metrics and Streaming expresses support for continuous, online data collection and evaluation. In contrast to the existing benchmarks, our approach and benchmark captures all of these aspects, representing dynamic topologies, multi-layer contextual features, rich numeric metrics and streaming behaviour. Thereby enabling the proper evaluation of anomaly detection approaches in conditions that more closely reflect real-world microservice systems.

III. BENCHMARK ENVIRONMENT

The benchmark dataset contains non-intrusive monitoring data from different sources which capture the relationships between microservices. The infrastructure they are running on makes it possible to trace the dynamic changes the microservice application undergoes throughout time. This section describes the architecture used to collect this dataset and its most important components, such as load generation and fault injection, as well as the different measures taken to ensure it is representative of real-world microservice applications.

The lab environment is deployed on our Fed4Fire testbed [34], which makes a large amount of hardware resources available for research. Seven nodes are used to deploy and configure a Kubernetes cluster. Each node has a quad-core Intel Xeon E3-1220 v3 CPU (4 cores, 4 threads), 16 GB RAM and a 250 GB HDD. The node therefore exposes 4 logical CPUs

A. Architecture

The proposed architecture consists of two layers: (i) an application layer, consisting solely of application microservices required for the functioning of the monitored system, and (ii) a control layer, consisting of all components used to enable monitoring, chaos engineering and data collection. To ensure that there is no interference or residual monitoring data from the control layer services, the hardware nodes are separated

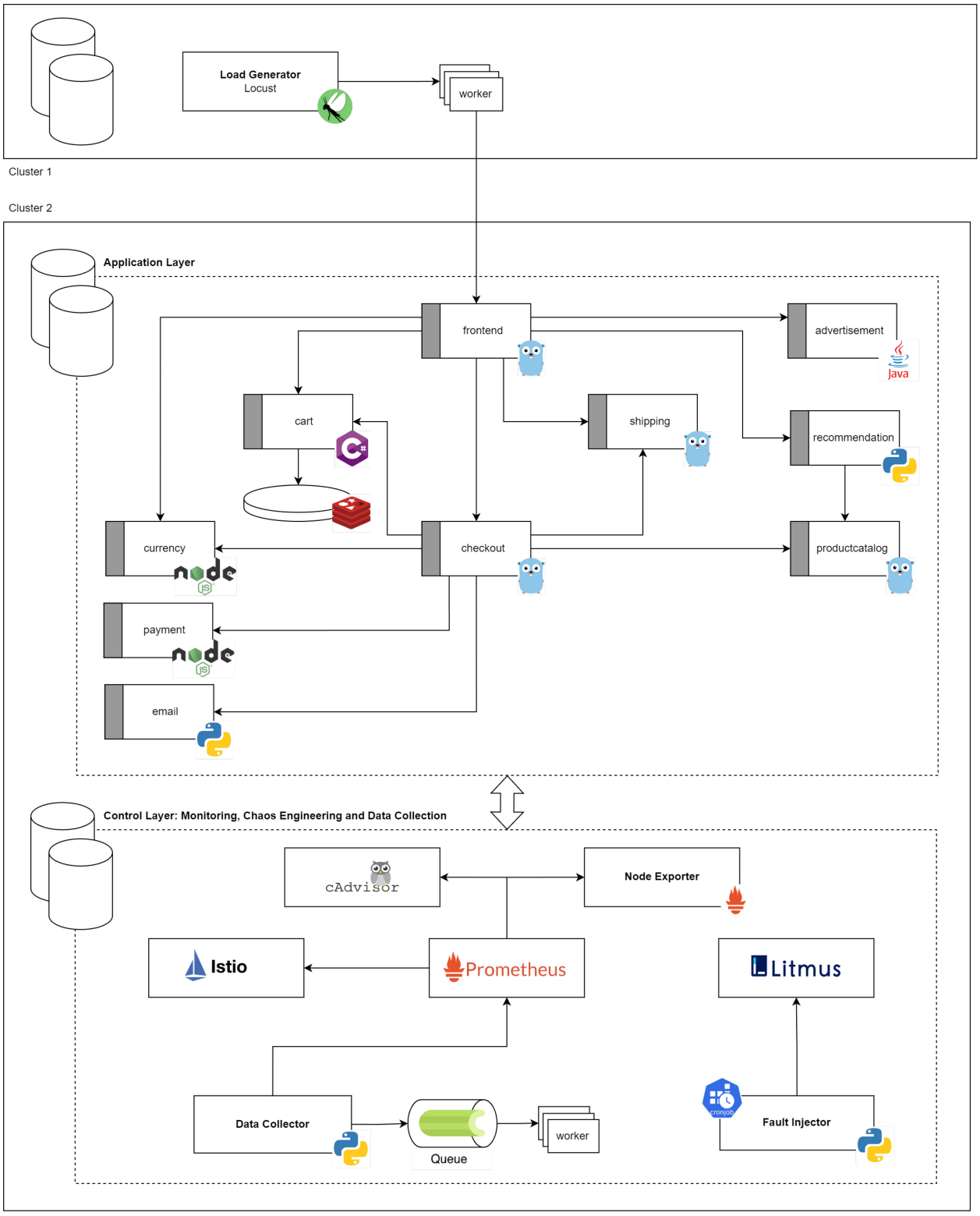


Fig. 1. Complete architecture of the proposed solution including the application layer (i.e., Online Boutique microservices demonstrator), the control layer consisting of monitoring/control components (e.g., Prometheus, Istio), the chaos engineering platform (i.e., Litmus) and custom components for load generation, data collection and fault injection. cAdvisor and Prometheus Node Exporter are deployed on the application nodes as local metric-collection agents and are shown within the control layer only as a logical abstraction for readability purposes. In the application layer, the gray boxes denote the Istio-injected sidecar proxy co-located with the corresponding microservice.

between the two layers. A subset of four nodes is used to run the application components, while the other three nodes host the control-layer services. The complete architecture is shown in Fig. 1. For metric collection, the Prometheus Node Exporter and cAdvisor run on the application nodes they monitor but in Fig. 1 they are shown within the control layer as a logical

abstraction to highlight their role rather than their physical placement. Both the application and control layer are discussed throughout this section with a detailed description of their most relevant components.

The monitoring and injection components of our architecture rely on widely adopted, production-grade tools and will be discussed more in depth in the following subsections. Using these established tools ensures fidelity to real deployments and eases reproducibility. The end-to-end design choices that enable research on dynamic anomaly detection.

1) Application Layer: To showcase the potential of the proposed solution for microservice-based applications independent of the use case domain, a representative benchmark for microservices is used as application layer. As mentioned ind Section II, a number of benchmark microservice applications have been used in literature, e.g., Train Ticket [25], Tea Store [30], and Sock Shop [28]. In this paper, the Online Boutique [35] was chosen to build upon, which is a cloud-first microservices demo application which has been used by Google to demonstrate technologies, such as Kubernetes, Google Kubernetes Environment (GKE), Istio, Google Cloud Operations (Stackdriver), and gRPC. It represents an e-commerce platform and consists of 10 microservices written in Golang, Python, C#, Node.JS and Java.

2) Control Layer: The control layer consists of a number of components used for monitoring and data collection, as well as chaos engineering and fault injection.

Node Exporter is a Prometheus component designed to monitor the host system [6]. It is commonly used within a Kubernetes environment to monitor the nodes within the cluster. It collects valuable metrics about the CPU, memory and file system usage of each node. The resulting data is exposed on an HTTP endpoint, collected by Prometheus and persisted in a time-series database.

Container Advisor (cAdvisor) is a component developed by Google to monitor the resource usage and performance characteristics of Docker containers [36]. It collects, aggregates, processes and exports information about these containers at run-time, including their CPU and memory usage. While the Prometheus Node Exporter focuses on node-level metrics, cAdvisor solves the observability challenge at container-level.

Istio is an open-source service mesh technology based on Kubernetes container orchestration [37]. A service mesh is a dedicated infrastructure layer that can be added to the application to enable features, such as observability (e.g., network tracing), traffic management (e.g., load balancing), and security [38]. It divides the application in two layers. The first layer, called the data plane, consists of the application specific microservices and a number of network proxies. These proxies guide and manage all service-to-service communication. The second layer, indicated as the control plane, consists of services dedicated to performing the service mesh specific tasks, such as managing and configuring the network proxies. The Istio control plane consists of three services: (i) the pilot, which communicates with the proxies to handle traffic management and routing, (ii) the citadel, which provides secure communication between services, and (iii) the galley, which is responsible for configuration management, distribution and processing. The Istio control plane deploys Envoy network proxies as sidecar proxies [39], which run in the same Kubernetes Pod as the application container and therefore on the same node/device. This way communication between the proxy and the service is local and does not influence the network. By deploying network proxies as sidecar proxies, the service mesh can be seamlessly applied to any cloud native application without altering the underlying microservices.

B. Load Generation

Locust [40] is an open-source load testing tool, designed and implemented for scalability, that supports running distributed load tests over multiple machines. It enables developers and Site Reliability Engineering (SRE) engineers to define user behaviour in Python code and swarm their system with millions of simultaneous users. While Locust is built towards load testing over a fixed period of time, its flexibility allows extension towards continuous load generation in order to collect realistic monitoring data under load.

The Online Boutique benchmark microservice application ships with a default load generation scenario. However, to ensure the generalization of the proposed solution beyond the benchmark environment, additional measures are taken to create a more realistic situation. First, as seen in Fig. 1, the **Load Generator** is deployed on a second Kubernetes cluster to simulate user traffic from outside of the target cluster. Second, different user scenarios are defined to represent a variety of users (e.g., browsing the catalogue, actively shopping). Third, six sequential steps were identified to describe a user's journey throughout the system, affecting all microservices in the application. A Markov chain is created to describe these different stages and the probabilities between the steps. This way, we introduce the randomness and complexity that can be observed in real microservice applications. The complete chain is visualized in Fig. 2. Lastly, the temporal variation in user load throughout the day is replicated using a custom distribution over 24 hours. The distribution of simultaneous users and the resulting throughput captured at the front-end of the application is shown in Fig. 3.

C. Data Collection

As described in Section III-A2, three categories of monitoring are available in the proposed architecture: (i) node-level metrics captured by the Prometheus Node Exporter including resource usage of the different nodes in the Kubernetes cluster, e.g., CPU, memory and file system usage, (ii) container-level metrics captured by cAdvisor including resource usage of the different containers within the monitored application, e.g., CPU and memory usage, and (iii) service-level metrics provided through the Istio service mesh capturing monitoring data between microservices at the network layer, e.g., number of requests, and request duration. A complete overview of the monitoring features captured in the dataset is provided in Table II.

The **Data Collector** derives the topology graph by querying the Kubernetes API and obtaining service dependencies from

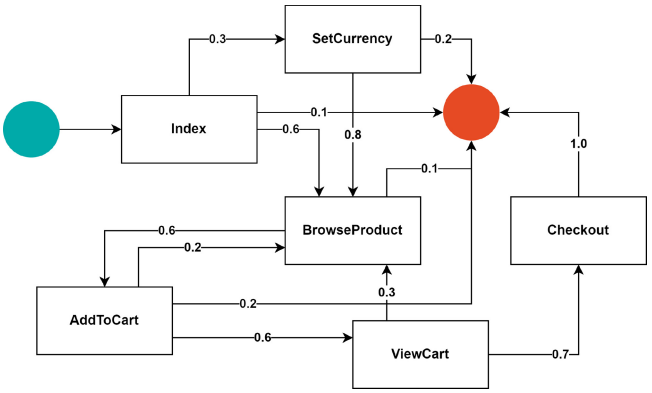


Fig. 2. A Markov Chain that describes the realistic user scenario used for load testing, including six unique stages of the user journey starting from the moment the user visits the Web application (teal dot) until they close the page (red dot). The outgoing edge weights are the probabilities of the next step being chosen during the random walk.

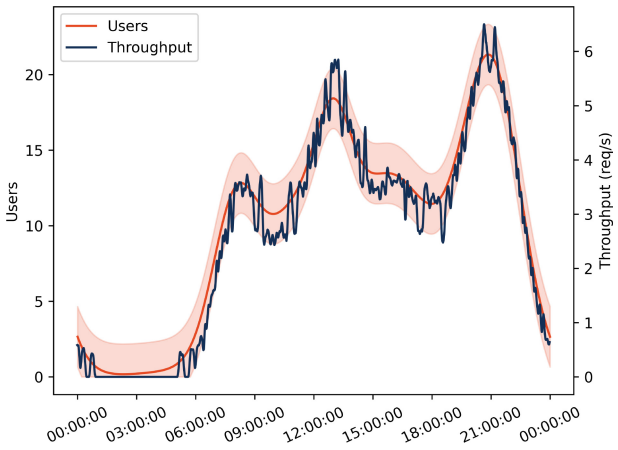


Fig. 3. Visualization of the load distribution shape (number of users) and the resulting captured throughput (requests/s) at the front-end service.

the network monitoring data captured by Istio. It then collects the monitoring data for all nodes and containers in the system from the Prometheus API using the Prometheus Query Language (PromQL), which enables querying of aggregated data. For each of the aforementioned metrics, which can be queried as distributions, the following aggregations are calculated over the last windows of five minutes: minimum, maximum, average, median (P50), P90, P95, deviation and variance. We aggregate monitoring metrics over five-minute windows to balance responsiveness with stability in highly dynamic microservice environments. This design choice is consistent with operational expectations that in production microservice settings, anomalies typically need to be detected within 2–5 minutes to prevent cascading failures and minimize service impact [41]. Our five-minute aggregation therefore targets this practical incident-response horizon while reducing short-lived volatility introduced by autoscaling and rescheduling. Shorter aggregation windows can increase sensitivity but may also amplify transient spikes and increase false positives, while longer windows risk delaying detection. In practice, the ideal aggregation window is likely a tunable deployment parameter that depends on the application domain, workload

TABLE II
MONITORING FEATURES CAPTURED IN THE DATASET, INCLUDING RESOURCE USAGE METRICS FROM PROMETHEUS NODE EXPORTER AND CADVISOR, AS WELL AS NETWORK LAYER MONITORING FROM ISTIO

Source	Feature	Type
Node Exporter	Node CPU Usage (in seconds)	Distribution
	Node Memory Usage (in bytes)	Distribution
	Node FileSystem Usage (in bytes)	Distribution
cAdvisor	Container CPU Usage (in seconds)	Distribution
	Container Memory Usage (in bytes)	Distribution
Istio	Request Count	Counter
	Request Count (4XX)	Counter
	Request Count (5XX)	Counter
	Request Duration (in milliseconds)	Distribution
	Request/Response Size (in bytes)	Distribution

patterns, and tolerance for false positives versus detection delay. The aggregation window is configurable in our pipeline and can be adjusted for deployments requiring faster or more conservative alerting.

The Data Collector component is implemented in Python, using Celery [42] to enable distributed processing of (asynchronous) tasks through a task queue or message broker, such as Redis or RabbitMQ. Every 15 seconds, i.e., the default rate at which Prometheus scrapes new monitoring data from its targets, a task is created to collect the data from the Prometheus API and store it in a Persistent Volume (PV) on the cluster. Using distributed processes allows the horizontal scaling of worker instances and ensures that no window is missed when the execution time of the data collection process would exceed 15 seconds.

D. Fault Injection

Litmus Chaos is a chaos engineering platform for SRE that enables the identification of weaknesses in applications and infrastructures [43]. Other chaos engineering solutions include (i) managed services through cloud providers, such as Amazon Web Services (AWS) Fault Injection [44] and Azure Chaos Studio [45], (ii) commercial platforms, such as Gremlin [46], and (iii) open-source projects, such as Chaos Mesh [47]. We choose to employ Litmus Chaos as it is an open-source project adopted by the Cloud Native Computing Foundation (CNCF) and backed by a strong community. The Litmus Chaos Hub provides 50 community-created experiments out-of-the-box for Kubernetes, AWS, Google Cloud Platform (GCP), Azure, VMWare and Spring Boot.

Similarly to the Istio service mesh, Litmus Chaos consists of a control plane and an execution or application plane. The control plane components include (i) the Chaos Center, which is the front-end dashboard which can be used to create, manage and monitor workflows, (ii) the Authentication Server for authentication with the dashboard and back-end APIs, and (iii) the GraphQL Server, which is used by the Chaos Center for creating, updating and monitoring workflows. The GraphQL Server can also be directly accessed by an external application to automate the chaos engineering process. The execution plane includes components, such as (i) the Argo Workflow Controller [48], an open-source container-native workflow engine for orchestrating parallel jobs on Kubernetes,

TABLE III
LIST OF IMPLEMENTED FAULTS USING LITMUS CHAOS. SEVEN TYPES OF FAULTS HAVE BEEN USED IN THE BENCHMARK. THREE OF THESE FAULT TYPES RESULT IN *Dynamic* CHANGES TO THE SYSTEM AND ITS TOPOLOGY, DENOTED AS FAULT TYPE *D*. THE REMAINING FOUR FAULT TYPES ARE CONSIDERED *Static*, DENOTED AS FAULT TYPE *S*

Fault	Description	Type	Count
Node CPU Hog	Consumes the CPU resources of the targeted node	S	11
Node Drain	Forces rescheduling of the resources running on the targeted node	D	3
Node Memory Hog	Consumes the memory resources of the targeted node	S	4
Pod Autoscaler	Changes the desired number of replicas for the targeted workload	D	6
Pod CPU Hog	Consumes the CPU resources of the targeted application container	S	7
Pod Delete	Causes forced/graceful pod failure of a random replica of the targeted workload	D	4
Pod Network Latency	Injects latency on the targeted application container through a traffic control process	S	4

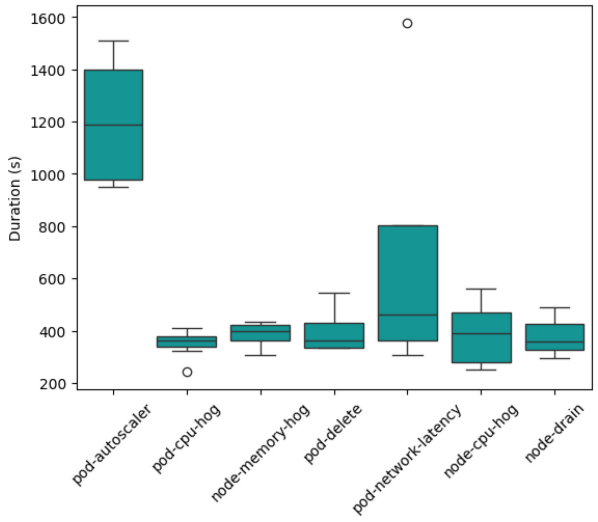


Fig. 4. Box plot summary of the fault injection duration (in seconds) per fault type, providing the time interval during which the fault is actively enforced on the target.

(ii) the Subscriber, or Chaos Agent, which serves as a link between the execution plane and the control plane and performs health checks, monitors events during execution of faults and sends back the results to the control plane.

As summarized in Table III, a total of seven faults are injected using Litmus Chaos. These faults have been chosen to represent both static faults, which directly impact one or more monitored metrics of a specific *Node* or *Pod*, and dynamic faults, which result in temporal changes to the topology of the monitored application. For example, draining a specific node causes all pods running on that node to be rescheduled to other available nodes in the system. While this is common practice before conducting maintenance of a node, it can also be caused by anomalous behavior, such as the node becoming unavailable due to hardware issues or the loss of network connectivity. Similarly, the pod autoscaler in Kubernetes automatically scales the number of replicas based on their resource consumption, resulting in the addition or deletion of pods in the system [49].

The **Fault Injector** is deployed as a Kubernetes cronjob, which is executed every two hours. When triggered, it randomly selects a fault to inject from the list in Table III, a corresponding target, i.e., specific node or pod, and executes the job with an optional delay of maximum one hour to maximize the randomness of the dataset. The Litmus GraphQL API is used to automatically create workflows and collect the experiment results after execution. Similarly to the load generation discussed in Section III-B, all parameters are fully configurable and can be changed to support the creation of additional data with more or less frequent faults. We report the fault injection duration, defined as the time interval during which the experiment actively enforces the fault on the target resource. For example, in pod-network-latency, an additional network latency is applied continuously for the configured duration with a varying range between 150 and 800 seconds in our setup. An overview of all resulting fault statistics is shown in Fig. 4.

IV. KG CONSTRUCTION AND DATASET

The collected data described in Section III-C is heterogeneous and originates from numerous data sources, i.e., Node Exporter, cAdvisor and Istio. This data includes node-level resource metrics (e.g., CPU, memory, filesystem), service-level resource metrics (e.g., CPU, memory) and service-level network metrics (e.g., request count, duration and size). To link the different data sources together in a data format that can be easily understood by both human and machine, we propose the use of an ontology. This ontology imposes a uniform schema that expresses which type of data is generated by which type of sources and how they are interlinked. KGs are then created by applying ontologies (cf. Section IV-A) to annotate the collected data (cf. Section IV).

A. Ontology

An ontology has been designed using the Web Ontology Language (OWL) [50] to describe the key concepts within the Kubernetes ecosystem. It captures relations between different resources, such as nodes, pods and containers, which are used to represent the state of the monitored microservice application. A visualization of the complete ontology is shown in Fig. 5 and the most relevant concepts are discussed throughout this section.

A Kubernetes *Cluster* consists of a set of worker machines, named *Node*s, used to run the containerized application, connected through the *hasNode* relation. When shipping microservices, an *Image* of the application is provided. These images are lightweight, standalone, executable packages that include everything needed to run an application, including the code, runtime, libraries, environment variables, and configuration files. Each (micro-)service of the application runs within a *Container*, a lightweight virtualization layer to ensure

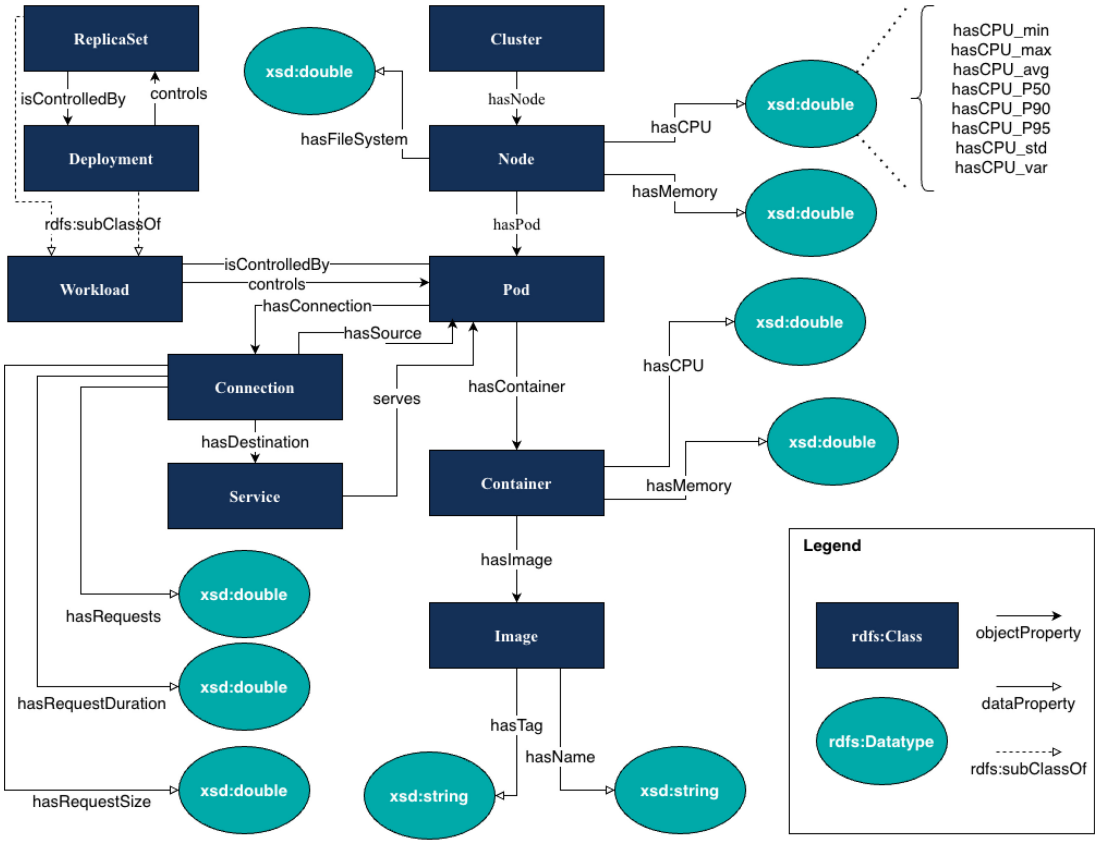


Fig. 5. Ontology describing key concepts in a Kubernetes environment.

the application runs on different host machines, regardless of their specifications or operating system. Each container thus consists of an image making up the application, indicated through the *hasImage* relation. Containers are encapsulated in a *Pod*, which is the smallest deployable unit of computing available within the Kubernetes ecosystem. Each pod can again contain one or more containers, indicated through the *hasContainer* relation, with shared storage and network resources. Best practice dictates that containers within the same pod should be tightly coupled as they will always be co-located and co-scheduled, and are therefore not individually scalable. An example of a tightly coupled container is the sidecar container used to proxy all network traffic to a specific container within a service mesh. Each pod runs on a particular node within the cluster, indicated through the *hasPod* relation between node and pods. A *Deployment* is used for workload management and is thus responsible for keeping a set of pods running, indicated through the *controls* relation to the different pods under its management. Each pod controlled by a deployment is interchangeable and can be updated, added and removed if needed, e.g., in case of a pod failure or when scaling the application. A *Service* is an abstraction within the Kubernetes ecosystem to expose a group of pods over a network. Pods are ephemeral resources, meaning that they have a limited lifespan and should not be expected to be reliable and durable. Services are therefore used to keep track of which pods are available and how to reach them. Additionally, services can also be used for more complex traffic control, such as load balancing.

In addition to the different concepts in the Kubernetes ecosystem, the collected numerical features are also described in the ontology, and included as literals in the graph. These features, as described in Section III-C, are aggregated over a rolling window. To describe the distinct aggregation types, each aggregation is represented as a unique property, e.g., "hasCPU_average".

Kubernetes [51] is functionally the most complete and adopted container orchestration system on the market [52], however, the ontology proposed in this research can easily be extended towards other container-based architectures, such as Docker Swarm [53] or Apache Mesos [54].

B. Dataset and KGs

The resulting dataset comprises of roughly five days of monitoring data. As mentioned in Section III-C, the dataset has a temporal granularity of 15 seconds, resulting in 24,770 total time steps. The KG at each step consists of on average 1402 ± 109 triples. As reference for the healthy state of the monitored system, roughly one day of monitoring data is captured before the start of the fault injection. Using the fault injection configuration discussed in Section III-D, a total of 39 faults have been injected during data collection, resulting in 1,018 anomalous graphs. A detailed statistical overview of the average duration and frequency of the different faults is given in Table III and Fig. 4 to highlight the randomness introduced into the dataset to make it as representative of a real application as possible.

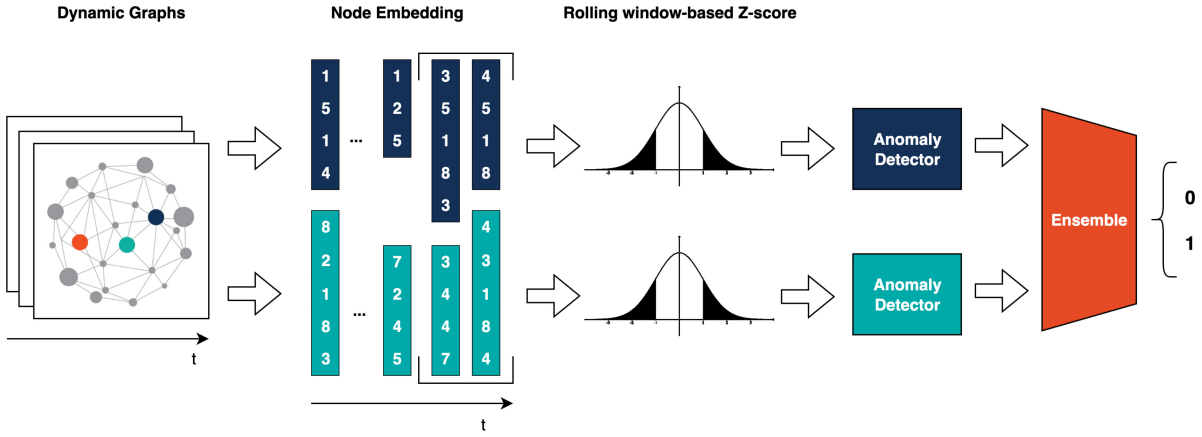


Fig. 6. Anomaly detection pipeline of the proposed methodology. The dark blue and teal colors indicate the difference between embeddings generated from different starting nodes and their respective AD models.

In total, the dataset consists of 250 unique entities and 847 unique relations between them. When taking into account the monitoring data captured in the graphs, a total of 5,715,493 triples are included. The size and complexity of the collected dataset is comparable to existing benchmark datasets for dynamic (knowledge) graphs [55], such as social network graphs commonly used for link prediction tasks. As the data is collected from a microservice application in a research environment, and the complete architecture can be easily deployed, the dataset can easily be expanded upon, and additional faults can be introduced.

V. Context-Aware Anomaly Detection

The captured dataset is particularly challenging due to the dynamic character caused by the evolving state and topology of the monitored system. Anomaly detection models, such as AEs and Isolation Forests [56], often attempt to learn the normal behavior from training data. When the new data changes too much due to temporal drift, or dynamic events in this case, the model will throw too many false positives and thus has to be retrained to deal with the context shift. To prevent frequent retraining of the model, and the need for complex incremental learning strategies, we propose a context-aware anomaly detection method, which takes into account the contextual-features and how they evolve throughout time. The contextual information about the current state of the system is represented in a KG using the ontology described in Section IV-A.

The following sections discuss each component of our proposed solution and countermeasures taken to overcome the challenges with real-time AD based on dynamic graphs. The complete context-aware anomaly detection pipeline is visualized in Fig. 6.

A. Node Embedding

Traditional embedding techniques, such as TransE [57] and RDF2Vec [58], that reduce the dimensionality of the graph into fixed-length embeddings in the latent feature space cannot be applied efficiently to dynamic graphs for real-time or online prediction tasks. This is due to the fact, that for each new graph at timestamp t_i, all the embeddings would have to be recalculated on the entire dataset. This way, the embeddings would have the same length, but the latent features can not be compared, making it impossible to train an AD model on these embeddings.

To overcome this issue, we propose the use of INK, a neighborhood-based node embedding technique [59]. Starting from a set of graphs $\mathcal{G}_t(V_t, E_t)$, INK is applied to each graph $\mathcal{G}_{t_i}(V_{t_i}, E_{t_i})$ in the dataset. INK uses a random walk approach starting from a node of interest n with a predefined depth d to capture contextual information about its neighborhood. This means that starting from node n INK will follow a relation in the graph to a neighboring node (depth 1) and will then continue to follow the relationships from this node to subsequent nodes until the d relationships were crossed (depth d). Based on the example visualized in Fig. 7 (timestamp t), the neighborhood at depth one starting from the cluster node includes features such as:

$$\begin{aligned}\text{is\_a} &\rightarrow [k8s : Cluster]\\ \text{hasNode} &\rightarrow [n1]\\ \text{hasNode} &\rightarrow [n2]\end{aligned}$$

Relations at depths ≥ 2 are appended to the features set by concatenating the steps taking during the random walk. A simplified example for the second hop neighborhood is given by:

$$\begin{aligned}\text{hasNode(n1)is\_a} &\rightarrow [k8s : Node]\\ \text{hasNode(n1)hasMemory\_P50} &\rightarrow 0.173431\\ \text{hasNode(n1)hasPod} &\rightarrow [frontend]\\ \text{hasNode(n2)is\_a} &\rightarrow [k8s : Node]\\ \text{hasNode(n2)hasMemory\_P50} &\rightarrow 0.120647\\ \text{hasNode(n2)hasPod} &\rightarrow [cartservice]\end{aligned}$$

To describe the neighborhood of the node of interest n, INK extracts binary features representing the relations in the KG, i.e., indicating whether this relation is present in the neighborhood or not. As INK generates node embeddings using explicit features, the features can be traced throughout

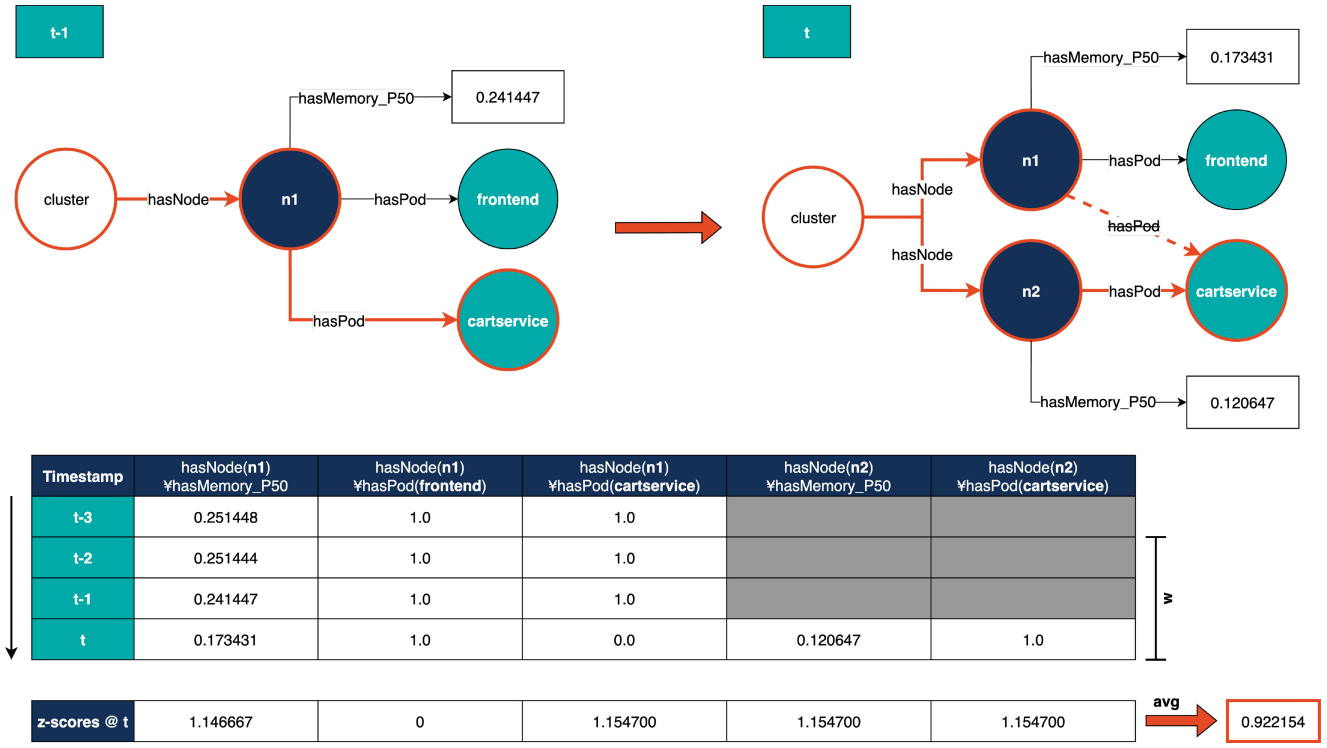


Fig. 7. An example of generated INK embeddings starting from the *cluster* with depth 2, and the average z-score calculated at the last timestamp based on a sample window size. The example between timestamps t and $t-1$ show the addition of a Kubernetes node in the cluster which results in the running pods being rescheduled, causing relationships to be added and removed (dotted line) in the KG.

time. New features are created and deleted as the relationships in the graph are added or removed. This ensures that not all embeddings have to be recalculated each time a new graph becomes available at timestamp t_{i+1}, solving the shortcoming of other node embedding approaches. Furthermore, having explicit features makes the INK embeddings interpretable and usable for root cause analysis [60].

B. Anomaly Detection

Contrary to the other node embeddings, generating INK embeddings for each graph at timestamp t_i results in a set of node embeddings of variable length as here the feature set continuously expands over time. This makes it impossible to use INK embeddings directly as input for ML models, such as an AE or Isolation Forest (IF). To address this issue, the features are aggregated over a rolling window. For each feature extracted from the graphs in the window, the Z-score is calculated per feature based on its value X at timestamp t_i, the mean μ and standard deviation σ of all values of the feature in the window as shown in Equation (1).

$$Z = |\frac{X - \mu}{\sigma}| \tag{1}$$

The Z-scores for all features in the window are then aggregated, resulting in a fixed set of features for the new graph $\mathcal{G}_{t_i}$, i.e., the last graph in the rolling window. This process is visualized in Fig. 7. As these Z-scores are calculated on INK embeddings including both relations in the graph as well as the raw monitoring metrics, it provides an indication of the similarity of the new graph in comparison to the graphs in the considered window. These graph similarity scores are then used as input for the anomaly detector.

While there are a multitude of univariate time-series AD models, three models are considered here, namely a percentile-based detector (QuantileAD), a threshold-based detector (ThresholdAD) and a local outlier detector (OutlierAD), which is a density-based technique for outlier identification [61]. These detectors have been implemented using the Anomaly Detection Toolkit (ADTK), a Python package for unsupervised and rule-based time series anomaly detection [62]. These detectors were chosen intentionally as the baseline methods for our proposed benchmark dataset rather than as a comprehensive survey of all AD families. They provide a practical, reproducible reference point that operates directly on the aggregated Z-score representation derived from INK embeddings while being lightweight and suitable for online streaming cases. The benchmark itself is designed in a generic, modular way so that additional detection methods can be evaluated against the same data and metrics, facilitating future extensions and comparisons.

The hyper-parameters for the threshold-based detector and local outlier detector are dynamically optimized with a grid-search approach, which uses the validation dataset containing a mixture of healthy and unhealthy samples to determine the best hyper-parameters. The percentile-based detector is an unsupervised model that simply uses a Gaussian distribution at prediction time to determine if the new data point at timestamp t_i lies within the high-confidence interval. If this is the case, the new graph is assumed to be normal, otherwise it is flagged as anomalous.

C. Ensemble

Due to the complexity of the expanding feature set generated by using INK in a real-time detection setting, the high dimensional features are reduced to a single similarity score calculated at timestamp t_i, e.g., the average of the rolling window-based Z-scores. This results in a reduction of $\pm$ 3200 features to a single feature when generating embeddings for the entire graph ($d = 5$). To do so while avoiding this amount of information loss, we propose generating embeddings from different starting nodes. From the ontology described in Section IV-A and expert knowledge about Kubernetes environments, it can be deduced that the instances of *k8s:Cluster* and *k8s:Workload* are ideal candidates to be considered as starting nodes. In contrast to nodes and pods, clusters and workloads remain mostly static, meaning that their instances are available in all graphs and can thus be traced through time. To capture all information from the graph while minimizing the information loss due to feature reduction, the embeddings are generated for a neighborhood depth of 3 ($d = 3$). Using this smaller neighborhood compared to capturing the entire graph, an average of only 286 features are generated starting from an instance of *k8s:Cluster*.

For our deployment composed of one Kubernetes cluster and ten microservices, we constructed an ensemble of eleven models: one ThresholdAD model per monitored entity and the cluster anomaly voter. The use of ThresholdAD models across all microservices for this ensemble keeps the pipeline lightweight, explainable, and fully reproducible. Each model operates on the aggregated Z-score features derived from the INK embeddings within its own entity scope and applies a dynamically optimized threshold to flag abnormal behaviour as discussed before. The voter combines the ten model outputs using a logical OR rule: as soon as one per-entity detector signals an anomaly, the ensemble flags the system as anomalous. This aggregation prioritizes high recall and early fault exposure, which is desirable in this production monitoring context where delayed detection is more costly than an occasional false positive. Moreover, the alarms originate from per-entity models and each alert is directly tied to the corresponding service or cluster node in the KG, enabling straightforward root-cause inspection

VI. Evaluation

The proposed solution is evaluated against two baseline AD techniques commonly used in related work (see Section II), namely an LSTM-AE and an IF. For training, optimization and evaluation, the dataset has been split in a healthy training set, an unhealthy training set, and a test set including both healthy and unhealthy test samples. The same 20-40-40 split is used for all considered models so that half of the injected faults are present in the unhealthy training set and the other half are used during evaluation. The complete quantitative results for each model are shown in Table IV and discussed throughout the remainder of this section.

Baseline. To highlight the impact of contextual awareness when dealing with the dynamic nature of microservice applications, the baseline AD techniques. i.e., LSTM-AE and IF, are trained on two distinct feature sets. The first feature set contains static features inspired by Nobre et al. [27] who trained a Multi-Layer Perceptron (MLP) model on aggregated application-level and service-level metrics. These features include the mean, 50th percentile and 95th percentile of the latency, and the amount of 2xx and of 4xx/5xx requests reported by the Locust during load generation as discussed in Section III-B. These features are aggregated over the different endpoints accessed during the load test, resulting in five features. Since these features are measured at the endpoint of the application, i.e., the front-end, the feature set remains the same throughout the entire dataset. The second feature set includes dynamic relations between entities in the graph through features extracted through the INK embedding, as described in Section V-A. The dynamic graphs therefore result in the addition or removal of features throughout time. To evaluate the importance of these dynamic features, the two baseline AD techniques are also applied on these INK embeddings in an offline setting. In this offline setting, we can train the models on the complete set of features by setting them to 0 if they are not yet present or no longer present at timestamp t.

TABLE IV
Performance Evaluation of Two Baseline Models From Literature, i.e., an LSTM-AE Model and an IF Model, Compared to Our Context-Aware Approaches as Proposed in This Paper

Model	Precision	Recall	F1
(Online) Baseline			
LSTM-AE	0.174	0.200	0.186
IF	0.125	0.263	0.169
(Offline) Baseline			
INK-LSTM-AE	0.123	1.000	0.218
INK-IF	0.327	0.895	0.466
Context-Aware			
INK-Z-IF	0.209	1.000	0.345
INK-Z-QuantileAD (P95)	0.818	0.474	0.600
INK-Z-QuantileAD (P90)	0.484	0.789	0.600
INK-Z-OutlierAD	0.333	0.800	0.471
INK-Z-ThresholdAD	**0.786**	0.579	**0.667**
Ensemble			
Log. OR INK-Z-ThresholdAD	0.481	0.684	0.565
Log. OR INK-Z-ThresholdAD extra	0.283	**0.895**	0.430

The resulting models on this second feature set, INK-LSTM-AE and INK-IF, can however not be deployed for online anomaly detection and are thus solely considered for performance evaluation purposes. We implemented the INK-LSTM-AE model with a symmetric architecture. The encoder consists of two LSTM layers with 256 and 128 units, followed by a bottleneck layer or latent space of size 64. The decoder mirrors this structure with two LSTM layers of 128 and 256 units, reconstructing the input sequence. The model was trained using the Adam optimizer, a learning rate of 0.001 and a Mean Absolute Error (MAE) loss. We then train a Linear Support Vector Classifier (SVC) on the validation losses to derive the threshold. The hyperparameters of the INK-IF model were achieved through GridSearchCV on the training and validation datasets.

As already stated, a first evaluated baseline model is the LSTM-AE approach which learns the normal state of the system based on the healthy training set. The quantitative

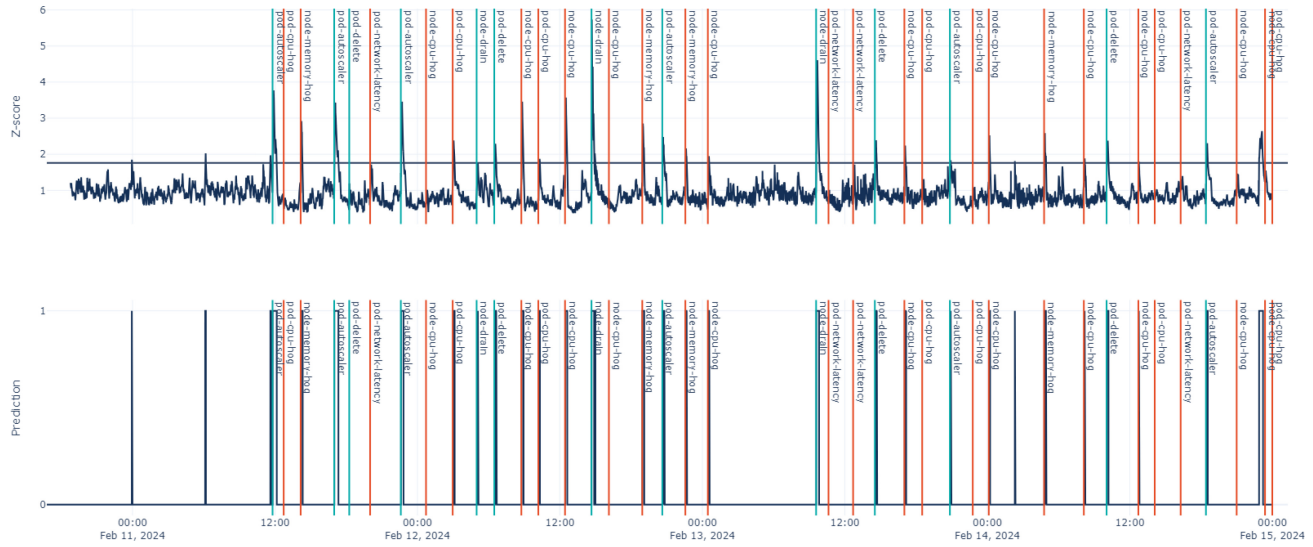


Fig. 8. Calculated Z-scores for a rolling window of two hours (480 graphs) and the corresponding predictions of the INK-Z-ThresholdAD model with a threshold derived from grid search optimization on the validation dataset. The teal vertical lines are considered dynamic faults, the red lines indicate static faults.

evaluation shows that the model is unable to generalize this healthy state of the system towards the rest of the dataset. Due to the dynamic faults, the topology of the system changes constantly as pods are added, updated and removed during scaling or rescheduling. The impact of these events results in a temporal shift in the state of the system, making it challenging for the AE model to match the newly observed state with the learned normal behavior during training. In practice, this means that reconstruction-based models can misinterpret legitimate topology shifts, such as rescheduling without user-visible degradation, as anomalies. This results in missing faults whose primary signal manifests through changing infrastructure-to-service context rather than a strong local metric deviation. As can be noted from the results, even with the dynamic features, the INK-LSTM-AE model struggles to generalize the changes in context.

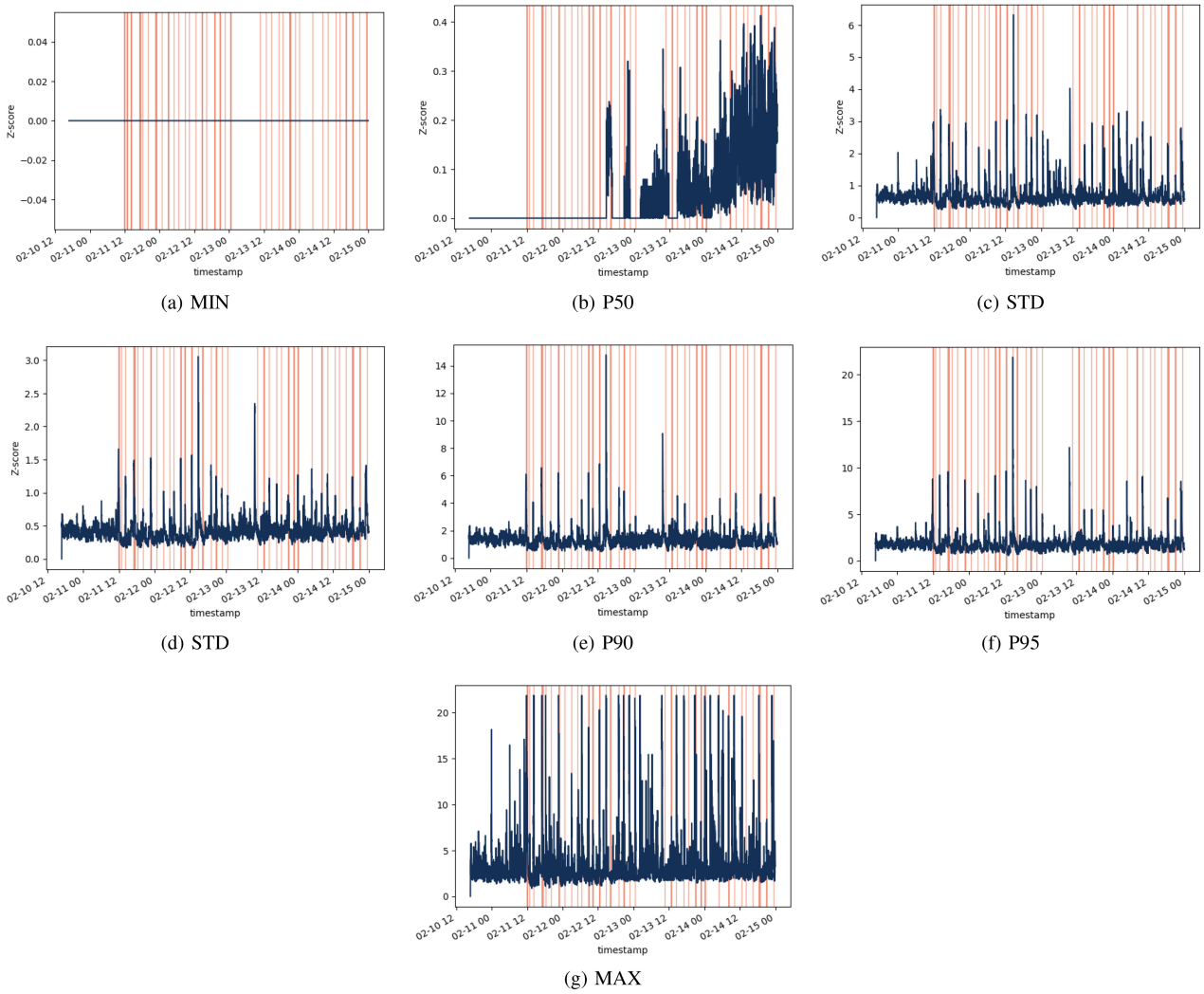
Contrary, the INK-IF model shows a noteworthy improvement of 29.7% absolute increases in F1 compared to its online counterpart IF model trained on solely static features. Especially in terms of recall, the INK-IF model detects 89.5% of the faults in the dataset while the baseline IF model only detects 26.3%. After investigation of the detected faults, it becomes clear that the IF model is primarily capable of detecting a limited number of static faults related to network latency, such as *pod-network-latency* while the context-aware version is able to detect nearly all faults in the dataset, including dynamic faults.

Context-Aware AD. Our proposed context-aware solutions compute a rolling window-based Z-score on features extracted using INK, which are then aggregated using various functions, as listed in Fig. 9. Initially, all aggregated features, including minimum, mean (P50), standard deviation, average, 90th percentile (P90), 95th percentile (P95) and maximum, were used as input for the IF model, resulting in the INK-Z-IF model achieving an F1-score of 0.345 after hyperparameter tuning. While this model outperforms its online counterpart based on the baseline feature set, indicating the value of contextual features for anomaly detection, Fig. 9 shows that certain features, such as minimum and P50, contribute little information, while others, such as the maximum, are overly sensitive. The limited informativeness and redundancy within this feature set reduce the model's ability to effectively isolate anomalies, as IF relies on recursive partitioning, which becomes less effective in low-dimensional spaces. This constraint is evident from the model's relatively low performance, suggesting that a more complex approach may not be justified for such a restricted feature set.

As described in Section V-B, to further assess the impact of feature selection, a set of simpler anomaly detectors, i.e., QuantileAD, ThresholdAD, and OutlierAD, were evaluated using a single aggregated feature. Given the limitations of training an IF on a restricted feature set, these simpler methods provide a more targeted approach to anomaly detection. After quantitative evaluation of all the aggregation functions, the *average* emerged as the most effective feature, highlighting the potential for reduced model complexity.

The QuantileAD model is representative of an unsupervised rule-based detection model that uses a confidence interval to detect outliers. The choice of interval influences the obtained performance. Table IV shows the performance results for the 95% percentile (P95) and 90% percentile (P90) confidence intervals. As can be seen, both context-aware model configurations yield an identical F1-score. As expected, the higher the confidence interval, the higher the obtained precision due to a low report of false positives, but the lower the obtained recall due to the lack of detected anomalies, or true positives, and vice versa.

In contrast to the QuantileAD, the ThresholdAD model optimizes its hyper-parameters, i.e., threshold values, based on the validation set. The tuned INK-Z-ThresholdAD model shows a high precision of 78.6% while reporting a recall of 57.9% which is an improvement of 31.6% over the online



(a) MIN (b) P50 (c) STD

(d) STD (e) P90 (f) P95

(g) MAX

Fig. 9. Visualization of different aggregation features on the derived Z-scores for INK embedding depth $d = 2$ and Z-score window size $w = 240$, the best performing settings for the INK-Z-ThresholdAD model. For each figure, the faults in the dataset are highlighted in red.

baseline IF model. Fig. 8 shows a graphical representation of the rolling window-based Z-scores on the dataset and the obtained results for the proposed INK-Z-ThresholdAD model. Investigation of the results shows that the detected anomalies are mostly node-level faults (e.g., *node-cpu-hog* and *node-drain*) or pod-level faults that influence the state of a node (e.g., *pod-autoscaling*). As discussed in Section V-C, these results are to be expected as the INK-Z-ThresholdAD model is trained on features generated from INK embedding at a neighborhood depth of 2 ($d = 2$). The full impact of the INK embedding depth parameter d and Z-score window size w for each of these models is visualized in Figure 10. At the best performing settings ($d = 2$ and $w = 240$), not all pod features are taken into account.

Ensemble. In addition to these context-aware models, a logical OR ensemble is therefore also evaluated for the INK-Z-ThresholdAD technique. Inclusion of additional models trained on the neighborhood of specific workloads show an increase in recall, enabling detection of previously undiscovered faults. The ensemble is thus capable of capturing pod-level faults that the base model could not detect. This increase in recall, however, comes at a cost of the overall precision. In our AIOps setting, missing a false negative is therefore indicated to be more costly than raising an extra alert (false positive). Operators can aggregate, de-prioritize or tune away noisy alarms over time, but they might have difficulties to manually detect new ones. We acknowledge that this aggregation strategy lowers the precision when a single entity-level model is temporarily unstable. More sophisticated fusion strategies could mitigate this effect, including (i) majority voting across entity detectors, (ii) weighted voting based on per entity validation performance, or (iii) score-level fusion where the normalized anomaly scores are combined before thresholding. We see these strategies as natural extensions of our modular pipeline and expect them to potentially further improve the precision–recall balance in future work.

VII. Discussion and Future Work

Our whole evaluation has been conducted on the Online Boutique microservices application, which serves here as a representative open-source reference for cloud-native systems.

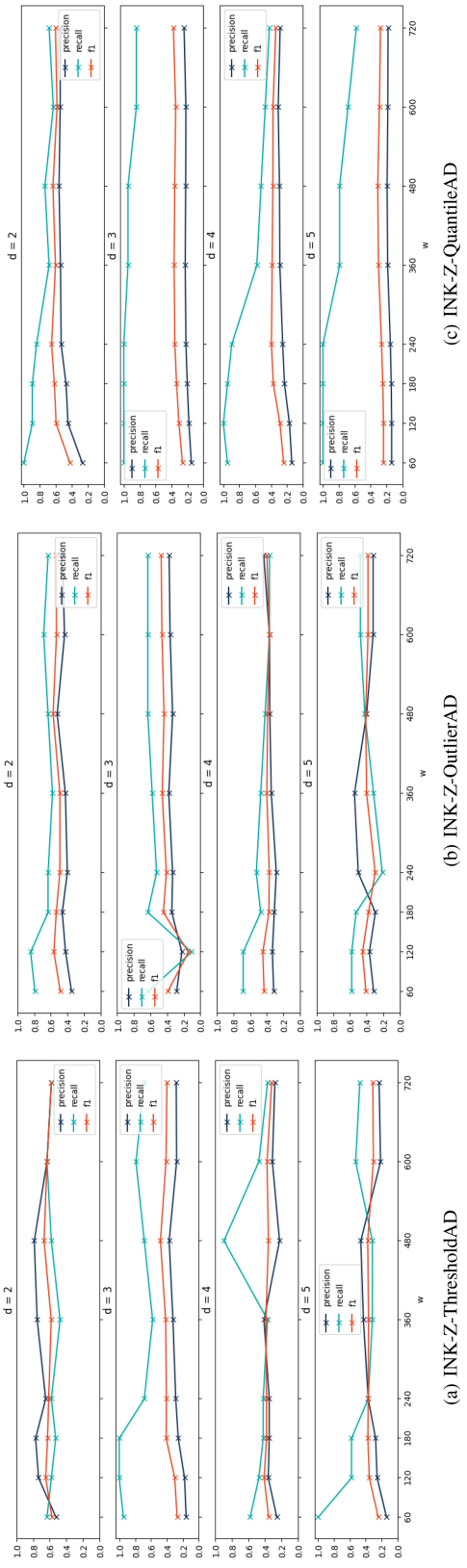


Fig. 10. Performance of various models under varying INK embedding depth parameters d and Z-score window sizes w.

This environment combines a realistic service topology, multiple programming languages, and standard observability tooling. This made it well suited for evaluating our AD approaches. The obtained results are applicable to this single benchmark, which might limit the generalizability of the quantitative results. However, our objective was to provide a reproducible, fully instrumented framework and dataset that others can readily extend to additional benchmarks or larger-scale deployments. Our open-source release of the entire pipeline ensures that future studies can build upon this foundation to perform broader comparative analyses. The evaluation results highlight the complexity of this dataset, but also the importance of contextual features. In contrast to the state-of-the-art that focuses on detecting static faults solely, this paper has introduced a benchmark that also contains dynamic faults, such as node-level anomalies, that force scaling and rescheduling of the pods running on it. These faults do not always translate into service-level changes, e.g., a noticeable network delay or sudden spike in resource usage, and are therefore difficult to detect.

Whereas state-of-the-art anomaly detection methods fail to take into account the dynamic character of microservice architectures (included in our benchmark), our proposed solutions score significantly higher than the models trained on purely static features, or the AE-based model that attempts to learn the healthy state of the system. Our baseline AD detector family focuses on lightweight, unsupervised, and explainable methods, providing a reproducible foundation for operational root-cause workflows. Future research could extend our INK features with ensemble and kernel methods (e.g., Isolation Forest, One-Class SVM), deep reconstruction and probabilistic models (AE, VAE), temporal sequence models (LSTM-AE), and graph-aware detectors [63]. Hybrid pipelines combining fast rule-based detection with higher-precision models and explainability tools, such as feature-attribution mapped to knowledge graph entities, could further improve detection quality while preserving interpretability [64].

Despite room for improvement of our preliminary evaluation results, they highlight the need for contextual awareness in microservice AD. The presented solution and its evaluation show an important trade-off between precision and recall. The best performing model has an F1-score of 66.7%. This model, however, is only capable of detecting 57.9% of the faults in the dataset. These results reflect the high difficulty of our benchmark, which contains many dynamic, topology-driven faults (node drains, rescheduling, autoscaling) that often produce weak service-level signals. We intentionally tuned our INK-Z-ThresholdAD model towards higher precision (0.786) to reduce operational false alarms. This lowers, however, the recall score (0.579) and thus the combined F1. Additionally, these moderate absolute F1 scores stem from online deployability constraints and not from a lack of benefit from context-aware features. The detected faults, as discussed in Section VI, are limited to node-level faults. By introducing an ensemble of multiple small models for which the node embeddings are extracted from different start entities of interest in the KG, an increase in recall can be achieved up to 89.5%, however, at a cost of precision. The smaller models are less robust due to their training on a limited number of specific faults relating to that entity. We emphasize that the absolute score of our best online model

(INK-Z-ThresholdAD) should be interpreted in the context of a challenging benchmark that includes topology-driven faults (e.g., node drain, rescheduling, autoscaling) which often generate weak or delayed service-level symptoms. We stated here our approach as a context-aware monitoring layer that prioritizes reliable, interpretable signals over purely maximizing a single performance metric. To mitigate false positive fatigue in real deployments, the proposed pipeline can be combined with operational strategies such as (i) requiring anomaly persistence across multiple consecutive windows before triggering an alert, (ii) introducing per-entity cooldown periods, and (iii) using confidence-tiered alerting where low-confidence anomalies are logged for investigation rather than immediately escalated [6].[1] These measures can be combined with our method and can be applied without altering the KG construction or the INK feature extraction. Future work includes techniques to increase the precision of the whole proposed anomaly detection framework.

The architecture used for monitoring, fault injection and data collection is described in detail. Since it consists of purely open-source technologies that can easily be deployed on a Kubernetes environment, it can be replicated and expanded upon. Furthermore, this open-source solution has been made publicly available on GitHub[2] and can be used to expand upon the dataset. Due to its complexity, the dataset can serve as an interesting benchmark for future dynamic graph research, and the extension of existing Knowledge Graph Embedding (KGE) techniques. Since the proposed solution is based on KGs, it can be generalized in future research and applied to other application domains beyond microservice monitoring.

A full evaluation of the overhead introduced by deploying each presented method for real-time anomaly detection has not been included. The scope of this paper is limited to addressing the challenges that arise when applying traditional AD techniques to a highly dynamic environment. There are, however, different methodologies for optimizing real-time AD models and their deployment strategies with regard to resource consumption and overhead on the monitored system [65]. Our pipeline was, however, designed to be lightweight. Each KG snapshot contains ± 1402 triples and the online INK step derives small, explicit neighborhood feature sets from nodes of interest (cluster and workloads). At depth 3, this yields ± 286 features for the cluster entity and across the 10 workloads plus the cluster this corresponds to roughly 3.000 explicit features per timestamp. The calculated Z-score update and accompanying ThresholdAD/QuantileAD/OutlierAD inference are linear in the feature count and they require no retraining. Consequently, the evaluated online boutique deployment, the online embedding and detection steps are expected to fit comfortably within the 15s monitoring. The primary runtime cost is thus dominated by data acquisition and aggregation of the results from Prometheus. Our design intentionally avoids global graph recomputation or deep graph model training in the online path to become real-time applicable.

While the online INK extraction itself is bounded by the neighborhood depth and the number of start entities, a practical overhead in our current prototype can be the construction of rolling-window feature matrices. In particular, the aggregated INK feature set per timestamp is sparse and partially time-dependent. Some of the features are stable across time, while others may only occur rarely and thus be unique to specific windows. As a consequence, naively constructing window-level matrices by concatenating per-timestamp embeddings (e.g., dataframe concatenation in a for loop) can introduce avoidable overhead for large window sizes. This cost is not fundamental and can be reduced by maintaining incremental window statistics and sparse feature representations, pre-allocating consistent feature indices and updating window matrices through vectorized or streaming operations instead of recomputing them step-by-step. We therefore expect that a more optimized window-construction strategy will further reduce the runtime footprint of the online pipeline, especially when scaling to larger deployments or longer windows. This optimization, the analysis of the best performing model and a quantitative evaluation of the overhead will be considered in future work.

The proposed solution is currently designed to be minimally invasive and to be deployed out-of-the-box for any application running on a Kubernetes cluster. However, state-of-the-art solutions have shown that monitoring data obtained from distributed traces can positively impact the performance of the AD solution, especially in detecting service-level faults, such as configuration errors and malicious queries. The inclusion of traces, e.g., OpenTracing [12], in the dataset could therefore lead to more fine-grained data which can increase the performance, and even assist with root cause analysis. This, however, will require additional code in each microservice of the targeted software application, contrary to the proposed non-invasive solution which is completely application agnostic.

While quantitative evaluation of explainability and RCA has not been the focus of this work, the use of INK as a node embedding technique inherently supports both [60]. Traditional embedding approaches such as TransE or RDF2Vec project entities into a latent vector space, making it difficult to interpret which relations or features contribute to an anomaly. INK, in contrast, constructs embeddings by explicitly enumerating and weighting walk-based features that correspond to concrete graph relations, such as *has_node(node1)has_cpu_average* or *depends_on(serviceA,serviceB)*. During anomaly detection, the Z-score calculated on these explicit features allows identification of which relations contributed most to the detected deviation. By ranking these features according to their individual Z-scores at anomaly time, the system can highlight the dominant factors that led to the anomaly. Because these features are directly linked to entities and dependencies in the knowledge graph, they can be traced backward through the graph to reveal the likely source or propagation path of the fault, thereby enabling explainable RCA. This mechanism has been successfully demonstrated for static knowledge graphs in earlier work [64]. Our benchmark can extend

[1] https://prometheus.io/docs/alerting/latest/alertmanager/
[2] https://github.com/predict-idlab/microservices-addkg

this interpretability principle to the dynamic graph setting, laying the groundwork for future explainable RCA research on evolving microservice infrastructures.

While this has not been the focus of this paper, the use of INK as node embedding technique, instead of traditional embedding techniques, enables explainability and RCA. Traditional node embedding techniques such as TransE and RDF2Vec calculate embeddings in a latent embedding space. INK, however, uses a walk-based approach to capture the neighborhood of a node, or entity, of interest and generate explicit features. When calculating the Z-score on these explicit features, the features with the highest impact on this score can be extracted to provide an explanation for the detected fault. Furthermore, due to these features being explicit, e.g., *has_node(node1)has_cpu_average*, these features and the relationships captured in the KG can be used to trace the anomaly to its root cause [60].

VIII. Conclusion

This paper introduces a dynamic KG-based approach to AD for AIOps in microservices. It specifically highlights the importance of contextual awareness and demonstrates the shortcomings of state-of-the-art solutions that consider the topology of the monitored system to be static. Overcoming the complexity stemming from the dynamic nature of microservice applications, and research towards dynamic AD solutions is paramount for its adoption in critical domains, such as e-commerce but also for example healthcare. Apart from the proposed solution, a complete architecture has been presented consisting of open-source tools for monitoring, chaos engineering and data collection. It is non-intrusive and can easily be integrated in any Kubernetes-based platform.

The dataset collected during this research has the potential and complexity to serve as a benchmark for future research towards anomaly detection in microservices, and can be used for dynamic graph research beyond microservice monitoring. The presented architecture allows easy extension of this dataset, both in amount of data collected and the addition of fault types.

References

[1] W. Hasselbring and G. Steinacker, "Microservice architectures for scalability, agility and reliability in e-commerce," in *Proc. IEEE Int. Conf. Softw. Archit. Workshops (ICSAW)*, 2017, pp. 243–246.

[2] J. Zaki, S. R. Islam, N. S. Alghamdi, M. Abdullah-Al-Wadud, and K.-S. Kwak, "Introducing cloud-assisted micro-service-based software development framework for healthcare systems," *IEEE Access*, vol. 10, pp. 33332–33348, 2022.

[3] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Softw.*, vol. 35, no. 3, pp. 24–35, May 2018.

[4] M. Usman, S. Ferlin, A. Brunstrom, and J. Taheri, "A survey on observability of distributed edge & container-based microservices," *IEEE Access*, vol. 10, pp. 86904–86919, 2022.

[5] J. Soldani and A. Brogi, "Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey," *ACM Comput. Surv.*, vol. 55, no. 3, pp. 1–39, 2022.

[6] B. Brazil, *Prometheus: Up & Running: Infrastructure and Application Performance Monitoring*. Santa Rosa, CA, USA: O'Reilly Media Inc., 2018.

[7] A. Lahmadi and F. Beck, "Powering monitoring analytics with elk stack," in *Proc. 9th Int. Conf. Auton. Infrastruct., Manag. Security (AIMS)*, 2015, pp. 1–28.

[8] L. Giamattei et al., "Monitoring tools for DevOps and microservices: A systematic grey literature review," *J. Syst. Softw.*, vol. 208, Feb. 2024, Art. no. 111906.

[9] Y. Dang, Q. Lin, and P. Huang, "AIOps: Real-world challenges and research innovations," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng., Companion (ICSE-Companion)*, 2019, pp. 4–5.

[10] "Dynatrace: Unified observability and security." Accessed: Apr. 30, 2024. [Online]. Available: https://www.dynatrace.com/

[11] A. Nandi, A. Mandal, S. Atreja, G. B. Dasgupta, and S. Bhattacharya, "Anomaly detection using program control flow graph mining from execution logs," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discov. Data Min.*, 2016, pp. 215–224.

[12] A. Janes, X. Li, and V. Lenarduzzi, "Open tracing tools: Overview and critical comparison," *J. Syst. Softw.*, vol. 204, Oct. 2023, Art. no. 111793.

[13] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, 2017, pp. 33–40.

[14] T. Jia, L. Yang, P. Chen, Y. Li, F. Meng, and J. Xu, "LogSed: Anomaly diagnosis through mining time-weighted control flow graph in logs," in *Proc. IEEE 10th Int. Conf. Cloud Comput. (CLOUD)*, 2017, pp. 447–455.

[15] "Acme air sample and benchmark." BLUEPERF. Accessed: May 6, 2024. [Online]. Available: https://willemjiang.github.io/acmeair/

[16] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly detection and diagnosis from system logs through deep learning," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 1285–1298.

[17] "IEEE vast challenge." 2011. [Online]. Available: https://visualdata.wustl.edu/varepository

[18] M. Catillo, A. Pecchia, and U. Villano, "AutoLog: Anomaly detection by deep autoencoding of system logs," *Expert Syst. Appl.*, vol. 191, Apr. 2022, Art. no. 116263.

[19] Y. Wang et al., "ContainerGuard: A real-time attack detection system in container-based big data platform," *IEEE Trans. Ind. Informat.*, vol. 18, no. 5, pp. 3327–3336, Dec. 2020.

[20] S. Guo et al., "A zero-day container attack detection based on ensemble machine learning," in *Proc. IEEE 28th Int. Conf. Emerg. Technol. Factory Autom. (ETFA)*, 2023, pp. 1–8.

[21] R. R. Karn, P. Kudva, H. Huang, S. Suneja, and I. M. Elfadel, "Cryptomining detection in container clouds using system calls and explainable machine learning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 674–691, Mar. 2021.

[22] S. Nedelkoski, J. Cardoso, and O. Kao, "Anomaly detection and classification using distributed tracing and deep learning," in *Proc. 19th IEEE/ACM Int. Symp. Clust., Cloud Grid Comput. (CCGRID)*, 2019, pp. 241–250.

[23] C. Zhang et al., "DeepTraLog: Trace-log combined microservice anomaly detection through graph-based deep learning," in *Proc. 44th Int. Conf. Softw. Eng.*, 2022, pp. 623–634.

[24] L. Chen, Q. Dang, M. Chen, B. Sun, C. Du, and Z. Lu, "BertHTLG: Graph-based microservice anomaly detection through sentence-bert enhancement," in *Proc. 20th Int. Conf. Web Inf. Syst. Appl.*, 2023, pp. 427–439.

[25] X. Zhou et al., "Benchmarking microservice systems for software engineering research," in *Proc. 40th Int. Conf. Softw. Eng., Companion (ICSE)*, 2018, pp. 323–324. [Online]. Available: https://doi.org/10.1145/3183440.3194991

[26] Q. Du, T. Xie, and Y. He, "Anomaly detection and diagnosis for container-based microservices with performance monitoring," in *Proc. 18th Int. Conf. Algorithms Archit. Parallel Process. (ICA3PP)*, 2018, pp. 560–572.

[27] J. Nobre, E. S. Pires, and A. Reis, "Anomaly detection in microservice-based systems," *Appl. Sci.*, vol. 13, no. 13, p. 7891, 2023.

[28] "Sock shop: A microservice demo application." WeaveWorks. Accessed: Mar. 11, 2024. [Online]. Available: https://github.com/microservices-demo/microservices-demo

[29] M. Panahandeh, A. Hamou-Lhadj, M. Hamdaqa, and J. Miller, "Serviceanomaly: An anomaly detection approach in microservices using distributed traces and profiling metrics," *J. Syst. Softw.*, vol. 209, Mar. 2024, Art. no. 111917.

[30] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "TeaStore: A micro-service reference application for benchmarking, modeling and resource management research," in *Proc. 26th IEEE Int. Symp. Model., Anal., Simul. Comput. Telecommun. Syst.*, 2018, pp. 223–236.

[31] R. Trivedi, H. Dai, Y. Wang, and L. Song, "Know-evolve: Deep temporal reasoning for dynamic knowledge graphs," in *Proc. 34th Int. Conf. Mach. Learn.*, 2017, pp. 3462–3471. [Online]. Available: https://proceedings.mlr.press/v70/trivedi17a.html

[32] J. Wu, M. Cao, J. C. K. Cheung, and W. L. Hamilton, "TeMP: Temporal message passing for temporal knowledge graph completion," 2020, *arXiv:2010.03526*.

[33] A. Senaratne, P. Christen, P. Omran, and G. Williams, "Anomaly detection and classification in knowledge graphs," 2024, *arXiv:2412.04780*.

[34] P. Demeester, P. Van Daele, T. Wauters, and H. Hrasnica, "Fed4FIRE—The largest federation of testbeds in Europe," in *Building the Future Internet Through FIRE*. Denmark, U.K.: River Publ., 2016, pp. 87–109.

[35] "Online boutique: Cloud-first microservices demo application." Google. 2015. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo

[36] N. Tolaram, "Cadvisor," in *Software Development with Go: Cloud-Native Programming using Golang with Linux and Docker*. Berkeley, CA, USA: Apress, 2022, pp. 347–376.

[37] L. Calcote and Z. Butcher, *Istio: Up and Running: Using a Service Mesh to Connect, Secure, Control, and Observe*. Santa Rosa, CA, USA: O'Reilly Media, 2019.

[38] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service mesh: Challenges, state of the art, and future research opportunities," in *Proc. IEEE Int. Conf. Service-Orient. Syst. Eng. (SOSE)*, 2019, pp. 122–1225.

[39] O. Sheikh, S. Dikaleh, D. Mistry, D. Pape, and C. Felix, "Modernize digital applications with microservices management using the istio service mesh," in *Proc. 28th Annu. Int. Conf. Comput. Sci. Softw. Eng.*, 2018, pp. 359–360.

[40] J. Heyman. "Locust: Open-source performance and load testing." Accessed: Jan. 20, 2024. [Online]. Available: https://locust.io/

[41] S. Xing, Y. Wang, and W. Liu, "Multi-dimensional anomaly detection and fault localization in microservice architectures: A dual-channel deep learning approach with causal inference for intelligent sensing," *Sensors*, vol. 25, no. 11, p. 3396, 2025.

[42] F. Pierfederici, *Distributed Computing with Python*. Birmingham U.K.: Packt Publ. Ltd, 2016.

[43] (Litmus, Cambridge, MA, USA). *Litmuschaos: Open Source Chaos Engineering Platform*. Accessed: Jan. 20, 2024. [Online]. Available: https://litmuschaos.io/

[44] (Amazon, Seattle, WA, USA).) *AWS Fault Injection Service*. (2021). [Online]. Available: https://aws.amazon.com/fis/

[45] (Microsoft, Redmond, WA, USA). *Azure Chaos Studio*. (2021). [Online]. Available: https://learn.microsoft.com/en-us/azure/chaos-studio/

[46] T. Butow. "Chaos engineering: The history, principles, and practice." Tutorial, Gremlin. 2021. [Online]. Available: https://www.gremlin.com/community/tutorials/chaos-engineTutorial

[47] "Chaos mesh: Break your system constructively." Accessed: May 6, 2024. [Online]. Available: https://chaos-mesh.org/

[48] "Argo: Workflow engine for kubernetes." GitHub. Accessed: Jan. 20, 2024. [Online]. Available: https://github.com/argoproj/argo

[49] M.-N. Tran, D.-D. Vu, and Y. Kim, "A survey of autoscaling in kubernetes," in *Proc. 13th Int. Conf. Ubiquitous Future Netw. (ICUFN)*, 2022, pp. 263–265.

[50] G. Antoniou and F. V. Harmelen, "Web ontology language: Owl," in *Handbook on Ontologies*. Berlin, Germany: Springer, 2004, pp. 67–92.

[51] B. Burns, J. Beda, K. Hightower, and L. Evenson, *Kubernetes: Up and Running*. Santa Rosa, CA, USA: O'Reilly Media, Inc., 2022.

[52] I. M. Al Jawarneh et al., "Container orchestration engines: A thorough functional and performance comparison," in *Proc. IEEE Int. Conf. Commun. (ICC)*, 2019, pp. 1–6.

[53] F. Soppelsa and C. Kaewkasi, *Native Docker Clustering with Swarm*. Birmingham, U.K.: Packt Publish. Ltd, 2016.

[54] M. Frampton and M. Frampton, "Apache mesos," in *Complete Guide to Open Source Big Data Stack*. New York, NY, USA: APress, 2018, pp. 97–137.

[55] S. M. Kazemi et al., "Representation learning for dynamic graphs: A survey," *J. Mach. Learn. Res.*, vol. 21, no. 1, pp. 2648–2720, 2020.

[56] M. Jin et al., "An anomaly detection algorithm for microservice architecture based on robust principal component analysis," *IEEE Access*, vol. 8, pp. 226397–226408, 2020.

[57] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko, "Translating embeddings for modeling multi-relational data," in *Proc. Adv. Neural Inf. Process. Syst.*, 2013, pp. 1–9.

[58] P. Ristoski and H. Paulheim, "RDF2Vec: RDF graph embeddings for data mining," in *Proc. 15th Int. Semant. Web Conf. (ISWC)*, 2016, pp. 498–514.

[59] B. Steenwinckel, G. Vandewiele, M. Weyns, T. Agozzino, F. D. Turck, and F. Ongenae, "INK: knowledge graph embeddings for node classification," *Data Min. Knowl. Discov.*, vol. 36, no. 2, pp. 620–667, 2022.

[60] B. Steenwinckel et al., "FLAGS: A methodology for adaptive anomaly detection and root cause analysis on sensor data streams by fusing expert knowledge with machine learning," *Future Gener. Comput. Syst.*, vol. 116, pp. 30–48, Mar. 2021.

[61] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "LOF: identifying density-based local outliers," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2000, pp. 93–104.

[62] A. Analytics. "A python toolkit for rule-based/unsupervised anomaly detection in time series." 2019. [Online]. Available: https://github.com/arundo/adtk

[63] L. Akmeemana, C. Attanayake, H. Faiz, and S. Wickramanayake, "GAL-MAD: Towards explainable anomaly detection in microservice applications using graph attention networks," 2025, *arXiv:2504.00058*.

[64] M. Weyns, T. Blyau, B. Steenwinckel, F. D. Turck, S. V. Hoecke, and F. Ongenae, "Explainable knowledge graph embeddings for industrial process monitoring & control," *Inf. Fusion*, vol. 123, Nov. 2025, Art. no. 103242.

[65] P. Moens, B. Andriessen, M. Sebrechts, B. Volckaert, and S. Van Hoecke, "Edge anomaly detection framework for AIOPs in cloud and IoT," in *Proc. 13th Int. Conf. Cloud Comput. Services Sci. (CLOSER)*, 2023, pp. 204–211.