iAnomaly: A Toolkit for Generating Performance Anomaly Datasets in Edge-Cloud Integrated Computing Environments

Duneesha Fernando, Maria A. Rodriguez and Rajkumar Buyya
Cloud Computing and Distributed Systems (CLOUDS) Laboratory
School of Computing and Information Systems
The University of Melbourne, Australia
Email: dtfernando@student.unimelb.edu.au, {marodriguez, rbuyya}@unimelb.edu.au

***Abstract*—Microservice architectures are increasingly used to modularize IoT applications and deploy them in distributed and heterogeneous edge computing environments. Over time, these microservice-based IoT applications are susceptible to performance anomalies caused by resource hogging (e.g., CPU or memory), resource contention, etc., which can negatively impact their Quality of Service and violate their Service Level Agreements. Existing research on performance anomaly detection in edge computing environments is limited primarily due to the absence of publicly available edge performance anomaly datasets or due to the lack of accessibility of real edge setups to generate necessary data. To address this gap, we propose iAnomaly: a full-system emulator equipped with open-source tools and fully automated dataset generation capabilities to generate labeled normal and anomaly data based on user-defined configurations. We also release a performance anomaly dataset generated using iAnomaly, which captures performance data for several microservice-based IoT applications with heterogeneous QoS and resource requirements while introducing a variety of anomalies. This dataset effectively represents the characteristics found in real edge environments, and the anomalous data in the dataset adheres to the required standards of a high-quality performance anomaly dataset.**

***Index Terms*—Edge computing, Microservices, IoT, Performance anomaly detection, Datasets, Emulators**

I. Introduction

Edge-cloud integrated environments consist of devices with heterogeneous computing, storage, and networking capabilities. Microservice architectures are increasingly used to modularize IoT applications and deploy them in these distributed environments to meet the Quality of Service (QoS) requirements of each module while optimizing resource usage [1], [2]. Over time, these microservice-based IoT applications are susceptible to performance anomalies caused by resource hogging (e.g., CPU or memory) and resource contention, which can negatively impact their QoS and violate their Service Level Agreements [3]–[5]. Therefore, it is crucial to conduct performance anomaly detection on microservice-based IoT applications in edge computing environments and eventually mitigate such anomalies.

Currently, there is limited research on performance anomaly detection in edge computing environments. One of the main reasons for this is the absence of publicly available edge performance anomaly datasets, which are crucial for training and evaluating algorithms proposed in such research. The few existing studies rely on cloud datasets [6], [7] or data collected from private edge setups [3], [4], [8] to evaluate their proposed approaches. The cloud datasets have been collected from applications (mostly web applications) deployed on cloud servers. However, these cloud servers lack the heterogeneity found in edge devices in terms of computing, storage, and networking capabilities. Additionally, the microservices in cloud applications do not demonstrate the same diversity in terms of QoS and resource requirements as those in an IoT application. As a result, cloud datasets fail to capture characteristics inherent to real edge environments. On the other hand, private edge setups have not been publicly released and lack detailed information, which makes it difficult to replicate their environments, generate the necessary data, and reproduce the results of the anomaly detection experiments. It also hinders research in the field because not everyone has access to a real edge-cloud deployment for data collection purposes. Hence, relying on cloud datasets and private edge setups does not facilitate performance anomaly detection research in edge computing environments, thus posing a challenge to the progression of the field. Therefore, there is an opportunity to create a performance anomaly dataset that reflects the characteristics of edge computing environments and release the setup used for dataset generation.

Edge computing emulators are a suitable platform to generate performance anomaly datasets. They are more representative of real edge environments when compared to simulators, and are more easily accessible and cost-effective when compared to real edge deployments. The main aim of existing edge computing emulators is to create a staging environment that achieves compute and network realism similar to a real edge environment and facilitate testing of IoT applications before deploying them into production [9]–[14]. However, these general-purpose emulators do not incorporate in their design, tools and mechanisms required to autonomously and transparently generate large-scale performance anomaly datasets useful for model training and evaluation. For example, they lack adequate monitoring tools to collect performance and system-level metrics, workload generation tools to generate and capture normal performance data, and chaos engineering mechanisms to inject performance anomalies into applications.

This work addresses this gap by presenting the iAnomaly framework, a performance anomaly-enabled full-system emulator that accurately models an edge computing environment hosting microservice-based IoT applications. iAnomaly is designed with open-source tools and provides fully automated dataset generation capabilities to generate labeled normal (data collected under normal conditions without anomalies) and anomaly (data collected under anomalous conditions) data based on user-defined configurations. In addition, we present a performance anomaly dataset generated using the proposed framework. The dataset captures performance data for several microservice-based IoT applications with heterogeneous QoS and resource requirements across a wide range of domains, software architectures, service composition patterns, and communication protocols by introducing a variety of client/sensor-side as well as server-side anomalies. To the best of our knowledge, this multivariate dataset is the first open-source edge performance anomaly dataset.

The analysis of the dataset showed that the microservices within it vary in terms of their QoS and resource usage during regular operation, thus successfully capturing the characteristics of a real edge dataset. Further analysis confirmed that the anomalous data in the dataset meets the necessary standards for a high-quality performance anomaly dataset. This includes having an anomaly ratio comparable to other standard anomaly datasets and the dataset's non-triviality.

The rest of the paper is organized as follows: Section II reviews the existing related works. Section III presents the architecture of the iAnomaly toolkit, while section IV discusses the implementation aspects of iAnomaly. Section V provides details of the generated performance anomaly dataset followed by an analysis of the dataset. Section VI concludes the paper and draws future research directions.

II. Related Work

Out of the existing research studies conducted around performance anomaly detection in edge computing environments, Becker et al. [3], Soualhia et al. [4], and Skaperas et al. [8] evaluated their proposed approaches using data collected from private edge setups. However, the lack of detailed information about these private edge setups makes it challenging to reproduce their environments and generate the necessary data to replicate the results of their anomaly detection experiments. Tuli et al. [6] evaluated their proposed approaches using two publicly available cloud datasets: the Server Machine Dataset (SMD) collected from a large Internet company [15], and the Multi-source Distributed System (MSDS) dataset generated from microservices deployed on a cluster of bare metal nodes with homogeneous computing, storage, and network capabilities [16]. As a result, these cloud datasets are unable to accurately represent the properties inherent to real edge environments. Tuli et al. also conducted a further evaluation on three self-created datasets collected from a private edge setup. However, similar to the rest of the literature, they also do not provide sufficient details required for reproducing the data generation environments. Therefore, the reliance on cloud datasets and private edge setups presents a challenge to the progression of the field of performance anomaly detection research in edge computing environments. Additionally, there is a lack of publicly available normal traces from real edge environments into which anomalies can be injected to generate synthetic datasets. We further explore this gap in current research by creating and releasing a performance anomaly dataset that reflects the characteristics of edge computing environments along with the setup used for dataset generation.

There are three options of platforms for generating a performance anomaly dataset. They are simulators, emulators, and real edge environments. Out of these, emulators employ real applications deployed on testbed hardware to emulate real-world infrastructure configurations [14], while simulators do not support real-world IoT protocols and services [10]. Simulations make a number of simplifications that may not always hold true, especially with an infrastructure as dynamic as edge computing [9]. Most simulators lack detailed network simulation capabilities and focus on specific aspects of edge modeling, such as service scheduling [14]. Therefore, we identify emulators as the most suitable platform for generating data as they provide a higher Degree of Realism (DoR) than simulators and because they enable the generation of large-scale data in a cost-effective manner as opposed to real edge environments.

Existing edge computing emulators can be organized under two main categories based on the level of virtualization and abstraction used to model the edge devices. They are 1) full-system emulators and 2) container-based emulators. In container-based emulators, edge devices are represented as docker containers, while full-system emulators provide a higher granularity of emulation by allowing the deployment of multiple containerized microservices within edge devices modeled as virtual machines (VMs). Early emulators such as EmuFog [9], FogBed [10], MockFog [11], and EmuEdge [12] do not support the microservice-level granularity of IoT applications. Fogify [13] is the first edge emulator to support the microservice-level granularity of IoT applications. However, it is limited to deploying only a single microservice per edge device due to being a container-based emulator. Extending such emulators with dataset generation capabilities restricts data collection to device-level anomalies only. In contrast, iContinuum [14] is a full-system emulator with support for microservices deployment. Unlike container-based emulators, full-system emulators provide a higher level of realism and also allow the injection of both device-level and microservice-level anomalies. Consequently, our research aims to bridge the identified gap by developing a full-system emulator with performance anomaly dataset generation capabilities. A comparison of performance anomaly dataset generation capabilities in existing edge emulators [9]–[14] along with our proposed iAnomaly toolkit is shown in Table I.

The main intention of existing emulators is to test IoT

TABLE I
Comparison of performance anomaly dataset generation capabilities in edge emulators

Edge emulator/ toolkit	**Main objective of work**	**Fault injection capabilities**	**Performance anomaly dataset generation capabilities**	**Emulation capabilities/ architecture**	**Microservice support**	**Applications**
EmuFog [9]	Testing IoT apps in a staging environment before deploying into the real edge, Automatic placement of fog nodes.	×	×	Focused on network emulation	×	Not reported
FogBed [10]	Creating an environment to conduct resource management/service orchestration experiments.	×	×	Focused on network emulation, Container-based emulator	×	Healthcare prevention and monitoring system
MockFog [11]	Testing IoT apps in a staging environment before deploying into the real edge.	×	×	Full-system emulator	×	Ambulance cars communicating vital measures to hospitals
EmuEdge [12]	Achieving compute and network realism of a real edge environment.	×	×	Full-system emulator	×	Not reported
Fogify [13]	Testing IoT apps in a staging environment before deploying into the real edge.	✓	×	Container-based emulator	✓	Smart transport applications
iContinuum [14]	Achieving compute and network realism of a real edge environment, Intent-based emulation.	✓	×	Full-system emulator	✓	Image processing application
iAnomaly (proposed)	Creating a toolkit to generate performance anomaly datasets.	✓	✓	Full-system emulator	✓	Face detection/recognition application, Industrial machinery predictive maintenance application, Location retrieval application

applications in a staging environment before deploying them into production. They are designed to achieve the compute and network realism of an edge environment, and the evaluation of these studies is also focused on those aspects. Modern emulators such as Fogify [13] and iContinuum [14] also have the capability to perform fault injections. However, the main intention of such fault injection capabilities is not to collect data for performance anomaly detection model training but to test the fault tolerance and availability aspects of IoT applications in the face of faults.

Although both Fogify and iContinuum have implemented and evaluated fault injection capabilities, neither of them has specified the tools used to inject anomalies. Since these emulators only conducted injections of a limited number of anomaly types, it can be inferred that they likely utilized basic tools, such as stress-ng, for this purpose. However, such mechanisms do not allow for the introduction of failures or disruptions in a controlled manner.

Both Fogify and iContinuum are capable of collecting both system and application metrics during monitoring. However, Fogify utilizes an in-house developed monitoring tool for this purpose, whereas open-source tools are preferred in emulators to support interoperability and transparency of code. iContinuum employs sFlow-RT, an open-source tool, to capture network and host-level metrics such as CPU and memory usage. It also integrates sFlow agents with Prometheus, another open-source tool, to collect application metrics. While Fogify requires explicit instrumentation, i.e. manually embedding monitoring code within the source code, in order to capture performance metrics, sFlowRT can only capture application metrics without explicit instrumentation from IoT applications that communicate via HTTP protocol. As most IoT applications deployed in edge computing environments are not limited to HTTP protocol and use a variety of protocols such as MQTT, RTSP, etc., it is important to be able to collect metrics from applications communicating via such non-HTTP protocols as well.

Fogify has not provided details of its workload generation tool, while iContinuum uses Locust for workload generation. However, Locust primarily focuses on generating HTTP/HTTPS workloads, while it is important to incorporate a workload generation tool supporting a wide range of protocols, not just HTTP.

Consequently, it is evident that the current emulators lack the necessary tools for generating performance anomaly data. These tools include a monitoring tool for collecting metrics, a workload generation tool for creating normal performance data, and a chaos engineering tool for injecting performance anomalies. Identifying this research gap, our paper aims to address it by developing a toolkit with an emulator that incorporates a set of open-source tools for generating performance anomaly datasets.

In addition to finding the best open-source tools for generating performance anomaly datasets and creating a full-system emulator with performance anomaly dataset generation capabilities, we also integrate automated dataset generation features into our proposed iAnomaly toolkit. Moreover, as shown in Table I, most emulators have released only one IoT application which they used in their experiments. However, we generate (and release) an open-source performance anomaly dataset using iAnomaly by deploying three IoT applications consisting of microservices with varying QoS and resource requirements. These applications span a wide range of domains, software architectures, service composition patterns, and communication protocols.

III. iAnomaly Architecture

Figure 1 depicts the architecture of iAnomaly. At the core of the framework is a full system emulator that comprises multiple distinct layers with a set of components to build all the layers from infrastructure to applications. The infrastructure layer hosts a diverse array of computing and networking resources. While the heterogeneity of compute nodes can be emulated using Virtual Machines (VM) with different resource capacities in the cloud, a network emulator is used to construct the network topology (by creating virtual

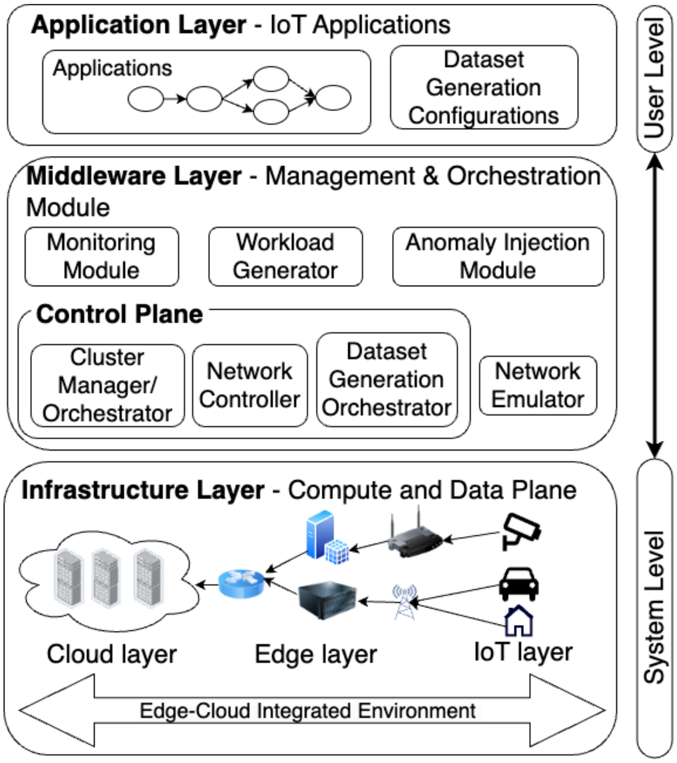

Application Layer - IoT Applications
Applications
Dataset Generation Configurations
User Level
Middleware Layer - Management & Orchestration Module
Monitoring Module
Workload Generator
Anomaly Injection Module
Control Plane
Cluster Manager/ Orchestrator
Network Controller
Dataset Generation Orchestrator
Network Emulator
Infrastructure Layer - Compute and Data Plane
Cloud layer
Edge layer
IoT layer
Edge-Cloud Integrated Environment
System Level

Fig. 1. System architecture of iAnomaly

switches and network elements) and simulate the network within the infrastructure layer. Users can define the application structure of the microservice-based IoT applications through the application layer. The middleware layer of a full-system emulator manages the deployment and operation of applications across the emulated infrastructure. Its control plane consists of a cluster manager/orchestrator that utilizes containerization and orchestration technologies to manage the computing cluster and its resources, and a network controller that uses Software-Defined Networking (SDN) technologies to manage the network (such as effectively regulating the network flow while considering resource usage conditions).

In addition to a full system emulator, iAnomaly includes components to facilitate the data generation and collection process within its middleware layer. These components consist of a monitoring module, a workload generation tool, and a tool for injecting anomalies.

The monitoring module is responsible for gathering system and application metrics from the deployed microservices. The collected data will be stored in a database and retrieved back through queries when creating the dataset. An important requirement of the monitoring tool is to be able to collect data from IoT applications communicating not only via HTTP-based protocols but also via non-HTTP protocols such as Kafka, MQTT, and RTSP. In addition, it is preferable for the tool to be capable of collecting application metrics from programs without the need for explicit instrumentation.

The workload generator is in charge of sending request/sensor data to the microservices. This tool is used during normal data generation as well as for introducing client/sensor-side anomalies such as user surges and spikes. Details of request workloads, including concurrency and duration, are specified using test plans. It is important that the workload generation tool also supports a wide range of protocols, not just HTTP. When generating data from a specific microservice, we need at least two instances of workload generators, one for generating normal workloads, and another for introducing client/sensor-side anomalies.

The anomaly injection tool is responsible for injecting server-side anomalies, such as resource hogging and service failures. MockFog [11], which is one of the early edge emulators, suggested using chaos engineering tools for injecting anomalies and conducting performance testing. Chaos engineering tools are designed to test the fault tolerance of systems by deliberately introducing failures or disruptions in a controlled manner, making them suitable for inclusion in the proposed toolkit to inject performance anomalies. Moreover, by incorporating chaos engineering tools, we can inject a diverse range of anomalies, unlike with basic tools such as stress-ng.

Generating a significant amount of normal and anomaly data by using these data collection tools is a time-consuming and repetitive task necessitating human intervention. Additionally, there is a learning curve associated with using the tools, notably in terms of creating necessary test plans (incorporating varying parameters representing normal and anomalous workloads) using the workload generator, designing chaos engineering configurations, and scripting data collection for retrieving information from the monitoring tool.

To overcome these challenges, we further extend the iAnomaly framework with automated dataset generation capabilities, where users can provide the configurations of the expected dataset, and the resulting labeled normal and anomaly data will be stored in a predefined location. As depicted in Figure 1, users can define the dataset generation configurations through the application layer. We also introduce a dataset generation orchestrator to the middleware layer, which interprets the content from the dataset generation configuration and coordinates with the data generation tools in the toolkit to generate the normal and anomaly data required for the performance anomaly dataset. This process will be explained in detail in the next section.

IV. iAnomaly Implementation

This section describes the implementation details of the architecture presented in section III. The deployment diagram of the iAnomaly toolkit is shown in Figure 2. iAnomaly relies on iContinuum [14] as a full-system emulator to accurately model edge computing environments hosting microservice-based IoT applications. iContinuum utilizes Virtual Machines (VMs) with varying resource specifications to demonstrate the heterogeneity of edge devices in terms of computing and storage. Additionally, a VM with higher resource capacities acts as the master node, hosting only the tools required for compute orchestration. Acknowledging the critical role of the master node as the system's control plane, we ensure that

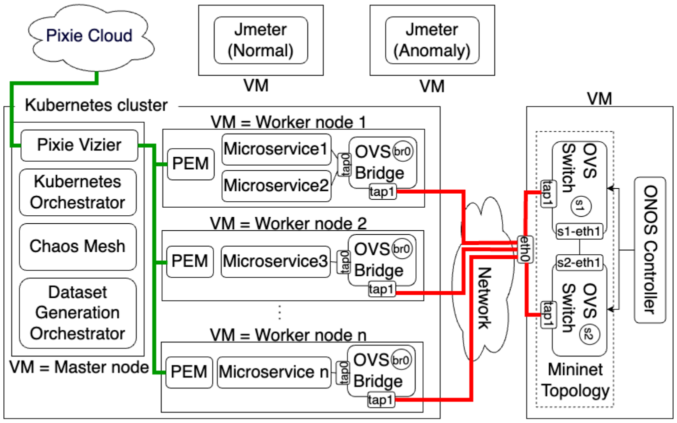


Fig. 2. Deployment diagram of the iAnomaly toolkit

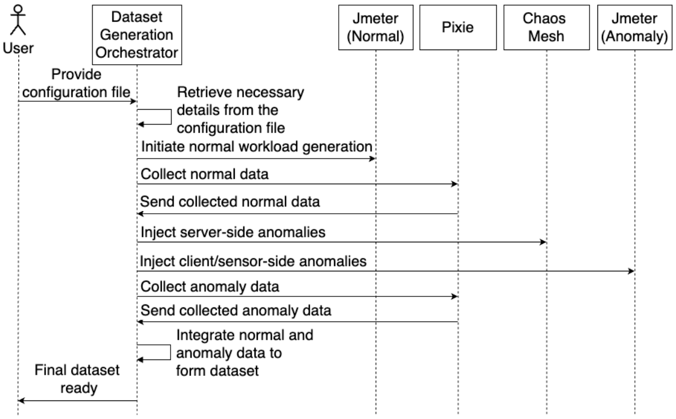

User
Dataset Generation Orchestrator
Jmeter (Normal)
Pixie
Chaos Mesh
Jmeter (Anomaly)
Provide configuration file
Retrieve necessary details from the configuration file
Initiate normal workload generation
Collect normal data
Send collected normal data
Inject server-side anomalies
Inject client/sensor-side anomalies
Collect anomaly data
Send collected anomaly data
Integrate normal and anomaly data to form dataset
Final dataset ready

Fig. 3. Interactions between the dataset generation orchestrator and other components

microservices are deployed only on VMs other than the master node. Following the implementation of iContinuum, iAnomaly uses Mininet[1] as the network emulator to construct the network topology and simulate the bandwidth between edge devices. The OVS switches that form the Mininet topology are deployed in a separate VM, which is also nondeployable for microservices.

In line with iContinuum's control plane implementation, iAnomaly also uses Kubernetes, specifically K3s[2], which is a lightweight Kubernetes distribution designed for resource-constrained environments such as edge computing or IoT devices, as the cluster manager/orchestrator. Additionally, iAnomaly utilizes Open Network Operating System (ONOS)[3] as the network controller, and it manages the OVS switches in the Mininet topology. The ONOS controller is also deployed in the VM where the OVS switches are deployed. Figure 2 shows how the Kubernetes orchestrator is deployed in the master node and forms a multi-node Kubernetes cluster with the worker nodes. Each worker node is configured with an OVS bridge featuring two virtual interfaces, tap0, and tap1, out of which tap1, which is configured as a GRE interface, is linked to the tap1 port of the corresponding OVS switch. This ensures that the worker nodes have a bi-directional connection with the Mininet-created network topology through Generic Routing Encapsulation (GRE) tunnelling.

The Monitoring Module is realized by using Pixie[4], a lightweight and open-source eBPF(extended Berkeley Packet Filter)-based monitoring tool specifically designed for Kubernetes applications. eBPF-based monitoring tools, which gained popularity recently, allow sandboxed programs to execute directly inside the Linux kernel and automatically capture telemetry data in a non-intrusive manner, i.e., without requiring modifications to user-space applications [17]. It also supports monitoring a wide variety of protocols, including HTTP, Kafka, AMQP, and MySQL, making it well-suited for monitoring edge computing environments. When Pixie is deployed via the toolkit, a Pixie Edge Module (PEM) is deployed for each worker node. These modules capture monitoring data from microservices deployments on the worker nodes and send those to the Pixie Vizier deployed on the Kubernetes master node, from where they are transferred to the Pixie cloud. Pixie Vizier acts as Pixie's central collector and is also responsible for managing PEMs. When retrieving back the collected data, data retrieval queries written in Pixie language (PxL) are executed against the Pixie cloud via a Pixie API client.

The Workload Generator is implemented by leveraging Jmeter[5] as it supports a wide range of protocols, not just HTTP. We are using two separate instances of JMeter: one to generate normal workloads and the other to create client/sensor-side anomalies. Both instances are deployed outside the Kubernetes cluster to run the JMeter loads as needed and to avoid interfering with the master node's operations.

Chaos Mesh[6], an open-source chaos engineering platform for Kubernetes, is used as the anomaly injection tool and deployed in the Kubernetes master node. From there, CRD (Custom Resource Definition) YAMLs are applied to introduce server-side anomalies into the target deployments. Chaos Mesh is capable of injecting a multitude of server-side anomalies such as CPU stress, memory stress, network delay, etc.

The dataset generation orchestrator is deployed in the Kubernetes master node. As illustrated in Figure 3, it acts as the central component that interprets the content from the dataset generation configuration/s and coordinates with the data generation tools in the toolkit to produce the labeled normal and anomaly data needed for the performance anomaly dataset. To start the data generation process, the dataset generation orchestrator reads the configuration file/s to retrieve the deployment details, normal data collection parameters, and anomaly injection settings. It then remotely executes the test plans on Jmeter's normal data generation instance using Paramiko's[7] SSH client to initiate normal workload generation. Once the normal data has been generated for

[1] https://mininet.org/
[2] https://k3s.io/
[3] https://opennetworking.org/onos/
[4] https://docs.px.dev/
[5] https://jmeter.apache.org/
[6] https://chaos-mesh.org/
[7] https://docs.paramiko.org/

the required duration, the orchestrator executes PxL queries to collect the generated normal data. Simultaneously, while the normal workload generation process is ongoing, the orchestrator proceeds to inject anomalies. Anomalies are injected either through server-side disruptions using Chaos Mesh or via client/sensor-side anomalies through Jmeter's anomalous data generation instance. For server-side disruptions, the dataset generation orchestrator applies the corresponding chaos YAMLs through the installation of helm charts[8]. After the entire anomaly injection period, the orchestrator executes PxL queries to collect the corresponding anomaly data. Finally, both normal and anomaly data are funnelled back to the orchestrator, which integrates and processes the data to create a comprehensive dataset suitable for training and evaluating performance anomaly detection models.

Therefore, the realization of the architecture proposed in section III using the open-source tools discussed earlier, has made the process of dataset generation easily accessible for researchers. We have released the source code of the iAnomaly toolkit, which includes iContinuum as its full-system emulator together with the chosen open-source data collection tools and the code for automated dataset generation, in a public repository[9].

V. Case Study: Dataset Generation

This section showcases how iAnomaly was used to create an open-source labeled dataset consisting of normal and anomalous data collected for three different IoT applications, followed by an analysis of the generated performance anomaly dataset.

A. IoT Applications

The generated dataset records data for three IoT applications typically deployed in edge environments:

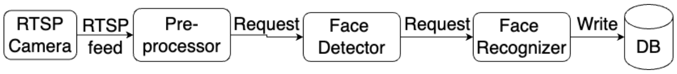


Fig. 4. Face detection/recognition application

a) Face detection/recognition: This application showcases the scenario of using computer vision for secure access control at a corporate office building. As illustrated in Figure 4, it comprises four microservices: 1) preprocessor, 2) face detector, 3) face recognizer, and 4) database. Cameras at entry points (e.g., doors, gates) produce an RTSP stream. The preprocessor microservice reads from this video stream, performing resizing and grayscale conversion on the images and carrying out motion detection by thresholding the difference between consecutive frames. Upon detection of motion (i.e., when an employee approaches), the frame is sent to the face detector microservice. This microservice utilizes a Multi-Task Cascaded Convolutional Neural Network (MTCNN) to detect bounding boxes and landmark points of faces in the frame. Additionally, it conducts face alignment using affine transformation and supports multi-face alignment within a single frame. Aligned faces are then input to the face recognizer microservice. The face recognizer microservice employs a ResNet-50 model to extract features from the detected faces and calculates the cosine distance between these features and the features of the authorized personnel's faces to calculate the similarity between detected faces and faces of authorized personnel. Upon successful recognition, the timestamp is logged in the database - which is implemented as another microservice - marking the employee's entry or exit from the building.

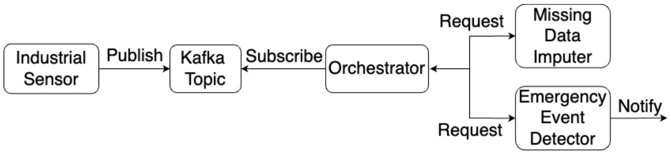


Fig. 5. Industrial machinery predictive maintenance application

b) Predictive maintenance for industrial machinery: In this application, IoT sensors that measure temperature, vibration, and pressure continuously generate multivariate time series data, which is then written to a Kafka topic. As shown in Figure 5, an orchestrator microservice subscribes to this Kafka topic, reads the raw sensor data, and forms time-series windows, which are sent to the emergency event detector microservice. Before the windowed data is sent for emergency event detection, the orchestrator imputes any missing values by calling the missing data imputer microservice and also standardizes the data using standard score normalization. The emergency event detector uses an isolation forest model to detect whether a window of standardized time series data is anomalous by detecting patterns that deviate from the norm, such as overheating, excessive vibration, or mechanical stress. When an anomaly is detected, alerts are triggered to the maintenance teams.

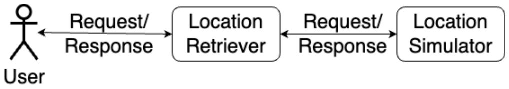


Fig. 6. Location retrieval application

c) Location retrieval: This application depicts the scenario of fleet management for logistics or delivery services. It facilitates tracking the real-time location of delivery vehicles in a fleet. As shown in Figure 6, when a dispatcher or a customer requests the location of a specific vehicle, the request is directed to the location retriever microservice. This microservice checks its Least Recently Used (LRU) cache, and if the vehicle's location was recently queried, it provides the cached location for quick access. If the location is not in the cache, the microservice queries the vehicle's current location, updates the cache, and returns the most up-to-date information. In our implementation, the location simulator microservice is

[8] https://helm.sh/

[9] https://github.com/Cloudslab/iAnomaly

TABLE II
PROPERTIES OF IoT APPLICATIONS USED TO GENERATE PERFORMANCE ANOMALY DATASET

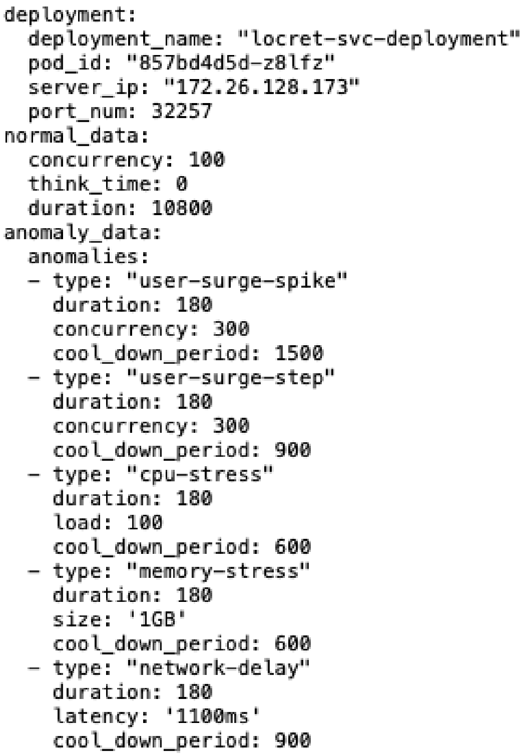
Application	Microservice	Type of task performed	Software architecture	Service composition pattern	QoS properties
Face detection/ recognition application	Preprocessor	Computer vision-based	Stream processing	Chained	LC, HTp,HCI, BI
	Face Detector	Computer vision-based	Request-response		LC, MTp, HCI
	Face Recognizer	Computer vision-based	Request-response		LC, LTp, HCI
	Database	General purpose	Request-response		LT, LTp
Industrial machinery predictive maintenance application	Orchestrator	Time-series processing	Publish-subscribe	Aggregator	LC, LTp
	Emergency event detection	Time-series processing	Request-response		LC, LTp, MCI
	Missing data imputation	General purpose	Request-response		LC, LTp, MCI
Location retrieval application	Location service with timed cache	Simple non-resource-intensive	Request-response	Passthrough	LC, HTp
LC: Latency Critical, LT: Latency Tolerant, HTp: High Throughput, MTp: Moderate Throughput, LTp: Low Throughput, HCI: High Compute Intensive, MCI: Moderate Compute Intensive, BI: Bandwidth Intensive					

used to mock the location of the delivery vehicle by generating GPS locations in a trajectory.

As shown in Table II, the aforementioned applications span the properties of a wide range of IoT applications. For instance, the face detection/recognition application falls under Computer Vision (CV), the industrial machinery predictive maintenance application focuses on time-series processing, and the location retrieval application is a simple, non-resource-intensive application. Each application relies on different software architectures, including stream processing, request-response, and publish-subscribe, which are implemented using different communication protocols, including HTTP, Kafka, RTSP, and MySQL. The applications also employ different service composition patterns, such as chained, aggregator, and passthrough. Most importantly, the microservices in these applications have unique QoS and resource requirements.

The iAnomaly repository also contains the Python implementation for all three IoT applications. In addition, we have made the Docker images of the microservices accessible to the public[10].

B. Dataset Generation



```
deployment:
  deployment_name: "locret-svc-deployment"
  pod_id: "857bd4d5d-z8lfz"
  server_ip: "172.26.128.173"
  port_num: 32257
normal_data:
  concurrency: 100
  think_time: 0
  duration: 10800
anomaly_data:
  anomalies:
  - type: "user-surge-spike"
    duration: 180
    concurrency: 300
    cool_down_period: 1500
  - type: "user-surge-step"
    duration: 180
    concurrency: 300
    cool_down_period: 900
  - type: "cpu-stress"
    duration: 180
    load: 100
    cool_down_period: 600
  - type: "memory-stress"
    duration: 180
    size: '1GB'
    cool_down_period: 600
  - type: "network-delay"
    duration: 180
    latency: '1100ms'
    cool_down_period: 900
```

Fig. 7. Dataset generation configuration YAML file of the location retrieval application

The applications presented in Section V-A were deployed in a Kubernetes cluster, where iAnomaly was responsible for the orchestration and automation of the data generation and collection process. Specifically, the physical environment consisted of ten VMs with heterogeneous computing and storage specifications created in the Melbourne Research Cloud[11] to emulate the worker nodes. In particular, two 2vCPU/8G VMs represented the IoT layer, four 2vCPU/8G VMs represented Fog level 1, three 4vCPU/16GB VMs represented Fog level 2, and one 8vCPU/32GB VM represented Fog level 3. The configuration of the emulated network was determined based on existing related research [18], with the following specifications: IoT layer → Fog level 1: 5ms/100Gbps, Fog level 1 → Fog level 2: 20ms/10Gbps, Fog level 2 → Fog level 3: 50ms/0.15Gbps, and 2ms bandwidth among nodes at the same level. The applications were assigned to the worker nodes by following the QoS-aware scheduling algorithm proposed by Pallewatta et al. in their research study [19].

Thereafter, normal and anomaly data were generated from each application by providing dataset generation configurations specified in the form of YAML files. For example, the configuration shown in Figure 7 was used to generate data from the location retrieval application. This configuration instructs iAnomaly to generate and collect normal performance data from the location retriever microservice over a duration of three hours. Additionally, it specifies the injection of five types of anomalies—CPU hog, memory stress, user surge spike, user surge step, and network delay—over a total duration of two hours. Similar configurations were used to collect performance data from all applications. While five types of anomalies

[10] https://hub.docker.com/repository/docker/dtfernando/ianomaly
[11] https://dashboard.cloud.unimelb.edu.au/

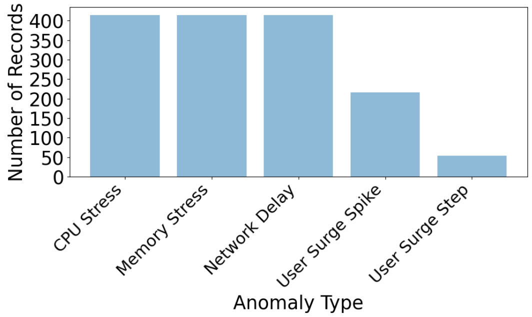


Fig. 8. Distribution of records by anomaly type

(two client-side and three server-side) were injected into the location retrieval application, only a subset of these anomalies was introduced into the other applications. This selection was based on the likelihood of each anomaly type occurring in real-world conditions for the respective applications. Figure 8 depicts the distribution of different types of anomalies across the dataset. It is also important to note that data for each application was collected independently to avoid anomalies caused by colocation.

Pixie's default granularity of 10 seconds was used when collecting data. For each application, data was collected across 12 metrics, covering key aspects of system and application performance: disk read and write throughput (total_disk_read_throughput, total_disk_write_throughput), memory usage (rss, vsize), CPU utilization (cpu_usage), network activity (rx_bytes_per_ns, tx_bytes_per_ns), latency percentiles (latency_p50, latency_p90, latency_p99), request throughput (request_throughput), and error rate (errors_per_ns), collectively providing a comprehensive view of resource consumption, network efficiency, and service reliability.

The final dataset comprises a total of 30240 records. Of these, 19260 records attribute to 54 hours of normal data, and 10980 records account for 31 hours of anomalous data. Within this dataset, there are 1512 records labelled as anomalous data points, resulting in an anomaly ratio of 5%. This ratio is consistent with the anomaly ratio of other standard anomaly datasets, such as SMD (5.84%) and ASD (4.61%) [20], indicating that our dataset maintains a realistic anomaly density—an important characteristic of a high-quality anomaly dataset [21]. The collected dataset is also made available at the iAnomaly repository.

C. Analysis of the Generated Dataset

Figure 9 illustrates the colinearity among the metrics in the generated dataset after excluding errors_per_ns metric, which contains all-zero values. The plot shows that inherently related groups of metrics, such as disk read/write throughputs, as well as latency percentiles, are highly correlated. In addition, metrics such as request_throughput and tx_bytes_per_ns, as well as disk read/write throughputs and rx_bytes_per_ns, exhibit a strong positive correlation with each other. Outside of these groups, most other metrics do not show strong correlations with each other, indicating that each metric serves a unique purpose within the dataset. To simplify our analysis, we select a single metric from each identified group of correlated metrics to serve as a proxy for the others in the group. Based on this correlation analysis, we identify cpu_usage, rss, rx_bytes_per_ns, vsize, request_throughput, and latency_p50 as the subset of metrics with the lowest collinearity. Consequently, we will focus on these metrics for further analysis.

Figure 10 illustrates the distribution of normal data for each shortlisted metric, focusing on three instances of the location retriever microservice and one instance of each of the other microservices. The three instances of the location retriever

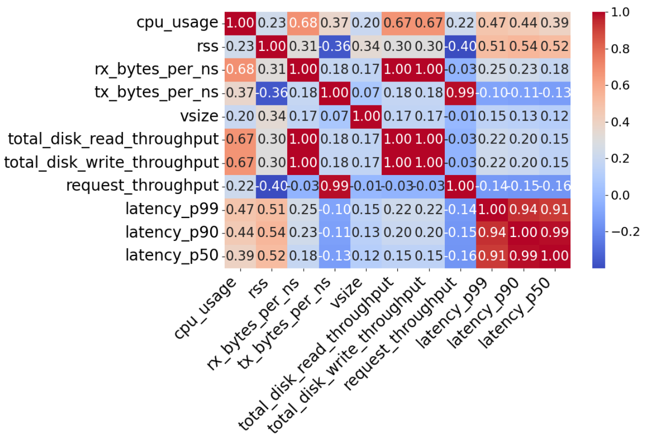


Fig. 9. Collinearity among metrics in the generated dataset

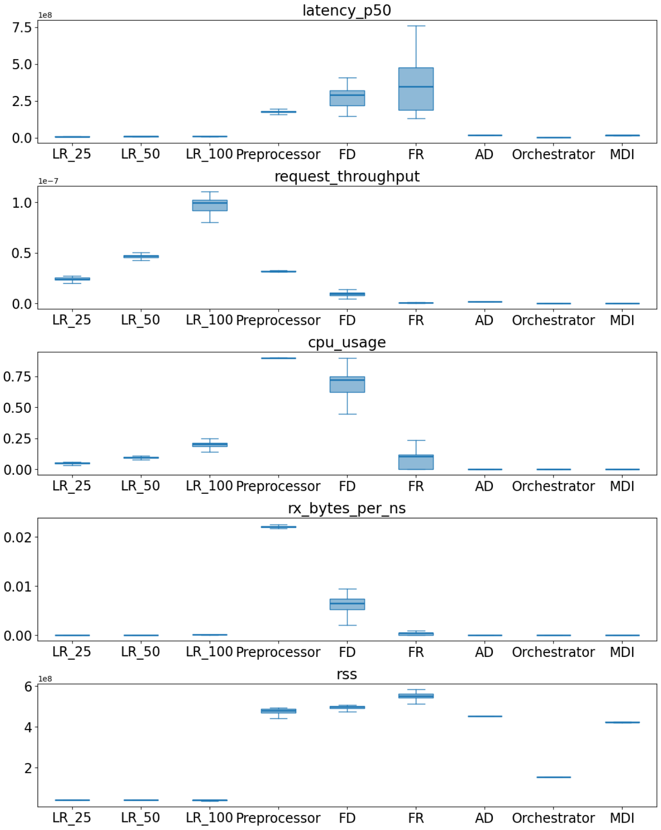


Fig. 10. Distribution of normal data across shortlisted metrics

correspond to different deployments in regions with varying user populations. In the latency_p50 subplot, it is evident that the preprocessor, face detector, and face recognizer microservices experience the highest latency. This increased latency is attributed to their highly compute-intensive (HCI) nature, which requires more time to process and respond to individual requests. In contrast, the other microservices demonstrate lower latency due to their relatively lower compute intensity.

The second subplot represents the request_throughput metric. Here, we can observe that the preprocessor and location retriever microservices exhibit high request throughput (HTp), while the face detector shows moderate throughput (MTp). The other microservices fall into the low throughput (LTp) category. These observations confirm the expected QoS properties of the microservices, as listed in Table II. The third subplot corresponds to the cpu_usage metric. Despite being HTp, the location retriever microservices result in low CPU usage since they are not computationally intensive. The preprocessor microservice shows the highest CPU usage due to its HCI nature and high request throughput. The face detector microservice, while also HCI, has moderate throughput and, therefore, has the second-highest CPU usage. The rest of the microservices have a low CPU usage due to their LTp nature.

The rx_bytes_per_ns subplot confirms the bandwidth-intensive (BI) nature of the preprocessor microservice. The final subplot, which corresponds to the rss metric, indicates that three computer vision microservices, together with the anomaly detector and the missing data imputer (which also utilize machine learning models for processing), have high rss values, demonstrating significant memory usage. By comparing these subplots, we can see that the diversity of the selected applications allows our collected dataset to effectively capture the variations in QoS and resource requirements of the microservices, as expected from an edge dataset.

Figure 11 contains the Probability Density Functions (PDFs) for the normal and anomalous data distributions of two randomly selected metrics from the datasets of the preprocessor and face detector microservices. Subfigure 11(a) corresponds to latency_p50 of the preprocessor microservice while figure 11(b) corresponds to cpu_usage of the face detector microservice. Both subplots illustrate that the anomalous data overlaps with the distribution of normal data. This overlap proves that the anomalies present in our dataset are non-trivial and not merely outliers. Renjie et al. [21] have identified the presence of such non-trivial anomalies as a property of a good anomaly dataset.

Furthermore, Figure 12 visualizes a few selected anomalies from the dataset. While certain anomalies are easily noticeable using their respective metrics - for example, user surge anomalies are evident from the increase in the latency_p50 metric (Figure 12(a)), and memory stress is apparent from the rss metric (Figure 12(b)) - some anomalies, such as CPU stress, cannot be detected simply by looking at the cpu_usage metric (Figure 12(c)), especially when it occurs in compute-intensive microservices. During such scenarios, which are non-trivial to detect, algorithms that are capable of analyzing the higher-order relationships and behavior of several metrics are required to make an accurate detection.

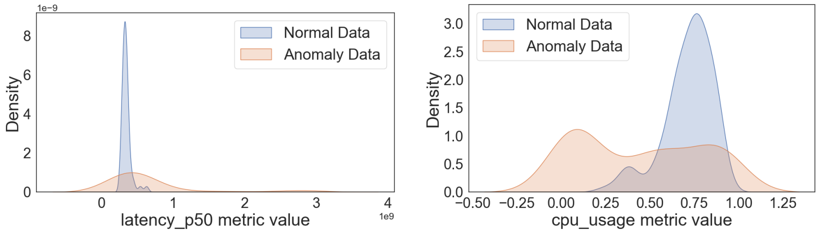
Successful collection of the dataset was possible due to iAnomaly's use of an optimal set of open-source tools. In particular, leveraging Pixie as the monitoring tool allowed it to gather metric data from all three IoT applications, each using different communication protocols. In contrast, using a regular full-system emulator like iContinuum would only allow data collection from the location retrieval application, which uses HTTP for communication. Furthermore, iAnomaly's automated dataset generation capabilities led to an 87% reduction in code lines compared to using a regular full-system emulator such as iContinuum during our dataset generation. Notably, iAnomaly completely eliminated the need for human intervention during the data collection process, requiring only 31 lines of configurations per microservice, while iContinuum needs 307 lines of code and significant human involvement to generate the same dataset.

VI. Conclusions and Future Work

Since existing research on performance anomaly detection in edge computing environments is limited due to the absence of publicly available edge performance anomaly datasets and due to the lack of accessibility of real edge setups to generate necessary data, we propose iAnomaly: a full-system emulator with performance anomaly dataset generation capabilities. Towards that, it is equipped with open-source tools such as Pixie for monitoring, Jmeter for normal workload generation, and client/sensor-side anomaly injection, as well as Chaos Mesh to introduce server-side anomalies. It also incorporates a dataset generation orchestrator to facilitate automatic data generation and collection based on user-defined configurations.

As a case study, we generated a performance anomaly dataset using iAnomaly. It contains performance data for various microservice-based IoT applications with different QoS and resource requirements, injected with anomalies on both the client/sensor side and the server side. Analysis of this dataset showed that it represents the characteristics of real edge environments, and the anomalous data in the dataset meets the required standards for high-quality performance anomaly datasets. We have made this dataset available to the public. Additionally, we have released the iAnomaly toolkit for other researchers who may need to collect a more extensive dataset or conduct further anomaly detection research.

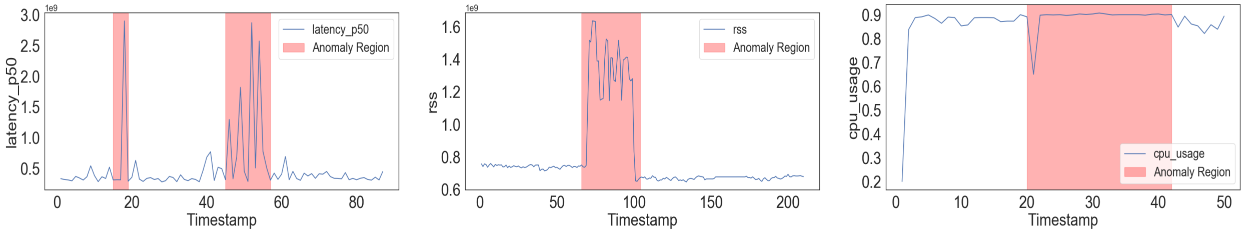
iAnomaly toolkit can easily be extended to collect trace data alongside metrics, which is particularly useful for research on root cause localization (RCL). Extending to such multi-source datasets is possible due to Pixie's support. Researchers can further enhance this framework to conduct real-time experiments on anomaly-aware resource management. The toolkit and the released dataset are not limited to anomaly detection research. Normal data from the dataset, as well as normal data generated from the iAnomaly toolkit, can be used as foundational traces for experiments in other related research areas, such as resource scheduling and resource management.



(a) latency_p50 of the preprocessor

(b) cpu_usage of the face detector

Fig. 11. PDFs for normal and anomalous data distributions of selected metrics



(a) User surges using latency_p50

(b) Memory stress using rss

(c) CPU stress using cpu_usage

Fig. 12. Visualization of selected anomalies from the dataset

References

[1] F. Al-Doghman, N. Moustafa, I. Khalil, N. Sohrabi, Z. Tari, and A. Y. Zomaya, "Ai-enabled secure microservices in edge computing: Opportunities and challenges," *IEEE Transactions on Services Computing*, vol. 16, pp. 1485–1504, 2023.

[2] C. Wu, Q. Peng, Y. Xia, Y. Jin, and Z. Hu, "Towards cost-effective and robust ai microservice deployment in edge computing environments," *Future Generation Computer Systems*, vol. 141, pp. 129–142, 2023.

[3] S. Becker, F. Schmidt, A. Gulenko, A. Acker, and O. Kao, "Towards aiops in edge computing environments," in *Proceedings of the 2020 IEEE International Conference on Big Data*, Atlanta, GA, USA, 2020.

[4] M. Soualhia, C. Fu, and F. Khomh, "Infrastructure fault detection and prediction in edge cloud environments," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, Arlington, Virginia, 2019.

[5] J. Hunter, "Deep learning-based anomaly detection for edge-layer devices," Master's thesis, University of Tennessee at Chattanooga, 2022. [Online]. Available: https://scholar.utc.edu/theses/740/

[6] S. Tuli, S. Tuli, G. Casale, and N. R. Jennings, "Generative optimization networks for memory efficient data generation," 2021. [Online]. Available: https://arxiv.org/abs/2110.02912

[7] S. Tuli, F. Mirhakimi, S. Pallewatta, S. Zawad, G. Casale, B. Javadi, F. Yan, R. Buyya, and N. R. Jennings, "Ai augmented edge and fog computing: Trends and challenges," *Journal of Network and Computer Applications*, vol. 216, 2023.

[8] S. Skaperas, G. Koukis, I. A. Kapetanidou, V. Tsaoussidis, and L. Mamatas, "A pragmatical approach to anomaly detection evaluation in edge cloud systems," in *International Workshop on Intelligent Cloud Computing and Networking*, ser. ICCN '24, Vancouver, Canada, 2024.

[9] R. Mayer, L. Graser, H. Gupta, E. Saurez, and U. Ramachandran, "Emufog: Extensible and scalable emulation of large-scale fog computing infrastructures," in *2017 IEEE Fog World Congress (FWC)*, Santa Clara, CA, USA, 2017.

[10] A. Coutinho, F. Greve, C. Prazeres, and J. Cardoso, "Fogbed: A rapid-prototyping emulation environment for fog computing," in *2018 IEEE International Conference on Communications (ICC)*, Kansas City, MO, USA, 2018.

[11] J. Hasenburg, M. Grambow, E. Grünewald, S. Huk, and D. Bermbach, "Mockfog: Emulating fog computing infrastructure in the cloud," in *2019 IEEE International Conference on Fog Computing (ICFC)*, Prague, Czech Republic, 2019.

[12] Y. Zeng, M. Chao, and R. Stoleru, "Emuedge: A hybrid emulator for reproducible and realistic edge computing experiments," in *2019 IEEE International Conference on Fog Computing (ICFC)*, Prague, Czech Republic, 2019.

[13] M. Symeonides, Z. Georgiou, D. Trihinas, G. Pallis, and M. D. Dikaiakos, "Fogify: A fog computing emulation framework," in *2020 IEEE/ACM Symposium on Edge Computing*, San Jose, CA, USA, 2020.

[14] N. Akbari, A. N. Toosi, J. Grundy, H. Khalajzadeh, M. S. Aslanpour, and S. Ilager, "icontinuum: an emulation toolkit for intent-based computing across the edge-to-cloud continuum," in *Proceedings of the 2024 IEEE 17th International Conference on Cloud Computing*, ser. CLOUD '24, Shenzhen, China, 2024.

[15] Y. Su, Y. Zhao, C. Niu, R. Liu, W. Sun, and D. Pei, "Robust anomaly detection for multivariate time series through stochastic recurrent neural network," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD'19, Anchorage, AK, USA, 2019.

[16] S. Nedelkoski, J. Bogatinovski, A. K. Mandapati, S. Becker, J. Cardoso, and O. Kao, "Multi-source distributed system data for ai-powered analytics," in *Proceedings of the 8th European Conference on Service-Oriented and Cloud Computing*, ser. ESOCC'20, Crete, Greece, 2020.

[17] J. Levin and T. A. Benson, "Viperprobe: Rethinking microservice observability with ebpf," in *Proceedings of the IEEE 9th International Conference on Cloud Networking*, ser. CloudNet'20, 2020.

[18] S. Pallewatta, V. Kostakos, and R. Buyya, "Microservices-based iot application placement within heterogeneous and resource constrained fog computing environments," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, ser. UCC'19, Auckland, New Zealand, 2019.

[19] ——, "Qos-aware placement of microservices-based iot applications in fog computing environments," *Future Generation Computer Systems*, vol. 131, p. 121–136, 2022.

[20] Z. Li, Y. Zhao, J. Han, Y. Su, R. Jiao, X. Wen, and D. Pei, "Multivariate time series anomaly detection and interpretation using hierarchical inter-metric and temporal embedding," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, ser. KDD '21, Virtual Event, Singapore, 2021.

[21] R. Wu and E. J. Keogh, "Current time series anomaly detection benchmarks are flawed and are creating the illusion of progress (extended abstract)," in *Proceedings of the 2022 IEEE 38th International Conference on Data Engineering*, ser. ICDE'22, 2022.