Enhanced Training of Response Time Anomaly Detectors Using Diffusion Models

Abstract

Machine learning approaches have grown in popularity in recent years as a way to identify anomalies, such as failures or atypical contention, in microservice-based applications. However, continuous delivery of microservices requires frequent updates to these models, which may suffer from time constraints and limited real-world failure data availability. In this paper, we propose a method to mitigate this issue using diffusion models for training data augmentation. Response time data collected using distributed traces is used to train diffusion models leveraging a customised UNet proposed in the paper. The resulting diffusion models can then generate new data to improve the training of response time anomaly detectors. Experiments using the *DeathStarBench* microservices architecture demonstrate that the proposed approach increases the accuracy after training of anomaly detection models by 20%. We further show that classic discriminators are unable to accurately distinguish the response time data produced by our UNet compared to the real traces confirming their plausibility.

ACM Reference Format:

. 2026. Enhanced Training of Response Time Anomaly Detectors Using Diffusion Models. In . ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The microservice-based architecture has been proposed as a modern software development approach, and many companies have adopted it to develop new applications. When this architecture is implemented, the applications are divided into small independent microservices that are designed to handle a specific task or a limited set of tasks [1], and communicate with the other microservices through lightweight communication mechanisms [23]. The microservices within these applications are typically developed by different teams [6] and are frequently updated to remove errors, improve performance, and introduce new features.

However, the high frequency of updates in these microservices results in an increased likelihood of failure since there is always insufficient time for exhaustive testing to identify errors in the source code [28]. In order to address this problem, machine learning methods are increasingly used to identify anomalies during runtime [24]. However, these models may need to be retrained after each microservice release, as the updates change the normal behavior of the applications. Retraining these models is a costly process that requires substantial time and computation resources, and the frequent updates of these microservices lead to a data shortage. This issue becomes even worse since the majority of the limited data are normal instances, and anomalies are the minority because the applications are typically running under normal conditions. This results in the models being unable to identify anomalous data accurately, which restricts their effectiveness.

Data augmentation is an effective technique to improve the performance of machine learning models when there is limited data. By increasing the size and diversity of the training dataset, data augmentation improves classification accuracy and allows the model to generalize well to unseen data [31]. This technique is particularly useful when there is a data shortage or imbalanced datasets since it offers additional data that is similar, but nonetheless non-identical, to the original one and from thus further insights may be learned [14]. Therefore, models that use data augmentation are able to more effectively extract patterns within the dataset, which leads to more reliable and accurate predictions.

Diffusion models are a type of popular generative model in recent years because of their ability to generate realistic data. In this paper, a new method is proposed to use diffusion models for data augmentation in the context of microservice applications, which aims to mitigate the challenges caused by data shortage and imbalanced datasets. Microservice traces are initially collected by running a benchmark, and features, such as the execution time of each span and response time between spans, are extracted from raw traces to create datasets. These datasets are then preprocessed to be suitable for training diffusion models to generate new traces that capture patterns of these datasets. After training, new microservice traces are generated by these diffusion models, which are subsequently augmented into the original datasets to introduce unseen patterns. By using both real and generated data, machine learning models are able to effectively extract patterns of anomalies, which leads to an increased accuracy in anomaly detection.

The rest of this paper is organized as follows. Section 2 presents the related work about anomaly detection for microservice applications. The background knowledge on diffusion models is provided in Section 3. Section 4 presents the problem statement and a motivating example that illustrates the negative effect of imbalanced datasets. Section 5 presents the proposed methodology, and the validation follows in Section 6. Finally, the conclusion is given in Section 7.

2 Related Work

Data augmentation typically involves generating new data. Generative adversarial networks (GANs) contain two networks: a generator and a discriminator. The generator transforms noise into realistic data, while the discriminator determines if its inputs are real or generated [10]. Variational autoencoders (VAEs) are another kind of generative model that utilizes an encoder and a decoder structure for generation[15].

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM

https://doi.org/10.1145/nnnnnnn.nnnnnnn

Diffusion models are one of the most popular deep generative models in recent years, and they show a remarkable ability to generate new data in various fields [30]. It is reported that GANs can generate more realistic data than VAEs, but they suffer from training instability [7]. In contrast, diffusion models generate better data than GANs and are simpler to train [5]. The success of these studies and comparisons demonstrate that diffusion models can gain insight into the datasets and generate high-quality data. Therefore, diffusion models could potentially be applied to generate microservice traces to solve the problem caused by the data shortage and imbalanced datasets.

In recent microservice applications, the number of microservices and their interactions have increased significantly. This poses challenges in determining if microservices in applications have failed. By detecting deviations from normal status, software maintenance teams are capable of determining the cause of failure and ensuring that uninterrupted service is delivered to users.

TraceAnomaly [17] uses the response times of spans to train a variational autoencoder, which attempts to reconstruct the inputs. TraceGra [3] is an anomaly detection method that trains a variational graph autoencoder to reconstruct the inputs. UAC-AD [16] employs contrastive learning for anomaly detection. It trains an autoencoder to reconstruct the input data, and contrastive learning guides the generator in learning normal data patterns and excluding anomalous patterns. Nedelkoski et al. [21] propose a supervised learning method for anomaly detection and classification using an autoencoder and a convolution neural network. Seer [9] uses a deep neural network that comprises convolution layers and LSTM layers to predict the likelihood that a particular microservice is responsible for an anomaly.

The applications of previous methods are limited by their shortcomings when compared to the methods proposed in this paper. All previous approaches used the collected data directly to train machine learning models, and they may suffer from performance issues, as the amount of collected data is limited and highly imbalanced because of the frequent update of microservices. The method proposed in this paper mitigates these challenges by using data augmentation, which involves developing diffusion models to generate new data. It allows models to effectively extract patterns of anomalies with a small amount of training data, which is beneficial for situations where collecting anomalous data is challenging.

3 Background

Diffusion models can be divided into several categories, and one of them is the denoising diffusion probabilistic model (DDPM), which is applied in this paper. The training process of DDPMs consists of a forward process and a backward process. In the forward process, Gaussian noise is gradually added to the input data until it becomes pure Gaussian noise. The backward process starts at the pure Gaussian noise obtained at the end of the forward process. A neural network is trained in this process, and it estimates the noise that needs to be removed to restore the input data.

Let $q(x_0)$ be the distribution of the uncorrupted data, $\beta_t \in (0, 1)$ be a hyperparameter that determines how much Gaussian noise is added to the input data, and a training sample be obtained as $x_0 \sim q(x_0)$. The definition of the forward process [13] is

$$q(x_t|x_0) = \mathcal{N}(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t)I) \quad \forall t \in \{1, 2, 3, ..., T\} \tag{1}$$

where $\alpha_t = 1-\beta_t$ and $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$, T is the number of time steps in the forward process, I is an identity matrix with the same size as the input data, and $\mathcal{N}(x; \mu, \sigma)$ is the normal distribution that outputs x with mean μ and variance σ. If the input data x_0 is provided, the corrupted data x_t is obtained by

$$x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon \tag{2}$$

where $\epsilon \sim \mathcal{N}(0, I)$.

The backward process starts from the pure noise $p_T = \mathcal{N}(x_T; 0, I)$, which is obtained at the end of the forward process. A neural network is trained in this process, and it estimates the noise that needs to be removed to restore the input data. As Gaussian noise is introduced during the forward process, the backward process is also defined by a mean μ_θ and a variance $\beta_t I$

$$p_\theta(x_{t-1}|x_t) = \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t), \beta_t I) \tag{3}$$

where θ represents the parameters of the neural network trained in the backward process. Ho et al. [13] suggest that the model only learns to predict the mean and set the variance dependent on time steps. The suggestion is subsequently validated, and it also found that the mean has a more decisive influence than the variance [22]. By applying this simplification, the the relationship between x_{t-1} and x_t is

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}}(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(x_t, t)) + \sqrt{\beta_t}\epsilon \tag{4}$$

A loss function is required to train the neural network in the backward process, and Ho et al. [13] suggests that it is advantageous to use the loss function below for training

$$L = ||\epsilon_t - \epsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, t)||^2 \tag{5}$$

where ϵ_t is Gaussian noise sampled at time step t. Equation (5) indicates that the objective of training the neural network is to minimize the mean squared error (MSE) between the noise introduced during the forward process and the predicted one.

A common architecture to implement a diffusion model is to build a UNet [25], which is initially designed for image segmentation tasks where the sizes of inputs and outputs are identical. This is similar to diffusion models where the input is corrupted images, and the model predicts the amount of noise that should be removed to restore the original image. The UNet achieves this goal by extracting features using down-sampling layers and reconstructing the input data by up-sampling layers. Residual connections could be introduced to preserve the information between the down-sampling and up-sampling parts. There are studies using UNet for anomaly detection in medical areas [32], but to the best of our knowledge it has not been used in microservices for response time anomaly detection. This contribution is developed in the present paper.

4 Motivation

4.1 Problem Statement

4.1.1 Environment. We consider a microservice application that contains S services packaged into C containers. These containers are assumed to be deployed on N machines with different hardware

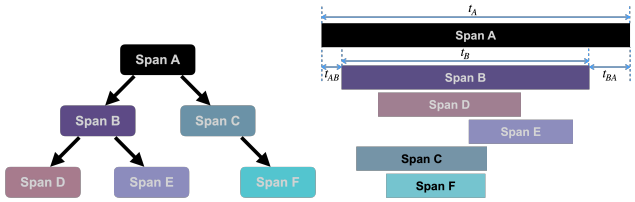
configurations. The system is monitored via distributed tracing with the goal of detecting anomalies in the microservices. The collected traces from the monitoring toolchain are assumed to be used as an input to a machine learning model for anomaly detection. If the model detects any deviation from the normal status for one or more microservices, an error message is generated, which enables the maintenance teams to investigate and address the problem.

On top of the above, some additional assumptions used in this paper are as follows. First, some container orchestration tools may dynamically migrate containers within the distributed system, we assume instead them to be pinned upon first deployment. Next, we assume that training and production runs are on similar technical environments, so that training data obtained via testing may be deemed as representative of production. Lastly, we assume that microservice response time depend only on the application and the technical infrastructure it runs in, not on human factors such as the delay a human imposes for manually supplying data to a process. Hence, the latter delays are not considered as part of the system response time.

4.1.2 Feature Extraction. In the paper, we focus on continuous features that describe latencies observed in distributed tracing spans. While in general anomaly detection can rely upon multiple continuous, discrete and categorical features, a broad catalogue of techniques exists from machine learning for blackbox identification of anomalies. However, response times are fairly different from most other features in that they are closely related to the software architecture and software call inter-dependencies, thus they can additional benefit from correlation to the graph topology of the microservices based application. As the latter can therefore leverage a different treatment than other features, we insist specifically only on response time metrics throughout the paper. However, several of the ideas presented may be generalized in principle to embrace a broader spectrum of system metrics.

Response time features for model training are extracted from raw traces after these are collected from the distributed tracing system. The selected features should be sufficient to temporally reconstruct the sequence of the events in the original traces. To illustrate our approach, Figure 1 provides an example of the feature extraction process on a trace consisting of six spans. Span A is the root span and it has two children spans, B and C. Spans B and C also call their children. A tree structure, shown in 1(a), is first built to represent the invocation chain between spans within each trace. Based on the invocation chain, two types of features are then extracted from the trace: the execution time of each span, denoted as t_A and t_B, and the transition time from the parent span A to one of its children B, denoted as t_{AB} and t_{BA}, which represent the time for transition from span A to B and from span B to A, respectively. However, the transition times between spans B and C are not included, as there is no direct invocation between these spans. This extraction rule applies to the remaining microservices as well.

After the extraction, the execution times and response times extracted from all spans in a single trace are used to create a tuple. The tuples obtained from multiple traces are stacked to create a vector for training UNet, represented as $(t_A^{(j)}, t_B^{(j)}, ..., t_{AB}^{(j)}, t_{BA}^{(j)}, ...)$, where the subscripts denote the feature names and superscripts show the index. The features in the resulting vector follow a fixed



(a) Invocation Chain (b) Response Time

Figure 1: Extracting Features from Traces

order, starting with the execution time of the root span, followed by its child spans in sequence. The response times are ordered similarly, starting with the responses between the root span and its children.

4.1.3 Model Training. Training machine learning models typically requires balanced datasets to ensure the model performance is reliable[11]. This indicates that the datasets should consist of 50% normal data and 50% anomalous data for anomaly detection. However, in real-world microservice applications, it is challenging to obtain a balanced dataset that is suitable to train an anomaly detection model with high accuracy.

Some solutions have been proposed to mitigate the problem caused by imbalanced datasets, but they are ineffective in this case. Software fault injection evaluates the system behavior by deliberately injecting faults into the system to identify errors during development [19]. This method required carefully selected fault type and injection location to ensure the potential faults are identified and removed. The drawbacks are that there are a significant number of injection locations because of the software complexity, and the source code should be available for the injection [20]. Removing excess normal traces in the collected data to create a balanced dataset is impractical as it leads to a dataset too small for effective training. Machine learning models rely on sufficient data to extract and learn patterns of anomalies, and a small dataset significantly restricts the model performance, resulting in the models struggling to accurately predict whether their inputs are anomalous or not. Over-sampling methods are introduced to address the problems caused by imbalanced datasets. SMOTE [2] is a common technique that creates new training data by interpolating between the minority instances and their neighbors. However, the data generated by SMOTE lies within the range of existing minority data and leads to poor representation of underlying patterns when the data distribution of minority class becomes complex [4]. One-class classification is an anomaly detection technique that is trained by using data from a single class, whereas it is found that this kind of classifier suffers from performance issues in some situations, such as high dimensionality [26]. This indicates that it may not be appropriate for microservice applications, which typically contain a substantial number of microservices.

The anomaly detection model trained by imbalanced datasets can identify normal data accurately as it is the majority class in the dataset, which allows the model to gain insight into the normal data easily. This results in a high overall performance for the models. However, the model is unable to detect anomalies accurately

Table 1: Low vs. Middle workloads: 5% and 1% Anomalous Data (IF = isolation forest, KM = K-means)

Anomal. (%)	Model	Performance			
		Accuracy	Precision	Recall	F1 Score
5%	IF	0.980	1.000	0.581	0.735
	KM	0.970	1.000	0.453	0.624
	MLP	0.988	1.000	0.769	0.869
	SVM	0.987	1.000	0.732	0.845
1%	IF	0.984	1.000	0.402	0.574
	KM	0.983	1.000	0.346	0.514
	MLP	0.997	1.000	0.647	0.785
	SVM	0.995	1.000	0.541	0.702

since anomalous data is the minority class in the dataset, and it is insufficient for the model to extract patterns of error status. This causes low performance in determining whether the input data is anomalous, although the model is capable of identifying normal inputs. Therefore, the overall performance is high, but the model still lacks the ability to detect anomalies, which makes it less useful.

4.2 Illustrative Example

In this section, we illustrate a basic experiment to showcase the reduced affectiveness of microservices anomaly detectors in the presence of imbalanced datasets. In this experiment, four machine learning models are trained using imbalanced datasets that contain only a fraction of 5% or 1% anomalous data points. The results are displayed in Table 1. The inputs to these models are vectors, with rows representing individual traces and columns containing features such as execution and response times. It is evident that the precision of these models is high, indicating that there is no misclassification of normal as anomalous. The reason is that these models are trained using highly imbalanced datasets, and the normal data is the majority of the dataset. Hence, the models are capable of identifying normal data with high accuracy, which leads to the overall performance of these models being high as more than 95% of instances in the testing set are classified without error. However, it is clear that the recall is low, particularly for unsupervised learning algorithms, which becomes lower than 0.5. This low recall indicates a high number of false negative predictions, meaning that a significant number of anomalies are classified as normal data. The reason for this low recall is caused by the highly imbalanced dataset, where the models cannot extract patterns for anomalies effectively. This leads to a low F1 score, which is dominated by recall in this case.

From the results shown in Table 1, it is evident that imbalanced datasets can significantly affect the performance of anomaly detection models and also lead to wrong conclusions about their actual usefulness. When datasets contain a small portion of anomalies, models tend to focus on the majority class but ignore the minority data, which leads to a high accuracy of the majority class. This is revealed by high precision and low recall during evaluation, indicating that models correctly classify the majority of normal instances, whereas they fail to identify a significant number of anomalies. As a result, the overall performance of models is high, but they lack the ability to identify anomalies, which restricts their effectiveness in real-world applications.

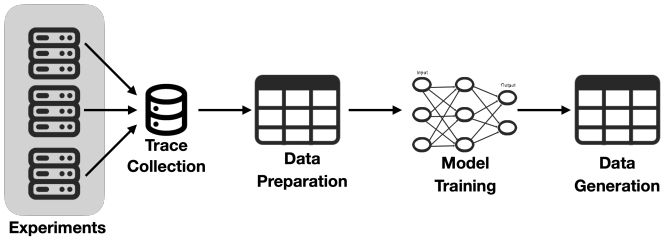


Figure 2: Outline of the proposed methodology

5 Methodology

The outline of the methodology is shown in Figure 2, which consists of five stages. In the first two stages, experiments are performed, and microservice traces are collected. The collected data is then preprocessed to ensure that it is suitable for training machine learning models. The preprocessed data is used to train diffusion models in the fourth stage, and new traces are generated in the last stage.

5.1 Experiments & Trace Collection

Traces are first generated by running the benchmark for a specific period, and the workload is configured by three parameters: number of threads, number of connections, and requests per second. Then, these traces are downloaded from the monitoring tools. To retrieve the desired traces, a URL is constructed with specific parameters to filter the data accordingly. The output of the selected traces varies based on the provided parameters, and these traces are subsequently obtained by sending a request to the configured URL.

5.2 Data Preparation

5.2.1 Outlier Removal. In this paper, machine learning model is employed to extract the distribution of execution time and transition time. It is critical to preprocess the data by removing outliers to ensure the performance of the model is not adversely affected. The outlier removal process involves calculating quartiles, which are values that divide a dataset into four equal parts, each representing a different portion of the data. The first and third quartiles, denoted as Q_1 and Q_3, are obtained in the removal process. The interquartile range (IQR) is then calculated as the difference between Q_3 and Q_1, which measures the statistical dispersion. A value is considered an outlier if it lies beyond three times the IQR from the first or third quartile. This method ensures that extreme values, which could distort the analysis, are effectively excluded.

5.2.2 Feature Selection. The diffusion model developed in this paper is essentially a UNet, comprising several down-sampling and up-sampling layers. These layers perform the operations of division and multiplication by two. Down-sampling layers can handle an odd number of features as input, while a size mismatch occurs during up-sampling, leading to exceptions in model training. Therefore, it is necessary to ensure the number of input features is even to guarantee the down-sampling and up-sampling operations. To achieve this, several features need to be removed from the dataset

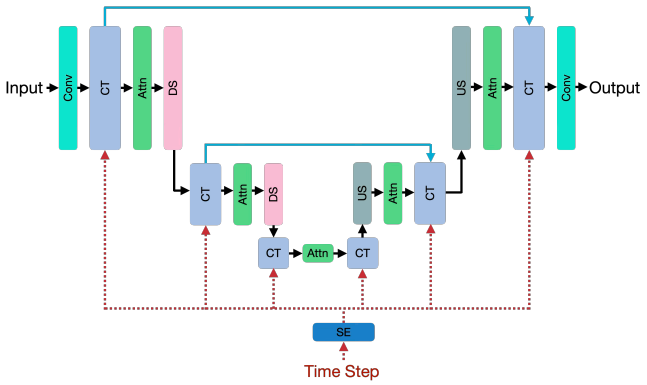


Figure 3: The architecture of the proposed UNet

before training the model to ensure an even number of features after multiple divisions by two.

Principal Component Analysis (PCA) is used as a feature selection technique to identify and eliminate less important features while preserving the information of the dataset. The importance of each feature to the PCs is analyzed which shows the extent to which each feature influences the PCs. By evaluating these influences, the least significant features, which have the least influence, are identified and subsequently removed. This ensures that an even number of features are input into the model, allowing down-sampling and up-sampling operations to be performed without resulting in an exception.

5.2.3 Data Standardization. The ranges of values for features extracted from raw traces vary considerably, which causes the model to be biased towards the features with larger values. To prevent this bias, the data should be standardized before being used as inputs. Equation (6) is used for this purpose, and it standardizes the data by subtracting the mean from each feature and scaling it to unit variance. The reason for using this scaler is that the model predicts the Gaussian noise introduced in the forward process, and the backward process starts from pure Gaussian noise. Standardizing data using this method ensures that all features are on the same scale and that the data and the noise are on the same scale, which is beneficial for the model to effectively learn to restore the input and generate high-quality data.

$$x_n = \frac{x - \mu}{\sigma} \tag{6}$$

5.3 Model Training

5.3.1 Architecture. The neural network built in this paper is a UNet with several modifications, and its basic structure is shown in Figure 3, where *SE* is a sinusoidal embedding block, *Conv* represents convolution layers, *CT* denotes convolution time blocks, *Attn* is attention layers, and *DS* and *US* represent down-sampling layers and up-sampling layers, respectively.

Sinusoidal Embedding Block. Sinusoidal embedding has been used in transformers [27] to encode positional information to address the challenge of encoding sequence order. Transformers process the input in parallel, meaning that the order of elements in the sequence must be explicitly encoded and input to the model, which enables the model to understand the order and relationships between elements. The same encoding technique is also employed in diffusion models since the time steps are critical for the model to understand the amount of noise introduced into the input data. By using sinusoidal embeddings, diffusion models can effectively capture the progression of noise introduced into the input data at each time step, which enables the model to understand the noise introduction in the forward process and the noise removal in the backward process.

Both sine and cosine functions are used to encode particular time steps for every input data in a batch, and the calculations are

$$PE(pos, 2i) = \sin \frac{pos}{10000^{2i/d}} \tag{7}$$

$$PE(pos, 2i+1) = \cos \frac{pos}{10000^{2i/d}} \tag{8}$$

where d is a hyperparameter representing the dimension of the time embedding, pos is the time step that needs to be encoded, and i denotes the dimension index in the embedding. The result of the encoding is a matrix with a size of $batch\_size \times d$, where each row contains the embedding for a specific time step in a mini-batch. This encoding enables the model to recognize which time step it is processing, allowing it to extract the relationship between the time steps and the amount of noise introduced. This matrix is subsequently passed to an MLP with two linear layers and a *GELU* activation function to obtain the final time embeddings TE.

Convolution Time Block. The structure of this block is displayed in Figure 4, and it receives the time embeddings TE from the sinusoidal embedding block and the input data IN. The time embedding is first passed to a *GELU* activation function and a linear layer to match the size of the input data. The embedding is then reshaped and added to the input data as,

$$H = IN + rearrange(MLP(TE)) \tag{9}$$

where H is a matrix representing the latent representation of the output from the hidden layers. This allows the model to process both time and spatial information effectively. Group normalization is employed in the model to normalize the outputs from an activation function. Unlike batch normalization, which presents poor performance when the batch size is small [18], group normalization performs well for small batches, which is the case in this paper. It divides channels into groups and computes the mean and variance of each group, which ensures effective normalization despite the small batch size [29]. The model extracts patterns from the sum of the input data and the time embeddings by passing them to group normalization layers and convolution layers,

$$H = Conv(GN(GELU(Conv(GN(H))))) \tag{10}$$

where GN and $Conv$ represent group normalization layers and convolution layers, respectively. Additionally, there is a residual connection between the input and the output in this block. It is a powerful technique in neural networks to address the problems of vanishing gradients and improve training efficiency [12]. This connection allows the input data to bypass multiple layers and be added directly to the output, which facilitates the preservation of

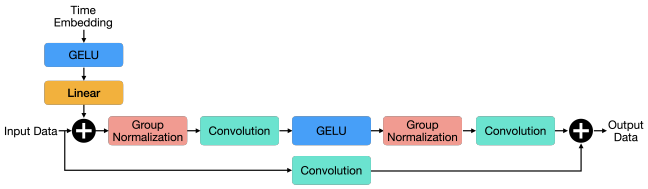


Figure 4: Convolution Time Block

Algorithm 1 The Training Process

1: **while** $epoch < n_{epochs}$ **do**
2: $x_0 \sim q(x_0)$ ▷ Obtain an input data from the dataset
3: $t \sim \mathcal{U}(\{1, 2, 3, ..., T\})$ ▷ Sample time steps
4: $\epsilon \sim \mathcal{N}(0, I)$ ▷ Sample noise for the forward process
5: $\min ||\epsilon_t - \epsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, t)||^2$ ▷ Loss function in (5)
6: **end while**

the original information. The residual connection in this block is expressed as,

$$OUT = Conv(IN) + H \tag{11}$$

where OUT represents the output of this block.

Down-Sampling and Up-Sampling. The size of the input data is divided by two in down sampling layers and the size is doubled in up sampling layers. The reason for performing down-sampling and up-sampling is to enable the model to learn to preserve the most significant information and subsequently use the information to restore the input.

5.3.2 Training. Hyperparameters are configured at the start of the training, and the training process is shown in Algorithm 1. The input data is treated as an square image for two reasons. First, the size of the image should matches the number of features in the dataset. Second, the blocks in the neural network, including convolution, down-sampling, and up-sampling, are designed to handle square images by default. In line two, the *DataLoader* collects samples from the dataset, and they are reshaped to the required input size. This ensures that the input size is compatible with the architecture of the neural network. In line three, several timesteps are then sampled from the uniform distribution, and the corresponding noise is obtained in line four to corrupt the input images. In line five, the neural network predicts the noise added based on the corrupted images and the time step, and its performance is improved by minimizing the MSE loss between real noise and the predicted noise.

5.4 Data Generation

New traces are then generated by the trained diffusion models using the process shown in Algorithm 2. The generation starts by sampling Gaussian noise, as shown in line one. The diffusion models predict the amount of noise that should be removed from the sampled Gaussian noise in line eight. After iterating all timesteps, the generated traces x_0 are obtained, and the values within x_0 are scaled back to normal ranges using the *StandardScaler* described in 5.2.3.

Algorithm 2 The Generation Process

1: $x_T \sim \mathcal{N}(0, I)$ ▷ Sample noise for generation
2: **for** `t = T, ..., 2, 1` **do**
3: **if** $t > 1$ **then**
4: $\epsilon \sim \mathcal{N}(0, I)$ ▷ Sample noise for the backward process
5: **else**
6: $\epsilon = 0$ ▷ No noise for the last step
7: **end if**
8: $x_{t-1} = \frac{1}{\sqrt{\alpha_t}}(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(x_t, t)) + \sqrt{\beta_t}\epsilon$ ▷ Denoise via (4)
9: **end for**
10: **return** x_0

6 Validation

6.1 Environment

The datasets used in this paper are collected from a *SocialNetwork* application, which is one of the end-to-end applications offered by the open-source benchmark *DeathStarBench* [8]. *SocialNetwork* is a microservice application designed to simulate a social networking platform, enabling users to follow/unfollow users, create/delete posts, and send messages to another user. This application leverages the benefits of microservice architecture. It utilizes various techniques such as C++ and Python for application development, *MongoDB* for persistent storage, and *Memcache* for caching. Each microservice communicates to others using synchronous communication and is packaged into separate Docker containers, allowing easy scalability and updates to the latest version. In addition, *Jaeger* is integrated into the application, which simplifies the process of data preparation.

The benchmark is deployed on three machines to simulate the operations of a social network application under different conditions, and the hardware configurations of three machines are shown in Table 2. Two of the machines are used to execute tasks related to the social network application, while the third machine is dedicated to running *Jaeger* for network trace collection and storage. The container of each service is pinned to a specific machine to ensure that it is always executed on the same machine throughout the experiments. This setup ensures data consistency, as containers are always executed in the same environment, avoiding issues caused by being moved between different machines.

The benchmark runs for 60 minutes during the data collection stage, and the number of replicas of each container is set to one. The benckmark provides three types of workloads: *compose-post*, *read-home-timeline*, and *read-user-timeline*. The benchmark offers four *lua* scripts for continuously sending requests to the benchmark to simulate these workloads. Three of these scripts correspond to three types of workloads, and the fourth script sends a mixture of three workloads and their ratio can be adjusted. In this paper, the fourth *lua* script is used in this paper, with equal percentages for all three workloads. However, when downloading traces, only the *compose-post* workload is selected, which contains 31 spans. The other two workloads are excluded because they are relatively simple: *read-user-timeline* contains seven spans, and *read-home-timeline* includes six spans.

In order to introduce variations into traces collected from the benchmark, three levels of workload, low, medium, and high, are

Table 2: Hardware Configuration of Machines

No.	*CPU*	*Memory*	*Disk*
1	Intel(R) Xeon(R) CPU E5-2470 v2 @ 2.40GHz	32 GB	931.5 GB
2	Intel(R) Xeon(R) CPU E5-2440 v2 @ 1.90GHz	32 GB	931.5 GB
3	Intel(R) Xeon(R) CPU E3-1230 v5 @ 3.40GHz	16 GB	931.5 GB

defined, which corresponds to 20%, 40%, and 60% resource usage. These levels are used to simulate different operational states of the benchmark, allowing for a more comprehensive analysis of performance under varying conditions.

6.2 Evaluation

6.2.1 Metrics. The evaluation metrics selected for this paper are accuracy, precision, recall, and F1 score. We use these metrics to determine whether a trace is normal or anomalous, as it is a binary classification task. Accuracy shows the overall performance of baseline models, and it is calculated as

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{12}$$

where TP and TN represent true positive and true negative predictions, while false positive and false negative predictions are denoted as FP and FN, respectively. However, accuracy is insufficient to reveal detailed insights as the training datasets in this paper are highly imbalanced. Therefore, precision and recall are also included to assess the models' ability to identify true normal traces, and they are expressed as,

$$precision = \frac{TP}{TP + FP} \tag{13}$$

$$recall = \frac{TP}{TP + FN} \tag{14}$$

In addition, the F1 score is selected to provide more comprehensive results of the models' performance in anomaly detection, as it combines both precision and recall into one metric. It is computed as

$$F1 = \frac{2 \cdot precision \cdot recall}{precision + recall} \tag{15}$$

6.2.2 Baselines. Two unsupervised algorithms, Isolation Forest (IF) and K-Means, as well as two supervised learning algorithms, SVM and MLP, are selected as the baseline models for anomaly detection, and their configurations are shown in Table 4. The reason for selecting both supervised and unsupervised learning algorithms is that traces collected from real-world applications are unlabeled, making it challenging to apply supervised learning methods. Unsupervised learning algorithms detect anomalies without requiring labeled data, which is suitable for this case. On the other hand, when labeled datasets are available, supervised learning algorithms are expected to provide more accurate performance as they are trained with labeled data, which is beneficial for distinguishing normal and anomalous data.

Table 3: Hyperparameter Configuration

Hyperparameter	*Value*
Number of Time Steps (T)	1000
Beta Start (β_1)	1×10^{-4}
Beta End (β_T)	0.02
Beta Schedule Method	Linear
Number of epochs (n_{epochs})	500
Optimizer	Adam
Learning Rate	1×10^{-3}
Batch Size	10
Time Embedding Dimension	128

Table 4: Hyperparameter Configuration for Baseline Models

Model	*Hyperparameter*
IF	n_estimators = 400
K-means	n_clusters = 2
SVM	kernel = linear
MLP	layers=10, solver=adam, activation=sigmoid, hidden_layer_size = 128

In order to evaluate whether diffusion models are able to effectively mitigate the problems caused by imbalanced datasets, diffusion models are trained using the hyperparameters shown in Table 3, and traces are generated using these models during the validation phase. After the generation, these traces are augmented into the original dataset at various percentages to train baseline models and determine if their performance in anomaly detection improved. The effectiveness of the augmentation is assessed by comparing the evaluation metrics before and after the augmentation.

6.3 Performance: 10% Anomalous Data

In order to evaluate the effect of augmenting traces generated by diffusion models, two imbalanced datasets containing 10% high or middle workload are first used to train baseline models, and the results are shown in Figure 5 and Figure 6, respectively. In these figures, the X-axis represents the augmentation percentage, from 0% to 50%, while the Y-axis represents the evaluation metrics. The baseline models trained by original datasets are represented by 0% augmentations. The performance at this point shows the ability of the baseline models to identify normal and anomalous traces, and it would be compared with models trained with data augmentation.

In these figures, it is evident that precision is close to 1.0, which indicates that the number of false positive predictions is low, and the model is able to identify normal traces with high accuracy. The reason is that the models are trained using highly imbalanced datasets, with a significantly greater number of normal traces than anomalous traces. This results in the models becoming highly accurate in predicting normal data.

Since the baseline models are able to accurately identify normal traces, which comprise 90% of the datasets, the minimum accuracy should be 90% even if the models predict all the inputs as normal. On the other hand, the recall demonstrates that these models can identify anomalous traces to some extent, suggesting that they

may classify some of anomalies as normal instances. Therefore, the accuracy is around 96% or higher in these results. In addition, the F1 score provides a harmonic mean of the precision and recall, but it is mainly affected by the recall in this case as precision is close to one.

After augmenting generated traces into the original datasets, the performance of baseline models improves, and most of them achieve the highest performance when 20% of traces are generated by diffusion models. It is clear that the recall for all models increases, which suggests that the number of false negative predictions decreases, meaning that the models are capable of identifying anomalous traces more accurately. This increase in recall leads to an increase in the F1 score, which is heavily influenced by recall in this case. However, the precision is not affected as it shows the ability of the models to identify normal traces. In addition, the influence of augmenting generated traces on the accuracy is small, as the anomalous traces are the minority class in the datasets, and the majority class dominates the accuracy. Therefore, augmenting generated traces results in an increase in recall and F1 score, but the precision and accuracy are not largely affected.

Although the performance increases from 0% to 20% augmentation, it decreases when the augmentation percentage increases from 20% to 50%. The potential reason for this decrease is that augmenting too many generated traces may change the original data distribution. The baseline models learn the patterns that do not accurately reflect the original distribution, which results in a decrease in the effectiveness of data augmentation and impacts the performance of the baseline models.

6.4 Sensitivity Analysis

More experiments are performed in order to validate the effectiveness of augmenting generated traces in extreme cases, and the results are shown in Table 5 and Table 6. There are two types of anomalous traces that the baseline models should be able to identify, high and middle, which are shown in the left-most column. The baseline models are trained using datasets with 5% or 1% anomalous data. For each baseline model in these cases, its performance at 0% augmentation and the highest performance after augmentation are shown in the table.

As shown in Figure 5 and Figure 6, the baseline models are capable of identifying normal traces with high accuracy when they are trained using datasets with 90% normal data. This conclusion is also valid when analyzing results in the table, where the precision of these models becomes 1.0, suggesting that there are no false positive instances. This means that the models perform better in identifying normal traces since they are trained by more imbalanced datasets, where the distribution is heavily skewed towards normal instances. As a result, the accuracy of these models is expected to be above 95% when trained with 5% anomalous data and above 99% with 1% anomalous data, which is true in most cases. However, the accuracy is slightly below this threshold in some cases when unsupervised learning models are trained using 1% anomalous data. This occurs because these models struggle to detect anomalies with such a small amount of anomalous training data. The low recall in these cases indicates that the models fail to extract patterns for anomalies, and there is a significant number of false negative instances where anomalies are classified as normal. As a result, the recall and accuracy are low.

It is evident that performance for all models increases after augmenting generated traces into the original datasets. The majority of these models achieve the highest performance when 20% of data is generated by diffusion models. Recall for all these cases increases, indicating that the number of false negative predictions reduces, and the models become more capable of identifying anomalous traces. This increase in recall results in a slight improvement in accuracy, as the total number of correct predictions increases, but it is still largely determined by normal traces, which is sufficiently accurate without augmentation. The impact of increased recall is also shown in the F1 score, which improves as the models become more effective at identifying anomalies. Therefore, augmenting generated traces leads to an increase in recall and F1 score, while precision and accuracy are slightly affected.

6.5 Discriminators

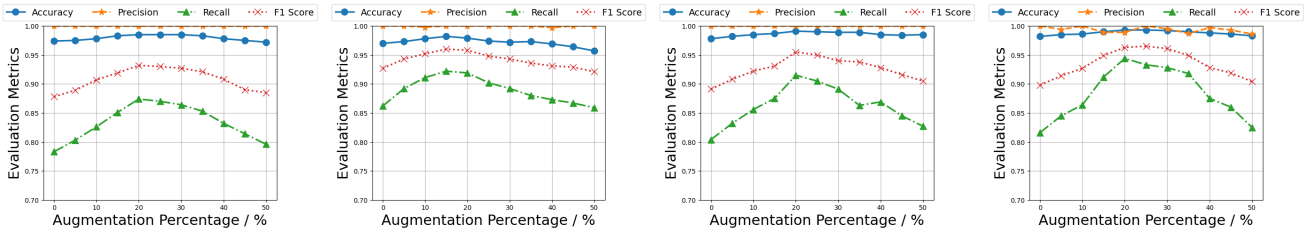
The traces generated by diffusion models are analyzed from another aspect, where discriminators are trained using balanced datasets containing real and generated data, and the results are displayed in Figure 7. Unlike previous experiments that focus on using generated traces to augment original datasets to improve the performance of baseline models. The goal of this experiment is to evaluate if the baseline models are able to identify whether their inputs are real or generated. This evaluation provides insight into the quality of the generated traces.

To evaluate the performance of discriminators, the generated data is defined as positive class, and real data is defined as negative class. It is evident that accuracy for all unsupervised learning models is around 0.5, with even lower performance when IF is used to distinguish middle workload traces. This suggests that these models are not able to predict the inputs correctly. On the other hand, supervised learning models show better performance, achieving around 0.65, when they are predicting middle and high workloads. However, their performance is still lower than the results for data augmentation, indicating that although supervised learning models outperform unsupervised learning models, they still have difficulties in identifying whether their input is real or generated.

Moreover, precision and recall for all unsupervised learning models are low, which suggests that they struggle to predict either generated or real traces accurately. When supervised learning models are used for low and high workloads, either precision or recall is high, but the other metric is low, leading to a low F1 score. This indicates that these models may perform well in predicting real or generated traces, whereas their performance is low for another class. This imbalance shows that these models perform poorly and cannot distinguish real and generated traces, which suggests that the traces generated by diffusion models are of high quality and are highly realistic.

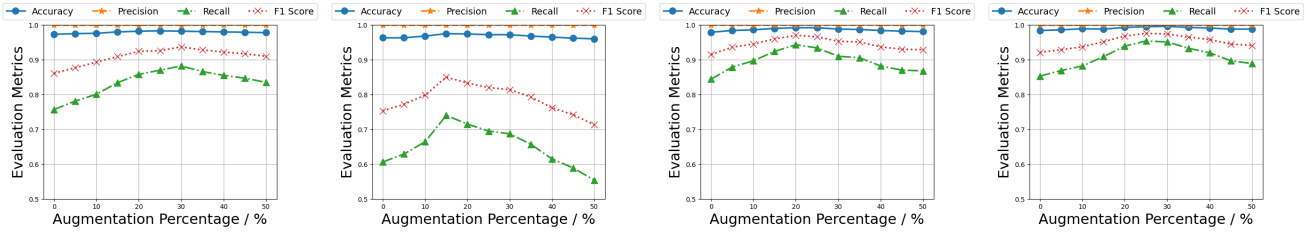
7 Conclusion

Microservice applications are updated frequently, but this increases the likelihood of failure due to a lack of time for for exhaustive testing to identify errors. Machine learning algorithms are introduced to detect anomalies for microservice applications during runtime,



(a) IF (b) K Means (c) SVM (d) MLP

Figure 5: Performance: High vs. Low, 10% Anomalous Data



(a) IF (b) K Means (c) SVM (d) MLP

Figure 6: Performance: Middle vs. Low, 10% Anomalous Data

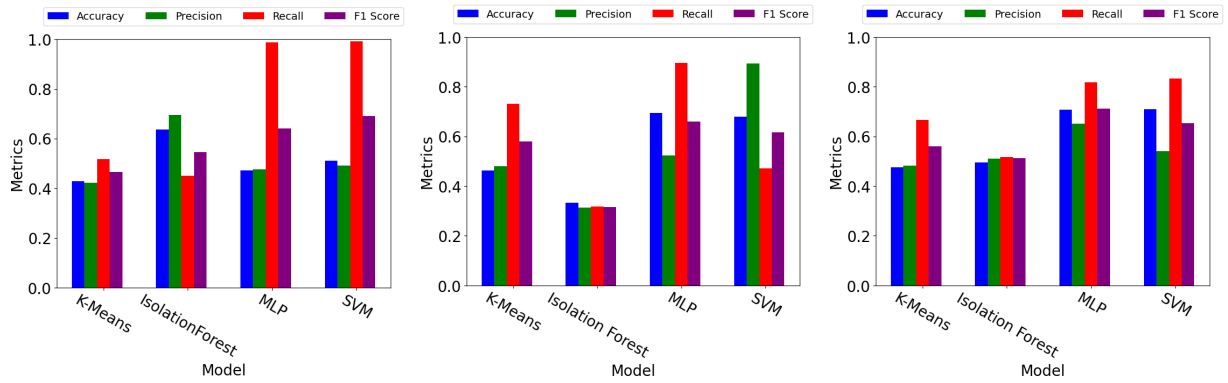
Table 5: Low vs. High workloads: 5% and 1% Anomalous Data

Anomal. (%)	Model	Augmentation	Before Augmentation				After Augmentation			
			Accuracy	Precision	Recall	F1 Score	Accuracy	Precision	Recall	F1 Score
5%	IF	20%	0.980	1.000	0.709	0.828	0.989	1.000	0.813	0.897
	K-means	15%	0.980	1.000	0.609	0.757	0.985	1.000	0.723	0.839
	MLP	20%	0.988	1.000	0.754	0.859	0.994	1.000	0.912	0.952
	SVM	20%	0.985	1.000	0.716	0.834	0.991	1.000	0.828	0.906
1%	IF	20%	0.987	1.000	0.486	0.654	0.994	1.000	0.643	0.782
	K-means	20%	0.985	1.000	0.462	0.632	0.995	1.000	0.669	0.802
	MLP	25%	0.997	1.000	0.722	0.838	0.998	1.000	0.846	0.916
	SVM	20%	0.995	1.000	0.666	0.800	0.998	1.000	0.789	0.882

Table 6: Low vs. Middle workloads: 5% and 1% Anomalous Data

Anomal. (%)	Model	Augmentation	Before Augmentation				After Augmentation			
			Accuracy	Precision	Recall	F1 Score	Accuracy	Precision	Recall	F1 Score
5%	IF	20%	0.980	1.000	0.581	0.735	0.987	1.000	0.703	0.825
	K-means	20%	0.970	1.000	0.453	0.624	0.983	1.000	0.658	0.793
	MLP	25%	0.988	1.000	0.769	0.869	0.994	1.000	0.906	0.950
	SVM	20%	0.987	1.000	0.732	0.845	0.990	1.000	0.857	0.923
1%	IF	20%	0.984	1.000	0.402	0.574	0.996	1.000	0.630	0.773
	K-means	25%	0.983	1.000	0.346	0.514	0.996	1.000	0.645	0.785
	MLP	20%	0.997	1.000	0.647	0.785	0.998	1.000	0.782	0.878
	SVM	20%	0.995	1.000	0.541	0.702	0.998	1.000	0.736	0.848

whereas they suffer from performance issues because of data shortage and imbalanced datasets. Data augmentation is an effective approach to mitigate this problem. In this paper, *DeathStarBench* is used for data collection. Then, features such as execution time and response time are extracted from traces to create datasets. Diffusion models are trained to gain insight into these datasets and generate



(a) Low Workload (b) Middle Workload (c) High Workload

Figure 7: Discriminator results

new data. The generated traces are augmented into imbalanced datasets, and the performance of baseline models improves by 20%, indicating that using the diffusion models to generate traces can enhance the model performance in anomaly detection for microservices. In addition, discriminators are trained using both real and generated traces, and the results show that they are unable to identify whether their inputs are real or generated, which suggests that the traces generated by diffusion models are realistic and of high quality.

References

[1] Işıl Karabey Aksakalli, Turgay Çelik, Ahmet Burak Can, and Bedir Tekinerdoğan. 2021. Deployment and communication patterns in microservice architectures: A systematic literature review. *Journal of Systems and Software* 180 (2021), 111014.

[2] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357.

[3] Jian Chen, Fagui Liu, Jun Jiang, Guoxiang Zhong, Dishi Xu, Zhuanglun Tan, and Shangsong Shi. 2023. TraceGra: A trace-based anomaly detection for microservice using graph deep learning. *Computer Communications* 204 (2023), 109–117.

[4] Wangzhi Dai, Kenney Ng, Kristen Severson, Wei Huang, Fred Anderson, and Collin Stultz. 2019. Generative oversampling with a contrastive variational autoencoder. In *2019 IEEE International Conference on Data Mining (ICDM)*. IEEE, 101–109.

[5] Prafulla Dhariwal and Alexander Nichol. 2021. Diffusion models beat gans on image synthesis. *Advances in neural information processing systems* 34 (2021), 8780–8794.

[6] Qingfeng Du, Tiandi Xie, and Yu He. 2018. Anomaly detection and diagnosis for container-based microservices with performance monitoring. In *Algorithms and Architectures for Parallel Processing: 18th International Conference, ICA3PP 2018, Guangzhou, China, November 15-17, 2018, Proceedings, Part IV 18*. Springer, 560–572.

[7] Mohamed El-Kaddoury, Abdelhak Mahmoudi, and Mohammed Majid Himmi. 2019. Deep generative models for image generation: A practical comparison between variational autoencoders and generative adversarial networks. In *Mobile, Secure, and Programmable Networking: 5th International Conference, MSPN 2019, Mohammedia, Morocco, April 23–24, 2019, Revised Selected Papers 5*. Springer, 1–8.

[8] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 3–18.

[9] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*. 19–33.

[10] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2020. Generative adversarial networks. *Commun. ACM* 63, 11 (2020), 139–144.

[11] Manish Goyal and Raman Kumar. 2020. Machine learning for malware detection on balanced and imbalanced datasets. In *2020 International Conference on Decision Aid Sciences and Application (DASA)*. IEEE, 867–871.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[13] Jonathan Ho, Ajay Jain, and Pieter Abbeel. 2020. Denoising diffusion probabilistic models. *Advances in neural information processing systems* 33 (2020), 6840–6851.

[14] Xiaoyu Jiang and Zhiqiang Ge. 2020. Data augmentation classifier for imbalanced fault classification. *IEEE Transactions on Automation Science and Engineering* 18, 3 (2020), 1206–1217.

[15] Diederik Kingma, Tim Salimans, Ben Poole, and Jonathan Ho. 2021. Variational diffusion models. *Advances in neural information processing systems* 34 (2021), 21696–21707.

[16] Hongyi Liu, Xiaosong Huang, Mengxi Jia, Tong Jia, Jing Han, Ying Li, and Zhonghai Wu. 2024. UAC-AD: Unsupervised Adversarial Contrastive Learning for Anomaly Detection on Multi-modal Data in Microservice Systems. *IEEE Transactions on Services Computing* (2024).

[17] Ping Liu, Haowen Xu, Qianyu Ouyang, Rui Jiao, Zhekang Chen, Shenglin Zhang, Jiahai Yang, Linlin Mo, Jice Zeng, Wenman Xue, et al. 2020. Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 48–58.

[18] Chunjie Luo, Jianfeng Zhan, Lei Wang, and Wanling Gao. 2020. Extended batch normalization. *arXiv preprint arXiv:2003.05569* (2020).

[19] Roberto Natella. 2011. *Achieving Representative Faultloads in Software Fault Injection*. Ph. D. Dissertation. University of Naples Federico II, Italy.

[20] Roberto Natella, Domenico Cotroneo, Joao A Duraes, and Henrique S Madeira. 2012. On fault representativeness of software fault injection. *IEEE Transactions on Software Engineering* 39, 1 (2012), 80–96.

[21] Sasho Nedelkoski, Jorge Cardoso, and Odej Kao. 2019. Anomaly detection and classification using distributed tracing and deep learning. In *2019 19th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGRID)*. IEEE, 241–250.

[22] Alexander Quinn Nichol and Prafulla Dhariwal. 2021. Improved denoising diffusion probabilistic models. In *International conference on machine learning*. PMLR, 8162–8171.

[23] João Nobre, EJ Solteiro Pires, and Arsénio Reis. 2023. Anomaly Detection in Microservice-Based Systems. *Applied Sciences* 13, 13 (2023), 7891.

[24] Sushant Kumar Pandey, Ravi Bhushan Mishra, and Anil Kumar Tripathi. 2021. Machine learning based methods for software fault prediction: A survey. *Expert Systems with Applications* 172 (2021), 114595.

[25] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. In *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*. Springer, 234–241.

[26] Naeem Seliya, Azadeh Abdollah Zadeh, and Taghi M Khoshgoftaar. 2021. A literature review on one-class classification and its potential applications in big data. *Journal of Big Data* 8 (2021), 1–31.

[27] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[28] Muhammad Waseem, Peng Liang, Gastón Márquez, and Amleto Di Salle. 2020. Testing microservices architecture-based applications: A systematic mapping study. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 119–128.

[29] Yuxin Wu and Kaiming He. 2018. Group normalization. In *Proceedings of the European conference on computer vision (ECCV)*. 3–19.

[30] Ling Yang, Zhilong Zhang, Yang Song, Shenda Hong, Runsheng Xu, Yue Zhao, Wentao Zhang, Bin Cui, and Ming-Hsuan Yang. 2023. Diffusion models: A comprehensive survey of methods and applications. *Comput. Surveys* 56, 4 (2023), 1–39.

[31] Suorong Yang, Weikang Xiao, Mengchen Zhang, Suhan Guo, Jian Zhao, and Furao Shen. 2022. Image data augmentation for deep learning: A survey. *arXiv preprint arXiv:2204.08610* (2022).

[32] Ramin Yousefpour Shahrivar, Fatemeh Karami, and Ebrahim Karami. 2023. Enhancing Fetal Anomaly Detection in Ultrasonography Images: A Review of Machine Learning-Based Approaches. *Biomimetics* 8, 7 (2023), 519.