Adaptive Performance Regression Detection Using A Semi-Supervised Siamese Network

Yongqian Sun[†¶], Mengyao Li[†], Xiao Xiong[†], Lei Tao[†], Yimin Zuo[†], Wenwei Gu[†],
Shenglin Zhang[†§*], Junhua Kuang[†], Yu Luo[†], Huandong Zhuang[††], Bowen Deng[††], Dan Pei[‡]
[†]Nankai University, {sunyongqian, wwgu, zhangsl}@nankai.edu.cn,
{limengyao, xiongxiao, leitao, 2213023, 2120240797, 2111934}@mail.nankai.edu.cn
[††]Huawei Cloud, {zhuanghuandong, dengbowen10}@huawei.com
[‡]Tsinghua University, peidan@tsinghua.edu.cn
[§]Key Laboratory of Data and Intelligent System Security, Ministry of Education, China
[¶]Tianjin Key Laboratory of Software Experience and Human Computer Interaction

***Abstract*—Timely detection of performance regression issues is critical to ensuring the stability and user experience of software systems. Traditional methods often rely on high-quality annotated data or data distribution assumptions, which cannot effectively adapt to performance changes in dynamic workload environments. To solve this problem, we propose *DynamicRegress*, a performance regression detection method based on Siamese network and semi-supervised learning. *DynamicRegress* integrates multi-dimensional key performance indicators (KPIs) with workload context to accurately characterize system states and detect performance regressions in real time. By employing a dual weight-shared LSTM network, *DynamicRegress* reduces training complexity while retaining strong feature extraction capabilities. Data augmentation and a weighted loss function are incorporated to enhance the learning of minority regression cases, mitigating the class imbalance issue. Additionally, a semi-supervised learning strategy generates high-quality pseudo-labels to expand the training dataset, effectively addressing the challenge of limited labeled data. Experiments on production data from a top-tier global cloud service provider demonstrate that *DynamicRegress* achieves a superior *F1 Score* of 0.958 (outperforming the best baseline method by 0.282) while maintaining a low detection latency of 0.006 seconds per KPI pair. *DynamicRegress* provides a robust adaptive solution for performance regression detection in dynamic and complex software systems, and we have made the code publicly available to facilitate further research.**

***Index Terms*—Performance Regression Detection, Siamese Network, Semi-supervised Learning**

I. Introduction

In the digital age, software system stability and performance are critical to ensuring user experience. Sudden increases in traffic, iterative version changes, and other factors may cause performance regression, which refers to the deterioration of a system's performance and manifests as increased response time, reduced throughput, etc [1]. The impact of performance regression on user experience and company costs is significant. Previous research has shown that increasing load times from one to three seconds can negatively impact user exploration while increasing exit rates by 32% [2]. British Broadcasting Corporation (BBC) loses 10% of its users for every extra second it takes to load content [3]. Besides, in 2013, Amazon's website

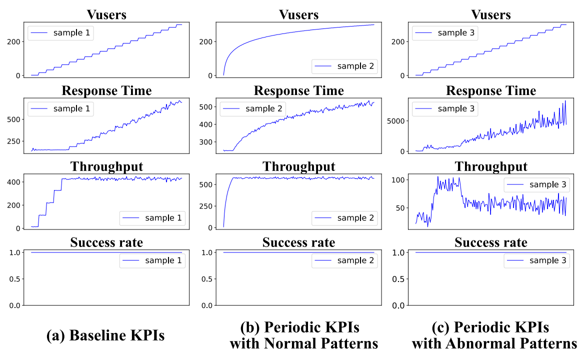

Fig. 1: Three-Version Interface Load Test Using Vusers: Measuring Response Time, Throughput, and Success Rates.

experienced a 45-minute outage, resulting in a loss of $5 million in business revenue [4]. Therefore, performing performance regression detection is essential after a new software version is released [5].

With each software version update, necessary testing is conducted to detect potential performance regressions by analyzing collected KPIs. Assuming no significant functional or architectural changes, baseline KPIs (Fig. 1 (a)) from the previous version are used to represent the expected behavior of each interface under stable conditions. Periodic KPIs are compared against these baselines, and significant deviations, such as increased response time, reduced throughput, or unstable success rates, indicate performance regressions caused by factors like increased load or resource exhaustion. (1) Fig. 1 (c) illustrates a regression scenario where solely relying on the success rate can lead to misjudgments. Despite maintaining a high success rate, the increased response time and reduced throughput signal performance degradation under higher load conditions. (2) Conversely, Fig. 1 (b) illustrates a stable performance scenario under a logarithmic ramp-up load pattern, which exhibits behavior consistent with the baseline in Fig. 1 (a). Specifically, the throughput increases rapidly at first and then

[*]Shenglin Zhang is the corresponding author.

stabilizes, while the response time remains steady initially and then gradually rises. These trends reflect a normal adaptation to increasing load rather than performance degradation. These cases emphasize the importance of considering workload information and echo insights from recent industry frameworks such as SuperBench [6], which highlight that performance regressions are often workload-dependent. However, challenges remain: monitoring produces large volumes of data, but the annotation process is resource-intensive, leading to limited labeled samples. This issue is further compounded by the diversity of software functionalities and interfaces, resulting in sparse labeled data for individual interface types. Addressing these challenges is critical to accurately identify performance regressions and manage the nuanced scenarios depicted in Figs. 1 (b) and 1 (c).

There has been extensive research on this issue. Statistical approaches based on the Kolmogorov-Smirnov test [7] (KS) and Mann-Whitney U test [8] (MWU) were applied to analyze distribution differences between two-phase KPIs. While these unsupervised methods make it efficient and low-cost for detecting distributional shifts between samples, they are also susceptible to numerical factors such as mean, median, and variance. As shown in Figs. 1 (a) and Figs. 1 (b), both tests reported p-values of 0 for throughput, and 0 (KS) / 0.0004 (MWU) for response time, suggesting significant differences. In reality, no performance regression occurred—these differences stemmed from normal variations in load conditions. This reveals a key limitation: such tests may trigger false alarms when facing expected, non-critical value shifts. Other methods adopt feature extraction techniques and classifier training to detect anomalous samples. However, these methods also rely on fixed data structures and fail to account for the dynamic changes in metrics caused by user behavior variations. AutoPerf [9] detects performance regressions by clustering similar functional codes and training autoencoders for each cluster, flagging reconstruction errors above a threshold as regressions. While effective for hardware performance counter (HWPC) metrics, its performance declines in real-time scenarios involving service-level metrics, where the need for manual cluster configuration further limits its adaptability in dynamic environments. Siamese Convolutional Neural Networks [10] (SCNNs) have also been used for time series similarity measurement by comparing embeddings extracted via shared convolutional branches. However, SCNNs require the two input sequences to be of equal length, due to the fixed-size outputs of convolution and fully connected layers. While preprocessing techniques like zero-padding or truncation can align sequence lengths, they may introduce noise or discard critical information, thus affecting model accuracy.

Therefore, our goal is to design a performance regression detection method that captures the temporal trends and quickly identifies instances of performance regression. However, it faces the following three challenges:

(1) Comparing variable-length KPIs under Dynamically Changing Loads. In real-world testing, KPI sequences in real-world tests often differ in length due to varying durations or sampling. Methods like SCNNs [10] require equal-length inputs and rely on padding or truncation, which may distort trends. A key challenge is comparing such sequences accurately while preserving temporal and workload-dependent patterns.

(2) Diverse Interfaces and Scarce Labeled Samples per Interface. Modern software systems are often composed of multiple functional interfaces, and different types of interfaces exhibit different normal data patterns. It is impractical to train a separate model for each interface to recognize its normal data patterns because of the scarce normal labeled samples in each type of interface to support the training.

(3) Imbalanced Labeled Samples. Further observations reveal that regression samples are significantly rarer compared to the abundant normal status data, making it challenging to design detection methods that can effectively capture subtle trend changes while maintaining robustness to imbalanced data distributions.

To address the aforementioned challenges, we propose *DynamicRegress*, an end-to-end performance regression detection method based on multi-dimensional KPIs. To address Challenge 1, *DynamicRegress* utilizes multi-dimensional KPIs and workload to represent system status, and it naturally supports variable-length inputs through LSTM encoders, avoiding the need for artificial sequence alignment. To address Challenge 2, *DynamicRegress* adopts a synergistic strategy of dual weights-shared LSTM branches. Inspired by the success of Siamese networks in facial recognition tasks (detailed in Section §II-A3), we apply Siamese networks to our scenario, comparing new data with known data patterns to determine whether the new data belongs to the correct pattern. This weights-shared strategy not only reduces model complexity but also enables learning general and effective feature representations from limited labeled data. To address Challenge 3, *DynamicRegress* uses a semi-supervised learning strategy to expand the diversity of training data while ensuring quality. By applying data augmentation and weighted loss function techniques [11], the quantity and diversity of minority (regression) samples are increased, enabling *DynamicRegress* to focus more on capturing subtle and evolving patterns of minority samples. The combination of these three strategies enables effective performance regression detection with imbalanced labeled data, improving the model's robustness and generalization ability.

In summary, the contributions of our work are as follows:

- As far as our knowledge extends, *DynamicRegress* is among the first methods to represent system performance using multi-dimensional KPIs jointly with workload information, enabling robust modeling under dynamically changing environments and variable-length KPI sequences.
- *DynamicRegress* employs a dual weights-shared LSTM network branch to compare data pairs, eliminating the need to train separate models for each interface. This approach reduces training complexity while enhancing detection accuracy.
- *DynamicRegress* uses the strategy of semi-supervised learning, data augmentation strategy, and weighted loss function strategy, which expands the diversity of minority

samples and solves the challenge of imbalanced labeled samples.
- *DynamicRegress* has been deployed in a top-tier global cloud service provider *Huawei Cloud*, and evaluated on *Huawei Cloud*'s production environment data and achieved an *F1 Score* of 0.958 for accurately identifying performance regression, outperforming the best baseline method by 0.282.

II. Background

Stable performance is crucial for modern software systems, especially with frequent updates and dynamic workloads. To better understand the research objectives of this study, it is important to first establish a clear understanding of key concepts. This section introduces the foundational concepts that inform our approach and discusses the motivations.

A. Preliminaries

1) KPIs: Key Performance Indicators (KPIs) are metrics directly related to applications, used to assess application performance and user experience [12], [13]. Examples include response time (RT), transactions per second (TPS), and success rate. These indicators reflect the operational efficiency, stability, and quality of service provided to users by the application [14].

2) Performance Regression Detection: Performance regression refers to the degradation of system performance due to code changes, environmental changes, or workload shifts [15], [16]. Methods for detecting performance regression can be divided into two types: low-level hardware performance regression detection, which primarily uses HWPC data to identify significant performance bottlenecks [17]–[19]. HWPC data is generally stable under normal conditions, and performance issues are rare but catastrophic. Such methods usually rely on unsupervised learning and statistical analysis to detect deviations from normal hardware behavior. Still, HWPC data is typically collected at a lower frequency due to the high-performance overhead of too high a sampling frequency. The other is advanced application performance regression detection, which focuses on service-level degradation that directly impacts the user experience, such as a sudden increase in response time or a significant drop in throughput. This type of approach leverages application-layer monitoring tools to collect KPIs at a low cost [20]. These metrics are dynamic and time-correlated, and support high-frequency sampling, enabling real-time monitoring and alerting.

Our research focuses on advanced application performance regression detection by analyzing the distribution of KPIs over time. In our scenario, a system typically consists of numerous functional interfaces, each serving a different purpose, such as user authentication, data retrieval, or transaction processing. Each interface presents a unique normal data pattern due to differences in business logic, dependencies, and user behavior.

3) Siamese Network: The Siamese network is a specialized neural network architecture primarily used for tasks involving similarity comparison [21], [22]. It comprises two identical subnetworks that share the same weights and parameters. Each subnetwork processes an input sample and maps it to a feature space, generating a feature vector. The similarity between two input samples is then evaluated by calculating the distance (*e.g.,* Euclidean distance or cosine similarity) between their feature vectors. A smaller distance indicates higher similarity, whereas a larger distance implies dissimilarity. Siamese networks excel in tasks with limited labeled data as they do not require large-scale datasets for training [23]. They are simple in structure, easy to implement, and effective in learning the similarity between samples. These networks perform exceptionally well on paired data and are suitable for tasks requiring similarity measurements, such as facial recognition [24], signature verification [25], and object tracking [22].

In facial recognition, the Siamese network compare the features of two face images, and even if the target face image is not in the database, it can also identify individuals by comparing the similarity between the input image and the known image [21]. In our scenario, this idea can be likened to that when encountering new data that is not in the training set, the Siamese network can accurately determine whether it belongs to the correct pattern of an interface by comparing the characteristics of the new data with the known data.

4) Semi-supervised Learning.: Semi-supervised learning is a machine learning approach that combines a small amount of labeled data with a large amount of unlabeled data to improve model performance [26], [27]. Its core strength lies in reducing reliance on expensive labeled datasets while leveraging the predictive power of unlabeled data. Compared to traditional supervised learning, which typically yields better performance by leveraging large amounts of high-quality labeled data, semi-supervised learning mitigates the reliance on expensive labeled datasets. In practical production environments, obtaining high-quality labeled data is often costly and labor-intensive. While unsupervised learning does not rely on labeled data, its performance in this task is significantly inferior to supervised learning. To bridge this gap, we employ a semi-supervised learning approach that reduces the dependence on extensive labeled datasets while achieving satisfactory performance.

B. Motivation

Modern software systems are becoming increasingly complex due to new features, architectural optimizations, and service expansions. These changes introduce significant challenges in accurately measuring performance and detecting regressions. While traditional approaches focus on individual technical bottlenecks, they often fail to address the broader issues caused by dynamic KPI sequences and workloads, diverse interfaces, and imbalanced labeled data. Our work is driven by the need to tackle these limitations and enhance the accuracy and robustness of performance regression detection.

1) Comparing variable-length KPIs under Dynamically Changing Loads: System performance metrics are jointly influenced by workload intensity, resource conditions, and system configurations [28]. In real-world testing scenarios, dynamic workloads and varying test durations result in KPI sequences of different lengths and fluctuation patterns, even

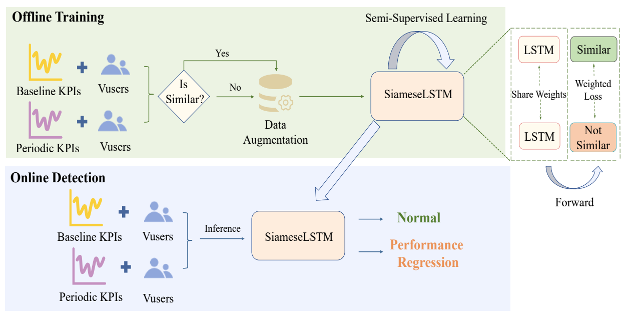

Fig. 2: The framework of *DynamicRegress*.

under normal system behavior. These variations make it difficult to distinguish genuine regressions from expected changes. Therefore, an effective performance regression detection method must accurately characterize system behavior by modeling multi-dimensional KPIs together with workload context, while supporting comparison across variable-length sequences.

2) Diverse Interfaces and Scarce Labeled Samples per Interface: Modern software systems feature diverse functional interfaces, each exhibiting unique data patterns due to differences in business logic, dependencies, and user behavior. This diversity, combined with the scarcity of labeled samples for individual interfaces, creates a substantial challenge. Training separate models for each interface is not feasible due to high computational costs and maintenance demands. Instead, a unified approach is needed to handle diverse data patterns efficiently and learn effectively from limited labeled samples. This requires methods capable of capturing the commonalities across interfaces.

3) Imbalanced Labeled Samples: In user-centric metrics, normal samples are plentiful, but regression samples are rare, resulting in a significant imbalance. This imbalance hampers traditional methods, which struggle to accurately identify minority class patterns. The scarcity of regression-labeled samples makes it difficult to detect subtle anomalies without introducing bias or overfitting. To overcome this limitation, it is essential to develop methods that enhance sensitivity to minority samples, such as regression cases, while maintaining robustness to imbalanced data distributions. This ensures reliable detection of performance regressions in real-world scenarios.

III. Approach

In real-world stress-testing scenarios, it is often impractical to ensure identical workload patterns across multiple testing runs. Labeling large volumes of KPI data for performance regression is similarly infeasible due to the extensive manual effort required. Given the large number of functionalities and interfaces in most software systems, labeled data for each interface is scarce, making it unfeasible to train separate models for each interface type. To address this, we propose *DynamicRegress*, which learns the differences in data patterns between periodic and baseline KPIs. Rather than treating each interface type independently, we aggregate labeled data across all interfaces, assuming that, within the same category, baseline KPIs and normal periodic KPIs share similar patterns, while patterns diverge for abnormal periodic KPIs. *DynamicRegress* is designed to capture these pattern differences, learning the regularities of their similarity or dissimilarity.

A. Overview

As shown in Fig. 2, *DynamicRegress* consists of four key modules.
(1) **Data Preprocessing (§III-B).** To address Challenge 1, *DynamicRegress* constructs a unified input by combining multi-dimensional KPIs with contextual workload information. Rather than enforcing fixed-length input, the design preserves variable-length sequences, which are natively handled by LSTM encoders.
(2) **Training of *DynamicRegress* (§III-C).** To address Challenge 2, *DynamicRegress* employs a twin weights-shared LSTM network, which processes paired sequences with shared parameters and enables *DynamicRegress* to learn sample similarities effectively.
(3) **Semi-supervised Learning (§III-D).** To address Challenge 3, we propose three techniques that effectively mitigate sample imbalance. First, the semi-supervised learning strategy combines pseudo-labeling and confidence filtering, enabling *DynamicRegress* to generate high-quality pseudo-labels for unlabeled data using a small number of labeled samples. Second, data augmentation methods enhance the diversity of minority-class samples (detailed in Section §III-B). Finally, a weighted loss function prioritizes regression samples, ensuring the model focuses more on detecting performance regressions (detailed in Section §III-C). These combined techniques effectively tackle Challenge 3.
(4) **Online Detection (§III-E).** Here we describe the *DynamicRegress*'s output and the performance regression of expert rule definitions.

B. Data Preprocessing

The dataset originates from *Huawei Cloud*'s cluster environment, established using CodeArts PerfTest. Analysis of the dataset revealed a notable class imbalance, with regression KPI pairs accounting for only 7.91% of the total samples. This imbalance poses a challenge to *DynamicRegress*, potentially limiting its ability to effectively learn the features of the minority class. To address this issue, data augmentation was applied during preprocessing. By transforming the original data to generate additional samples, this technique enhances the diversity of the regression KPIs data, thereby improving *DynamicRegress*'s capacity to learn regression data patterns and increasing its robustness [29], [30]. For time series data, Wen *et al.* summarized common augmentation methods such as flipping, scaling, window warping, slicing, and adding noise [31]. In this study, we adopt two effective strategies: adding noise and time shifting.

Noise addition introduces random perturbations to regression samples, simulating slight variations in KPIs, which involves

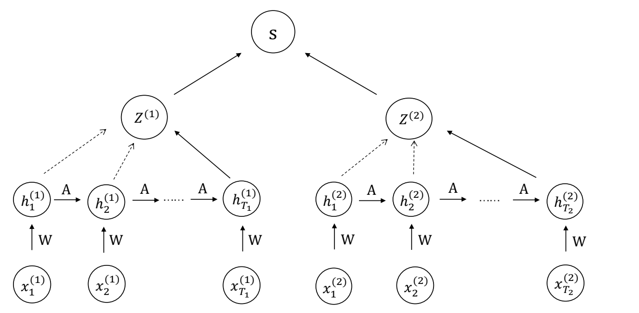

Fig. 3: Training Process of SiameseLSTM with Variable-Length Sequences.

modifying performance metrics to reflect insignificant deviations and creating diverse examples of regression scenarios. Time shifting applies slight temporal offsets to regression samples, generating variations of the sequences by shifting timestamps while retaining core patterns.

Inspired by the intuition behind VPerfGuard's workload-aware design, we propose a more robust and model-friendly preprocessing strategy: rather than adjusting KPI values, we incorporate the workload (i.e., the number of virtual users) as a first-class feature into the input time series. Specifically, each input sample is represented as a multivariate time series, where each time step contains both the workload and one KPI value of interest.

In software performance regression detection, traditional KPIs are typically analyzed independently, ignoring the impact of dynamic workloads, as shown in Fig.1 (b). Failing to account for the load as a critical factor can lead to model learning features that do not align with real-world scenarios, ultimately reducing the accuracy of performance evaluations. To address this challenge, we combine multi-dimension KPIs with the contextual information of workloads and input it into *DynamicRegress* for feature learning.

In our task, each input sample is a multivariate time series that combines a performance KPI with the contextual workload (*i.e.*, number of virtual users). Specifically, the input to the SiameseLSTM is defined as:

$$X^{(1)} = \{x_1^{(1)}, x_2^{(1)}, \ldots, x_{T_1}^{(1)}\}, \quad X^{(2)} = \{x_1^{(2)}, x_2^{(2)}, \ldots, x_{T_2}^{(2)}\}$$

where each $x_i^{(j)} \in \mathbb{R}^2$ ($j \in \{1, 2\}$) is a two-dimensional vector at time step i, consisting of the workload (Vusers) and one KPI value (TPS, RT, or success rate). For a complete characterization of the system state, the model processes these three types of KPIs independently. T_1 and T_2 denote the sequence lengths of the two samples, which may differ.

By combining these two augmentation strategies, we effectively expand the diversity of performance regression samples, helping *DynamicRegress* learn richer representations of the regression KPIs.

C. Training of DynamicRegress

1) LSTM: The core module of *DynamicRegress*, the Long Short-Term Memory (LSTM) network, is a type of recurrent neural network (RNN) specifically designed to handle sequential data. It excels at capturing both long-term dependencies and short-term dynamic patterns, making it suitable for analyzing time-series data such as performance KPIs and user workloads. LSTM effectively addresses the vanishing and exploding gradient problems often encountered in traditional RNNs by introducing memory cells and gating mechanisms, which control the flow of information through time.

Let h_t denote the hidden state of the input at time step t, calculated as:

$$h_t = \phi(Wx_t + Ah_{t-1} + b) \tag{1}$$

Where $x_t \in \mathbb{R}^2$ is the input vector at time t, composed of vusers and KPI metrics. W and A are learnable weight matrices that model the current input and the previous hidden state, respectively, while b is a bias term. The activation function $\phi(\cdot)$ introduces non-linearity to capture complex patterns in the sequence. Note that this simplified formulation abstracts away the internal gating mechanisms of LSTM (such as input, forget, and output gates) for clarity and brevity. And in our implementation, the LSTM network is configured with 2 layers and a hidden size of 128.

To obtain a total feature representation from a variable-length input sequence, we use the hidden state from the last time step:

$$Z = h_T \tag{2}$$

As shown in Fig. 3, this method condenses the entire temporal dynamics of the input into a single vector. Because LSTM processes sequences in a stepwise manner without requiring fixed length, this approach naturally supports variable-length inputs. Regardless of their lengths, sequences can be encoded into comparable embeddings.

These embeddings are then compared using absolute difference due to the advantage of computational efficiency and robustness to outliers compared to Euclidean distance (sensitive to squared errors) or cosine similarity (requiring normalization):

$$\begin{gathered} \Delta Z = |Z^{(1)} - Z^{(2)}| \\ \hat{y} = \mathrm{softmax}(W_{\text{fc}} \cdot \Delta Z + b_{\text{fc}}) \end{gathered} \tag{3}$$

The final output $\hat{y} \in \mathbb{R}^2$ gives the probability of whether the two sequences are similar or not, enabling the model to detect potential performance regressions based on learned temporal patterns and workload-aware representations.

2) Weights-Shared: To address the challenge of limited labeled samples, *DynamicRegress* adopts a twin LSTM network structure with shared weights. By sharing parameters between the two networks, the model ensures that paired sequences are processed identically during feature extraction. This strategy not only reduces the number of parameters, which is crucial for training with limited labeled data, but also improves the model's ability to learn consistent and effective feature representations.

The weight-sharing mechanism enables the model to generalize well from a small set of labeled samples, as the shared parameters maximize the utilization of available data. Furthermore, it ensures fairness and accuracy in similarity judgments by applying the same transformation to both input sequences, making the learned features robust and reliable.

Reconstruction-based deep learning methods [9] are unsuitable in our context because they require a large number of normal samples for each interface to effectively learn and reconstruct normal data patterns. In real-world production environments, such abundant normal data may not be available, especially for less frequently used interfaces. Consequently, these methods struggle to handle unseen normal patterns and may lead to significant reconstruction errors. In contrast, *DynamicRegress* focuses on learning the similarities and differences, allowing it to operate effectively with limited labeled data and avoiding the limitations of reconstruction-based approaches.

Compared to traditional single LSTM network [32] that concatenate data from two instances into a single input, the weights-shared LSTM networks in *DynamicRegress* process the two inputs separately, effectively preventing the model from overutilizing similarity or redundant information, which allows the model to focus more on the differences between the data.

3) Train: In order to improve the learning ability of *DynamicRegress* on the features of minority samples, we utilized a weighted cross-entropy loss function [33] during model training. The traditional cross-entropy loss function is widely used in classification tasks and is typically employed to evaluate how well the predicted distribution matches the ground truth. However, on imbalanced datasets, the traditional cross-entropy loss may cause the model to favor the majority class, as errors in predicting minority class samples have a relatively smaller impact on the overall loss, making it difficult to effectively guide the model to focus on the minority class. To address this issue, we adopted a weighted cross-entropy loss function. The formula for the weighted cross-entropy loss is as follows:

$$\text{Loss} = -\frac{1}{N}\sum_{i=1}^{N}\left[w_1 \cdot y_i \cdot \log(\hat{y}_i) + w_0 \cdot (1-y_i) \cdot \log(1-\hat{y}_i)\right] \tag{4}$$

Where w_i represents the weight of the class i, y_i denotes the true label of the sample i, and $\hat{y}_i$ is the predicted probability of sample i belonging to the positive class.

Through extensive experiments, we found that setting the loss weight ratio between the majority and minority classes to 1:2 effectively enhanced the model's performance without introducing significant bias. Specifically, this weight ratio avoids two issues: (1) overly small weights for the minority class, which would cause the model to remain biased toward the majority class, and (2) excessively large weights for the minority class, which would lead to over-prediction of the minority class. This balance achieves an optimal trade-off.

Subsequently, we used the Adam optimization algorithm to iteratively adjust the weight parameters in the LSTM layers.

D. Semi-supervised Learning

To overcome the challenge of sample imbalance, we employed the strategy of semi-supervised learning, combined with data augmentation and weighted loss function strategy. The data augmentation strategy, introduced in the Section §III-B, focuses on increasing the number and diversity of minority samples. The weighted loss function strategy, introduced in the Section §III-C3, assigns higher importance to minority sample. Besides, semi-supervised learning strategy leverages both labeled and unlabeled data to expand the training set. As shown in Fig. 4, we integrated pseudo-labeling and confidence filtering techniques [34] into the performance regression detection task.

The pseudo-labeling method generates labels for unlabeled data using a model trained on high-quality labeled data. These pseudo-labels are treated as new labels and combined with the original labeled data during training. To avoid contamination by incorrect labels, we use a confidence filtering mechanism, where only samples with predicted probabilities above a threshold are added.

As training progresses, the training set expands iteratively until the model converges and the loss stabilizes. This process enables the model to achieve strong detection performance with limited labeled data, addressing the imbalanced data challenge.

E. Online Detection

As illustrated in Fig. 2, during the online detection phase, the model trained in the offline phase is utilized to compare the incoming periodic KPI sample with the baseline KPI sample. The comparison involves evaluating KPIs such as response time, throughput, and success rate. Experts believe that if the distribution of any one of these three KPIs between the two samples differs, the system should be classified as having a performance regression. Otherwise, the system is considered normal.

IV. EVALUATION

We aim to answer the following research questions (RQs):
RQ1: How effective is *DynamicRegress* in performance regression detection compared to the baseline method?
RQ2: Does each component of *DynamicRegress* contribute significantly to *DynamicRegress*'s performance?
RQ3: What is the impact of different hyperparameters?

A. Experimental Setup

TABLE I: Dataset information

Dataset	**Labeled Train Dataset**	**Unlabeled Train Dataset**	**Test Dataset**
Normal Pairs	3171	-	7573
Regression Pairs	351	-	572
Total Pairs	3522	6840	8145

Dataset. The experiments were conducted on *Huawei Cloud*'s CodeArts PerfTest platform, where we set up a cluster environment encompassing 78 functional interfaces and all KPIs are sampled at 1-second granularity to capture transient anomalies. CodeArts PerfTest is *Huawei Cloud*'s internal

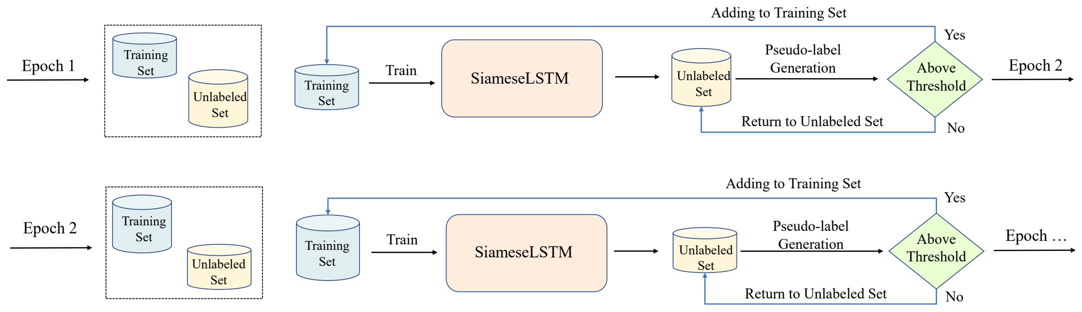

Fig. 4: Training process of semi-supervised learning.

performance testing platform, serving millions of microservices by enabling routine performance diagnostics, fault localization, and troubleshooting for developers. As shown in Table I covers diverse system performance data. It consists of three parts: the labeled training dataset, the unlabeled training dataset, and the test dataset. The labeled training dataset includes 3522 pairs of KPI samples, with 3171 normal pairs and 351 regression pairs. The unlabeled training dataset contains 6840 pairs of KPI samples, which are used to expand the training dataset through pseudo-labeling techniques, ensuring data diversity and broader coverage. Finally, the test dataset comprises 8145 pairs of KPI samples, including 7573 normal pairs and 572 regression pairs, and is used to evaluate the model's ability to detect performance regressions under varied conditions.

Settings. *DynamicRegress* is implemented using Python 3.9. We've made the source code [35] public. In model training, we adopt a fixed training schedule of 30 epochs, which proves sufficient for convergence, as evidenced by the training loss stabilizing around 0.3 in later training stages. The weighted loss function ratio is set to 1:2, and the threshold for filtering the generated pseudo-labels is set to 0.7, which we will discuss in Section IV-D.

Baselines. We use the following methods as baselines: (1) KS Test: Following [7], the Kolmogorov-Smirnov test quantifies the discrepancy between empirical CDFs of two samples. The test statistic is the maximum vertical distance between the CDFs, with the p-value derived from this statistic and sample sizes. The standard significance level α=0.05 determines whether the samples come from distinct distributions. (2) MWU Test: As proposed in [8], the MWU test evaluates the difference between two sample distributions by comparing the rank sums of the samples. Similar to the KS test, we use a significance level of α=0.05 to assess whether the distributions differ significantly. (3) TSFEL Feature Extraction and Classification: Based on [36], we use TSFEL to extract features. A traditional classifier is then trained to distinguish between positive and negative samples using these features. To enhance efficiency, we select the top 10 features ranked by importance scores for anomaly detection. (4) AutoPerf: AutoPerf [9] employs K-means clustering on training-phase data to group similar functions based on normal patterns. An autoencoder is trained per cluster to model normal behavior. During online detection, new/altered code is assigned to a cluster, and its corresponding AE computes the reconstruction error. Instances exceeding a predefined error threshold are flagged as potential performance regressions. Parameter tuning determined that 3 clusters yield the optimal *F1 Score*.

Evaluation metrics. To evaluate *DynamicRegress* and baseline methods, we used $Precision$, $Recall$, and $F1\ Score$ to assess their results. $Precision = TP/(TP + FP)$ represents the proportion of predicted dissimilar pairs that are dissimilar. $Recall = TP/(TP+FN)$ measures the proportion of actual dissimilar pairs correctly identified by the model. $F1\ Score = 2*(Precision * Recall)/(Precision + Recall)$ is the harmonic mean of $Precision$ and $Recall$, providing a balance between the two metrics. In addition, we also used the $Run\ Time$ to calculate the time required to detect a pair of KPIs.

B. Overall Performance (RQ1)

TABLE II: Effectiveness of performance regression detection

Method	Precision	Recall	F1 Score	Run Time
DynamicRegress	**0.991**	**0.927**	**0.958**	**0.006**
KS test [7]	0.160	0.998	0.276	0.038
MWU test [8]	0.159	0.998	0.274	0.038
TSFEL [36]	0.630	0.729	0.676	0.214
Autoperf [9]	0.181	0.603	0.279	0.003
C1	0.424	0.932	0.583	0.006
C2	0	0	0	0.007
C3	0.168	0.918	0.284	0.001
C4	1.000	0.792	0.884	0.001
C5	0.762	0.965	0.852	0.001

To evaluate *DynamicRegress*'s performance regression detection capability, we compared it against multiple baselines: statistical methods (KS test [7], MWU test [8]), feature extraction-based methods [36], and AutoPerf [9]. And results are summarized in Table II.

The baseline methods perform poorly on our dataset due to their inherent limitations. Both the KS test [7] and the MWU test [8] achieve near-perfect recall (0.998) but suffer from very low precision, indicating that they falsely classify many normal samples as performance regressions. This issue primarily arises from the sensitivity of the p-value threshold, as discussed earlier, and the fact that these methods fail to account for the impact of workloads on the performance metrics, as described in Challenge 1. Feature extraction-based methods [36] perform better than other methods in terms of detection accuracy. Still, its detection time significantly increases due to the additional feature extraction step required before detection, which makes it relatively time-consuming. AutoPerf [9] also fails to perform well, primarily because we lack sufficient normal data for training the model per interface, as described in Challenge 2.

In contrast, *DynamicRegress* demonstrates the best overall performance among all evaluated methods. It achieves an $F1Score$ of 0.958 (outperforming the best baseline method by 0.282) and processes each detection task in just 0.006 seconds. Although its detection time is 0.003 seconds slower than the fastest baseline, considering both $F1Score$ and detection time, we still regard *DynamicRegress* as the best. These results highlight *DynamicRegress*'s ability to address the challenges posed by dynamic service-level metrics and imbalanced samples.

C. Contribution of Key Components (RQ2)

An ablation study evaluating the contribution of individual components shows that *DynamicRegress* outperforms all variants across various scenarios (Table II), validating the importance of each component.

Notably, excluding workload input (**C1**) results in a significant performance degradation, with the *F1 Score* dropping to 0.583. This highlights the critical role that workload information plays in capturing system performance changes, particularly under dynamic load conditions. Similarly, normalizing KPIs by dividing their values by the number of users (**C2**) leads to an *F1 Score* of 0. As the training progresses, the *precision* gradually worsens until it reaches zero at epoch 25. The low *precision* occurs because this simple normalization approach mistakenly classifies normal samples as performance regressions. Removing the data augmentation module (**C3**) results in a significant drop in performance, with the *F1 Score* declining to 0.284. This demonstrates that data augmentation enhances the diversity of minority-class samples in the training data, improving the model's ability to recognize regression patterns. Furthermore, replacing the weighted loss function with a standard binary cross-entropy loss (**C4**) causes the *F1 Score* to decrease to 0.884. This shows that the weighted loss function effectively mitigates class imbalance, preventing the model from overfitting to the majority class. Finally, excluding the semi-supervised learning strategy (**C5**) reduces the *F1 Score* to 0.852, confirming that the semi-supervised module addresses the challenge of insufficient labeled data by generating high-quality pseudo-labels.

D. Hyperparameters Sensitivity (RQ3)

To assess the influence of key hyperparameters on the performance of *DynamicRegress*, we conducted sensitivity analyses focusing on two critical parameters: the confidence threshold (p_threshold) used for pseudo-labeling in the semi-supervised learning strategy, and the weighted loss ratio, which balances the learning emphasis between majority and minority classes. The results of these experiments are shown in Fig. 5.

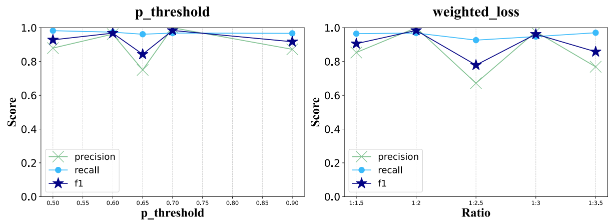

Fig. 5: Parameters sensitivity on *DynamicRegress*.

(1) P_threshold determines the minimum confidence required to include pseudo-labeled data in the training set. We evaluated *DynamicRegress* with five threshold values: 0.5, 0.6, 0.65, 0.7, and 0.9. The *F1 Score* remains relatively stable as p_threshold changes. At p_threshold=0.7, *DynamicRegress* achieves the highest *F1 Score* (0.958).
(2) The weighted loss ratio adjusts the relative importance assigned to the majority and minority classes in the loss function. We tested *DynamicRegress* with five different ratios: 1:1.5, 1:2, 1:2.5, 1:3, and 1:3.5. The *F1 Score* also remains stable across the tested values, demonstrating the robustness of *DynamicRegress* to variations in this parameter. The best performance is observed at a ratio of 1:2.

E. Deployment

DynamicRegress has been deployed in the industrial environment of *Huawei Cloud* for over 6 months, enabling real-time notifications for detected regressions, with over 90% of the regression pairs detected. Its integration into the existing workflow leverages *Huawei Cloud*'s CodeArts PerfTest platform, monitoring 78 interfaces across five critical business domains in production environments, including resource management, version control, data retrieval, pipeline operations, and system administration. The implementation architecture aligns with the industrial workflow shown in Fig. 6, with *DynamicRegress* embedded in the "Algorithm Service" module as the core algorithmic component. And key deployment details and component interactions are as follows:

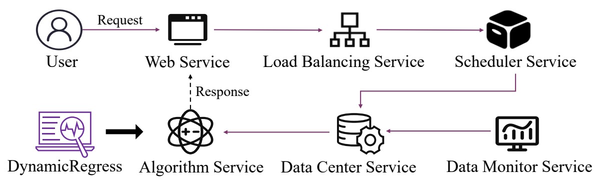

Fig. 6: The workflow of *DynamicRegress* in *Huawei Cloud*' industrial environment.

The model achieves an average detection latency of 0.006 seconds per KPI pair in our experiments. For the three core KPIs (response time, throughput, and success rate), the measured total latency is approximately 0.018 seconds, which satisfies the industrial requirement of detecting regressions within 1 second. Regarding future scalability, the architecture is designed to support additional KPIs and resource-type indicators (*e.g.*, resource utilization metrics like CPU/memory usage). The latency scales roughly linearly with the number

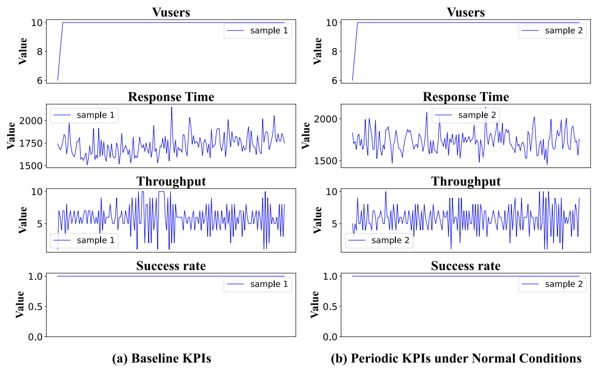
Fig. 7: Distribution of normal KPIs in system status.

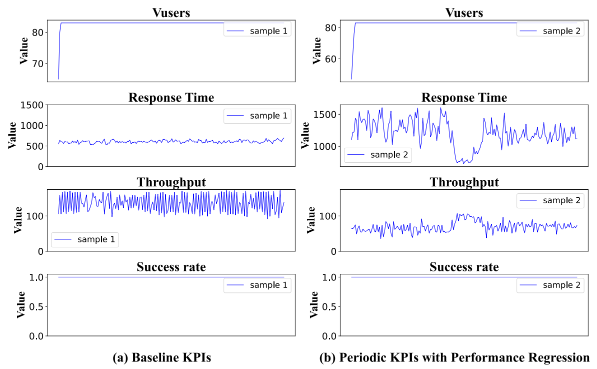
Fig. 8: Distribution of KPIs in the performance regression status.

of KPIs. Based on the current implementation, we estimate the total latency would be in the range of 0.006 × N ± 20% seconds for N KPIs. However, this may vary depending on hardware configuration and workload characteristics.

V. Discussion

A. Case Study

We evaluated *DynamicRegress* using 2715 pairs of stress test reports (including 8145 metrics) from the cluster environment at company. *DynamicRegress* achieved a Precision of 0.991, Recall of 0.927, and F1 Score of 0.958, with a detection time of just 0.006 seconds per pair. Among the 8145 metrics, 7598 showed no regression, while 277 regressed due to response time changes, 254 due to throughput changes, and 16 due to success rate fluctuations.

(1) **Normal Case.** Fig. 7 compares periodically executed KPIs (b) with baseline KPIs (a) to detect regressions. Response time shows matching fluctuation ranges and central values, indicating no regression. Throughput maintains similar fluctuation ranges without decline, demonstrating stable capacity. Success rate remains consistently near 1, confirming all requests succeeded. These results collectively confirm no performance regression occurred.

(2) **Regression Case.** As shown in Fig. 8, Fig. 8 (a) shows the baseline KPIs and Fig. 8 (b) shows the periodically executed KPIs. For response time, periodic execution exhibits sharp drops followed by recovery, with overall values exceeding the stable baseline, indicating significant changes in distribution. Throughput exhibits transient spikes then sustained decline, with values consistently below baseline, which further shows the decline of the system's processing capacity. In terms of success rate, both show a smooth horizontal line, and the success rate is close to 1, indicating that the system is still functional and no request errors have occurred.

It should be noted that the success rate only reflects whether the request is completed, but cannot reflect the processing efficiency and system performance. If response time and throughput deteriorate beyond what is acceptable to the business, even a success rate of 1 indicates a performance regression. This phenomenon also validates our expert rule: a change in the data distribution of one metric can indicate a performance regression in the system.

B. Limitations and Possible Solutions

Based on the experiments performed, we identified the following two limitations of *DynamicRegress* and proposed possible solutions:

(1) **Limited Augmentation Diversity.** Currently, we use time-shifting and noise addition to augment regression samples. While effective, these methods may not fully capture the variety and complexity of real-world anomalies. In future work, we plan to incorporate more advanced techniques such as DTW-based augmentation [37] and Mixup [38], which could better simulate timing misalignments and enrich minority-class sample diversity.

(2) **Limited Loss Function Flexibility.** We adopt a weighted binary cross-entropy loss to mitigate the impact of class imbalance, effectively guiding the model to pay more attention to regression cases. An alternative commonly used in Siamese networks is *contrastive loss*, which encourages similar pairs to stay close and dissimilar pairs to stay apart in the embedding space. Its formulation is:

$$L = \frac{1}{2N}\sum_{i=1}^{N}\left[y_i \cdot D(x_i, x_i')^2 + (1-y_i)\cdot \max(0, m - D(x_i, x_i'))^2\right] \tag{5}$$

Here, y_i indicates whether a pair is similar, D is the distance function, and m is a margin for dissimilar pairs.

While contrastive loss is theoretically well-suited for pairwise similarity modeling, we did not adopt it due to its higher computational cost during training. In our production context, fast training and deployment are critical, and binary cross-entropy has proven both efficient and sufficiently effective.

(3) **Lack of Joint Reasoning Across Multiple KPIs.** In the current implementation, each KPI is independently compared and classified, with the final decision derived via simple majority voting. However, performance regressions in real-world systems may arise only when multiple indicators degrade together. To address this, *DynamicRegress* allows SREs to define customized decision rules that combine per-KPI results in various ways, such as:

- **Rule-based voting strategies**, such as "two-out-of-three KPIs must indicate regression";

- **Weighted fusion**, where each KPI's importance can be adjusted according to business criticality;
- **Hybrid logic**, where specific combinations of metric deviations trigger regression alerts (*e.g.*, high response time and low throughput).

These strategies can be tailored per interface or service, allowing SREs to balance sensitivity and precision based on operational priorities.

VI. Related Work

We discuss related work in following directions: predicting the impact of source code changes on system performance, contrastive learning and identifying anomalous traffic samples in network intrusion detection.

A. Detecting Performance Regression via Code Analysis and Runtime Modeling

Accurately diagnosing the root causes is essential for modern software systems undergoing continuous evolution. Several prior works have attempted to assess the impact of code changes or system-level factors on performance regression.

PerfImpact [39], PRICE [40], and Perphecy [41] analyze code changes for performance regression risks. PerfImpact [39] uses a genetic algorithm with dynamic change impact analysis to rank impactful changes. PRICE [40] combines static (code complexity) and dynamic metrics (execution time) with NSGA-II to generate detection rules. Perphecy [41] employs metric templates and greedy algorithms to threshold static/dynamic metrics for regression prediction. In contrast, vPerfGuard [42] detects performance regressions by comparing predicted versus observed service metrics under production workloads. It builds analytical performance models using historical data, incorporating workload patterns, resource demands, and service dependencies to predict component-level latency/throughput. During operation, it flags regressions when actual metrics deviate significantly from predictions beyond normal workload variations.

However, *DynamicRegress* focuses on post-deployment regression detection via behavioral monitoring. In practical industrial scenarios, resource-level metrics such as CPU and memory are typically collected at coarse granularity (*e.g.*, five minutes per point), which makes it difficult to associate them with real-time interface performance. Moreover, service-level dependency tracing is often incomplete or unavailable across all tested interfaces. These constraints make *DynamicRegress* particularly suitable for black-box monitoring scenarios where internal system information is limited or inaccessible.

B. Contrastive Learning

Contrastive learning, inspired by Siamese networks, is an unsupervised or semi-supervised learning method that constructs sample pairs to learn effective representations by pulling positive pairs closer and pushing negative pairs apart in feature space.

Contrastive learning has been widely applied across computer vision (CV), natural language processing (NLP), and time series analysis (TS). In CV, SimCLR [43] leverages image pairs to learn representations from unlabeled data, while MoCo [44] employs dynamic dictionaries with query (q)/ positive (k_+)/ negative representations (k_-) for contrastive optimization. NLP applications include Minsearch [45] for string edit similarity, Word2Vec [46] for semantic vector relationships, and Sentence-BERT [47] for sentence-level tasks. For TS, TS-TCC [48] enhances feature learning, SimMTM [49] uses cosine-based reconstruction, and TimeCLR [37] applies transformations like cropping/shifting to improve sample similarity. These demonstrate the versatility of contrastive learning in cross-domain representation learning.

C. End-to-End Similarity Analysis Methods in Network Traffic Anomaly Detection

Network traffic anomaly detection similarly identifies anomalous behavior or threats by comparing new samples to normal patterns [50]. For example, CANN [51] utilizes Euclidean distance to measure similarity, clustering traffic data with K-means and classifying anomalies using a K-NN classifier. However, its fixed number of clusters and lack of handling for imbalanced data reduce its flexibility in detecting minority-class anomalies, such as attack traffic. Other approaches, including [52]–[54], employ Gaussian distance metrics to define decision boundaries. These methods often train models using normal traffic data to construct thresholds that separate normal and anomalous patterns. Techniques such as feature selection, dimensionality reduction, and incremental clustering further enhance their performance in complex traffic scenarios. GARUDA [54], for instance, uses Gaussian similarity to group traffic data into clusters and applies classifiers for anomaly detection, achieving better scalability and adaptability.

VII. Conclusion

To address the challenge of performance regression detection in dynamic software environments, we introduce *DynamicRegress*, a novel methodology that leverages multi-dimensional KPIs, semi-supervised learning, and a dual weight-shared network. By integrating workload context and enhancing training data diversity through semi-supervised learning, *DynamicRegress* ensures robust performance under dynamic workloads. The data augmentation and weighted loss function strategies further enhance *DynamicRegress*'s ability to learn from minority regression cases, improving its generalization and sensitivity. Our experimental evaluations, conducted on CodeArts PerfTest platform from *Huawei Cloud*, demonstrate the efficacy of *DynamicRegress*. *DynamicRegress* achieves an *F1 Score* of 0.958 and detects a pair of indicators in 0.006 seconds, which validates the robustness of *DynamicRegress* in detecting performance regressions. The source code for *DynamicRegress* has been made publicly available to support further research and development in this area [35].

VIII. Acknowledgement

This work is supported by the National Natural Science Foundation of China (62272249, 62302244), and the Fundamental Research Funds for the Central Universities (XXX-63253249).

References

[1] Lizhi Liao. Addressing performance regressions in devops: Can we escape from system performance testing? In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 203–207. IEEE, 2023.

[2] Xiao Bai, Ioannis Arapakis, B Barla Cambazoglu, and Ana Freire. Understanding and leveraging the impact of response latency on user behaviour in web search. *ACM Transactions on Information Systems (TOIS)*, 36(2):1–42, 2017.

[3] Christina Xilogianni, Filippos-Rafail Doukas, Ioannis C Drivas, and Dimitrios Kouis. Speed matters: What to prioritize in optimization for faster websites. *Analytics*, 1(2):175–192, 2022.

[4] Xiaohui Nie, Youjian Zhao, Kaixin Sui, Dan Pei, Yu Chen, and Xianping Qu. Mining causality graph for automatic web-based service diagnosis. In *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8. IEEE, 2016.

[5] Tarek M Ahmed, Cor-Paul Bezemer, Tse-Hsun Chen, Ahmed E Hassan, and Weiyi Shang. Studying the effectiveness of application performance management (apm) tools for detecting performance regressions for web applications: an experience report. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 1–12, 2016.

[6] Yifan Xiong, Yuting Jiang, Ziyue Yang, Lei Qu, Guoshuai Zhao, Shuguang Liu, Dong Zhong, Boris Pinzur, Jie Zhang, Yang Wang, et al. {SuperBench}: Improving cloud {AI} infrastructure reliability with proactive validation. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 835–850, 2024.

[7] Rafi Abbel Mohammad and Achmad Imam Kistijantoro. Development of performance regression analysis tool using distributed tracing on microservice-based application. In *2022 9th International Conference on Advanced Informatics: Concepts, Theory and Applications (ICAICTA)*, pages 1–6. IEEE, 2022.

[8] Lizhi Liao, Jinfu Chen, Heng Li, Yi Zeng, Weiyi Shang, Jianmei Guo, Catalin Sporea, Andrei Toma, and Sarah Sajedi. Using black-box performance models to detect performance regressions under varying workloads: an empirical study. *Empirical Software Engineering*, 25:4130–4160, 2020.

[9] Mejbah Alam, Justin Gottschlich, Nesime Tatbul, Javier S Turek, Tim Mattson, and Abdullah Muzahid. A zero-positive learning approach for diagnosing software performance regressions. *Advances in Neural Information Processing Systems*, 32, 2019.

[10] Linshan Hou, Xiaofeng Jin, and Zhenshuang Zhao. Time series similarity measure via siamese convolutional neural network. In *2019 12Th international congress on image and signal processing, biomedical engineering and informatics (CISP-BMEI)*, pages 1–6. IEEE, 2019.

[11] Sylvestre-Alvise Rebuffi, Sven Gowal, Dan Andrei Calian, Florian Stimberg, Olivia Wiles, and Timothy A Mann. Data augmentation can improve robustness. *Advances in Neural Information Processing Systems*, 34:29935–29948, 2021.

[12] Issa Atoum. Measurement of key performance indicators of user experience based on software requirements. *Science of Computer Programming*, 226:102929, 2023.

[13] Adam Trendowicz, Eduard C Groen, Jens Henningsen, Julien Siebert, Nedo Bartels, Sven Storck, and Thomas Kuhn. User experience key performance indicators for industrial iot systems: A multivocal literature review. *Digital Business*, 3(1):100057, 2023.

[14] Caixiang Fan, Sara Ghaemi, Hamzeh Khazaei, and Petr Musilek. Performance evaluation of blockchain systems: A systematic survey. *IEEE Access*, 8:126927–126950, 2020.

[15] Jinfu Chen and Weiyi Shang. An exploratory study of performance regression introducing code changes. In *2017 ieee international conference on software maintenance and evolution (icsme)*, pages 341–352. IEEE, 2017.

[16] Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. Performance regression testing target prioritization via performance risk analysis. In *Proceedings of the 36th International Conference on Software Engineering*, pages 60–71, 2014.

[17] Malcolm Bourdon, Eric Alata, Mohamed Kaâniche, Vincent Migliore, Vincent Nicomette, and Youssef Laarouchi. Anomaly detection using hardware performance counters on a large scale deployment. In *10th European Congress Embedded Real Time Systems (ERTS 2020)*, 2020.

[18] Lai Leng Woo. Hardware performance counters (hpcs) for anomaly detection. *Hardware Supply Chain Security: Threat Modelling, Emerging Attacks and Countermeasures*, pages 147–165, 2021.

[19] Jingyi Xu, Sehoon Kim, Borivoje Nikolic, and Yakun Sophia Shao. Memory-efficient hardware performance counters with approximate-counting algorithms. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 226–228. IEEE, 2021.

[20] Christoph Heger, André van Hoorn, Mario Mann, and Dušan Okanović. Application performance management: State of the art and challenges for the future. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 429–432, 2017.

[21] Gregory Koch, Richard Zemel, Ruslan Salakhutdinov, et al. Siamese neural networks for one-shot image recognition. In *ICML deep learning workshop*, volume 2, pages 1–30. Lille, 2015.

[22] Luca Bertinetto, Jack Valmadre, Joao F Henriques, Andrea Vedaldi, and Philip HS Torr. Fully-convolutional siamese networks for object tracking. In *Computer Vision–ECCV 2016 Workshops: Amsterdam, The Netherlands, October 8-10 and 15-16, 2016, Proceedings, Part II 14*, pages 850–865. Springer, 2016.

[23] Nilakshi B Pokharkar. *Improving Anomaly Detection in the Cognitive Test Scores using Siamese Neural Network and Metric Learning as Ordinal Classification Task*. PhD thesis, 2024.

[24] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.

[25] Sounak Dey, Anjan Dutta, J Ignacio Toledo, Suman K Ghosh, Josep Lladós, and Umapada Pal. Signet: Convolutional siamese network for writer independent offline signature verification. *arXiv preprint arXiv:1707.02131*, 2017.

[26] Avital Oliver, Augustus Odena, Colin A Raffel, Ekin Dogus Cubuk, and Ian Goodfellow. Realistic evaluation of deep semi-supervised learning algorithms. *Advances in neural information processing systems*, 31, 2018.

[27] Jesper E Van Engelen and Holger H Hoos. A survey on semi-supervised learning. *Machine learning*, 109(2):373–440, 2020.

[28] Mohammad S Aslanpour, Sukhpal Singh Gill, and Adel N Toosi. Performance evaluation metrics for cloud, fog and edge computing: A review, taxonomy, benchmarks and standards for future research. *Internet of Things*, 12:100273, 2020.

[29] Terry T Um, Franz MJ Pfister, Daniel Pichler, Satoshi Endo, Muriel Lang, Sandra Hirche, Urban Fietzek, and Dana Kulić. Data augmentation of wearable sensor data for parkinson's disease monitoring using convolutional neural networks. In *Proceedings of the 19th ACM international conference on multimodal interaction*, pages 216–220, 2017.

[30] Brian Kenji Iwana and Seiichi Uchida. An empirical survey of data augmentation for time series classification with neural networks. *Plos one*, 16(7):e0254841, 2021.

[31] Qingsong Wen, Liang Sun, Fan Yang, Xiaomin Song, Jingkun Gao, Xue Wang, and Huan Xu. Time series data augmentation for deep learning: A survey. *arXiv preprint arXiv:2002.12478*, 2020.

[32] Yong Yu, Xiaosheng Si, Changhua Hu, and Jianxun Zhang. A review of recurrent neural networks: Lstm cells and network architectures. *Neural computation*, 31(7):1235–1270, 2019.

[33] Ziyun Zhou, Hong Huang, and Binhao Fang. Application of weighted cross-entropy loss function in intrusion detection. *Journal of Computer and Communications*, 9(11):1–21, 2021.

[34] Kihyuk Sohn, David Berthelot, Nicholas Carlini, Zizhao Zhang, Han Zhang, Colin A Raffel, Ekin Dogus Cubuk, Alexey Kurakin, and Chun-Liang Li. Fixmatch: Simplifying semi-supervised learning with consistency and confidence. *Advances in neural information processing systems*, 33:596–608, 2020.

[35] dynamicregress. https://github.com/limengyaoi/SiameseLSTM, 2025.

[36] Marília Barandas, Duarte Folgado, Letícia Fernandes, Sara Santos, Mariana Abreu, Patrícia Bota, Hui Liu, Tanja Schultz, and Hugo Gamboa. Tsfel: Time series feature extraction library. *SoftwareX*, 11:100456, 2020.

[37] Xinyu Yang, Zhenguo Zhang, and Rongyi Cui. Timeclr: A self-supervised contrastive learning framework for univariate time series representation. *Knowledge-Based Systems*, 245:108606, 2022.

[38] Hongyi Zhang. mixup: Beyond empirical risk minimization. *arXiv preprint arXiv:1710.09412*, 2017.

[39] Qi Luo, Denys Poshyvanyk, and Mark Grechanik. Mining performance regression inducing code changes in evolving software. In *Proceedings*

of the 13th International Conference on Mining Software Repositories, pages 25–36, 2016.

[40] Deema Alshoaibi, Kevin Hannigan, Hiten Gupta, and Mohamed Wiem Mkaouer. Price: Detection of performance regression introducing code changes using static and dynamic metrics. In *Search-Based Software Engineering: 11th International Symposium, SSBSE 2019, Tallinn, Estonia, August 31–September 1, 2019, Proceedings 11*, pages 75–88. Springer, 2019.

[41] Augusto Born De Oliveira, Sebastian Fischmeister, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. Perphecy: Performance regression test selection made simple but effective. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 103–113. IEEE, 2017.

[42] Pengcheng Xiong, Calton Pu, Xiaoyun Zhu, and Rean Griffith. vperfguard: An automated model-driven framework for application performance diagnosis in consolidated cloud environments. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, pages 271–282, 2013.

[43] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PMLR, 2020.

[44] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9729–9738, 2020.

[45] Haoyu Zhang and Qin Zhang. Minsearch: An efficient algorithm for similarity search under edit distance. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 566–576, 2020.

[46] Emi Suryati, Styawati Styawati, and Ahmad Ari Aldino. Analisis sentimen transportasi online menggunakan ekstraksi fitur model word2vec text embedding dan algoritma support vector machine (svm). *Jurnal Teknologi dan Sistem Informasi*, 4(1):96–106, 2023.

[47] N Reimers. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

[48] Emadeldeen Eldele, Mohamed Ragab, Zhenghua Chen, Min Wu, Chee Keong Kwoh, Xiaoli Li, and Cuntai Guan. Time-series representation learning via temporal and contextual contrasting. *arXiv preprint arXiv:2106.14112*, 2021.

[49] Jiaxiang Dong, Haixu Wu, Haoran Zhang, Li Zhang, Jianmin Wang, and Mingsheng Long. Simmtm: A simple pre-training framework for masked time-series modeling. *Advances in Neural Information Processing Systems*, 36, 2024.

[50] Lei Tao, Shenglin Zhang, Junhua Kuang, Xiao-Wei Guo, and Canqun Yang. Real-time anomaly detection for large-scale network devices. *IEEE Transactions on Networking*, pages 1–12, 2025.

[51] Wei-Chao Lin, Shih-Wen Ke, and Chih-Fong Tsai. Cann: An intrusion detection system based on combining cluster centers and nearest neighbors. *Knowledge-based systems*, 78:13–21, 2015.

[52] Bouchra Lamrini, Augustin Gjini, Simon Daudin, Pascal Pratmarty, François Armando, and Louise Travé-Massuyès. Anomaly detection using similarity-based one-class svm for network traffic characterization. In *DX*, 2018.

[53] Arun Nagaraja, Uma Boregowda, Khalaf Khatatneh, Radhakrishna Vangipuram, Rajasekhar Nuvvusetty, and V Sravan Kiran. Similarity based feature transformation for network anomaly detection. *IEEE Access*, 8:39184–39196, 2020.

[54] Shadi A Aljawarneh and Radhakrishna Vangipuram. Garuda: Gaussian dissimilarity measure for feature representation and anomaly detection in internet of things. *The Journal of Supercomputing*, 76(6):4376–4413, 2020.