

# ARGOS: Agentic Time-Series Anomaly Detection with Autonomous Rule Generation via Large Language Models

Yile Gu<sup>1,2\*</sup>, Yifan Xiong<sup>2</sup>, Jonathan Mace<sup>2</sup>, Yuting Jiang<sup>2</sup>, Yigong Hu<sup>1,3</sup>, Baris Kasikci<sup>1</sup>, and Peng Cheng<sup>2</sup>

<sup>1</sup>University of Washington

<sup>2</sup>Microsoft Research

<sup>3</sup>Boston University

## Abstract

Observability in cloud infrastructure is critical for service providers, driving the widespread adoption of anomaly detection systems for monitoring metrics. However, existing systems often struggle to simultaneously achieve explainability, reproducibility, and autonomy, which are three indispensable properties for production use. We introduce ARGOS, an agentic system for detecting time-series anomalies in cloud infrastructure by leveraging large language models (LLMs). ARGOS proposes to use explainable and reproducible anomaly rules as intermediate representation and employs LLMs to autonomously generate such rules. The system will efficiently train error-free and accuracy-guaranteed anomaly rules through multiple collaborative agents and deploy the trained rules for low-cost online anomaly detection. Through evaluation results, we demonstrate that ARGOS outperforms state-of-the-art methods, increasing  $F_1$  scores by up to 9.5% and 28.3% on public anomaly detection datasets and an internal dataset collected from Microsoft, respectively.

## 1 Introduction

Ensuring the reliability and availability of cloud services is a key challenge for service providers [8, 19, 31, 32, 71], as downtime or interruptions can severely impact both customer experience and business operations. In December 2021, an unexpected surge in connectivity triggered by autoscaling led to a major outage in Amazon Web Services (AWS), causing disruptions to downstream services and impacting millions of users worldwide for more than 10 hours [54].

To minimize the negative consequences of service interruptions, early-detection of anomalies in metrics monitoring is crucial, as such metrics provide real-time insights into the health and performance of cloud services. Large companies often develop and deploy anomaly detection systems in production environments [49, 60, 69]. These systems are

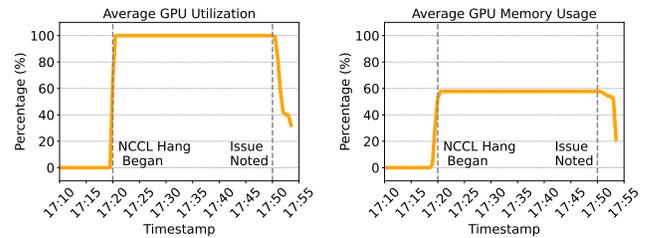


Figure 1: GPU utilization and memory usage metrics for a distributed model training on 256 A100 GPUs, where a hang issue occurred in NCCL since the job launched.

tailored to address the scale, complexity, and unique requirements of their vast and dynamic infrastructures. For instance, Google’s Borg [60] features robust monitoring tools that track task health and performance metrics, automatically restarting failed tasks and scaling up to tens of thousands of machines.

However, detecting anomalies in time with high accuracy is challenging due to the variety of anomalies. For example, Fig. 1 illustrates a real-world interruption, where distributed model training on 256 A100 GPUs encountered a network hang issue [44]. At timestamp 17:20, the issue occurred, and GPUs started to busy wait—with high utilization and memory usage—for communication to resume and stalled the training process. Although at first glance it seems normal that GPU utilization is saturated and GPU memory is effectively utilized, in normal training process there should be variation for GPU utilization and GPU memory, which is not observed in the figure. Monitors equipped with manually-written anomaly rules failed to detect the issue in time, which was only mitigated after human intervention at timestamp 17:50 by terminating training, resulting in significant waste of resources and time. Such network hanging incidents have also been observed in multiple large companies [14, 24, 66]. To improve the accuracy of the monitors, the engineers have to manually update them with new anomaly rules such as “All GPUs are continuously running at 100% utilization for periods exceeding 15 minutes”.

\*This work was primarily done during an internship at Microsoft Research.

Prior work [10, 28, 34, 46, 74], as well as our own experience deploying anomaly detection systems for large-scale incident management at Microsoft, highlights three indispensable properties for designing such systems:

(i) **Explainability**. No system is perfect, and false alarms are inevitable in anomaly detection systems. When alarms occur, on-call engineers (OCEs) must be able to understand the underlying reasons for these outcomes. An explainable anomaly detection system will enable OCEs to easily improve any inaccuracies. (ii) **Reproducibility**. The system should also produce consistent results for the same input metrics, ensuring that when an alarm is triggered in production, OCEs can reproduce it in a different environment and conduct further root cause analysis. A reproducible anomaly detection system will also eliminate non-deterministic alarms, avoiding wasted engineer effort. (iii) **Autonomy**. The system also needs to be frequently updated as data distributions shift when new metrics or dominant workloads are introduced [5, 32]. An autonomous anomaly detection system can adapt to these changes in data distribution without any human intervention.

Prior work on time-series anomaly detection can be broadly categorized into three directions, yet none of these methods simultaneously address explainability, reproducibility, and autonomy. *Conventional deep learning-based methods* [38, 49, 51, 56, 59, 63, 67, 68, 70, 75] often lacks explainability since they generate anomaly labels directly from input data. To improve model accuracy, engineers typically tune hyperparameters or model architectures, which makes these methods have only partial autonomy. *LLM-based methods* [4, 13, 18, 35] enhance explainability and autonomy by enabling OCEs to prompt with anomaly descriptions and providing anomaly labels along with explanations for why the data is classified as anomalous. However, due to the inherent non-determinism of LLMs [45, 57], these methods suffer from a lack of reproducibility and often produce inconsistent results when the same data is input across multiple trials. Consequently, *rule-based methods* [11, 37, 69] are widely used in industry for time-series anomaly detection to achieve explainability and reproducibility. These methods use anomaly detection rules which are easy for developers to understand, as the exact monitor logic is explicitly outlined and can be understood both prior to deploying the monitor and after the monitor has been triggered. However, current rule generation and threshold tuning heavily rely on manual efforts, thereby lacking autonomy. As demonstrated by the example in Fig. 1, this results in cases where monitors haven't been configured correctly, due to a lack of developer resources or human error in developing the rules.

This paper explores how to simultaneously achieve explainability, reproducibility, and autonomy in anomaly detection systems. We observe that structured detection rules serve as an effective intermediate representation for such systems. The rule-based method suggests that the detection rules can be written in the format of executable code, which is both re-

producible and explainable. On the other hand, LLMs can be leveraged to make rule generation autonomous for time-series anomaly detection. LLMs have shown promising results in time-series tasks [18, 25, 47], demonstrating a strong understanding of data patterns. Moreover, LLMs can generate executable code for various tasks [7, 12, 52]. These capabilities make LLMs an ideal candidate for autonomously generating rules.

A key insight of our work is that integrating LLM-generated rules with classical rule-based methods bridges the gap between autonomy and explainability while retaining reproducibility. Unlike existing LLM-based methods [4, 13, 18, 35] that leverage LLMs at runtime detection, which often suffer from randomness and lack reproducibility, we use LLMs to identify and codify the anomaly rules in the training phase. These explainable and reproducible rules are then deployed in runtime to detect anomalies.

Nevertheless, employing LLMs to generate anomaly detection rules and implement in executable code presents unique challenges. First, LLMs can produce code or rules that have syntax errors or are inaccurate due to a misunderstanding of data patterns. It is challenging to address the syntactic and accuracy issues. Second, it is hard to guarantee that the anomaly rules autonomously generated by LLMs have better accuracy than existing production anomaly detection systems, which have been well-tuned over years of use. Finally, due to the inherent randomness in LLM behavior, producing accurate anomaly detection rules with a limited number of trials (as LLM generation is expensive) remains challenging.

To address these challenges, we propose ARGOS, a time-series anomaly detection system that autonomously generates rules using LLMs. First, ARGOS employs an agent-based pipeline with feedback loops to iteratively correct anomaly detection rules and improve accuracy. In each iteration, multiple agents collaborate to propose, validate, fix, and refine the rules, reducing both syntax errors and improving accuracy. Second, to further guarantee better accuracy, ARGOS aggregates the predictions of its anomaly detection rules with the predictions of existing anomaly detection systems. During training, ARGOS primarily learns data patterns from the incorrect samples identified by an existing anomaly detector. During runtime inference, ARGOS uses an aggregation algorithm to merge the predictions from both the rules and the existing anomaly detectors to generate the final anomaly prediction. Finally, to improve the efficiency of generating accurate anomaly detection rules, ARGOS simultaneously proposes a set of  $n$  rule candidates in each iteration and selects the best  $k$  rules for further refinement in the next iteration.

We evaluate ARGOS on two widely used public time-series anomaly detection datasets, KPI [33] and Yahoo [27], as well as an internal dataset collected from Microsoft. Compared to the best baselines, our proposed system improves the average  $F_1$  score by 9.5% and 4.8% on the KPI [33] and Yahoo [27] datasets, respectively. It also achieves up to a sig-

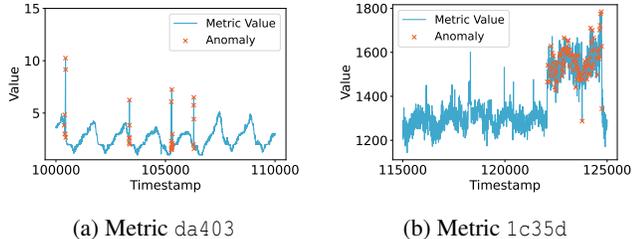


Figure 2: Two example metrics from the KPI dataset, where the blue line shows the metric data and the orange crosses represents the anomalies.

nificant 28.3%  $F_1$  score improvement on our internal dataset. Besides, ARGOS speeds up inference by 3.0 $\times$ , 34.3 $\times$ , and 1.5 $\times$  on the KPI, Yahoo, and Internal datasets, respectively.

In summary, our contributions are as follows:

- We show that current state-of-the-art time-series anomaly detection systems fall short in simultaneously achieving explainability, reproducibility, and autonomy.
- We observe that LLMs could be employed to autonomously generate explainable and reproducible rules for anomaly detection.
- We propose ARGOS, a time-series anomaly detection system that autonomously trains and deploys anomaly detection rules through an LLM-based agentic pipeline.
- We evaluate ARGOS on public and internal datasets, demonstrating its effectiveness and efficiency compared with state-of-the-art methods.

## 2 Motivation and Challenges

### 2.1 Why Existing Methods Fall Short?

**Background.** In cloud infrastructure, various metrics such as CPU usage, memory consumption, network latency, disk I/O, and others are periodically collected and continuously monitored to ensure the health of cloud services [9, 48, 72]. Anomalies, which represent deviations in key operational metrics that significantly diverge from normal patterns, can be caused by various factors in production, including hardware failures, software bugs, resource contention, and so on [14, 21, 24, 64, 69]. In practice, there is a variety of anomaly patterns, and each application or each metric in the cloud infrastructure may exhibit different behaviors. Fig. 2 shows two example metrics from the KPI dataset [33]. The anomalies in Fig. 2a are characterized by multiple sudden spikes in the metric values, while the anomalies in Fig. 2b are represented by a persistent shift-up.

**Rule-Based Methods.** In monitoring systems, it is common practice for engineers to write rules to detect anomalies for monitoring metrics [11, 37, 69]. These rules are typically based on domain knowledge and the engineers’ experience

Table 1:  $F_1$  scores of different methods on example metrics.

Method	da403	1c35d	
FCVAE (best setting)	0.97	0.80	
Manual Rule	(threshold=1)	0.05	0.01
	(threshold=3)	<b>0.99</b>	0.17
	(threshold=5)	0.94	0.43
LLM Rule	(for da403)	<b>0.99</b>	N/A
	(for 1c35d)	N/A	<b>0.91</b>

```

1 def rule(sample: np.ndarray, threshold: float)
2     -> np.ndarray:
3     # Get values and create labels
4     values = sample[:, 0]
5     labels = np.zeros(sample.shape[0])
6     # Calculate mean and standard deviation
7     mean_val = np.mean(values)
8     std_val = np.std(values)
9     # If a value is more than
10    # threshold * standard deviations away
11    # from the mean, it is considered abnormal
12    labels[np.abs(values - mean_val) >
13           threshold * std_val] = 1
14    return labels

```

Figure 3: An example rule written by human and implemented in Python, modified from [37].

with the metrics. Fig. 3 demonstrates an example anomaly detection rule implemented in Python to detect anomalies in time-series data, adapted from the rule used in [37]. The rule function takes a time-series sample and a threshold as input, and returns a sequence of labels indicating whether each data point is abnormal or not.

Table 1 compares the  $F_1$  scores of the FCVAE model [63], a state-of-the-art DL-based model, and the manual rule with different thresholds on the two example metrics shown in Fig. 2. We observe that the result of the manual rule is highly sensitive to the specific characteristics of the anomalies in each metric, thereby which may be tackled using specific thresholds yielding a high  $F_1$  score. While the manual rule performs similarly to the FCVAE model on the da403 metric, it consistently underperforms on the 1c35d metric across different thresholds. This discrepancy is caused by different anomaly patterns in two metrics, highlighting that no single ‘one-size-fits-all’ rule can effectively handle all types of anomalies. When new anomalies or metrics emerge in cloud services, engineers must manually design or adjust anomaly detection rules, which is time-consuming and requires significant expertise [69].

**Conventional DL-Based Methods.** Deep learning (DL) models have shown promising results in time-series anomaly

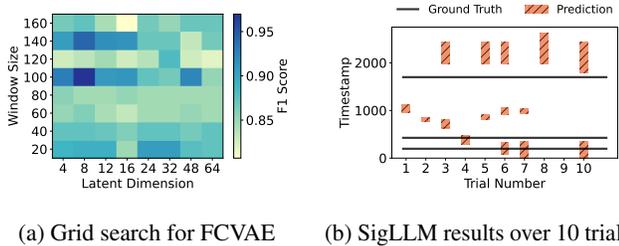


Figure 4: Analysis of prior methods on the KPI dataset metric da403.

detection by learning data labels to classify anomalies [38, 49, 51, 56, 59, 63, 67, 68, 70, 75]. However, the lack of explainability in these models poses a significant challenge for engineers in practice. These DL models contain many hyperparameters, which are difficult to interpret in terms of their impact on anomaly detection performance. As a result, engineers often need to perform an exhaustive grid search across a wide range of hyperparameters to find the optimal configuration, which can be both time-consuming and computationally expensive.

Fig. 4a shows the grid search results for the FCVAE model [63] on the da403 metric in the KPI dataset, considering two hyperparameters: the latent dimension and the window size. The latent dimension defines the dimension of the hidden space where the time-series input is projected, while the window size determines the number of data points the model uses for prediction. We observe that the model  $F_1$  score varies significantly across different hyperparameter configurations, indicating the difficulty of finding the optimal hyperparameters for the model. Although a larger latent dimension incorporates more information in the hidden space, it does not always result in better model performance. For the window size, setting it to 100 or 140 performs significantly better than a setting of 120, yet there is no clear intuition behind this performance difference.

**LLM-Based Methods.** Recent advancements in large language models (LLMs) have expanded their application in time-series anomaly detection [4, 13, 18, 35], where they accept data from prompts to predict anomalies and optionally generate natural language explanations, thereby enhancing result explainability. However, due to the stochastic nature of LLMs, their outputs can exhibit high variance across different trials, sacrificing reproducibility.

Fig. 4b shows the  $F_1$  scores of the SigLLM system [4] over ten trials on the da403 metric. We observe that the output labels of the LLMs can vary significantly in terms of the number and locations of the anomalies detected, with no two trials producing the same predictions. Four out of ten trials (No. 4, 6, 7, and 10) are able to detect one anomaly partially, while there exists three ground-truth anomalies in the metric. If a majority vote is performed to produce a consistent final result from these trials [61], none of the anomalies will be correctly detected.

```

1 values = sample[:, 0]
2 labels = np.zeros(sample.shape[0])
3 # Calculate z-scores
4 z_scores = np.abs(stats.zscore(values))
5 # Abnormal Rule 1: If z-score is greater
6 # than 3, the data point is abnormal
7 labels[z_scores > 3] = 1

```

(a) Metric da403

```

1 for i in range(window_small, len(values)):
2     last_N = values[i-window_small:i]
3     last_M = values[i-window_large:i]
4     avg_N = np.mean(last_N)
5     avg_M = np.mean(last_M)
6     # Abnormal Rule 1: A value differs from
7     # the mean of last N values by over 20%
8     if values[i] > 1.2 * avg_N or \
9         values[i] < 0.8 * avg_N:
10        labels[i] = 1
11    # Abnormal Rule 2: The mean of last N
12    # values differs from the mean of last M
13    # values by more than 20%
14    elif avg_N > 1.2 * avg_M or \
15        avg_N < 0.8 * avg_M:
16        labels[i-window_small:i] = 1

```

(b) Metric 1c35d

Figure 5: Anomaly rules and code generated by LLM.

## 2.2 How LLMs can Help in Rule Generation?

Although anomaly detection rules are not autonomously generated, they are both explainable and reproducible. For instance, the manual rule in Table 1 provides an explainable parameter `threshold`, which allows engineers to tune the sensitivity of the rule to deviations from the mean. Since the threshold is fixed during deployment and the rule’s logic is deterministic, it also ensures the reproducibility of the detection results.

To address the lack of autonomy in rule generation, our intuition is to leverage LLMs to create and tune anomaly detection rules for time-series data, and to implement them in executable code, mimicking the rule development process of engineers. Although LLMs have been used in time-series anomaly detection, prior work [4, 13, 18, 35] only focuses on involving LLMs during deployment time to directly predict anomalies from the time-series data. Instead, we propose to use LLMs in a training phase before the deployment to generate anomaly detection rules from time-series data collected and labeled offline. The rules are then deployed in production to monitor the metrics, where LLMs are not involved in the real-time detection process.

## 2.3 Opportunities and Challenges in Applying LLMs

Fig. 5 shows the promising results of anomaly rules and code generated by the LLM for the two example metrics. We prompt the LLM with time-series data and ask it to first propose an anomaly detection rule in natural language and then write a Python implementation. For the `da403` metric, the LLM generates a rule functionally equivalent to the manual rule, where data points are flagged as anomalies if their z-scores are greater than 3. For the `1c35d` metric, the LLM generates a more complex rule that detects both spikes (Abnormal Rule 1) and level shifts (Abnormal Rule 2) in the metric values. Both LLM-generated rules achieve similar or better  $F_1$  scores compared to the manual rules, as shown in Table 1, demonstrating the potential of LLMs in generating effective rules for time-series anomaly detection while simultaneously satisfying explainability, reproducibility, and autonomy.

Although the initial results are promising, there exist several unique challenges in applying LLMs to anomaly detection rule generation.

**Correctness and Accuracy Issues.** The anomaly detection rules are implemented in code. Despite being pretrained on billions of lines of public source code [7], LLMs may still generate rule implementations that contain syntax errors. Additionally, misunderstandings in the patterns of time-series data can lead to inaccuracy in both the description of the anomaly detection rules and their implementation.

For example, when using LLMs to generate anomaly detection rules 50 times for each metric in the KPI dataset, we observe an overall rate of 4.8% syntax errors in the code implementation. In addition, for the rules that are correct, the average  $F_1$  score across all metrics is only 0.129, while the best deep learning-based method achieves a score of 0.819.

**No Accuracy Guarantee.** A mature anomaly detection system in production will likely have well-tuned models for each metric over time. Even if correctness and accuracy issues in rule generation are addressed, there is no guarantee that LLM-generated rules will always outperform existing mature anomaly detection systems. For instance, compared to the best existing model in the KPI dataset, we observe that the best LLM-generated rules have  $F_1$  score regression with respect to 3 metrics out of 19 metrics.

**Low Efficiency.** LLMs can generate a diverse range of responses even under identical inputs. As a result, the anomaly detection system will need to invoke the LLM for multiple trials repeatedly, with each trial reflecting a different interpretation of the input data by the LLM, leading to variability in the accuracy of the generated rules. Achieving accurate rules with a small number of trials is challenging due to the inherent variability in LLM outputs. For instance, the LLM takes an average of 113 minutes to converge to an accurate rule on the KPI dataset.

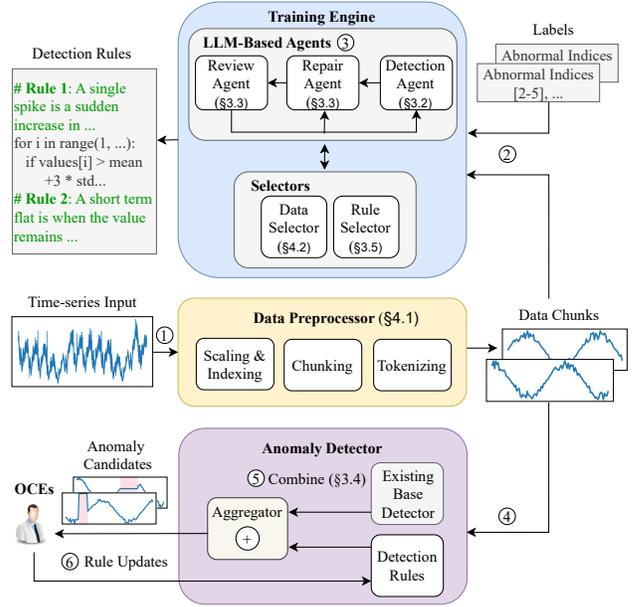


Figure 6: The overall design of ARGOS.

## 3 System Design

### 3.1 Overview

As shown in Fig. 6, ARGOS detects anomalies in time-series data using a three-stage approach: data preprocessing, rule training, and deployment.

- Data Preprocessing.** Upon receiving a time-series input, ① the Data Preprocessor first chunks the input into smaller segments and applies data preprocessing techniques such as scaling and indexing.
- Rule Training.** During the rule training stage, ② the data chunks and corresponding ground-truth labels are fed into the Training Engine, which iteratively trains detection rules based on the preprocessed data chunks and labels using an agentic pipeline. The pipeline consists of three agents: the Detection Agent, the Repair Agent, and the Review Agent. ③ In each iteration, the Detection Agent first proposes a set of detection rules based on the input data. The Repair Agent then checks the proposed rules for syntax errors and corrects any issues. Next, the Review Agent evaluates the accuracy of the proposed rules using validation data. If any issues are detected, the rules will be sent back to the Repair Agent; otherwise, they will be fed back to the Detection Agent to incorporate new rules. This iterative loop continues to improve accuracy monotonically.
- Deployment.** During deployment, ④ the Anomaly Detector receives processed data chunks from the runtime time-series data. ⑤ An Aggregator combines the outputs from a base detector in the existing system and the anomaly detection rules to identify anomalies in the input data chunks. If

```

1  # import necessary libraries for your code
2  def inference(sample: np.ndarray)
3      -> np.ndarray:
4      labels = np.zeros(len(sample))
5      # Your comment to describe
6      # how normal data behave
7      # Normal Rule 1
8      # Normal Rule 2
9      # Your code to detect
10     # if the given sample is abnormal
11     # Abnormal Rule 1
12     if ...
13     # Abnormal Rule 2
14     if ...
15     # return labels as a 1d numpy array
16     return labels

```

Figure 7: Code Template for the Detection Agent in ARGOS.

an anomaly is detected, the Anomaly Detector will report an incident, allowing on-call engineers to perform root cause analysis and incident mitigation. ⑥ Engineers can also update the detection rules based on incident feedback to improve detection accuracy.

### 3.2 Autonomous Rules Generation

Detection rules must be in a format suitable for LLM generation, easily understandable by engineers, and generalizable across different types of anomalies. ARGOS chooses Python as the implementation language for anomaly detection rules. Python is the dominant programming language in datasets used to pretrain LLMs [7], and it is commonly used as the target language when finetuning LLMs for code-related tasks [52]. Furthermore, as a Turing-complete language, it can perform any computation required for detecting anomalies in time-series data.

Fig. 7 shows the code template for the Detection Agent in ARGOS. The Detection Agent is tasked with writing a Python function, `inference(sample: np.ndarray) -> np.ndarray`, to output prediction labels based on the input sample. We prompt the Detection Agent to first describe the abnormal rules in the form of comments, and then write the corresponding Python code that implements these abnormal rules. This step-by-step process generates commented anomaly detection rules that are easily understandable and verifiable by engineers. Additionally, the Detection Agent is asked to describe the behavior of normal data in the comments and ensure that the implementation of the abnormal rules does not conflict with the normal rules. The complete prompt used for the Detection Agent could be found in [Appendix A](#).

### 3.3 Correctness and Accuracy Improvement via Feedback Loops

Despite clear instructions in the prompt, LLMs may still generate anomaly detection rules with syntax errors or inaccuracies, making it difficult to ensure the correctness and accuracy. Inspired by backpropagation in deep learning training [30, 50], ARGOS introduces an iterative feedback loop involving two agents, the Repair Agent and the Review Agent, to train rules in an iterative manner. The Repair Agent corrects syntax errors in the generated anomaly detection rules, while the Review Agent verifies their accuracy on a validation set. In each iteration, the two agents collaborate together to ensure that the anomaly detection rules are syntactically correct and that their performance on the validation set is at least as good as the previous iteration.

**Repair Agent.** Upon receiving anomaly detection rules from the Detection Agent, the Repair Agent first checks for syntax errors by executing the rules on dummy data that mimics the format of the input data. If syntax errors are detected, ARGOS provides error messages and call stack information to the Repair Agent, which then proposes a corrected version of the rules using a new prompt and re-checks them until all syntax errors are resolved.

**Review Agent.** The Review Agent evaluates the accuracy of the anomaly detection rules on the validation data. If the accuracy of the current anomaly detection rules is worse than the previous iteration, ARGOS provides a comparison of the accuracy metrics and the code differences between the two versions to the Review Agent. ARGOS also provides data samples where the rule from the previous iteration labels correctly, while the new rule labels incorrectly. The Review Agent then proposes an improved version of the rules based on these observations and re-evaluates the accuracy until there is no accuracy regression in current iteration. If the new anomaly detection rules contain syntax errors, they will be sent back to the Repair Agent for further correction. In practice, this process will not block training infinitely, as the Review Agent can revert the code changes from the current iteration, restoring accuracy to the previous iteration’s level.

### 3.4 Accuracy Guarantee via Model Fusion

Due to the lack of domain knowledge, LLMs may generate anomaly detection rules that underperform compared to existing anomaly detectors that have been well-tuned over time in production. Instead of directly using the LLM-generated rules, ARGOS proposes a model fusion approach that combines these rules with the well-established anomaly detectors deployed in production, ensuring an accuracy guarantee.

During the training stage, ARGOS separately trains two sets of anomaly detection rules: one from false negatives and the other from false positives outputted by the existing base detector. ARGOS defines false negatives to be ground-truth

---

**Algorithm 1** Anomaly Prediction Aggregation Algorithm

---

**Input:** predicted labels from base detector  $L_{base}$ , from false positive rules  $L_{fp}$ , and from false negative rules  $L_{fn}$

**Output:** final predicted labels  $L_{agg}$

```
1: function AGGREGATION( $L_{base}$ ,  $L_{fp}$ ,  $L_{fn}$ )
2:    $L_{agg} \leftarrow L_{base}$ 
3:   for  $t \leftarrow 1$  to  $length(L_{base})$  do
4:     if  $L_{base}[t] = \text{normal}$  and  $L_{fn}[t] = \text{abnormal}$  then
5:        $L_{agg}[t] \leftarrow \text{abnormal}$ 
6:     end if
7:     if  $L_{base}[t] = \text{abnormal}$  and  $L_{fp}[t] = \text{normal}$  then
8:        $L_{agg}[t] \leftarrow \text{normal}$ 
9:     end if
10:  end for
11:  return  $L_{agg}$ 
12: end function
```

---

anomaly samples that are mis-classified by the base detector to be normal. Similarly, false positives are normal samples that are mis-classified by the base detector to be abnormal. To train the set of anomaly detection rules from false negative samples, ARGOS first performs inference on the training data using the existing base detector and identifies false negatives by comparing the predictions of the base detector with ground-truth labels. It then trains anomaly detection rules by feeding the false negative samples and their corresponding ground-truth labels into the Training Engine. To prevent overfitting to the false negatives, ARGOS also provides sampled true negative data to the Training Engine, and the Detection Agent is instructed to generate rules that detect anomalies in the false negative samples while excluding the true negative samples. In each iteration, the validation accuracy of the anomaly detection rules is evaluated by combining the outputs of the rules and the existing base detector on the validation set using the Aggregator. ARGOS trains detection rules for false positives in a similar manner.

During the deployment stage, the Anomaly Detector receives processed data chunks from the runtime time-series data. The Aggregator then combines the output labels from the existing base detector with the anomaly detection rules for false negatives and false positives to identify anomalies in the input time-series data chunks.

**Aggregator.** Algorithm 1 presents the aggregation algorithm for the Aggregator in ARGOS. The anomaly detection rule for false negatives effectively corrects the normal labels from the base detector but does not address the abnormal labels. Similarly, the anomaly detection rule for false positives corrects the abnormal labels from the base detector but does not affect the normal labels. This is because a set of anomaly detection rules only look at either false negative or false positive examples at a time during the training, which makes the anomaly detection an one-class classification task [53].

### 3.5 Efficiency Enhancement

Due to the stochastic nature of the autoregressive process in LLMs, they may not generate optimal anomaly detection rules within a small number of trials. Inspired by the beam search algorithm [58], ARGOS employs a top- $k$  selection strategy to identify the best rules and early terminates the inaccurate rules during training. In each iteration, the Detection Agent uses the same input to propose  $n$  detection rules, which are passed to both the Repair Agent and the Review Agent. Once the Review Agent verifies that there is no accuracy regression in the generated  $n$  rules, the Rule Selector selects the top- $k$  rules based on user-defined criteria. Currently, ARGOS chooses validation accuracy as the criterion, selecting the  $k$  rules with the highest accuracy on the validation set. The Rule Selector can also be extended to incorporate additional criteria, such as the inference time of the anomaly detection rules.

## 4 Implementation

### 4.1 Data Preprocessing

Since time-series data are continuous and unbounded, they must first be chunked into smaller segments for processing by the Training Engine. Meanwhile, the Data Preprocessor also applies scaling, indexing, and tokenization-specific preprocessing, which are essential for LLMs to learn patterns in the data.

**Scaling and Indexing.** Time-series inputs may have arbitrary precisions, which limits the amount of data that can fit within the context window of LLMs. To address this, the Data Preprocessor scales the input data to a specified significant figure. Similarly, timestamps, often represented by long numeric values, are not informative to LLMs. Therefore, the Data Preprocessor removes the timestamp and re-indexes the data based on the order of the data points. ARGOS assumes that data points are collected at fixed time intervals.

**Chunking.** Selecting an appropriate chunk size is crucial for training anomaly detection rules. A chunk size that is too small may lack sufficient contextual information for the LLMs to learn the data patterns, while a chunk size that is too large may exceed the context window size of LLMs or result in overly general rules. To determine an appropriate chunk size, ARGOS performs calibration on a metric of a given dataset. ARGOS uses only the Detection Agent to repeatedly propose anomaly detection rules on the given metric, and selects the chunk size that results in the highest  $F_1$  score on the training set.

**Tokenizer-Specific Preprocessing.** Recent studies have shown that different tokenizers handle numerical values in different ways [4, 18, 43], which impacts how LLMs interpret the data. For the same numerical value, tokenizers may segment the value into chunks that do not align with its individual digits. To address this issue, the Data Preprocessor

applies tokenizer-specific preprocessing to the data based on the type of LLM used in the Training Engine. For GPT-based models, the Data Preprocessor adds space between each digit in a time-series value, similar to techniques used by prior work [4].

## 4.2 Data Selection

**Automatic Metrics Selection.** Given a time-series dataset, the Data Selector controls how the dataset is partitioned and which partition to use during the training process. The Data Selector supports two dataset modes: *one-for-one* and *one-for-all*. The one-for-one mode is suitable for datasets with a large number of continuous metrics, as it allows the Training Engine to focus on learning the patterns of each metric individually. The one-for-all mode is useful when the user wants to train a set of anomaly detection rules that generalize across multiple metrics. ARGOS first chunks each metric in the dataset using the chunk size determined by the Data Preprocessor. For metrics with at least 10 data chunks (i.e., where at least 10 training iterations can be run), ARGOS uses one-for-one mode to train these metrics. For the rest metrics, ARGOS uses the one-for-all mode. Additionally, ARGOS allows users to directly specify the dataset modes.

In the one-for-one mode, the Data Selector partitions the dataset based on the number of continuous time-series metrics. In each training trial, the Data Selector selects one continuous metric from the dataset to train a set of anomaly detection rules.

In the one-for-all mode, the Data Selector partitions the dataset based on a group identifier provided by the user. Each group contains a set of continuous time-series metrics. In each training trial, the Data Selector selects one group from the dataset to train a set of anomaly detection rules. This mode also serves as a data augmentation technique when the length of each continuous metric is short.

**Contrastive Example Retrieval.** When training anomaly detection rules from false negative or false positive examples of existing anomaly detection models, the Data Selector needs ensure that the contrastive examples retrieved (i.e. true negatives or true positives) are from a similar distribution as the false negative or false positive examples. This is because selecting true negative or true positive examples from a different data distribution may lead to false assumptions about the data patterns, which can result in the generation of inaccurate anomaly detection rules.

In the *one-for-one* mode, the Data Selector retrieves true negative or true positive examples randomly, since all data samples are from the same distribution. In the *one-for-all* mode, the Data Selector first calculates the mean and standard deviation of the false negative or false positive examples, then retrieves true negative or true positive examples sorted by their Euclidean distance to the mean and standard deviation of the false negative or false positive examples. We believe more

Table 2: An example of different evaluation metrics in time-series anomaly detection.

<b>Ground-Truth:</b> [0, 1, 1, 0, 1, 1, 1, 1, 0, 0]						
<b>Predictions:</b> [1, 1, 0, 0, 0, 0, 0, 0, 1, 1]						
Eval Method	TP	FP	FN	Pr	Re	$F_1$
Point- $F_1$	2	3	4	0.40	0.33	0.36
Point- $F_1$ PA	6	3	0	0.66	1.00	0.80
Overlap- $F_1$	2	0	0	1.00	1.00	1.00
Event- $F_1$ PA	2	3	0	0.40	1.00	0.57

advanced retrieval techniques based on embedding similarity can be applied to further improve the retrieval quality and leave it as future work.

## 5 Evaluation

### 5.1 Experiment Setup

**Metrics.** ARGOS uses the  $F_1$  score as the primary evaluation metric when comparing against baselines. The  $F_1$  score is calculated as the harmonic mean of precision and recall. Precision is the ratio of true positives (TP) to the sum of true positives and false positives (FP), while recall is the ratio of true positives to the sum of true positives and false negatives (FN). Formally, the  $F_1$  score is calculated as:

$$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (1)$$

where Precision =  $\frac{TP}{TP + FP}$  and Recall =  $\frac{TP}{TP + FN}$ .

In time-series anomaly detection, defining positive and negative samples in the context of the application is crucial [55]. In cloud infrastructure monitoring, each anomaly is treated as an incident, which may span multiple data points across time. For example, the ground-truth (GT) labels in Table 2 contain two incidents: one spanning indices 1-2 and the other spanning indices 4-7. The prediction labels correctly identify both incidents, but only partially overlap with the ground-truth indices. As illustrated in Table 2, four different evaluation methods are used by prior work to compute the  $F_1$  score for example predictions based on the ground-truth:

- Point-Based  $F_1$  Score (Point- $F_1$ ) [26]: Treats each data point as an individual instance, often leading to a very low  $F_1$  score.
- Point-Based  $F_1$  Score with Point Adjustment (Point- $F_1$  PA) [70]: Considers all points within a segment to be detected if at least one point in the segment is detected, often leading to a high  $F_1$  score.
- Overlap  $F_1$  Score (Overlap- $F_1$ ) [3]: Treats each incident as a single instance but ignores false positives in the prediction, often resulting in an unreasonably high  $F_1$  score.

- Event-Based  $F_1$  Score with Point Adjustment (Event- $F_1$  PA) [16]: Treats each incident as a single instance and penalizes precision by treating each data point outside the ground-truth incident as a false positive, leading to a more balanced  $F_1$  score.

In ARGOS, we use Event- $F_1$  PA as the primary evaluation method for calculating the  $F_1$  score metric.

**Baselines.** We compare the proposed ARGOS against various state-of-the-art methods, including five DL-based methods from EasyTSAD framework [55] and two LLM-based methods:

- AnomalyTransformer [68]: An unsupervised model using the Anomaly-Attention mechanism to detect anomalies by exploiting differences in association patterns between normal and abnormal points.
- AutoRegression [51]: A supervised model with several linear layers that transform input data into anomaly score logits with same length as the input.
- FCVAE [63]: An unsupervised model that integrates both global and local frequency features to capture similar yet different periodic patterns and detailed trends in time series.
- LSTMAD [38]: A supervised LSTM model that trains on normal data and detects anomalies using a statistical strategy based on the prediction error for observed data.
- TFAD [70]: A supervised model that uses time-series decomposition to transform input data into the frequency domain, leveraging both frequency and time-domain features for anomaly detection.
- LLMAD [35]: An LLM-based method that prompts the LLM with time-series data, in-context learning examples, and contextual information for anomaly detection.
- SigLLM [4]: An LLM-based method in Detector mode and Prompter mode. The Detector mode prompts LLMs to predict the next steps in the time series and detects anomalies by comparing predictions with actual output, while the Prompter mode directly prompts the LLMs with the time-series data to identify anomaly indices.

**Datasets.** We evaluate the proposed ARGOS on three datasets: KPI, Yahoo, and Internal.

- KPI Dataset [33]: A real-world dataset with manually labeled incidents collected from five large internet companies, comprising 27 continuous metrics over several months. Due to the scarcity of anomalies in some metrics, we filter them out and use the remaining 19 metrics for evaluation.
- Yahoo Dataset [27]: The dataset contains 367 real and synthetic time series across four partitions  $A_1 \sim A_4$ , with anomalies exhibiting various characteristics, including spikes, trends, and seasonal changes.

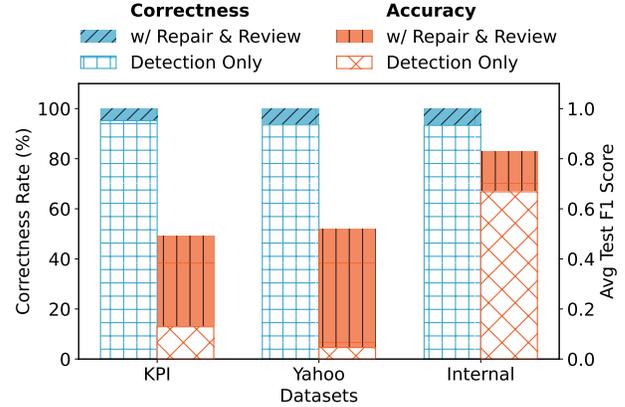


Figure 8: Comparison of the correctness rate and average test  $F_1$  score of the Training Engine with only the Detection Agent versus full Training Engine with Repair and Review Agents.

- Internal Dataset: The dataset is collected from an AI training platform at Microsoft and includes 20 hardware metrics such as GPU utilization, GPU memory usage, CPU utilization, disk usage, and network traffic.

We set the split ratio to 0.7 for each metric or partition in the datasets to ensure a similar anomaly ratio between the training and test sets. For Yahoo dataset  $A_2$  partition, we use a 0.5 split ratio to ensure both sets contain anomaly samples. The chunk size is set to 2500 for the KPI dataset, 500 for the Yahoo dataset and 1000 for the Internal dataset. We use one-for-one dataset mode for the KPI dataset, where ARGOS trains a set of anomaly detection rules for each metric. We use one-for-all dataset mode for the Yahoo dataset and the Internal dataset, where ARGOS trains a set of rules for each partition and each hardware metric respectively.

**Test Machines.** All LLM experiments are conducted on an Azure Standard\_D16as\_v4 VM [41] with 16 vCPUs and 64 GiB memory, while all model training and evaluation are performed on an Azure Standard\_ND96amsr\_A100\_v4 VM [42] with 96 vCPUs and 8 NVIDIA A100 GPUs.

**LLM Endpoints.** We use the Azure OpenAI service [40] to access the endpoint of LLM models [39], including GPT-3.5 [6], GPT-4-32k [1], and GPT-4o [22], for all LLM experiments in both baselines and ARGOS.

## 5.2 Correctness and Accuracy Improvement

To demonstrate how feedback loops with the Repair and Review Agents improve the correctness and accuracy of the anomaly detection rules, we evaluate the Training Engine in two settings: with only the Detection Agent and with all three agents including the Detection, Repair, and Review Agents. We run each metric or partition in the dataset for 50 trials with each trial containing 20 iterations. In the Detection Agent only setting, at each iteration, the agent receives the time-series

Table 3:  $F_1$  Score comparison on all metrics where **ARGOS w/o Aggregator** has regressions compared to baseline model.

Metric	Baseline Model	ARGOS w/o Aggregator	ARGOS
KPI-02e99	0.99	0.96 (-0.03)	0.99 (+0.00)
KPI-07927	0.99	0.67 (-0.32)	0.99 (+0.00)
KPI-1c35d	0.89	0.80 (-0.09)	0.99 (+0.10)
Yahoo-A2	0.90	0.87 (-0.03)	0.95 (+0.05)
Yahoo-A4	0.85	0.84 (-0.01)	0.87 (+0.02)

data along with the anomaly detection rule from the previous iteration. For correctness rate, we measure how many anomaly detection rules generated in each trial have no syntax errors. For accuracy, we calculate the average  $F_1$  score on the test set for anomaly detection rules generated in all trials of all iterations.

Fig. 8 shows the comparison of correctness rate and average test  $F_1$  score between two settings. We observe that the correctness rate of the Detection Agent only setting is 95.2% in the KPI dataset, 93.5% in the Yahoo dataset, and 93.3% in the Internal dataset. With the help of the Repair Agent, all anomaly detection rules generated by the Training Engine have no syntax errors. In addition, the Review Agent helps improve the average test  $F_1$  score of the anomaly detection rules by  $3.8\times$  in the KPI dataset,  $11.3\times$  in the Yahoo dataset, and  $1.2\times$  in the Internal dataset, clearly demonstrating the effectiveness of feedback loops in the Training Engine.

### 5.3 Accuracy Guarantee

To demonstrate how model fusion provides an accuracy guarantee for the anomaly detection rules, we compare two versions of ARGOS, one without and one with the Aggregator module. In the version without the Aggregator, we train the rules on all time-series data from the training set of each dataset, and during inference, the prediction labels are directly generated from the rules. In the version with the Aggregator, we first select the deep learning model with the highest  $F_1$  score on the training set as the base detector for each dataset, then collect false positives and false negatives in the training set from this base detector to train the anomaly detection rules. Table 3 shows all metrics in three datasets where ARGOS without the Aggregator has accuracy regressions compared to the baseline. In the KPI and Yahoo datasets, ARGOS without the Aggregator shows accuracy regressions on 3 and 2 metrics, respectively, with up to 32% accuracy drop. In contrast, ARGOS with the Aggregator has no accuracy regressions across any metrics in all three datasets.

### 5.4 Efficiency Enhancement

To measure the efficiency of the top- $k$  rule selection, we compare the accuracy of ARGOS and ARGOS with no selection. In the no-selection setting, the Detection Agent proposes a single rule per iteration, which is then validated by the Repair and Review Agents. In top- $k$  rule selection, the Detection Agent proposes 5 rules in parallel, all of which are validated, and the best-performing rule is selected. In the next iteration, the Detection Agent proposes new 5 rules based on the best rule from the previous iteration. We use GPT-4-32k as the LLM back-end for evaluation and run each metric or partition for 5 trials. We compare the average  $F_1$  scores on the training set for the generated rules across all trials, with both settings executed by ARGOS without the Aggregator.

Fig. 9 shows the average  $F_1$  score across all trials and metrics for each dataset on the training set. For a fair comparison, we evaluate the average  $F_1$  score at points where the iteration number for no-selection is five times that of top- $k$  rule selection, as the latter generates 5 rules per iteration. On the KPI dataset, the  $F_1$  score for top- $k$  selection at iteration 8 is 0.736, outperforming 0.682 for no-selection at iteration 40. On the Yahoo dataset, top- $k$  selection achieves an  $F_1$  score of 0.597 at iteration 8, outperforming 0.523 for no-selection at iteration 40. On the Internal dataset, the  $F_1$  score for top- $k$  selection is 0.870 at iteration 8, outperforming 0.795 for no-selection at iteration 40. Additionally, we observe that the converged  $F_1$  scores at the end of the training are consistently higher with top- $k$  rule selection than with no-selection across all datasets. This demonstrates that top- $k$  rule selection accelerates training and improves the accuracy of generated rules.

### 5.5 End-to-End Results

For end-to-end results, we compare ARGOS with all baselines across three datasets, calculating the average precision, recall, and  $F_1$  score for each method on test sets.

For deep learning-based methods, we perform a grid search on hyperparameters for each dataset and select the configuration yielding the highest  $F_1$  score on the training set. For LLM-based methods, we evaluate 10 trials per metric on the test set and report the best accuracy across all trials. To match the configurations in the original papers, LLMAD uses GPT-4-32k as the LLM back-end [35] while SigLLM uses GPT-3.5 [4] in the baselines.

For ARGOS, we first select the baseline model with the highest  $F_1$  score on the training set as the base detector. We then train anomaly detection rules from the false negative samples (ARGOS w/ FN Rules Only) and false positive samples (ARGOS w/ FP Rules Only) of this base detector. These two sets of rules are then fused with the base detector using Algorithm 1 to generate the final results. We train each metric for 5 trials and report the best accuracy across all trials.

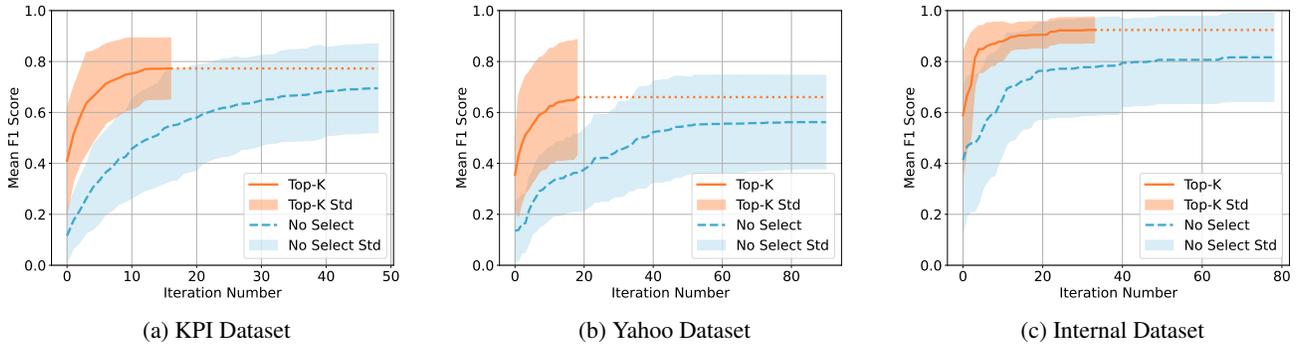


Figure 9: Average  $F_1$  score comparison on the training sets between top- $k$  rule selection and no selection.

Table 4: Accuracy comparison of different anomaly detection methods on three datasets.

Method	KPI			Yahoo			Internal		
	Precision	Recall	$F_1$	Precision	Recall	$F_1$	Precision	Recall	$F_1$
AnomalyTransformer [68]	0.482	0.395	0.282	0.297	0.055	0.058	0.721	0.581	0.598
AutoRegression [51]	0.731	0.699	0.668	0.508	0.623	0.526	0.735	0.618	0.582
FCVAE [63]	0.881	0.808	0.818	0.721	0.438	0.464	<b>0.944</b>	0.479	0.581
LSTMAD [38]	<b>0.887</b>	0.789	<b>0.819</b>	0.425	0.493	0.350	0.865	<b>0.655</b>	<b>0.724</b>
TFAD [70]	0.690	0.670	0.564	<b>0.830</b>	<b>0.726</b>	<b>0.773</b>	0.783	0.500	0.588
LLMAD [35]	0.033	<b>0.940</b>	0.053	0.090	0.687	0.142	0.623	0.575	0.537
SigLLM - Detector [4]	0.001	0.239	0.002	0.004	0.543	0.009	0.668	0.486	0.443
SigLLM - Prompter [4]	0.007	0.498	0.014	0.006	0.027	0.010	0.178	0.424	0.188
<b>ARGOS</b>	0.951	<b>0.859</b>	<b>0.897</b>	0.860	0.768	<b>0.810</b>	0.955	<b>0.910</b>	0.929
– w/ FN Rules Only	0.882	<b>0.859</b>	0.860	0.793	<b>0.772</b>	0.782	0.955	<b>0.910</b>	0.929
– w/ FP Rules Only	<b>0.997</b>	0.785	0.864	<b>0.920</b>	0.697	0.763	0.894	0.643	0.722
– w/o Aggregator	0.954	0.850	0.890	0.901	0.719	0.800	<b>1.000</b>	0.882	<b>0.936</b>

GPT-4o is used as the LLM back-end for ARGOS.

Table 4 shows the overall accuracy comparison of all methods across the three datasets. ARGOS outperforms all baselines in every dataset. In the KPI dataset, ARGOS achieves an  $F_1$  score of 0.897, which is 9.5% higher than the best baseline, LSTMAD. In the Yahoo dataset, ARGOS achieves an  $F_1$  score of 0.810, 4.8% higher than the best baseline, TFAD. In the Internal dataset, ARGOS achieves an  $F_1$  score of 0.936, 28.3% higher than the best baseline, LSTMAD. ARGOS w/ FN Rules Only achieves the highest recall in all datasets, demonstrating the effectiveness of addressing false negatives in the base model, as recall measures the false negative rate. Similarly, ARGOS w/ FP Rules Only achieves the highest precision in the two public datasets.

## 5.6 Cost Analysis

**API Budget for Training.** Table 5 shows the input and output token counts for each dataset. We collect these statistics from the training of anomaly detection rules using false negative

Table 5: Cost analysis of rule training per iteration.

Category	KPI	Yahoo	Internal
Input Tokens	38238	39064	70754
Output Tokens	1119	1669	909
Input API Budget (USD)	0.096	0.098	0.177
Output API Budget (USD)	0.011	0.017	0.009

and false positive samples. The average token count per iteration is calculated across all trials for each dataset. Using GPT-4o as the LLM backend, the cost of input tokens per iteration is up to \$0.177, while the cost of output tokens per iteration is up to \$0.017. For a total of 50 iterations, the cost of training the anomaly detection rules is under \$10 for all datasets in total.

**Runtime Latency for Inference.** Table 6 compares the inference latency of ARGOS with the baselines across three datasets. We perform inference on the test set and calculate

Table 6: Inference latency (seconds) comparison of different methods.

Method	KPI	Yahoo	Internal
AnomalyTransformer [68]	171.23	171.58	53.38
AutoRegression [51]	0.68	0.88	0.26
FCVAE [63]	20.04	25.67	7.82
LSTMAD [38]	2.92	3.15	0.90
TFAD [70]	49.19	67.43	19.08
<b>ARGOS w/o Aggregator</b>	<b>0.97</b>	<b>1.97</b>	<b>0.59</b>

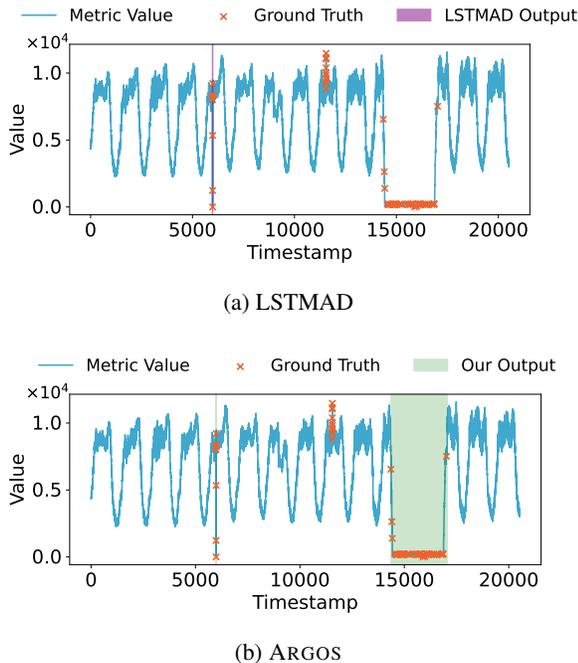


Figure 10: Comparison of ARGOS’s output with the LSTMAD baseline on the e0770 metric in KPI dataset.

the average latency for all metrics or partitions in the dataset. All baselines and ARGOS are run on a Standard\_D16as\_v4 VM [41] using the CPU. ARGOS achieves low inference latency compared to the baselines, as the anomaly detection rules, implemented in Python, are lightweight. Compared to the baseline with the highest average  $F_1$  score on each dataset, ARGOS achieves a  $3.0\times$  speedup on the KPI dataset, a  $34.3\times$  speedup on the Yahoo dataset, and a  $1.5\times$  speedup on the Internal dataset.

## 5.7 Case Study

In this case study, we demonstrate how the anomaly detection rules trained by ARGOS, based on an existing anomaly detection model, can improve model accuracy. We choose

```

1 labels = np.zeros(sample.shape[0], dtype=int)
2 # Abnormal Rule 1: If there are sudden spikes
3 # or drops in values.
4 spikes_or_drops = np.abs(np.diff(sample[:, 0]))
5 labels[1:][spikes_or_drops > 4000] = 1
6 # Abnormal Rule 2: If there is a prolonged
7 # period (e.g., more than 10 consecutive
8 # points) of extreme values.
9 extreme_values = (sample[:, 0] < 2000) |
10 (sample[:, 0] > 15000)
11 for i in range(len(extreme_values) - 10):
12     if np.all(extreme_values[i:i+10]):
13         labels[i:i+10] = 1

```

Figure 11: Anomaly detection rules trained from false negative samples of LSTMAD on metric e0770 in KPI dataset.

the e0770 metric from the KPI dataset as an example, where the best baseline model, LSTMAD, has a low recall of 0.33 on the test set. Fig. 10a shows LSTMAD’s output on the test set. There are 3 ground-truth anomalies: two spikes and one prolonged period of extreme values, while LSTMAD detects only the spike around timestamp 5,000, missing the other two anomalies.

We train anomaly detection rules using the false negative samples from LSTMAD on the training set in ARGOS. Fig. 11 shows the rules generated by ARGOS, which consist of two conditions: the first detects sudden spikes or drops, while the second identifies prolonged periods of extreme values. Fig. 10b shows ARGOS’s output on the test set, where the results are aggregated from LSTMAD and the rules using the Aggregator module. Particularly, the second condition helps ARGOS detect a missed anomaly around timestamp 15,000, improving recall from 0.33 to 0.67. It is also worth noting that a false negative still persists around timestamp 12,000 in ARGOS’s output. This could be due to the minor deviation behavior in this sample not being present in the training set, or the threshold for detecting spikes in the rules may require further tuning.

## 6 Related Works

**Time-Series Anomaly Detection.** Prior works on time-series anomaly detection are primarily deep learning-based [38, 49, 51, 56, 59, 63, 67, 68, 70, 75] and rule-based methods [11, 37, 69]. Early deep learning-based models focus on directly predicting the next value based on the observation from current window [38, 51]. With the introduction of encoder-decoder and variational encoder, later models reconstruct the time-series signal by denoising the input and flag anomalies if the observed signal deviates largely from the reconstructed signal [49, 59, 63, 67, 68, 70]. Compared to deep learning-based methods, rule-based methods are more prevalent in industry due to their explainability. Resin [37] traces memory usage

and reports an anomaly if the usage of the current period exceeds mean plus three times the standard deviation of the previous period. Similarly, FBDetect [69] enables users to establish a detection threshold that defines the required magnitude of metric deviation for it to be classified as an anomaly.

**LLMs for Anomaly Detection.** More recently, the emergence of LLMs have led to their applications in anomaly detection. There have been works focusing on leveraging LLMs to perform log-based anomaly detection [15, 20, 29, 36]. For example, NeuralLog [29] transforms log messages into semantic vectors and uses a Transformer-based model to classify anomalies. LogLLM [20] preprocesses logs with regular expressions, employs BERT for semantic extraction, and uses an LLM for sequence classification. Some works have also studied using LLMs for time-series anomaly detection [4, 13, 35]. They either fine-tune the LLMs based on time-series data or directly prompt the LLMs for output labels. To the best of our knowledge, our work is the first to leverage LLMs to generate explainable and reproducible detection rules for time-series anomaly detection.

**LLMs for Cloud Reliability.** Apart from cloud monitoring, LLMs have also been applied in other aspects of cloud reliability from root cause analysis to incident mitigation [2, 8, 17, 23, 62, 65, 73]. RCACopilot [8] is an on-call system for automated cloud incident root cause analysis, combining customizable incident handler workflows and LLM-based root cause prediction to streamline diagnostic data collection. [73] proposes an in-context learning method for root cause analysis using LLMs without fine-tuning, employing historical incidents as examples to inform the model, outperforming fine-tuned models in accuracy and utility. Atlas [65] is a tool leveraging large language models to automate the transformation of unstructured system information into structured causal graphs, enhancing fault localization in cloud systems by constructing high-quality causal representations.

## 7 Conclusion

Production anomaly detection systems require three indispensable properties at the same time: explainability, reproducibility, and autonomy. In this paper, we present ARGOS, a time-series anomaly detection system that autonomously generates rules via LLMs to detect anomalies. By leveraging LLMs' capabilities in time-series understanding and code generation, ARGOS ensures that the rules are both explainable and reproducible. ARGOS comprises an agent-based pipeline that iteratively proposes, repairs, and reviews anomaly detection rules to ensure their quality. Additionally, ARGOS incorporates model fusion for accuracy guarantees and performs top- $k$  rule selection to enhance efficiency. Our evaluation on both public and internal datasets demonstrates that ARGOS outperforms state-of-the-art time-series anomaly detection methods, achieving up to a 28.3% improvement in  $F_1$  score.

## References

- [1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.
- [2] Toufique Ahmed, Supriyo Ghosh, Chetan Bansal, Thomas Zimmermann, Xuchao Zhang, and Saravan Rajmohan. Recommending root-cause and mitigation steps for cloud incidents using large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1737–1749. IEEE, 2023.
- [3] Sarah Alnegheimish, Dongyu Liu, Carles Sala, Laure Berti-Equille, and Kalyan Veeramachaneni. Sintel: A machine learning framework to extract insights from signals. In *Proceedings of the 2022 International Conference on Management of Data, SIGMOD '22*, page 1855–1865, New York, NY, USA, 2022. Association for Computing Machinery.
- [4] Sarah Alnegheimish, Linh Nguyen, Laure Berti-Equille, and Kalyan Veeramachaneni. Large language models can be zero-shot anomaly detectors for time series? *arXiv preprint arXiv:2405.14755*, 2024.
- [5] Dan Ardelean, Amer Diwan, and Chandra Erdman. Performance analysis of cloud applications. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 405–417, Renton, WA, April 2018. USENIX Association.
- [6] Tom B Brown. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.
- [7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- [8] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, Jun Zeng, Supriyo Ghosh, Xuchao Zhang, Chaoyun Zhang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Tianyin Xu. Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys '24*, page 674–688, New York, NY, USA, 2024. Association for Computing Machinery.
- [9] Qian Cheng, Doyen Sahoo, Amrita Saha, Wenzhuo Yang, Chenghao Liu, Gerald Woo, Manpreet Singh, Silvio Saverese, and Steven CH Hoi. Ai for it operations

- (aiops) on cloud platforms: Reviews, opportunities and challenges. *arXiv preprint arXiv:2304.04661*, 2023.
- [10] Ludmila Cherkasova, Kivanc Ozonat, Ningfang Mi, Julie Symons, and Evgenia Smirni. Automated anomaly detection and performance modeling of enterprise applications. *ACM Trans. Comput. Syst.*, 27(3), November 2009.
- [11] Mike Chow, Yang Wang, William Wang, Ayichew Hailu, Rohan Bopardikar, Bin Zhang, Jialiang Qu, David Meisner, Santosh Sonawane, Yunqi Zhang, Rodrigo Paim, Mack Ward, Ivor Huang, Matt McNally, Daniel Hodges, Zoltan Farkas, Caner Gocmen, Elvis Huang, and Chunqiang Tang. ServiceLab: Preventing tiny performance regressions at hyperscale through Pre-Production testing. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 545–562, Santa Clara, CA, July 2024. USENIX Association.
- [12] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, pages 423–435, 2023.
- [13] Manqing Dong, Hao Huang, and Longbing Cao. Can llms serve as time series anomaly detectors? *arXiv preprint arXiv:2408.03475*, 2024.
- [14] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.
- [15] Chris Egersdoerfer, Di Zhang, and Dong Dai. Early exploration of using chatgpt for log-based anomaly detection on parallel file systems logs. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, pages 315–316, 2023.
- [16] Astha Garg, Wenyu Zhang, Jules Samaran, Ramasamy Savitha, and Chuan-Sheng Foo. An evaluation of anomaly detection and diagnosis in multivariate time series. *IEEE Transactions on Neural Networks and Learning Systems*, 33(6):2508–2517, 2022.
- [17] Drishti Goel, Fiza Husain, Aditya Singh, Supriyo Ghosh, Anjaly Parayil, Chetan Bansal, Xuchao Zhang, and Saravan Rajmohan. X-lifecycle learning for cloud incident management using llms. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 417–428, 2024.
- [18] Nate Gruver, Marc Finzi, Shikai Qiu, and Andrew G Wilson. Large language models are zero-shot time series forecasters. *Advances in Neural Information Processing Systems*, 36, 2024.
- [19] Jiawei Tyler Gu, Xudong Sun, Wentao Zhang, Yuxuan Jiang, Chen Wang, Mandana Vaziri, Owolabi Legunsen, and Tianyi Xu. Acto: Automatic end-to-end testing for operation correctness of cloud system management. *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.
- [20] Wei Guan, Jian Cao, Shiyu Qian, and Jianqi Gao. Logllm: Log-based anomaly detection using large language models. 2024.
- [21] Yigong Hu, Gongqi Huang, and Peng Huang. Automated reasoning and detection of specious configuration in large systems with symbolic execution. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 719–734. USENIX Association, November 2020.
- [22] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.
- [23] Yuxuan Jiang, Chaoyun Zhang, Shilin He, Zhihao Yang, Minghua Ma, Si Qin, Yu Kang, Yingnong Dang, Saravan Rajmohan, Qingwei Lin, et al. Xpert: Empowering incident management with query recommendations via large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.
- [24] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, Santa Clara, CA, April 2024. USENIX Association.
- [25] Ming Jin, Shiyu Wang, Lintao Ma, Zhixuan Chu, James Y Zhang, Xiaoming Shi, Pin-Yu Chen, Yuxuan Liang, Yuan-Fang Li, Shirui Pan, et al. Time-llm: Time series forecasting by reprogramming large language models. *arXiv preprint arXiv:2310.01728*, 2023.

- [26] Siwon Kim, Kukjin Choi, Hyun-Soo Choi, Byunghan Lee, and Sungroh Yoon. Towards a rigorous evaluation of time-series anomaly detection. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(7):7194–7201, Jun. 2022.
- [27] Nikolay Laptev, Saeed Amizadeh, and Youssef Billawala. A benchmark dataset for time series anomaly detection. *Yahoo Research*, 2015.
- [28] Nikolay Laptev, Saeed Amizadeh, and Ian Flint. Generic and scalable framework for automated time-series anomaly detection. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, page 1939–1947, New York, NY, USA, 2015. Association for Computing Machinery.
- [29] Van-Hoang Le and Hongyu Zhang. Log-based anomaly detection without log parsing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 492–504. IEEE, 2021.
- [30] Yann LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–50. Springer, 2002.
- [31] Sebastien Levy, Randolph Yao, Youjiang Wu, Yingnong Dang, Peng Huang, Zheng Mu, Pu Zhao, Tarun Ramani, Naga Govindaraju, Xukun Li, Qingwei Lin, Gil Lapid Shafirri, and Murali Chintalapati. Predictive and adaptive failure mitigation to avert production cloud VM interruptions. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1155–1170. USENIX Association, November 2020.
- [32] Ze Li, Qian Cheng, Ken Hsieh, Yingnong Dang, Peng Huang, Pankaj Singh, Xinsheng Yang, Qingwei Lin, Youjiang Wu, Sebastien Levy, and Murali Chintalapati. Gandalf: An intelligent, End-To-End analytics service for safe deployment in Large-Scale cloud infrastructure. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 389–402, Santa Clara, CA, February 2020. USENIX Association.
- [33] Zeyan Li, Nengwen Zhao, Shenglin Zhang, Yongqian Sun, Pengfei Chen, Xidao Wen, Minghua Ma, and Dan Pei. Constructing large-scale real-world benchmark datasets for aiops. *arXiv preprint arXiv:2208.03938*, 2022.
- [34] Zhong Li, Yuxuan Zhu, and Matthijs Van Leeuwen. A survey on explainable anomaly detection. *ACM Trans. Knowl. Discov. Data*, 18(1), September 2023.
- [35] Jun Liu, Chaoyun Zhang, Jiaxu Qian, Minghua Ma, Si Qin, Chetan Bansal, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. Large language models can deliver accurate and interpretable time series anomaly detection. *arXiv preprint arXiv:2405.15370*, 2024.
- [36] Yilun Liu, Shimin Tao, Weibin Meng, Feiyu Yao, Xiaofeng Zhao, and Hao Yang. Logprompt: Prompt engineering towards zero-shot and interpretable log analysis. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pages 364–365, 2024.
- [37] Chang Lou, Cong Chen, Peng Huang, Yingnong Dang, Si Qin, Xinsheng Yang, Xukun Li, Qingwei Lin, and Murali Chintalapati. RESIN: A holistic service for dealing with memory leaks in production cloud infrastructure. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 109–125, Carlsbad, CA, July 2022. USENIX Association.
- [38] Pankaj Malhotra, Lovekesh Vig, Gautam Shroff, and Puneet Agarwal. Long short term memory networks for anomaly detection in time series. 04 2015.
- [39] Microsoft. Azure openai service models - azure openai. <https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/models>. Accessed December 8, 2024.
- [40] Microsoft. Azure openai service – advanced language models. <https://azure.microsoft.com/en-us/products/ai-services/openai-service>. Accessed December 8, 2024.
- [41] Microsoft. Dasv4 size series - azure virtual machines. <https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/general-purpose/dasv4-series>. Accessed December 8, 2024.
- [42] Microsoft. Ndm\_a100\_v4 sizes series - azure virtual machines. <https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/gpu-accelerated/ndma100v4-series>. Accessed December 8, 2024.
- [43] Rodrigo Nogueira, Zhiying Jiang, and Jimmy Lin. Investigating the limitations of transformers with simple arithmetic tasks. *arXiv preprint arXiv:2102.13019*, 2021.
- [44] NVIDIA. How to detect which node causes a nccl hang. <https://github.com/NVIDIA/nccl/issues/968>. Accessed December 4, 2024.
- [45] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. An empirical study of the non-determinism of chatgpt in code generation. *ACM Transactions on Software Engineering and Methodology*, 2024.

- [46] Guansong Pang and Charu Aggarwal. Toward explainable deep anomaly detection. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, KDD '21, page 4056–4057, New York, NY, USA, 2021. Association for Computing Machinery.
- [47] Kashif Rasul, Arjun Ashok, Andrew Robert Williams, Arian Khorasani, George Adamopoulos, Rishika Bhagwatkar, Marin Bilos̃, Hena Ghonia, Nadhir Hassen, Anderson Schneider, et al. Lag-llama: Towards foundation models for time series forecasting. In *R0-FoMo: Robustness of Few-shot and Zero-shot Learning in Large Foundation Models*, 2023.
- [48] Youcef Remil, Anes Bendimerad, Romain Mathonat, and Mehdi Kaytoue. Aiops solutions for incident management: Technical guidelines and a comprehensive literature review. *arXiv preprint arXiv:2404.01363*, 2024.
- [49] Hansheng Ren, Bixiong Xu, Yujing Wang, Chao Yi, Congrui Huang, Xiaoyu Kou, Tony Xing, Mao Yang, Jie Tong, and Qi Zhang. Time-series anomaly detection service at microsoft. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '19, page 3009–3017. ACM, July 2019.
- [50] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.
- [51] Peter J. Rousseeuw and A. Leroy. Robust regression and outlier detection. In *Wiley Series in Probability and Statistics*, 2005.
- [52] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.
- [53] Lukas Ruff, Robert Vandermeulen, Nico Goernitz, Lucas Deecke, Shoaib Ahmed Siddiqui, Alexander Binder, Emmanuel Müller, and Marius Kloft. Deep one-class classification. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4393–4402. PMLR, 10–15 Jul 2018.
- [54] Amazon Web Services. Summary of the aws service event in the northern virginia (us-east-1) region. <https://aws.amazon.com/message/12721/>, December 10 2021. Accessed December 2, 2024.
- [55] Haotian Si, Jianhui Li, Changhua Pei, Hang Cui, Jingwen Yang, Yongqian Sun, Shenglin Zhang, Jingjing Li, Haiming Zhang, Jing Han, et al. Timeseriesbench: An industrial-grade benchmark for time series anomaly detection models. *arXiv preprint arXiv:2402.10802*, 2024.
- [56] Alban Siffer, Pierre-Alain Fouque, Alexandre Termier, and Christine Largouet. Anomaly detection in streams with extreme value theory. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 1067–1075, New York, NY, USA, 2017. Association for Computing Machinery.
- [57] Yifan Song, Guoyin Wang, Sujian Li, and Bill Yuchen Lin. The good, the bad, and the greedy: Evaluation of llms should not ignore non-determinism. *arXiv preprint arXiv:2407.10457*, 2024.
- [58] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, page 3104–3112, Cambridge, MA, USA, 2014. MIT Press.
- [59] Shreshth Tuli, Giuliano Casale, and Nicholas R Jennings. Tranad: Deep transformer networks for anomaly detection in multivariate time series data. *arXiv preprint arXiv:2201.07284*, 2022.
- [60] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.
- [61] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.
- [62] Zefan Wang, Zichuan Liu, Yingying Zhang, Aoxiao Zhong, Jihong Wang, Fengbin Yin, Lunting Fan, Lingfei Wu, and Qingsong Wen. Rcagent: Cloud root cause analysis by autonomous agents with tool-augmented large language models. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, pages 4966–4974, 2024.
- [63] Zexin Wang, Changhua Pei, Minghua Ma, Xin Wang, Zhihan Li, Dan Pei, Saravan Rajmohan, Dongmei Zhang, Qingwei Lin, Haiming Zhang, et al. Revisiting vae for unsupervised time series anomaly detection: A frequency perspective. In *Proceedings of the ACM on Web Conference 2024*, pages 3096–3105, 2024.

- [64] Lingmei Weng, Yigong Hu, Peng Huang, Jason Nieh, and Junfeng Yang. Effective performance issue diagnosis with value-assisted cost profiling. In *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys '23*, page 1–17, New York, NY, USA, 2023. Association for Computing Machinery.
- [65] Zhiqiang Xie, Yujia Zheng, Lizi Ottens, Kun Zhang, Christos Kozyrakis, and Jonathan Mace. Cloud atlas: Efficient fault localization for cloud systems using language models and causal insight. *arXiv preprint arXiv:2407.08694*, 2024.
- [66] 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.
- [67] Haowen Xu, Wenxiao Chen, Nengwen Zhao, Zeyan Li, Jiahao Bu, Zhihan Li, Ying Liu, Youjian Zhao, Dan Pei, Yang Feng, et al. Unsupervised anomaly detection via variational auto-encoder for seasonal kpis in web applications. In *Proceedings of the 2018 world wide web conference*, pages 187–196, 2018.
- [68] Jiehui Xu, Haixu Wu, Jianmin Wang, and Mingsheng Long. Anomaly transformer: Time series anomaly detection with association discrepancy. *arXiv preprint arXiv:2110.02642*, 2021.
- [69] Dong Young Yoon, Yang Wang, Miao Yu, Elvis Huang, Juan Ignacio Jones, Abhinay Kulkadapu, Osman Kocas, Jonathan Wiepert, Kapil Goenka, Sherry Chen, Yanjun Lin, Zhihui Huang, Jocelyn Kong, Michael Chow, and Chunqiang Tang. Fbdetect: Catching tiny performance regressions at hyperscale through in-production monitoring. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP '24*, page 522–540, New York, NY, USA, 2024. Association for Computing Machinery.
- [70] Chaoli Zhang, Tian Zhou, Qingsong Wen, and Liang Sun. Tfad: A decomposition time series anomaly detection architecture with time-frequency analysis. In *Proceedings of the 31st ACM International Conference on Information and Knowledge Management, CIKM '22*, page 2497–2507. ACM, October 2022.
- [71] Chaoyun Zhang, Randolph Yao, Si Qin, Ze Li, Shekhar Agrawal, Binit R. Mishra, Tri Tran, Minghua Ma, Qingwei Lin, Murali Chintalapati, and Dongmei Zhang. Deoxys: A causal inference engine for unhealthy node mitigation in large-scale cloud infrastructure. In *Proceedings of the 2024 ACM Symposium on Cloud Computing, SoCC '24*, page 361–379, New York, NY, USA, 2024. Association for Computing Machinery.
- [72] Lingzhe Zhang, Tong Jia, Mengxi Jia, Yifan Wu, Aiwei Liu, Yong Yang, Zhonghai Wu, Xuming Hu, Philip S Yu, and Ying Li. A survey of aiops for failure management in the era of large language models. *arXiv preprint arXiv:2406.11213*, 2024.
- [73] Xuchao Zhang, Supriyo Ghosh, Chetan Bansal, Rujia Wang, Minghua Ma, Yu Kang, and Saravan Rajmohan. Automated root causing of cloud incidents using in-context learning with gpt-4. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024*, page 266–277, New York, NY, USA, 2024. Association for Computing Machinery.
- [74] Yue Zhao. Towards reproducible, automated, and scalable anomaly detection. In *AAAI Conference on Artificial Intelligence*, 2024.
- [75] Haoyi Zhou, Shanghang Zhang, Jieqi Peng, Shuai Zhang, Jianxin Li, Hui Xiong, and Wancai Zhang. Informer: Beyond efficient transformer for long sequence time-series forecasting. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 11106–11115, 2021.

## Appendices

### A Prompts

#### A.1 Detection Agent

Fig. 12 shows the prompt template used by the Detection Agent in ARGOS. The template begins with a high-level task summary for the Detection Agent, followed by step-by-step instructions on the data format and function invariants. Finally, we provide important notes that we empirically found useful for the Detection Agent to follow to improve the accuracy of anomaly detection rules.

##### Prompt Template for the Detection Agent

**Task Summary:**

You are an AI assistant that helps people write detection rules to determine whether a piece of time series data is abnormal (negative) or not (positive). The time series data is collected during a task for cloud service ...

**Step-by-step Instructions:**

You should achieve the task in the following steps:

1. You will be given the data sample in the following format: ...
2. You should give a Python function `inference(sample: np.ndarray) -> labels: np.ndarray` to write various rules to describe the pattern of given negative/abnormal samples and exclude all given positive/normal samples.
3. The function will take a sample of numpy array with shape  $(X, 2)$  as input, where each row is a tuple of (value, index). You should return the labels as an `np.ndarray` of shape  $(X,)$ , and for each index, `value=1` means the data of the index is abnormal, and `value=0` means the data of the index is normal.
4. Beyond anomalies, you can describe how normal data behave in comments, in the format of "Normal Rule 1 \n Normal Rule 2 ...". Ideally, if the inference function returns no abnormal indices, then the data MUST satisfy all normal rules you describe in comments. You should strictly use the following format: ...

**Important Notes:**

1. Your code should not hard code any information about the label given to you in example data.
2. You should not hard code the indices of anomalies.
3. You should make sure the python code is correct and can be executed without any error.
4. If your code uses any external libraries, you should include the import statements in the code.

Figure 12: A prompt template used by the Detection Agent in ARGOS.

## A.2 Repair Agent

Fig. 13 shows the prompt template used by the Repair Agent in ARGOS. If the code contains syntax errors, the Repair Agent is provided with error messages and the incorrect rule. The Repair Agent suggests a revised version of the rules until all syntax errors are fixed.

### Prompt Template for the Repair Agent

#### Task Summary:

You are an AI assistant that fixes syntax and runtime errors in Python code. You will be given a Python code snippet along with the error message indicating details for syntax and/or runtime errors. You should fix the errors in the code and make sure the code can be executed without any error ...

#### Step-by-step Instructions:

You should achieve the task in the following steps:

1. You will be given the data sample in the following format: ...
2. You are given a python function `inference(sample: np.ndarray) -> labels: np.ndarray` to write various rules to describe and remember the pattern of given negative/abnormal samples and exclude all given positive/normal samples. The function will take a sample of numpy array with shape  $(X, 2)$  as input ...

#### Important Notes:

1. You should output the fixed code following the same format as the input code, wrapping the code with `*** python begin ***` as the first line and `*** python end ***` as the last line. You must only use `*** python begin ***` and `*** python end ***` to wrap your fixed code for only once, don't use them for any other purpose.
2. You should only focus on fixing the errors in the code and make sure the code can be executed without any error. You must not change other logic of the code unrelated to the errors.

Figure 13: A prompt template used by the Repair Agent in ARGOS.

### A.3 Review Agent

Fig. 14 shows the prompt template used by the Review Agent in ARGOS. If the code has accuracy regression compared to the previous iteration or compared to the existing anomaly detection model, the Review Agent is provided with the current code, code difference with the last iteration if exists and performance metrics comparison. Additionally, the Review Agent is also provided with incorrect examples where the current code incorrectly predicts while the previous code or the model correctly predicts. The Review Agent proposes a new set of rules until the performance regression is mitigated.

#### Prompt Template for the Review Agent

##### Task Summary:

You are an AI assistant that reviews Python code changes and propose modifications. You will be given a Python code that contains various anomaly detection rules to describe and remember the pattern of given negative/abnormal samples and exclude all given positive/normal samples. The rules in this Python code will be used in combination with an anomaly detection model that performs anomaly detection. Both the rules and the anomaly detection model will generate anomaly labels, and the performance is a combination of anomaly labels from both sides, comparing against the ground-truth labels ...

##### Step-by-step Instructions:

You should achieve the task in the following steps:

1. You will be given the data sample in the following format: ...
2. You are given a python function `inference(sample: np.ndarray) -> labels: np.ndarray` to write various rules to describe and remember the pattern of given negative/abnormal samples and exclude all given positive/normal samples ...
3. The review process depends on the stage of rule generation.
4. If previous code does not exist, that means we are at the first iteration for the rules. You will be given the current code for the rules. You will also be given the performance metrics of the current code combined with the anomaly detection model and the baseline performance from running only the anomaly detection model ...
5. If previous code exists, that means we are iterating over the rules. You will be additionally given a code difference comparing the current code with the previous code. Our goal is make sure that the performance metric of the current code is better than the previous code.
6. You will be given the performance metrics of the current code and either the baseline performance or performance from the previous code in the following format ...
7. You will be given the code difference in the following format, if previous code exists ...
8. You will also be given incorrect examples that the current code incorrectly predicts while the previous code or the anomaly detection model correctly predicts ...

##### Important Notes:

1. You should output the fixed code following the same format as the input code, wrapping the code with `*** python begin ***` as the first line and `*** python end ***` as the last line. You must only use `*** python begin ***` and `*** python end ***` to wrap your fixed code for only once, don't use them for any other purpose.
2. You should make sure the python code is correct and can be executed without any error.
3. If your code uses any external libraries, you should include the import statements in the code.

Figure 14: A prompt template used by the Review Agent in ARGOS.