AWS cloud watch logs Anomaly Detection

Lohit Aryan Gopikonda*
* University of Maryland, College park
Email: lgopik2@umd.edu

***Abstract*—I present this unsupervised anomaly detection framework method designed specifically for AWS CloudTrail data, which can spot unusual trends brought on by setup errors, security lapses, or performance issues. Temporal, categorical, and statistical variables—such as event timestamps, API call frequencies, response durations, and error codes—are extracted from raw log entries through preprocessing. These features are then supplemented with publically accessible datasets and artificially injected anomalies. In order to simulate typical behavior and instantly identify deviations, we test a variety of detection algorithms, such as Local Outlier Factor and Isolation Forest. Our method achieves low false-alarm rates and good detection accuracy, allowing for quick issue response while reducing manual oversight, according to experimental results on representative AWS workloads. The suggested approach provides a workable answer for existing monitoring pipelines by integrating seamlessly and scaling horizontally.**

***Index Terms*—AWS, Amazon Web services, logs, Machine Learning, Anomaly Detection**

I. Introduction

The accelerated adoption of cloud computing requires service providers to maintain constant up-time for thousands of customers who build their services upon these platforms. As cloud environments rapidly scale in size and complexity, continuous monitoring becomes a critical necessity. This vigilance is essential to proactively detect and address potential failures before they can impact the customers who depend on these vital services. Hence this project aims to build an unsupervised anomaly detection model designed to detect abnormal patterns which might lead to potential security breaches, configurations, or performance issues. Having caught such issues early would lead to improving overall system reliability and reducing manual monitoring efforts

II. Setup and data collection

For the Cloud setup, I setup 4 Elastic Compute cloud (EC2) instances with the ”aws-logs” yum package setup that tracks all resource metrics and system control services logs. The ”aws-logs” yum package installs the Amazon CloudWatch Logs agent. This agent collects and sends log data from the EC2 instances to Amazon CloudWatch Logs. Further details on the setup can be found in the AWS documentation [1].

All logs from each instance were sent to 4 log groups, one for each instance. In order to generate a robust dataset anomalies have to also be generated. For that we need to hone in on the definition of an ”anomaly” pertaining to our research. An ”anomaly” is any log that differs from standard operations, i.e. outliers. Characteristics of these outliers can be found through natural language i.e words like ERROR, access denied, segmentation fault, timeout etc. essentially these anomalies are any events that might affect the security, reliability, or scalability of our cloud infrastructure.

To capture the full spectrum of native cloud anomalies and accurately capture a production level scenario, we use the following methods:

- **Service Failures and Errors:** Stop or kill critical daemons (e.g. `systemctl stop amazon-ssm-agent`), misconfigure SSH keys to trigger authentication failures, or send malformed requests to force kernel panic scenarios.
- **Resource Exhaustion Events:** Use `stress` or `dd if=/dev/zero of=/tmp/bigfile bs=1M` to drive CPU, memory, disk, and I/O usage to capacity, provoking OOM killer events and filesystem full alerts.
- **Network and IMDS Failures:** Bring interfaces down with `ip link set eth0 down`, apply `iptables` rules to block IMDS endpoint (`169.254.169.254`), or introduce latency/loss via `tc qdisc` commands to simulate DHCP timeouts and link flaps.
- **Logging Anomalies:** Flood the syslog with high-rate `logger` invocations or disable log rotation to trigger “excessive logging rate” warnings and dropped-message events.

Using the above setup and anomaly generation I was able to generate two hundred thousand (200,000) logs. These logs were stored within the 4 different log groups. Since AWS Cloudwatch has a view limit of ten thousand (10,000) all 4 log group were exported into a single AWS Simple Storage Service (S3) bucket. Each log group’s logs were stored as a .log file on the bucket. These files were downloaded and concatenated into a single ”ec2.log” file that will be used as our dataset.

Since my anomaly detection leverages semantic analysis of log messages, it naturally generalizes to diverse, previously unseen big-data environments. In fact, model performance improves as I ingest more data,making it especially well-suited to large-scale scenarios. To process these vast log streams efficiently, I adopted Apache Spark via the PySpark API (Application Programming Interface). Spark’s core abstraction, Resilient Distributed Datasets (RDDs), along with its high-level Data Frame and Dataset APIs, allows for parallel in-memory computation across a cluster of nodes. I use Spark SQL for rapid feature extraction and Spark MLlib for scalable machine learning, allowing my anomaly detection pipeline to train and score models on terabytes of data in minutes rather

than hours. Built from the ground up for real-time analytics, Spark provides fault tolerance, dynamic resource allocation, and seamless integration with cloud storage systems, ensuring that my framework remains robust and resilient as data volumes grow.

III. Features and Logs Analysis

Each log entry is parsed to extract the following features for anomaly detection:

- **Timestamp:** The precise date and time at which the event occurred, used for temporal pattern analysis and correlation.
- **Source IP Address:** The IP of the originating host or client, enabling network-level anomaly detection and clustering by origin.
- **Process Name (`proc`):** The daemon or application that generated the log message (e.g. `sshd`, `kernel`, `systemd`), used to segment behavior by service.
- **Message Content:** The raw text of the log entry, which is further processed (tokenization, TF–IDF, keyword flags, embeddings) to extract semantic features.

Each of these features is concatenated into a single feature vector per log entry. By focusing on the message content and the originating process, the model can generalize across unseen message patterns while preserving essential service-specific context. Figure 1 below shows log counts per service and Figure 2 shows log counts that have anomaly keywords like error, timeout, segmentation fault etc.

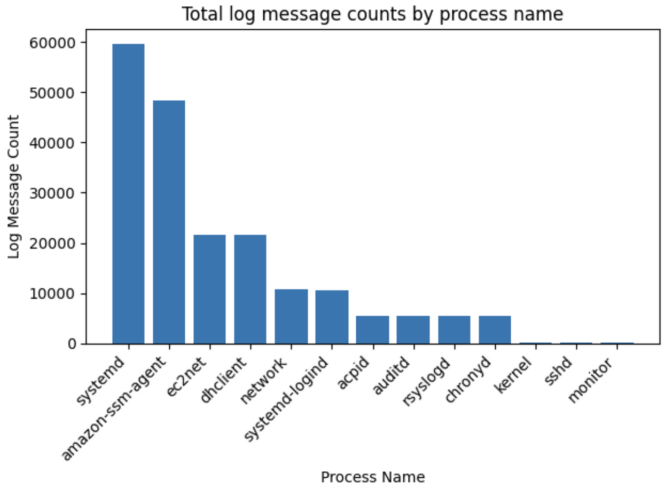

Fig. 1. Total log message counts by process name.

From Figure 1 we can see that `systemd` service has the highest log counts, this should include resource logging and memory trace logging as well since systemd logs all service or worker actions causing the high counts.

`systemd`, as the Linux init system, logs every service lifecycle event—including unit starts, stops, dependency checks, and timer expirations—resulting in continuous output. `amazon-ssm-agent` similarly produces frequent telemetry

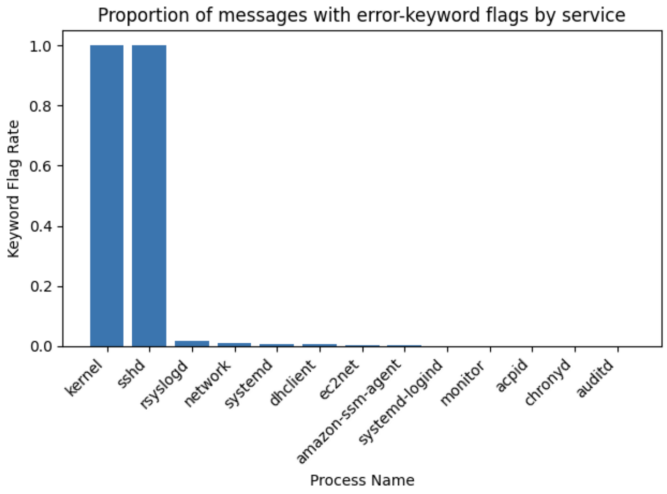

Fig. 2. Proportion of messages with error-keyword flags, by service.

and status updates by polling AWS Systems Manager, sending heartbeats, and reporting command execution results. In contrast, `sshd` only emits entries during SSH connection attempts, authentications, and session teardowns, while the `monitor` script logs resource metrics or threshold breaches at scheduled intervals, leading to significantly lower message counts.

Figure 2 shows the processes that produce the highest error logs as a proportion or ratio of their overall log counts. This helps us understand what processes are producing errors vs normal log entries. The `kernel` and `sshd` process have a ratio of 1.0 i.e 100% error rate since these processes are primarily used to log errors such as panic or fatal exceptions or failed authentication attempts. This indicates that the process name and message are the most crucial features for our anomaly detection model.

IV. Pre Processing

The raw EC2 log data, collected as described in Section II, undergoes a multi-stage pre-processing pipeline implemented using Apache Spark [2] to prepare it for anomaly detection. This ensures efficient and scalable handling of the 200,000 log entries. The primary goal of pre-processing is to transform the unstructured log messages into a structured numerical format suitable for machine learning algorithms, while retaining salient information for distinguishing anomaly entries.

A. Initial Log Parsing and Cleaning

Each log line from the concatenated `ec2.log` file is first parsed using regular expressions to extract key fields. The raw log format generally follows a syslog-like structure. We extract the Zulu timestamp, syslog timestamp, hostname, process name (e.g., `sshd`, `kernel`, `systemd`) with its process ID (PID), and the core log message content. A sample log entry is shown below:

```
2025-04-01T00:00:01.361Z Apr 1 00:00:01 ip-172-31-1-213
    dhclient[4277]: bound to 172.31.165.128 renewal in
    1441 seconds.
```

After initial parsing, entries with empty messages are filtered out. Several fields like the Zulu timestamp, syslog timestamp, hostname, and PID are dropped as the primary focus is on the semantic content of the message and the originating process name for this stage of anomaly detection. The Zulu timestamp is initially converted to an event time for potential future time-series analysis but is not used in the current feature vector for the Isolation Forest model.

Further cleaning of the message content involves removing extraneous information that might introduce noise. This includes stripping leading date, time, and numerical ID patterns that sometimes prefix the core message, as well as removing IP addresses and port numbers using regular expressions, for example, patterns like `'([0-9]+)?3[0-9]+(:[0-9]+|)'`. This step aims to generalize the message content by focusing on textual patterns rather than specific network identifiers.

B. Rule-based Anomaly Labeling (for Evaluation)

Although the primary detection model (Isolation Forest) is unsupervised, to evaluate its performance in identifying known types of problematic events, we introduce a binary label (‘label‘) to the dataset. This label is assigned based on the presence of specific keywords within the log message content. These keywords are indicative of errors, warnings, or critical system states, aligned with the anomaly generation methods described in Section II. The regular expression used for keyword spotting is case-insensitive and targets whole words:

Listing 1. Regular Expression for Anomaly Keyword Spotting

```
(?ix)\b(?:
   ERROR|
   WARN|
   excessive|
   panic|
   OutOfMemory|
   full|
   timeout|
   credential\s+rotation\s+failed|
   failed\s+password\s+for\s+invalid\s+user|
   failed\s+to\s+start|
   filesystem\s+full|
   CPU\s+utilization\s+at\s+\d+(?:\.\d+)?%|
   IMDSv2\s+token\s+request\s+failed|
   Link\s+is\s+down|
   DHCPREQUEST\s+timeout
)\b
```

Log entries matching these keywords are labeled as ‘1‘ (anomaly), and ‘0‘ (normal) otherwise. This labeling strategy serves as a baseline for comparing the unsupervised model’s findings against human-defined anomaly patterns.

C. Feature Engineering Pipeline

The core of the pre-processing for the machine learning model involves transforming the cleaned textual log messages into numerical feature vectors using a sequential Spark ML Pipeline. This pipeline standardizes the feature creation process and is composed of the following stages, applied in order to each log entry’s message content:

1) **Tokenization:** The raw `message` content is first segmented into individual words or tokens. This is achieved using the `RegexTokenizer`, which splits the message string based on a regular expression pattern matching non-alphanumeric characters (`"W+"`). The output is a sequence of these tokens.
2) **Stop Word Removal:** Commonly occurring English words that typically do not convey significant semantic meaning for differentiating log messages (e.g., ”the”, ”is”, ”in”, ”a”) are removed from the token sequences. The `StopWordsRemover` transformer filters these out, resulting in a more concise list of relevant terms.
3) **Term Frequency (TF) Vectorization with Hashing:** The filtered token sequences are converted into numerical feature vectors representing term frequencies using `HashingTF`. This technique applies a hashing function to each term to map it to a predefined feature index within a vector of a fixed size (set to 4096 in our implementation). The value at each index corresponds to the frequency of the term(s) hashed to that index. HashingTF is selected for its computational efficiency and memory scalability, as it avoids the need to build and store a global vocabulary.
4) **Inverse Document Frequency (IDF) Weighting:** The raw term frequency vectors produced by `HashingTF` are then re-weighted using the `IDF (Inverse Document Frequency)` model. IDF assigns higher weights to terms that are rare across the entire corpus of log messages and lower weights to terms that are common. This helps to emphasize terms that are more discriminative. The combination of TF and IDF (TF-IDF) provides a robust measure of term importance.
5) **Dimensionality Reduction with Principal Component Analysis (PCA):** The high-dimensional TF-IDF feature vectors (4096 dimensions) are subsequently reduced to a more manageable and potentially more informative lower-dimensional space using `Principal Component Analysis (PCA)`. PCA identifies the principal components that capture the most variance in the data. We project the TF-IDF vectors onto the top 50 principal components, effectively summarizing the feature space while reducing noise and multicollinearity.
6) **Feature Scaling:** As a final step, the 50-dimensional feature vectors resulting from PCA are standardized using `StandardScaler`. This process scales each feature (principal component) to have zero mean and unit standard deviation. Standardizing features is crucial for algorithms that are sensitive to feature magnitudes or rely on distance calculations, ensuring that all features contribute equitably to the model’s learning process. The output of this stage is a dense vector of scaled features.

The output of this comprehensive pipeline is a Spark DataFrame where each row corresponds to a log entry, now including a column with the final `scaledFeatures` vector alongside the rule-based `label` (as described in Section IV-B). This structured dataset is then used for training and evaluating the anomaly detection models.

V. Models

Following the pre-processing and feature engineering stages detailed in Section IV, the resultant numerical feature vectors are subjected to unsupervised anomaly detection algorithms. This paper primarily investigates two established algorithms: Local Outlier Factor (LOF) and Isolation Forest (IF). These methods were selected for their differing approaches to identifying outliers in data. Initial algorithm exploration was conducted using scikit-learn [3], while the main experimentation involving larger log volumes leveraged a distributed implementation of Isolation Forest provided by the SynapseML library [**?**] for Apache Spark, aligning with our goal of scalable log analysis.

A. Local Outlier Factor (LOF)

Local Outlier Factor (LOF) is a density-based algorithm renowned for its ability to identify anomaly data points by comparing their local density with that of their immediate neighbors [4]. The underlying principle is that an outlier will be situated in a region of significantly lower local density compared to its surrounding data points, whereas an inlier will share a comparable local density with its neighbors.

The LOF score for a data point p is derived through a series of conceptual steps:

1) **Identifying the Neighborhood:** For each point p, the algorithm first determines its k closest neighbors (the $N_k(p)$ set) and the distance to the k-th farthest of these neighbors (the k-distance). The choice of k defines the scale of the "locality."
2) **Assessing Reachability:** The "reachability distance" from point p to a neighbor o is a smoothed version of the actual distance. It's defined as the greater of the true distance between p and o, or the k-distance of o. This prevents points in very dense clusters from having disproportionately small reachability distances, which could skew density estimates.
3) **Calculating Local Density:** The "local reachability density" (lrd) of point p quantifies how densely packed its neighborhood is. It is essentially the inverse of the average reachability distance from p to its k neighbors. A higher lrd value signifies that p is in a denser region.
4) **Determining the Outlier Factor:** Finally, the LOF score of point p is calculated. This score is the average ratio of the local reachability density of p's neighbors to its own local reachability density.

A LOF score close to 1 indicates that the point p resides in a region with a density similar to that of its neighbors, suggesting it is an inlier. Conversely, a score significantly greater than 1 implies that p is in a sparser region than its neighbors, thus classifying it as a local outlier. The parameter k is a critical hyperparameter that influences the algorithm's sensitivity to local variations in density. A conceptual diagram illustrating the LOF mechanism is presented in Figure 3.

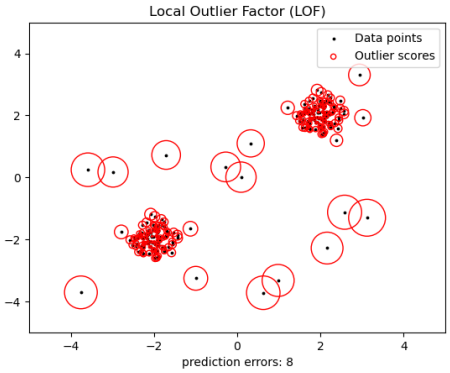

Fig. 3. Conceptual illustration of Local Outlier Factor.

B. Isolation Forest (IF)

Isolation Forest (IF) is an ensemble-based anomaly detection algorithm particularly effective for high-dimensional datasets [5], [6]. It operates on the foundational principle that anomalies are "few and different," which makes them more susceptible to being isolated via random partitioning of the feature space compared to normal, more common data points.

The IF algorithm constructs an ensemble of "Isolation Trees" (iTrees). The process for building each iTree is as follows:

1) A random subsample of the input data is selected for constructing the tree. This helps in creating diverse trees within the forest.
2) The data is recursively partitioned by randomly selecting a feature and then randomly choosing a split value for that feature, typically between the minimum and maximum observed values for that feature in the current subset of data at a node.
3) This partitioning process continues until each data point is isolated in its own leaf node, or until a predefined maximum tree height is reached.

Due to their distinct characteristics, anomaly data points are expected to be isolated closer to the root of the iTrees, generally requiring fewer partitions. This results in shorter average path lengths from the root to the terminal leaf node for anomalies across the ensemble of trees. Conversely, normal data points, which typically reside in denser and more homogeneous regions of the feature space, require more partitions to be isolated, and thus exhibit longer average path lengths. Figure 4 provides a schematic representation of this isolation mechanism.

The degree of anomaly for a data point x is quantified by an anomaly score. This score is derived from the average path length, $E[h(x)]$, of x across all iTrees in the forest. Specifically, a shorter average path length $E[h(x)]$ relative to a normalization factor $c(n)$ (which represents the expected path length in a similar-sized random tree) indicates a higher anomaly score.

An anomaly score approaching 1 suggests a high likelihood that x is an anomaly (indicative of a consistently short average path length). Conversely, an anomaly score substantially less than 0.5 suggests x is a normal instance (indicative of a long average path length). Scores around the 0.5 mark are generally considered inconclusive. In practical applications, such as the SynapseML implementation utilized in this work, a "contamination" parameter is often specified. This parameter guides the algorithm in setting an internal threshold on these anomaly scores to perform a binary classification of data points as normal or anomaly.

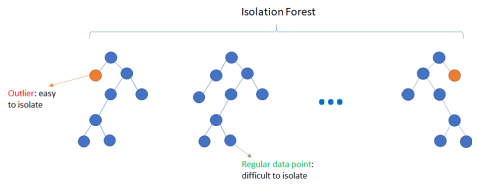

Fig. 4. Conceptual illustration of an Isolation Tree. anomaly instances (e.g., star marker) are typically isolated with fewer random partitions, resulting in shorter path lengths compared to normal instances (e.g., circle markers).

Isolation Forest offers advantages such as computational efficiency, scalability to high-dimensional spaces, and robustness to irrelevant features, as it does not rely on distance computations.

VI. Results and Discussion

The performance of the Isolation Forest and Local Outlier Factor (LOF) models was evaluated on the pre-processed AWS CloudWatch log dataset, using the rule-based anomaly labels generated as described in Section IV-B. The evaluation focused on standard classification metrics: True Positives (TP), False Positives (FP), False Negatives (FN), True Negatives (TN), Precision, Recall, and F1-Score. These metrics provide insights into the models' effectiveness in distinguishing between normal and anomaly log entries.

The consolidated results for both algorithms are presented in Table I.

TABLE I
Performance Metrics for Anomaly Detection Models

Metric	Isolation Forest	Local Outlier Factor
True Positives (TP)	116	159
False Positives (FP)	1831	1788
False Negatives (FN)	845	802
True Negatives (TN)	191894	191937
Precision	0.060	0.082
Recall	0.121	0.165
F1-Score	0.080	0.109

From Table I, it is evident that both models face challenges in accurately identifying anomalies within the dataset, as reflected by the relatively low Precision, Recall, and F1-Scores.

The **Isolation Forest** model identified 116 true anomalies (TP) but also incorrectly flagged 1831 normal log entries as anomaly (FP). This resulted in a very low Precision of 0.060, indicating that a large majority of the alerts generated by this model would be false alarms. Furthermore, the model missed 845 actual anomalies (FN), leading to a Recall of 0.121. This suggests that the model fails to detect approximately 87.9% of the true anomalies. The F1-Score, which balances Precision and Recall, is consequently low at 0.080. The high number of True Negatives (191894) shows the model is effective at correctly identifying normal instances, which is expected given the highly imbalanced nature of anomaly detection datasets where anomalies are rare.

The **Local Outlier Factor (LOF)** It identified 159 true anomalies (TP) and had a slightly lower number of false positives (1788 FP) than the Isolation Forest. This contributed to a higher Precision of 0.082. The Recall for LOF was 0.165, as it missed 802 true anomalies (FN), indicating it detected a larger fraction of actual anomalies than the Isolation Forest. The F1-Score for LOF was 0.109.

Discussion: While the metrics presented in Table I for this specific experimental run show Local Outlier Factor (LOF) with slightly higher scores than Isolation Forest, both models demonstrated broadly similar challenges in achieving high performance, with low absolute values for precision, recall, and F1-scores. Importantly, across multiple experimental runs (not detailed in Table I but observed during experimentation), Isolation Forest exhibited more stable results, particularly when its contamination hyperparameter was appropriately set. This stability across various data shuffles or minor variations can be a crucial practical advantage.

The high number of False Positives (FP) for both models is a critical concern. In a production environment, a high FP rate translates to a high volume of false alarms, which can lead to alert fatigue among system administrators and potentially cause genuine alerts to be overlooked. For instance, the Isolation Forest model generated 1831 false alarms for only 116 correctly identified anomalies, and LOF generated 1788 false alarms for 159 correctly identified anomalies.

Simultaneously, the substantial number of False Negatives (FN) indicates that a significant proportion of true anomalies are not being detected. For security or critical performance issues, such missed detections could have severe consequences. Given that the total number of rule-defined anomalies in the evaluation set was 961, the detection rates are low.

These results suggest that while unsupervised methods like Isolation Forest and LOF can provide a baseline for anomaly detection in AWS CloudWatch logs, their performance with the current feature engineering and model configurations is limited. The complexity of log data, the subtlety of certain anomalies, and the inherent difficulty of unsupervised learning in highly imbalanced datasets are likely contributing factors. The definition of an anomaly, currently based on keywords (Section IV-B), might also influence these perceived results; the models might be identifying patterns that do not perfectly align with these keyword-based definitions but could still be considered anomaly from a different perspective.

Further investigation into feature engineering, hyperparam-

eter tuning, the use of more sophisticated unsupervised or semi-supervised techniques, or incorporating domain-specific knowledge more deeply into the models might be necessary to achieve the desired "low false-alarm rates and good detection accuracy" mentioned in the abstract for production level deployment. The current performance levels highlight the challenges in developing a robust and reliable anomaly detection system for complex log data.

VII. Conclusion and Future Work

This paper introduced an unsupervised framework for detecting anomalies in AWS CloudWatch logs, leveraging Apache Spark for scalable pre-processing and evaluating Isolation Forest and Local Outlier Factor models. Our findings indicate that while the developed framework provides a scalable basis for log analysis, both algorithms exhibited limited performance in accurately identifying anomalies with the current feature engineering and configurations. The high rates of false positives and false negatives were prevalent for both.

While individual test runs showed minor variations in performance metrics between the models, observations across multiple iterations suggested that Isolation Forest provided more consistent and stable detection behavior, a valuable characteristic for operational deployment.

These results highlight the inherent complexities of unsupervised anomaly detection in diverse log data and underscore that significant improvements are needed to achieve the reliability required for production environments.

The study opens several avenues for future work aimed at enhancing the efficacy of the anomaly detection system:

- **Enhanced Feature Engineering and Representation:**
  - Explore advanced text representation techniques, including the use of different encoders such as Word2Vec or other contextual embeddings (e.g., BERT), to capture richer semantic meaning from log messages.
  - Investigate the incorporation of more sophisticated temporal features and sequential patterns within log data.
- **Advanced Anomaly Detection Models and Techniques:**
  - Incorporate and evaluate deep learning models, particularly Autoencoders, which are well-suited for learning normal data representations and identifying deviations. Long Short-Term Memory (LSTM) networks could also be explored for sequence-based log anomaly detection.
  - Investigate other unsupervised algorithms and ensemble methods that combine the strengths of multiple detectors.
- **Rigorous Model Optimization and Validation:**
  - Conduct extensive hyperparameter tuning for all models investigated. This systematic optimization is crucial for maximizing the performance of each algorithm on the specific dataset.
  - Adapt and apply validation strategies, such as K-fold cross-validation, to assess model stability and generalization. For unsupervised settings, this might involve evaluating the consistency of outlier scores or using internal validation metrics across different data subsets.

Addressing these areas will be critical in advancing the capabilities of the anomaly detection framework, aiming for a more robust and accurate system for maintaining the security and reliability of cloud infrastructures.

References

[1] Amazon Web Services, "Quick start: Install and configure the cloudwatch logs agent on a running ec2 linux instance," https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/QuickStartEC2Instance.html, 2024.

[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud '10)*. Boston, MA: USENIX Association, 2010, pp. 10–10.

[3] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[4] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "Lof: Identifying density-based local outliers," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*. Dallas, Texas, USA: ACM, 2000, pp. 93–104.

[5] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest," in *Proceedings of the 8th IEEE International Conference on Data Mining (ICDM '08)*. Pisa, Italy: IEEE Computer Society, 2008, pp. 413–422.

[6] ——, "Isolation-based anomaly detection," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 6, no. 1, pp. 3:1–3:39, 2012.