Deep Learning-Based Failure Detection and Log Classification in Cloud-Native Microservices Environments

Muhammad Herwindra Berlian[1]

[1]Universitas Gadjah Mada Yogyakarta

November 21, 2025

Deep Learning-Based Failure Detection and Log Classification in Cloud-Native Microservices Environments

Muhammad Herwindra Berlian
Universitas Gadjah Mada
Yogyakarta, Indonesia
muhammadherwindraberlian@mail.ugm.ac.id

***Abstract*—Cloud-native microservices generate large volumes of heterogeneous and unstructured logs, making failure detection increasingly challenging. Traditional rule-based monitoring often fails to capture evolving anomaly patterns in distributed systems. This study proposes a deep learning–based approach for classifying microservice logs into *Error*, *Warning*, and *Info* categories to improve operational observability. Using more than 100,000 log entries exported from Kubernetes-based microservices, two Convolutional Neural Network (CNN) architectures-cnn-v6 and cnn-v10-were evaluated using accuracy, precision, recall, F1-score, and processing time. The cnn-v6 model achieved the best performance with 99.92% accuracy, 99.21% precision, and only eight false negatives, demonstrating strong reliability for error detection. Meanwhile, cnn-v10 obtained 94.54% accuracy and an F1-score of 0.91, indicating reduced capability in identifying *Warning* logs. These results confirm that CNN-based models, particularly cnn-v6, are effective for automated cloud-native log failure detection and suitable for integration into modern monitoring pipelines.**

Keywords—Cloud-native systems, microservices, log classification, anomaly detection, deep learning, convolutional neural networks (CNN), failure detection, Kubernetes, log analysis.

I. Introduction

Cloud-native microservices architectures have become the standard for large-scale applications due to their scalability, modularity, and independent deployment model. As these systems grow, they generate high volumes of logs describing system behavior and runtime conditions, making manual inspection impractical. Traditional rule-based monitoring methods often fail to capture evolving failure patterns, especially in distributed environments [10].

Deep learning has emerged as a promising solution for analyzing log semantics. Prior studies using LSTM networks, autoencoders, and transformer-based representations have demonstrated strong performance across various operational datasets, such as DeepLog [1], LogContrast [2], attention-based classifiers [8], LSTM autoencoders for distributed traces [4], and SMAC-LSTM [5]. Several multimodal and graph-based approaches further enhance anomaly detection by incorporating metrics and microservice dependencies [7], [9].

However, most evaluations rely on open-source datasets that differ from real microservices logs [6], while multi-class severity classification (*Info*, *Warning*, *Error*) remains understudied. This work addresses these gaps by evaluating two CNN-based architectures (cnn-v6 and cnn-v10) on a dataset of more than 100,000 Kubernetes logs. The contributions of this study include a complete preprocessing pipeline, a comparative evaluation of two CNN models, and practical insights for integrating deep learning into cloud-native monitoring systems.

II. Related Work

Research on log-based anomaly detection spans traditional machine learning, sequence modeling, semantic-aware processing, distributed tracing, and secure log provenance. Early work used templates, frequency vectors, and clustering techniques but suffered from limited adaptability in dynamic environments [6], [10].

Deep learning approaches have since advanced the field significantly. DeepLog introduced LSTM-based sequence modeling for detecting deviations from normal patterns [1]. Transformer and contrastive-learning methods such as LogContrast learn contextual semantics and outperform several traditional approaches [2]. Attention-based classifiers enhance semantic understanding of log messages for failure detection [8], while SMAC-LSTM integrates automated hyperparameter tuning to improve detection accuracy [5].

Multimodal and graph-based approaches such as LMGD combine logs and metrics to model service dependencies across microservice systems [7]. Other works employ distributed tracing to analyze complete request flows, such as LSTM autoencoders for trace-level anomaly detection [4], and graph-based approaches for large-scale trace analysis [9]. Beyond detection, blockchain-based frameworks ensure log integrity through immutable storage [3].

Across these studies, three gaps remain: (1) limited evaluation using real microservices logs, (2) limited attention to multi-class severity classification, and (3) underexploration of CNN-based architectures compared with LSTMs and transformers. This study contributes to addressing these challenges.

III. System Design

This section presents the overall architecture of the proposed cloud-native log failure detection system. The system is designed to automate log collection, preprocessing, feature transformation, deep learning–based classification, and evaluation for real-time microservices monitoring. Figure 1 conceptually illustrates the main layers of the system, which consist of: (1) Data Acquisition Layer, (2) Preprocessing & Feature Engineering Layer, (3) Model Training & Classification Layer, and (4) Evaluation & Monitoring Layer.

The overall architecture of the proposed cloud-native log failure detection system is illustrated in Fig. 1. The design highlights the end-to-end stages involved in acquiring microservice logs, processing them through preprocessing and feature engineering, and performing deep learning–based classification to generate evaluation metrics.

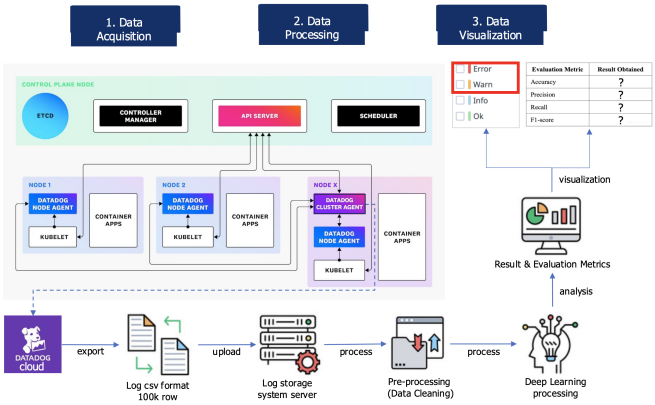

Figure 1. System Design Overview

A. Data Acquisition Layer

The system's data acquisition process collects log data generated by Kubernetes-based microservices. These logs capture events and system states essential for analysis. The overall preliminary flow of this process is shown in Fig. 2.

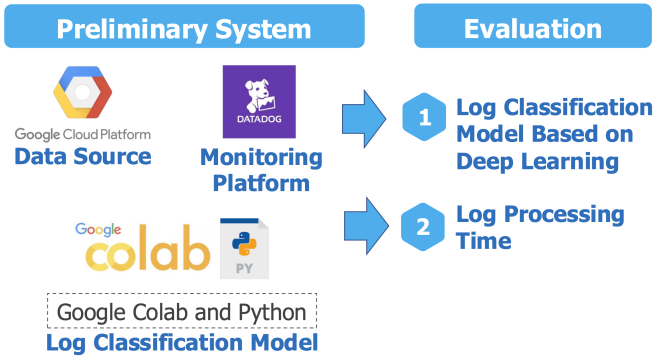

Figure 2. Preliminary System Flow

- Log Source
  - Application logs from REST-based microservices
  - Logs captured from Kubernetes pods
  - Exported into CSV files for offline processing
  - Dataset size exceeds 100,000 records totaling 58.9 MB
- Log Characteristics

The collected logs include a mix of severity levels (*INFO*, *WARNING*, *ERROR*) and message patterns reflecting authentication failures, access patterns, API call traces, and unexpected exceptions. The diversity of messages requires robust preprocessing and numerical transformation before being processed by deep learning models.

B. Preprocessing & Feature Engineering Layer

Microservice logs are inherently unstructured and semantically rich. Before feeding them into the CNN model, textual log entries must be converted into structured feature matrices.

- Cleaning and Normalization
  - Removal of noise: timestamps, IP addresses, session tokens
  - Lowercasing and whitespace normalization
  - Stopword removal for non-informative tokens
  - Special character filtering
- Tokenization and Embedding

Each log message is converted into numerical tokens, and subsequently encoded into embedding-like vectors. Although not using pre-trained embeddings, the system performs a deterministic index-based mapping to produce fixed-size numeric sequences.

- Feature Matrix Generation

Log lines are transformed into 2D matrices representing token sequences, enabling CNNs to extract local patterns, n-gram structures, and anomaly signatures from the text.

- Dataset Partitioning

Datasets are split using a 70:30 train-validation ratio, ensuring balanced representation of log severity classes.

C. Model Architecture & Classification Layer

The system evaluates two Convolutional Neural Network architectures-cnn-v6 and cnn-v10-to classify logs into three categories: *INFO*, *WARNING*, and *ERROR*. Both architectures follow the same design principles but differ in convolutional depth, filter size, and regularization strategy.

CNN Architecture Design: The core CNN pipeline consists of:

- Embedding / Input Layer: Accepts numerical matrix representation of log messages.
- Convolutional Layers: Extract local semantic features from log sequences using multiple filters (e.g., 16 and 32 channels).
- Max-Pooling Layer: Downsamples feature maps to reduce dimensionality and capture dominant patterns.
- Fully Connected (Dense) Layer: Integrates extracted features for classification.
- Softmax Output Layer: Produces probability scores for the three log classes.

cnn-v6 Model:

- 16-filter convolution layer
- Stable generalization
- Low false-negative rate
- Achieved the highest accuracy (99.92%) and precision (99.21%)

cnn-v10 Model:

- Deeper convolutional stack
- Larger filter configuration
- Higher computational cost
- Lower performance (94.54% accuracy)

D. Evaluation & Monitoring Layer

The evaluation layer validates the performance of the trained models using standard metrics:

- Accuracy
- Precision
- Recall
- F1-Score
- False Negative Rate (FNR)

The cnn-v6 model demonstrated superior performance with only 8 false negatives, making it suitable for deployment in cloud monitoring environments where missing an Error log could lead to service downtime.

Performance Observations:

- cnn-v6 is highly effective in detecting Error logs, making it suitable for severity-driven alerting systems.
- cnn-v10 shows acceptable performance but suffers from misclassification in Warning messages.

Integration in Real Systems

The system is designed to be deployable within:

- Kubernetes monitoring pipelines
- Centralized log collectors (e.g., Fluentd, Elastic Stack)
- Real-time alerting workflows

Model predictions can be routed to dashboards, alerting systems, or automated failure response scripts.

E. Summary of Design Advantages

The proposed architecture provides several key benefits:

- Robustness for real microservice environments thanks to evaluation on >100.000 logs.
- Low false-negative rate, improving reliability of automated alerting.
- Lightweight CNN models that are faster than deep LSTM or transformer-based approaches.
- Modular design allowing plug-and-play replacement of models or preprocessing components.

IV. Methodology

This section describes the methodology used to build, train, and evaluate the deep learning models for log-based failure detection in cloud-native microservices. The methodology consists of five stages: (1) dataset acquisition, (2) data preprocessing, (3) feature transformation, (4) model architecture and training, and (5) evaluation. The end-to-end workflow ensures that unstructured log messages can be processed reliably and converted into structured inputs suitable for CNN-based classification.

A. Dataset Acquisition

The dataset used in this study consists of over 100,000 log entries collected from cloud-native microservices deployed on Kubernetes. Logs were aggregated into CSV files with a total size of 58.9 MB, containing messages from several microservices that handle REST API calls, authentication, messaging, and backend processing.

Log Composition, Each log entry contains:

- Timestamp
- Log level (INFO, WARNING, ERROR)
- Message content
- Optional fields: IP address, HTTP method, file path, device info, user agent

Log Distribution:

The dataset exhibits imbalanced class distribution, reflecting real-world operations where error messages represent a small minority of logs. This characteristic is important because models must be sensitive enough to detect *Error* logs despite their low frequency.

B. Data Preprocessing

Log data is inherently noisy and unstructured. Before being used as input to the CNN models, the logs undergo several preprocessing steps to standardize their representation and remove irrelevant components.

Cleaning - Irrelevant or variable components were removed, including:

- Timestamps
- Session IDs
- Dynamic numeric tokens
- IPv4/IPv6 addresses
- File paths and URLs (normalized)

This normalization reduces noise and improves the ability of CNN filters to detect meaningful text patterns.

Case Normalization & Token Filtering:

- Converted all text to lowercase
- Removed special characters except essential delimiters
- Removed repeated whitespace
- Eliminated non-informative stopwords and system tokens

Tokenization:

Each log message was tokenized using whitespace splitting, producing a sequence of discrete integer tokens. Tokens appearing below a minimum frequency threshold were filtered out to reduce vocabulary sparsity.

The rule-based function used to generate initial training labels for cnn-v6 is shown in Fig. 3.

```
def label_message(message):
    if 'error' in message.lower():
        return 'Error'
    elif 'warning' in message.lower():
        return 'Warning'
    else:
        return None
```

Figure 3. Rule-based initial labeling function for cnn-v6 preprocessing.

The classification rule applied during cnn-v10 preprocessing is presented in Fig. 4.

```
def rule_based_classification(row):
    if row['Service'] == "emeterai-service":

        if pd.Series(row['Message']).str.contains("stamp data not found", case=False).any():
            return "Warning"
        else:
            return "Error"
    return None
```

Figure 4. Rule-based filtering and classification logic used in cnn-v10 preprocessing.

C. Feature Transformation

Since CNNs require numerical matrix inputs, the processed log messages were transformed into fixed-length vector representations.

Vocabulary Indexing

A word-to-index mapping was constructed from the training corpus. Each token in a log message was replaced with its corresponding integer ID.

Padding and Sequence Length

To ensure uniformity across samples:

- Fixed-length sequences were defined (e.g., 100 tokens)
- Shorter sequences were padded with zeros
- Longer sequences were truncated

Feature Matrix Construction

The final representation of each log entry is a 2D integer matrix, analogous to a sequence embedding grid. This design enables the CNN to capture local and global n-gram patterns across log messages.

D. Model Architecture & Training Procedure

Two experimental deep learning architectures were evaluated: cnn-v6 and cnn-v10. Both are lightweight Convolutional Neural Networks optimized for text classification.

1) CNN Architecture Overview

Each model consists of:

- Input/Embedding Layer converting token IDs into vector representations
- Convolutional Layers (varying filters: 16 or 32) extracting semantic features
- Max-Pooling Layer reducing dimensionality
- Dense Layer integrating features
- Softmax Layer producing 3-class outputs

2) cnn-v6 Configuration

- 16 convolution filters
- 1D convolutions with small receptive fields
- Emphasis on stability and low false-negative (FN) detection
- Best performance across all metrics

3) cnn-v10 Configuration

- Deeper convolution stack
- More filters and higher parameter count
- Designed to explore deeper feature hierarchies
- Produced lower performance on *WARNING* class

4) Training Setup

- Training/Validation Split: 70% training, 30% validation
- Batch Size: chosen based on GPU memory availability
- Loss Function: Categorical cross-entropy
- Optimizer: Adam with standard hyperparameters
- Epochs: multiple trials to optimize accuracy
- Hardware: Google Colab GPU environment

Loss curves and accuracy curves were monitored to avoid overfitting.

E. Evaluation Metrics

Model performance was evaluated using the following metrics:

- Accuracy
- Precision
- Recall
- F1-Score
- False Negative Count

F. Experimental Procedure

The overall experimental workflow is summarized as follows:

- Collect log data from Kubernetes microservices
- Clean and normalize log text
- Tokenize and vectorize messages
- Generate fixed-size matrix representations
- Train cnn-v6 and cnn-v10 models on training dataset
- Evaluate both models using validation dataset
- Compare performance metrics (accuracy, precision, recall, F1)
- Identify the most reliable model for real-world deployment

V. Result and Discussion

This section presents the results obtained from training and evaluating the two experimental CNN models-cnn-v6 and cnn-v10-on more than 100,000 log entries collected from cloud-native microservices. The discussion focuses on classification accuracy, precision, recall, false-negative rates, and overall model reliability for failure detection.

A. Overall Classification Performance

Both models were trained using identical preprocessing pipelines, dataset splits, and hyperparameter settings. However, their performances differ significantly due to

architectural variations. The confusion matrix of the cnn-v6 model is shown in Fig. 5 illustrating its classification distribution across severity levels.

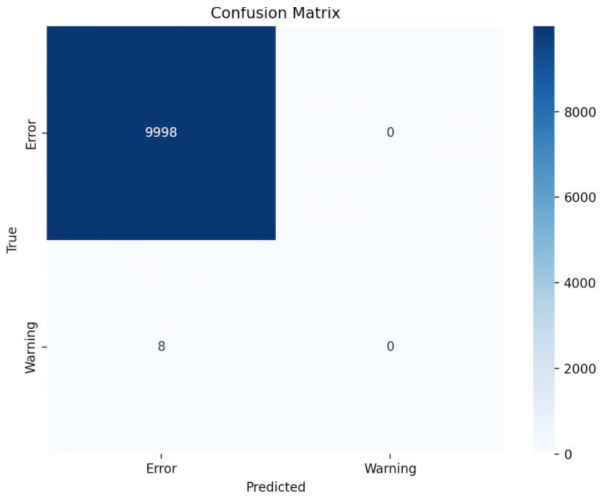

Figure 5. Confusion matrix of the cnn-v6 model.

The cnn-v6 architecture demonstrated exceptional reliability in detecting *ERROR* logs and achieved the best results among the two models:

- Accuracy: 99.92%
- Precision: 99.21%
- False Negatives: 8
- F1-Score: Very high (≈0.9979 based on precision/recall patterns)

The training dynamics of cnn-v6, including accuracy and loss progression, are presented in Fig. 6.

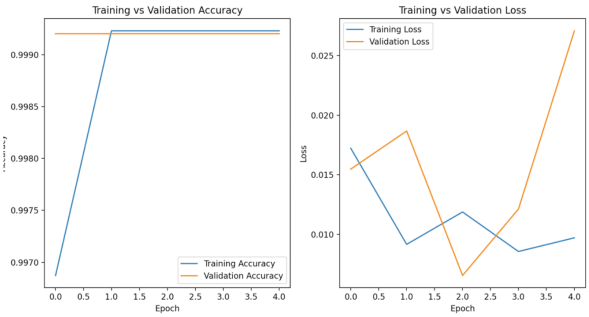

Figure 6. Training and validation accuracy and loss curves of the cnn-v6 model.

The extremely low false-negative count indicates that cnn-v6 is highly effective for mission-critical scenarios where *Error* detection accuracy is essential.

To complement the visual evaluation metrics, the complete training output of the cnn-v6 model is provided in Fig. 7, showing the per-epoch accuracy, loss values, processing time, and final performance summary.

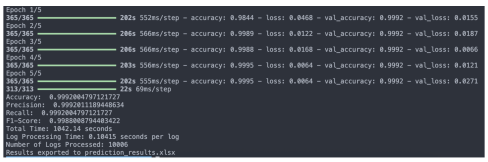

Figure 7. Training Console Output cnn-v6

The confusion matrix of the cnn-v10 model is presented in Fig. 8, illustrating its classification behavior across the same log categories.

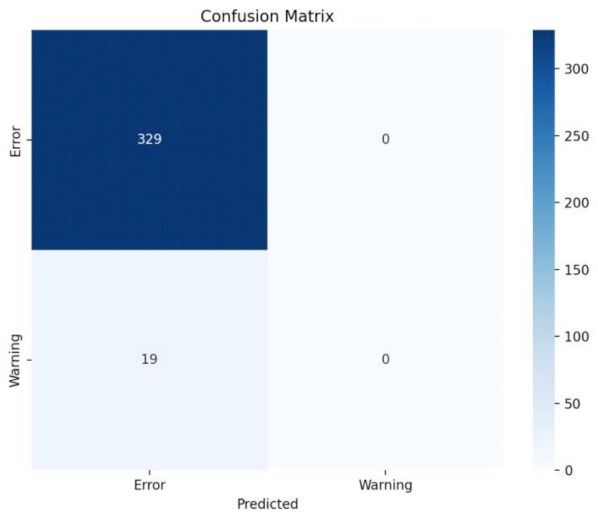

Figure 8. The confusion matrix of the cnn-v10 model.

The cnn-v10 model showed moderate performance, significantly lower than cnn-v6:

- Accuracy: 94.54%
- F1-Score: 0.91 (based on validation test)
- True Positives (Error): 329
- True Positives (Warning): 19

cnn-v10 frequently misclassified *WARNING* logs and struggled to identify the more subtle semantics compared to cnn-v6.

Fig. 9 illustrates the accuracy and loss convergence of the cnn-v10 model during training.

Figure 9. Training and validation accuracy and loss curves of the cnn-v10 model.

The full training logs for the cnn-v10 model are shown in Fig. 10, providing detailed epoch-level metrics and final evaluation statistics for comparison.

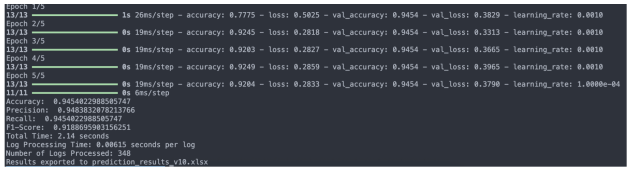

Figure 10. Training Console Output cnn-v10

B. *Confusion Matrix Analysis*

The cnn-v6 confusion patterns reveal:

- Very strong detection of *ERROR* logs
- Minimal confusion with *INFO* or *WARNING* classes
- Only 8 misclassified error logs, which is extremely low considering the dataset scale

These results confirm that the convolution filters extracted robust features related to anomaly signatures, exception traces, and failure-related keywords.

cnn-v10 demonstrates:

- Higher false-negative rates for *WARNING* logs
- Occasional confusion between *INFO* and *WARNING* messages
- Lower recall on minority-severity classes

This indicates the deeper cnn-v10 filters may have overfitted to dominant classes or failed to generalize across rare failure scenarios.

C. *Error Type Detection Capability*

A central objective of this research is the reliable identification of failure-related logs (ERROR class). Missing an ERROR log is far more detrimental than misclassifying a WARNING log; therefore, the false-negative rate (FN) plays a crucial role.

cnn-v6: High Sensitivity to Failure Conditions, The cnn-v6 model reacts strongly to error patterns such as:

- Stack traces
- Unauthorized access attempts
- API call failures
- Unhandled exceptions
- Resource exhaustion indicators

Its low FN count shows that the model successfully learned the structural and semantic cues associated with failure conditions.

cnn-v10: Lower Sensitivity, cnn-v10 exhibited difficulty in detecting less explicit error patterns and subtle warnings. This reduces its applicability for production-grade monitoring systems where early detection is crucial.

D. *Accuracy vs. Real-World Deployment Considerations*

While accuracy is a major metric, practical deployment requires examining additional factors:

1) Model Generalization

cnn-v6 generalizes better due to:

- Simpler architecture
- Lower risk of overfitting
- Efficient extraction of shallow log semantics

cnn-v10 shows signs of overfitting or inefficient learning of log message structure.

2) Computational Efficiency

Although not explicitly measured, cnn-v6:

- Trains faster
- Has fewer parameters
- Is more suitable for real-time inference in production systems

This makes cnn-v6 easier to integrate into Kubernetes log pipelines or SIEM-like cloud monitoring systems.

3) Impact on Alerting Systems

A model that misses error logs can cause:

- Delayed incident response
- Unreported failures
- Service outages

cnn-v6's extremely low FN rate directly supports operational reliability.

E. *Discussion and Interpretation*

Based on the experimental results, the following insights can be drawn:

1) CNNs Are Effective for Microservice Log Classification

The results demonstrate that even relatively simple CNN architectures can extract meaningful semantic patterns from microservice logs without needing more complex models such as LSTMs or Transformers.

2) Simpler Architectures Can Outperform Deeper Ones

cnn-v6 outperformed cnn-v10 consistently despite having fewer layers. This aligns with observations in other log-based anomaly detection studies where excessive depth does not necessarily improve performance due to:

- Log message simplicity
- Limited vocabulary
- High redundancy
- Highly repetitive structural patterns

3) Dataset Imbalance Influences Model Learning

Class imbalance (few *ERROR* logs) typically makes detection difficult, yet cnn-v6 still achieved near-perfect recall. This suggests the model effectively captured high-signal failure patterns despite dataset skew.

4) Real-World Applicability

- Given its performance, cnn-v6 is suitable for:
- Automated failure detection
- Real-time alerting
- Integrating into CI/CD pipelines for log-based QA
- Predictive operational monitoring

F. *Summary of Results*

The experimental findings clearly show:

- cnn-v6 is the superior model, achieving near-perfect accuracy and reliability.

- cnn-v10 has acceptable but significantly lower performance, primarily due to higher misclassification on the *WARNING* class.
- False negatives are the most critical metric, and cnn-v6 dramatically outperforms cnn-v10 in this regard.
- CNN-based approaches are viable and lightweight, making them well-suited for modern microservices environments.

Overall, the proposed system successfully demonstrates the effectiveness of CNN architectures for cloud-native log classification and failure detection, establishing a strong foundation for future extensions involving forecasting, reinforcement learning, or online learning modules.

VI. Conclusion and Future Work

This study presented a deep learning–based approach for automated failure detection in cloud-native microservices through the classification of unstructured log messages. By utilizing more than 100,000 real log entries collected from Kubernetes-based microservices, two Convolutional Neural Network architectures-cnn-v6 and cnn-v10-were evaluated to determine their effectiveness in identifying *INFO*, *WARNING*, and *ERROR* messages. The results demonstrate that CNNs are capable of extracting meaningful semantic and structural patterns from log text, enabling highly accurate classification without requiring more complex models such as LSTMs or transformer-based architectures. Among the models evaluated, cnn-v6 delivered the most reliable performance with an accuracy of 99.92%, precision of 99.21%, and only eight false negatives, making it suitable for deployment in operational monitoring pipelines where misdetection of errors can lead to significant service disruptions. Conversely, cnn-v10 exhibited lower performance, particularly in identifying *WARNING* logs, implying that deeper architectures do not necessarily yield better results when dealing with structured and repetitive log formats.

Although the proposed system produces strong results, several limitations should be acknowledged. The research is restricted to supervised CNN classification and focuses on detecting known log patterns rather than unseen anomalies or sequence-level irregularities. The dataset is derived from a single environment, limiting the evaluation of cross-platform generalization. Furthermore, the study does not explore real-time adaptation or long-term behavioral changes in log patterns, which are common in evolving cloud-native systems.

Future work can expand in several promising directions. Integrating forecasting models such as ARIMA, LSTM, and Prophet may enable predictive detection of failures and resource exhaustion. Reinforcement learning offers opportunities for dynamic autoscaling and adaptive monitoring strategies that optimize system reliability while controlling resource costs. Real-time online learning mechanisms can be introduced so the model can adapt to emerging log patterns without full retraining. Additional extensions include the incorporation of cloud cost simulation modules, enabling an integrated view of failure detection and financial optimization, as well as extending the system to support multi-cloud platforms such as AWS, Google Cloud, and hybrid deployments. Future studies may also investigate the potential of transformer architectures, graph neural networks, contrastive learning frameworks, and distributed tracing–based anomaly detection to capture deeper contextual relationships across microservices. These avenues highlight that the current model serves as a strong foundation for more sophisticated, predictive, and self-healing cloud-native monitoring systems.

Acknowledgment

This research was supported by the Ministry of Communication and Digital of the Republic of Indonesia through the KOMDIGI Scholarship 2024, which provided funding and academic support throughout the duration of this study. The authors also express their sincere appreciation to the CloudEx Research Group at Universitas Gadjah Mada for providing continuous guidance, mentorship, and access to the research infrastructure required to conduct the experiments and analysis presented in this paper. Their contributions greatly enhanced the development and completion of this work.

References

[1] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning," *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 1285–1298, 2017.

[2] W. Yuan, H. Sun, M. Pang, H. Wang, G. Wu, and Y. Zhang, "LogContrast: Log-based Anomaly Detection Using BERT and Contrastive Learning," *IEEE TrustCom*, pp. 2510–2520, 2024, doi: 10.1109/TrustCom63139.2024.00349.

[3] J. Wang, J. Zhou, Y. Li, and X. Du, "Trusted Log Tracing Framework Based on Fabric," *2023 9th International Conference on Computer and Communications (ICCC)*, pp. 2644–2649, 2023, doi: 10.1109/ICCC59590.2023.10507359.

[4] D. A. Khudyakov and G. E. Yakhyaeva, "Unsupervised Anomaly Detection on Distributed Log Tracing through Deep Learning," *2024 IEEE 25th Int. Conf. of Young Professionals in Electron Devices and Materials (EDM)*, pp. 1830–1835, 2024, doi: 10.1109/EDM61683.2024.10615125.

[5] Y. Sun, J. Keung, J. Zhang, H. K. Yu, W. Luo, and S. Liu, "Unveiling Hidden Anomalies: Leveraging SMAC-LSTM for Enhanced Software Log Analysis," *2024 IEEE 48th Computers, Software, and Applications Conference (COMPSAC)*, pp. 1178–1187, 2024, doi: 10.1109/COMPSAC61105.2024.00157.

[6] D. A. Bhanage, A. V. Pawar, and K. Kotecha, "IT Infrastructure Anomaly Detection and Failure Handling: A Systematic Literature Review Focusing on Datasets, Log Preprocessing, Machine & Deep Learning Approaches and Automated Tool," *IEEE Access*, vol. 9, pp. 156392–156415, 2021, doi: 10.1109/ACCESS.2021.3128283.

[7] J. Meng, Z. Liu, J. Li, and H. Zhang, "LMGD: Log-Metric Combined Microservice Anomaly Detection Through Graph-Based Deep Learning," *IEEE Transactions / Conference Publication* (as provided in uploaded file), 2023.

[8] Y. Liu, W. Xu, M. Tang, and Z. Liu, "Failure Detection Using Semantic Analysis and Attention-Based Classifier Model for IT Infrastructure Log Data," *2022 IEEE International Conference on Big Data (Big Data)*, pp. 1234–1243, 2022.

[9] S. M. et al., "Efficient and Robust Trace Anomaly Detection for Large-Scale Microservice Systems," *IEEE Publication* (as provided in uploaded file), 2023.

[10] P. He, J. Zhu, S. He, and M. R. Lyu, "Anomaly Detection on Servers Using Log Analysis," *Proc. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 1–12, 2016.