

Using AI for Predictive Debugging in Runtime Environments

¹Manish Patil, ²Shraddha Sanjay Mhetre

¹ MTS Software Engineer, Salesforce, Inc.

² Android Engineer

Abstract

Runtime failure debugging for advanced software systems is still a time-consuming and error-ridden process, particularly when using conventional rule-based approaches. To counter this, the paper introduces a predictive debugging system based on Hugging Face Transformers, namely the CodeT5 model to detect anomalies and make failure predictions based on sequences of software logs. The approach entails structured preprocessing, temporal log sequencing, and semantic token embedding for enhanced analysis. The data used is obtained from Stack Overflow Logs in Kaggle. Elasticsearch is used for scalable storage of logs, and Kibana is used for real-time visualization and alerting. Experimental outcomes indicate that the designed model has an accuracy over 98%, precision, recall, and F1-scores over 0.97 for both runtime and non-runtime error classes. The confusion matrix validates negligible misclassifications, and the ROC curve results in an AUC of 0.99, which indicates nearly perfect classification performance. Attention heatmaps improve explainability by making salient failure tokens stand out. Overall, the system provides accurate, scalable, and interpretable predictive debugging in intricate runtime environments.

Keywords Artificial Intelligence; Debugging; Error Prediction; Machine Learning; Runtime Environment.

1 INTRODUCTION

Current software systems are growing to be complicated, dynamic, and distributed (Kumar, 2023). Maintaining runtime reliability is a crucial issue in software engineering, particularly with the increasing interdependence on cloud-native applications, microservices, and real-time data processing pipelines (Ugwueze, 2024) (Ugwueze, 2024). Runtime failures can lead to poor system performance, service failures, or even data losses, as failures occur during its execution (Zhao et al., 2023). Conventional methods of debugging, which cannot work without the help of manual examination of the code, the addition of breakpoints, and review of the logs after a failure have occurred, can be reactively inefficient. Such techniques are particularly deficient in high availability, low latency, and continuous integration environments. Common runtime systems generate huge amounts of log data that may include structured and unstructured information, such as timestamps, as well as function calls, warnings, exceptions, and stack traces. Among the methods to analyze such logs is manual analysis, but it only goes to discovering anomalies, which is not only tedious but also prone to mistakes. Therefore, early signs of failure should be detected and interpreted with automated intelligent mechanisms that can identify and interpret the early signs during the run of the program. Artificial intelligence (AI) and machine learning (ML) present excellent opportunities in this respect, especially with methods of anomaly detection, predictive analytics, and automatic fault localization. Such techniques make predictive debugging possible—a new paradigm in which failures are detected before they occur—and so cut down time when a system is down, increase reliability, and allow proactive maintenance of the system.

There is a range of AI-fueled approaches to detecting faults and analyzing logs discussed in the literature. The previous studies concentrated on the application of statistical methods to system logs anomaly detection, including clustering algorithms and rule-based systems. Such approaches tend to be rigid concerning predefined thresholds and hand-tuned features, and their suitability to multiple environments is constrained (Harjunpää, 2023). In the more recent past, deep learning works on Long Short-Term Memory (LSTM) networks, autoencoders, and convolutional neural networks (CNNs) have modeled sequential dependencies in log data and trained the model to detect abnormal patterns by looking at deviations against learned normal patterns. Although they are a better version of the traditional approaches that waste a lot of manual effort and are less accurate in detecting data that cannot be presented as a demographic breakdown, they have a few limitations. Most of these models assume that log entries are purely flat text sequences without treating structural and semantic information present in the source code of an application. This limits their capacity to make situational interpretation or track anomalies to certain lines or

modules in the codebase. Also, the usage of such models is too often subject to the needs of big volumes of labeled training data, which are hard to attain in such production systems. The other significant weakness is that they are not integrated into effective near-real-time operating solutions like the CI/CD pipeline, where instant accountability, low-latency inference, and natural alerting are required. Lastly, the majority of the currently proposed methods aim only at the detection of anomalies and lack functionality to automatically resolve the bugs found or to make suggestions at the code level.

The paper explores the idea of an AI-based program framework to predict application bugs in runtime environments to fill in the gap in the existing methods (Fortier, 2024). The most important point is that they will incorporate Transformer-based models that can learn both the semantics of the code and log behavior. Some of the models that are used include CodeT5, because they process and generate programming code in a context-sensitive manner. These models are trained on fine-tuned datasets, that can be used to pretrain code-aware code language models. The proposed framework is aimed at processing the runtime logs and code segments in order to pinpoint the possible points of failure and develop estimates of the likelihood of failures. It also includes an inbuilt suggestion module that generates candidate bug fixes over the learnt patterns. The system was built as a real-time system where it is deployed to interact with CI/CD pipelines, Elasticsearch to ingest logs, and Kibana to visualize in real-time. The integration is integrated, allowing active, smart monitoring and early fault detection in active runtime environments. The proposed system brings predictive analytics into code understanding to make software debugging a proactive rather than reactive science.

1.1 Problem Statement

The current industrial environment in Industry 4.0/5.0 has more dependencies on software-based infrastructures, which live in complex, dynamic, and disseminated environments (Zanbouri et al., 2024). Such systems produce enormous amounts of run-time information, and it is not easy to monitor, analyze, and foresee the behaviour of software manually. Conventional debugging techniques are reactive, expensive, and cannot detect failures until they affect performance (Hrusto, 2024). The absence of proactive tools for monitoring restricts the possibility of predicting faults in real-time to make decisions and improve operational efficiency (Adepoju et al., 2022). Current AI-based models of anomaly detection apply to isolated log sequences, neglecting the semantic organization of software and failing to interface with execution environments. Moreover, present methods rarely give practical feedback like solution proposals or decision assistance. The urgent problem is why smart systems can explain the most common runtime patterns, foresee potential missteps, and enhance the observability of software. Moreover, the lack of real-time hits with CI/CD pipelines and monitoring systems also adds strain to the difference between analysis and real implementation. Such issues should be addressed by a unified solution that integrates predictive modeling, code awareness, and a visual form of feedback. With this aim, the research that will be proposed implements an AI-driven, modifiable agent-driven architecture that embraces Transformer models and real-time log analysis to forecast metrics of software applications and prescribe active responses within industrial runtime systems.

1.2 Research Motivation

The complexity of software systems within an industrial setting has augmented to a point where conventional debugging techniques are no longer adequate to guarantee the reliability of such systems at run-time (Atemkeng et al., 2024). The huge size of log data and dynamic execution patterns require smart and automated proactive fault detection solutions. The semantic understanding of code is also rarely available with existing models and is not well interfaced with real-time systems. The gap needs to be filled using predictive mechanisms and context-aware mechanisms to a great extent. It becomes an incentive to create an AI-based framework that can predict failures and improve the observability of software in the course of its execution.

1.3 Research Significance

The study is important in the development of software maintenance using predictive debugging based on AI in industrial execution environments. It improves its process of identifying possible

failures before they happen through incorporating the Transformer models and real-time log analytics. The strategy helps lower system downtimes, enhances the efficiency of operations, and aids in making intelligent decisions. It also covers existing shortcomings with regard to semantic log meaning and corrective action. The framework provides a foundation for intelligent future software systems capable of adaptive and self-monitoring.

1.4 Key Contributions

- Design of an AI-driven agent-based architecture for real-time predictive debugging in dynamic industrial runtime environments.
- Integration of Transformer-based models to semantically interpret execution logs and source code for early fault detection.
- Utilization of pre-trained Hugging Face Transformers (such as CodeT5) for context-aware analysis and failure prediction.
- Deployment of real-time log monitoring and visualization tools to support proactive system observability and automated debugging actions.

1.5 Rest of the Section

- Section 2 provides the Literature Review, summarizing existing work on predictive debugging, anomaly detection, and Transformer-based approaches, along with their limitations.
- Section 3 details the Methodology, covering the proposed architecture, datasets, model integration using Hugging Face Transformers, and real-time implementation workflow.
- Section 4 discusses the Results and Analysis, focusing on model behavior, system performance, visualization outcomes, and real-time debugging effectiveness.
- Section 5 concludes the study and outlines future research directions for enhancing predictive intelligence in runtime environments.

2 LITERATURE REVIEW

Yang and Harris (2025) introduces LogLLaMA, a transformer-based anomaly detection framework trained on LLaMA 2, but only on normal logs, to learn standard patterns of run-time behavior. The model will produce the next log messages in a generative style and report a deviation as an anomaly. Reinforcement learning will also increase the capacity of the model to identify detectors in real-time log files. When the superior performance on the BGL, Thunderbird, and HDFS datasets was tested, it was better than the state-of-the-art procedures. The present work brings to the fore the possibility of large language models in semantic logs analysis and predictive fault in complex runtimes.

Guan et al. (2024) suggests an anomaly detection framework based on log, called LogLLM, that combines BERT to extract semantic features and LLaMA as a decoder Transformer to classify the sequence. An intermediate projection layer is included, and the two semantic spaces of BERT and LLaMA are aligned with consistent log representation. Unlike other log parsers, LogLLM employs regex for lightweight preprocessing, and so it does not depend upon templates. Training the model follows a unique three-phase initialization to enhance generalizability over fragile log formats. These results on four benchmark datasets are demonstrated experimentally to indicate that LogLLM is better than other methods in accuracy as well as flexibility.

Li et al. (2025) introduces AnomalyGen, an automated log synthesis system that provides a solution to these weaknesses of the available public log datasets and supports research with anomalies. Its architecture is used to analyze the programs in a more enhanced program analysis augmented with Chain-of-Thought reasoning that is interactive with the creation of realistic logs and annotation of anomaly occurrence without the execution of the system. Results of experiments on the Hadoop and HDFS systems indicate that the difference between the coverage of events and the degree of realism of the operations is significant. The paper presents a fresh paradigm of the generation of log data sets and demonstrates that a software engineering task can be assisted by the large language models.

Lim et al. (2025) explores the possibility of parameter-efficient fine-tuning (PEFT) techniques, like LoRA and ReFT, to adapt large language models (LLMs) to the tasks of log anomaly detection (LAD). The paper compares RoBERTa, GPT-2 and LLaMA-3 on four publicly available datasets in

terms of detection accuracy, stability, and cross-dataset generalization. The proposed method circumvents this shortcoming of full fine-tuning in that it allows scalable and efficient applications of LADs. Findings indicate that PEFT methods are robust with unstable logs as they improve sample efficiency. This paper provides some background information on using lightweight LLM versions to detect anomalies in system logs in the real world.

Qi et al. (2023) suggests LogGPT, which is a log-based anomaly detection framework that exploits ChatGPT's language understanding to help analyze complex log data. Linguistics knowledge is transferred to the log analysis tasks in the model by prompt-based interaction based on large-scale corpora. LogGPT is competitive on BGL and Spirit datasets when evaluated at par with the traditional deep learning approaches, and it is more interpretable. Although successful, the study has its narrow specialization as the authors apply the concepts of anomaly detection and do not cover the areas of runtime debugging and system fault remediation. This paper recognises the possibility of LLMs such as ChatGPT in log comprehension but does not go beyond real-time predictive debugging integration.

Almodovar et al. (2024) presents a LogFiT, a self-supervised deep learning model based on BERT regarding language modeling to identify log anomalies without requiring log templates or labeled data. It is trained on standard logs with masked sentence prediction on how to recognize the standard log patterns in the model. The anomalies are determined using top-k token prediction precision over previously unseen log sequences. The results of experiments on HDFS, BGL, and Thunderbird data indicate that LogFiT is better than baseline techniques and is not sensitive to log variability. Compared to earlier supervised methods, this model is more practical, but it has not yet reached real-time debugging or runtime remediation.

Zhou et al. (2024) investigates the use of ChatGPT and BERT on log parsing and anomaly detection in the field of distributed computing. BERT encoder is applied as a transformer encoder context to learn regular patterns of log sequences through masked language modeling and prediction of adjacent sentences. The presence of ChatGPT advances the comprehensibility and meaning of log messages. The experimental results research performed in two data sets suggests that the hybrid model is better than classic approaches like Deeplog and LogAnomaly. Inasmuch as it works well in anomaly detection, the study does not cover any real-time debugging, and the capability of intervention at run-time.

Lee et al. (2023) present a LAnoBERT as a parser-free log anomaly detection system founded on BERT that aims to remove the reliance on log templates and minimize information loss. It relies on masked language modeling of unsupervised learning and estimates the probability of anomalies using loss functions on each log key when testing. Tests of the HDFS, BGL, and Thunderbird data indicate that LAnoBERT exceeds unsupervised counterparts and achieves similar performance to the supervised metrics. Nevertheless, similarly to other detection procedures, it only identifies faults, does not give any predictive debugging, or real-time fault correction.

Zhang et al. (2024) offers an Artificial Intelligence for IT Operations (AIOps) methods that apply to LLMs and take failure management in large-scale distributed systems. In the paper, the emerging method of AIOps is examined in terms of its capabilities, data requirements, task-specificity, etc. It outlines some of the most important flaws (low levels of generality and cross-task abilities) and how LLMs ought to counter them. It is, however, conceptual, where little attention has been given to real-time predictive debugging of executing applications in live running environments. Table. I shows the Summary on Literature review.

Table 1: Summary on literature review

Author(s)	Key Contribution	Advantages	Disadvantages
Yang & Harris (2025)	Proposed LogLLaMA, trained only on normal logs to predict next log line and flag anomalies.	Works well with reinforcement learning, tested on multiple datasets.	Does not use faulty logs for training, limited to generative prediction.

Guan, Cao, Qian, Gao, & Ouyang (2024)	Introduced LogLLM, combining BERT and LLaMA with an alignment layer for semantic consistency.	High accuracy and flexibility; regex-based lightweight preprocessing.	Complex architecture; limited explanation of real-time use.
Li et al. (2025)	Developed AnomalyGen, a log synthesis tool with Chain-of-Thought reasoning.	Helps generate realistic logs without actual system execution.	Focuses on log creation, not real-time fault detection.
Lim, Zhu, & Pang (2025)	Explored PEFT (e.g., LoRA) to fine-tune LLMs for log anomaly detection.	Scalable, efficient, and robust across datasets.	Focused only on detection, not predictive debugging.
Qi et al. (2023)	Proposed LogGPT, using prompt-based ChatGPT for log anomaly detection.	High interpretability and competitive results.	No predictive debugging or remediation features.
Almodovar, Sabrina, Karimi, & Azad (2024)	Introduced LogFiT, a BERT-based self-supervised method using masked log modeling.	Template-free, handles log variability well.	Lacks support for real-time prediction or debugging.
Zhou et al. (2024)	Used ChatGPT + BERT for log parsing and anomaly detection.	Good accuracy and interpretability.	No support for real-time debugging or root cause tracing.
Lee, Kim, & Kang (2023)	Presented LAnoBERT, a parser-free BERT model for anomaly detection.	Template-free, performs well in unsupervised settings.	Only detects faults, no predictive or corrective capabilities.
Zhang et al. (2024)	Surveyed AIOps methods using LLMs for failure management.	Highlights potential of LLMs in operations.	Conceptual only, no real-time execution focus.

Current research on log anomaly detection through large language models (LLMs) attests to substantial advancements in harnessing transformers to produce predictive information. Models such as LogLLaMA, LogLLM, and LogGPT reflect advances in log comprehension, anomaly identification, and semantic encoding. Some methods, including PEFT and LogFiT, emphasize efficiency and template-free identification and thus are more suitable for real-world logs. Tools such as AnomalyGen and LDB contribute to log synthesis and runtime-aware debugging, pushing the frontiers of automated fault analysis. Most models perform well with high accuracy on benchmark datasets; however, in general, most models fall short when it comes to integration with real-time predictive debugging. LDB, for instance, gets close to actual runtime remediation, but, at large, most works remain detection-focused. These results strengthen the potential of LLMs in system log analysis and reveal limitations in proactive debugging in live fault correction.

3 METHODOLOGY

The methodology adopted in this paper describes a comprehensive and rational pipeline for obtaining predictive debugging of runtime environments through novel AI methodologies. It starts with persistent gathering of real-time system logs and error traces from heterogeneous software components, so that both usual and unusual usage patterns are collected for end-to-end analysis. These raw logs are subjected to structured preprocessing, such as log parsing, noise elimination, temporal sequence building, and tokenization, thus transforming them into machine-readable and semantically rich formats. Utilizing the strength of Hugging Face Transformers, namely the CodeT5 model, the textual log data is converted into high-dimensional semantic embeddings that maintain contextual relationships and time-based dependencies. The model then uses attention-based mechanisms to examine these embeddings and detect unusual patterns that could

indicate possible failures or system anomalies. The framework uses unsupervised and self-supervised learning to make it generalizable across different runtime conditions without the need for labeled data. A specific feedback loop provides continuous model adaptation and anomaly improvement, with increasing accuracy over time. In addition, the approach combines real-time alerting, root cause localization, and performance monitoring, thus being a scalable, explainable, and proactive debugging solution for contemporary software systems. Figure 1 depicts the entire architecture of the proposed framework.

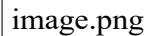


image.png

Figure 1: Proposed Methodology

The architecture of the envisioned predictive debugging system based on the Hugging Face CodeT5 transformer model is presented in the figure. It starts with log data gathering, then proceeds with preprocessing activities including parsing, cleaning, creation of temporal sequences, and tokenization. Elasticsearch stores the structured logs for subsequent indexing and visualization. At the same time, the preprocessed logs are passed into the transformer pipeline, beginning with log sequence modelling to learn normal and anomalous behavior. The model proceeds to perform predictive debugging to identify anomalies and predict future failures. Root cause identification is then performed by localizing the source of the error. Ultimately, real-time alerts are visualized through Kibana, and system performance is monitored in real-time for proactive response.

3.1 Data Collection

Data Collection in this study is from the benchmark dataset Stack Overflow Code Error Logs (Kaggle), which includes varied samples of software bugs and traces of errors in actual programs. Such data contains timestamped log messages, exception traces and structured tube activity, both during normal operations and during events of failure (Stack Overflow and 1 collaborator 2019)

3.2 Data Preprocessing

The process of data preprocessing in the research presented is of great importance, being set to turn raw runtime log data and convert it into a well-arranged and sensible format, which can be taken and rearranged to be used in AI-based predictive analysis. This starts with log parsing, where raw log lines are divided into recognizable fields like timestamps, error classes and environments where the execution took place. It is then done through the clearing up of the data, which involves the elimination of irrelevant or noisy data items in order to improve the quality of the input. A time order is then used to maintain execution order and to record dependencies between events in a context. The processing of that text into a numerical vector with semantic relationships is performed through tokenization and embedding with Hugging Face Transformers. All

these steps combine in preparing the input data in the best possible conditions to provide enough information to note the intents of anomalies and errors, which result in high consistency of anomaly alarming.

3.2.1 Log Parsing

Log parsing involves the derivation of structured data like time stamps, severity, components and text from a raw runtime log created by an application. This makes the data ready to be analyzed with AI by converting log lines into a form that could be parsed step-by-step into models such as those offered by Hugging Face Transformers.

$$L_i = \text{Parse}(l_i) = (t_i, s_i, c_i, m_i) \quad (1)$$

Where, l_i is raw log entry, L_i is parsed structured entry, t_i is timestamp, s_i is severity level, c_i is component/module, m_i is log message.

3.2.2 Data Cleaning

Noise Catches irrelevant, malformed, and noisy log entries are discarded by data cleaning, and formats are standardized. This enhances the reliability of the models since only clean, informative inputs are being fed to the Hugging Face Transformer, giving the model an insight on how the model was running normally or abnormally.

$$L_{clean} = L_i / \text{valid}(L_i) = \text{True} \quad (2)$$

Where, L is all parsed logs, L_{clean} is cleaned subset, $\text{valid}(L_i)$ is Boolean function verifying integrity.

3.2.3 Temporal Sequence Construction

It constructs sequences of log events by time and by execution context (e.g. session ID or transaction), maintaining causality. Such sequences are required so that Hugging Face Transformer models can discover patterns of normal run time behavior and can detect precursors of anomalies.

$$S_j = [L_{j1}, L_{j2}, \dots, L_{jn}] \text{ with } t_{j1} < t_{j2} < t_j \quad (3)$$

Where, S_j is sequence of events, L_{jn} is sequence, t_{jn} is timestamp.

3.2.4 Tokenization and Embedding

Tokens that are constructed based on the literal content of log messages are converted into dense vectors representations with the Hugging Face Transformer model, or more precisely the CodeT5 model. Such an embedding procedure retrieves the semantics and syntactic details of every token and makes the model comprehend the contextual relevancy of various log entries. By saving both the nature and the timing of the log entries, the transformer can denote the sequential way of the system operations throughout the functioning. Such embeddings enable the model to distinguish both the normal and abnormal events with the differences in log patterns. Consequently, the system learns more about the behavior at run time exhibited by more than matching keywords or pattern rules. This semantic meaning is critical in the occurrence of obscured signs of failure which cannot be realized through routine analysis. The learned vectors are used as inputs to downstream tasks including cost, fault classification. The idea of embedding log tokens using CodeT5 in the debugging framework makes the debugging framework smarter and more predictive.

$$x_{jk} = f_{HF}(m_{jk}) \text{ and } X_j = x_{j1}, \dots, x_{jn} \quad (4)$$

Where, m_{jk} is message text from log, f_{HF} is Hugging Face tokenizer, x_{j1} embedded vector, X_j is complete embedded sequence.

3.3 Problem Formulation

Within a large and dynamic software system, debugging at run time is a time-sensitive and non-trivial exercise. The failures in these settings are characterized by a lack of clear static code violations, so the intelligent systems have to be capable of anticipating and localizing the possible faults. This research attempts to answer this challenge by proposing predictive debugging as a sequence-based classification task on log data of the program execution. Rather than concerning analysis of future failures, the objective is to model the run-time behavior so that the system has the ability to predict probable faults before they are triggered. Let a log sequence collected from a runtime environment be denoted by,

$$X = x_1, x_2, \dots, x_n \quad (5)$$

Where, x_n is a tokenized and vectorized representation of a log message. They are preprocessed and encoded by Hugging Face Transformer-based encoders, which record both the syntax and semantics of the runtime process. The sequence captures the temporal flow of system operations, allowing for modeling long-distance dependencies and error propagation channels. The predictive debugging model seeks to approximate a function is given by,

$$f(X) = y[0, 1] \quad (6)$$

Where, y represents the predicted probability of a runtime failure, X is log sequence. A high value of y indicates a high probability of system malfunction, thus raising alarms or invoking self-healing processes. The output of classification acts as an early warning sign, which facilitates runtime anomaly detection and debugging in mission-critical systems. It utilizes Transformer models of Hugging Face, which are based on multi-head self-attention to capture contextual relationships,

$$Attention(Q, K, V) = softmax((QK^T)/(d_k))V \quad (7)$$

Where, (Q, K, V) are query, key, and value matrices derived from the input embeddings, d_k is the dimensionality of the key vectors. This process allows the model to balance the relative significance of every log entry against the others, improving its ability to pick up on faint signs of failure. The contextual representation derived from the Transformer layers, is aggregated into a fixed-length vector on the entire log sequence. This vector is then passed through a fully connected layer and a sigmoid function to generate the final failure probability,

$$y = (W^T h + b) \quad (8)$$

Where, W and b are the learnable parameters of the classifier, σ is the sigmoid activation function that maps the output to the interval. Such a formulation converts log sequences into a runtime behavioral signature, allowing for the detection of suspicious patterns that could potentially result in faults at an early stage. In contrast to traditional log parsers or rule-based approaches, the presented model works on a data-driven basis and is well-suited to process unseen runtime situations. By utilizing pre-trained Transformer models and fine-tuning them on domain-specific log datasets, the method sustains high generalization capacity while retaining context sensitivity.

3.4 Transformer-Based Predictive Modelling

The central element of the suggested system is based on state-of-the-art Transformer-based models, namely CodeT5, which are accessed through the Hugging Face framework. These models are selected because of their competency in comprehending the syntax of programming languages as well as natural language patterns of log data. The models are fine-tuned over preprocessed inputs derived from filtered datasets such as BigCode, Defects4J, and Stack Overflow Error Logs. The fine-tuning procedure allows the models to learn domain-specific runtime sequences for adapting to the behavioral flow of software systems. All input sequences preserve temporal continuity and semantic context of the execution logs, as it is necessary for learning significant representations. It aims at conditioning the model to identify anomalies that cannot be found by static rule-based systems. Transformer models are best fit for this job as they can pay attention to relationships between all the components within a sequence, irrespective of distance. This enables the model to emphasize important dependencies and patterns in long or intricate log traces, for example, repeated anomalies or multi-stage failure sequences. Attention mechanisms assist in determining the most significant tokens used in forecasting an upcoming error, providing increased interpretability. The model is able to differentiate between common log noise and actual fault signals, learning these differences automatically from labeled and unlabeled sequences. Unlike typical anomaly detection systems that usually need handcrafted features or regex templates, the Transformer learns these structures data-driven. This results in better generalization, particularly in dynamic or previously unseen runtime scenarios.

After deployment, the model reads in real-time incoming log sequences and returns a scalar score for failure likelihood. Higher risk score sequences are marked for review or automated action. Learning from the structural and contextual cues contained in past logs, the model can predict problems before fully occurring in the system. In addition, attention heatmaps and token contribution analysis provide insight into the model's decision-making process, helping developers comprehend why a specific sequence had been flagged as anomalous. This predictive modeling methodology provides increased runtime observability, allows for preemptive debugging, and facilitates continuous monitoring in extensive software ecosystems. Finally, Transformer-based modeling closes the gap between log-based monitoring and insightful, context-aware debugging.



Figure 2: Transformer based Predictive modelling

Figure.2 depicts an end-to-end flow of the suggested predictive debugging system with transformer-based models such as CodeT5. It starts with real-time log gathering from software contexts, which passes through preprocessing operations like tokenization and filtering to make structured inputs ready. The inputs along with filtered datasets such as Stack Overflow Logs are utilized to fine-tune the CodeT5 model through the Hugging Face framework. The transformer produces semantic and temporal embeddings of log streams, which are processed with attention mechanisms to identify key tokens. From this, the system makes predictions on possible failures by giving risk scores to log streams. High-risk logs activate alerts automatically, while heatmaps and token examination offer interpretability to the system for better insight. The process allows for continuous monitoring and proactive debugging, making it possible to carry out proactive system reliability and fault handling.

3.5 Integration and Real-Time Monitoring

In order to implement the predictive debugging system, the trained Transformer-based models are integrated into a real-time software pipeline with contemporary DevOps techniques. The integration comes in the form of continuous integration and continuous deployment (CI/CD) pipelines, which support effortless rollout into live environments. The pipelines support automated updates, model retraining loops, and quick rollout of enhancements with minimal human intervention. The system, when deployed, oversees running applications by consuming real-time log data created during program runtime. The steady flow of logs captures the real-time running state of software elements, such as performance metrics, exception call stacks, and contextual runtime state variables. The architecture enables low-latency inference to achieve timely anomaly detection. Real-time log ingestion is enabled by Elasticsearch, which is a high-throughput distributed search and indexing engine. Runtime logs are captured, parsed, and formatted into searchable forms with customized log collectors. When new logs are streamed in, the predictive model receives

incoming sequences and computes failure likelihood scores on the fly. These scores are added as metadata, facilitating contextual analysis and traceability. Integration ensures that potential faults are caught before escalation, allowing proactive debugging. In addition, the Elasticsearch usage permits effective querying and filtering of historical information, improving incident response and root cause analysis. This environment does away with manual log inspection, which tends to be cumbersome and error-susceptible in complex settings.

The prediction results and anomaly notifications are displayed through Kibana dashboards, offering user-friendly interfaces for developers and operators. These dashboards feature dynamic panels of real-time risk scores, alert levels, anomaly timelines, and categorized failure types. Such visualizations enable prompt situational awareness and support immediate corrective action. Alerts may be set up to initiate automated notifications in the form of email, messaging systems, or webhook integrations.

4 RESULTS AND DISCUSSION

The Results and Discussion section is an extensive analysis of a transformer model namely CodeT5 designed for predictive debugging in runtime environments. It starts from a well-balanced dataset of runtime and non-runtime errors to token classification and embedding. It is measured using accuracy, precision, recall, and F1-score, all of which indicate values greater than 0.97 for both classes, which reflects high dependability. Confusion matrix ensures negligible misclassification, and training-validation plots show outstanding generalization without overfitting. The ROC curve with an AUC value of 0.99 also testifies to the strength of the model. Predicted vs. actual label visualizations also validate the strength of the model in real-world deployments. This analysis testifies to the model's usefulness in early detection of faults and assistive intelligent runtime debugging.

image3.png

Figure 3: Enter Caption

Figure.3 shows two-word clouds of the most common words used in Stack Overflow questions categorized as Runtime Error (left) and Non-Runtime Error (right). Prevailing words such as "problem," "error," "code," and "using" in runtime error questions imply a high emphasis on debugging and failure cases. Conversely, the non-runtime error questions contain words like "create," "use," "method," and "application" more often, reflecting emphasis on development and functionality over failure. The frequency is represented by the word size, with words being used more frequently in the given category. The visual comparison illustrates different semantic patterns in the two categories of questions. These findings validate the application of contextual language models for successful classification.



image4.png

Figure 4: Distribution of labels

Figure.4 shows the distribution of labels in the data, dividing questions into two classes: "Runtime Error" and "Not Runtime Error." Both are almost of equal size, having approximately 35,000 entries each, showing a balanced dataset. This balance is important for training machine learning models so that they do not become biased towards one class. The x-axis is the label categories, and the y-axis is the number of questions for each. The even distribution promotes stable model performance to differentiate runtime and non-runtime errors.

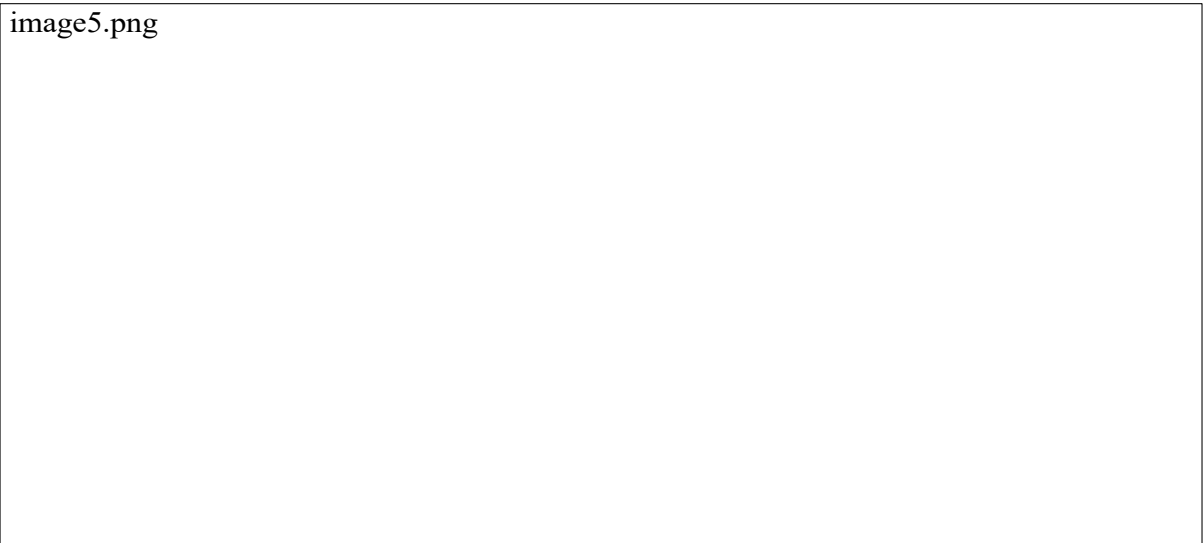


image5.png

Figure 5: Classification Metrics per class

Figure.5 displays the classification measures—precision, recall, F1-score, and accuracy—of two classes: "Not Runtime Error" and "Runtime Error." The measures for each class are almost 1.0, indicative of superb performance of the prediction model. The similarity in performance of the model across the measures and classes is indicative of the fact that the model performs equally well on both types of logs without any bias. The identical scores indicate the stability and consistency of the approach. This visualization reinforces that the model achieves high accuracy and consistency in classifying runtime errors in system logs.



image6.png

Figure 6: Actual Vs Predicted labels

Figure.6 shows the bar chart for comparison of actual and predicted counts of labels for two classes is "Not Runtime Error" and "Runtime Error." For the "Not Runtime Error" class, the model had predicted 6983 instances against 7040 actual cases and provided a very close fit. For the "Runtime Error" class, it over-predicted slightly with 7107 predictions for 7050 actual cases. The slight variation in bar heights is a testament to the high precision and low error margin of the model between expected and actual values. The diagram graphically verifies the efficacy of the model in accurately classifying both forms of log entries. This is indicative of the predictive debugging method's ability.



image7.png

Figure 7: Confusion Matrix

Figure.7 demonstrates the predictive model's classification performance on runtime error detection. Of all "Not Runtime" logs, 6816 were accurately predicted and 224 were wrongly predicted as "Runtime." Likewise, out of all "Runtime" logs, 6883 were accurately predicted, and just 167 were wrongly labeled as "Not Runtime." The values on the diagonal line (6816 and 6883) are actual positives for both classes, which is indicative of high model accuracy. The off-diagonal values (224 and 167) are low, which is an indication of low misclassifications. The deeper blue along the diagonal also visually verifies high accuracy. This matrix verifies the model's consistent and solid performance in both classes.

image8.png

Figure 8: Enter Caption

Figure.8 training vs validation accuracy for five epochs. Validation accuracy is initially higher, but training accuracy picks up very quickly and overtakes it by the second epoch. Both the accuracies increase consistently, with training accuracy approaching a value of 1.0 by epoch 4, and validation accuracy plateauing at about 0.97. This implies good learning and low overfitting. The insignificant gap between the two plots implies good generalization. The model in general proves to be highly accurate and consistent across training and validation periods.

image9.png

Figure 9: Training Vs Validation Loss

Figure.9 illustrates the evolution of training and validation loss over five epochs. At first, both losses are high, with training loss beginning above 0.30 and validation loss at approximately 0.22. As training continues, both losses fall sharply, suggesting effective learning. Training loss continues to drop steadily from the second epoch onwards, while validation loss levels off at about 0.10. The difference between training and validation loss increases slightly, suggesting slight overfitting. Nevertheless, the general trend is indicative of model convergence and stability of learning. This trend indicates a well-optimized model with excellent generalization capability.



image10.png

Figure 10: ROC Curve

Figure.10 depicts the classification capability of the model proposed, with the True Positive Rate versus the False Positive Rate. The orange line tightly hugs the top-left corner, which demonstrates superior discriminative power. The value of Area Under the Curve (AUC) is 0.99, indicating almost perfect accuracy to separate runtime error from non-error. The sharp incline and plateau of the curve indicate low false positives and high sensitivity. The dotted diagonal line denotes a no-skill classifier, and the curve of the model well above it indicates its good performance. This confirms the reliability of the model for predictive debugging jobs.

5 CONCLUSION AND FUTURE WORKS

An end-to-end predictive debugging system in runtime environments based on transformer models, CodeT5, with real-time log monitoring tools such as Elasticsearch and Kibana. The system gathers, preprocesses, and semantically parses log data to detect suspicious patterns and make predictions about possible runtime failures. Experimental results validate the robust performance of the model with high precision, recall, F1-score, and accuracy values for runtime and non-runtime error classes. Visual outputs-such as label distribution plots, classification metrics, confusion matrix, ROC curves, and training-validation accuracy/loss graphs-indicate model stability, generalization, and near-perfect classification ability. Confusion matrix and ROC-AUC also substantiate low misclassifications and high sensitivity. Moreover, the methodology allows for enhanced interpretability via visual examination and can be used for prompt alerts and minimizing debugging overhead in intricate software environments. For the future, the system can be expanded to include multimodal inputs, such as telemetry data, system counters, or app traces, to further enhance anomaly detection. Adding explainable AI (XAI) techniques would also enhance developer trust and interpretability of predictions. Also, extension of the model to cross-platform or container environments would further improve scalability in current DevOps pipelines. Applying online learning methods could enable the model to learn constantly from changing runtime behaviors. Lastly, benchmarking the system in various programming languages and log formats would additionally ensure its generalizability and resilience across different software environments.

References

- Adepoju AH, Austin-Gabriel B, Hamza O, Collins A (2022). Advancing monitoring and alert systems: A proactive approach to improving reliability in complex data ecosystems. *IRE Journals*, 5(11): 281–282.
- Almodovar C, Sabrina F, Karimi S, Azad S (2024). LogFiT: Log anomaly detection using fine-tuned language models. *IEEE Transactions on Network and Service Management*, 21(2): 1715–1723.

Publisher: IEEE.

Atemkeng M, Hamlomo S, Welman B, Oyetunji N, Ataei P, Fendji JLK (2024). Ethics of software programming with generative AI: is programming without generative AI always radical? *arXiv preprint arXiv:2408.10554*.

Fortier P (2024). Programming language abstractions for the Internet of Things era. PhD Thesis, INSA de Lyon.

Guan W, Cao J, Qian S, Gao J, Ouyang C (2024). Logllm: Log-based anomaly detection using large language models. *arXiv preprint arXiv:2411.08561*.

Harjunpää N (2023). Log management system technologies and methods for near real-time fault analysis systems: An exploration of log shipping and storage.

Hrusto A (2024). *Enhancing DevOps with Autonomous Monitors: A Proactive Approach to Failure Detection*. Lund University.

Kumar S (2023). Reviewing software testing models and optimization techniques: an analysis of efficiency and advancement needs. *Journal of Computers, Mechanical and Management*, 2(1): 32–46.

Lee Y, Kim J, Kang P (2023). Lanobert: System log anomaly detection based on bert masked language model. *Applied Soft Computing*, 146: 110689. Publisher: Elsevier.

Li X, Huo Y, Mao C, Shan S, Su Y, Li D, et al. (2025). AnomalyGen: An Automated Semantic Log Sequence Generation Framework with LLM for Anomaly Detection. *arXiv preprint arXiv:2504.12250*.

Lim YF, Zhu J, Pang G (2025). Adapting Large Language Models for Parameter-Efficient Log Anomaly Detection. In: *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 325–337. Springer.

Qi J, Huang S, Luan Z, Yang S, Fung C, Yang H, et al. (2023). Loggpt: Exploring chatgpt for log-based anomaly detection. In: *2023 IEEE International Conference on High Performance Computing & Communications, Data Science & Systems, Smart City & Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, 273–280. IEEE.

Ugwueze V (2024). Cloud Native Application Development: Best Practices and Challenges. *International Journal of Research Publication and Reviews*, 5(12): 2399–2412.

Yang Z, Harris IG (2025). LogLLaMA: Transformer-based log anomaly detection with LLaMA. *arXiv preprint arXiv:2503.14849*.

Zanbouri K, Noor-A-Rahim M, John J, Sreenan CJ, Poor HV, Pesch D (2024). A comprehensive survey of wireless time-sensitive networking (TSN): Architecture, technologies, applications, and open issues. *IEEE Communications Surveys & Tutorials*. Publisher: IEEE.

Zhang L, Jia T, Jia M, Wu Y, Liu A, Yang Y, et al. (2024). A survey of aiops for failure management in the era of large language models. *arXiv preprint arXiv:2406.11213*.

Zhao C, Ma M, Zhong Z, Zhang S, Tan Z, Xiong X, et al. (2023). Robust multimodal failure detection for microservice systems. In: *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 5639–5649.

Zhou Y, Chen Y, Rao X, Zhou Y, Li Y, Hu C (2024). Leveraging large language models and BERT for log parsing and anomaly detection. *Mathematics*.