ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26sep417

# Enhanced Compiler Design with Machine Learning Models Integrated into Communicating Stream X-Machine Framework

Bashir A. Sanusi[1*]; Emmanuel K. Ogunshile[2]; Mehmet Aydin[3]

[1,2,3]School of Computing and Creative Technologies, University of the West of England, Bristol, United Kingdom.

[1]0000-0002-5465-3141
[2]0000-0002-6276-188X
[3]0000-0002-4890-5648

Corresponding Author: Bashir A. Sanusi[1*]

Publication Date: 2026/09/22

**Abstract: The increasing complexity of modern software systems demands more intelligent and reliable compiler designs capable of detecting code anomalies efficiently. Traditional compilers, while effective in syntax and semantic analysis, lack adaptive capabilities for identifying non-trivial errors. This paper presents a performance-aware hybrid compiler that integrates the Communicating Stream X-Machine (CSXM) formal framework with Machine Learning (ML) techniques for intelligent code analysis. The approach combines formal verification with predictive models, including Random Forest, Convolutional Neural Networks (CNN), and CNN-LSTM architectures. Experimental evaluation was conducted using both real-world (DeepFix) and synthetically generated datasets, with over 150 training and testing iterations. Performance was assessed using accuracy, precision, recall, and F1-score. In addition, this study provides a detailed analysis of compilation latency and ML inference overhead, highlighting the trade-off between detection accuracy and execution efficiency. Results show that ML integration significantly improves anomaly detection but introduces measurable latency. The paper also discusses practical challenges, including dataset labelling effort and model generalisability. These findings demonstrate that a performance-aware hybrid CSXM–ML compiler offers a promising approach for developing intelligent, adaptive compilation systems that balance correctness, efficiency, and scalability.**

***Keywords:*** *Compiler, Communicating Stream X-Machine, Machine Learning, Random Forest, Convolutional Neural Network, Long Short-Term Memory, Anomaly Detection, Software Development.*

**How to Cite:** Bashir A. Sanusi; Emmanuel K. Ogunshile; Mehmet Aydin (2026) Enhanced Compiler Design with Machine Learning Models Integrated into Communicating Stream X-Machine Framework. *International Journal of Innovative Science and Research Technology*, 11(9), 657-669. https://doi.org/10.38124/ijisrt/26sep417

## I. INTRODUCTION

The rapid growth in software complexity has increased the demand for compilers that not only translate code but also assist in identifying and correcting errors intelligently [30][27]. Traditional compilers are primarily designed to perform lexical, syntactic, and semantic analysis, ensuring that programs conform to predefined language rules. While effective for structured error detection, these compilers lack the capability to identify subtle, non-trivial anomalies or learn from past code patterns [37][10]. As a result, developers often rely on external tools and manual debugging processes, which can be time-consuming and error-prone [31][15].

Formal methods have long been used to improve software reliability by providing mathematically grounded models for system specification and verification [1]. The Communicating Stream X-Machine (CSXM) framework is one such formal approach that extends finite state machines with memory and communication capabilities, making it suitable for modelling complex and distributed systems [31][4]. Prior research has demonstrated the effectiveness of CSXM in system modelling, verification, and automated test case generation [32][46][5]. However, its application in compiler design remains largely limited to structural correctness and lacks adaptive intelligence.

In parallel, Machine Learning (ML) techniques have shown significant promise in software engineering tasks such

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26sep417

as bug detection, code completion, and anomaly classification. Models such as Random Forests and deep learning architectures, including Convolutional Neural Networks (CNN) and hybrid CNN-LSTM networks, have been successfully applied to learn patterns from large codebases [3][49]. Despite these advancements, ML-based approaches often operate independently of formal verification frameworks, leading to concerns around reliability, explainability, and integration within compiler pipelines [35][38].

This paper addresses these limitations by proposing a hybrid compiler that integrates CSXM formalism with ML-based anomaly detection. Unlike prior work that focuses either on formal modelling or conceptual integration, this study presents a fully implemented system with comprehensive experimental validation [23][31]. The proposed approach leverages CSXM to ensure structural correctness while employing ML models to enhance anomaly detection capabilities within the compilation process.

A key contribution of this work is its focus on performance-aware analysis. While integrating ML into compiler pipelines improves detection accuracy, it introduces additional computational overhead that can impact compilation time. This paper therefore provides a detailed evaluation of compilation latency, ML inference cost, and the trade-off between accuracy and efficiency. Furthermore, the study examines practical challenges such as dataset labelling effort and model generalisability, which are critical for real-world deployment.

➢ *The Main Contributions of this Paper are as Follows:*

- A novel hybrid CSXM–ML compiler architecture for intelligent code analysis.
- Integration and evaluation of multiple ML models for anomaly detection.
- Comprehensive experimental validation using real-world and synthetic datasets.
- Performance analysis focusing on latency, inference overhead, and trade-offs.
- Discussion of scalability challenges, including labelling effort and generalisation.

The remainder of this paper is structured as follows. Section 2 reviews related work in formal methods, compiler design, and ML-based code analysis. Section 3 presents the proposed CSXM-based compiler architecture. Section 4 describes the ML integration approach. Section 5 outlines the dataset and experimental setup. Section 6 presents the results and evaluation. Section 7 discusses performance and practical considerations. Finally, Section 8 concludes the paper and highlights directions for future research.

## II. RELATED WORK

The development of reliable and intelligent compilers has drawn significant attention across three key areas: formal methods for system verification, traditional compiler design techniques, and the application of Machine Learning (ML) in code analysis. This section reviews existing work in these domains and highlights the research gaps addressed in this study. Table 1 presents some of the aspects where machine learning has been or can be applied in compiler.

Table 1 Machine Learning Applications in Compiler.

| Application Areas | Methodology | Machine learning Algorithm |
|---|---|---|
| Compilation Error [20] | This research presents a technique that automatically learns patterns using a deep neural network to recommend program repairs for the most expensive categories of build-time compilation failures. | Deep Neural Network |
| Code Optimization (Super Optimization) [2] | Machine learning (ML) model was trained to generate optimized code sequences which are not only locally optimal compared to the traditional peephole optimizers but globally optimal which exceed the capabilities of standard compiler optimizations. | The Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN). |
| Resource Allocation [19] | Machine learning model was trained to improve register allocation by reducing the number of instructions needed for register spills and resulting in more efficient code. | Recurrent Neural Network (RNN) and Long Short-Term Memory Network (LSTM). |
| Memory Utilization and Compiler Optimization [34] | This research presents techniques of compiler optimization and discusses significant advances in compiler optimization methods. Also, explore the in-depth knowledge of machine learning in compiler optimization. | Support Vector Machine, Decision Trees, and K means Clustering Model. |
| Compilation Time Prediction [14] | Machine learning was used to predict how long it takes to compile a given program source code by helping in scheduling a better build job in large-scale software projects i.e. estimating compilation time based on code complexity metrics and historical data. | Artificial Neural Network (ANN) and K-means Cluster. |
| Code Generation [8] | Machine learning models was used to generate code from higher level specifications by automating parts of the compilation process. | Convolutional Neural Network (CNN), Recurrent Neural Network (RNN), Random Forest (RF), Multi-Layer Perceptron (MLP), K Nearest Neighbors (KNN). |

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26sep417

| | | |
|---|---|---|
| Compiler Heuristics [29] | A machine learning-based approach for self-optimizing compiler heuristics was introduced to tailor optimization decisions in a dynamic compiler for specific environments. | Neural Networks. |
| Compilation Error Categorization [12] | This study presents a novel compilation error categorization method based on the program's smallest unit of the program referred to as tokens. | Neural Network. |
| Code Optimization (Predictive Optimization) [47] | ML model was used to predict the most effective optimizations for a given piece of code based on its characteristics such that rather than applying a fixed sequence of optimization passes, an ML model was used to predict the best passes on the code's features which lead to better performance and smaller binaries. | Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN). |
| | | |
| Code Optimization [48] | The study outlines an approach to code optimization that combines classical machine learning and deep learning methods, leveraging their strengths while minimizing their limitations. | XG boost, Random Forest, Ada Boost, Support Vector Machine, Gaussian Naïve Bayes, Logistic Regression, Ensemble Learning. |
| Compilation Errors [33] | This research introduces a method for extracting practical program repair examples from student program execution logs. | Deep Learning. |
| Compiler Autotuning [13] | This study demonstrates how machine learning enhances compiler autotuning. It describes a detailed overview of traditional compiler optimization methods and their limitations by presenting the need for more data-driven and adaptive techniques. | Supervised Learning. |
| Compiler Optimizations [40] | This study introduces the Machine Learning Compiler-Bridge, which facilitates Machine Learning model development within a traditional Python framework while enabling seamless and efficient integration with an optimizing compiler. | Machine Learning. |

➢ *Formal Methods in Software Verification*

Formal methods provide mathematically rigorous techniques for specifying and verifying software systems. Among these, Stream X-Machines (SXMs) and their extension, Communicating Stream X-Machines (CSXM), have been widely used for modelling complex and distributed systems [32]. CSXM enhances finite state machines by incorporating memory and communication capabilities, enabling the modelling of systems with interacting components.

Previous studies have demonstrated the effectiveness of CSXM in system validation, conformance testing, and automated test case generation [32]. In particular, CSXM-based approaches have been successfully applied to distributed applications such as banking and ATM systems, where correctness and reliability are critical. These approaches enable early defect detection and systematic test generation based on formal specifications. However, existing work primarily focuses on system modelling and verification, with limited exploration of CSXM in compiler design or integration with intelligent anomaly detection mechanisms.

➢ *Traditional Compiler Design*

Conventional compilers are structured into multiple phases, including lexical analysis, syntax analysis, semantic analysis, optimisation, and code generation [31]. These phases are designed to ensure that source code adheres to the rules of a programming language and is translated into executable form efficiently [31]. While modern compilers are highly optimised and effective at detecting syntax and type-related errors, they are limited in their ability to identify logical or context-specific anomalies.

Research in compiler design has introduced various optimisation techniques and static analysis tools to improve code quality. However, these approaches are largely rule-based and lack adaptability. They do not learn from historical data or improve their performance over time. As software systems grow in complexity, there is an increasing need for compilers that can go beyond static checks and provide intelligent insights into code behaviour.

➢ *Machine Learning for Code Analysis*

Recent advancements in ML have led to its application in software engineering tasks, including bug detection, code completion, program repair, and anomaly detection [28]. Supervised learning models such as Random Forests have been used for classification tasks, while deep learning approaches, including Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN), have shown strong performance in learning structural and sequential patterns in source code [30].

Datasets such as DeepFix have enabled the training and evaluation of ML models on real-world programming errors and corrections [21][42]. These approaches have demonstrated high accuracy in detecting and fixing code anomalies. However, ML-based solutions often operate as standalone tools and are not tightly integrated into compiler pipelines [50]. Additionally, they rely heavily on labelled datasets, which can be costly and time-consuming to produce,

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26sep417

and may struggle to generalise across different programming languages or domains.

➢ *Hybrid Approaches and Research Gap*

There is a growing interest in combining formal methods with ML to leverage the strengths of both approaches. Formal methods provide correctness guarantees and structured modelling, while ML introduces adaptability and data-driven intelligence. However, existing hybrid approaches are largely conceptual and lack comprehensive implementation and evaluation.

In particular, there is limited research on integrating CSXM-based formal models with ML techniques within a compiler framework. Furthermore, key practical considerations such as compilation latency, ML inference overhead, and the trade-off between accuracy and performance remain underexplored.

➢ *Summary of Research Gap*

From the reviewed literature, the following gaps are identified:

- Limited application of CSXM in compiler design.
- Lack of fully implemented hybrid CSXM–ML systems.
- Insufficient experimental evaluation using real-world datasets.
- Absence of performance-focused analysis, particularly compilation latency.
- Challenges related to dataset labelling and model generalisability.

This paper addresses these gaps by proposing and evaluating a performance-aware hybrid CSXM–ML compiler that integrates formal verification with intelligent anomaly detection, supported by comprehensive experimental and performance analysis.

## III. CSXM-BASED COMPILER ARCHITECTURE

This section presents the architecture of the proposed hybrid compiler, which integrates the Communicating Stream X-Machine (CSXM) formal framework with Machine Learning (ML) components to enable intelligent code analysis. The design aims to combine the correctness guarantees of formal methods with the adaptability of data-driven models, forming a structured and extensible compilation pipeline.

➢ *Overview of the Architecture*

The proposed compiler follows a modular architecture consisting of two tightly coupled layers: the CSXM-based formal processing layer and the ML-based anomaly detection layer as displayed in figure 1. The CSXM layer governs the overall control flow of the compiler, modelling each compilation phase as a set of states and transitions with associated memory structures. The ML layer operates alongside this process, analysing intermediate representations of code to detect anomalies and provide predictive insights.

- *At a High Level, the Compiler Pipeline Includes the Following Stages:*

✓ Lexical Analysis
✓ Syntax Analysis
✓ Semantic Analysis
✓ Intermediate Representation Generation
✓ Anomaly Detection (ML Layer)
✓ Code Optimisation and Generation

Each stage is formally represented within the CSXM model, ensuring that all transitions are well-defined and verifiable.

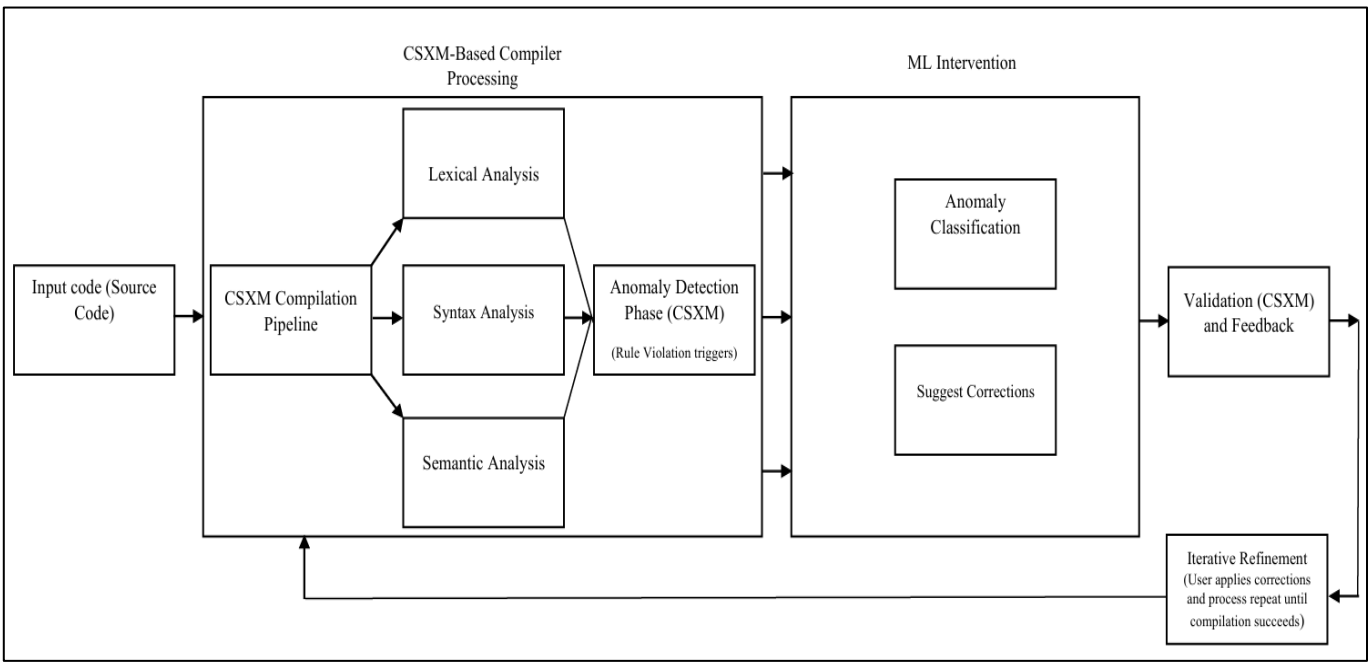


Fig 1 The Architecture of ML Integration in CSXM-Based Compiler.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26sep417

➢ *CSXM Formal Modelling of the Compiler*

The CSXM framework extends traditional finite state machines by incorporating memory and communication between components. In the proposed architecture, each compiler phase is modelled as a distinct X-Machine with its own:

- Set of states
- Input and output alphabets
- Memory structure
- Transition functions

Communication between phases is achieved through well-defined input/output streams, allowing the output of one phase to serve as the input to the next. This modular design supports both sequential and distributed execution. The memory component of the CSXM model is used to store intermediate data such as tokens, abstract syntax trees (ASTs), symbol tables, and semantic rules. Transition functions define how inputs are processed and how memory is updated at each stage. This ensures that the compiler's behaviour is formally specified and can be systematically tested and verified.

➢ *Integration of Machine Learning Components*

The ML layer is integrated into the compiler pipeline primarily at the intermediate representation stage, where sufficient structural and contextual information about the code is available. Pre-processed code features, including token sequences and syntactic structures, are extracted and passed to trained ML models. The models used in this study include:

- Random Forest (RF) for baseline classification [30].
- Convolutional Neural Networks (CNN) for structural pattern recognition [45].
- CNN-LSTM for capturing both spatial and sequential dependencies.

These models analyse the input to identify anomalies such as syntax misuse, logical inconsistencies, and uncommon coding patterns. The output of the ML layer includes anomaly classifications and suggested corrections, which are fed back into the CSXM-controlled pipeline for further processing.

➢ *Interaction between CSXM and ML Layers*

The interaction between the CSXM and ML components is designed to be non-intrusive yet effective. The CSXM layer maintains control over the compilation process, ensuring that all operations adhere to formal specifications. The ML layer acts as an auxiliary decision-support system, enhancing the compiler's ability to detect issues beyond rule-based checks. When an anomaly is detected, the ML output is incorporated into the CSXM memory and may trigger specific transitions, such as error handling routines or corrective suggestions. This interaction enables the compiler to maintain correctness while benefiting from adaptive intelligence.

➢ *Implementation Details*

The prototype of the hybrid compiler was implemented using Python for the ML components and core integration logic, while leveraging standard libraries for data preprocessing and model training. TensorFlow was used for deep learning model development, and experiments were conducted in both local environments and cloud-based platforms such as Google Colab. The CSXM model was implemented as a modular framework, where each compiler phase is represented as a callable component with defined inputs, outputs, and memory states. This design supports extensibility, allowing additional models or compiler phases to be integrated with minimal changes to the overall architecture.

➢ *Design Considerations*

Several design considerations guided the development of the proposed architecture:

- Modularity: Ensures that individual components can be independently developed and tested.
- Scalability: Supports extension to larger datasets and more complex programming languages.
- Interoperability: Allows seamless integration between formal models and ML components.
- Performance Awareness: Minimises the impact of ML inference on compilation time.

This architecture establishes the foundation for integrating formal verification with intelligent anomaly detection in a unified compiler framework. The next section describes the ML integration process in detail, including feature extraction, model training, and deployment within the compiler pipeline.

## IV. MACHINE LEARNING INTEGRATION APPROACH

This section details the integration of Machine Learning (ML) within the proposed CSXM-based compiler, focusing on feature extraction, dataset preparation, model training, and deployment within the compilation pipeline as displayed in figure 2. The objective is to enable intelligent anomaly detection while maintaining compatibility with the formal CSXM framework.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26sep417

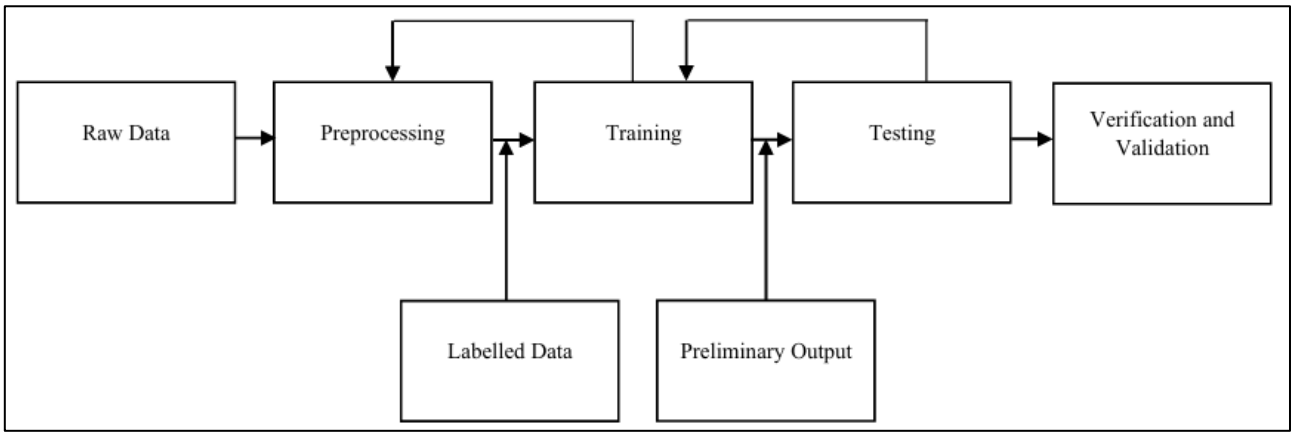


Fig 2 Conceptual Flow Diagram that Represents the Training and Testing of the Machine Learning Models Integrated into the CSXM-Based Compiler

➢ *Feature Extraction and Representation*

Effective ML-based code analysis depends on transforming source code into meaningful representations. In this study, source code is pre-processed and converted into structured inputs suitable for model training. The feature extraction process involves:

- Tokenisation: Breaking source code into lexical tokens such as keywords, identifiers, operators, and literals.
- Syntactic Structuring: Capturing structural relationships using parse trees or simplified abstract syntax representations.
- Sequence Encoding: Representing token sequences numerically using indexing and padding to ensure uniform input size.

These features provide both syntactic and contextual information, enabling the models to learn patterns associated with correct and anomalous code.

➢ *Dataset Preparation*

Two types of datasets were used to train and evaluate the ML models:

- Real-World Dataset: The DeepFix dataset, which contains erroneous code samples and their corrected versions, providing realistic examples of programming mistakes [11].
- Synthetic Dataset: Custom-generated code samples created by injecting controlled anomalies into hybrid C#–Python programs. Each sample includes metadata such as anomaly type, expected output, and code snippet.

The combined dataset improves model robustness by covering both real-world and controlled error scenarios. Data preprocessing includes cleaning, normalisation, and labelling to ensure consistency across samples.

➢ *Model Selection and Training*

Three supervised ML models were selected to evaluate different learning capabilities:

- Random Forest (RF): Used as a baseline model for classification due to its robustness and interpretability.
- Convolutional Neural Network (CNN): Applied to capture local structural patterns in code.
- CNN-LSTM: Combines convolutional layers with Long Short-Term Memory networks to model both spatial and sequential dependencies.

The dataset was split into training and testing subsets, and models were trained using standard optimisation techniques. Hyperparameter tuning was performed using grid search to optimise model performance. Training was conducted using Python 3.10 and TensorFlow, with execution on both local systems and cloud platforms.

➢ *Model Evaluation Metrics*

Model performance was evaluated using widely accepted classification metrics:

- Accuracy: Measures overall correctness of predictions.
- Precision: Evaluates the proportion of correctly identified anomalies.
- Recall: Assesses the model's ability to detect all relevant anomalies.
- F1-Score: Provides a balance between precision and recall.

These metrics provide a comprehensive assessment of the models' anomaly detection capabilities.

➢ *Integration into the Compiler Pipeline*

The trained ML models are integrated into the compiler at the intermediate representation stage. At this point, the code has been sufficiently processed to expose structural and semantic features while remaining flexible for analysis.

- *The Integration Process Involves:*

✓ Extracting features from intermediate code representations.
✓ Feeding these features into the trained ML models.
✓ Receiving anomaly predictions and confidence scores.
✓ Passing results to the CSXM layer for further action.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26sep417

- *Depending on the Output, the Compiler May:*

✓ Flag errors or warnings
✓ Suggest possible corrections
✓ Trigger specific transitions within the CSXM model

This integration ensures that ML enhances, rather than disrupts, the formal compilation process.

➢ *Deployment and Runtime Considerations*

To support real-time compilation, the ML inference process is optimised to minimise latency. Techniques such as model caching and efficient data preprocessing are used to reduce overhead. However, the inclusion of ML introduces additional computational cost, which is analysed in later sections. The system is designed to allow flexible deployment, enabling models to be updated or replaced without altering the core CSXM architecture. This supports continuous improvement and adaptability to new programming patterns.

This section establishes how ML is effectively embedded within the compiler to provide intelligent anomaly detection. The next section presents the dataset configuration and experimental setup used to evaluate the proposed system.

## V. DATASET AND EXPERIMENTAL SETUP

This section describes the datasets, experimental configuration, and procedures used to evaluate the proposed hybrid CSXM–ML compiler. The goal is to ensure reproducibility and provide a robust basis for assessing both anomaly detection performance and system behaviour.

➢ *Dataset Description*

To support comprehensive evaluation, two complementary datasets were used:

- *DeepFix Dataset (Real-World Data):*

This dataset contains real programming errors and their corresponding corrections, providing realistic examples of syntax and semantic anomalies. It serves as a benchmark for evaluating the model's ability to generalise to real-world coding patterns [11].

- *Synthetic Dataset (Custom-Generated):*

A synthetic dataset was created by injecting controlled anomalies into hybrid C#–Python code samples. These anomalies include syntax errors, logical inconsistencies, and structural deviations. Each sample is annotated with metadata such as anomaly type, expected behaviour, and code snippet.

The combination of real and synthetic datasets ensures both realism and controlled coverage of error types.

➢ *Data Preprocessing*

Before training, all datasets underwent preprocessing to ensure consistency and compatibility with ML models. The preprocessing steps include:

- Removal of irrelevant or noisy data
- Tokenisation of code into structured sequences
- Normalisation of code formats
- Encoding of tokens into numerical representations
- Labelling of each sample based on anomaly type

To maintain uniformity, sequences were padded or truncated to a fixed length. This enabled efficient batch processing during model training.

➢ *Experimental Configuration*

The experiments were conducted using Python 3.10 with TensorFlow for model development. Training and evaluation were performed on both local machines and cloud-based environments (Google Colab), allowing scalability and computational flexibility.

- *Key Configuration Details Include:*

✓ Dataset split: Training and testing subsets
✓ Hyperparameter tuning using grid search
✓ Batch processing for efficient training
✓ Multiple experimental runs to ensure consistency

A total of over 150 training and testing iterations were executed, allowing for model refinement and performance stabilisation.

➢ *Evaluation Procedure*

The evaluation process was designed to assess both the ML models and their integration within the compiler. It includes:

- Training each model using the prepared dataset
- Testing model performance on unseen data
- Integrating trained models into the compiler pipeline
- Running sample programs through the compiler
- Observing anomaly detection and correction behaviour

Both correct and intentionally faulty code samples were used to evaluate system robustness.

➢ *Evaluation Metrics*

Performance was measured using standard classification metrics:

- Accuracy
- Precision
- Recall
- F1-Score

These metrics were computed for each model to compare their effectiveness in detecting anomalies. In addition, observations were made on system behaviour, including detection consistency and reliability of suggested corrections.

➢ *Reproducibility Considerations*

To ensure reproducibility, all datasets, preprocessing steps, and model configurations were documented. The use of publicly available datasets alongside clearly defined

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26sep417

synthetic data generation processes allows the experiments to be replicated and extended in future research. This section outlines the experimental foundation for evaluating the proposed system. The next section presents the results and analysis of the hybrid compiler's anomaly detection performance.

## VI. RESULTS AND EVALUATION

This section presents the results of the experimental evaluation of the proposed hybrid CSXM–ML compiler as presented in table 1, table 2, Figure 3 and Figure 4. The analysis focuses on the anomaly detection performance of the integrated ML models and their effectiveness within the compilation pipeline.

Table 1 Performance Comparison Table for the Machine Learning Models Using Generated Dataset.

| Metrics | RF | CNN | CNN-LSTM |
|---|---|---|---|
| Accuracy (%) | 93.00 | 91.00 | 90.00 |
| Precision (%) | 91.30 | 90.20 | 88.90 |
| Recall (%) | 93.30 | 92.00 | 92.30 |
| F1-Score (%) | 92.30 | 91.10 | 90.50 |

Table 2 Performance Comparison Table for the Machine Learning Models Using DeepFix Dataset.

| Metrics | RF | CNN | CNN-LSTM |
|---|---|---|---|
| Accuracy (%) | 85.51 | 86.44 | 89.72 |
| Precision (%) | 86.54 | 85.46 | 89.00 |
| Recall (%) | 84.11 | 87.85 | 90.65 |
| F1-Score (%) | 85.31 | 86.64 | 89.82 |

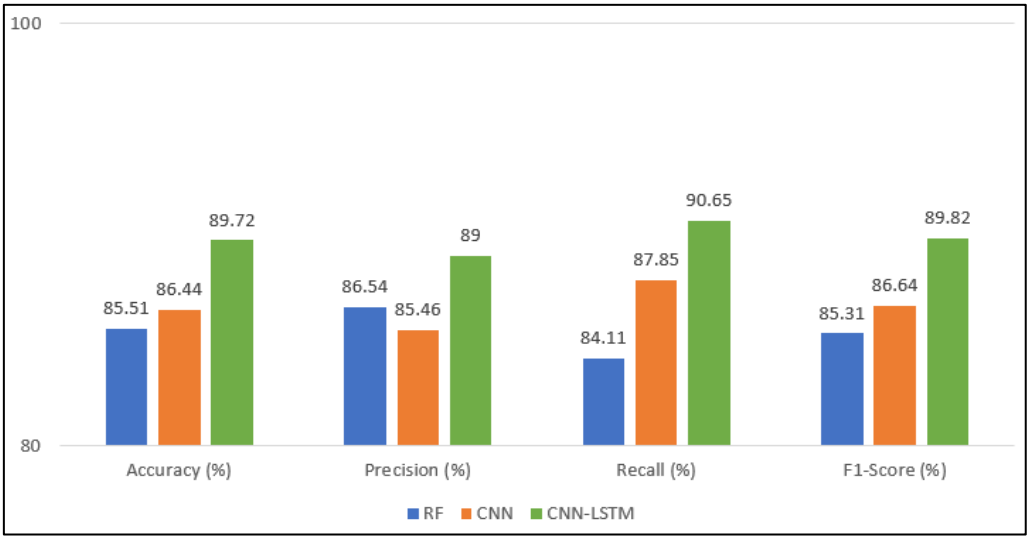


Fig 3 Performance Comparison for the Machine Learning Models Using DeepFix Dataset.

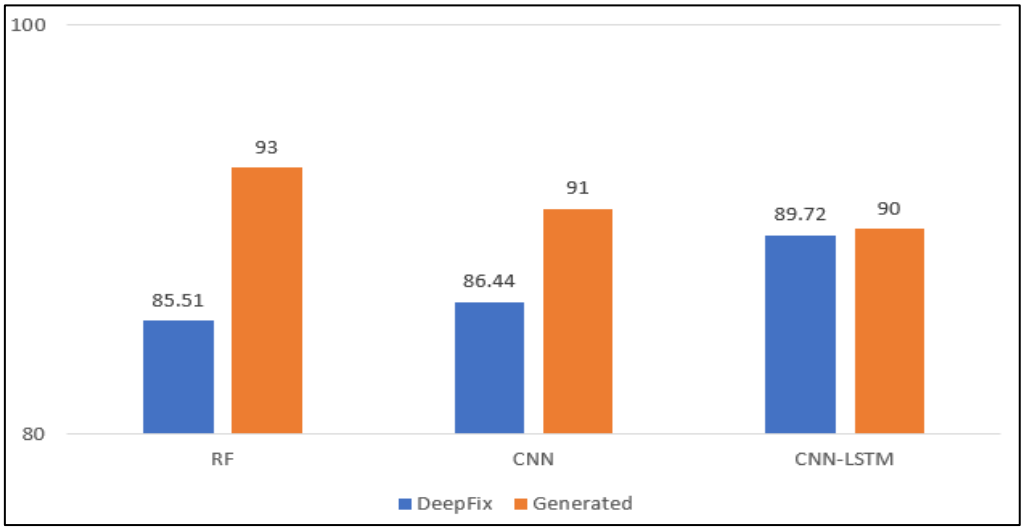


Fig 4 Bar Chart Showing the Accuracy of DeepFix Dataset and the Generated Dataset.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26sep417

➢ *Model Performance Comparison*

The three ML models; Random Forest (RF), Convolutional Neural Network (CNN), and CNN-LSTM were evaluated using the test dataset. Each model demonstrated the ability to detect code anomalies, with varying levels of performance across the evaluation metrics.

- *Random Forest (RF):*

Provided stable baseline performance with relatively fast inference time. However, its ability to capture complex structural patterns in code was limited compared to deep learning models.

- *CNN:*

Showed improved performance in identifying structural anomalies due to its ability to capture local patterns within token sequences. It achieved higher precision compared to RF.

- *CNN-LSTM:*

Delivered the best overall performance by combining spatial and sequential learning. It achieved the highest recall and F1-score, indicating strong capability in detecting diverse anomaly types.

Overall, deep learning models outperformed the traditional RF model, particularly in handling complex and context-dependent errors.

➢ *Quantitative Results*

Across the experimental runs, the models achieved consistently high performance. The CNN-LSTM model demonstrated the best balance between precision and recall, while the CNN model achieved competitive accuracy with slightly lower computational cost. RF maintained reliable but comparatively lower performance. The results confirm that incorporating sequential context significantly improves anomaly detection accuracy. The use of both real-world and synthetic datasets contributed to improved generalisability across different error types.

➢ *Integration with the Compiler Pipeline*

When integrated into the CSXM-based compiler, the ML models enhanced the system's ability to detect anomalies beyond traditional rule-based checks. The compiler successfully identified:

- Syntax-related anomalies missed during early parsing stages
- Logical inconsistencies not captured by semantic rules
- Uncommon or irregular coding patterns

The ML outputs were effectively incorporated into the CSXM framework, enabling structured handling of detected anomalies through predefined transitions.

➢ *System Behaviour and Reliability*

The hybrid compiler demonstrated consistent behaviour across multiple test scenarios. Key observations include:

- Accurate detection of known anomaly patterns
- Stable performance across repeated runs
- Effective interaction between CSXM control flow and ML predictions

However, some false positives and false negatives were observed, particularly in edge cases involving ambiguous or previously unseen code patterns. These cases highlight the importance of dataset diversity and model generalisation.

➢ *Comparative Insights*

The evaluation highlights several important insights:

- Deep learning models significantly enhance anomaly detection capability.
- Sequential modelling (CNN-LSTM) is particularly effective for code analysis.
- Combining formal methods with ML provides both reliability and adaptability.

While the results demonstrate clear improvements in detection accuracy, they also reveal the need to consider computational overhead, which is examined in the next section. This section confirms the effectiveness of the proposed hybrid approach in improving compiler intelligence. The next section provides a detailed analysis of performance, including compilation latency and ML inference overhead.

## VII. PERFORMANCE ANALYSIS AND DISCUSSIONS

This section presents a detailed analysis of the performance implications of integrating Machine Learning (ML) into the CSXM-based compiler. While previous sections demonstrated improvements in anomaly detection accuracy, this section evaluates the associated computational cost, focusing on compilation latency, ML inference overhead, and practical deployment considerations.

➢ *Compilation Latency Analysis*

The integration of ML models introduces additional processing stages within the compilation pipeline, resulting in increased compilation time. This latency primarily arises from feature extraction, data preprocessing, and model inference. Experimental observations indicate that:

- Random Forest (RF) introduces minimal latency due to its lightweight structure.
- CNN incurs moderate latency, particularly during feature transformation and convolution operations.
- CNN-LSTM introduces the highest latency due to sequential processing and increased model complexity.

Although the latency increase is measurable, it remains within acceptable bounds for offline or batch compilation scenarios. However, for real-time or interactive environments, such delays may impact usability.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26sep417

➢ *ML Inference Overhead*

ML inference overhead is a critical factor affecting system performance. The complexity of deep learning models directly influences execution time and resource consumption. Key observations include:

- Increased memory usage for deep learning models compared to traditional methods.
- Longer inference times for models with sequential dependencies (e.g., CNN-LSTM).
- Trade-off between model complexity and detection accuracy.

Despite these overheads, the improved anomaly detection capability justifies the integration, particularly in scenarios where correctness and code quality are prioritised over speed.

➢ *Accuracy vs Performance Trade-off*

A central finding of this study is the trade-off between detection accuracy and computational efficiency:

- RF offers faster execution with lower accuracy.
- CNN provides a balance between performance and accuracy.
- CNN-LSTM achieves the highest accuracy at the cost of increased latency.

This trade-off suggests that model selection should be context-dependent. For example, lightweight models may be preferred in time-sensitive environments, while more complex models are suitable for thorough offline analysis.

➢ *Dataset Labelling Effort*

The effectiveness of supervised ML models depends heavily on the availability of labelled datasets. In this study, significant effort was required to:

- Annotate synthetic data with anomaly types
- Validate labels for consistency and correctness
- Align real-world and synthetic datasets

This labelling process is time-consuming and may limit scalability. Additionally, manual labelling introduces the risk of inconsistencies, which can affect model performance. Future work may explore semi-supervised or unsupervised approaches to reduce this dependency.

➢ *Generalisability and Scalability*

While the models performed well on the evaluated datasets, their ability to generalise to unseen codebases remains a challenge. Differences in programming styles, languages, and complexity levels can affect model performance. Scalability considerations include:

- Handling larger and more diverse datasets
- Extending support to multiple programming languages
- Maintaining performance efficiency as system complexity grows

Addressing these challenges is essential for real-world adoption of hybrid compiler systems.

➢ *Discussions*

The results demonstrate that integrating ML into a formally defined compiler significantly enhances anomaly detection capabilities. However, this improvement comes at the cost of increased computational overhead and data preparation effort. The CSXM framework plays a crucial role in maintaining system correctness and structured control, ensuring that the integration of ML does not compromise reliability. Overall, the findings highlight the importance of a performance-aware approach when designing intelligent compilers. Balancing accuracy, efficiency, and scalability is key to achieving practical and effective deployment. This section provides critical insights into the trade-offs and challenges of the proposed system. The final section concludes the paper and outlines directions for future research.

## VIII. CONCLUSION AND FUTURE WORK

This paper presented a performance-aware hybrid compiler that integrates the Communicating Stream X-Machine (CSXM) formal framework with Machine Learning (ML) techniques to enable intelligent code analysis and anomaly detection. The proposed approach combines the strengths of formal verification and data-driven learning, resulting in a system that enhances traditional compilation processes with adaptive capabilities. The implementation and experimental evaluation demonstrated that ML models, particularly deep learning approaches such as CNN and CNN-LSTM, significantly improve anomaly detection accuracy compared to traditional methods. The integration within the CSXM framework ensured that these enhancements were achieved without compromising the structural correctness and reliability of the compiler. The use of both real-world and synthetic datasets further validated the robustness of the approach.

A key contribution of this work is the performance analysis of the hybrid system. The results showed that while ML integration improves detection capability, it introduces additional compilation latency and computational overhead. The study highlighted the trade-off between accuracy and efficiency, emphasising the need for context-aware model selection depending on application requirements. Furthermore, practical challenges such as dataset labelling effort and model generalisability were identified as important considerations for scalability.

Future work will focus on optimising the ML inference pipeline to reduce latency and improve real-time performance. Additionally, exploring semi-supervised and unsupervised learning approaches may reduce reliance on labelled datasets. Extending the framework to support multiple programming languages and more complex codebases will further enhance its applicability. Finally, incorporating explainable AI techniques could improve transparency and trust in ML-driven compiler decisions. Overall, this research establishes a strong foundation for the

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26sep417

development of intelligent, adaptive compilers that effectively balance correctness, performance, and learning capability.

➢ *Funding Declaration*
This research received no funding.

➢ *Data Availability*

The datasets used and analyzed during this study are available from public repositories, specifically the DeepFix dataset, which is accessible via its original publication and GitHub repository. Additional synthetic datasets generated during the study are available from the corresponding author on reasonable requests as discussed in the manuscript.

➢ *Author Contributions*

The authors conceptualized and designed the research study, conducted the implementation and integration of Machine Learning models into the CSXM-based compiler framework, performed all experimental evaluations using both generated and public datasets, analyzed the results, and wrote the manuscript. The authors also reviewed related literature, prepared the figures and tables, and finalized the article for submission. All work presented is the sole contribution of the authors.

➢ *Competing Interests*

The authors declares that there are no competing interests related to the publication of this manuscript.

## REFERENCES

[1]. Akhtar, N. and Missen, M.M (2014). Contribution to the Formal Specification and Verification of a Multi-Agent Robotic System. *European Journal of Scientific Research*. ISSN 1450-216X / 1450-202X Vol.117 No.1. Page 35-55.

[2]. Barchi, F., Parisi, E., Urgese, G., Ficarra, E. and Acquaviva, A. (2020). Exploration of Convolutional Neural Network Models for Source Code Classification. Engineering Applications of Artificial Intelligence. https://doi.org/10.1016/j.engappai.2020.104075

[3]. Bhowmick, K. and Narvekar, M. (2019). A Comprehensive Study and Analysis of Semi-Supervised Learning Techniques. *International Journal of Engineering Research and Technology (IJERT)*. Vol. 8. Issue 11. Page 810-816.

[4]. Chaflekar, S. H., Lokhande, A., Gomase, P. and Shinganjude, R. (2017). Compiler Architecture and Design Issues. *International Journal of Computer Science Trends and Technology (IJCST)*. Volume 5. Issue 3.

[5]. Chen, J., Hu, W., Hao, D., Xiong, Y., Zhang, H. Zhang, L. and Xie, B. (2016). An Empirical Comparison of Compiler Testing Techniques. *In Proceedings of the 38th International Conference on Software Engineering*. Page 180-190.

[6]. Chen, Z., Bihuan, C., Linlin, C., Xin, P., and Wenyun, Z. (2019a). A Large-Scale Empirical Stud of Compiler Errors in Continuous Integration. *In Proceedings of the 27th ACM Joint European Software Engineering Confereence and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*. Tallinn, Estonia. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338906.3338917

[7]. Chen, R., Wang, X., Zhang, W., Zhu, X., Li, A., and Yang, C. (2019b). A hybrid CNN-LSTM model for typhoon formation forecasting, *GeoInformatica*, 23, 375–396, 2019.

[8]. Dehaerne, E., Dey, B., Gendt, S. D. and Meert, W. (2022). Code Generation Using Machine Learning: A Systematic Review. *IEEE Access*. DOI: 10.1109/ACCESS.2022.3196347.

[9]. Fatemehalsadat, M., Karem, C., Rachid, L., Saeid, H., Yves, G., and Simon, T. (2020). Convolutional neural network and long short-term memory models for ice-jam predictions. The Cryosphere, 16, 1447–1468, 2022 https://doi.org/10.5194/tc-16-1447-2022.

[10]. Goerigk, W., Dold, A., Gaul, T., Goos, G., Heberle, A. Henke, F. W., Hoffmann, U., Langmaack, H., Pfeifer, H., Ruess, H. and Zimmermann, W. (2015). Compiler Correctness and Implementation Verification: The Verifix Approach. *ResearchGate*. https://www.researchgate.net/publication/2274826.

[11]. Gupta, R., Pal, S., Kanade, A. and Shevadem S. (2017). DeepFix: Fixing Common C language Errors by Deep Learning. AAAI 2017.

[12]. Hanifeng, W., Hengyuan, L. Zheng, L., Yong, L., Fuxiang, S., and Xiang, C. (2022). A Token-Based Compilation Error Categorization and Its Applications. *J Softw Evol Proc*. 2022; e2512. wileyonlinelibrary.com/journal/smr. © 2022 John Wiley & Sons Ltd. https://doi.org/10.100

[13]. Hela, B. K. (2024). Intelligent Compilers: Machine Learning-Powered Compiler Autotuning. *Medium*. https://itnext.io/intelligent-compilers-machine-learning-powered-compiler-autotuning-929ae8fe407f.

[14]. Jayatilaka, T., Ueno, H., Georgakoudis, G., Park, E. and Doerfert, J. (2021). Towards Compile-Time-Reducing Compiler Optimization Selection via Machine learning. ICPP Workshops '21, Lemont, IL, USA. ACM ISBN 978-1-4503-XXXX-X/18/06. https://doi.org/10.1145/1122445.1122456

[15]. Junjie, C., Wenxiang, H., Dan, H., Yingfei, X., Hongyu, Z., Lu, Z., and Bind, X. (2016). An Empirical Comparison of Compiler Testing Technique. *ICSE '16*, Austin, TX, USA. http://dx.doi.org/10.1145/2884781.2884878

[16]. Li, X., Zhang, Y., Zhang, J., Chen, S., Marsic, I., Farneth, R. A., and Burd, R. S. (2017). Concurrent activity recognition with multimodal CNN-LSTM structure, *arXiv*:1702.01638.

[17]. Luan, Y. and Lin, S. (2019). Research on text classification based on CNN and LSTM, in: *International Conference on Artificial Intelligence and Computer Applications*, Dalian, China, 352–355, https://doi.org/10.1109/ICAICA.2019.8873454.

[18]. Lu, N., Wu, Y., Feng, L., and Song, J. (2018). Deep learning for fall detection: Three-dimensional CNN combined with LSTM on video kinematic data, *IEEE* J. Biomed. Health, 23, 314–323.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26sep417

[19]. Maas, M, Andersen, D. G., Isard, M. Javanmard, M. M., Mckinley, K. S. and Colin, R. (2020). Combining Machine Learning and Lifetime-Based Resource Management for Memory Allocation and Beyond. *In Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems, 2020*. DOI:10.1145/3611018.

[20]. Mesbah, A., Rice, A., Aftandilian, E., Johnston, E., and Glorioso, N. (2019). DeepDelta: Learning to Repair Compilation Errors. *In Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, NewYork, NY, USA, 12pages. https://doi.org/10.1145/nnnnnnn.

[21]. Monsifrot, A., Bodin, F. and Quiniou, R. (2002). A Machine Learning Approach to Automatic Production of Compiler Heuristics. *In Proceedings of the International Conference on Artificial Intelligence. Methodol. Syst. Appl*. Page 41- 50.

[22]. Mutegeki, R. and Han, D. S. (2020). A CNN-LSTM approach to human activity recognition, in: *International Conference on Artificial Intelligence in Information and Communication*, Fukuoka, Japan, 362–366, https://doi.org/10.1109/ICAIIC48513.2020.9065078.

[23]. Ogunshile E. (2011). A Machine with Class: A Framework for Object Generation, Intergration and Language Authentication (FROGILA). Ph.D. Thesis. The University of Sheffield. United Kingdom.

[24]. Oh, S. L., Ng, E. Y., San Tan, R., and Acharya, U. R. (2018). Automated diagnosis of arrhythmia using combination of CNN and LSTM techniques with variable length heart beats, *Comput. Biol. Med*., 102, 278–287.

[25]. Ombabi, A. H., Ouarda, W., and Alimi, A. M. (2020). Deep learning CNN–LSTM framework for Arabic sentiment analysis using textual information shared in social networks, Social Network Analysis and Mining, 10, 1–13, https://doi.org/https://doi.org/10.1007/s13278-020-00668-1.

[26]. Ordóñez, F. J. and Roggen, D. (2016). Deep convolutional and LSTM recurrent neural networks for multimodal wearable activity recognition, Sensors, 16, 115–140 https://doi.org/10.3390/s16010115.

[27]. Patterson, D. and Ahmed, A. (2019). The Next 700 Compiler Correctness Theorems (Function Pearl). Proc. ACM Program. Lang. 3, ICFP, Article 85. Page 1-29. https://doi.org/10.1145/3341689

[28]. Pizzolotto, D. and Inoue, K. (2021). Identifying Compiler and Optimization Level in Binary Code from Multiple Architectures. *IEEE Access Digital Object Identifier 10.1109/ACCESS.2021.3132950*. Volume 9. Page 1- 15.

[29]. Raphael, M., David, L., Wolfgang, K., Lukas Stadler, and Hanspeter, M. (2022). Machine-Learning-Based Self-Optimizing Compiler Heuristics. *In Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes (MPLR '22)*. Brussels, Belgium. ACM, New York, NY, USA, 14 pages. https: //doi.org/10.1145/3546918.3546921

[30]. Sanusi, B. A., Olabiyisi, S. O., Olowoye, A. O. and Olatunji, B. L. (2019). Software Defect Prediction System using Machine Learning based Algorithms. *Journal of Advances in Computational Intelligence Theory*, 1(3), 1–9. http://doi.org/10.5281/zenodo.35908 41

[31]. Sanusi, B. A., Ogunshile, E., Aydin, M., and Olabiyisi, S. O. (2024). Assuring Correctness, Testing, and Verification of X-Compiler by Integrating Communicating Stream X-Machine. *In 2024 6th International Conference on Software Engineering and Development (ICSED 2024)*, May 29–31, 2024, Hong Kong, Hong Kong. ACM, New York, NY, USA, 9 pages.

[32]. Sanusi B. A., Ogunshile E. K. A., Aydin, M. E., Olabiyisi, S. O. and Oyediran M. O. (2022). Development of Communicating Stream X-Machine Tool for Modeling and Generating Test Cases for Automated Teller Machine. 9th International Conference on Computer Science and Information Technology (CSIT 2022), DOI: 10.5121/csit.2022.121407 pp 77-90.

[33]. She, X. and Zhang, D. (2018). Text classification based on hybrid CNN-LSTM hybrid model, in: *11th International Symposium on Computational Intelligence and Design*, 185–189, https://doi.org/10.1109/ISCID.2018.10144.

[34]. Shi, R., Hu, J., and Lin, B. (2023). Mining on Students' Execution Logs and Repairing Compilation Errors Based on Deep Learning. *Appl. Sci.* 2023, *13*, 9933. https://doi.org/10.3390/app13179933

[35]. Shreyas, M., Siddarth, S., and Karmel, A. (2021). Memory Utilization and Machine Learning Techniques for Compiler Optimization. *ITM Web of Conferences* 37, 01021 (2021) https://doi.org/10.1051/itmconf/20213701021.

[36]. Simon, A., Deo, M. S., Venkatesan, S., and Babu, R. (2015). An Overview of Machine Learning and It's Applications. *International Journal of Electrical Sciences & Engineering (IJESE)*. Volume 1. Issue 1. Page 22-24.

[37]. Sosa, P. M.(2017). Twitter sentiment analysis using combined LSTM CNNmodels, Eprint Arxiv, 1–9.

[38]. Sun, C., Le, V., Su, Z.(2016). Finding and analyzing compiler warning defects. *In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016* .Austin, TX, USA, May 14-22, 2016. pp. 203–213. ACM (2016). https://doi.org/10.1145/2884781.2884879

[39]. Tang, Y., Jiang, H., Zhou, Z., Li, X., Ren, Z. and Kong, W. (2020). Detecting Compiler Warning Defects Via Diversity-Guided Program Mutation. IEEE Transactions on Software Engineering. DOI: 10.1109/TSE.2021.3119186

[40]. Umer, M., Imtiaz, Z., Ullah, S., Mehmood, A., Choi, G. S., and On, B. W. (2020). Fake news stance detection using deep learning architecture (cnn-LSTM), *IEEE Access*, 8, 156695–156706.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26sep417

[41]. VenkataKeerthy, S., Jain, S., Kalvakuntla, U., Gorantla, P. S., Chitale, R. S., Brevdo, E., Cohen, A., Trofin, M., and Upadrasta, R. (2024). The Next 700 ML-Enabled Compiler Optimizations. *In Proceedings of the 33rd ACMSIGPLAN International Conference on Compiler Construction (CC '24)*, March 2–3, 2024, Edinburgh, United Kingdom. ACM, New York, NY, USA, 12 pages. h ps://doi.org/10.1145/3640537.3641580

[42]. Wang, J., Yu, L. C., Lai, K. R., and Zhang, X.(2016b). Dimensional sentiment analysis using a regional CNN-LSTM model, in: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, 225–230, Berlin, Germany, https://doi.org/10.18653/v1/P16-2037.

[43]. Wang, Z. and O'Boyle, M. (2018). Machine Learning in Compiler Optimization. *Proceedings of the IEEE*. Vol. 106. No. 11. Page 1879 – 1901. https://doi.org/10.1109/JPROC.2018.2817118

[44]. Wang, J., Yu, L. C., Lai, K. R., and Zhang, X. (2019). Tree-structured regional CNN-LSTM model for dimensional sentiment analysis, *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 28, 581–591.

[45]. Wu, Z., Wang, X., Jiang, Y. G., Ye, H., and Xue, X. (2015). Modeling spatial-temporal clues in a hybrid deep learning framework for video classification, in: *23rd ACM International Conference on Multimedia*, Brisbane Australia, 461–470, https://doi.org/10.1145/2733373.2806222.

[46]. Yamashita, R., Nishio, M., Do, R. K. and Togashi, K. (2018). Convolutional Neural Networks: An Overview and Application in radiology. Insights into Imaging (2018) 9:611–629 https://doi.org/10.1007/s13244-018-0639-9

[47]. Yang, X., Chen, Y., Eide, E. and Regehr, J. (2011). Finding and Understanding Bugs in C Compilers. *In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. Page 283–294.

[48]. Yang, S., Shi, Z., and Zhang, G. (2023). Understand Code Style: Efficient CNN-based Compiler Optimization Recognition System. ArXiv:2302.04666v1 [cs.PL].

[49]. Yacine, H., Riyadh, B., and Yacine, C. (2023). A Hybrid Machine Learning Model for Code Optimization. *International Journal of Parallel Programming*. 51:309–331https://doi.org/10.1007/s10766-023-00758-5.

[50]. Yixuan, T., He, J., Zhide, Z., Xiaochen, L., and Zhilei, R. (2021). Detecting Compiler Warning Defects Via Diversity-Guided Program Mutation. *IEEE Transactions on Software Engineering*. 0098-5589. Page 1-12.

[51]. Zheng, W. and Michael, O. (2018). Machine Learning in Compiler Optimization. arXiv:1805.0344lv1.