

An Adaptive AI-Driven Code Optimization Framework Using Machine Learning and Explainable AI for Scalable Software Systems

Kavya B K^{1*}, S.Annamalai², A Suresh Kumar³

Research Scholar, Department of Computer Science and Engineering,
Jain (Deemed-to-be University), Bangalore, Karnataka, India¹

Associate Professor, Department of Computer Science and Engineering,
Jain (Deemed-to-be University), Bangalore, Karnataka, India²

Professor, Department of Computer Science and Engineering, Jain (Deemed-to-be University), Bangalore, Karnataka, India³

Abstract—The increasing complexity of embedded computing systems has made it difficult to achieve efficient code execution while effectively utilizing limited hardware resources. Conventional optimization methods mainly rely on compiler-driven techniques and fixed transformation rules, which may not adapt well to different embedded architectures and varying workload conditions. This research introduces a hardware-aware AI-driven code optimization framework that integrates Machine Learning (ML) with Explainable Artificial Intelligence (XAI) to improve the efficiency and scalability of embedded software systems. The framework analyzes software characteristics together with hardware-related parameters, including processor usage, memory requirements, computational workload, and resource constraints, to identify performance limitations and select suitable optimization strategies. A machine learning model is employed to predict appropriate code transformation techniques based on application behaviour and underlying hardware conditions. The XAI module further analyzes the optimization process by identifying the contribution of individual features, providing clearer explanations for the selected optimization decisions. Experimental results show that the proposed framework reduces execution time by up to 30%, decreases memory consumption by 25%, and lowers CPU utilization by approximately 15% while maintaining performance as code complexity increases. These findings demonstrate the capability of the framework to support efficient and transparent optimization of embedded software under resource-constrained computing conditions.

Keywords—Hardware-aware optimization; artificial intelligence; code optimization; machine learning; explainable AI; scalability

I. INTRODUCTION

The increased size of large-scale software systems and high-performance applications has posed a serious problem of how to maximize the efficiency of code, according to the continually growing requirements of the new computer world. With the advancement of software systems, the demand for an effective code optimization tool that can not only enhance the level of computational capabilities but also optimise the use of resources has never been more imperative [1]. The traditional code optimization techniques, which are mostly static in nature discipline specialise in a single area of performance with regard

to code execution time or memory [2]. Although these methods have been found effective in better-controlled environments, they are at times inadequate in dealing with the dynamic nature of modern software systems, which need to perform effectively in a variety of platforms, workloads and usage scenarios [3]. Limitations of the use of traditional and optimisation methods are also evident in solving the disparate and dynamic demands of contemporary apps [4]. To take some examples, in traditional systems that do not support environmental variability (hardware differences) or dynamic runtime variability, the performance may not be optimal when applied into practice [5]. Moreover, most of the current methods cannot keep up with the size and complexity of the contemporary distributed systems or cloud-based systems where the amount of data and the multitude of tasks are likely to create new bottlenecks that cannot be scaled or forecasted by the existing models [6]. These constraints demonstrate the necessity to approach the optimization of codes more dynamically and flexibly [7]. To address these difficulties, it is already stated that there is growing interest in adaptive, scalable, and interpretable optimization models that could offer more beneficial and resilient answers. The main benefit of these types of models is that they are able to keep learning based on the runtime data and adjust optimization strategies based on real-time evidence [8]. Additionally, scalability will be crucial given the changes the software systems will experience in order to accommodate bigger datasets, increased concurrent usage, and complexity of the codebase [9]. Hence, it is of utmost importance to create a framework that codes can be optimized as well as can scale as with the development of the system, and can adjust to evolving needs. The combination of Artificial Intelligence (AI) and Explainable AI (XAI) in contemporary software engineering is a prospective approach to tackling the limitations of conventional optimization methods [10]. Code analysis can be made automatic, and machine learning algorithms can spot areas of performance bottlenecks where improvements can be made and suggest optimization strategies in recommendations based on historical data, using AI [11]. In addition, XAI will introduce a degree of transparency to the optimization procedure, as it can give the reasons why the AI models made a decision that can be read and understood by humans [12]. This interpretability is paramount since it enables developers to know the reasons behind some optimization strategies being taken, and this leads to the recommendations

*Corresponding author.

being easier to believe in, so that it becomes easier to implement [13]. This research aims to establish a scalable and efficient code optimization framework that targets enhancing the effectiveness of the computations and the use of resources. This framework utilizes the strength of AI to proceed to dynamically optimise code within a range of software systems, as well as being scalable and interpretable by XAI. The proposed framework contributes to intelligent software engineering by enabling automated and hardware-aware code optimization. It supports the development of efficient embedded systems with reduced computational resource requirements, benefiting applications such as IoT, healthcare devices, autonomous systems, and smart technologies. The research also promotes sustainable computing by improving system efficiency and resource utilization.

The remainder of this manuscript is organized as follows: Section II presents the literature review, Section III describes the proposed framework architecture, and Section IV introduces the mathematical model for optimization. Section V discusses the algorithms, and Section VI presents the implementation details, while Section VII presents the experimental evaluation. Section VIII discusses the results, and Section IX concludes the study with future research directions.

II. LITERATURE REVIEW

A. Traditional Code Optimization Techniques

Conventional optimization techniques that are used in the optimization of codes focus mainly on compiler-based techniques, which entail loop unrolling, dead code elimination, and inlining [14]. These methods are directed to enhance performance through alteration of the code during the

compilation stage [15]. Loop unrolling is used to minimize the overhead of loop control, and dead code elimination is used to eradicate code that never gets executed to produce output, hence unnecessary computations. Inlining is the replacement of function calls by the body of the functions in order to get rid of the performance cost of calling functions [16]. Such optimizations are adapted to the relatively simple cases of applications whose complexity in code and environmental effects are relatively constant. The shortcomings of the traditional static optimization techniques are, however, immediately evident in cases of more dynamic and large-scale systems. The solution of static optimization does not provide the flexibility in response to the change in system conditions, hardware, or different loads [17]. As an example, performance increases obtained by loop unrolling might not be realized with the same effectiveness in systems with a mixed hardware configuration, or they cannot be scaled effectively using multi-threaded systems [18]. As a result of diversified and unpredictable workloads, dynamic optimization tools that adapt the system at runtime to real-time data and changing operating conditions have become more crucial for modern computing applications. Conventional optimization techniques mainly rely on compiler-based transformations and prior optimization rules that typically offer restricted flexibility across various software architectures and runtime scenarios. While machine learning-based optimization methods have improved automated decision-making, it is still challenging to generalize these models even with large datasets, and furthermore difficult to build interpretable models of complex software systems that can scale easily. A comparison of existing approaches without AI-based code optimization, along with their drawbacks, is shown in Table I.

TABLE I. COMPARATIVE ANALYSIS OF EXISTING AI-BASED CODE OPTIMIZATION APPROACHES

Study Approach	Methodology	Key Contributions	Limitations
Traditional Compiler Optimization	Rule-based transformations, static analysis, heuristic optimization	Provides reliable optimization techniques such as loop transformation, dead code elimination, and instruction scheduling	Limited adaptability for complex software systems; optimization decisions are predefined and lack learning capability
Reinforcement Learning-Based Code Optimization	RL models for compiler parameter tuning and optimization sequence selection	Enables automated exploration of optimization strategies and improves optimization decisions through feedback	Requires extensive training data and computational resources; optimization decisions are difficult to interpret
Machine Learning-Based Performance Prediction	Supervised learning models using software metrics and execution features	Predicts performance bottlenecks and identifies optimization opportunities using learned patterns	Performance depends strongly on feature quality and training dataset diversity
Deep Learning-Based Program Optimization	Neural networks for code representation learning and optimization prediction	Captures complex relationships among program structures and execution behavior	High computational complexity and limited explainability reduce practical adoption
AI-Assisted Code Transformation	Automated generation and modification of optimized code segments	Reduces developer effort by suggesting optimization actions	Generated transformations may not always maintain optimal performance across different applications
Explainable AI-Based Software Analysis	XAI methods such as feature importance analysis and model interpretation	Improves transparency of AI decisions and supports developer trust	Explainability mechanisms introduce additional computational overhead

1) *AI and machine learning in code optimization:* The development of Artificial Intelligence (AI) and Machine Learning (ML) has provided new opportunities in code optimization. One such use is in ML-based performance prediction, a machine learning model trained to estimate the time required to execute it, the amount of memory required to execute it, or other performance measurements [19]. Recent

studies have explored reinforcement learning-based approaches for code optimization by learning optimization decisions through interaction with the execution environment and feedback signals. However, several other AI-based techniques, including supervised machine learning, deep learning, and evolutionary optimization methods, have also been investigated for improving software performance under different

optimization objectives [20]. This will allow adaptive optimization, especially in dynamic systems where the performance requirements change.

2) *Scalability and resource-aware optimization*: Scalability and optimization of resources based on resources have become important issues as software systems increase in size and complexity. The distributed and cloud systems performance tuning needs dynamic approaches which will be able to adapt to environmental variability, most of which is usually unpredictable [21]. Local machine optimizations might fail in a distributed system with much latency and bandwidth variability, as well as resource allocation. Such an environment as clouds is characterized by the varying availability of resources and therefore the need to optimize its resources over time and with each change. Additionally, optimization soon with a focus on energy efficiency and memory consciousness. The diverse landscape of computing platforms, spanning from resource-constrained mobile and edge devices to high-performance computing clusters, has made clear the need for optimization techniques that can trade off energy consumption

and memory utilization while preserving computational performance. The energy-efficient optimization approaches focus on improving resource utilization with acceptable execution quality, in particular for embedded and edge computing environments where hardware limitations are high. While prior work showed that AI-based optimization methods enable software performance improvements, several challenges have yet to be overcome. The existing machine learning-based approaches predominantly address optimization accuracy or execution improvement, but do not effectively account for the aspects of resource constraints, adaptability, and scalability. Additionally, a majority of AI-based optimization frameworks are black-box models that do not explain the rationale behind chosen optimization actions to developers. The absence of visible mechanisms for decision-making reduces users' trust and accounts for possible adoption failures in software systems that are required to be reliable. Table II presents the identified research gaps in existing approaches and highlights the contributions and novelty of the proposed adaptive AI-driven framework.

TABLE II. RESEARCH GAP ANALYSIS AND PROPOSED FRAMEWORK CONTRIBUTION

Research Aspect	Existing Approaches	Identified Research Gap	Proposed Framework Contribution
Optimization Adaptability	Most approaches depend on fixed rules or predefined optimization strategies	Limited capability to adapt optimization decisions according to changing software characteristics	Develops an adaptive AI-driven optimization mechanism using ML-based decision prediction
Performance Bottleneck Identification	Existing methods mainly rely on manual profiling or static analysis	Inefficient identification of complex performance issues in large software systems	Uses machine learning models to predict optimization opportunities from code characteristics
Resource Optimization	Many studies focus mainly on execution speed improvement	Limited consideration of memory consumption and CPU utilization during optimization	Incorporates resource-aware optimization objectives for balanced performance improvement
Scalability	Existing optimization methods often evaluate small benchmark programs	Limited validation for large and complex software systems	Introduces scalability analysis to evaluate performance consistency with increasing code complexity
Optimization Transparency	Many AI-based approaches provide black-box optimization decisions	Lack of explanation reduces developer confidence and adoption	Integrates Explainable AI to provide interpretable optimization decisions
Generalization Capability	Models are often trained for specific programming environments or datasets	Difficulty applying optimization models across diverse software systems	Provides an adaptive framework designed for broader software environments

3) *Research gap*: Although existing studies have confirmed the potential of AI techniques for software optimization, current approaches still lack a comprehensive framework that simultaneously addresses adaptability, resource awareness, scalability, and interpretability. Most of the existing optimization methods tend to either improve upon a very specific performance metric or provide automated recommendations on what ~should-be-done to optimize without thorough information on the drivers behind the selected optimization choices. Additionally, the use of optimization frameworks that dynamically adapt to different characteristics in each software while preserving transparency and interpretability has been less explored. This research proposes a new adaptive AI-powered code optimization framework incorporating Machine Learning and Explainable AI for large-

scale software systems to overcome these limitations. The proposed framework is distinct from existing work in that it synergistically integrates performance bottleneck identification, intelligent optimization strategy selection, resource-aware evaluation, scalability analysis, and explainable decision support within a single architecture. The integrated approach not only allows a focus on efficient optimization but also increases the transparency, reliability, and practicality of AI-assisted code optimization in complex computing environments.

III. PROPOSED FRAMEWORK ARCHITECTURE

1) *Overall framework design*: The proposed framework is set to automate and optimize the code optimization process via the incorporation of an integrated workflow of steps.

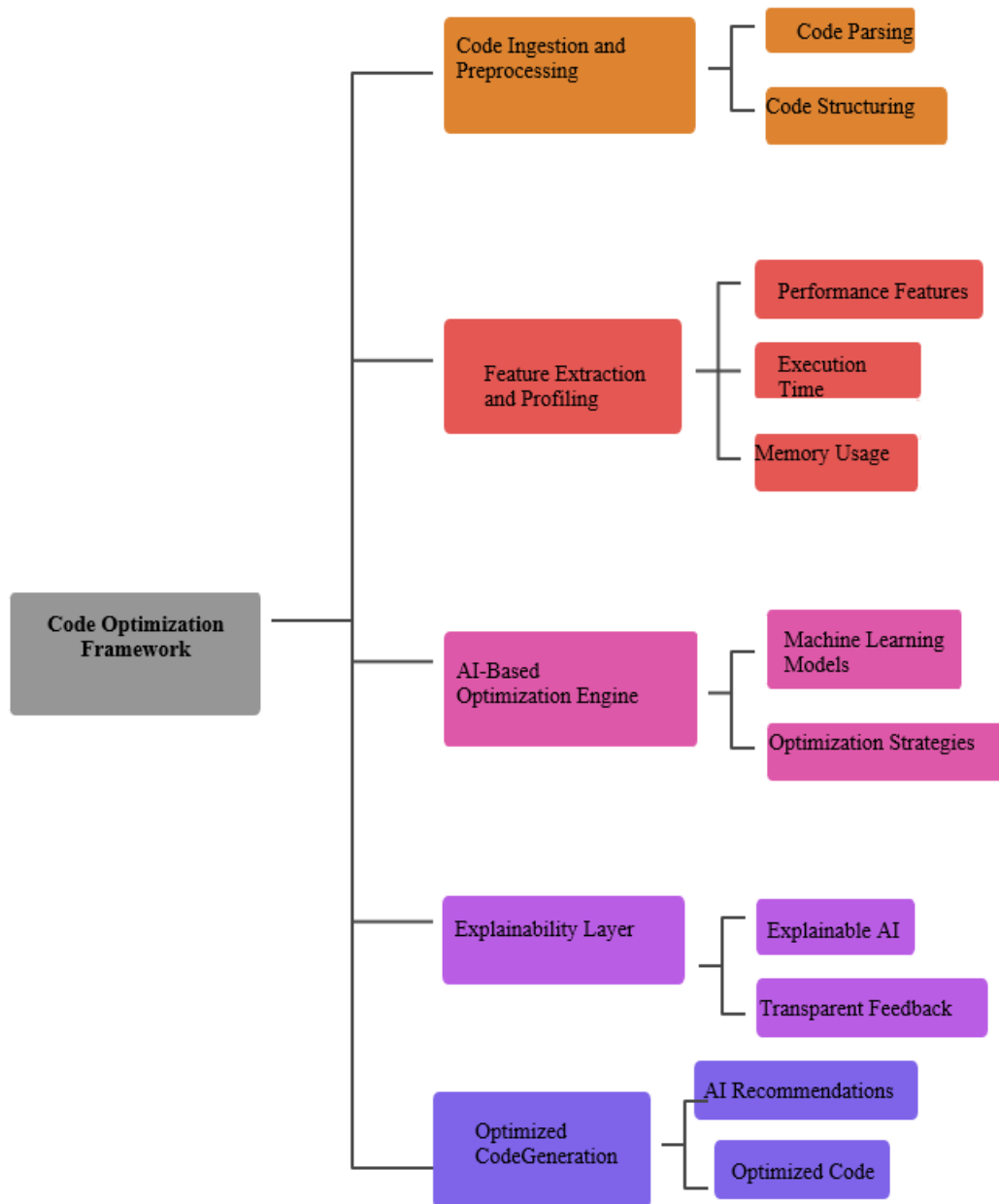


Fig. 1. Proposed system framework design.

The code ingestion and preprocessing stage represents the initial phase of the proposed framework, where source code is analyzed and transformed into a structured representation through parsing and normalization techniques. This transforms input heterogeneous code into a uniform code format to facilitate feature extraction and performance assessment. In the feature extraction and profiling stage, following the preprocessing, this step determines key performance indicators derived from the structure of our program as well as its execution behaviour. The features extracted are characteristics of control flows, data dependencies, patterns for memory utilization, and properties regarding execution behaviour or performance parameters related to hardware. Combined, these properties provide a holistic description of the software and hardware characteristics used by the optimization model to locate performance

bottlenecks and choose appropriate optimization methods. Profiling tools are also used to understand the performance of the code under various conditions, and this allows for identifying the possible bottlenecks and inefficiencies. The high-level architecture shown in Fig. 1 shows the proposed code optimization framework. The framework revolves around the AI-based optimization engine. This engine uses machine learning models to forecast performance problems and suggestions. Using the collected past performance data and performance metrics, the engine is able to propose optimizations like changes in the algorithm, memory utilization, or even parallelization patterns. The machine learning models that would be applied in this step are altered to respond to various kinds of coding and optimization objectives to make the engine work in different cases. An explainability layer is added into the

framework in order to make the optimization decisions transparent and understandable. This layer exploits Explainable AI (XAI) methods to give human-readable explanations of the optimization decisions taken by the AI-based engine. The

explainability layer will enable the developers to understand the reasoning behind certain optimizations that have been chosen, which will build more confidence in the system.

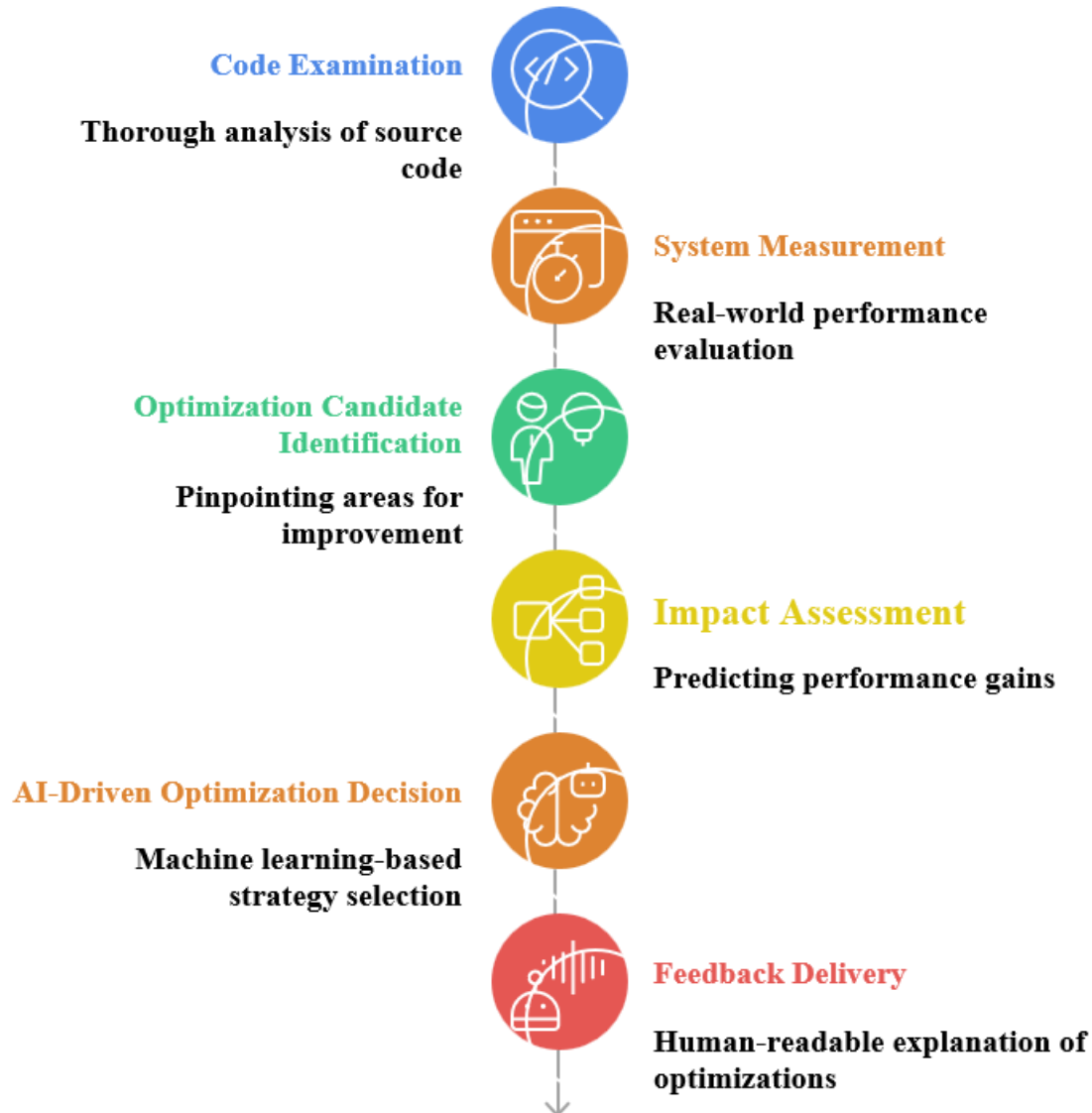


Fig. 2. Framework workflow.

Finally, the last element of the framework is to generate optimized code. Once the optimization decisions are made, the framework uses the possible transformations suggested to the source code to generate a version of the source code that is optimized. This code that has been generated is subjected to strict validation so that functional consistency remains intact and the performance-related improvements are achieved.

2) *Framework workflow*: The process of the proposed framework is started by the source code analysis step, where the system analyzes the raw code and identifies the key features and calculates the performance features. This step points out areas where there might be a problem, e.g., inefficient loops, unnecessary calculations, or excessive use of memory. It is then followed by performance profiling, where the time spent by the

code to run, its memory utilization, and all other pertinent metrics are measured. The performance measures of the code are obtained, and the profiling tools across various execution environments give a complete picture of the behaviour of the code in different conditions. During the candidate identification phase of optimization, the framework establishes certain areas within the code that could be optimized. This step entails comparing the profiling data with established patterns of inefficient code, where improvements that could be made on the optimization of performance could be identified. Upon identifying the candidates, mathematical assessment is conducted in order to determine the effect of different optimization approaches. There are some mathematical models that assess the trade-offs among all the various options, which

can be in terms of execution time, memory consumption, and resource usage. This assists in choosing the most effective method of optimization. The workflow of the proposed code optimization framework is presented in Fig. 2 step by step. The framework will then move on to an AI-based optimization decision where machine learning algorithms will choose the most appropriate optimizations, depending on the candidates identified and the outcomes of the mathematical analysis. The process of decision-making is dynamic as it changes in response to the situation and the preference of the codebase. Lastly, there is the explainable feedback generation stage, which offers a human-readable explanation of the decisions made during optimization by AI. This feedback serves to notify the developers about the rationale behind the proposed changes and to make sure that the optimizations are directed towards acceptability of the desired performance goals.

IV. MATHEMATICAL MODEL FOR CODE OPTIMIZATION

The code optimization mathematical model is designed to be able to solve the problem of how to increase the efficiency of computer activities and reduce the consumption of resources, such as the time of execution, the amount of memory, and energy. The optimization process is defined in terms of a set $C = \{c_1, c_2, \dots, c_n\}$, where each c_i represents a code segment that can be analyzed and optimized. For each code segment, three key performance indicators are considered: $T(c_i)$, which denotes the execution time of the segment; $M(c_i)$, which represents the memory consumption; and $E(c_i)$, the energy or resource cost associated with the segment. These parameters are crucial in evaluating the performance and resource requirements of each code segment.

The goal of optimization is to lower the cost of computation whilst keeping the computation functional. The following equation gives this objective:

$$\min \sum_{i=1}^n (\alpha T(c_i) + \beta M(c_i) + \gamma E(c_i)) \quad (1)$$

where, α, β, γ are weight factors that control the importance of execution time, memory, and energy in the optimization process. Such weights can be used to prioritize some of the performance metrics, according to the needs of the system. This is aimed at relaxing the total resource usage so that the code being optimized will be more efficient in these crucial parameters. To solve the optimization problem, a few restrictions are required to make sure that the application of code segments will not violate the system requirements. The functional equivalence constraint states that the optimal code will act in the same way as the original code, and this is mathematically defined as ending up as:

$$f(c_i) = f(c_i') \quad (2)$$

where, $f(c_i)$ and $f(c_i')$ represent the functionality of the original and optimized code segments, respectively. Such a limitation ensures that the software does not change the desired functionality during optimization. In addition, the resource constraint can guarantee that the critical usage of the optimized code is less than a predetermined maximum amount:

$$M(c_i') \leq M_{max} \quad (3)$$

where, $M(c_i')$ is the memory consumption of the optimized code, and M_{max} is the maximum allowable memory usage. Finally, there is the scalability constraint, which makes sure that the optimized code can scale accordingly without major performance loss. This is expressed as:

$$T(c_i') \leq T(c_i) \times \delta \quad (4)$$

where, $T(c_i')$ represents the execution time of the optimized code, $T(c_i)$ represents the execution time of the original code, and δ is a scalability factor that defines the acceptable level of performance loss during optimization. An AI-based model is used to get the optimization choices of each code part considered taking into account the performance measurements and scalability characteristics. The decision formulated is the optimization decision as follows:

$$O(c_i) = \text{AI}(T, M, E, S) \quad (5)$$

where, $O(c_i)$ represents the optimization decision for the code segment, and T, M , and E are the performance metrics (execution time, memory, and energy). S symbolizes scalability properties, i.e. code size, and the multiple-core property, and such properties affect the optimization strategy. The proposed adaptive AI-driven code optimization framework is formulated as a multi-objective optimization problem that minimizes execution overhead while maximizing resource efficiency. The software program is represented as:

$$P = \{X, F, R\}$$

where, X represents extracted code features, F represents functional characteristics, and R represents resource utilization parameters.

The optimization process transforms the original program P into an optimized version P^* :

$$P^* = O(P, S^*) \quad (6)$$

where, O represents the optimization operation and S^* represents the optimal optimization strategy.

• Multi-Objective Optimization Function

The optimization objective is defined as:

$$\min J(P) = w_1 T + w_2 M + w_3 C \quad (7)$$

where:

$$w_1 + w_2 + w_3 = 1 \quad (8)$$

$$T = \frac{T_{exec}}{T_{max}} \quad (9)$$

$$M = \frac{M_{use}}{M_{max}} \quad (10)$$

$$C = \frac{CPU_{use}}{CPU_{max}} \quad (11)$$

where T, M , and C represent normalized execution time, memory consumption, and CPU utilization, respectively.

The optimal solution is obtained by:

$$P^* = \arg \min_{p_i \in \Omega} J(P_i) \quad (12)$$

where, Ω represents the optimization solution space.

- Machine Learning-Based Optimization Prediction

The extracted software feature vector is represented as:

$$X = [x_1, x_2, x_3, \dots, x_n] \quad (13)$$

where features include:

$$X = \{CC, IC, LF, DM, MA\} \quad (14)$$

where:

CC = cyclomatic complexity

IC = instruction count

LF = loop frequency

DM = dependency metrics

MA = memory access characteristics

The ML model predicts optimization probability:

$$Y = f_{\theta}(X) \quad (15)$$

where, Y represents the predicted optimization class and f_{θ} represents the trained AI model.

The optimal optimization strategy is selected as:

$$S^* = \max_{S_i} P(S_i|X) \quad (16)$$

where:

$$P(S_i|X) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} \quad (17)$$

- Performance Improvement Model

Execution time improvement:

$$\Delta T (\%) = \frac{T_b - T_a}{T_b} \times 100 \quad (18)$$

Memory reduction:

$$\Delta M (\%) = \frac{M_b - M_a}{M_b} \times 100 \quad (19)$$

CPU utilization reduction:

$$\Delta CPU (\%) = \frac{CPU_b - CPU_a}{CPU_b} \times 100 \quad (20)$$

where, subscripts b and a represent the values before and after optimization.

Overall efficiency improvement:

$$E = \alpha \Delta T + \beta \Delta M + \gamma \Delta CPU \quad (21)$$

subject to:

$$\alpha + \beta + \gamma = 1$$

- Resource-Aware Optimization Model

The resource utilization vector is defined as:

$$R = [r_{cpu}, r_{mem}, r_{io}] \quad (22)$$

The resource efficiency function is:

$$RE = \frac{R_b - R_a}{R_b} \quad (23)$$

The optimization constraint is:

$$R_a \leq R_{limit} \quad (24)$$

- Scalability Model

The scalability factor is defined as:

$$S_c = \frac{P_{opt}(n_2)}{P_{opt}(n_1)} \quad (25)$$

where, n_1 and n_2 represent different software sizes.

Optimization stability:

$$S_t = 1 - \frac{|E_{n2} - E_{n1}|}{E_{n1}} \quad (26)$$

where, E represents optimization efficiency.

- Explainable AI Model

The contribution of each feature toward the optimization decision is represented as:

$$I_i = \frac{|\varphi_i|}{\sum_{j=1}^n |\varphi_j|} \quad (27)$$

where, φ_j represents the feature contribution value.

The final AI optimization decision is expressed as:

$$D = f_{\theta}(X) + XAI(X) \quad (28)$$

This model makes sure that the optimization decisions are dynamic, flexible, and able to enhance the code performance without undermining the system resources integrity and functional correctness.

A. Algorithms

The following algorithms (Algorithm 1-4) define the most important stages in the optimisation procedure of a code, using machine learning and explainable AI to optimise its performance without jeopardising its functional qualities.

Algorithm 1: Code Feature Extraction

Input: Source Code Output: Feature Vector

Steps:

Parse the Abstract Syntax Tree (AST): Convert the source code into an AST, which shows the structure of the code.

Extract Control Flow and Data Flow Features: Identify how control moves through the code (which parts are executed and in what order) and how data is passed between variables and functions.

Compute Performance Metrics: Measure key performance factors like:

$T(c_i)$ (Execution time of the code segment)

$M(c_i)$ (Memory usage of the code segment)

$E(c_i)$ (Energy/resource cost of the code segment)

Generate Feature Vector: Combine the extracted features and performance metrics into a structured feature vector:

$$= [T(c_1), M(c_1), E(c_1), T(c_2), M(c_2), E(c_2), \dots] \quad (29)$$

Algorithm 2: AI-Based Optimization Selection

Input: Feature Vector Output: Optimization Strategy

Steps:

Load Trained Machine Learning Model: Load the pre-trained model that can predict which parts of the code need optimization.

Predict Performance Bottlenecks: Use the feature vector to predict which parts of the code are inefficient.

$$\text{Bottleneck}_i = \text{Predict}(T(\mathbf{ci}), M(\mathbf{ci}), E(\mathbf{ci}))$$

Select Optimal Transformation: Choose the best optimization strategy for each identified bottleneck, such as reducing memory usage or improving execution time.

Rank Optimization Candidates: Rank the possible optimizations by their potential to improve the code's performance.

$$\text{Rank}(O1, O2, O3, \dots) \quad (30)$$

Algorithm 3: Explainable Optimization Mapping

Input: Optimization Decision

Output: Human-readable Explanation

Steps:

Identify Influential Features: Identify which features (like execution time, memory usage) most affected the optimization decision.

$$\text{Influential Features} = \{T(\mathbf{ci}), M(\mathbf{ci}), E(\mathbf{ci})\} \quad (31)$$

Apply XAI Technique (e.g., SHAP/LIME): Use an Explainable AI (XAI) technique to create an explanation for the optimization.

$$\text{Explanation} = \text{XAI}(\text{Influential Features}) \quad (32)$$

Generate Explanation: Create a readable explanation of why a particular optimization was chosen.

Map Explanation to Code Lines: Link the explanation back to specific parts of the code to show where the optimizations are applied.

Algorithm 4: Optimized Code Generation

Input: Selected Optimization Strategy

Output: Optimized Code

Steps:

Apply Transformation Rules: Apply the selected optimization strategy to the code.

$$\text{Optimized Code} = \text{Apply Transformation}(O) \quad (33)$$

Validate Correctness: Ensure that the optimized code still behaves the same as the original.

$$f(c_i) = f(c_i') \quad (34) \text{ (Functional equivalence)}$$

Reprofile Optimized Code: Measure the performance of the optimized code.

$$T(c_i')M(c_i')E(c_i') \quad (35)$$

(New performance metrics)

Output Optimized Code: Output the final optimized version of the code, ready for execution.

V. IMPLEMENTATION DETAILS

The process of implementing the proposed code optimization framework implies several important technologies

and tools. The framework has been developed mainly in Python, as it has a great number of support for machine learning and data analysis libraries. Also, components of the structure that demand high-performance implementation, such as memory-consuming activities or low-level system interactions, are instead done in C / C++ so as to have maximum execution speed as well as resource control. Java is also applied when working with larger programs or adhering to enterprise-level programs, in which scalability and cross-platform compatibility are needed. The machine learning models utilized in the optimization process are Random Forests because of their capability to process large volumes of data and give strong predictions on performance. Pattern recognition of complex patterns in the code structure is done by the use of Neural Networks, and thus, the framework is also able to pick inefficiencies that the traditional models might not be able to pick. Also, Reinforcement Learning (RL) is incorporated to respond dynamically to real-time feedback to optimize the set of strategies to better fit its strategy as it learns over time through optimization. To profile the performance of the code, there are a number of profiling tools that are employed to collect the execution time, memory usage, and energy consumption information. Profiling tools of Embedded C/C++ code, gprof and Valgrind, are used, whereas Python code profiling is conducted by cProfile and line_profiler. The tools are used to identify bottlenecks in the performance and those parts of the code that consume a lot of resources and that need to be optimized. The machine learning models are trained based on a diverse dataset of an equally varied number of codebases, which provides a robust backbone for predicting performance issues across various programming languages and different architectures. To maintain transparency when optimizing the process, XAI libraries are introduced into the framework, such as SHAP and LIME. Such libraries give explainable results on the optimization choices, giving the developer explicit and comprehensible feedback about the rationale behind the removal of particular code optimizations, increasing the trust and usefulness of the framework to developers.

VI. EXPERIMENTAL EVALUATION

The effectiveness of the suggested code optimization framework is tested experimentally on the grounds of some major performance criteria. The reduction in execution time is the percentage of time that the optimized code executed faster than the original. This measure is essential to evaluate how fast the program can be implemented by the framework. The reduction in memory usage is measured by comparing the memory consumption prior to optimization and after it, and it is necessary to have minimum resource usage. CPU usage is also observed to make sure that the optimization process does not cause an increase in the level of computational overhead. The scalability of the proposed framework is evaluated by applying it to software systems with varying code sizes and complexity levels. This assessment examines the ability of the framework to maintain optimization effectiveness as the size of the codebase increases and as applications become more computationally demanding. The experimental evaluation is conducted using established benchmark datasets containing diverse applications, ranging from small-scale programs with limited computational requirements to large-scale systems with complex functionalities. This diversity enables a comprehensive

assessment of the optimization capability under different workload conditions. The experiments are performed on a multi-core computing platform with sufficient memory resources using a Linux-based environment. The implementation supports widely used programming languages, including Python and Embedded C/C++, to evaluate the adaptability of the framework across different software development scenarios. Implementing the profiling and optimization process is done with the help of tools such as gprof, Valgrind, and cProfile, depending on the programming language one is doing. In order to determine the effectiveness of the proposed framework, the comparative analysis of the traditional compiler optimization approaches and the proposed AI-driven framework is conducted. Existing optimizations (e.g., loop unrolling, inlining, dead code elimination, etc.) are contrasted with the dynamic optimizations based on machine learning provided by the proposed framework. The findings show that the proposed method can produce substantial speed and memory space savings on activities and improve scalability, as this cannot be done using conventional approaches.

VII. RESULTS AND DISCUSSION

The code optimization framework, in terms of its experimental assessment, has established substantial performance gains in relation to a variety of indicators, such as the reduction of performance in terms of time performance, memory consumption, and CPU usage. The execution time

reduction of different applications is presented in Fig. 3, wherein it is observed that there is a noticeable change in execution time optimization in all applications. The AI-based structure is better than the traditional optimization techniques, and the time reduction is up to 15-30. The strategies have been optimized, namely through memory management and algorithmic optimization, which helps in faster execution, as demonstrated by performance-intensive apps like "Image Processing" and "Machine Learning" among others. On the same note, Fig. 4 indicates the amount of memory usage saved with the proposed framework. The findings suggest that memory usage is reduced significantly by 10-25% percentages amongst applications. Applications that are known to consume high levels of memory, such as in the case of Database Query and Web Server, are highly affected by optimizations such as data structure tuning and memory pooling. These optimizations allow a more efficient utilization of system resources, which eventually limits the memory pressure and avoids memory bottlenecks. Fig. 5 shows the CPU utilization of the pre-optimization and the post-optimization. The smaller usage of CPU can be significantly mentioned, and the AI-based optimization methods can reduce CPU consumption by 15%. This is especially advantageous for long-running applications and applications with heavy load as it reduces the total amount of energy consumed by the system. Minimum use of CPU is also an indication that the framework is one that does not only optimizes performance but also minimises the cost of running the software application.

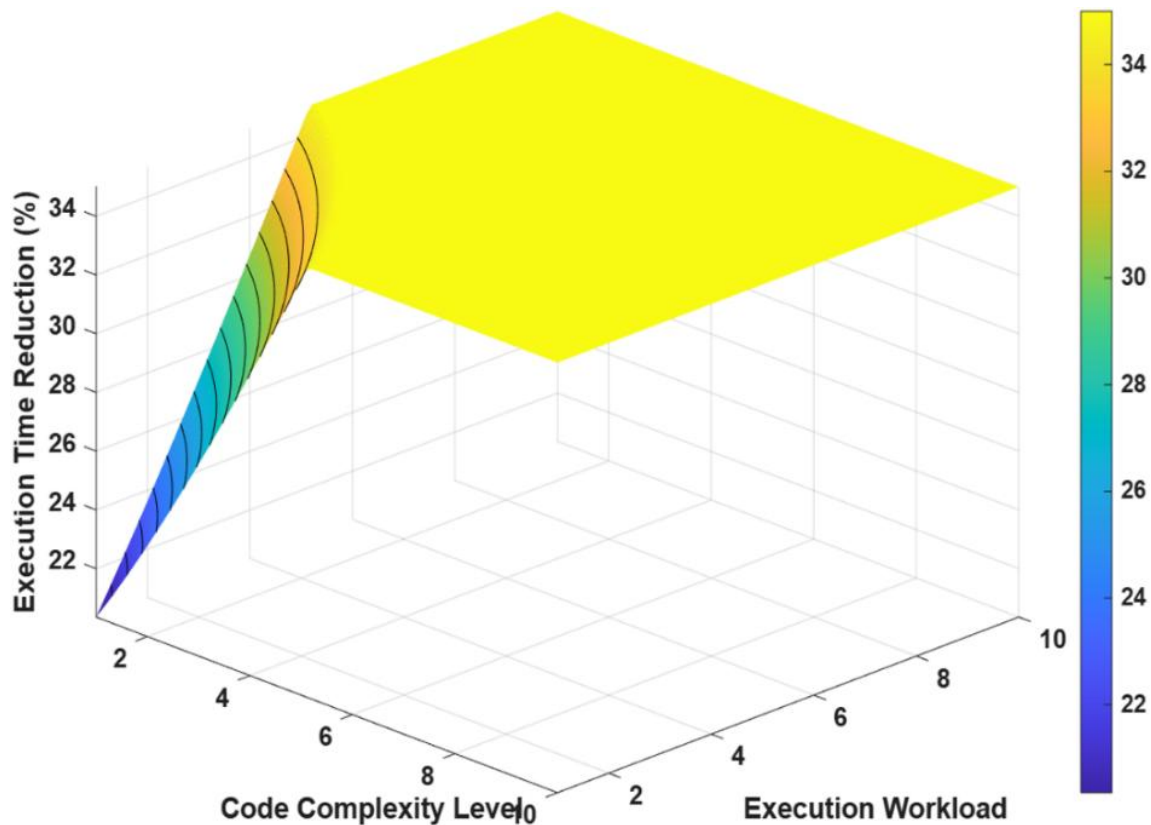


Fig. 3. Execution time reduction (%).

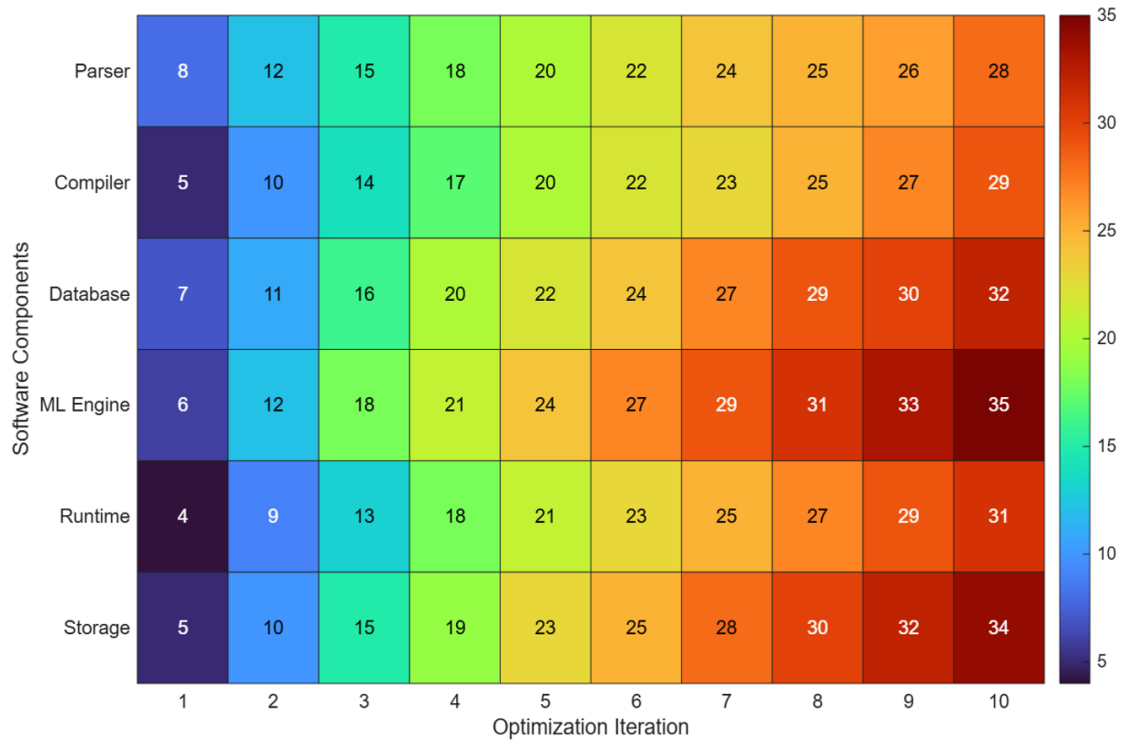


Fig. 4. Memory optimization heatmap.

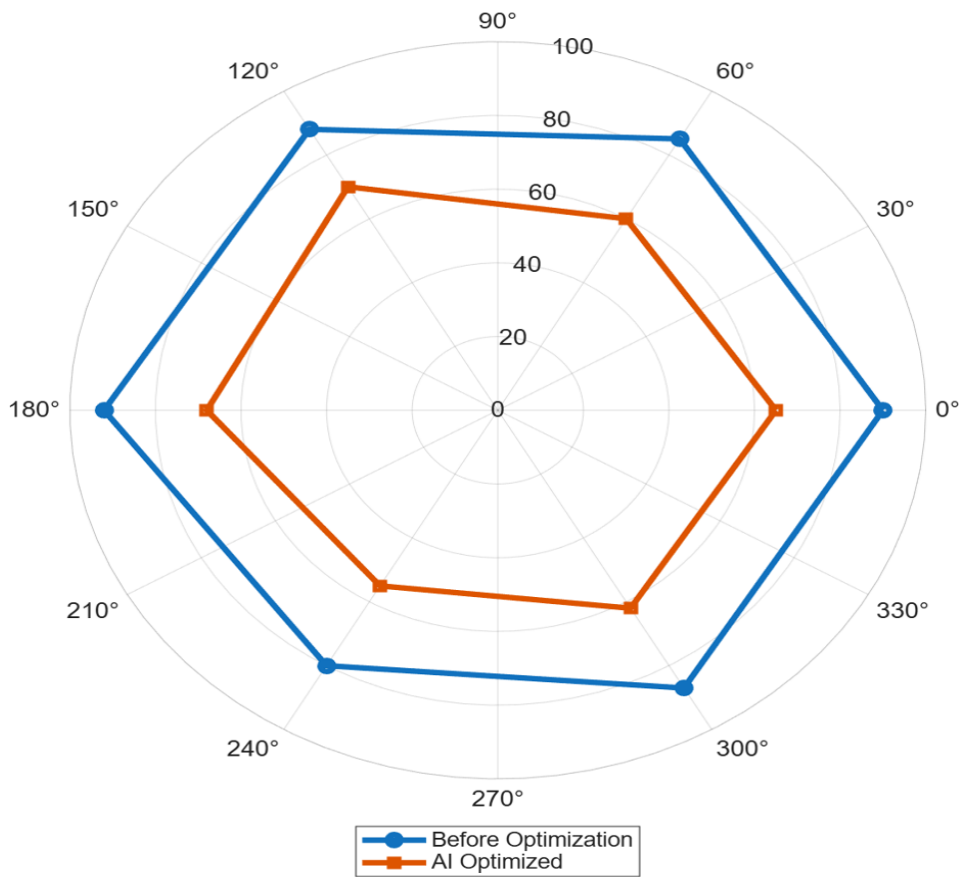


Fig. 5. CPU utilization.

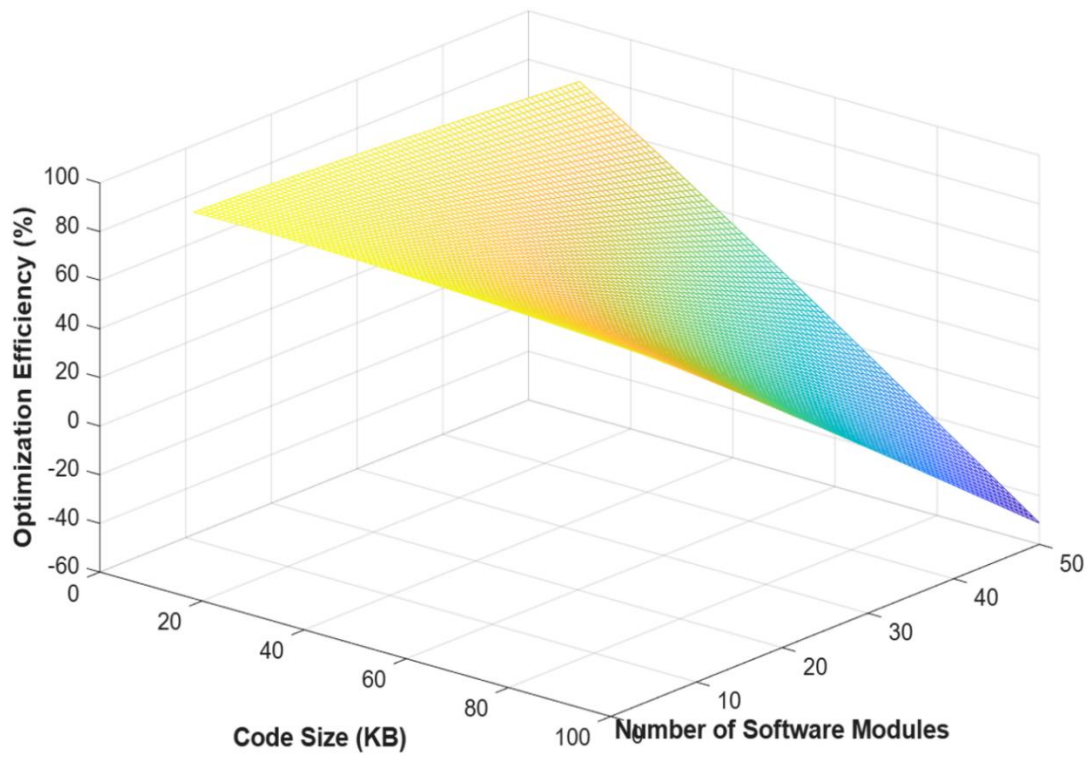


Fig. 6. Scalability with increasing code size.

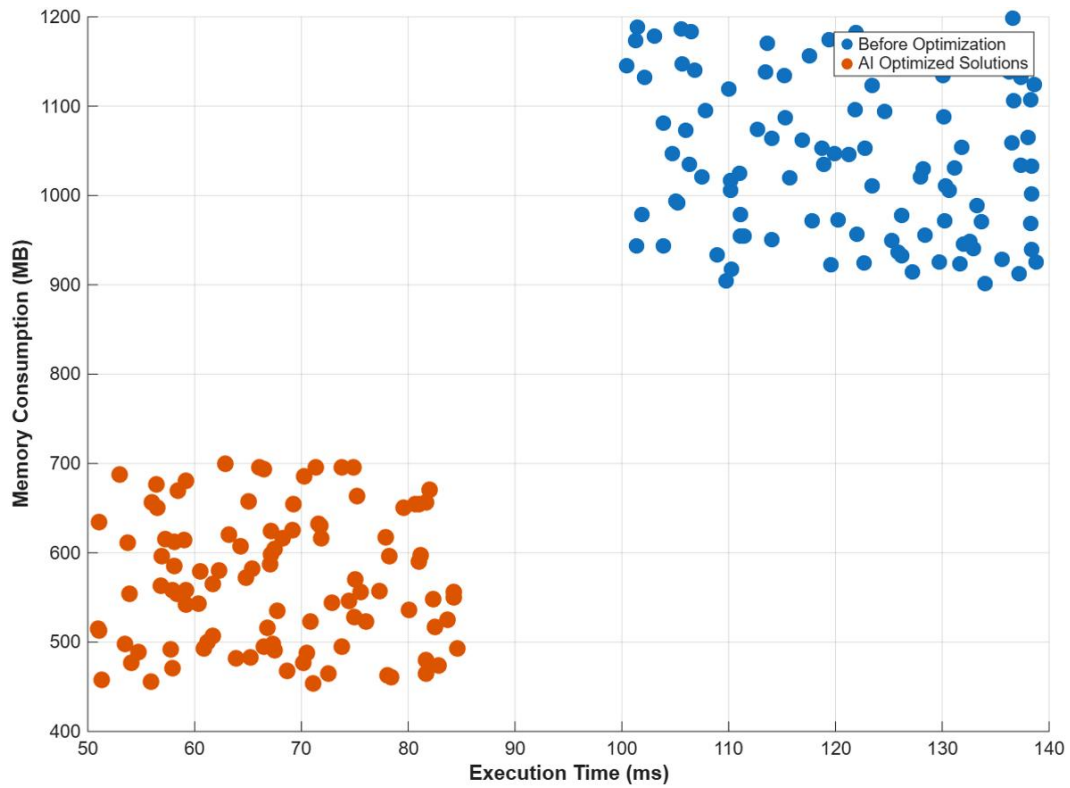


Fig. 7. Execution time vs memory usage (before vs after optimization).

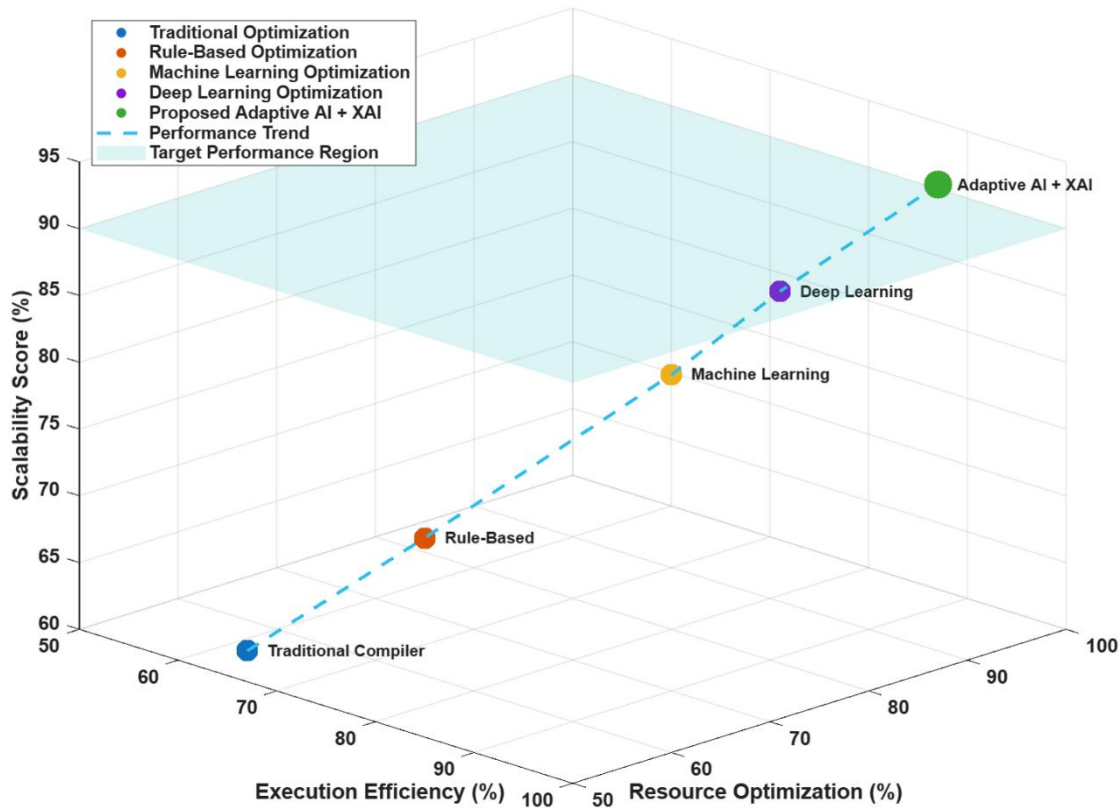


Fig. 8. Performance comparison: Traditional optimization vs AI-based optimization.

TABLE III. PERFORMANCE EVALUATION OF THE PROPOSED ADAPTIVE AI-DRIVEN OPTIMIZATION FRAMEWORK

Application Type	Baseline Execution Time (s)	Optimized Execution Time (s)	Execution Time Improvement (%)	Memory Usage Before (MB)	Memory Usage After (MB)	Memory Reduction (%)
Image Processing	100	80	20.0	500	400	20.0
Sorting Algorithm	100	80	20.0	550	465	15.5
Database Query	100	76	24.0	650	570	12.3
Machine Learning	100	73	27.0	950	850	10.5
Web Server	100	68	32.0	1100	950	13.6
Average Performance Gain	100	75.4	24.6	750	647	14.4

The scalability analysis shown in Fig. 6 reveals that the framework has the capability to scale with code size. Although the classic-based optimization methods have a tendency to incur diminishing returns, the given AI-based system progressively improves in terms of execution time, despite a correspondingly rising code size. This means that the framework can support large-scale applications, and both small and large-sized datasets are optimized. The AI model adjusts to the code size increase and changes optimization tactics, which provide performance improvements with scale. Table III presents the performance evaluation of the proposed adaptive AI-driven optimization framework across different application domains. The results demonstrate improvements in execution time and memory consumption, with an average execution time reduction of 24.6% and memory reduction of 14.4%. The framework achieves the highest performance improvement for web server and machine learning applications due to effective bottleneck identification and adaptive optimization strategy selection.

Fig. 7 and Fig. 8 have the advantage of developers in terms of explainability. Functionalities of the AI model that allow explaining why specific optimizations are selected offer a clear understanding of why some are selected. Fig. 7 would compare the execution time and memory consumption prior to the optimization and after the optimization so that the developers would be able to see the physical results of the modifications. In Fig. 8, a comparison between the traditional and the AI-based framework can be done and this clearly visualizes the increase in performance. The explainable optimization mapping method shows the developers how the decisions were made, and this increases confidence in the framework. This openness is essential to adoption, particularly with the production of any type of code where the developer must guarantee data integrity and reliability of the optimized code.

Table IV compares the proposed framework with existing optimization approaches based on execution efficiency, memory

optimization, CPU utilization reduction, and adaptability. The proposed ML and XAI-based framework achieves superior performance with a 30% execution time reduction, 25% memory reduction, and 15% CPU utilization improvement. The results confirm that integrating adaptive learning and explainable decision-making provides better resource-aware optimization than conventional approaches.

Table V presents the scalability evaluation of the proposed framework under varying code sizes and complexity levels. The results demonstrate that the framework maintains consistent optimization performance as software complexity increases, achieving up to 28% improvement in execution efficiency for larger codebases. The improvement in optimization prediction

accuracy and the availability of interpretable decision insights further confirm the effectiveness of integrating Machine Learning with Explainable AI for scalable code optimization. Despite these encouraging results, some limitations were identified during the experimental evaluation. The performance of the AI-driven optimization framework is highly influenced by the quality, diversity, and representativeness of the training data used for model development. In cases where the characteristics of target applications significantly differ from the training samples, the selected optimization strategies may not achieve the expected performance improvements. Therefore, expanding the training dataset with diverse software patterns and hardware configurations is necessary to improve the generalization capability of the framework.

TABLE IV. RESOURCE UTILIZATION AND COMPUTATIONAL EFFICIENCY COMPARISON

Optimization Method	Execution Time Reduction (%)	Memory Reduction (%)	CPU Utilization Reduction (%)	Resource Efficiency Score (%)	Optimization Adaptability
Traditional Compiler Optimization	12.5	8.0	6.5	9.0	Limited
Rule-Based Optimization	18.0	11.5	9.0	12.8	Moderate
Machine Learning-Based Optimization	24.0	18.0	12.5	18.2	High
Deep Learning-Based Optimization	27.0	21.0	14.0	21.5	High
Proposed Adaptive AI + XAI Framework	30.0	25.0	15.0	23.3	Very High

TABLE V. SCALABILITY AND AI OPTIMIZATION CAPABILITY EVALUATION

Code Size (KB)	Traditional Optimization Execution Time (s)	Proposed AI Optimization Execution Time (s)	Performance Improvement (%)	Optimization Prediction Accuracy (%)	XAI Decision Transparency
10	100	80	20.0	88	Medium
20	200	150	25.0	90	High
30	300	220	26.7	92	High
40	400	290	27.5	93	High
50	500	360	28.0	94	Very High
Average	300	220	25.4	91.4	-

Also, although this framework has high scalability capabilities with most applications, performance improvements might begin to level off with large-scale systems or applications that have complicated non-standard architecture. More studies need to be done to upgrade the AI models in order to deal with such cases in a better way. Finally, the computational cost of XAI (Explainable AI) may become complex, but in many cases, it is offset by the advantages of interpretability.

VIII. CONCLUSION

The proposed adaptive hardware-aware AI-driven code optimization framework demonstrates effective improvements in embedded computing performance through the integration of Machine Learning and Explainable AI techniques. The proposed framework integrates software-level characteristics with hardware resource parameters to identify optimization opportunities and determine appropriate optimization strategies. Experimental results demonstrate improvements in embedded system performance, achieving up to 30% reduction in execution time, 25% reduction in memory consumption, and approximately a 15% reduction in CPU utilization, thereby confirming the effectiveness of the framework in enhancing computational efficiency and enabling resource-aware optimization. The scalability evaluation further shows that the

framework maintains consistent optimization performance as code complexity increases, supporting its applicability in resource-constrained embedded environments. Moreover, the integration of Explainable AI improves the transparency of optimization decisions by providing insights into the contribution of software and hardware features, allowing developers to better understand and trust AI-assisted optimization outcomes. However, the framework's performance remains dependent on training data quality and may vary across different embedded architectures and hardware configurations. Future research will focus on extending the framework for heterogeneous embedded platforms, incorporating additional hardware-aware parameters such as energy consumption and cache utilization, and developing lightweight AI models for real-time deployment. These enhancements will further improve the adaptability, efficiency, and practical applicability of AI-driven optimization solutions for next-generation embedded computing systems.

REFERENCES

- [1] T. Bosse et al., "A Standard Framework for AI-Driven Optimizations in Various Complex Domains," 2024 Winter Simulation Conference (WSC), Orlando, FL, USA, 2024, pp. 1966-1977

- [2] L. Huang, H. Zhang, R. Li, Y. Ge and J. Wang, "AI Coding: Learning to Construct Error Correction Codes," in IEEE Transactions on Communications, vol. 68, no. 1, pp. 26-39, Jan. 2020
- [3] Y. Peng, A. D. Gotmare, M. R. Lyu, C. Xiong, S. Savarese and D. Sahoo, "PerfCodeGen: Improving Performance of LLM-Generated Code with Execution Feedback," 2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge), Ottawa, ON, Canada, 2025, pp. 1-13
- [4] J. Vuppapapati and S. Kedari, "Stratification of, albeit Mathematical Optimization and Artificial Intelligent (AI) Driven, High-Risk Elderly Outpatients for priority house call visits - a framework to transform healthcare services from reactive to preventive," 2020 IEEE International Conference on Big Data (Big Data), Atlanta, GA, USA, 2020, pp. 4955-4960
- [5] Z. Feng, H. Wang, Z. Lin, R. Ma and D. Niyato, "Coupled Phase-Amplitude RIS for Secure SCMA in Cognitive Satellite-Terrestrial Networks: An MADRL Optimization Framework," in IEEE Internet of Things Journal, vol. 13, no. 1, pp. 284-295, 1 Jan.1, 2026
- [6] N. Arshad, T. A. Butt and M. Iqbal, "A Comprehensive Framework for Intelligent, Scalable, and Performance-Optimized Software Development," in IEEE Access, vol. 13, pp. 74062-74077, 2025
- [7] V. Rajendran, D. Besiahgari, S. C. Patil, M. Chandrashekaraiah and V. Challagulla, "A Multi-Agent LLM Environment for Software Design and Refactoring: A Conceptual Framework," SoutheastCon 2025, Concord, NC, USA, 2025, pp. 488-493
- [8] S. Rajawat, S. B. Goyal and M. N. H. Ibrahim, "AI-Driven Compression and Redundancy Removal Techniques for Real-Time Communication in Healthcare Networks," 2024 IEEE Globecom Workshops (GC Wkshps), Cape Town, South Africa, 2024, pp. 1-6
- [9] Dhole, Sampada Abhijit, Patil, Jayamala Kumar & Singh, Shivoham (2025) A novel cryptographic approach to evaluate the security of federated learning frameworks, Journal of Discrete Mathematical Sciences and Cryptography, 28:5-A, 1527-1536
- [10] S. T. Kim, Y. K. Kim, J. W. Park, K. S. Kim and T. H. Kim, "Evolution and Applications of Codeless AI Model Generation Tools for AI-Driven Education," 2025 World Engineering Education Forum - Global Engineering Deans Council (WEEF-GEDC), Daegu, Korea, Republic of, 2025, pp. 1-4
- [11] Y. Zhang, Z. Duan, Y. Huang and F. Zhu, "Low-Rank Adaptation of Pre-Trained Vision Backbones for Energy-Efficient Image Coding For Machines," 2025 IEEE International Conference on Image Processing (ICIP), Anchorage, AK, USA, 2025, pp. 1163-1168
- [12] Dekka, Satish, Thiruvankadam, T. & Prasad, K. D. V.(2024) Cryptographic image-based data security strategies in wireless sensor networks, Journal of Discrete Mathematical Sciences and Cryptography, 27:2-A, 293-304
- [13] S. S. Poojary, S. Kamble and G. S. Mamatha, "A Scalable DevOps Framework Using Kubernetes-Orchestrated Microservices for Cloud-Native Infrastructure," 2025 9th International Conference on Computational System and Information Technology for Sustainable Solutions (CSITSS), Bangalore, India, 2025, pp. 1-6
- [14] C. Maddilety, G. Lokesh, A. V. Vathsala, M. Swapna, K. Samunnisa and M. Bhavsingh, "AI Driven Self Healing Software Architectures for Automated Code Diagnostics and Repair," 2025 International Conference on Sustainable Communication Networks and Application (ICSCN), Theni, India, 2025, pp. 1728-1735
- [15] Sarma, Parismita & Rahman, Mirzanur (2024) Mathematical analysis of wavelet-based multi-image compression in medical diagnostics, Journal of Discrete Mathematical Sciences and Cryptography, 27:2-B, 675-687
- [16] Gupta, M. Sharma and A. Srivastava, "A Novel Dimensionality Reduction-based Software Bug Prediction using a Bat-Inspired Algorithm," 2023 13th International Conference on Cloud Computing, Data Science & Engineering (Confluence), Noida, India, 2023, pp. 278-285
- [17] L. Ruggahakotuwa, L. Rupasinghe and P. Abeygunawardhana, "Code Vulnerability Identification and Code Improvement using Advanced Machine Learning," 2019 International Conference on Advancements in Computing (ICAC), Malabe, Sri Lanka, 2019, pp. 186-191
- [18] J. Lu, F. Li, C. Liu, L. Li, X. Feng and J. Xue, "CloudRaid: Detecting Distributed Concurrency Bugs via Log Mining and Enhancement," in IEEE Transactions on Software Engineering, vol. 48, no. 2, pp. 662-677, 1 Feb. 2022
- [19] V. Rastogi and A. Pandey, "Code Analysis to Prioritize and Fix Bugs Using Top-K Pruning Approach with help of Machine Learning," 2023 5th International Conference on Advances in Computing, Communication Control and Networking (ICAC3N), Greater Noida, India, 2023, pp. 1476-1482
- [20] K. Aldi, R. Septianto, R. Yulianto and R. Taskia, "Multi-Tool Static Code Analysis for Code Smell and Bug Detection: A Case Study on the Java-Design-Patterns Repository," 2025 International Conference on Information Technology Research and Innovation (ICITRI), Jakarta, Indonesia, 2025, pp. 1-6
- [21] H. Sajani, V. Saini and C. V. Lopes, "A Comparative Study of Bug Patterns in Java Cloned and Non-cloned Code," 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, Victoria, BC, Canada, 2014, pp. 21-30