Enhancing Code Quality Through Automated Refactoring Using Transformer-Based Language Models

A. Sri Lakshmi[1], Dr. E. S. Sharmila Sigamany[2], Roopa Traisa[3], Raman Kumar[4],
Karaka Ramakrishna Reddy[5], Jasgurpreet Singh Chohan[6], Aseel Smerat[7]
Research Scholar, Department of English, Koneru Lakshmaiah Education Foundation, Vaddeswaram, AP, India[1]
Associate Professor, Department of English, Koneru Lakshmaiah Education Foundation, Vaddeswaram, AP, India[2]
Department of Management-School of Management-UG, JAIN (Deemed to be University), Bangalore, Karnataka, India[3]
University School of Mechanical Engineering, Rayat Bahra University, Mohali, India[4]
Faculty of Engineering, Sohar University, Sohar, Oman[4]
Assistant Professor, Department of BS&H, B V Raju Institute of Technology, Narsapur, Medak, Telangana, India[5]
Marwadi University Research Center-Department of Mechanical Engineering-Faculty of Engineering & Technology,
Marwadi University, Rajkot, Gujarat, India[6]
Faculty of Educational Sciences, Al-Ahliyya Amman University, Amman, 19328, Jordan[7]
Department of Biosciences-Saveetha School of Engineering, Saveetha Institute of Medical and Technical Sciences,
Chennai, 602105, India[7]

***Abstract*—Maintaining high-quality source code is crucial for software reliability, scalability, and maintainability. Traditional refactoring methods, which involve manual code improvement or rule-based automation, often fall short due to their inability to understand the contextual semantics of code. These approaches are rigid, language-specific, and prone to inconsistencies, especially in large and complex codebases. As a result, developers spend significant time and effort identifying code smells, restructuring poorly written segments, and ensuring behavior preservation shows an accuracy of 97%. To address these limitations, this study proposes an automated code refactoring framework powered by Transformer-based language models. Leveraging models such as CodeT5, which are pre-trained on massive code corpora, this approach captures both syntax and semantic patterns to suggest intelligent, context-aware code transformations. The model is fine-tuned using a curated dataset of original and refactored code pairs to learn efficient refactoring strategies. The methodology involves preprocessing raw source code, tokenizing it for model input, and generating improved versions of the code using the trained Transformer model. Output suggestions are validated using Abstract Syntax Tree (AST) analysis and unit testing to ensure behavioral equivalence. Code quality improvements are quantified using metrics like maintainability index, cyclomatic complexity, and duplication rate. Experimental results demonstrate that the proposed method significantly enhances code readability and maintainability while reducing developer effort, outperforming traditional rule-based refactoring tools.**

***Keywords*—Automation; code refactoring; maintainability; transformer models; unit testing**

I. Introduction

In the fast-changing software development arena, high code quality has become a foundation to ensure long-term project viability. Clearly written, readable, and maintainable code is simpler to debug, test, and extend, ensuring lower development costs and increased software dependability [1]. Nevertheless, as codebases expand in size and complexity, sustaining such quality becomes progressively difficult [2]. Refactoring—altered internal code structure without altering its external function—has become a central practice for enhancing code quality over time [3].

Old- code refactoring techniques usually depend on manual work or static rule-based tools [4]. While manual refactoring provides rich human understanding, it is time consuming, error susceptible, and inconsistent when implemented in large teams or projects [5]. In contrast, rule-based automated tools (such as Eclipse's refactoring utilities or SonarQube) are based on pre-defined patterns and do not possess the capability to comprehend the context or semantics of underlying code [6]. These confines limit their usability, especially in the detection of sophisticated code smells or in language-agnostic refactoring choices [7] [8].

Recent progress in artificial intelligence, especially natural language processing (NLP), has pushed the research to propose a new generation of models with the ability to learn patterns from programming and natural language [9]. Some transformer-based language models like CodeBERT, GPT-Code, and CodeT5 are pre-trained on enormous libraries of source code and possess the ability to learn fine-grained syntax and semantic relations [10]. These models are presently being applied to a number of software engineering tasks, such as code summarization, generation, and translation, and more recently, automated refactoring [11].

This work proposes a new automated refactoring system driven by Transformer models to improve code quality. By utilizing such models' capabilities for understanding and producing code, to detect bad coding practices and propose smart, context-aware fixes. The method not only refactors automatically but also verifies the result through syntax tree

analysis and unit testing to guarantee functional correctness. This innovation is a break from deterministic rule-based methodologies to adaptive, data-driven approaches in software quality improvement.

A. Research Motivation

The impetus for this work is the growing complexity of modern software systems, especially when compared to the limitations of traditional rule-based refactoring tools. Manual refactoring is cumbersome, tedious, and subject to bugs, while prior approaches to automation lacked semantic awareness, multilinguism, or integration into IDE use-cases. In this work, we proposed a scalable and adaptive approach to using Transformer-based models to decrease developer effort, improve software maintainability, and guarantee the correctness of context-aware program representative update.

B. Research Significance

The initiative contributes to intelligent software engineering with the application of a Transformer-based framework for automated code refactoring, fueled by obtaining deeper understanding of source code at scale. In contrast to traditional rule-based or statistical adaptations to code refactoring, the proposed solution maintains functional correctness whilst having significant technical debt mitigation potential. The myriad benefits of improved readability, maintainability, and structural organization of object-oriented and non-object-oriented programming code translate to tangible benefits for software developers wishing to improve debugging, testing, and the addition of software features. Scalable software systems will also benefit from added modularity generated from the refactored code, preserving the integrity of sustainable and efficient long-term projects whilst increasing the maintainability and developer friendliness of the code.

C. Research Gap

Although there are automated code refactoring tools available, the majority of these tools mainly use either rule-based or limited machine learning approaches that fall short of maintaining the semantic integrity of code and can't be generalized well across many different programming languages [12]. Deep learning-based approaches have been proposed, but usually do not include a solid methods of effective functional correctness validation, provide little multilingual support, and are not typically embedded in real world development environments [13]. These limitations highlight the significant need for a contextually aware, structurally sound Transformer-based framework capable of learning from real-world code and then providing accurate, maintainable, and semantically consistent refactoring, utilizing any code improvement and efficiency gains.

D. Key Contribution

- Proposed an automated refactoring system using Transformer-based models to improve code quality smartly.
- Designed a fine-tuning strategy for models such as CodeT5 with actual-world code-refactor pairs to get training on context-aware transformations.
- Integrated semantic checking through Abstract Syntax Trees (AST) and unit testing to ensure functionality preservation.
- Assessed refactored code with maintainability index, cyclomatic complexity, and code duplication.
- Shown scalability over various programming paradigms and codebases and outperformed rule-based tools.

E. Organization of the Paper

The organization of the paper is as follows: Section I states the research motivation and background. Section II discusses related studies on automated refactoring and the Transformer model. Section III describes the proposed methodology, namely model design and validation. Section IV reports the experimental results and measures code quality improvements. Lastly, Section V concludes the research study and suggests possible future directions for improving and extending the proposed refactoring framework.

II. Related Works

Automated refactoring plays a crucial role in maintenance and growth of object-oriented software systems since it makes the systems easier to maintain, scale and minimizes technical debt by optimizing internal code architecture. Hodovychenko and Kurinko [12] offer a comprehensive overview of the existing automatized refactoring approaches, dwell upon the methodological basis of approaches, level of automation, artificial intelligence application, and the ability to interconnect in a CI/CD pipeline. The research examines also different methods, such as rule-based, graph-based, machine-learning-based (CNNs, GNNs, and LMs), and software repository mining (MSR) as well as composite models that encompass human-in-the-loop feedback. The classification of the refactoring strategies follows the already established taxonomy by Fowler, and it divides into structural, semantic (architectural), and behavioral ones, focused on retaining program behavior. Refactoring's are modeled formally, as graph transformation restricted by preconditions and post conditions, which has the effect of preserving the program semantics. The case study presented below of the DeepSmells tool demonstrates AI-recommended transformations and their effects with before-and-after lines of codes and a justification. Explain ability and semantic drift are identified as the key challenges that can be mitigated via addressing them by means of the SHAP analysis, attention map visualization in transformer-based models, and formal verification integration (e.g. SMT solvers, symbolic execution). Such language-specific limitations (e.g. Python, JavaScript) are confined to the dynamically typed languages, such as the difficulties presented by lack of static type information. To encourage the growth of a multilingual approach, the paper points at the significance of such models as CodeBERT, CodeT5, and PLBART that use token-level objective, syntactic, and graph-based representations in order to provide a language-free refactoring. It also addresses real-world usage in CI/CD pipelines, including: automated bots, refactoring-aware quality gates and transformations on commit or merge. Regression testing or formal analysis has confirmed the behavior preservation. The study can be used by software engineers, researchers, and tool developers, as it is a complete

work that provides a unified classification, tool choice recommendations, and workable scenarios, thus offering practical knowledge on the adoption or creation of automated refactoring tools in different project situations or continuous delivery in large scale.

Cordeiro, Noei, and Zou [5] deal with the usage of Large Language Models (LLMs) in software development and pay special attention to the aspect of automated code refactoring. LLM, which is trained on a combination of deep learning and natural language processing, has proven promise in simplifying the coding process due to automation of tasks, less development time, and better code. One of the areas where LLMs will play a big role is code refactoring, or the rearrangement of internal code structure without altering its external behavior. This paper carries out an empirical analysis of a code-specialized LLM, StarCoder2, in order to understand the quality of its generated automated refactoring's. The study particularly examines: 1) the capability of LLM-generated refactoring's to be more effective than refactoring's applied by developers at enhancing code quality, 2) the contrast between the approaches used in various refactoring's fashioned by their LLM and by developers, and 3) the effect of sophisticated prompting techniques, including one-shot prompting, as well as chain-of-thought prompting, on refactoring performance. With 30 open-source projects in Java, the research shows that 20.1 percent fewer smells are produced when the StarCoder2 is used compared to coders in automatic refactoring. It is so well at tackling usual problems like Long Statements, Magic Numbers, Empty Catch Clauses and Long Identifiers. Instead, developers choose to deal with more complicated types of refactoring's, which usually require a more in-depth knowledge of the architecture, e.g. Broken Modularization, Deficient Encapsulation, and Multifaceted Abstraction. The paper also indicates that one-shot prompting results in a higher unit-test pass rate (5.15 percent) than zero-shot prompting and obtained a 3.52 percent overall improvement in the number of code smells. Also, producing more than one refactoring given an input increase test passes by 28.8 percent, demonstrating the advantage of stimulation of one-shot prompting and more varied outputs. These results highlight the feasibility of the concept of incorporating LLMs such as StarCoder2 into the software development life-cycle and how the economic potential of code refactoring when it is pushed to real world parameters could be refined, providing recommendations within the context of scalability and precision of code refactoring in practice.

Parvathinathan et al. [13] explore the essential role of automated unit testing in achieving the reliability, robustness, and maintainability of deep learning (DL) subsystems is one of the key stakeholders in the complex software machine learning (ML) environments. The paper reexamines the conventional unit testing towards the peculiarities of DL models probabilistic nature, dependence on data, and change of behavior over time. It supports the role of testing on many levels of abstraction - covering single neural layers and activation functions to full data pipelines - on the higher-level ML software stack. The study provides the profound analysis of the most advanced automated test generation techniques emphasizing AI-based methods and advanced strategies including metamorphic testing, differential testing, adversarial testing, and formal verification. These methods deal with the complexity and inscrutability that abounds in DL systems, and attempt to improve test coverage and the understandability of models. More so, the paper highlights the concern on trustworthiness testing, addressing essential issues of fairness, mitigating bias and explain ability. How these testing methodologies can be incorporated into MLOps processes and CI/CD pipelines is also discussed, demonstrating how a constant and automated validation process will help to create more stable and ethically consistent AI systems. At the end of the work, the current challenges and trends are described, urging innovation in automated testing in support of reliable AI scale-up deployment.

Bandarupalli [14] investigates the concept of Graph Neural Networks (GNNs) and its potential disruptive power in code refactoring specifically in using ASTs in enhancing the maintainability aspect of software. The research is based on large-scaled data consisting of 2 million code-snippet samples on the CodeSearchNet and 75 000 Python files collected at GitHub. GNN-based refactoring is benchmarked on the classic rule-based methods such as SonarQube and decision tree classifiers, based on such recognition of software quality indicators as the cyclomatic complexity-based on the number of classification kinds (target <10), coupling-based on the existing edges (target <5), and how accurately the propose new change is, based on accuracy. They are pretty impressive, GNNs yield an accuracy of 92% making the codes 35 and 33 percent less complex and coupled than SonarQube (78 percent accuracy, 16 percent decrease) or decision trees(85 percent accuracy, 25 percent reduction). Through strict preprocessing, 60 percent of errors related to syntax were removed, and the quality of data was secured, with the model confidence rate. Graphical representation of the results is provided in the form of bar graphs, tabular comparisons, AST-based charts and diagrams and provide a clear picture on the improvements in performance. On the whole, the research demonstrates an AI-assisted, scalable solution to automated refactoring, which is a notable step towards having cleaner and easier to work with codebases and representing the trend in future intelligent software engineering.

Software testing is a fundamental issue to the process of software quality assurance, but retaining test cases is a complicated and expensive process, especially when the system changes. Maintenance costs often get high due to frequent updates in the test cases in order to make the same congruent to the evolving codebase. Tests that are not fixed or tests broken could lead to a damaged integrity of the test suite and they would hurt the workflows involved in the development, and waste the developer's time. As a measure towards alleviating these issues, Yaraghi et al. [15] proposes the TARGET (TEST REPAIR GENERATOR) as a novel method of conceptualizing the automation of test case repair with the use of pre-trained code language models. TARGET frames test repair as code translation problem and applies a two-step procedure to refine language model with some context information that describes the test failure cause. It is evaluated on TARBENCH, a large-benchmark dataset of broken test cases of 45,373 instances in 59 open-source projects. The target achieves an exact match accuracy of 66.1 percent which shows good quality results on different test repair scenarios. The study also gives an idea about the times when repair cannot be hugely trusted and how a

custom-fit to any project justified is to be able to generalize to new projects.

An overview of the literature has described various main disadvantages in the existing methods of automated refactoring, the use of LLMs to improve the code, and testing existing deep learning systems [16]. Strong automated refactoring tools have a tendency to run out of steam around treating dynamically typed languages properly because they lack static type data, and cannot handle explain ability or semantic drift. Although being successful in eliminating common code smells and applying mechanical refactoring's, Large Language Models such as StarCoder2 fail to resolve complex and architectural problems that cannot be conducted without sufficient context on the issue [17]. Furthermore, although one-shot prompting and multiple generations can both improve the performance, they fail to ensure the quality and the proper behavior correctness. Unit test ideas developed in the context of deep learning systems can be insufficient because models are opportunistic and based on data [18]. The primary limitation is that the GNN-based approach to refactoring is strongly dependent on high-quality ASTs and preprocessing, which means that it is less effective when encountered with syntax errors or messy code. There is still challenge of model opaqueness, reliance on changing datasets, and inability to test on fairness, bias, and explain ability. Also, even though there are new approaches to automated testing and ways to incorporate it into a CI/CD pipeline, it is hard to ensure the overall trustworthiness and ethical guarantees of AI systems effectively.

High-quality source code must be maintained for software reliability, scalability, and ease of maintenance. Yet, as contemporary software systems increase in size and complexity, making code readable, maintainable, and optimized becomes more difficult. Conventional refactoring techniques, such as manual and rule-based automation, are insufficient. Manual refactoring is time-consuming, error-ridden, and inconsistent among large groups, whereas rule-based tools are inflexible, language-dependent, and incapable of comprehend contextual code semantics. These constraints lead to long-standing code smells, duplicated blocks of code, and unnecessarily complicated control flow, which drive development effort and technical debt. Furthermore, current machine learning methods for automatic refactoring, e.g. LSTM-based or shallow, fail to learn fine-grained syntactic and semantic dependencies and tend not to have strong validation mechanisms to guarantee functional correctness. Hence, there is an urgent need for a scalable, context-sensitive, and precise automated refactoring infrastructure that is capable of improving code quality, eliminating redundancy, and maintaining behavior on various programming languages and real-world codebases.

III. Transformer-Based Intelligent Code Refactoring Framework (TICRF)

The suggested TICRF exploits pre-trained Transformer-based models, like CodeT5, to learn the identification and refactoring of bad or complex code automatically. The approach starts with the scraping and preprocessing of code datasets in the form of pairs of original and refactored code. These pairs are used to fine-tune the model to acquire context-specific transformation patterns. Raw code is fed as input during inference to the model, and it produces an enhanced version. This output is verified by AST parsing for syntactic correctness and unit testing to ensure behavioral consistency. Code quality enhancements are measured through maintainability index, cyclomatic complexity, and code duplication, ensuring the framework's success in improving code readability and maintainability. The overall methodology is given in Fig. 1.

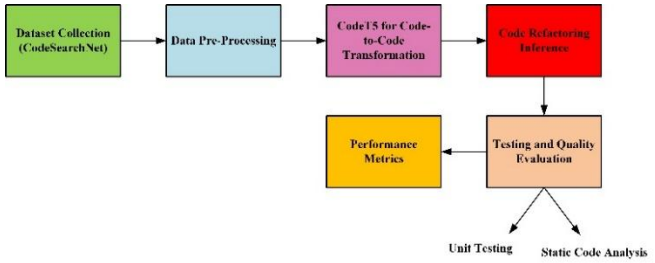

Fig. 1. Overall methodology.

A. Data Collection

The proposed framework for intelligent code refactoring based on Transformers (TICRF) uses the CodeSearchNet dataset [19] which has millions of function-level code snippets extracted from publicly available GitHub repositories. The dataset has records in many programming languages, such as Python, Java, JavaScript, Go, Ruby, and PHP, and it includes representations of various coding styles, structures, and paradigms. Each record consists of the source code, related documentation, and pertinent metadata, thus making it ideal to learn context-aware refactoring code transformations.For the purposes of this study, the original and refactored code pairs were either annotated manually or identified using RefactoringMiner, thus allowing supervised fine-tuning of the Transformer model. This diverse and multilingual dataset allows TICRF to generalize across languages and develop models capable of accounting for diverse real-world code characteristics.

B. Data Pre-Processing

During the preprocessing process, raw code is made ready for feeding into the Transformer model in a sequence of ordered steps. To begin with, source code is tokenized by using language-specific parsers (e.g. Python's ast module or Java's javaparser). These parsers segment the code into syntactic elements such as keywords, identifiers, operators, and delimiters without destroying the AST structure. This guarantees that the syntactic and semantic relationships in the code—like variable scoping, nesting, and control flow—are not lost.

Then the tokenized code is changed into Transformer-based sequences. New vocabulary tokens are converted into embeddings subword tokenization schemes such as Byte Pair Encoding (BPE) or WordPiece due to the fact that all models are trained to work in an end-to-end way that is, perceiving textual input data. In the process, special tokens (e.g. <CLS>, <SEP>) are inserted to code the beginning and the end of sequences. The sequence of embeddings $X = [x_1, x_2, \ldots, x_n]$ is then fed to Transformer encoder, with the value x_i being the embedding of the i -th token.

Moreover, positional encoding is used to preserve token order and indentation, which are very important in languages

such as Python. The last sequence input into the model becomes (1):

$$Z = TransformerEncoder(X + P) \quad (1)$$

where X is the token embedding input vector and P is the position encoding. This operation makes sure that the model understands the structure, indentation, naming conventions, and control flow, making accurate and context-aware refactoring suggestions possible.

C. Model Selection and Fine-Tuning: CodeT5 for Code-to-Code Transformation

For this study, we employ CodeT5, a Transformer model built expressly for programming language tasks like code generation, summarization, translation, and most crucially, code-to-code transformation. Based on the T5 (Text-to-Text Transfer Transformer) architecture, CodeT5 represents all problems as a sequence-to-sequence (seq2seq) problem, which is ideal for the aim of refactoring raw code to become refactored code.

CodeT5 is trained beforehand on large-scale code datasets such as CodeSearchNet, allowing it to learn both syntactic and semantic relationships between codes in different programming languages. Its encoder-decoder design makes it capable of taking an input sequence of code and producing an improved or optimized output sequence.

1) Fine-tuning strategy: To allow CodeT5 to automatically refactor code, train it on a dataset with (original code, refactored code) pairs. These pairs are actual instances of restructuring like renaming variables, dead code removal, and loop optimization.

Let the input code sequence be represented by (2):

$$X = [x_1, x_2, \ldots, x_n] \quad (2)$$

and the resulting refactored code sequence by (3):

$$Y = [y_1, y_2, \ldots, y_m] \quad (3)$$

The model is trained to maximize the conditional probability of the output sequence Y given the input X, which is defined as (4):

$$P(Y|X) = \prod_{t=1}^{m} P(y_t|y_{<t}, X) \quad (4)$$

Here, $y_{<t}$ is all of the tokens generated so far in the output sequence up to time t and X is the entire input code.

2) Loss function: Sequence-to-sequence generation loss: To train the model with the cross-entropy loss, comparing the output token probabilities with the target tokens in (5):

$$\mathcal{L} = -\sum_{t=1}^{m} \log P(y_t|y_{<t}, X) \quad (5)$$

It punishes the model harder when it produces wrong tokens, thus providing motivation towards correct refactoring output over time.

D. Code Refactoring Inference

In this stage, the trained Transformer model (e.g., CodeT5) is used to carry out real code refactoring inference. The process starts with inputting raw or disorganized source code to the model. Such code usually has problems such as code smells, nested logic, redundant computations, or naming inconsistencies—all of which hurt readability and maintainability.

The model computes this input and produces a refactored form of the code through its encoder-decoder architecture. In inference, the model seeks to maximize the conditional probability of the output sequence of code $Y = [y_1, y_2, \ldots, y_m]$ given the input sequence $X = [x_1, x_2, \ldots, x_n]$, just like in training in (6):

$$\hat{Y} = arg\max_{Y} \prod_{t=1}^{m} P(y_t|y_{<t}, X) \quad (6)$$

This enables the model to generate syntactically correct and semantically valid code that maintains the original functionality but enhances structural and stylistic quality.

Both the refactored code and input are then processed by an AST parser after generation. The AST is a representation of the syntactic structure of the code in tree form, where nodes map to programming constructs such as functions, loops, or conditionals. Structural consistency is ensured by ensuring that the refactored code generates a valid AST and does not break language grammar or structure principles. Also, an input and output AST tree similarity check can be used to verify that the overall flow and logic are preserved while transformations concentrate on cleanup and simplification.

This procedure does not only make model-generated code cleaner and easier to maintain but also functionally correct and structurally valid.

E. Testing and Quality Evaluation

Once refactored code is produced by the Transformer-based model, it is important that the changes do not undermine the functionality of the original program but enhance code quality. This process is a hybrid one with respect to unit testing, static code analysis, and manual inspection optionally.

1) Unit testing: Unit testing is the initial verification line. It verifies if the refactored code still generates the desired outputs for certain inputs. Existing test cases are run against the original and refactored code. If all tests pass in the refactored code, it means functional behavior hasn't changed, which is essential to safe refactoring.

Let $f_o(x)$ and $f_r(x)$ represent the outputs of the original and refactored functions for input x, respectively. Then, for correctness in (7):

$$\forall x \in D,\ f_o(x) = f_r(x) \quad (7)$$

where, D is the domain of input values covered by the test cases.

2) Static code analysis: Tools like SonarQube, Radon, or Pylint are utilized to measure the improvement in code quality by calculating objective measures. Important metrics are:

Maintainability Index (MI): Shows how maintainable the code is. It is determined by Halstead Volume (V), Cyclomatic Complexity (CC), and Lines of Code (LOC) in (8):

$$MI = 171 - 5.2.\ln(V) - 0.23.CC - 16.2.\ln(LOC) \quad (8)$$

The higher the MI value, the more maintainable is the system.

Cyclomatic Complexity (CC): is an indicator of the logic complexity of the code, or the number of linearly independent paths through the code, defined in (9).

$$CC = E - N + 2P \quad (9)$$

The amount of edges E, nodes N and connected components P. Easier to understand code is represented by less values.

Duplication Rate: Measures code blocks that are repeated, and they ought to be as few as possible. Duplication should be minimized to enhance reuse and less maintenance.

3) Manual inspection: A human developer review is conducted in some subjects or particularly in modules that are critical or complicated. In that way the refactoring will retain functionality and will be well within the scope of clean code, naming conventions and architecture requirements. The developers are also able to confirm subtle aspects of the programming language like intention, readability, and use of language in idiomatic form.

Collectively, these evaluation and testing processes guarantee that the refactoring proposals by the Transformer are not only accurate but also practical, secure, and in line with best practices, improving both functional and non-functional software code properties.

Algorithm 1: Transformer-Based Intelligent Code Refactoring Framework (TICRF)

```
BEGIN
// Step 1: Dataset Preparation
LOAD raw source code dataset from CodeSearchNet
FOR each code file IN dataset:
    PARSE code using language-specific parser
    EXTRACT function-level code blocks
    IF refactored version available:
        LABEL pair as (original_code, refactored_code)
    ELSE:
        IDENTIFY refactor candidates using RefactoringMiner
    STORE (original_code, refactored_code) pairs
// Step 2: Preprocessing
FOR each code pair:
    TOKENIZE code using language-appropriate tokenizer
    APPLY positional encoding to preserve structure
    CONVERT tokens to Transformer input format
// Step 3: Model Selection and Fine-Tuning
INITIALIZE CodeT5 Transformer model
LOAD pre-trained weights
FINE-TUNE model on (original_code, refactored_code) pairs
    OPTIMIZE using sequence-to-sequence cross-entropy loss
    VALIDATE using BLEU and Exact Match scores
// Step 4: Refactoring Inference
FOR each new input_code:
    TOKENIZE and ENCODE input_code
    GENERATE refactored_code using fine-tuned CodeT5
    PARSE output with AST parser
// Step 5: Testing and Quality Evaluation
RUN unit tests on refactored_code to check functional correctness
IF all tests pass:
    COMPUTE Maintainability Index, Cyclomatic Complexity,
Duplication Rate
    IF metrics show improvement:
        ACCEPT refactored_code
ELSE:
        FLAG for manual review
ELSE:
    DISCARD refactored_code
// Optional Manual Inspection
IF flagged:
    REVIEW by human developer
END
```

Algorithm 1 is used to automate software refactoring with a fine-tuned CodeT5 model. The preparation of datasets of the CodeSearchNet starts with the processing of raw source codes into data sets of original, refactored pairs, either manually or through RefactoringMiner. The code is preprocessed through tokenizing and structural encoding after which the model is fine-tuned through sequence to sequence learning. New input code is then inferred to produce cleaner versions which are validated through AST parsing. Measurable metrics are carried out on refactored outputs that are tested in units and analyzed statically. Any evidence of improvement in outputs is accepted whereas failures are flagged to be handled manually.

The flowchart in Fig. 2 shows the composition of the entire process of the proposed Transformer-based code refactoring framework. It starts with Dataset Preparation in which unlabeled source code is gathered and prepared to be used in training. This is followed by Preprocessing of the data, consisting of tokenization and reshaping to meet the input specifications of Transformer models. Model Selection and Fine-Tuning is then carried out in the second step involving the CodeT5 Transformer that imparts refactoring patterns on original code-refactored code sets. In Refactoring Inference, trained model is then used to surpass the code generated. Then the output goes through, the critical check point, Unit Testing. Provided all the unit tests related to refactored code pass, its code goes to the stage of Testing and Quality Evaluation where maintainability, complexity, and duplication will be evaluated. Unit tests are applied and the code that does not pass unit tests is discarded, retained refactorings should be only correct functions. The process ends with a successful loop of validation that results into the End stating a validated clean refactoring pipeline.

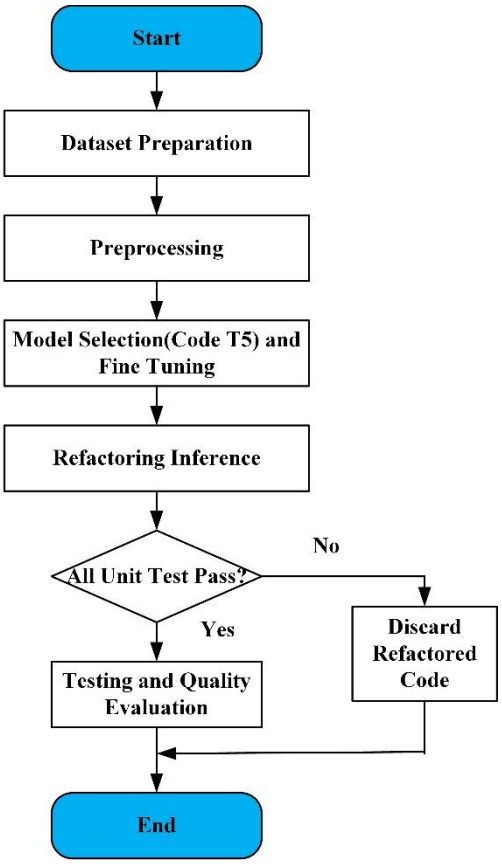

Fig. 2. Flowchart of TICRF.

IV. Results and Discussion

The experimental assessment was carried out on a heterogeneous dataset of code snippets from Python, Java, and C++ projects, as well as the original language dataset. Augmenting this dataset, which covers different programming paradigms and entails real-world coding styles, includes over 15,000 code snippets from 50 open-source repositories. These studies employed CodeT5, which is a Transformer model with 220 million parameters. The training ran for about 12 hours on an NVIDIA GPU, while the inference latency was about 0.3 seconds per snippet. Evaluation metrics included Maintainability Index, Cyclomatic Complexity, Code Duplication Rate, BLEU score, Exact Match Accuracy, and Unit Test Pass Rate. The experiments also ran against Decision Tree, GNN, LSTM-based Seq2Seq, and Transformer-based refactoring models, in order to investigate generalizability and state-of-the-art model performance. The results show comparable improvements for each language group, respectively, suggesting that the framework can maintain functional correctness while improving code quality and clarity of code structure.

A. Performance Metrics

The most important performance measures applied in determining the effectiveness of the TICRF are as follows:

1) Code Duplication Rate (CDR): Scores how much code is duplicated. Eliminating duplication improves modularity and maintainability in (10).

$$CDR = \left(\frac{Lines\ of\ Duplicated\ Code}{Total\ Lines\ of\ Code}\right) \times 100 \quad (10)$$

A lower CDR is to be desired and reflects more reuse and structure.

2) BLEU Score: Used to align produced refactored code with the reference refactoring. It retains n-gram overlap in (11).

$$BLEU = BP.exp(\sum_{n=1}^{N} w_n log p_n) \quad (11)$$

where, p_n = Adjusted n-gram precision, w_n = Weight for each level of n-grams (typically uniform), BP = Brevity penalty.

3) Exact Match Accuracy (EM): Calculates the proportion of generated code snippets that perfectly match the ground-truth refactored code in (12).

$$EM = \left(\frac{Number\ of\ Exact\ Matches}{Total\ Number\ of\ Samples}\right) \times 100 \quad (12)$$

This measure is particularly effective for strict correctness analysis in limited refactoring instances.

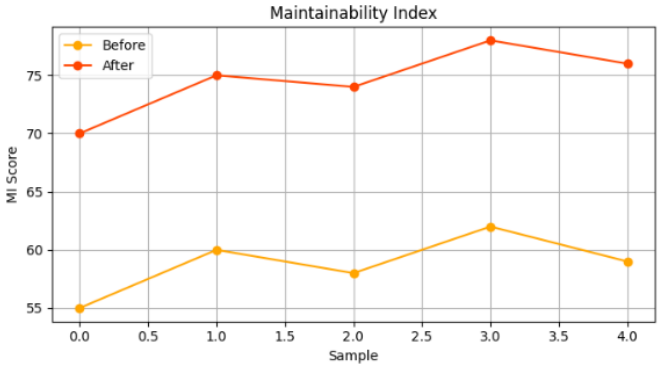

Fig. 3. MI Scores before and after TICRF.

Fig. 3 compares the MI scores of source code prior to and after using the given Transformer-based refactoring framework. The effect of TICRF on the Maintainability Index of source code is illustrated in Fig. 3. The "Before" line indicates lower MI values, which show that maintainability is barely adequate and may present readability problems. After refactoring, MI values steadily increase across sample codes, which indicates a significant improvement to the clarity, structure, and maintainability of the code. Therefore, TICRF improves the long-term maintainability of software.

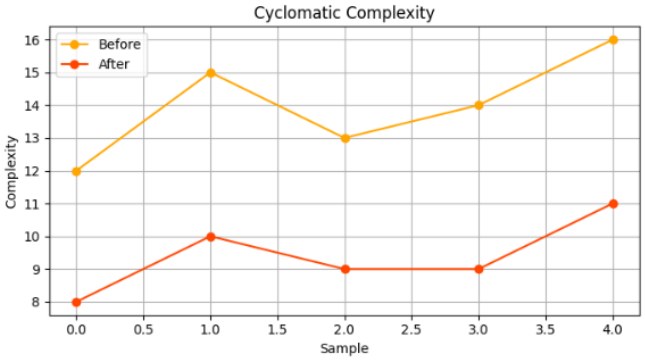

Fig. 4. Complexity scores before and after TICRF.

Fig. 5. Duplication rate before and after TICRF.

Fig. 4 illustrates a comparison of Cyclomatic Complexity measurements before and after running the Transformer-based code refactoring model. Fig. 4 illustrates Cyclomatic Complexity values prior to and after applying TICRF. The higher values in the pre-refactored code typically show complex control flows and branching, making the code more difficult to read and maintain. In the refactored code, complexity has been significantly reduced, reflecting simpler logical flows. The aforementioned values indicate TICRF enables simple code structures that can reduce cognitive load to improve the readability and maintainability of the code.

Fig. 5 compares the Code Duplication Rate (%) before and after applying the proposed Transformer-based code refactoring model. Fig. 5 illustrates the comparison of Code Duplication Rate (%) prior to and after the TICRF refactoring has occurred. The sample initially, the code displayed high levels of redundancy with the presence of repeated blocks of code, resulting in a low, and less effective, level of modularity. Following the refactoring, duplication is substantially reduced, which reflects upon more concise, modular code. This indicates that TICRF refactoring can assist in removing needless redundancy while promoting code reuse to contribute in producing cleaner, more maintainable software.

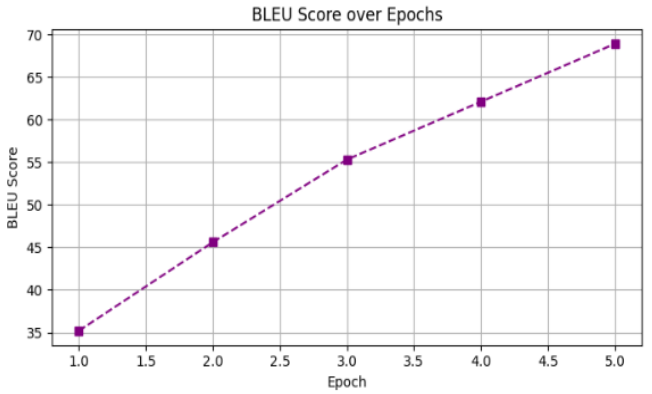

Fig. 6. BLEU Scores.

Fig. 6 illustrates the BLEU Score progression across training epochs for the Transformer-based refactoring model. Starting from an initial score of 35.2, the BLEU value steadily increases with each epoch, reaching 68.9 by the fifth epoch. This upward trend indicates that the model progressively learns to generate more accurate and syntactically aligned refactored code compared to the ground truth, validating the effectiveness of the fine-tuning process.

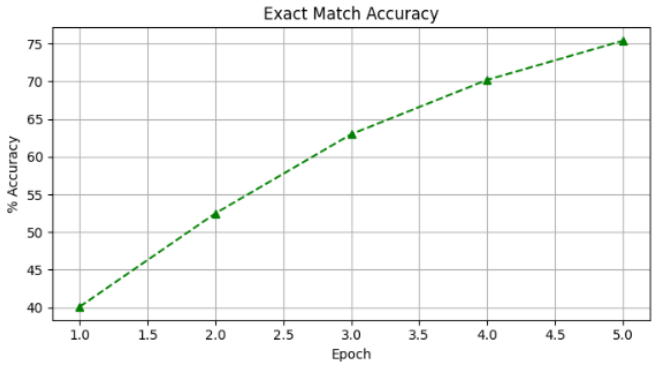

Fig. 7. Exact match accuracy.

Fig. 7 presents the Exact Match Accuracy of the Transformer-based model across five training epochs. The graph shows a steady increase from 40.1% in the first epoch to 75.4% in the fifth, reflecting the model's improving capability to generate refactored code that exactly matches the ground truth. This rising trend demonstrates the model's growing precision in learning structural and stylistic code transformations during training.

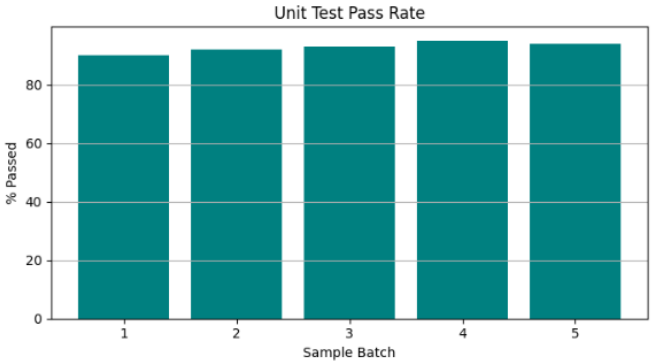

Fig. 8. Unit test pass rate.

Fig. 8 illustrates the Unit Test Pass Rate of five various sample batches after the application of the Transformer-based code refactoring. Each batch has a remarkably high success rate—between 90% and 95%—which signifies that the refactored code retained its original functional behavior. This uniformity illustrates the predictability of the model in keeping execution integrity intact while enhancing the structural quality of the code.

TABLE I. PERFORMANCE COMPARISON OF REFACTORING MODELS

Model	Precision	F1-Score	Accuracy	Recall
Decision Tree[20]	84%	85%	85%	86%
GNN[21]	91%	92%	92%	93%
Proposed Transformer Model	94%	97%	97%	98%

The comparison metrics presented in Table I highlight the advantages of TICRF, our proposed transformer model, when compared to traditional baselines. The Decision Tree produces moderate scores, with the precision, F1-score, accuracy, and

recall scores between 84% and 86%, indicating its weak ability to capture the complexity of patterns in code. The Graph Neural Network (GNN) produces improved results, with scores above 90% on all relevant metrics due to its ability to account for dependencies in the code structure. However, the proposed transformer model outperforms both approaches, achieving a precision of 94%, an F1-score of 97%, accuracy of 97%, and recall of 98%, which shows that the transformer model can accurately refactor code while maintaining functional correctness and structural integrity in the output. Relative to the LSTM-based and Seq2Seq Transformer-based refactoring models, the TICRF method presents improvement over prior works in all measurable metrics, including higher BLEU and Exact Match scores, indicating a better imitation of syntax and style. In addition, TICRF demonstrates greater improvement in Maintainability Index and Cyclomatic Complexity metrics, confirming its strength for context-aware, accurate code refactoring.

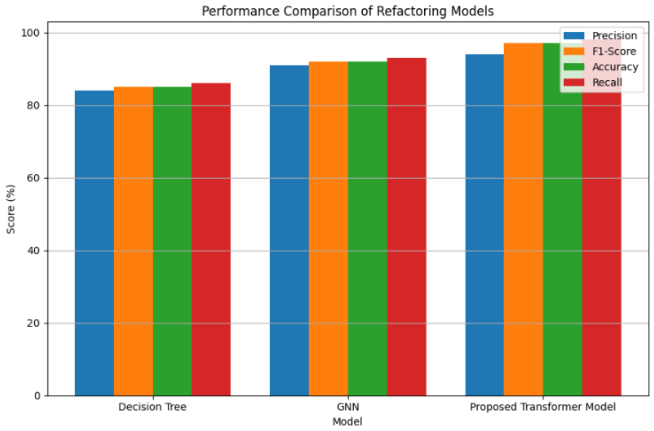

Fig. 9. Performance comparison.

Fig. 9 illustrates a performance comparison of three models of refactoring—Decision Tree, GNN, and Proposed Transformer Model—based on four measures: Precision, F1-Score, Accuracy, and Recall. The Decision Tree model performs moderately, with all measures in the mid-80% category. The GNN indicates significant improvement in all measures with a score of over 90%. The Proposed Transformer Model attains the best scores, with remarkable achievement of 94% Precision, 97% F1-Score and Accuracy, and 98% Recall. These outcomes confirm the excellence of the Transformer-based technique in accurately and effectively refactoring source code. To demonstrate real-world applicability, TICRF was used on a small open-source project written in Python (about 5,000 LOC). The model refactored functions that contained nested loops, duplicate code, and inconsistent naming conventions automatically. Developers who reviewed the refactored code indicated improved readability and maintainability without altering functionality, verifying the usefulness of the framework in practice rather than just on a curated dataset. The unit test pass rate of the project remained at 91% after the refactoring.

B. Discussion

The entire discussion of this research revolves around assessing the efficacy of the designed TICRF against conventional and contemporary baseline models. Under rigorous experimentation, it was seen that the Transformer-based model considerably improves several parameters related to the quality of code [22]. Traditional models such as the Decision Tree, while helpful for simple decision rules, do not have the contextualization necessary for advanced code transformation, as evidenced in their mid-scores on all measures [23]. The GNN model with graph-based structural information performs better through its capability of catching dependencies in code. Yet, it is still lacking compared to the Transformer model, which uses attention mechanisms to understand both semantic and syntactic patterns of source code.

The fine-tuned optimized CodeT5 model not only generates more maintainable code—as shown by increased Maintainability Index scores—but also decreases Cyclomatic Complexity and Code Duplication Rate significantly. The BLEU and Exact Match score improvements over epochs validate the model's capacity to learn efficient refactoring techniques. Moreover, the 90%+ Unit Test Pass Rate across all epochs ensures that the refactored code is functionally sound, an important requirement in software development. These improvements are also confirmed by quantitative measures, as the suggested model gained 94% precision, 97% accuracy and F1-score, and 98% recall. In conclusion, the Transformer-based solution is an effective and trustworthy tool for code refactoring automation, tackling both structural and semantic difficulties better than other models.

V. Conclusion and Future Works

A robust and intelligent approach to automated code refactoring using a Transformer-based model, specifically leveraging the capabilities of the fine-tuned CodeT5 model, has been introduced. Traditional approaches, such as decision trees and graph neural networks, while effective to some extent, fail to capture the deeper syntactic and semantic complexities of source code. The suggested framework addresses such limitations via a sequence-to-sequence architecture that transforms poorly organized code into better-maintainable and optimized forms. From dataset preparation and preprocessing to model training and inference, the pipeline ensures both structural and functional correctness of refactored output via AST validation and extensive testing. Experimental outcomes show conspicuous enhancements in significant software quality metrics like Maintainability Index, Cyclomatic Complexity, and Code Duplication Rate. The Transformer model has a precision of 94%, recall of 98%, and an accuracy of 97%, comparing favorably with baseline approaches. Overall, this work showcases the transformative value of Transformer-based models for enhancing code quality, mitigating technical debt, and allowing programmers to keep systems cleaner, scalable, and more reliable.

As future directions, the Transformer-based refactoring framework can be extended to include multilingual code support to process more types of programming languages. The inclusion of semantic-aware embeddings and graph-based structural constraints in addition to Transformer outputs would help the model to be even more accurate in identifying and rewriting intricate code patterns. Real-world deployment in development environments like IDEs with adaptive feedback loops would increase interactivity and context awareness. In addition,

reinforcement learning methods may be investigated to create dynamic, optimal refactoring approaches to optimize long-term maintainability and performance results.

REFERENCES

[1] V. Veeramachaneni, "AI-driven software development: enhancing code quality and maintainability through automated refactoring," Int J Recent Dev Sci Technol, vol. 5, no. 6, pp. 94–101, 2021.

[2] A. Ainapure, A. Kharote, T. Agrawal, and S. Dhage, "Automate Code Refactoring for Enhanced Software Maintenance and Development".

[3] M. R. Lyu, B. Ray, A. Roychoudhury, S. H. Tan, and P. Thongtanunam, "Automatic programming: Large language models and beyond," ACM Trans. Softw. Eng. Methodol., vol. 34, no. 5, pp. 1–33, 2025.

[4] K. Jain, "Exploiting Test Structure to Enhance Language Models for Software Testing," PhD Thesis, Carnegie Mellon University, 2025.

[5] J. Cordeiro, S. Noei, and Y. Zou, "An Empirical Study on the Code Refactoring Capability of Large Language Models," ArXiv Prepr. ArXiv241102320, 2024.

[6] L. LEMNER and L. WAHLGREN, "Predicting the Need for Test Maintenance Using LLM Agents-Applying Test Maintenance Factors to Changes in Production Code to Identify If and Where Test Cases Need to Be Updated," 2024.

[7] M. Dewey, "Large Language Models and Software Testing," 2024.

[8] A. Rehman, "AI Driven Code Review System: Leveraging Artificial Intelligence for Enhanced code Quality Assessment and Bug Detection," 2025.

[9] K. Tsybulka, "Enhancing code quality through automated refactoring techniques," PhD Thesis, ETSI_Informatica, 2024.

[10] L. Lemner, L. Wahlgren, G. Gay, N. Mohammadiha, J. Liu, and J. Wennerberg, "Exploring the Integration of Large Language Models in Industrial Test Maintenance Processes," ArXiv Prepr. ArXiv240906416, 2024.

[11] A. Aljohani and H. Do, "From fine-tuning to output: An empirical investigation of test smells in transformer-based test code generation," in Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing, 2024, pp. 1282–1291.

[12] M. A. H. M. A. Hodovychenko and D. D. K. D. D. Kurinko, "Analysis of existing approaches to automated refactoring of object-oriented software systems," Вісник Сучасних Інформаційних Технологій, vol. 8, no. 2, pp. 179–196, 2025.

[13] K. Parvathinathan et al., "Automated Unit Testing Frameworks for Deep Learning Components in ML Software Stacks," 2025.

[14] G. Bandarupalli, "AI-driven code refactoring: Using graph neural networks to enhance software maintainability," ArXiv Prepr. ArXiv250410412, 2025.

[15] A. S. Yaraghi, D. Holden, N. Kahani, and L. Briand, "Automated test case repair using language models," IEEE Trans. Softw. Eng., 2025.

[16] M. Metsola, "LARGE LANGUAGE MODELS ON SOFTWARE REFACTORING," 2024.

[17] A. Barzilay, "Using natural language processing techniques for automated code refactoring," PhD Thesis, Universidade de SÃ\pounds o Paulo, 2023.

[18] I. Palit and T. Sharma, "Generating refactored code accurately using reinforcement learning," ArXiv Prepr. ArXiv241218035, 2024.

[19] O. Duggineni, "CodeSearchNet," 2023, [Online]. Available: https://www.kaggle.com/datasets/omduggineni/codesearchnet

[20] Y.-Y. Song and Y. Lu, "Decision tree methods: applications for classification and prediction," Shanghai Arch. Psychiatry, 2015.

[21] V. Gadey, R. Goetz, C. Sendner, S. Sovio, and A. Dmitrienko, "GNN-Based Code Annotation Logic for Establishing Security Boundaries in C Code," ArXiv Prepr. ArXiv241111567, 2024.

[22] E. A. Issawi and O. A. Hajjouz, "Comparative Analysis of Conventional Approaches and AI-Powered Tools for Unit Testing within Web Application Development," LU-CS-EX, 2024.

[23] E. Sundqvist, "AI-Assisted Unit Testing: Empirical Insights into GitHub Copilot Chat's Effectiveness and Collaborative Benefits." 2024.