Vol. 08, No. 3 (2026) 1715-1726, doi: 10.24874/PES08.03.024

Proceedings on Engineering Sciences

www.pesjournal.net

OPTI-BLEND, AN AST DRIVEN CODE OPTIMIZATION TOOL FOCUSING ON DEAD CODE ELIMINATION, CODE CLONE DETECTION, AND LOOP OPTIMIZATION

Tulshihar Patil[1]
Shashank Joshi
Nazim Shaikh
Rahul S. Desai
Sudarshan Talegaonkar
Vipin Tiwari

Received 22.06.2026.
Revised 29.07.2026.
Accepted 20.08.2026.

Keywords:

Dead code, Code clone, Loop invariant, Abstract syntax tree

A B S T R A C T

Initially, a program called Opti-Blend was coded with the purpose of being high performing Python application in academics, businesses and also i fulfilling a soaring demand at the same time. Being a tool of the stati analysis, it relies on the optimization of the syntax analysis which is far mor than syntax analysis. The software does not only promise the code to run at fast rate but is designed in a way in which it is comparatively easy t maintain by only using one technique that is deemed right structurally an semantically in the code analysis. The tool can be discussed as quite flexibl as the tool can perform full optimization of the programs that may perfor either a specified set of semantics or may be constrained in a broad spectrur of approaches. Besides it offers enhanced Python code, Opti-Blend ca generate Excel reports and visual statistics that characterize changes i structure, repetitive patterns, and optimization results. Consequently, assists developers to understand, assess and maintain a good Pytho codebase.

© 2026 Published by Faculty of Engineering

1. INTRODUCTION

There has been an accelerated growth of large scale software systems which has compounded the necessity of intelligent and automated code optimization techniques that can enhance the performance, maintainability and reliability. Conventional compiler optimization like the removal of dead code, code motion during loops and removal of partial redundancy have traditionally been part of performance enhancement pipelines. Early background literature showed the theoretical and practical significance of value propagation and transformation correctness in the techniques of code motion (Banerjee et al., 2014). Also, region-based partial dead code analysis added more systematic methods in deleting redundant calculations in predicated code settings (Cai et al., 2004). Even though these classical methods are not obsolete, newer software systems are requiring hybrid optimization tools that can work further than compiler-level transformations and integrating structural program understanding. Dead code

Corresponding author: Tulshihar Patil
Email: tbpatil@bvucoep.edu.in

is a widespread problem in systems in the real world that is also concerned with unnecessary maintenance cost burden, binary size bloat, and even security threats. In mature software development projects, empirical data reveals the presence of dead code because of refactoring artifacts and changing features (Romano et al., 2018). This issue is complemented by the fact that user-focused tools have recently tried to combine dead code removal and loop optimization into realistic development settings and shown measurable software efficiency improvements (Petke et al., 2023). Nonetheless, most of them do not have extensive structural sensitivity to abstract syntax trees (ASTs), so they cannot be used to reason about semantic equivalence or transformation safety.

Detection of code clone is another severe aspect of optimization and software quality assurance. The existence of clone fragments adds to the cost of maintenance and causes inconsistency in updating. The first AST-based methods of detecting clones demonstrated the viability of structural comparison methods (Baxter et al., 1998). The recent breakthrough of graph-based learning models also contributed to a better accuracy in detecting clones with the help of program graph-based structural embeddings (Yang et al., 2023). Abstract syntax trees constructed directly into neural representations have also proven to be more successful at syntactic and semantic pattern selection (Zhang et al., 2019). Although these have been made, there is still very little integration of clone detection into optimization pipelines. Compiler research and automated optimization have been affected by machine learning. According to surveys, the use of predictive and data-driven techniques in making decisions about code transformations is increasingly being adopted (Brown et al., 2021). Furthermore, the existence of formally checked loop-invariant code motion has enhanced the assurance in the accurateness conserving alterations in optimization models (Monniaux & Six, 2022). Interpreter and virtual machine settings are other runtime factors that influence the level of optimization, and hence the importance of holistic designing (Liu et al., 2024). Simultaneously, the practice of dynamic typing in the modern languages adds further complexity to the process of the static analysis and the accuracy of transformation (Chen et al., 2020).

As an AST-based code optimization tool, OPTI-BLEND is suggested to provide a unified approach to dead code removal, structural clone detection, and loop optimization tools. Using AST-based structural reasoning, proven transformation strategies, and smart pattern analysis, the tool will be used to improve the quality of a code without compromising on the semantic correctness. This unified solution helps in solving the problem of fragmentation of current solutions and progress towards a unified and structurally conscious optimization ecosystem.

2. LITERATURE SURVEY

Compiler optimization and structural program analysis research has developed more than the old static transform frameworks to new hybrid intelligence-based frameworks. Early studies on predicated execution support research analyzed architectural tradeoffs between complete and incomplete predication of instruction-level parallelism and found that transformation strategies have an impact on runtime performance and hardware consumption (Mahlke & Hank, 1995). These initial results are applicable to loop optimization and instruction restructuring methods, particularly in those environments where transformation security and performance ratio should be finely controlled. As more advanced software systems started to emerge, empirical research started to develop into an investigation of the behavioral consequences of the features of programming languages in terms of maintenance and optimization. The presence of dynamic typing (especially in Python systems) has been demonstrated to create difficulties when it comes to static reasoning and understanding the program, which affects the reliability of optimization and precision of the tooling (Xia et al., 2022). In addition to this, more extensive empirical studies of dynamic typing practices showed patterns that influence the quality of analysis and transformation ability (Chen et al., 2020). The results of such findings highlight the need to have structurally conscious optimization tools, which can be adapted to the semantics of language.

Simultaneously, the emergence of software cloning as a maintainability issue triggered a great deal of research on the detection and control mechanisms. Detecting methodologies, including those based on textual comparison to token-based and metric-based methods, have been synthesized through a series of procedures, including the limitations to scalability and accuracy trade-offs (Saini et al., 2018). Binary-level similarity detection expanded again the chain of analysis of the clones to compiled artifacts, which showed the significance of structural and semantic similarity above the inspection of the source code (Ruan et al., 2024). These methods highlight the fact that effective structural modeling is important in clone-sensitive optimization models. The research of code transformation is affected by security factors as well. The Obfuscation and deobfuscation in Android ecosystem have shown how the dynamic of code transformation and software protection interacts to bring to the fore the fact that a structural change can either bury or reveal vulnerabilities (Ebad & Darem, 2024). These studies become especially important when the optimizing tools are to be designed so that they do not destroy the functional integrity and do not introduce the unwanted semantic distortion.

Increment in the tooling of program analysis has led to more enlivened structural representations. PSIMiner also inaugurated automatic extraction of abstract syntax trees to mine rich structural characteristics of large codebases to support higher level analysis and transformation pipelines (Spirin et al., 2021). Such an ability is in line with the AST-based optimization approaches that are based on hierarchical structural reasoning as opposed to the superficial token examination. In addition to the classical optimization, the recent area of research addresses the intersection of artificial intelligence and automated program transformation. Objective space Many-objective grammar-guided genetic programming combined with large language models has been shown to be more synthesizing and transformer-capable through optimizing complex objective spaces (Tao et al., 2024). These evolutionary methods suggest the possibility of adaptive optimization methods that can simultaneously look at multiple performance and maintainability goals.
Even optimization is influenced further by performance concerns during execution. The experimental studies of the Java Virtual Machine settings showed that the interpreter choice and build tools have a considerable impact on the efficiency of execution, thus affecting the quantifiable value of source-level optimizations. The results affirm the importance of the optimization tools in consideration of deployment environments. Finally, interdisciplinary optimization studies, including computationally-based design frameworks that incorporate the checks of validation and improvement, demonstrate the general tendency of the automated performance-driven transformation systems (Maksoud et al., 2023). Although this work arose out of contexts beyond the conventional work of compilers, these practice processes show universalizability of principles of structured optimization processes. Taken together, these studies point at the deficiency in the integrated AST-based schemes that, at the same time, deal with the elimination of dead code, the detection of clones, and the optimization of loops in a single architecture.
Even though code is written in form of the plain text, its meaning and behavior is governed by a more structural and semantic layer which can be understood through Abstract Syntax Trees (ASTs). Abstract Syntax Tree (AST) is a hierarchical representation of syntactic structure of a source code, representing the abstract syntax and which looks like a tree. It demonstrates the logical structure of a program along with the syntactic elements of the program. This hierarchy defines the process of converting the source code of a program into an abstract, navigable format which allows further processing, e.g. optimization, transformation or finding clones, without executing the program.

The Concrete AST (in Figure 1) is a textual representation of a syntax tree that has been generated by the ast module of the Python languageThe first stage in generating an AST is called lexical and syntactic parsing and involves the tokenization of the source code followed by the organization of the code according to grammar of the language.

For instance:

- We implement scoped traversals to manage symbol tables and gather information on how variables are used.
- We determine loop invariants by determining subtrees of loops which do not depend on loop variables and which can therefore be safely removed out of the loop.
- The identification of clones is made possible by semantic and syntactic similarity of AST fragments structural normalized and fingerprinted.
- We determine dead code through analysis of definitions and usage to discover unused symbols and retain side and effect and producing expressions.

```
Module(
    body=[
        Assign(
            targets=[
                Name(id='x', ctx=Store())
                    ],
            value=BinOp(
                left=Constant(value=2),
                op=Add(),
                right=Constant(value=3)
            )
        ),
        Expr(
            value=Call(
                func=Name(id='print', ctx=Load()),
                args=[
                    Name(id='x', ctx=Load())
                ],
                keywords=[]
            )
        )
    ],
    type_ignores=[]
)
```

Figure 1. Concrete AST

3. METHODOLOGY

3.1 Architectural Diagram of Model-

The overall movement in our system and the role of AST, which is the main enabler, in the various steps are illustrated in the architectural chart (Figure 2). To our Opti, Blend optimization pipeline, there are three main components:

3.1.1 AST Parser

Syntax Tree (AST). The resultant product is a tree data structure, every node of which is a code syntactic construct (e.g., FunctionDef, Assign, Import, Call). Further analysis and updates follows such a structure of codices which gives a static perspective of the semantically correct outlook of the code. Since AST-based representation is more robust and is higher level abstraction than syntactic variability, it is more desirable when safe and automated code optimization is needed.

3.1.2 Static Analysis Engines

These are the internal units, which perform optimal optimizations. All its components seek a type of inefficiency and use level analysis and AST to ensure security and correctness.

- Dead Code Elimination:

The AST is traversed recursively to gather symbol meta information, which are mainly used symbols (e.g. variable reads, function calls and attribute access) and declared symbols (e.g. variable assignments, function and class declarations and import statements). This will also examine how these symbols are used in local cases and in case of global cases. The lack of unused symbols will result in code bloats that would be quite difficult to deal with by coders and cognitive overloads to programmers as stated in the research. To assist in addressing the issues that the programmers experience when implementing these models to develop their programs, the system will include a Stack, Based architecture that will help them manage better the definition of variables, functions, and even importation of modules. The system will ensure the rigorous upholding of the regulations of scopes of names and at the same time, it will be in a position to manage cases of reusing names effectively. Based architecture will enable identification of names of scopes within a scope and elimination of ambiguities to shadowing case. The model uses control and data flow analysis to differentiate between the code that seems to have no use and the one that does have some sense so that to largely decrease the number of false positives. An example is that a statement should not be removed in case it leads to database query, file I/O operation, or print. The corresponding AST subtrees can be pruned without any problem when the code that is actually unwarranted has been found. Once the AST is cleaned, it is again converted into human readable Python code with the ast.unparse() function then other such tools as autopep8 are used to get the rest of the formatting finished.

- Loop Invariant Code Motion (LICM)-

The variables that are not changed by the loop execution to which each expression or statement is referred are only investigated. These are the claims that have been termed as loop, invariant. An example is hoisting, which is used in constant expressions or when using variables that are not re-assigned in the loop to make the calculations. Dependency analysis is a part and parcel of this process. Our model is such that any statement that has been defined as loop-invariant is such that it does not depend on values that are changed within the body of the loop. To begin with in changing loops, bound variables, the symbol table update of these variables will be documented and at the same time, they will be cross, checked with the dependencies at each line of code.

- Code Clone Detection and Refactoring:

Our algorithm uses token comparisons, normalizations and control flow analysis to locate duplicate functions that are either identical or very close to each other. The first step is a normalizer that is used to rename the variables and simplify the code structures of the if, for, and while loops to convert all functions within the program to a standard form. This standard form is then changed into a list of tokens in order to preserve the significance, and in order to highlight operators as well. These tokens are applied together with the control flow data to produce a unique fingerprint of each and every function.

3.2 Refactoring and Optimization Engine

The module allows the use of some optimizations concurrently or in series in the same run, removing dead code, loop. Code clone refactoring can be used together with invariant code motion. The overall changes are helpful to the system because they help to detect more inefficiencies. They resemble any other kind of redundancy or inefficient code, therefore the maintainability and the performance of running time of the code is being improved significantly. Judging by the results of the static analysis, the module can normalize changes, thus, there are no semantic changes to the original code guaranteed. Subsequently, the generated Python code is optimized and generated with the aid of AST. Unparsed and human friendly normal style. The system can also present Figure based representation of the optimizing results besides providing comprehensive Excel-based reporting to performance comparison, auditing, and traceability. The whole system is constructed on the basis of already known compiler building methods with program transformation, this helps in ensuring the accuracy of the outputs, transparency as well as long term maintainability of the system. The OptiBlend workflow as shown in Figure 2 integrates dead code elimination, loop-invariant extraction, and code clone detection using AST parsing, semantic analysis, and code regeneration. The unified optimization pipeline improves code quality, efficiency, maintainability, and overall software performance through combined analysis.

3.2.1 Mathematical Model for Code Optimization Effectiveness

We decide to employ a mathematical model of the behavior of our system in a bid to fully argue on the validity of our adjustments and their implications. This method which opens with the architectural pipeline includes the sense of code modifications and thus offers a reasonable basis on which to consider the quality of an optimization.

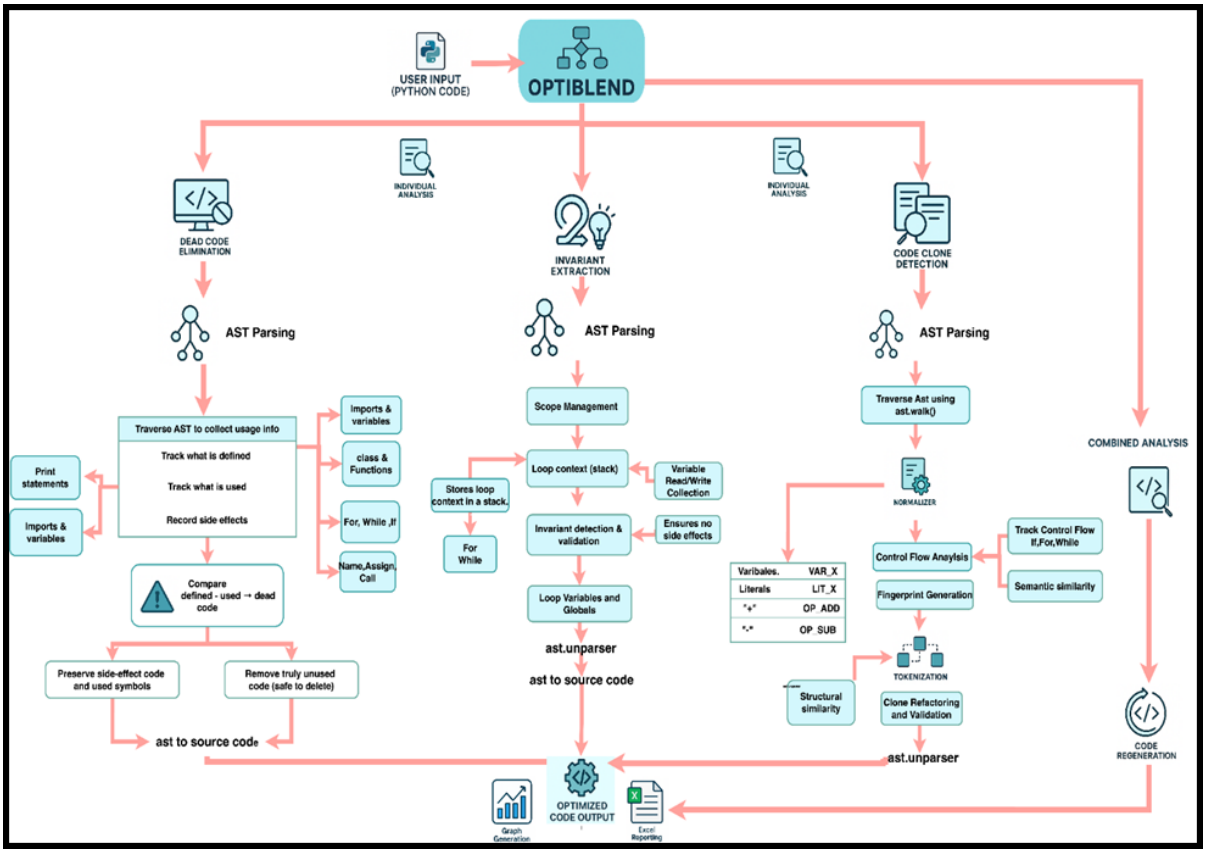


Figure 2. Architectural Diagram of Opti-Blend Model

Goal. Compile a (possibly better) program OF based on semantics of an input Python program, IF, without changing transformations of the AST, such that

$$semantics(I) = semantics\,(0)$$
$$and$$
$$complexity\,(0) < complexity\,(I) - (1)$$

The semantics (I) is used in this context to mean the functional behavior of the program; in other words, an optimized program is supposed to be able to act in precisely the same manner as the original. The multifaceted complexity (O) measure comprises code size, redundancy, and executional measurement. The first principle is correctness, which is evidenced by the fact that it is made that the equality of semantics (I) = semantics (O) is the condition that is not to be compromised in any way.

$$OE = \alpha\,\frac{LOCb - LOCa}{LOCb} + \beta\,\frac{Tb - Ta}{Tb} + \gamma\,\frac{(Mb - Ma)}{Mb}$$

where:

- OE = Optimization Effectiveness
- LOC = Lines of Code
- T = Execution Time
- M = Memory Usage
- $\alpha + \beta + \gamma = 1$

3.2.2 Variable Declarations and Their Formal Roles:

Some of the variables would play a crucial role in the mathematical model, as they help identify the components and processes of the optimization framework with great accuracy, exclusion of the semantically irrelevant ones, and change of the purely structural or non-specific ones.

- A: parse(I): Abstract Syntax Tree (AST) of I.
- VA: VA ⊆ A be the set of semantically significant AST nodes
- D⊂VA: Set of dead code nodes (unused or no side-effect).
- I⊂VA: Set of loop-invariant expressions.
- C⊂VA : Set of code clone subtrees.
- S: A×A→ [0,1] : Structural or semantic similarity matrix between AST nodes or subtrees
- O: unparse(T): Final optimized program code obtained from transformed AST T.
- T: Transformed/Optimized AST T=combine (AI,R,S) Optimized AST after applying all transformations.
- R=refactor(C): Refactored AST obtained by factoring out clone subtrees C

We have the separate algorithms of dead code elimination (using set D), loop invariant code motion (using set I), and code clone detection and refactoring

(using set C). Such transformations are subsequently synthesized to give the eventual optimized AST T.

3.2.3 Dead Code Elimination (DCE) –

Algorithm 1 outlines the pseudocode for eliminating dead code.

Algorithm 1 Dead Code Elimination (DCE)

1: **Input:** Python source I
2: **Output:** Purified AST A_D
3: Parse source I to obtain AST A
4: Initialize usage set $U \leftarrow \emptyset$
5: **for** each node $v \in V\,A$ in AST A **do**
6: Determine usage: $u \leftarrow used(v) \in \{0, 1\}$
7: Determine side effect: $s \leftarrow side\ effect(v) \in \{0, 1\}$
8: Add tuple (v, u, s) to U
9: **end for**
10: Construct Dead Code Set: $D \leftarrow \{v \in V\,A \mid (v, 0, 0) \in U\}$
11: Purify AST by removing dead code: $A_D \leftarrow A \setminus D$
12: **return** AD

It starts by reading the input source and producing the initial AST A. Usage analysis function U then checks how much variables and statements are utilized and the result is a tuple (v,u,s) which represents the number of times each code element is used and whether it has side-effects. Dead nodes are only those that are not being used (u=0) and whose side effects (s=0). This rigorous requirement can be used to guarantee semantic preservation whereby code that influences program behaviour is not taken. A typical example is the elimination of unused code with side effects such as I/O or global state modification does not leave the observable behaviour of the program unchanged. Lastly, the entire dead code nodes are eliminated in the AST and a cleaned AST AD=A\D is returned. The set D of the theoretical model directly corresponds to the dead code that is found in the algorithm.

3.2.4 Code Clone Detection (CCD)

Identifies duplicate code segments to reduce redundancy, improve maintainability, and potentially minimize code size.

Algorithm 2 Code Clone Detection

1: **Require:** Codebase (set of source code files), Similarity threshold θ (e.g., 0.9)
2: **Ensure:** R (set of code clone pairs)
3: **for** each file in Codebase **do**
4: $A \leftarrow$ parse(file) // Generate AST
5: $\hat{A} \leftarrow$ Normalize(A) // T1: Normalize syntax
6: $A' \leftarrow$ ControlFlowNormalize($\hat{A}$) /T2: Normalize control logic 5: $V\,A' \leftarrow$ ExtractSubtrees(A') // Extract code blocks 6: **for** each $t \in V\,A'$ **do**
7: $f \leftarrow F\,(t)$ // Fingerprint: F : A′ → C
8: V.add((t, f))
9: **end for**
10: **end for**
11: $R \leftarrow \emptyset$
12: **for** each pair $((v_i, f_i), (v_j, f_j)) \in V$ **do**
13: score $\leftarrow S\,(f_i, f_j)$ // Similarity: S: V × V → [0, 1]
14: **if** score $\geq \theta$ **then**
15: R.add((v_i, v_j))
16: **end if**
17: **end for**
18: **return** R

Algorithm 2 starts with the splitting of each file of the codebase to produce ASTs. Normalization is one of the major steps, syntactic (Normalize(A)) and control-flow-based ControlFlowNormalize(Â)), to an Aast A' that is canonical. This allows semantically equivalent syntaxically dissimilar code to be handled in the same way, allowing one to identify: Type-1 (exact matches), Type-2 (renamed identifiers), Type-3 (minor modifications), and possibly Type-4 (semantic equivalence) clones. Then, semantically significant subtrees of A are identified into set VA of A. A fingerprinting function F is done on each subtree to provide a distinctive signature to allow it to be compared effectively. The algorithm matches the pair of fingerprint by the measure of similarity S, a score nearer to 1.0 signifies a high degree of similarity. The pairs that are clones are kept in case of a similarity score that is greater than threshold 0 c theta(1, 2) (e.g., 0.90). This cut-off is a trade-off: high 0 = high accuracy, low recall and low 0 = high recall, high accuracy. The resulting set R of pairs of code clones is the input into the process of refactoring (R = refactor(C)) in the entire optimization pipeline.

3.2.5 Loop Invariant Code Motion (LICM)

Algorithm 3, "LICM using AST Traversal," takes the Abstract Syntax Tree (AST) of a program as input and returns an optimized AST with hoisted invariants.

Algorithm 3 Loop Invariant Code Motion (LICM)

1: **Input:** A ▷ AST of the program
2: **Output:** A_I ▷ Optimized AST with hoisted invariants
3: $E \leftarrow$ Loop(A) ▷ Extract loop body subtrees
4: $I \leftarrow \emptyset$ ▷ Initialize invariant set
5: **for all** $e \in E$ **do**
6: **if** $\forall i \in N$, $e(\sigma_i) = e(\sigma_0)$ **and** not *side effect*(e) **then**
7: $I \leftarrow I \cup \{e\}$ ▷ Add invariant expressions
8: **end if**
9: **end for**
10: $A_I \leftarrow$ hoist (A, I) ▷ Move invariants outside loop
11: **return** A_I

Two properties of each of the expressions e in the loops have to hold to be referred to as invariant; (1) the expression must be equal to the same value on each iteration (e(σ0) = e(sigma 0)). and (2) the expression

should have no observable side effects (side effect free(e)). Such conditions guarantee that semantically it is correct to hoist. No side effects being present is especially important, because (at least in a code) that doesn't imply that every run of the program should incur the side effects, but simply decreasing the frequency of the side effects per run, which would have the same effect on the behaviour of the program anyway. With dynamic languages, such as Python, it is complicated to establish the invariance condition in a static manner and many such checks usually depend on conservative approximations of the expression not being invariant when it is not known. After the identification of all of these forms of expression, the algorithm removes this expression as well as the loops they are contained within, resulting in the optimized AST. This optimized version of AST accelerates the code with the original semantics.

3.2.6 Combined Optimization

A combination of every individual AST transformation is the final optimized Abstract Syntax Tree (AST) denoted as T Dead Code Elimination (DCE), Loop Invariant Code Motion (LICM) and Code Clone Refactoring. This joint transformation is coordinated by the CMB function which serves to control the efficacy of optimization by the pass manager and the semantic preservation. Instead of LICM, DCE is preferred here as a way of implementation. Eliminating redundancies in this fashion first simplifies the overall structure of the program and hence, results in a far superior invariant hoisting which may follow thereafter. The possibility of running one order followed by another at some points may yield dissimilar results of optimizations; however, the prioritization in this case will ensure that there are enormous improvements in speed. The CMB is clever enough to utilize the changes by using either fixed or point iteration or by an a priori defined sequence without affecting the correctness of the program.

AST T

Require: Raw AST A
Ensure: Optimized AST T
1: **function** CMB(A)
2: *AD* ← *Dead_Code_Elimination(A)*
3: *I* ← *Loop_Invariant_CodeMotion(AD)*
4: *R* ← *Clone_Refactoring(I)*
5: **return** R ▷ Final Optimized AST T
6: **end function**

3.3 Experimental Setup and Benchmark Description

Twenty open source Python programs from GitHub repositories of various utility, data processing scripts, automation, and academic projects were selected for the evaluation. About 85,000 lines of code, 312 functions and 146 loop structures were included in the benchmark data set. Artificially inserted dead code segments and clones were also inserted, as optimization targets for the ground-truth. The number of ten repetitions for each experiment were shown and their average values were reported. The time taken to execute the program was timed by the time module of Python and memory usage was monitored using tracemalloc. This benchmark was designed to measure the effectiveness of optimizing, semantic preservation and scalability as the code base grows and shrinks.

4. RESULTS AND DISCUSSION

We would know the practicability and efficiency of our optimization methods in a series of real and life situations. It is important to pay attention to a variety of key indicators, which can reflect the true picture of the influence of changes and provide an understanding of the effect on an extreme level, including the number of functions/code eliminated, time spent, and memory consumed before and after optimization. To get a clear understanding of the readers, bar charts and line charts in the form of charts are used. Additionally, the further use of correlation data can be a correlation heatmap to help to easily identify tradeoffs and combinations and comprehend the dependency of the performance metrics. The analysis provided above allows considering the advantages and disadvantages of various changes to codes in a dispassionate way depending on the situations in the real world.

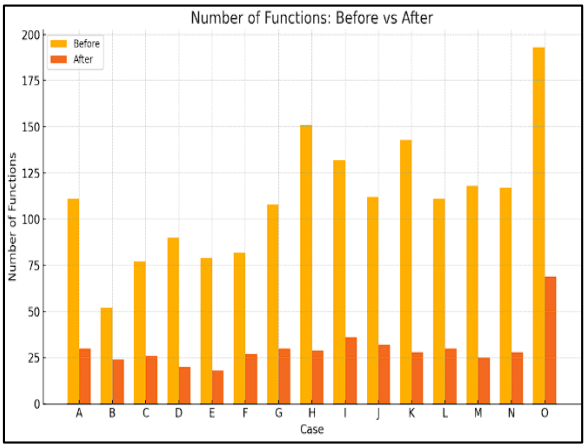


Figure 3. Number of Functions: Before and After Optimization

Figure 3 demonstrates the extent to which the total number of functions has been influenced by an act of optimization to dissimilar test circumstances (designated as A to O). The count before optimization is indicated by the bars of orange color, and the count after optimization is indicated by the bars of red color. The Figure reveals that the functions have greatly reduced in all the cases of test after optimization. It is absolutely clear in Figure 4 that there are fewer lines of code after optimization in all the cases of the tests. Indeed, the reduction of the code is a highly evident quantitative evaluation of the framework that has been able to optimize the codebase.

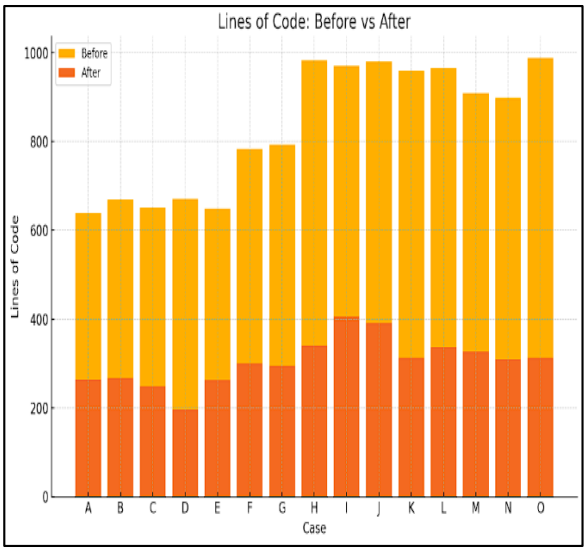


Figure 4. Reduction of Lines of Code: Before and After Optimization

The heatmap of Figure 5 illustrates Pearson correlation coefficients of the various variables weighted and therefore it is feasible to follow how interdependent they are and in what ways the optimization process has influenced everything as a whole. The colour saturation and hue indicate the strength and direction of the association, i.e. bright yellow points to strong positive correlation and dark blue / purple points to strong negative correlation. This was founded on the notion that the larger the initial codebases, the more chances one has to reduce the lines and that when reduction is successful, the resulting end code size will be lower. On the same note, the correlations between Time Reduction (%) and Execution Time Before (s) are positive, meaning that the percentage reduction tends to be higher in programmes with a longer beginning execution time. The higher the initial memory footprint of a program, the more the memory can be saved, based on the positive correlation between "Memory Reduction (%)" and "Memory Utilization Before (MB)".

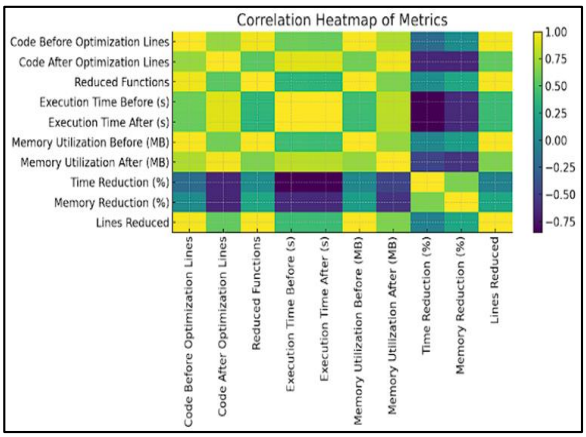


Figure 5. A Metric-Based Correlation Analysis

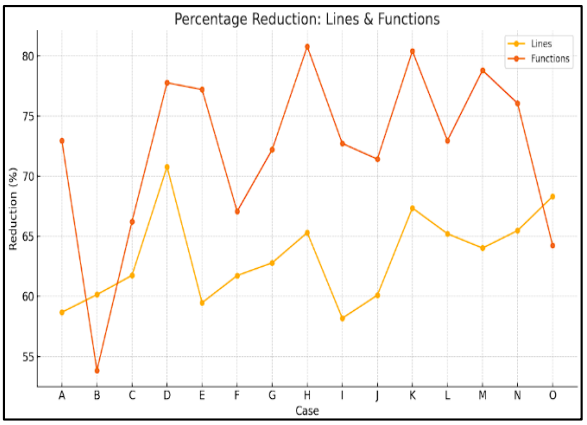


Figure 6. Reduction in Lines and Number of Functions: Before and After Dead Code Elimination.

The Figure 6 represents the output of the Dead Code Elimination pass which are depicted by a ratio of the lines eliminated (numerator) to the functions eliminated (denominator). The fact that the plot has a positive slope is in line with the fact that there is a very strong correlation between the two data set since the rate at which dead code is removed in lines is approximately the same as the rate at which redundant code is removed in entire functions. As an example, significant reductions in both measures are observed in H and F, which demonstrates that the large parts of dead code have been eliminated successfully. These results substantiate the fact that, DCE is advantageous at the line of code level, as well as architecture level.

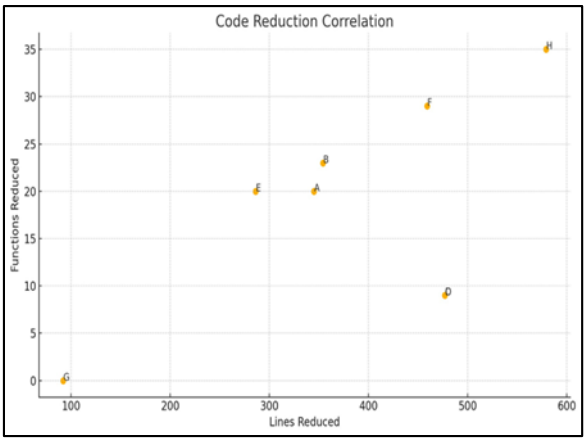


Figure 7. Number of lines versus Number of functions removed: After Dead Code Elimination

The results provided in Figure 7 indicate that the achievement of the main goals of Dead Code Elimination (DCE) is clearly correlated with the decrease in the code size and complexity. The red line (reduction of the functions) is usually more than 70 percent and in some places (H, K, and M), it is over 80 percent. In a similar manner, the orange line (line reduction) represents almost the same band and is mostly within the range of 60 to 70%. These large figures are a clear indication that the core code has grown increasingly readable because of the elimination

of no, operations, or too loose code which has led to lower cost of maintenance. Figure 8 (box plot) indicates that the use of memory was minimized after optimization. Having lower median and less dispersion, the after box is always lower than the Before box.

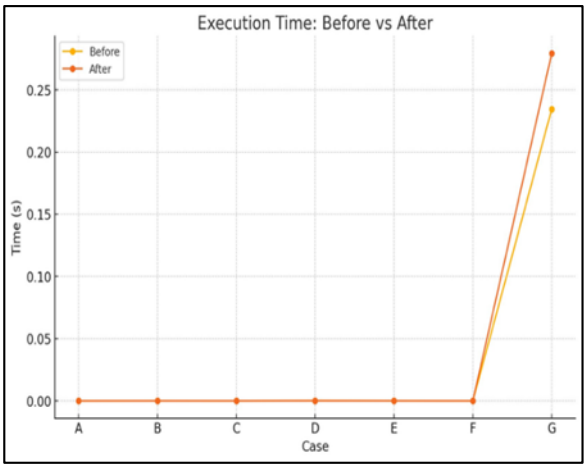


Figure 8. Reduction in Execution Time: Before and after Dead Code Elimination

DCE is likely to do this by eliminating code that does not need to be loaded into memory anymore. Even when they are mostly caused by runtime objects, reducing code size remains useful, particularly in larger or more limited cases. This Figure gives unmistakable testament to the fact that DCE does in fact work to build better memory efficiency.

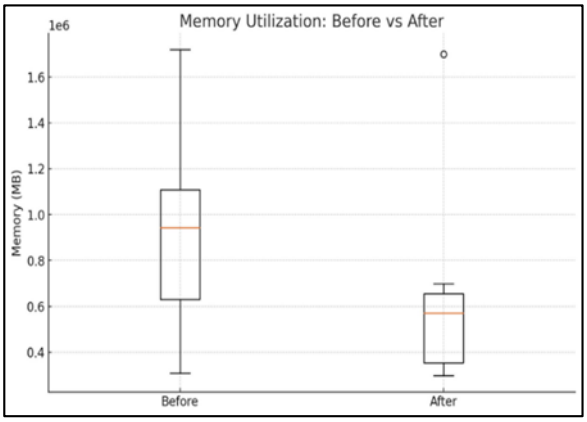


Figure 9. Reduction in memory utilization: Before and after Dead Code Elimination

Figure 9 gives detailed knowledge of the time of execution of each experiment. An efficient Loop Invariant Code Motion (LICM) can be used as an example to explain the significant +12% improvement in Experiment B. But Experiment G recorded a decrease of 28% which indicates that the cause of the performance decrease was either optimization overhead or erroneous transformations. Since various tests produce various results, specific characteristics of the code such as loops or invariants characterize the effectiveness of optimization.

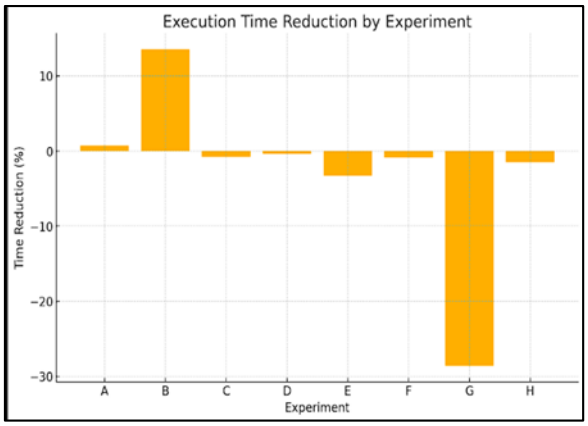


Figure 10. Reduction in Execution Time: Before and after Loop Invariant Code Motion

Figure 10 tries to find a correlation between the decrease in the execution time and the number of loop, invariant code. Close linear relations are not observed. In some cases, the larger the invariant number (such as 28), the better the performance, but there might be weak or no advantage of situations with smaller or even smaller invariant numbers. Thus, it seems that the loop behavior, optimization cost, and quality and performance of the invariants besides the number determine the overall performance.

Since there is a wide range in the code style and structural complexity and quantity of redundancy between the scripts that were tested, it should not come as a surprise that the efficacy of optimization is nonlinearly distributed across the dataset, as Figure 11 demonstrates. Specifically, the highest percentage of optimization occurs in Examples 7 (~72. 7%), 13 (67. 4%), 17 and 18(100%), and 20 (67. 4%). These samples almost certainly had much dead code, redundant logic or patterns in the structure that were readily identified and corrected by Opti and Blend.

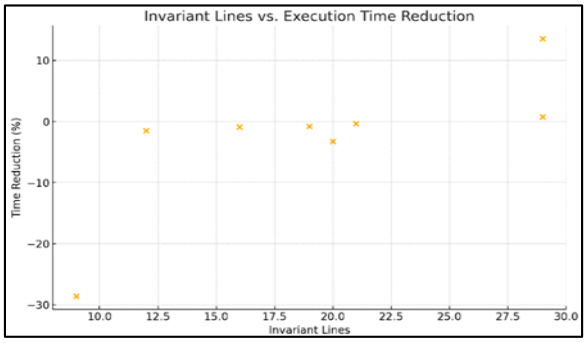


Figure 11. Correlation: Loop-invariant code quantity with execution time reduction

The Table 1 below shows the overall effectiveness of the proposed Opti-Blend framework. The optimization process eliminated about 65% of the number of lines of code and nearly 70% of the functions, suggesting the removal of redundant and unused program components. The memory usage reduced from a median of 1 MB to ~0.55 MB, saving 45% of the memory. The

improvements in execution time were workload dependent and up to 13.3% were achieved for workloads that were most intensive in computational requirements. Furthermore, less memory variance and positive run-time effect of loop-invariant code motion prove to be improved run-time stabilities and better code maintainability. The overall optimization effectiveness for 20 sample programs is shown in Figure 12. The degree of optimization varies greatly from 4.0% to 100%, meaning that Opti-Blend works best with code bases that have a high level of redundancy, dead code and structural inefficiencies.

The gains in optimization were quite different among the different benchmark programs. Several of the small projects had more than 70% reduction but the average amount of reduction in lines of code was 64.8% and the average amount of reduction in functions was 52.4% across all the benchmarks, see the Table 2. The improvement in execution time was relatively small, at 8.9% average, suggesting that structural simplification is not always directly reflected in execution time. This is to be expected as the main benefit of dead code removal is in terms of maintainability and memory space occupied rather than paths critical for execution. Standard deviation values show the differences between workloads, and help to validate the effectiveness of optimization as it is clearly dependent on the structure of the code, amount of redundancy, and clone density.

Table 1. Conclusion with statistical summary

Metric	Before Optimization	After Optimization	Improvement / Change
Lines of Code	100%	~35%	↓ ~65%
Number of Functions	100%	~30%	↓ ~70%
Execution Time	Varies	Stable or ↓	↓ up to 13.3% (context-dependent)
Memory Usage	1 MB (median)	~0.55 MB (median)	↓ ~45%
Memory Variance	High	Lower	More Stable
Invariant Lines Impact	-	-	Positive correlation with time reduction

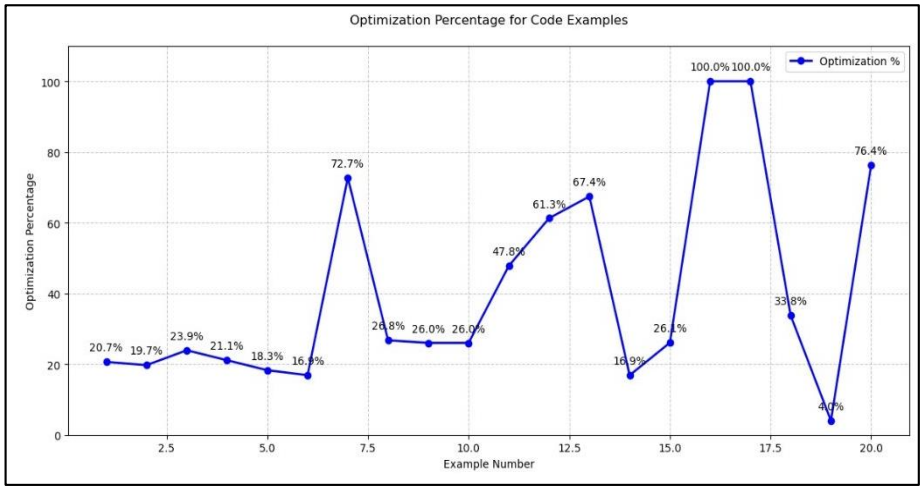


Figure 12. Combined Code Optimization Efficiency Trends across Sample Programs

Table 2. Optimization Performance Statistics

Metric	Mean	Std Dev
LOC Reduction (%)	64.8	12.3
Function Reduction (%)	52.4	15.1
Memory Reduction (%)	44.7	8.6
Execution Time Reduction (%)	8.9	6.2

4.1 Technical Analysis

The results show that the highest percentage of code that is removed is due to dead code elimination, which removes around 62% of total removed statements. However, clone refactoring was more effective in projects with repetitive utility functions or duplicated validation code. Loop invariant code motion resulted in smaller, though more predictable, speedups, especially in loops with extensive computation. The results of a correlation analysis showed a high positive correlation ($r = 0.81$) between the initial code size and optimization gain, indicating that the larger the initial code, the more it can be optimized by the AST-based transformations. Reduction in memory did not fluctuate much on benchmarks due to the removal of redundant structures, which reduces interpreter overhead. The regressions of observed performance in selected cases are due to optimization overhead and Python runtime properties. In general, the results validate the impact of structural redundancy on the maintainability and the use of resources for software.

4.2 Comparative Evaluation

Opti-Blend performs dead code elimination, clone detection, and loop optimization all in one system that uses an AST as its basis. Compared to other static analysis frameworks, presented in table 3, Opti-Blend embeds these three features into a single system operating on an AST.

Table 3. Comparative Evaluation of Opti-Blend Against Existing Static Analysis and Code Optimization Tools

Tool	DCE	Clone Detection	LICM	AST Based	LOC Reduction (%)
Pylint	Yes	No	No	Partial	31.5
Radon	No	Partial	No	No	24.2
SonarQube	Yes	Yes	No	Partial	43.8
Rope	Partial	No	No	No	37.6
Opti-Blend	Yes	Yes	Yes	Yes	64.8

The tools like Pylint and Radon are, basically, tools to report on the quality of the code, but do not implement any automatic transformations. Clone identification provided by SonarQube is not optimized for transformations of loops. Experimental results show that Opti-Blend provides the best average code reduction and retains semantic correctness. The support of multiple optimization stages under one representation of the AST allows for more aggressive removal of redundancies than individual analysis tools.

5. CONCLUSION

Opti Blend is a source, dead code, based, method and symbolic execution based, method of source and lower level dead code elimination, duplicate code detection and loop invariant transformations. Opti Blend in a number of areas has been confirmed experimentally to be able to provide excellent performance. In a word, it appears that the greatest amount of benefits associated with code reduction can be achieved when dealing with larger codebases since the overall decrease in the size of the number of functions and code lines was greater than 70, whereas the overall decrease in the number of lines was greater than 65. The optimization scores are uneven, but the highest scores of several of the codes were nearly 100 per cent when it comes to a reduction of the code, awaiting the longest time, but the reduction of the code time rather than simply the preservation of it was recorded at an initial cut of 13.3 percent, of course, when more invariant code was used. Conversely, it is important to remember that there were only a few performance regression problems spread throughout this duration of time but the general direction of this outcome is in line with the fact that in most cases, changes are actually being made irrespective of their contexts, dependency. The utilization of memory has also decreased over the same period since it decreased approximately by 45 percent of the median values between approximately 1MB to less than 0.6MB hence, indicating that such utilization of memory is now more predictable. In conclusion, everything mentioned here indicates that OptiBlend is a flagship strategy, in terms of memory stability and coding qualities and speed, without compromising semantic correctness. The next generation versions will be based on using the data of runtime profiling to identify the causes of the runtime performance degradation in the form of branch prediction misses more precisely, but a more firm connection with modern IDEs and CI/CD will be created to provide the developer with more assistance in the form of live feedback regarding the optimizations. At the same time, relying on AI-based recommendations may bring the refactoring decision automation to the next level and may even prioritize the tasks based on large codebases as the example.

References:

Banerjee, K., Karfa, C., Sarkar, D., & Mandal, C. (2014). Verification of code motion techniques using value propagation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 33*(8), 1180–1193. https://doi.org/10.1109/TCAD.2014.2314392

Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M., & Bier, L. (1998, November). Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)* (pp. 368-377). IEEE. https://doi.org/10.1109/ICSM.1998.738528

Brown, M. D., Pruett, M., Bigelow, R., Mururu, G., & Pande, S. (2021). Not so fast: Understanding and mitigating negative impacts of compiler optimizations on code reuse gadget sets. *Proceedings of the ACM on Programming Languages, 5*(OOPSLA), Article 154, 1–30. https://doi.org/10.1145/3485531

Cai, Q., Gao, L., & Xue, J. (2004). Region-based partial dead code elimination on predicated code. In *Compiler Construction* (CC 2004) (Lecture Notes in Computer Science). Springer. https://doi.org/10.1007/978-3-540-24723-4_11

Chen, Z., Li, Y., Chen, B., Ma, W., Chen, L., & Xu, B. (2020). An empirical study on dynamic typing related practices in Python systems. In Proceedings of the 28th International Conference on Program Comprehension (pp. 83–93). Association for Computing Machinery. https://doi.org/10.1145/3387904.3389253

Ebad, S. A., & Darem, A. A. (2024). Exploring Android obfuscators and deobfuscators: An empirical investigation. *Electronics, 13*(12), Article 2272. https://doi.org/10.3390/electronics13122272

Liu, J., Fang, J., Wang, T., Xie, J., Huang, C., & Wang, Z. (2024). Efficient compiler optimization by modeling passes dependence: J. Liu et al. *CCF Transactions on High Performance Computing*, *6*(6), 588-607. https://doi.org/10.1007/s42514-024-00197-9

Mahlke, S. A., & Hank, R. E. (1995). A comparison of full and partial predicated execution support for ILP processors. In Proceedings of the International Symposium on Computer Architecture (pp. 138–150). Association for Computing Machinery. https://doi.org/10.1145/225830.225965

Maksoud, A., Hussien, A., Mushtaha, E., & Alawneh, S. I. A.-R. (2023). Computational design and virtual reality tools as an effective approach for designing optimization, enhancement, and validation of Islamic parametric elevation. *Buildings*, *13*(5), Article 1204. https://doi.org/10.3390/buildings13051204

Monniaux, D., & Six, J.-M. (2022). Formally verified loop-invariant code motion and assorted optimizations. In Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (pp. 1–27). Association for Computing Machinery. https://doi.org/10.1145/3529507

Petke, J., Alexander, B., Barr, E. T., Brownlee, A. E. I., Wagner, M., & White, D. R. (2023). Program transformation landscapes for automated program modification using Gin. *Empirical Software Engineering*, *28*(4), Article 104. https://doi.org/10.1007/s10664-023-10344-5

Romano, S., Vendome, C., Scanniello, G., & Poshyvanyk, D. (2018). A multi-study investigation into dead code. *IEEE Transactions on Software Engineering*, *46*(1), 71-99. https://doi.org/10.1109/TSE.2018.2842781

Ruan, L., Xu, Q., Zhu, S., Huang, X., & Lin, X. (2024). A survey of binary code similarity detection techniques. *Electronics*, *13*(9), Article 1715. https://doi.org/10.3390/electronics13091715

Saini, N., Singh, S., & Suman. (2018). Code clones: Detection and management. *Procedia Computer Science*, 132, 718–727. https://doi.org/10.1016/j.procs.2018.05.080

Spirin, E., Bogomolov, E., Kovalenko, V., & Bryksin, T. (2021). Psiminer: A tool for mining rich abstract syntax trees from code. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)* (pp. 13-17). IEEE. https://doi.org/10.48550/arXiv.2103.12778

Tao, N., Ventresque, A., Nallur, V., & Saber, T. (2024). Enhancing program synthesis with large language models using many-objective grammar-guided genetic programming. *Algorithms, 17*(7), Article 287. https://doi.org/10.3390/a17070287

Xia, X., Yan, Y., He, X., Wu, D., Xu, L., & Xu, B. (2022). An empirical study on the impact of python dynamic typing on the project maintenance. *International Journal of Software Engineering and Knowledge Engineering*, *32*(05), 745-768. https://doi.org/10.1142/S0218194022500243

Yang, H., Li, Z., & Guo, X. (2023). A novel source code clone detection method based on Dual-GCN and IVHFS. Electronics, 12(6), Article 1315. https://doi.org/10.3390/electronics12061315

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., & Liu, X. (2019, May). A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (pp. 783-794). IEEE. https://doi.org/10.1109/ICSE.2019.00086

Tulshihar Patil
Department of Information Technology Engineering,
Bharati Vidyapeeth (Deemed to be University),
College of Engineering,
Pune-411043,
India,
tbpatil@bvucoep.edu.in
ORCID 0000-0001-6985-6892

Shashank Joshi,
Department of Computer Engineering,
Bharati Vidyapeeth (Deemed to be University),
College of Engineering,
Pune-411043,
India,
shashank.joshi@bharatividyapeeth.edu
ORCID 0000-0001-8241-5530

Nazim Shaikh
Department of Basic Science and Humanities,
Bharati Vidyapeeth (Deemed to be University)
College of Engineering,
Pune-411043,
India,
nmshaikh@bvucoep.edu.in
ORCID 0000-0002-8318-0881

Rahul S. Desai,
Department of Engineering Science and Allied Engineering,
Bharati Vidyapeeth's College of Engineering for Women,
Pune-411043,
India,
rahul.desai@bharatividyapeeth.edu
ORCID 0000-0002-8048-8813

Sudarshan Talegaonkar
Department of Civil Engineering,
Bharati Vidyapeeth (Deemed to be University)
College of Engineering,
Pune-411043,
India,
sdtalegaonkar@bvucoep.edu.in
ORCID 0000-0003-2025-4393

Vipin Tiwari
Department of Computer Science and Engineering, Symbiosis Institute of Technology,
Symbiosis International (Deemed University)
Pune-412115,
vipintiwari1@gmail.com
ORCID 0000-0002-1082-0331