Enhancing Code Insight through Semantic Change Impact Evaluation

Chellu Vyshnavi, Chukka Mishal Raj, Gokul Gopakumar, Guddeti Bindu Prasanna, Meena Belwal

Department of Computer Science and Engineering,

Amrita School of Computing, Bengaluru

Amrita Vishwa Vidyapeetham, India

[vyshnavichellu@gmail.com](mailto:vyshnavichellu@gmail.com) , [mishalchukka@gmail.com](mailto:mishalchukka@gmail.com) , [gokulgopakumar2002@gmail.com](mailto:gokulgopakumar2002@gmail.com), [binduprasanna3005@gmail.com](mailto:binduprasanna3005@gmail.com), [b\_meena@blr.amrita.edu](mailto:b_meena@blr.amrita.edu)

*Abstract*— In software development, system integrity is a measure of the impact code changes have on them. It is determined by the team’s comprehension. However, rapid evolution of change commits and interaction in complex code bases create problems. By evaluating the effect of reflection changes, this study contributes to the improvement of the comprehension of code, a critical success factor for quality and efficiency. This addresses issues by using semantic analysis, change detection, refactoring detection, and semantic change detection and diffing. Our approach documents the semantic differences between the original and changed source code and gives pragmatic conclusions on how to enhance code quality and maintainability. Specifically, we divide the code into AST files using simple python scripts for semantic analysis and compare them for differences to find implications. It executes the method through the focus on development workflows and major projects as it includes the past information while still simplifying argument parsing and file processing. Based on our study findings, a better understanding of code can help developers to improve their decision-making capacity and protect the performance during the enhancement activity of the program.

Keywords— software development, code changes, team comprehension, complex codebases, semantic changes, code comprehension, semantic analysis, change detection, refactoring detection, code diffing, Abstract Syntax Trees (ASTs), Python scripts, development workflows, argument parsing, file handling, code quality, maintainability, system performance.

# Introduction

The software development field can be considered unique because the entire environment is in constant flux due to innovation and change. Thus, maintaining the integrity and performance of codebases is vital to the field. It is possible to detect future potential issues within these codebases to identify and manage risks in advance. At the same time, this concept suffers from various problems, the most important of which are associated with the complexity and dynamic properties characteristic of modern software systems. In this way, one of the most significant obstacles can be considered the difficulty of detecting semantic changes within the codebases. As a structural, functional, or logical change may significantly affect the performance and reliability of the system. Such early detections of such modifications are challenging because codebases are too complex to understand, containing hundreds of thousands or even millions of lines of code distributed among numerous modules and files. Furthermore, the rapid pace of development and frequent code alterations exacerbate the problem, making it difficult for engineers to keep track of subtle but critical semantic changes.

The work presented by Jingyu Liu et al. [13] addresses this by proposing PM2-CNN, a model that leverages source code and external knowledge for improved defect prediction. Such approaches have an advantage since they reduce the limitation of using PM2 models by proposing PM2-CNN. The model is based on these two components: the source code and the whole knowledge of the system, they increase the defect prediction accuracy. Nevertheless, they argue that much information from unstructured natural language can be fed into future defect prediction models that can help identify possible errors in the manufacturing process. This paper employs semantic code enrichment in code search by using DL-based models, which feature the attention mechanism and CNNs, according to Zhongyang Deng et al. [2]. The overall assessment of SemEnr is done on two Java datasets and this comprises over 2.2 million code-description pairs. The study's weaknesses are multi-layered, vary from learning semantic relations from code to higher level user's intention queries though the experiment gave positive results over previous baseline models. Z. Yuan and team [3] employ machine learning techniques to enable one variously to develop a real time predictor of faults in software development. It describes three main components: changes the scope of the source code, we augment our neural networks in real time, and quickly adapt when the model is updated. It is appraised on the Eclipse open-source endeavor and employed to a case actual software development pattern, revealing its use in a practical situation. Limitations may appear either as challenges in correctly representing fast-changing codes and changes in complex ones or the chance for performance degradation due to real-time prediction engines. In this paper [4]. Frick is intended to improve precision of the source code where both textual and AST represents are considered. It probably uses machine learning algorithms and maybe takes a dependency analysis approach also the data set can range from actual code repositories, highlighting the code folder and their respective alterations among versions. Pitfalls can be the increasing difficulty of complex software systems analysis and exact reproduction of the large libraries, as well as possible issues with precisely defining dynamic software dependencies.

To solve these issues, we use advanced approaches like semantic analysis, change detection, and refactoring detection. Semantic analysis is the practice of decomposing code into abstract syntax trees (ASTs) in order to understand and investigate its underlying structure and meaning. Change detection techniques such as sequence matching and code diffing allow us to discriminate between code versions and identify semantic differences. Furthermore, refactoring detection enables us to discover instances in which code has been re-organized or optimized without changing its exterior behavior. By integrating these techniques, we want to give developers the tools and insights they need to identify and correct semantic changes early in the development process, resulting in greater code quality and system maintainability.

The presented code provides functionalities like semantic analysis, detection of changes, refactoring detection, code differ, natural language processing on commit messages, provisioning of collaborative feedback, and generation of documentation.

About the major contributions, it avails complete code difference analysis, where developers can find semantic differences, recognize refactoring, and compute the difference in code between the original and modified versions. With natural language processing added to commit message analysis, finding relevant keywords like 'refactor' would throw light on the very purpose of changes made in the code. Besides, the mechanism of collaborative feedback enhances teamwork in that developers may comment on differences in code, therefore bringing better communication and collaboration. Further, the generation of documentation makes the process of documenting tools, their usage, and concepts easier. All these contributions, taken together, allow developers to have tools and insights into how to manage and collaborate on codebases in the most effective and efficient way, improving good software development practice.

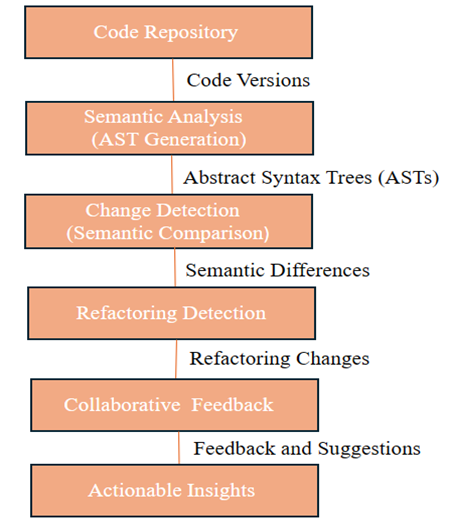


Fig 1. System Diagram

This system diagram depicts our software development tool's workflow, which begins with a code repository and progresses to semantic analysis, change detection, refactoring detection, and collaboration capabilities. It demonstrates how to effortlessly integrate analysis, detection, and collaboration capabilities into the software development workflow.

# Literature Survey

Ali et al. [1] introduces semantic change effect analysis to ensure that developers understand code changes during code reviews. While the SEMCIA tool improves accuracy and speed by giving semantic change relations for JavaScript contributions, it has certain limitations, such as difficulties with JavaScript 1.6 syntax handling, unsoundness in some cases, and potential accuracy variances among projects or languages. Despite these limitations, the technique shows promise in helping engineers understand code changes. Mithun et al. [2] describes the use of static programme slicing is a technique for determining the impact of changes in industrial software systems. The authors offer Imp, a tool that addresses performance and accuracy difficulties using static programme slicing. Imp was tested on a large ABB codebase and demonstrated encouraging results. The tool is compatible with development IDEs and nightly build environments. Despite the advantages of static slicing, analyzing big systems presents difficulties due to performance constraints and accuracy issues with core data structures. Imp seeks to automate the impact analysis process and increase efficiency in identifying the impacts of software modifications. Veit et al. [3] proposed a "Understanding Software Changes" addresses the difficulty of developers understanding and controlling developing software systems. Its goal is increase the accuracy and classification of source code. modifications, analyze fine-grained dependency changes, and dynamically assess the impact of changes on performance indicators. The study includes detailed change overviews to aid with the transition between coarse-grained solutions and raw code alterations. Key technologies include AST-based differencing algorithms such as MTDIFF and GumTree, as well as higher-level change overview tools such as Arena and ChangeScribe. The report offers a research plan for providing developers with precise, contextual information about code changes to help them better comprehend software evolution. Ling et al. [4] introduces SemEnr, a paradigm for improving code search efficiency. Using Java datasets from GitHub repositories, SemEnr beats baseline DeepCS, UNIF, CAT, and TabCS are examples of models that use Mean Reciprocal Rank (MRR) and SuccessRate@k. It uses a multi-perspective co-attention technique based on CNN embedding technology, which improves code search accuracy. Adding a code enrichment module considerably increases baseline model performance. SemEnr outperforms various models for training time efficiency. The MRR and SuccessRate@k metrics are utilized to evaluate model performance in the study, which uses Python 3.6, the Keras framework, and Adam optimization. The findings demonstrate SemEnr's superiority in code search tasks, emphasizing its efficacy, feature importance, and efficiency when compared to competing models. Michael et al. [5] introduced a CRaDLe, a deep learning model for code retrieval that includes semantic dependency learning. CRaDLe surpasses prior models at finding relevant code snippets by taking dependence and semantic information at the statement level into account. The paper uses neural networks for encoding and shows considerable performance gains in metrics such as R@1, R@5, R@10, and MRR through large-scale trials on public datasets. Scalability concerns, biases in training data, and the need to evaluate across several programming languages and repositories are also potential limits. Kak et al. [6] proposed a SCOR, which blends word embeddings with Markov Random Fields to improve source code retrieval accuracy, particularly in Java repositories such as AspectJ. SCOR outperforms classic Bag-of-Words models with considerable improvements in Mean Average Precision values. The evaluation was conducted using the Gensim library and precision metrics. However, drawbacks include an emphasis on Java repositories and worries about scalability with larger codebases. SCOR shows promise in enhancing the precision of source code retrieval using semantic embeddings and ordering constraints. Bird et al. [7] proposed a "Expectations, Outcomes, and Challenges of Modern Code Review" investigates motives, outcomes, and challenges associated with code review methods. Through interviews, observations, and analysis of 1,047 coherent units, it exposes code review's diverse role beyond defect identification, such as code style improvement and learning enhancement. Familiarity with altered code has a favorable impact on feedback value. The study emphasizes the significance of knowing change reasoning and proposes enhancements to code review tools to increase programme comprehension. It also emphasizes the ability of academics to address code review difficulties and influence real-world settings. Barnett et al. [8] investigate the issues of code review at Microsoft, focusing on large and complicated modifications. They introduce CLUSTERCHANGES, a technology that helps in understanding and partitioning code changes effectively. They illustrate CLUSTERCHANGES' success in partitioning code changes via changeset analysis and user research, with developers giving good comments on its capacity to improve the code review process. Martin et al. [9] provides a novel method for fine-grained and accurate source code differencing that uses a two-phase approach with top-down and bottom-up traversals of abstract syntax trees (ASTs). The runtime efficiency of a modular tool supporting multiple programming languages and different methods was assessed on real-world projects (Jenkins and jQuery). While parsing files is much slower than text diff, the GumTree technique is quite accurate, with raters agreeing that it is effective in explaining changes in 95.1% of situations. GumTree provides superior edit scripts when compared to similar work, with a focus on code differencing jobs. Ali et al. [10] proposed a BugAID is a data mining tool for identifying common bug patterns in JavaScript applications. BugAID analyses the commit histories of 134 projects to extract "Basic Change Types" (BCTs) and clusters commits to locate similar problem fixes. Manual inspection identifies 13 common bug patterns, which are assisted by tools like JGit, Mozilla Rhino, and GumTree. The results include 1,031 BCTs, 219 change types, and a considerable decrease in manual inspection time. The study emphasizes the usefulness of mining commit histories for detecting issue trends in dynamically typed languages such as JavaScript. David et al. [11] introduces LSdiff, a tool for automatically detecting systematic structural differences between programme versions. LSdiff detects anomalies in programmes without explicit rule specification by employing logic-based representation and rule inference. A focus group study of expert software developers verifies LSdiff's usefulness as a complement to existing differencing tools such as diff. Quantitative comparisons show that LSdiff can produce more concise results and discover extra structural facts than previous techniques. Overall, LSdiff is an excellent and complete technique for programmers to understand code changes. Isabella et al. [12] delves into Abstract Interpretation theory, with a focus on abstract domains, completeness, language semantics, and abstract dependencies. It describes the IMP language and its phrases, states, and semantics. Narrow abstract dependencies are specified and demonstrated, especially in programme slicing. EDEP and other algorithms compute narrow dependencies, which are critical for accurately approximating physical domains. Abstract dependencies are used in secure information flow, specifically in modelling non-interference for confidentiality. The paper focuses on their function in certifying abstract non-interference and linking computer science domains. Tools like EDEP and ATOMIZE are used for dependency computation and domain simplification, highlighting the significance of abstract dependencies in programme analysis and security.

Existing software defect prediction methods rely on handcrafted features and ignore comments/messages. This paper presented by Jingyu Liu [13] addresses this by proposing PM2-CNN, a model that uses source code and external knowledge to enhance defect prediction. The authors evaluate their method on a large dataset but acknowledge that future work could explore incorporating other unstructured natural language data. Z. Yuan et al. [14] utilizes machine learning models for real-time defect prediction in software development. It describes three main components: incremental feature collection and transformation, real-time detection of source code errors, and dynamic model changes. The tool is tested using the Eclipse open-source project and used in a real-world software development situation, showcasing its practical utility. Potential constraints can come from challenges in effectively capturing dynamic changes in intricate codebases and the increased performance load that can stem from live prediction mechanisms. Yoshiki and his research group [15] explain a technique known as TC2P for detecting change patterns in programming code. TC2P uses a tree-based approach to identify patterns in code changes by analyzing node positions to detect structure changes and errors. The research compares TC2P with NGR, highlighting that TC2P excels at identifying change patterns that NGR overlooks by concentrating on node positions and movements. Results from TC2P show better accuracy in detecting quality change patterns by capturing both structural and code changes more effectively than NGR. The efficiency of the technique and the time needed to detect pattern changes in software projects are both influenced by the minimum support value. Jian and his team introduced ASTNN, a novel neural network that employs abstract syntax trees (AST) for encoding source code. Unlike traditional methods, ASTNN breaks down large ASTs into smaller statement trees to capture syntax and vocabulary details within statements. A bidirectional RNN model is used to accurately encode statement vectors and capture their natural characteristics. ASTNN has shown its effectiveness in representing code snippets by surpassing other methods in tasks such as source code categorization and identifying duplicates. Both the code and experimental data are accessible for further inspection. Baojiang and colleagues [17] developed CCS to detect plagiarism in source code files by utilizing Abstract Syntax Trees (ASTs) effectively. CCS enables precise plagiarism detection by allowing the comparison of individual nodes through the calculation of hash values of ASTs. It minimizes incorrect results when calculating hashes and prepares a large source code database in advance to speed up detection. Due to its enhanced AST storage format, CCS is better at identifying various forms of code plagiarism and aiding in copyright protection measures. BASTS, developed by Chen et al. [18], is a distinctive method that enhances code summarization by segmenting code into blocks and encompassing hierarchical syntax structures within an AST. BASTS enhances Transformer-based summarization models and generates top-notch code summaries by incorporating a pre-training technique for detecting local nonlinear syntax encoding. BASTS outperforms previous methods in experiments, suggesting its potential to advance automatic code summarization techniques. Egor et al. introduce PSIMiner, a tool that enhances AST-based code representation by incorporating data from the IntelliJ Platform. It boosts code analysis abilities by leveraging PSI trees, leading to improved method name prediction with models like code2seq. Due to the flexibility of PSIMiner, various kinds of data can be effortlessly extracted from IDEs, resulting in enhancements in code representation for machine learning purposes. Roger et al.'s [20] efficient Prolog-based method enables precise matching of patterns in C/C++ programs by utilizing abstract syntax trees (ASTs). The method allows for efficient traversal and analysis of program structures, despite some challenges with embedded searches and arithmetic operations. Moreover, it demonstrates adaptability for compatibility with various parsers regardless of language, offering a hopeful direction for specialized syntactic examination. Meena and colleagues [21] provide a thorough overview of code obfuscation methods across three levels of transformation: low, medium, and high. It examines various perspectives, such as transformation level, obfuscation objectives, vulnerability to reverse engineering, and impact on performance. poll is a survey or questionnaire. The survey serves as a valuable instrument for pinpointing tactics of confusion for particular uses. It underscores the importance of conducting more research to create solutions that are both resilient and efficient, striking a balance between security and performance. Belwal and colleagues [22] presented a method for parsing and displaying mathematical expressions through Abstract Syntax Trees (AST). It accurately handles complex equations by utilizing lexical analysis, generating ASTs, and employing recursive descent parsing. The results indicate that interactive graph visualization supports effective assessment and visualization. Although not explicitly mentioned, scalability and error handling could pose challenges. In general, the method shows how ASTs and parsing methods can be valuable for processing mathematical expressions. Uma et al. [23] explains a software tool used to find, recognize, and study code clones using different approaches such as metrics, textual, token-based, syntax, and PDG-based comparisons. Although it lacks exact performance statistics, it emphasizes easy-to-use features and visual presentation. Potential limits include complexity management, accuracy based on input quality, scalability concerns, and interface refinement requirements. Overall, the programme provides a comprehensive solution for code clone detection, although it requires additional research and improvement to achieve peak performance. Mahmood et al. [24] proposed a machine learning-based bug prediction methods that can help minimize software development costs. It predicts bugs early and late in the development cycle by merging two datasets: one for code modifications and bug history, and another for CK metrics. Microsoft Azure offers machine learning models for bug prediction as a service. The evaluation measures are TP, TN, FP, and FN, with limits due to regression-based prediction accuracy. Overall, the study emphasizes the importance of early issue detection in improving software quality and reducing costs.

Even though various evaluations are taking place we thought to work on the domain of maintaining system integrity in the ever-changing context of software development is difficult due to codebase complexity. Our research aims to improve code comprehension by analyzing semantic changes, which are critical for maintaining code quality and efficiency. However, precisely recognizing and managing semantic changes is extremely difficult. We want to address this issue using a variety of techniques, including semantic analysis, change detection, refactoring detection, and code diffing. In addition to making developers understand what exactly changed semantically from the original code to the updated code, our approach is intentionally designed to guide a proper decision. Moreover, we enhance the engagement of developers through NLP and collaborative feedback systems, to make actionable insights that bring about improved code quality and maintenance.

# Proposed methodology

As such, this technique proposed shall be targeted at making the software development processes easier by mergers of various methodologies and technologies with the aim of raising the level of code comprehension, collaboration, and documentation. Now, let's take a closer look at each of the phases:

* Semantic Analysis: The ast performs in-depth semantic analysis of the code so that anomalies that may arise from the use of undefined variables or syntax errors can be shown. This kind of approach will ensure the sanctity and validity of the code base.
* Change Detection: The ast module compares the Abstract Syntax Trees of the original and modified code for detecting semantic differences. This feature will easily enable developers to find out the places where the changes occurred and what their effects were.
* Refactor Detection: This change into variable names or structure is typical of refactor operations. In this way, it can be very beneficial to be able to keep track of these kinds of changes to the code, which is tokenized by the module called tokenize. That helps for the sake of consistency and readability of the code.
* Code Diffing: The semantic differences between the original and updated code are calculated by the software difflib and a structured diff is returned to developers. This is useful to make obvious the changes carried out in a better code review.
* Natural Language Processing: The application of NLP methods to the study of commit messages in order to identify key words related to refactoring or other important changes, adds context to code modifications and gives a better understanding of the developer's intent.
* Collaborative Tools: The collaborative technologies allow engineers to make and review comments regarding the change in the code. This will increase communication and collaboration between the different development teams.
* Documentation: It automatically generates the documentation that explains its usage, principles, and intended changes. It is a lighthouse to a developer in getting deeper insight and an increased inclination towards future growth.

The suggested methodology includes these processes in the software development lifecycle to enhance the quality of the code, cooperation, and efficiency of software development, which would build strong and maintainable software systems.

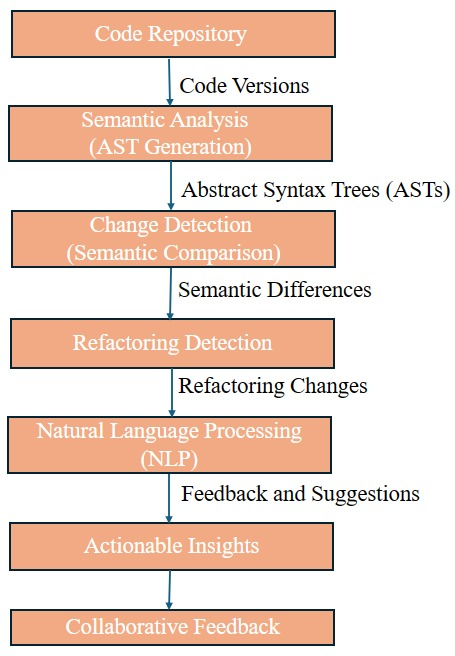


Fig 2: Block Diagram

In the following fig 2., the proposed methodology here shows that the input, given in code snippets, are provided from the code repository. Semantic analysis, change detection, and refactoring detection algorithms subject the code to determine the changes. Natural Language Processing techniques are applied to commit messages for providing further information. Collaborative feedback technologies enable team members to communicate and interact while providing feedback and suggestions. Finally, the findings offer actionable insights that can help to improve the software development process.

# experiments and results

## Experimental Setup

The experimental setup for the given code defines paths to datasets for code files—original and refactored—loads them, and then proceeds with a plethora of functions that carry out semantics change analysis, refactoring detection, computation of code differences, commit message analysis, and the generation of collaborative feedback and documentation. Ast, which is used for semantic analysis; difflib, for computing the differences between sequences; tokenize, for the tokenization of code; and BytesIO from the io module, for reading code as bytes. This enables the analysis, comparison, and processing of code in a meaningful way to let developers cope with changes in the code and to collaborate in software projects. We compare the original and modified Python implementations of basic algorithms: area computation, linear search, bubble sort, binary search, factorial computation, Fibonacci sequence generation, and a check for prime numbers.

## Results

Test Case – 1:

Original code:

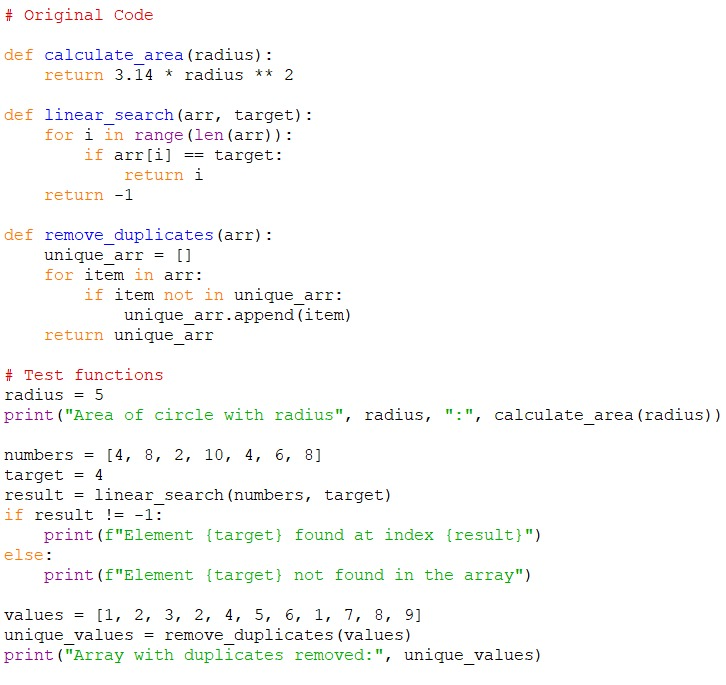
****

Fig 3: Original Code

In Fig 3, three functions are defined in this script: i.e., calculate\_area(radius), given its radius it calculates the area of the circle. Then comes linear\_search(arr, target), here linear search is used to find the index of a target element in an array. Finally remove\_duplicates(arr), this removes all the duplicates and forms a new array. These functions gets evaluated to compute the area of a circle with radius 5, that includes a linear search for element 4 in a list of numbers, and also deletes all the duplicates from another list of values.

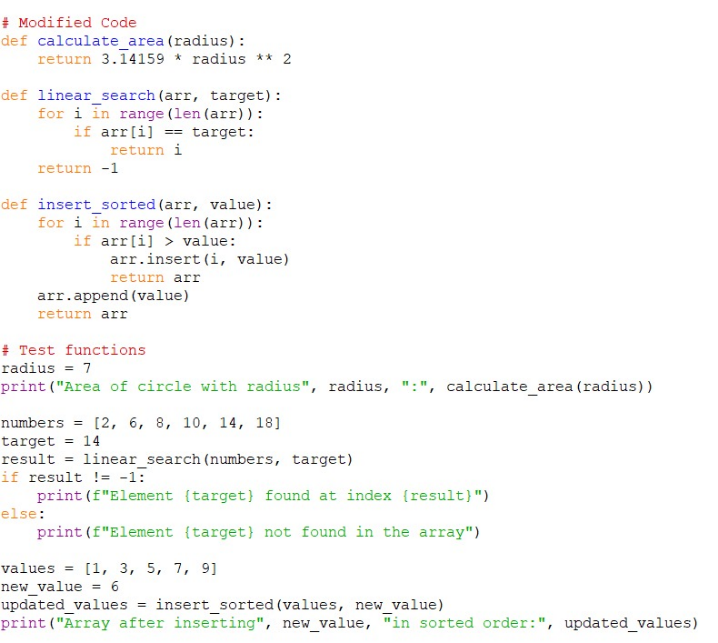
****

Fig 4: Modified Code

In Fig 4, along with defining the three functions here the calculation for the area of the circle is done with a more precise value of π. The function linear\_search(arr, target) locates the index of a target element in the array. The function insert\_sorted(arr, value), inserts a value into a sorted array while maintaining order. The script demonstrates by computing the area of a circle with radius 7 along with linear search for element 14 in a list of numbers, and then assigning value 6 to a sorted list. The output is displayed on the console.

Output:

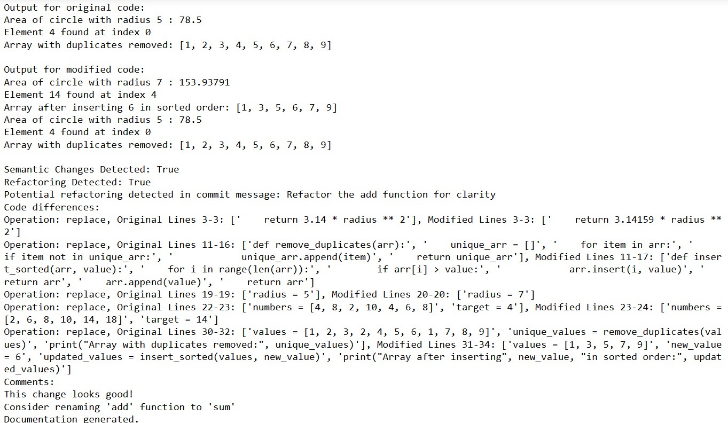
****

Fig 5: Comparing the original code1 and modified code1

The original code calculates the area of a circle with a radius of 5, tries to find element 4 in an indexed at 0 array, then removes the duplicates to get the unique collection. Refactored code puts value 6 in the already sorted array, calculates the area of a circle where the radius is 7, then finds element 14 at index 4 in another array. So, two semantic changes are about using more accurate value of π and insertion instead of a duplicate removal method in the sorted array. A refactoring, in other words, implies changes in functionality and updating of test cases to suit the new function. Line-level update/revision changes bring out the difference in the codes.

Test Case – 2:

**Original code:**

****

Fig 6: Original code

In this test case the script in Fig 6 defines three recursive functions: ‘is\_prime(n)’ for checking prime numbers; ‘factorial(n)’ to calculate the factorial of a number; and ‘fibonacci(n)’ to generate a Fibonacci sequence. This proves the capabilities of these functions by printing primes up to 15, thus calculating the factorial of 15, and showing the Fibonacci sequence upto the 15th integer that gets displayed to the console.

**Modified code:**

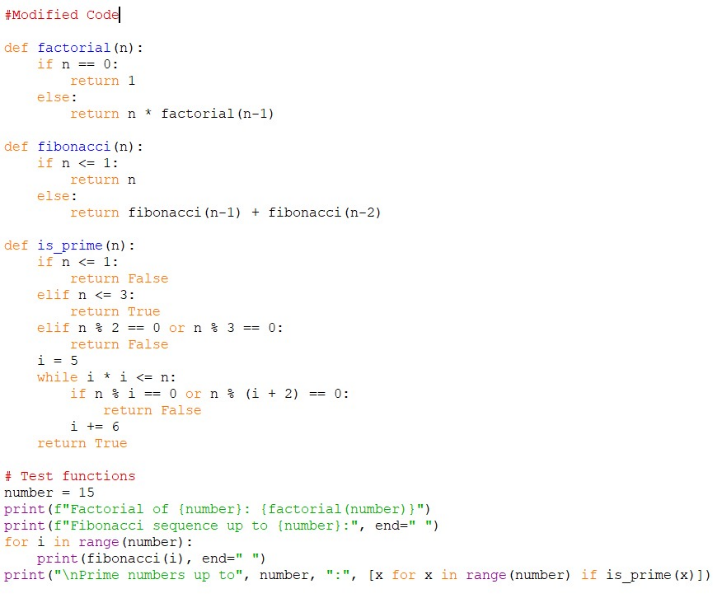
****

Fig 7: Modified Code

It defined three recursive functions: 'is\_prime(n)' to check for prime numbers; 'factorial(n)' to get the factorial of a number; and 'fibonacci(n)' to generate a Fibonacci sequence. It proves the functionality of these functions by printing primes up to 15, hence calculating the factorial of 15 and showing the Fibonacci sequence up to the 15th integer that gets displayed on the console.

**Output:**

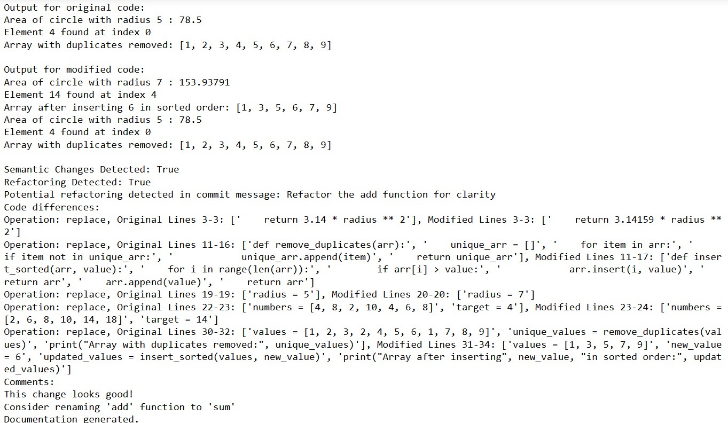
****

Fig 8: Comparing the original code1 and modified code1

The original program computed the factorial, Fibonacci sequence, and primes up to 10. This change in Fig 8 looks to extend all these computations up to 15, cleaning them in the process—more than likely for clarity and efficiency. It also renames functions, updates documentation for structure, and readability of code.

Test Case – 3:

**Original code:**

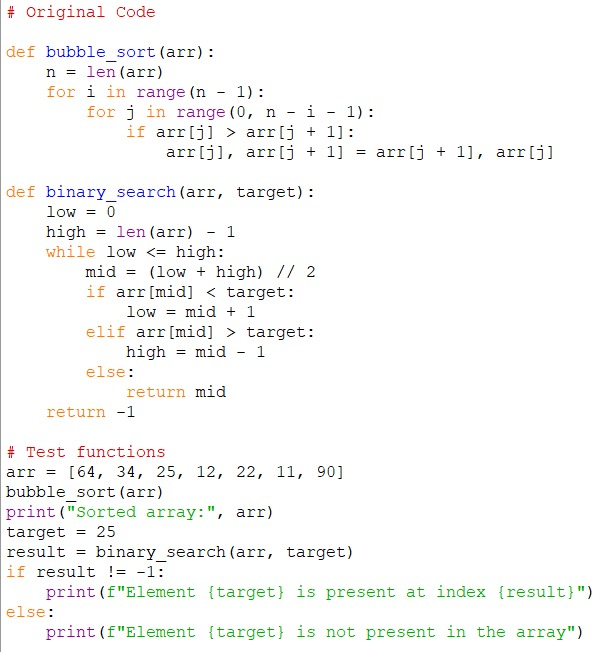
****

Fig 9: Original code

This code in Fig 9 sorts an array using Bubble Sort first, and then uses Binary Search to find a value inside the sorted array. The list is iteratively stepped through, nearby members are compared, and if they are out of order, they are switched in order to perform the bubble sort. Divide the list's potential locations in half repeatedly until there is only one left in order to conduct a binary search.

**Modified code:**

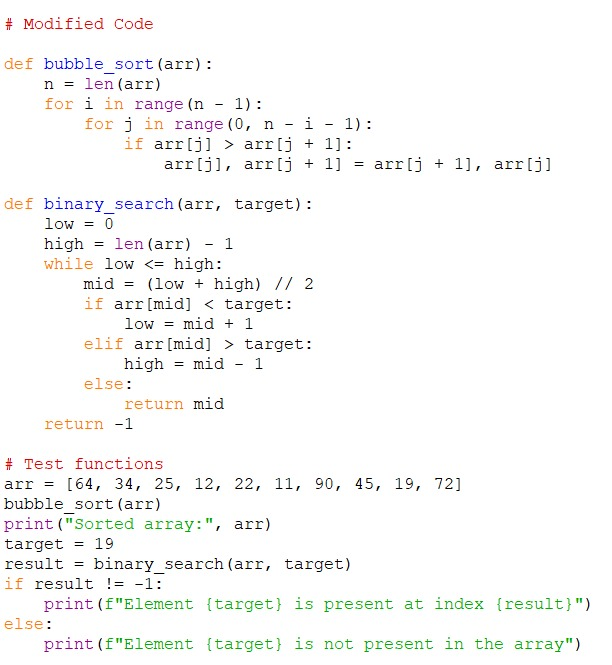
****

Fig 10: Modified Code

In Fig 10, bubble sort sorts an array ascendingly. It works by exchanging elements repeatedly to build up the invariant that at the start of the ith iteration, the largest i elements are sorted and are at the end of the array, for i=1 to N-1, where N is the length of the array. Binary search looks for a specific value in an ordered array. It searches by repeatedly halving the area under consideration until the target is reached or excluded. Then, it proceeds to show how the functions work by sorting a sample array and searching for a value of 19 in the sorted.

**Output:**

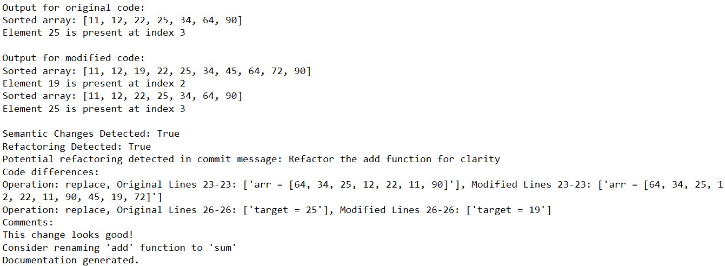
****

Fig 11: Comparing the original code3 and modified code3

It means that in this case Fig 11, the tool revealed refactoring on the function of sorting, which remains the same functionality. It also detects changing variable names. The reviewer suggested renaming the function "add" to "sum" for clarity.

Test Case – 4

**Original code:**

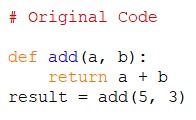


Fig 12: Original code

In Fig 12 the code here defines a function add in order to perform the sum of two variables assigned that is a and b and then return its sum. Then it shows the result by adding the initialized digits.

**Modified code:**

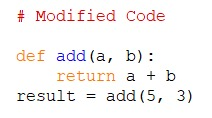
****

Fig 13: Modified Code

In Fig 13 the code defines a function in some programming language named add. This function takes two arguments denoted as a and b. It takes these two arguments to add them together. Return is the keyword used to define the output of a function that, in this case, is the combined value of a and b.

**Output:**

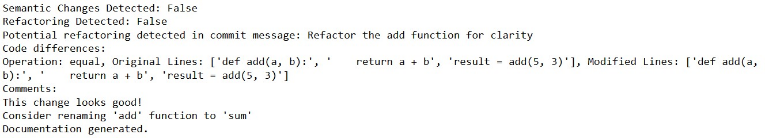
****

Fig 14: Comparing the original code4 and modified code4

The output here in Fig 14 shows that the semantic change checker of a code review tool didn't find anything substantive related to the code itself; it has identified some refactoring opportunities in the commit message. It suggests the refactoring of the add function for clarity and renaming it to sum.

# Conclusion and future scope

In conclusion, this paper offers , the suggested methodology takes a comprehensive approach to software development, including semantic analysis, change detection, refactoring detection, natural language processing, collaborative feedback, and documentation production. The goal is to improve code quality, maintainability, and developer collaboration by conducting systematic code analysis and providing actionable insights.

Moving forward, there are various options for improving the process. These include refining semantic analysis techniques, experimenting with advanced change detection algorithms, creating intelligent refactoring assistance tools, deepening integration with natural language processing, integrating with development environments, and supporting new programming languages. These actions will allow the methodology to adapt and better align with the changing requirements of current software development processes.

##### References

1. Q. Hanam, A. Mesbah and R. Holmes, "Aiding Code Change Understanding with Semantic Change Impact Analysis," 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), Cleveland, OH, USA, 2019, pp. 202-212, doi: 10.1109/ICSME.2019.00031.
2. M. Acharya and B. Robinson, "Practical change impact analysis based on static program slicing for industrial software systems," 2011 33rd International Conference on Software Engineering (ICSE), Honolulu, HI, USA, 2011, pp. 746-755, doi: 10.1145/1985793.1985898.
3. V. Frick, "Understanding Software Changes: Extracting, Classifying, and Presenting Fine-Grained Source Code Changes," 2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Seoul, Korea (South), 2020, pp. 226-229.
4. Deng, Z., Xu, L., Liu, C., Huangfu, L., & Yan, M. (2024). Code semantic enrichment for deep code search. Journal of Systems and Software, 207, 111856. https://doi.org/https://doi.org/10.1016/j.jss.2023.111856
5. Gu, W., Li, Z., Gao, C., Wang, C., Zhang, H., Xu, Z., & Lyu, M. R. (2021). CRaDLe: Deep code retrieval based on semantic Dependency Learning. Neural Networks, 141, 385–394. <https://doi.org/https://doi.org/10.1016/j.neunet.2021.04.019>
6. S. Akbar and A. Kak, "SCOR: Source Code Retrieval with Semantics and Order," 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), Montreal, QC, Canada, 2019, pp. 1-12, doi: 10.1109/MSR.2019.00012.
7. Bacchelli, Alberto. (2013). Expectations, Outcomes, and Challenges of Modern Code Review. Proceedings - International Conference on Software Engineering. 10.1109/ICSE.2013.6606617.
8. M. Barnett, C. Bird, J. Brunet and S. K. Lahiri, "Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets," 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, 2015, pp. 134-144, doi: 10.1109/ICSE.2015.35.
9. Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M., & Monperrus, M. (2014). Fine-grained and accurate source code differencing. Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, 313–324. https://doi.org/10.1145/2642937.2642982
10. Hanam, Q., Brito, F. S. de M., & Mesbah, A. (2016). Discovering bug patterns in JavaScript. Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 144–156. https://doi.org/10.1145/2950290.2950308
11. M. Kim and D. Notkin, "Discovering and representing systematic code changes," 2009 IEEE 31st International Conference on Software Engineering, Vancouver, BC, Canada, 2009, pp. 309-319, doi: 10.1109/ICSE.2009.5070531.
12. Mastroeni, I., & Zanardini, D. (2008). Data dependencies and program slicing: from syntax to abstract semantics. Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, 125–134. https://doi.org/10.1145/1328408.1328428
13. Liu, Jingyu & Ai, Jun & Lu, Minyan & Wang, Jie & Shi, Haoxiang. (2023). Semantic feature learning for software defect prediction from source code and external knowledge. Journal of Systems and Software. 204. 111753. 10.1016/j.jss.2023.111753.
14. Zi Yuan, C. Liu, L. Yu and L. Zhang, "ChangeChecker: A tool for defect prediction in source code changes based on incremental learning method," Proceedings of 2013 3rd International Conference on Computer Science and Network Technology, Dalian, China, 2013, pp. 349-354, doi: 10.1109/ICCSNT.2013.6967127.
15. Y. Higo, J. Matsumoto and S. Kusumoto, "Tree-based Mining of Fine-grained Code Changes to Detect Unknown Change Patterns," 2021 28th Asia-Pacific Software Engineering Conference (APSEC), Taipei, Taiwan, 2021, pp. 61-71, doi: 10.1109/APSEC53868.2021.00014.
16. J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang and X. Liu, "A Novel Neural Source Code Representation Based on Abstract Syntax Tree," 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 2019, pp. 783-794, doi: 10.1109/ICSE.2019.00086.
17. Baojiang Cui, Jiansong Li, Tao Guo, Jianxin Wang and Ding Ma, "Code Comparison System based on Abstract Syntax Tree," 2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT), Beijing, 2010, pp. 668-673, doi: 10.1109/ICBNMT.2010.5705174.
18. C. Lin, Z. Ouyang, J. Zhuang, J. Chen, H. Li and R. Wu, "Improving Code Summarization with Block-wise Abstract Syntax Tree Splitting," 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), Madrid, Spain, 2021, pp. 184-195, doi: 10.1109/ICPC52881.2021.00026.
19. E. Spirin, E. Bogomolov, V. Kovalenko and T. Bryksin, "PSIMiner: A Tool for Mining Rich Abstract Syntax Trees from Code," 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), Madrid, Spain, 2021, pp. 13-17, doi: 10.1109/MSR52588.2021.00014.
20. Crew, R.F., 1997, October. ASTLOG: A Language for Examining Abstract Syntax Trees. In DSL (Vol. 97, No. 18, p. 18).
21. G. Vishwas, N. S. Nithin, P. Varshith and M. Belwal, "Unveiling the World of Code Obfuscation: A Comprehensive Survey," 2023 7th International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS), Bangalore, India, 2023, pp. 1-8, doi: 10.1109/CSITSS60515.2023.10334127.
22. S. T. Pecheti, B. H. M, N. Kodurupaka and M. Belwal, "Recursive Descent Parser for Abstract Syntax Tree Visualization of Mathematical Expressions," 2023 7th International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS), Bangalore, India, 2023, pp. 1-6, doi: 10.1109/CSITSS60515.2023.10334103.
23. D. Tukaram and U. M. B, "Design and Development of Software Tool for Code Clone Search, Detection, and Analysis," 2019 3rd International conference on Electronics, Communication and Aerospace Technology (ICECA), Coimbatore, India, 2019, pp. 1002-1006, doi: 10.1109/ICECA.2019.8821928.
24. U. Subbiah, M. Ramachandran, and Z. Mahmood, “Software engineering approach to bug prediction models using machine learning as a service (MLaaS),” in ICSOFT 2018 - 13th International Conference on Software Technologies, Jan. 2019, pp. 879–887. [Online]. Available: https://eprints.leedsbeckett.ac.uk/id/eprint/61