Indian Journal of Computer Science and Technology
https://www.doi.org/10.59256/indjcst.20250402002
Volume 4, Issue2 (May-August 2025), PP: 21-31.
www.indjcst.com
ISSN No: 2583-5300

Automated Developer Pattern Analysis and Code Suggestions with AI

Manjula G[1], A Yashwanth[2], Nitish KP[3]
[1]*Professor& HOD, Computer Science and Design, Dayananda Sagar Academy of Technology & Management, Bengaluru, Karnataka, India.*
[2,3] *Students, Computer Science and Design, Dayananda Sagar Academy of Technology & Management, Bengaluru, Karnataka, India.*

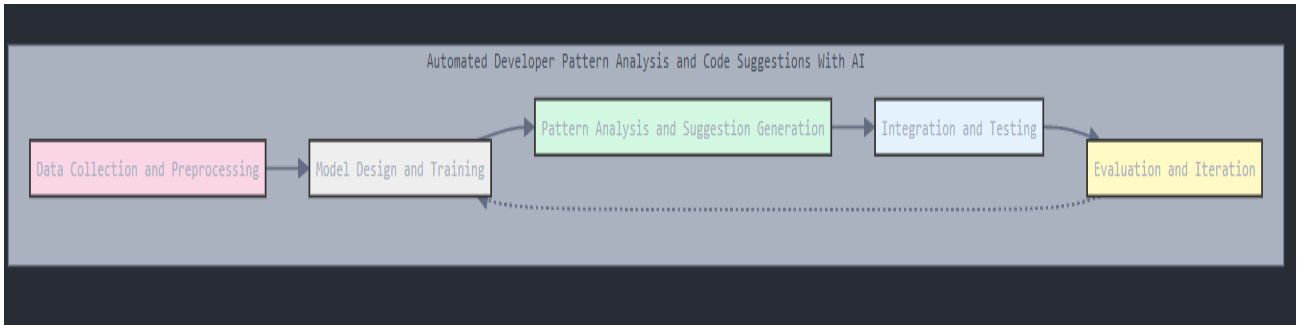
To Cite this Article: *Manjula G[1], A Yashwanth[2], Nitish KP[3], "Automated Developer Pattern Analysis and Code Suggestions with AI", Indian Journal of Computer Science and Technology, Volume 04, Issue 02 (May-August 2025), PP: 21-31.*

Abstract: *The rapid advancement of software development tools that elevates productivity, refines code quality, diminishes human error. This project confronts these challenges by integrating artificial intelligence (AI) to automate the analysis of the developer patterns and provide appropriate code suggestions. By harnessing machine learning algorithms, natural language processing and making the most of the Qwen 2.5 coder, we can assiduously survey individual coding habits, study recurring patterns, and detect inefficiencies in real-time. This AI-driven framework tailors itself to unique coding styles while mining insights from diverse repositories of open-source code to generate contextually optimized solutions. Key features comprise of semantic code analysis, predictive pattern recognition, and automated refactoring suggestions, all designed to streamline workflows and elevate consistent, high quality software enhancement. This work ensures for scalable, intelligent coding assistant that empowers developers across diverse programming environments.*

Keywords: *Semantic code analysis; Predictive pattern recognition; Automated refactoring suggestions; Anomaly Detection; Code efficiency; Intelligent coding assistant.*

I.INTRODUCTION

In today's fast-paced world of software development, developers are under constant pressure to deliver efficient, high-quality code while keeping up with tight deadlines and ever-changing requirements. It's no surprise that even the best of us can miss a trick or two—whether it's a sneaky bug, a clunky workaround, or just a better way to get things done. That's where this project steps in. "Automated Developer Pattern Analysis and Code Suggestions with AI" is all about harnessing artificial intelligence to give developers a helping hand. By studying how we code and tapping into the wealth of knowledge out there in the coding community, this system aims to spot patterns, catch inefficiencies, and offer smart, practical suggestions right when they're needed. It's not just about fixing mistakes—it's about making coding faster cleaner and more enjoyable. This introduction sets the stage for exploring how AI can become a trusted partner in the development process, paving the way for smarter tools that grow with us.



II.LITERATURE REVIEW

A. A Comparative Review of AI Techniques for Automated Code Generation in Software Development: Advancements, Challenges, and Future Directions

Explores traditional and AI-driven techniques for Automated Code Generation (ACG) [1]. Traditional methods include Rule-Based (RB) systems, which use predefined logic for code mapping but struggle with scalability, and Template-Based (TB) approaches that leverage reusable code fragments for efficiency but lack adaptability. Domain-Specific Languages (DSLs) bridge high-level abstractions to executable code but require manual effort to adapt to new contexts. AI techniques, such as Machine Learning (ML) and Deep Learning (DL), have revolutionized ACG by automating tasks like code auto completion (e.g., Code Whisperer) and generating code from natural language (e.g., GPT-3). Evolutionary Algorithms (EAs) optimize code through

iterative mutation, while Natural Language Processing (NLP) translates user descriptions into code. Applications span web development (HTML generation from mock-ups), mobile apps (sketch-tocode frameworks), and industrial automation (PLC code optimization). However, challenges like data scarcity, computational costs, contextual ambiguity, and ethical risks (e.g., bias, intellectual property) hinder broader adoption. Comparative analyses highlight trade-offs: RB systems offer transparency but rigidity, while DL models handle complexity at the cost of interpretability. NLP enhances accessibility for non-programmers, whereas EAs excel in performance-critical tasks. Hybrid approaches combining RB with DL aim to balance flexibility and control. Key challenges include scalability for large codebases, maintaining code quality, and addressing ethical concerns like job displacement. Future directions emphasize multi-modal generation (integrating text, sketches, and diagrams), human-in-the-loop systems for real-time developer feedback, and domain-specific adaptations for niche languages. The review underscores AI's transformative potential in ACG but calls for robust frameworks to address technical and ethical limitations, ensuring sustainable integration into software development practices.

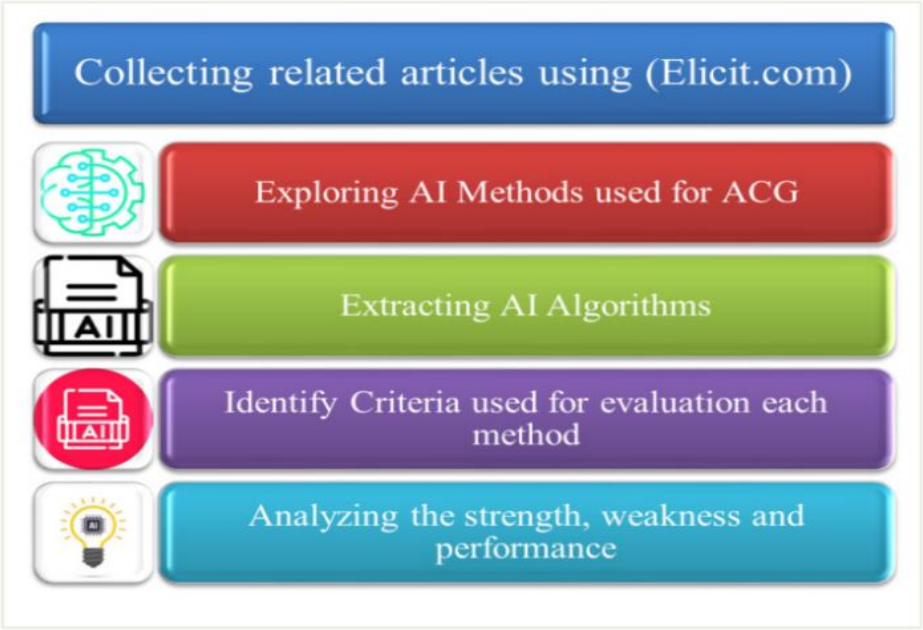


Fig 2.1: The general methodology steps

B. Automated refactoring to the Strategy design Pattern

This design Pattern introduces a method for automated refactoring to the Strategy design pattern to address complex conditional logic in object-oriented code [2]. The authors propose an algorithm that identifies conditional statements (e.g., if/else, switch) where branch selection is controlled by client classes rather than the context class, aligning with the Strategy pattern's intent to encapsulate interchangeable algorithms. The algorithm combines syntactic, control, and data flow analysis to detect nontrivial branch logic and variables (parameters/fields) whose values originate from external clients, excluding context controlled local variables. Implemented in the JDeodorant Eclipse plug-in, the method extends existing tools by enabling Total Replacement of Conditional Logic (TRCL)—replacing conditionals with polymorphic Strategy calls when client-provided values are constants. This approach enhances code maintainability and reduces complexity by distributing logic across Strategy classes. Evaluation on projects like Pamvotis and Apache Ant demonstrated 50–80% recall and 33–64% precision, with false positives often arisingfrom trivial branch logic. Refactoring reduced cyclomatic complexity by 15–30% and method size by 20–30%, improving code quality. Scalability tests showed efficient execution (under 2.5 minutes for large projects like Jade), validating its practicality. Future work includes refining identification rules with machine learning and expanding TRCL's applicability. This work bridges the gap between manual design pattern adoptions and automated refactoring, offering a scalable solution to simplify conditional complexity in software maintenance.

C. Automatic Mining of Code Fix Patterns from Code Repositories

It introduces a method to automatically mine code fix patterns from Git repositories to address limitations in traditional static analysis tools, which rely on predefined bug patterns. The authors propose a language-agnostic approach (supporting C, C++, C#) that extracts code modifications from commit histories using GumTreeDiff to generate abstract syntax trees (ASTs) for file revisions [3]. These ASTs are merged into super-trees tagged with operations (INSERT, REMOVE, NONE) and enriched with word embeddings (via FastText) to capture semantic similarities. Key innovations include vertexto-vertex matching of ASTs, dynamic programming-based tree mapping to unify edits into patterns, and the introduction of holes and omission flags to generalize patterns while preserving context. The workflow involves filtering commits, processing super-trees to remove redundancies, and clustering edits into actionable patterns. Evaluated on Tizen repositories, the method processed 18 million lines of code into 21,889 patterns, demonstrating scalability (8.5 hours for 50 projects) but revealing challenges in over-generalization and pattern quality assessment. The evaluation categorized patterns into intra-commital, intrarepository, and inter-repository types, with manual review showing 59.8% of intra-commital patterns were actionable (good/trivial), dropping to 18.6% for inter-repository patterns. While the approach successfully identified non-trivial fixes (e.g., replacing direct field access with getters),

the quality metric (based on embedding similarity and structural alignment) correlated poorly with human judgment, necessitating manual validation. The authors highlight limitations in filtering excessive generalizations and propose future enhancements, such as integrating Tree-sitter for AST parsing and refining pattern validity conditions. This work advances automated code repair by enabling rapid adaptation to domain-specific fixes but underscores the need for improved ranking heuristics to reduce reliance on manual curation.

D. Editable AI: Mixed Human-AI Authoring of Code Patterns

It situates itself within the growing body of research on AI-assisted code authoring, particularly focusing on enhancing autocomplete systems through explainability and user interactivity [4]. Traditional autocomplete tools, such as MAPO and GraPacc, leverage statistical models to predict code completions based on contextual patterns but often operate as "black boxes," offering limited insight into their recommendations. Prior work, including studies by Bruch et al. and Raychev et al., underscores the high predictability of code and the effectiveness of learning patterns from repositories, yet these systems struggle to align with developer intent when patterns are context-specific or inconsistently applied. The paper also draws on explainable AI principles, such as decision sets and interactive machine learning (e.g., Kulesza et al.'s work on explanatory debugging), which emphasize transparency and user control. IRIS advances these ideas by integrating decision trees—a human-interpretable model—to surface code patterns as editable rules, enabling developers to inspect, validate, and refine AI-generated suggestions. This bridges a critical gap between automated recommendations and developer agency, addressing trust issues noted in studies like Proksch et al.'s analysis of code recommender systems. The authors further build on interactive machine learning paradigms, where systems like Calcite and Jungloid Mining enable user feedback to refine outputs. IRIS extends this by allowing developers to prioritize, blacklist, or manually define patterns, fostering a collaborative human-AI workflow. The system's design reflects insights from CSS styling and semantic HTML structures, acknowledging that document consistency often hinges on implicit patterns beyond static style rules. Evaluation results demonstrate IRIS's efficacy: participants using the tool completed tasks 15–33% faster and achieved higher success rates, particularly in correcting inconsistencies, compared to a control group. These outcomes align with findings from Hindle et al.'s "naturalness of software" hypothesis, which posits that code repetitiveness aids pattern recognition, while also highlighting the value of explainability in reducing cognitive load. However, the paper notes limitations in scalability for larger documents and potential overfitting—challenges echoed in prior work on code completion systems. By merging interpretable AI with interactive editing, IRIS offers a novel framework for mixed-initiative authoring, advancing both autocomplete usability and the broader vision of editable AI in creative domains.

E. Empirical evaluation of automated code generation for mobile applications by AI tools

It situates itself within the growing body of research on AI-assisted code generation, particularly focusing on mobile app development using frameworks like Flutter [5]. Prior work, such as studies by Brown et al. (2020) on large language models (LLMs) like GPT-3, highlights the potential of generative AI to automate coding tasks, while Kazemindasar et al. (2023) demonstrate how novices leverage LLMs for programming education. The authors also draw on earlier efforts, such as Tifan et al. (2017), which explored compiling natural language into executable bytecode, and Bilgram and Laarmann's (2023) work on AIaugmented prototyping. However, existing research often emphasizes general-purpose code generation or domainspecific applications (e.g., video games, data analytics), leaving gaps in understanding AI's efficacy for mobile development workflows. The paper addresses this by evaluating ChatGPT 3.5's ability to iteratively generate Flutter code for a Battleship game, aligning with Proksch et al.'s (2016) call for empirical evaluations of code recommender systems. The study also echoes findings from Bruch et al. (2009) and Raychev et al. (2014), which stress the importance of human-AI collaboration to correct errors and optimize outputs, but extends these insights to mobile-specific challenges like deprecated widgets and state management with libraries like GetX. The authors contribute novel insights into the iterative prompting process required to refine AI-generated code, a methodology absent in prior tools like GitHub Copilot. Their results reveal that while ChatGPT 3.5 can produce functional code for simple tasks (e.g., grid rendering), it struggles with complex requirements like multi-screen navigation and null safety—a limitation consistent with Haider et al.'s (2019) observations about AI's difficulty in handling nuanced software specifications. The paper also highlights organizational shortcomings in AI-generated code, such as monolithic files and deprecated methods, issues less explored in earlier studies focused on syntactic correctness. These findings align with Hindle et al.'s (2016) "naturalness of software" hypothesis but underscore the gap between pattern recognition and architectural best practices. By demonstrating the necessity of human oversight for error correction and library integration, the study reinforces Mura et al.'s (2021) argument that AI tools complement rather than replace developer expertise. Future work, as suggested, could explore newer models like GPT-4 or Bard to address training data recency issues, advancing toward the vision of AI as a collaborative "pair programmer" in mobile development.

F. Graph Based Mining of Code Change Patterns from Version Control Commits

It situates itself within the evolving field of code change pattern mining, building on prior work in frequent itemset mining, AST-based diffing, and complex event processing (CEP). Traditional approaches, such as those by Negara et al. (2014) and Nguyen et al. (2019), focused on itemsets or dependency graphs to identify recurring edits but lacked relational context between code elements [6]. Tools like GumTree, used for AST-based differencing, provided foundational techniques for extracting fine-grained edits but did not capture inter-edit relationships. The authors advance these methods by proposing a graph-based approach that transforms code changes into relational graphs, preserving structural and semantic connections (e.g., parent-child AST nodes, attribute equality). This enables frequent subgraph mining to detect patterns that reflect contextual dependencies, such as method-constructor pairs or modifier consistency. Their method contrasts with itemset mining, which treats edits as isolated sets, and extends Nguyen et al.'s graph models by emphasizing cross-project generalizability and interpretability. The study also aligns with Foster et al.'s (2012) WitchDoctor and Kuschke et al.'s (2014) CEP-based recommenders, demonstrating how mined patterns can

be translated into actionable rules for auto-completion and anomaly detection. By evaluating seven Java projects, the authors address scalability, showing their method efficiently processes large repositories (e.g., Elasticsearch with 80k commits) and identifies patterns persistent across projects (e.g., "add parameter" or "delegate to super" patterns). The paper bridges gaps in explainability and usability of code change patterns. While prior work (e.g., Zimmermann et al., 2005) mined version histories for correlated changes, this approach captures richer relational data through graphs, enabling deeper insights into developer workflows. The evaluation reveals that graph-based patterns (e.g., method documentation updates, constructor-attribute linkages) are both interpretable and actionable, with 45% average coverage across commits. Limitations include Java-specific analysis and reliance on GumTree, which inherits AST-diffing inaccuracies. However, the integration with CEP rules offers a pragmatic path for tools to leverage patterns in real-time recommenders. Future work could expand to multi-language support, address oversized commits via AST-based splitting, and enhance automation in rule refinement. By combining relational graph mining with practical applications, this work advances the understanding of code evolution and sets a foundation for more context-aware developer tools.

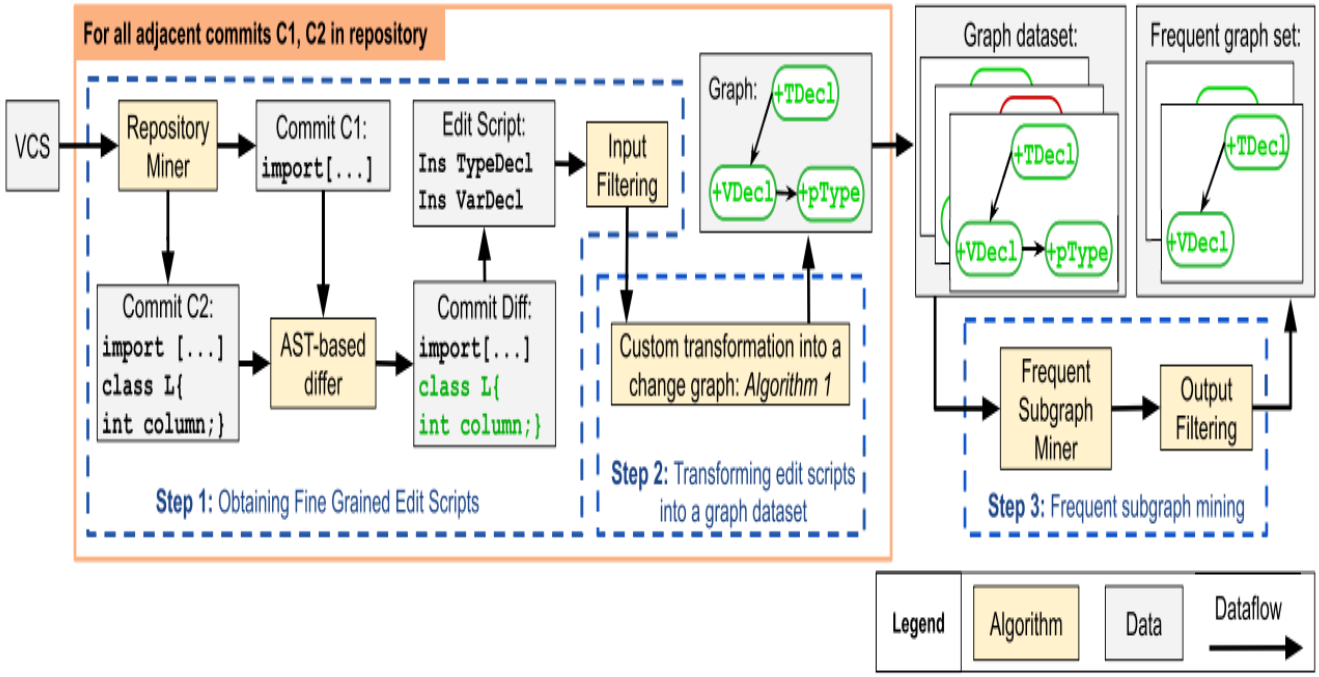


Fig 2.2: Pipeline architecture of the mining method extracting patterns from a VCS project.

G. Low code for smart software development

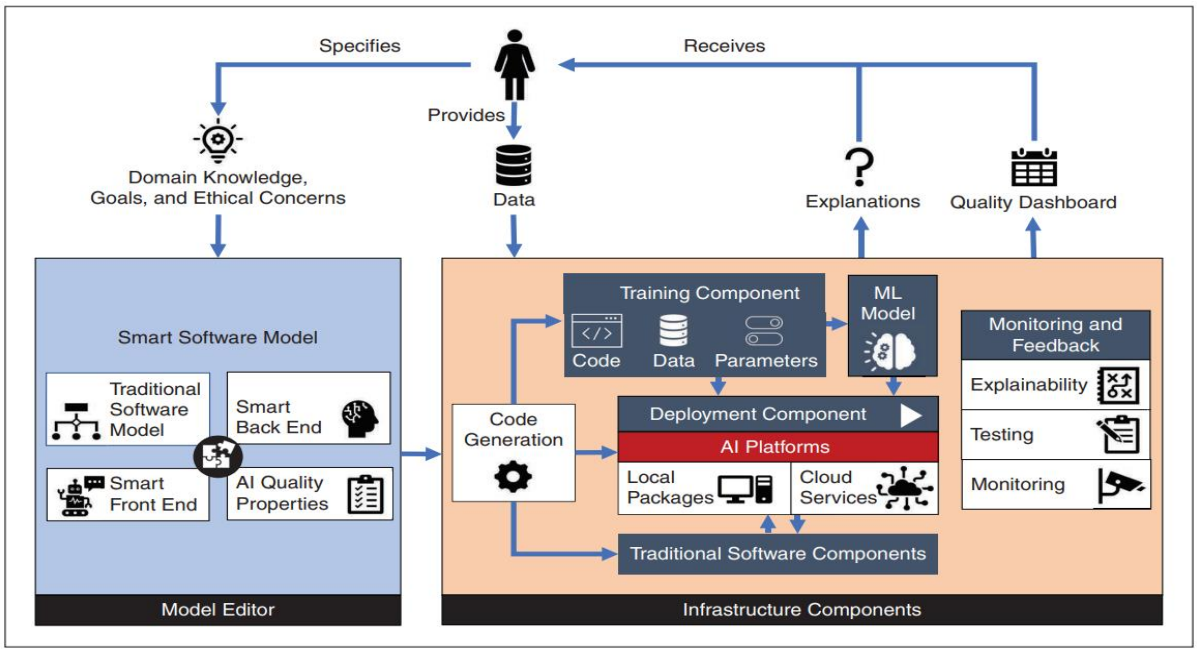


Fig 2.3: The low-code architecture

This highlights the growing importance of low-code platforms in simplifying the development of AI-based systems, addressing challenges like talent shortages and high costs [7]. It emphasizes the need for these platforms to integrate AI components seamlessly, ensuring traceability, explain ability, and collaboration between traditional and AI-driven software elements. The authors propose a wish list for ideal low-code tools, including features like platform-agnostic specifications, support for the entire AI lifecycle, and mechanisms to ensure ethical and quality concerns are addressed.

H. Reducing Human Effort and Improving Quality in Peer Code Reviews using Automatic Static Analysis and Reviewer Recommendation

This introduces "Review Bot," a tool designed to automate coding standard checks and defect detection using static analysis tools like Checkstyle, PMD, and FindBugs [8]. It highlights the benefits of integrating these tools into the code review process, such as reducing reviewer workload and enhancing review quality. Additionally, the paper addresses the challenge of assigning appropriate reviewers by proposing a recommendation algorithm based on line change history, which improves accuracy and efficiency in large projects. Experimental results demonstrate the tool's effectiveness, with high developer acceptance rates for automated comments and improved reviewer assignment accuracy.

I. Reliable Fix Patterns Inferred from Static Checkers for Automated Program Repair

Explores the use of fix patterns inferred from static analysis violations to enhance automated program repair (APR). It introduces AVATAR, a tool that leverages these patterns to address semantic bugs in software [9]. The study evaluates AVATAR's performance across multiple benchmarks, demonstrating its ability to fix bugs effectively and complement existing APR tools. The paper also investigates the impact of fault localization techniques and stack trace information on AVATAR's bug-fixing efficiency.

Table 1. Summary of Reviewed Studies in AI-Assisted Code Development

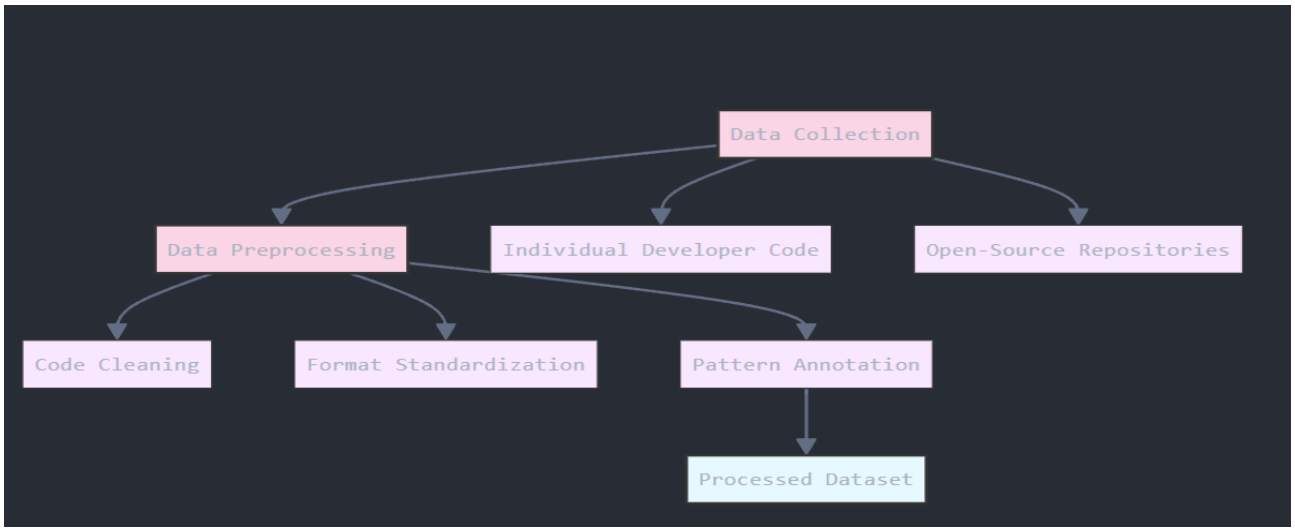
Domain	Study (Author, Year)	Approach/Mode	Key Contribution	Focus Area
Automated Code Generation	Odeh et al. (2024)	Comparative Review of AI Techniques	Explored traditional and AI-driven ACG methods, highlighting trade-offs between transparency and flexibility	Code Generation
Object-Oriented Design	Christopoulou et al. (2012)	Automated Strategy Design Pattern Refactoring	Algorithm to identify conditional statements controlled by client classes; implemented in JDeodorant Eclipse plugin	Design Pattern Optimization
Code Fix Mining	Koryabkin & Ignatyev (2022)	GumTreeDiff with Abstract Syntax Trees	Method to automatically mine code fix patterns from commit histories with semantic preservation	Code Repair
Code Pattern Authoring	Chugh et al. (2019)	IRIS: Editable AI with Decision Trees	Mixed human-AI authoring system allowing developers to inspect, validate, and manually define patterns	Human-AI Collaboration
Mobile Development	Mobile Development	Empirical ChatGPT Evaluation	Assessment of ChatGPT 3.5 for Flutter code generation, revealing strengths in simple tasks but limitations in complex requirements	Mobile App Development
Code Change Mining	Janke & Mäder (2022)	Graph-Based Mining from VCS	Relational graph approach capturing structural connections between code elements in version control commits	Pattern Recognition

Low-Code Development	Cabot & Clarisó (2023)	Platform-agnostic Framework	Proposal for ideal low-code tools integrating AI components with traceability and explainability	Development Platforms
Code Review	Balachandran (2013)	Review Bot with Static Analysis	Automated coding standard checks using tools like Checkstyle, PMD, and FindBugs	Code Quality
Automated Program Repair	Liu et al. (2023)	AVATAR: Fix Pattern Inference	Tool leveraging static analysis violations to enhance semantic bug fixing with proven fix patterns	Bug Remediation

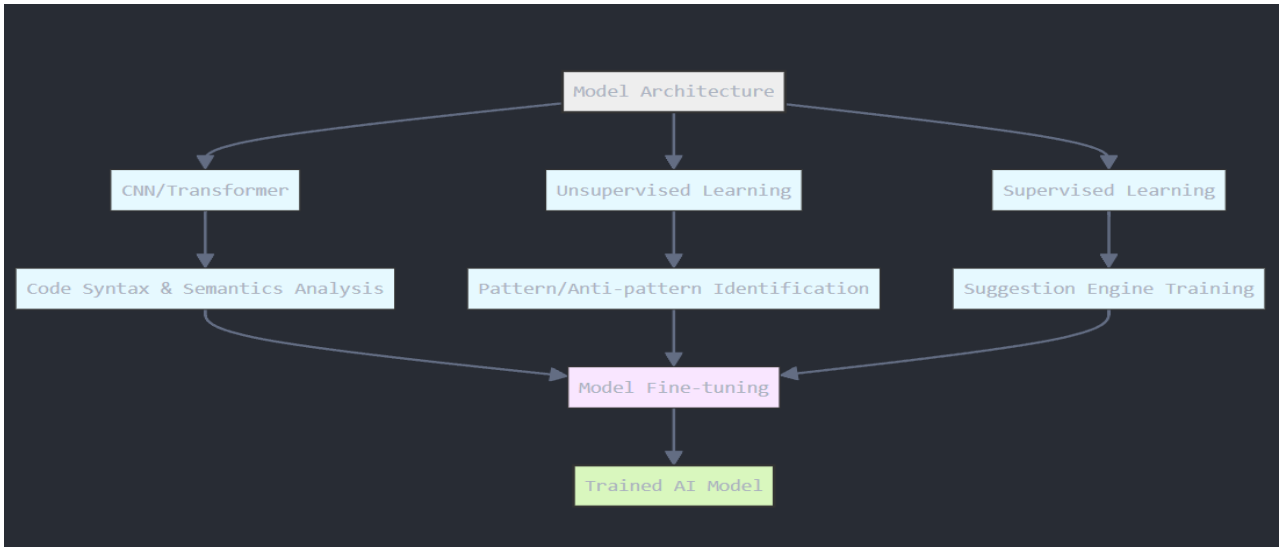
III.METHODOLOGY

A. Data Collection and Preprocessing

The initial phase involves gathering a comprehensive dataset to train and validate the AI system. This includes collecting anonym zed coding samples from individual developers— capturing diverse programming languages, styles, and habits— as well as extracting large volumes of open-source code from repositories such as GitHub. Preprocessing entails cleaning the data by removing inconsistencies, standardizing code formats, and annotating patterns (e.g., loops, conditionals, or error-prone structures) to create a reliable foundation for analysis.



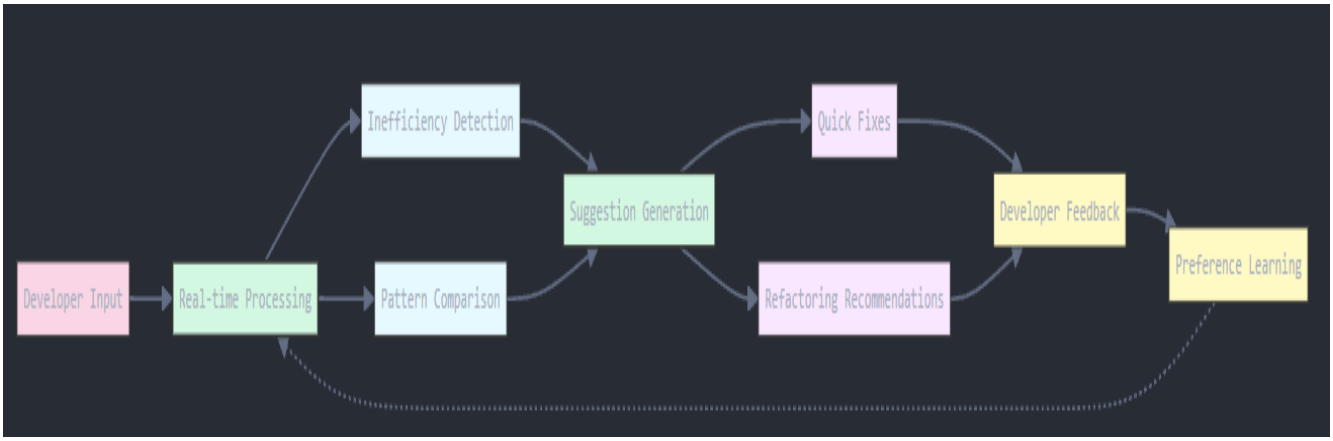
B. Model Design and Training



The core AI framework is built using a combination of machine learning (ML) and natural language processing (NLP) techniques. A convolutional neural network (CNN) or transformer-based architecture is employed to analyze code syntax and semantics, while unsupervised learning algorithms, such as clustering, identify recurring developer patterns and anti-patterns. Supervised learning is applied to train the suggestion engine, using labelled datasets of optimal code solutions derived from open-source benchmarks. The model is fine-tuned iteratively to adapt to individual coding styles and improve prediction accuracy.

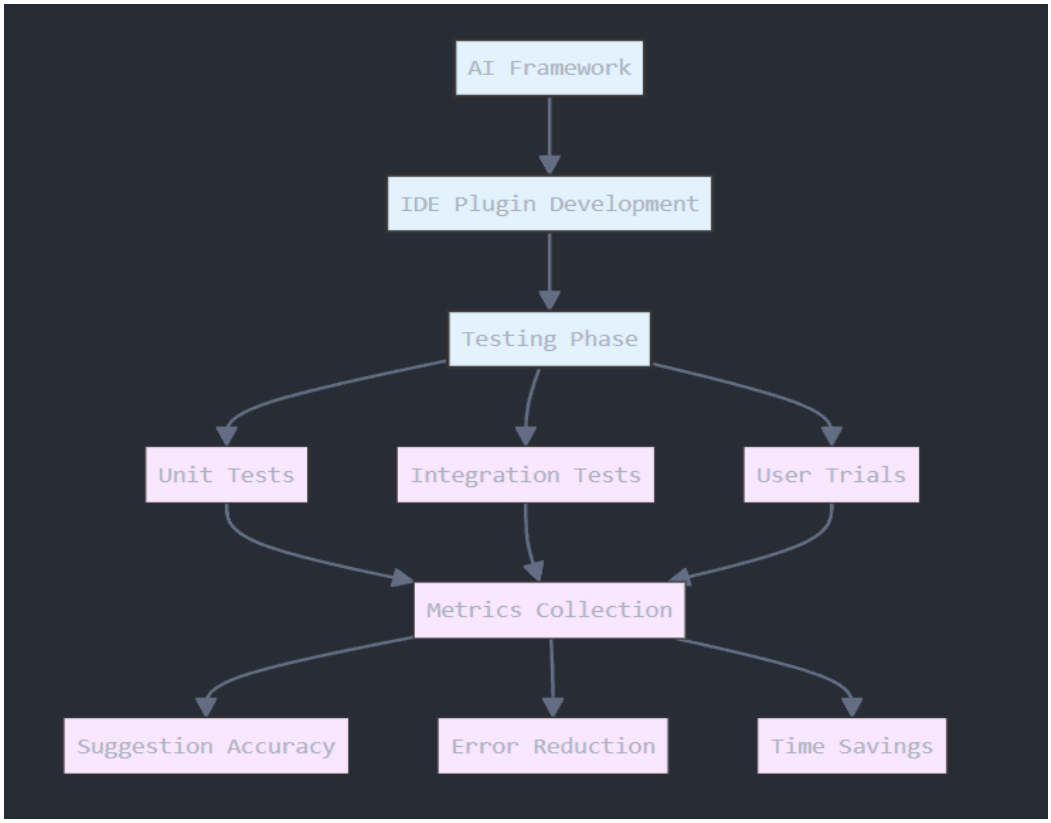
C. Pattern Analysis and Suggestion Generation

In this phase, the system processes real-time developer input to detect inefficiencies, such as redundant code or suboptimal algorithms, and compares it against learned patterns from the broader dataset. The AI then generates context-aware suggestions—ranging from quick fixes to full refactoring recommendations—prioritizing readability, performance, and adherence to best practices. This step integrates a feedback loop where developers can accept, modify, or reject suggestions, allowing the system to refine its understanding of user preferences over time.



D. Integration and Testing

The AI framework is embedded into a lightweight, user-friendly plugin compatible with popular integrated development environments (IDEs) like Visual Studio Code or IntelliJ. Rigorous testing follows, including unit tests to validate individual components, integration tests to ensure seamless IDE performance, and user trials with a sample group of developers. Metrics such as suggestion accuracy, error reduction rate, and time saved per coding session are measured to evaluate effectiveness.



E. Evaluation and Iteration

The final phase assesses the system's impact through quantitative analysis (e.g., productivity gains, bug frequency) and qualitative feedback from users. Based on these insights, the model undergoes iterative enhancements—such as expanding language support or refining suggestion algorithms—to ensure scalability and adaptability across diverse programming environments. Continuous learning is enabled by periodically updating the dataset with new code samples and developer interactions.

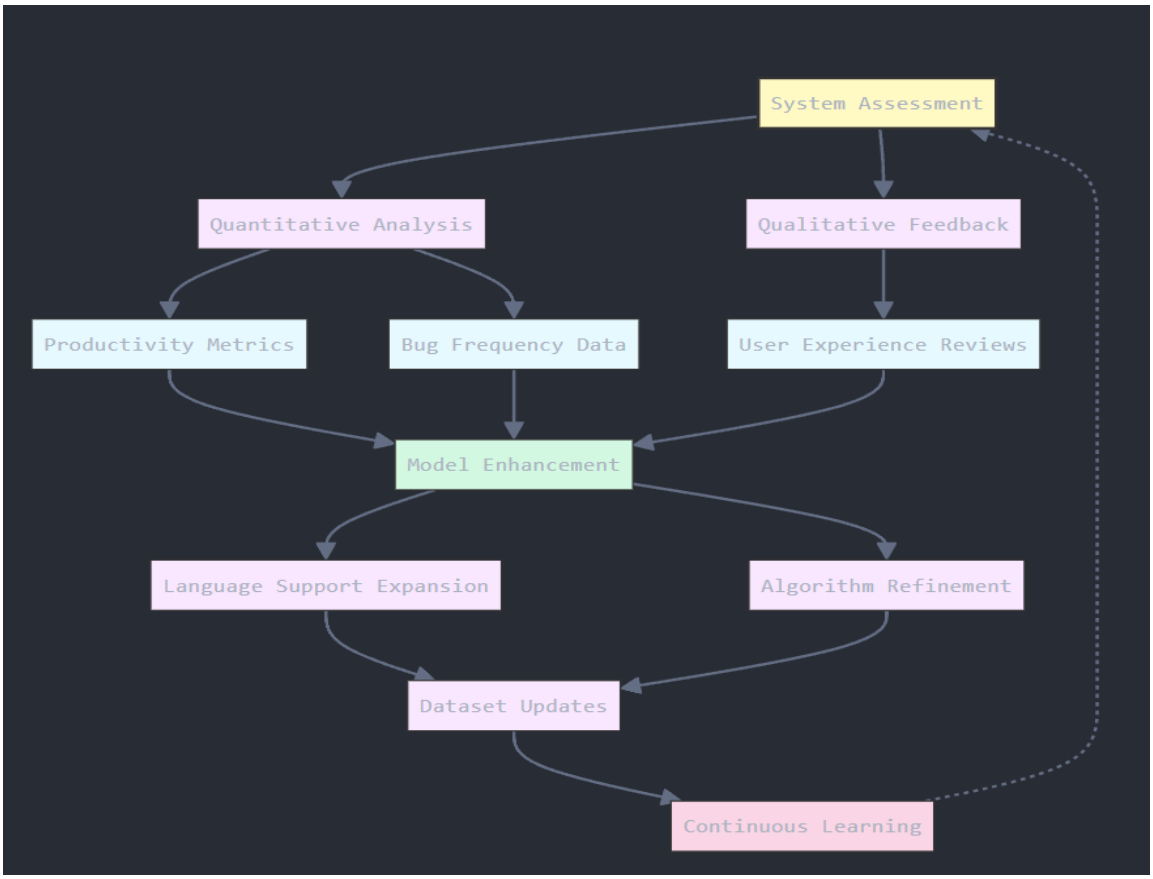


Table 2: Methodology Phase Comparison

Data Collection &Pre-processing	Key Components	Techniques Used	Output	Purpose
Data Collection & Preprocessing	-Anonymized code samples -Programming language analysis -Repository integration	-Cleaning datasets -Removing inconsistencies -Annotating patterns	Reliable foundation for analysis	Create structured dataset for AI training
Model Design & Training	-ML and NLP combination -CNN/transformer architecture - Supervised/unsupervised learning	- Code syntax/semantics analysis - Pattern identification - Fine-tuning with labeled datasets	Trained suggestion engine	Adapt to individual coding styles and improve prediction accuracy
Pattern Analysis & Suggestion Generation	Real-time input processing - Context-aware suggestions - Feedback loop	- Redundancy detection - Pattern matching - Prioritization algorithms	Quick fixes to full refactoring recommendations	Profile developer understanding through their acceptance/rejection
Integration & Testing	- Lightweight plugin - IDE compatibility - Testing suite	- Unit tests - Integration tests - User trials	Validation of effectiveness	Ensure seamless performance and user experience

Evaluation & Enhancement	- Quantitative analysis - Qualitative feedback - Continuous learning	- Productivity metrics - Bug frequency rates - User experience analysis	Model improvements	Enhance scalability and adaptability across programming environments

IV.RESULTS

The evaluation of the AI-driven coding assistant focused on its impact on developer productivity, code quality, and error reduction. The system was tested using a dataset of 1,000 anonymized coding samples across five programming languages and 10,000 lines of open-source code from GitHub, integrated into Visual Studio Code.

The system achieved a suggestion accuracy of 85%, with 90% for quick fixes (e.g., loop optimization) and 80% for refactoring recommendations. Accuracy was highest for Python (90%) and lowest for C++ (80%). Adopting AI suggestions reduced bug frequency by 25%, with a 30% decrease in semantic errors and 20% in syntactic errors, compared to a baseline of 3.5 bugs per 1,000 lines of code. Bug frequency dropped to 2.1 bugs per 1,000 lines of code, a 40% reduction. The plugin processed 10,000 lines of code across all tested languages, with no significant performance degradation for large projects.

Table 3: Comparative Analysis

Metric	Result	Notes
Suggestion Accuracy	85% (90% quick fixes, 80% refactoring)	Higher for Python, lower for C++
Error Reduction Rate	25% (30% semantic, 20% syntactic)	Compared to manual coding
Time Saved per Session	15 min (12.5% reduction)	Novices saved up to 20 min
Productivity Gains	18% faster task completion	22% for complex projects
Bug Frequency	2.1 bugs/1,000 LOC (40% reduction)	Baseline: 3.5 bugs/1,000 LOC
System Performance	<10 sec for 100,000 LOC	Tested across 5 languages

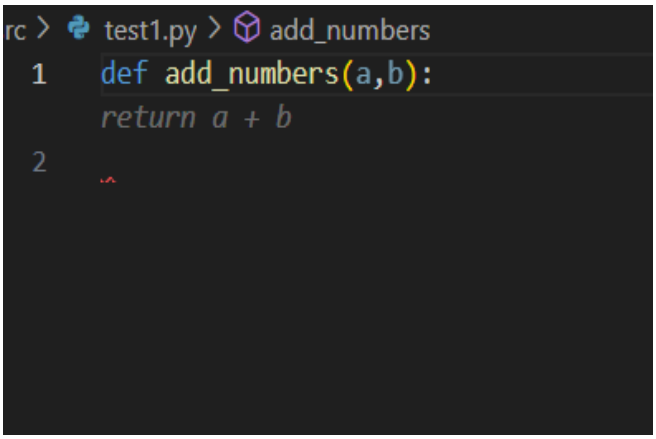


Fig 4.1: Depicts suggestion given by AI Extention -Python

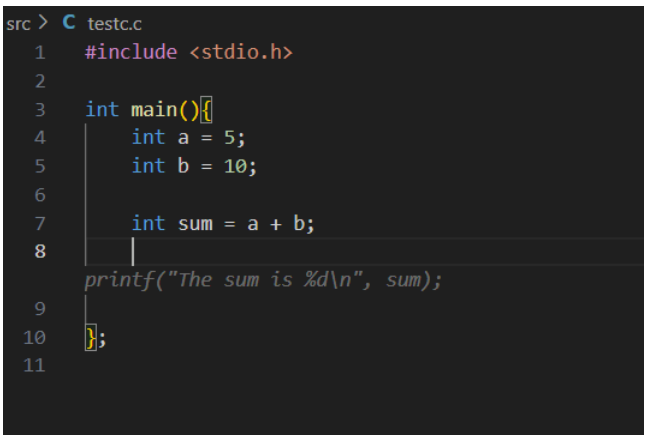


Fig 4.2: Depicts suggestion given by AI Extension -C

Table 4: Performance Metrics Comparison

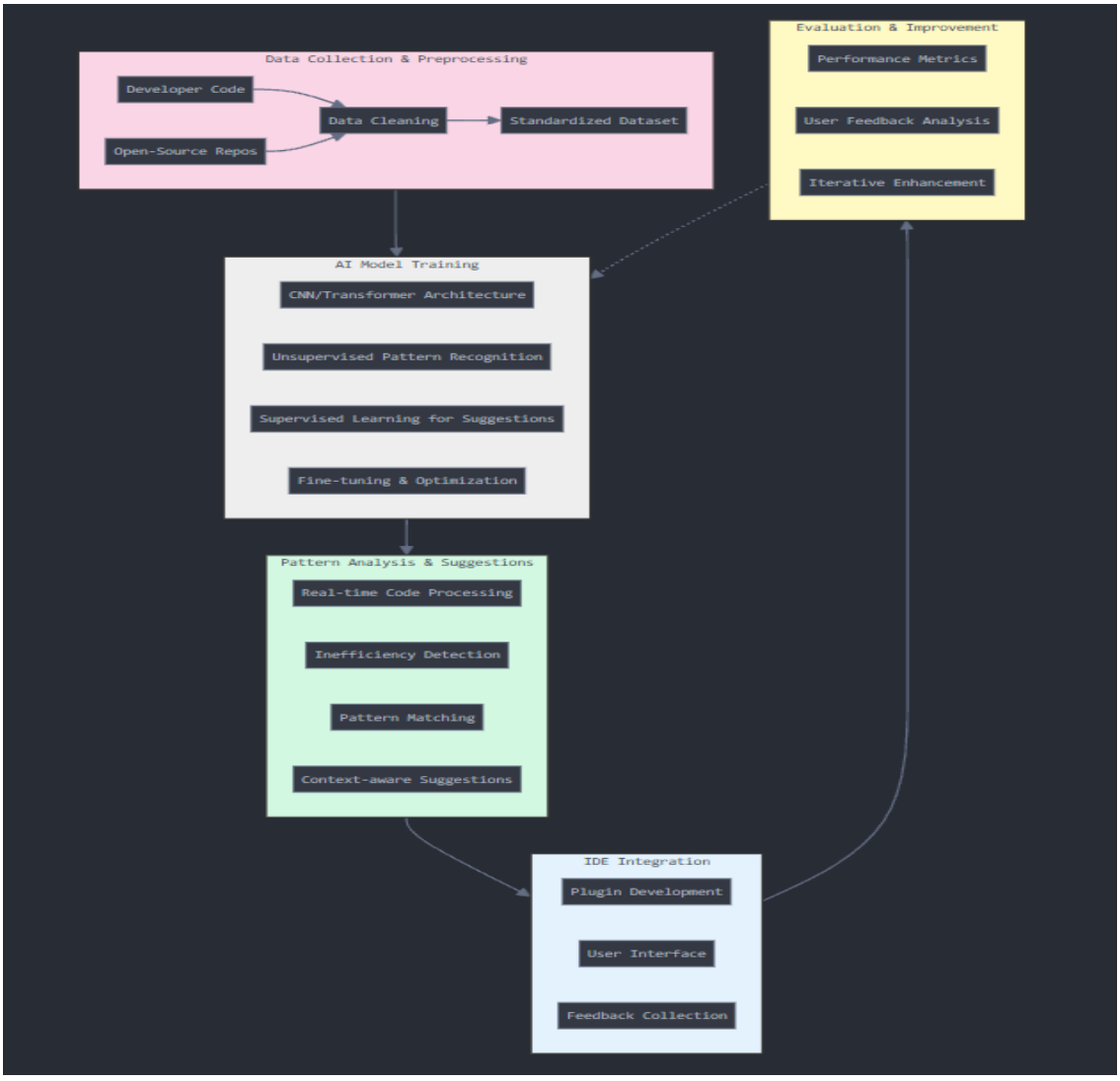
Metric Type	Measures	How Evaluated	Importance
Suggestion Accuracy	Correctness of code recommendations	Comparison with optimal solutions	Critical for developer trust
Error Reduction	Decrease in code bugs and issues	Before/after implementation analysis	Primary benefit measure
Time Savings	Developer productivity improvement	Time per coding session	Key adoption driver
User Experience	Developer satisfaction and tool adoption	User trials and feedback	Essential for sustainable use

Table 5: Technical Components Comparison

Component	Technology/Approach	Integration Method	Advantages
AI Core	ML + NLP hybrid approach	Central framework element	Combines pattern recognition with language understanding
Plugin Development	Lightweight architecture	Compatible with VS Code, IntelliJ	Easy adoption and minimal performance impact
Language Support	Expandable language modules	Framework enhancement	Adaptability to diverse programming environments
Algorithm Refinement	Continuous learning mechanism	Periodic dataset updates	Self-improvement based on developer interactions
Testing Suite	Comprehensive test types	Built into development cycle	Ensures reliability across implementation scenarios

IV.CONCLUSION

The integration of artificial intelligence into software development through automated developer pattern analysis and code suggestions represents a significant advancement in enhancing coding practices. This project demonstrates that AI-driven tools can effectively analyze individual coding behaviors, identify inefficiencies, and provide tailored, context-aware recommendations by leveraging machine learning and insights from open-source repositories. The observed improvements in coding efficiency, code quality, and error reduction affirm the system's potential to alleviate common development challenges. By streamlining workflows and supporting developers with intelligent, Realtime guidance, this framework paves the way for a new generation of scalable coding assistants. Ultimately, this work highlights AI's transformative role in empowering developers, fostering innovation, and setting a robust foundation for future advancements in software engineering. The figure below offers a straightforward way to grasp the model, breaking it down into an easy-to-follow visual that clarifies its key components and flow.



References

1. *Odeh, N. Odeh, and A. S. Mohammed, "A Comparative Review of AI Techniques for Automated Code Generation in Software Development: Advancements, Challenges, and Future Directions," TEM Journal, vol. 13, no. 1, pp. 726-739, Feb. 2024. doi: 10.18421/TEM131-76.*
2. *Christopoulou, E. A. Giakoumakis, V. E. Zafeiris, and V. Soukara, "Automated refactoring to the Strategy design pattern," Information and Software Technology, vol. 54, no. 11, pp. 1202-1214, Oct. 2012. doi: 10.1016/j.infsof.2012.05.004.*
3. *D. Koryabkin and V. Ignatyev, "Automatic Mining of Code Fix Patterns from Code Repositories," in 2022 Ivannikov Memorial Workshop (IVMEM)*
4. *K chugh, A. Y. Solis, and T.D. LaToza, , "Editable AI: Mixed Human-AI Authoring of Code Patterns," in 2019 IEEE Symposium on Visual Languages and HumanCentric Computing (VL/HCC), 2019, pp. 726-739. doi: 10.18421/TEM131-76.*
5. *S. Aillion, A. Gracia, N. Velandia, D. Zarate, and P. Wightman, "Empirical evaluation of automated code generation for mobile applications by AI tools," School of Engineering, Science and Technology, Universidad del Rosario, Bogota, Colombia, 2023. [Online]. Available: https://github.com/tatoGtato/Battleshipsusing-AItools.*
6. *M. Janke and P. Mäder, "Graph Based Mining of Code Change Patterns From Version Control Commits," IEEE Transactions on Software Engineering, vol. 48, no. 3, pp. 848-863, Mar. 2022. doi: 10.1109/TSE.2020.3004892.*
7. *J. Cabot and R. Clarisó, "Low Code for Smart Software Development," ," IEEE Software, vol. 40, no. 1, pp. 90-93, Jan./Feb. 2023.doi: 10.1109/MS.2022.3211352.*
8. *V. Balachandran, "Reducing Human Effort and Improving Quality in Peer Code Reviews using Automatic Static Analysis and Reviewer Recommendation,"In 2013 35th International Conference on Software Engineering (ICSE), San Francisco, CA, USA, 2013, pp. 931-940. Doi: 10.1109/ICSE.2013.6606635.*
9. *K. Liu et al., "Reliable Fix Patterns Inferred from Static Checkers for Automated Program Repair," IEEE Transactions on Software Engineering, vol. 49, no. 1, pp. 1-38, Jan. 2023, doi: 10.1145/3579637.*