International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 09 Issue: 09 | Sept - 2025 SJIF Rating: 8.586 ISSN: 2582-3930

A Literature Review on AI-Powered Smart Code Base Navigator

Radhika S K[1], Rashmitha R[2], Sanjana N[2], Shanthala M N[2], Sukanya G[2]

[1]*Assistant Professor, Department of Computer Science and Engineering, JNNCE, Shivamogga, Karnataka, India*
[2]*UG Students, Department of Computer Science and Engineering, JNNCE, Shivamogga, Karnataka, India*

Abstract - In contemporary software development, the vast size of codebases poses challenges in locating, comprehending, and reusing code. Traditional search tools that rely on keywords often fall short in capturing a developer's true intent, thus impeding efficiency. This initiative introduces an AI-Powered Smart Code Base Navigator, a system designed to facilitate semantic code search, context-aware code completion, and streamlined navigation within extensive Python codebases. By utilizing pre-trained models such as CodeBERT and retrieval-augmented methods, the system can interpret natural language queries and deliver pertinent code. Additionally, it offers features like jump-to-definition and intelligent suggestions, boosting developer productivity. As a web-based tool, the navigator exemplifies how AI can effectively connect natural language with programming code, greatly enhancing software engineering methodologies.

***Key Words*: AI, Code Navigation, Semantic Search, Retrieval- Augmented Generation, Large Language Models.**

INTRODUCTION

In recent times, software development has grown more intricate, with developers often dealing with extensive, multi-layered codebases. Tasks such as code navigation, understanding dependencies, and reusing existing code can be quite time-consuming when using traditional keyword-based searches, which lack semantic comprehension. The advent of AI and natural language processing has paved the way for smarter software tools. Drawing inspiration from advancements in pre-trained code models like CodeBERT, this project presents the AI-Powered Smart Code Base Navigator. This system aims to enhance developer efficiency by offering semantic code search, intelligent auto-completion, and context- aware navigation. Unlike conventional methods, our tool comprehends natural language queries, retrieves pertinent code snippets, and suggests code based on usage patterns. Developed as a web-based application, it ensures easy access and smooth integration into a developer's workflow. By integrating semantic embeddings, transformer models, and efficient indexing, the navigator facilitates quick and accurate navigation for large-scale projects. This tool ultimately demonstrates how AI-driven solutions can revolutionize software development, making coding more efficient.

Literature Survey

In this section, various authors have presented various Emotion detection techniques.

A. Code Generation using Transformer Models

J. R. Mahajan and K. Geetanjali [1] proposed transformer-based models have revolutionized AI-assisted programming by applying self-attention to code and natural language. They are trained on massive code repositories (e.g., GitHub, CodeSearchNet), learning patterns that improve code completion, language translation, and automated bug fixing. By deeply modeling syntax and semantics, transformers can produce highly accurate, human-like code snippets. In practice this yields significant productivity gains in tasks like predictive code completion. However, these models sometimes generate flawed or insecure code and may inadvertently reproduce copyrighted or irrelevant patterns from their training data. Researchers are now working on data curation and architectural improvements to reduce hallucinations and ensure that generated code is correct and safe.

B. Meta-RAG on Large Codebases Using Code Summarization

Tawosi et al. [2] proposed *Meta-RAG*, a multi-agent retrieval-augmented generation framework for debugging large codebases. It first uses an LLM to summarize or condense the codebase (shrinking it by ≈80%), then applies information retrieval and an LLM reasoning agent to pinpoint bug locations. This drastically reduces the search space: on the SWE-bench Lite dataset Meta-RAG achieved about 84.7% accuracy at locating the buggy file and 53.0% at the function level, state-of-the-art results. By combining summarization and retrieval, Meta-RAG efficiently handles very large code repositories. Its performance depends on the quality of those summaries and requires substantial computation for summarization upfront. In future work, optimizing this trade-off is key, as poor summaries or limited compute could undermine the method's effectiveness.

C. Position: Intelligent Coding Systems Should Write Programs with Justifications

Xu et al. [3] argue that AI coding assistants ought to generate not only code but also human-understandable justifications for their outputs. They introduce two properties of good explanations: cognitive alignment (the rationale matches how a user thinks about the task) and semantic faithfulness (the explanation accurately reflects the code's true behavior). Traditional approaches often fail these criteria. Instead, they advocate a neuro-symbolic approach where symbolic rules guide

the model during training and neural components capture semantics. In this framework, the system automatically checks that its explanation is consistent with the generated code. While conceptual and not yet implemented at scale, this vision aims to build trust by ensuring AI-generated code comes with clear, verifiable reasoning about how and why it works.

D. Conversational AI as a Coding Assistant: Understanding Programmers' Interactions with LLMs

Akhoroz and Yildirim [4] surveyed 143 student developers to understand how they use LLM-based chatbots for coding. Students found LLMs helpful for accelerating tasks and clarifying concepts, but also reported issues: the AI's answers were sometimes inaccurate, lacked awareness of the entire project, and could encourage over-reliance. A notable fraction of students avoided using LLMs altogether to force themselves to learn independently or due to distrust and ethical concerns over AI code. The authors conclude that while LLM assistants have educational value (e.g., interactive debugging or explanations), they need better context retention and transparency. They suggest design improvements such as stronger context awareness and clearer citations of information. This study highlights both the potential and the pitfalls of conversational coding assistants in real programming education.

E. A Deep Dive into Retrieval-Augmented Generation for Code Completion: Experience on WeChat

Yang et al. [5] examines retrieval-augmented generation (RAG) for C++ code completion within WeChat's large closed-source codebase. They compare *identifier-based* and *similarity-based* RAG, testing lexical (keyword) and semantic retrieval approaches across many LLMs. The study found that RAG consistently boosts completion accuracy: in particular, similarity-based retrieval (e.g., semantic search) outperformed simple identifier matching, and the best results came from combining both lexical and semantic retrieval methods. Developer feedback confirmed these improvements in practice. However, the authors caution that these empirical gains depend on the codebase's characteristics (WeChat's code and libraries) and note that standard automatic metrics may not fully capture real developer satisfaction. Nonetheless, this work demonstrates that integrating smart retrieval can greatly enhance code suggestion systems.

F. Deep Semantics-Enhanced Neural Code Search

Yin et al. [6] introduce *SENCS*, a neural code search model that fuses structural and semantic information. SENCS first serializes a code's dependency graph to capture its structure, then uses a two-stage attention network to emphasize meaningful code tokens. This joint encoding aligns a developer's query with relevant code snippets by deeply understanding the code's intent. On benchmarks (e.g., CodeSearchNet and JavaNet), SENCS notably outperformed previous code search methods: for instance, it increased metrics like MRR and SR@1 by double-digit percentages over the prior best models. These gains show that modeling rich semantic and structural patterns helps. A trade-off is that SENCS requires large amounts of training data to capture these features, and its complexity could hinder scalability or generalization to smaller or very different code corpora.

G. Intelligent Python Code Analyzer (IPCA)

Thottam et al. [7] present IPCA, an AI-driven static analyzer designed for learning Python programmers. Unlike simple linters, IPCA uses advanced syntax and semantic analysis (e.g., AST parsing, API usage checks) to provide context-aware feedback. It inspects code structure to flag errors or inefficiencies and suggests improvements beyond mere grammar fixes. IPCA integrates with common Python IDEs to offer interactive, educational hints – effectively teaching coding concepts as it checks code. Its strength lies in tailoring feedback to novice programmers and explaining why something is wrong. However, IPCA's evaluation has so far focused on classroom or student settings, and quantitative comparisons to commercial linters are lacking. Thus, while promising for education, its overall robustness and effectiveness in diverse real-world scenarios remain to be validated.

H. EVOR: Evolving Retrieval for Code Generation

Su et al. [8] propose *EVOR*, a retrieval-augmented code generation pipeline that dynamically updates both the query and the external knowledge base. Traditional pipelines use fixed, static documents; EVOR continuously evolves search queries and aggregates diverse sources (e.g., updated libraries, online code) to adapt to changing contexts. They created new benchmarks focusing on frequently-updated APIs and rare languages and show EVOR dramatically improves execution accuracy: it achieved roughly 2–4× higher correct code generation rates than recent baselines like Reflexion or DocPrompting. This indicates that EVOR's strategy of synchronous query and corpus expansion is effective. On the downside, maintaining evolving corpora is costly: EVOR requires high latency and computational effort to continually fetch and process new information. There is also a risk that noisy or biased retrieved data could lead to incorrect outputs. Balancing accuracy gains against efficiency and bias risks is thus crucial for EVOR's practical deployment.

I. REINFOREST: Reinforcing Semantic Code Similarity for Cross-Lingual Code Search Models

Saieva et al. [9] introduce REINFOREST, a method to boost cross-language code search by embedding dynamic runtime information into static code representations. REINFOREST incorporates execution-derived features into the training phase (without needing to run code at query time) and trains on both similar (positive) and dissimilar (negative) code-example pairs. This enriches the model's notion of code similarity. Evaluations show REINFOREST substantially outperforms previous cross-language search tools – in some cases by up to 44.7% on accuracy metrics. They also find that even including a single well-chosen positive/negative example during training yields large gains,

underlining the importance of contrastive learning in code search. While powerful, REINFOREST requires gathering aligned code examples across languages, which may be difficult for under-resourced languages. Its effectiveness also depends on the availability of quality static and dynamic features for each codebase.

J. A Comprehensive Survey of AI-Driven Advancements and Techniques in Automated Program Repair and Code Generation

Anand et al. [10] systematically review 27 recent works on AI-assisted bug fixing and code synthesis. They categorize approaches into groups like search-based or semantic repair, and ML/LLM-based code generation. For APR, they highlight methods that use LLMs for context-aware patching of semantic errors and vulnerabilities. For code generation, they contrast general LLM solutions with task-specific models, noting techniques like identifier-based fine-tuning and instruction- level tuning to improve output. The survey discusses strengths (e.g., iterative feedback loops boosting accuracy) and weaknesses of each approach. They note common challenges: many methods rely on limited datasets/benchmarks, making it hard to generalize across languages or domains. The authors stress gaps such as ensuring correctness and integrating domain knowledge. Overall, this work provides a structured overview of current techniques and suggests that future research focus on robust evaluation and cross-domain applicability.

K. Code Search Is All You Need: Improving Code Suggestions with Code Search

Chen et al. [11] demonstrate that code suggestion models can be greatly enhanced by integrating code search. Their retrieval- augmented framework first searches a large codebase for snippets similar to the developer's query, then uses these examples to guide a language model's suggestions. Testing multiple language models, they found that this approach yields very large gains: code completion BLEU scores improved by up to ~53.8% and code generation by ~130.8% compared to no retrieval. In essence, a practical retrieval step can compensate for some of the weaknesses of generative models. This also suggests a lightweight strategy (using a pre-built code index) as an alternative to fine-tuning massive models. A limitation is that quality hinges on the retrieval corpus: if the codebase has limited relevant examples, performance will suffer. Additionally, this method may not handle novel queries well, since it relies on existing code.

L. A Transformer-Based Approach for Smart Invocation of Automatic Code Completion

De Moor et al. [12] address *when* an IDE should invoke code completion to avoid interrupting the developer. They collected 200k real developer interactions and trained a small transformer classifier to predict optimal invocation points based on code context and editor telemetry. Their model significantly outperformed a naive baseline while maintaining low latency. In deployment with 34 developers (≈74k invocations), it effectively filtered out irrelevant suggestion prompts. This work shows that context-aware invocation policies – not just the content of suggestions – are important for developer productivity. However, the approach depends on representative telemetry data: different languages, coding styles, or IDE setups might require retraining. Also, adding such a layer increases system complexity and requires accurate modeling of developer intent signals, which may not transfer easily between development environments.

M. LLMs: Understanding Code Syntax and Semantics for Code Analysis

Ma et al. [13] examine how well state-of-the-art LLMs (GPT-4, GPT-3.5, StarCoder, CodeLlama) understand code at syntax vs. semantic levels. They design tasks to test parsing of abstract syntax trees and control-flow, as well as comprehension of dynamic behavior across languages. The results are clear: all tested LLMs can handle syntax tasks relatively well, acting like a parser, but they often fail to fully grasp semantics and runtime logic. In fact, the models frequently hallucinate or infer incorrect behaviors when reasoning about code. This suggests that while LLM-generated code is usually syntactically correct, its deeper correctness is brittle. The authors conclude that relying on LLMs for code analysis or verification is risky; external checks or hybrid methods (e.g., symbolic analysis) are needed to ensure reliability

N. Joint Embedding of Semantic and Statistical Features for Effective Code Search

Kong et al. [14] propose *JessCS*, a code search system that jointly embeds semantic meanings with statistical code features. The idea is to capture both what the code does (semantics from comments/descriptions) and how often patterns appear (statistics like token frequency or API usage). JessCS learns a unified vector space incorporating both types of features. Evaluated on a large dataset (≈1M code snippets), JessCS outperformed a uniform embedding baseline by about 13% on search accuracy metrics. This shows that combining semantic context with structural statistics can better match queries to code. The downside is added complexity: extracting multiple feature types increases the model's size and preprocessing time, making it heavier to deploy at large scale or on-the-fly for huge code repositories.

O. Explainable AI for Pre-Trained Code Models: What Do They Learn? When They Do Not Work?

Mohammadkhani et al. [15] apply explainable AI (attention analysis) to interpret pretrained models like CodeBERT and GraphCodeBERT on tasks such as code documentation, code refinement, and translation. By examining which tokens the models attend to, they identify patterns of what the models consider important when they succeed, and discover why they fail on seemingly easy examples. For instance, they observe cases where the attention focuses on irrelevant code tokens, leading to mistakes. Their analysis yields insights into the

models' blind spots and guides recommendations (e.g. modifying training data or attention mechanisms) to mitigate these issues. While this study does not directly improve model accuracy, it provides valuable transparency. The conclusions are mostly qualitative, helping researchers understand and trust code models rather than creating a new model.

P. Search4Code: Code Search Intent Classification Using Weak Supervision

Rao et al. [16] address the problem of understanding what programmers *mean* when they write natural language search queries. They collected over 1 million real C# and Java queries from Bing and used weak supervision to label each query's intent (e.g., seeking code snippet, documentation, or debugging help). They then trained a CNN classifier on this data, achieving about 76–77% accuracy in predicting the intent class. They also released this large dataset to the community. By classifying intent, a code search engine could route queries more effectively (for instance, prioritizing code examples vs. theoretical explanations). A potential drawback is that weak labels can be noisy, and 77% accuracy means some queries will be misclassified, which might lead the search to misunderstand the user's actual goal.

Q. PSCS: A Path-based Neural Model for Semantic Code Search

Sun et al. [17] propose PSCS, a neural model that embeds both the semantics and structure of code by leveraging AST (abstract syntax tree) paths. PSCS trains on hundreds of thousands of queries–code pairs, walking along AST paths to capture relationships between code tokens and constructs. In large-scale experiments (330k training pairs), PSCS achieved a 47.6% success rate (Top-10 accuracy) and 30.4% MRR, significantly beating earlier deep learning baselines (like DeepCS and CARLCS). Importantly, an ablation study showed that including the structural AST path information markedly improved results, confirming the value of code structure. The complexity lies in parsing and encoding ASTs: for massive codebases, constructing these paths can be resource-intensive, and performance depends on the parser's accuracy. Nevertheless, PSCS demonstrates that modeling syntax paths is a powerful way to improve semantic code search.

R. CodeBERT: A Pre-Trained Model for Programming and Natural Languages

Feng et al. [18] introduced CodeBERT, a bimodal transformer pre-trained on both code (from six languages) and natural language pairs. By learning joint NL–code representations, CodeBERT can be fine-tuned for tasks like code search or summarization. The hybrid pre-training objective (including a "replaced token detection" task) lets it leverage unlabeled code and paired doc–code data together. In benchmarks, CodeBERT set new state-of-the-art results on code search and documentation generation after fine-tuning. This work established large-scale NL–PL pre-training in the software domain. The trade-offs are familiar for large models: CodeBERT requires heavy computation to train, and it must be fine-tuned per task. It also has a fixed context window, limiting how much code it can process at once. Without further adaptation, it may struggle with very long code inputs or highly specialized programming languages.

S. Adaptive Deep Code Search

Ling et al. [19] present *AdaCS*, an adaptive code search model designed to transfer across codebases. AdaCS is first trained on a large public corpus (e.g., GitHub code) and then adapted to a specific target repository. It decouples the learning of general syntactic matching (via neural networks) from domain-specific word meanings. When applied to a new codebase, AdaCS learns the meanings of project-specific terms and constructs new matching matrices, while reusing the previously learned syntactic model. In experiments on industrial Java projects, this approach boosted the top-5 search hit rate from 65.9% (baseline) to 88.2%. Thus, AdaCS can be trained once and reused across projects. The limitation is that it still requires substantial initial training data and effort to adapt to each new codebase. Rare domains with very different vocabulary may require additional labeled examples to reach optimal performance.

T. When Deep Learning Met Code Search

Cambronero et al. [20] conduct a systematic comparison of neural code search architectures. They introduce *UNIF*, a simple supervised model using bag-of-words embeddings with attention, and compare it to more complex RNN-based models (CODEnn, SCS). Counterintuitively, UNIF outperforms CODEnn and SCS on their benchmarks: for example, on two test sets (Java-50 and Android-287) UNIF retrieved more correct results in the top-10 than the others. They find that minimal supervision (tuning embedding weights) significantly helps, but adding sophisticated sequential architectures offers little extra gain. Their results suggest that neural code search can be effective even with simpler models, though they note that supervised learning must still carefully match queries and code. This work serves as a reminder that model complexity does not always translate to better real-world performance.

TABLE: LITERATURE SURVEY SUMMARY

Authors	Title	Methodology	Remarks
J. R. Mahajan and K. Geetanjali 2025	Code Generation using Transformer Models	Reviews transformer-based code generation.	Pros: Strong syntax/semantics. Cons: Can produce invalid or insecure code; ethical issues.
V. Tawosi, S. Alamir, X. Liu, and M. Veloso 2025	Meta-RAG on Large Codebases	Multi-agent framework for bug localization and patch generation.	Pros: Reduces code size by 80%; SOTA performance. Cons: Resource-heavy; quality depends on LLM summaries.
X. Xu, S. Feng, Z. Su, C. Wang, and X. Zhang 2025	Intelligent Coding Systems Should Write Programs with Justifications	Advocates for neuro-symbolic systems with justifications.	Pros: Builds user trust with explainable outputs. Cons: Conceptual framework only; not yet implemented.
M. Akhoroz and C. Yildirim 2025	Conversational AI as a Coding Assistant	Survey of students' use of AI for coding.	Pros: Valuable for debugging and learning. Cons: Limited to student views; concerns over accuracy and over-reliance.
Z. Yang, C. Wang, T. Peng, H. Huang, Y. Deng, and C. Gao 2025	A Deep Dive into RAG for Code Completion	Empirical study of RAG for C++ code completion.	Pros: Improved completion with combined retrieval. Cons: Results may not generalize; automated metrics are limited.
Y. Yin, L. Ma, Y. Gong, Y. Shi, F. Wahab, and Y. Zhao 2024	Deep Semantics-Enhanced Neural Code Search 2024	Neural model for code search using deep semantic embeddings.	Pros: Improves search accuracy; better captures semantic intent. Cons: Requires large training data; may not generalize well.
C. Thottam, N. Fernandes, R. Joseph, and I. Mirza 2024	Intelligent Python Code Analyzer (IPCA)	AI-based static analysis tool for Python.	Pros: Provides interactive, educational feedback. Cons: Focused on education; lacks quantitative comparison.
H. Su, S. Jiang, Y. Lai, H. Wu, B. Shi, C. Liu, Q. Liu, and T. Yu 2024	EVOR: Evolving Retrieval for Code Generation	Retrieval-augmented code generation using evolving queries.	Pros: Higher execution accuracy; adaptable. Cons: High latency and energy use; risk of bias.
A. Saievat, S. Chakraborty, and G. Kaiser 2024	REINFOREST: Reinforcing Semantic Code Similarity	Enhances cross-language code search with a Semantic Similarity Score.	Pros: Outperforms SOTA; robust with limited data. Cons: Relies on suitable SSS inputs; less practical for large codebases.
A. Anand, N. Yadav, A. Gupta, and S. Bajaj 2024	A Comprehensive Survey of AI-Driven Advancements and Techniques in Automated Program Repair and Code Generation	Systematic literature review of AI-driven techniques in program repair and code generation.	Pros: Provides a structured categorization; valuable resource for researchers. Cons: Dependent on the quality of datasets; limited integration of domain-specific knowledge.
J. Chen, X. Hu, Z. Li, C. Gao, X. Xia, and D. Lo 2024	Code Search Is All You Need: Improving Code Suggestions with Code Search	Demonstrates that code search itself can drive effective code completion and suggestions	Pros: Lightweight compared to large language models. Cons: Limited by the quality and size of the indexed codebase.

A. de Moor, A. van Deursen, and M. Izadi 2024	A Transformer-Based Approach for Smart Invocation of Automatic Code Completion	Leverages transformer models to provide context-aware auto-completion.	Pros: High accuracy in suggesting context-specific completions. Cons: Requires significant computational resources.
W. Ma, W. Wang, S. Liu, Y. Liu, Q. Hu, L. Li, Z. Lin, C. Zhang, and L. Nie 2024	LLMs: Understanding Code Syntax and Semantics for Code Analysis	Evaluates GPT-4, GPT-3.5, StarCoder, and CodeLlama across tasks involving syntax, static behavior.	Pros: Strong syntax understanding. Cons: Weak in semantic and dynamic analysis.
X. Kong, S. Kong, M. Yu, and C. Du 2022	Joint Embedding of Semantic and Statistical Features for Effective Code Search	Combines semantic embeddings with statistical features into a joint vector space.	Pros: Balances semantic understanding,higher precision. Cons: Feature engineering increases complexity.
A. H. Mohammadkhani, C. Tantithamthavorn, and H. Hemmati 2022	Explainable AI for Pre-Trained Code Models: What Do They Learn? When They Do Not Work?	Investigates pre-trained code models (like CodeBERT) using explainable AI techniques to understand their decision-making and failure points.	Pros: Provides transparency and insights into model behavior. Cons: Explanations may be shallow.
N. Rao, C. Bansal, and J. Guan 2021	Search4Code: Code Search Intent Classification Using Weak Supervision	Introduces an intent classification system for code search queries using weak supervision.	Pros: Improves relevance by aligning queries with search intent. Cons: Weak supervision labels may be noisy.
Z. Sun, Y. Liu, C. Yang, and Y. Qian 2020	PSCS: A Path-based Neural Model for Semantic Code Search	Uses abstract syntax tree (AST) paths with neural networks to represent code semantics.	Pros: Effectively captures syntactic and semantic code patterns. Cons: Parsing large codebases into ASTs is resource-heavy.
Z. Feng et al. 2020	CodeBERT: A Pre-Trained Model for Programming and Natural Languages	Pre-trained transformer model trained on massive code and natural language corpora.	Pros: Strong baseline for code search, summarization, and completion. Cons: Requires fine-tuning.
C. Ling, Z. Lin, Y. Zou, and B. Xie 2020	Adaptive Deep Code Search	Proposes an adaptive framework that dynamically adjusts embeddings and retrieval strategies.	Pros: Flexible and generalizable. Cons: Adaptation process can be complex.
S. Kim, J. Cambronero, H. Li, and K. Sen 2019	When Deep Learning Met Code Search	Compares multiple neural code search models (NCS, CODEnn, SCS) with a simpler baseline called UNIF on common datasets.	Pros: UNIF outperforms more complex models; supervision improves results significantly. Cons: Complex networks may overfit; evaluation depends on manually set thresholds.

CONCLUSIONS

The AI-Powered Smart Code Base Navigator showcases the potential of incorporating AI methods like semantic search, intelligent recommendations, and natural language queries to boost developer efficiency. By utilizing sophisticated models and retrieval techniques, this initiative tackles significant obstacles in navigating extensive and intricate codebases. The system improves code comprehension, offers context-sensitive support, and shortens the time developers need to find and understand pertinent sections of the code.

The work also highlights future opportunities, such as incorporating domain-specific knowledge, expanding support across diverse programming languages, and improving scalability for industrial-scale software systems. Overall, this project contributes towards advancing intelligent software engineering tools, bridging the gap between human developers and complex code ecosystems.

REFERENCES

[1] J. R. Mahajan and K. Geetanjali, "Code generation using transformer models," *Int. J. Adv. Res. Sci., Commun. Technol.*, vol. 5, no. 3, pp. 90–96, May 2025.

[2] V. Tawosi, S. Alamir, X. Liu, and M. Veloso, "Meta-RAG on large codebases using code summarization," *arXiv preprint* arXiv:2508.02611, 2025.

[3] X. Xu, S. Feng, Z. Su, C. Wang, and X. Zhang, "Position: Intelligent coding systems should write programs with justifications," *arXiv preprint* arXiv:2508.06017, 2025.

[4] M. Akhoroz and C. Yildirim, "Conversational AI as a coding assistant: Understanding programmers' interactions with and expectations from large language models for coding," *arXiv preprint* arXiv:2503.16508, 2025.

[5] Z. Yang, C. Wang, T. Peng, H. Huang, Y. Deng, and C. Gao, "A deep dive into retrieval-augmented generation for code completion: Experience on WeChat," *arXiv preprint* arXiv:2507.18515, 2025.

[6] Y. Yin, L. Ma, Y. Gong, Y. Shi, F. Wahab, and Y. Zhao, "Deep Semantics-Enhanced Neural Code Search," Electronics, vol. 13, no. 23, p. 4704, Nov. 2024.

[7] C. Thottam, N. Fernandes, R. Joseph, and I. Mirza, "Intelligent Python code analyzer (IPCA)," *Int. J. Creative Res. Thoughts (IJCRT)*, vol. 12, no. 3, pp. 1987–1997, Mar. 2024.

[8] H. Su, S. Jiang, Y. Lai, H. Wu, B. Shi, C. Liu, Q. Liu, and T. Yu, "EVOR: Evolving retrieval for code generation," *arXiv preprint* arXiv:2402.12317, 2024.

[9] A. Saievat, S. Chakraborty, and G. Kaiser, "REINFOREST: Reinforcing Semantic Code Similarity for Cross-Lingual Code Search Models," *arXiv preprint* arXiv:2305.03843, 2024.

[10] A. Anand, N. Yadav, A. Gupta, and S. Bajaj, "A comprehensive survey of AI-driven advancements and techniques in automated program repair and code generation," *arXiv preprint arXiv:2411.07586*, 2024.

[11] J. Chen, X. Hu, Z. Li, C. Gao, X. Xia, and D. Lo, "Code Search Is All You Need? Improving Code Suggestions with Code Search," in Proc. 46th Int. Conf. on Software Engineering (ICSE '24), Lisbon, Portugal, Apr. 14–20, 2024,

[12] A. de Moor, A. van Deursen, and M. Izadi, "A Transformer-Based Approach for Smart Invocation of Automatic Code Completion," in *Proc. 1st ACM Int'l Conf. on AI-Powered Software (AIware '24)*, Porto de Galinhas, Brazil, Jul. 2024.

[13] W. Ma, W. Wang, S. Liu, Y. Liu, Q. Hu, L. Li, Z. Lin, C. Zhang, and L. Nie, "LLMs: Understanding code syntax and semantics for code analysis," *arXiv preprint* arXiv:2305.12138, 2024.

[14] X. Kong, S. Kong, M. Yu, and C. Du, "Joint Embedding of Semantic and Statistical Features for Effective Code Search," Applied Sciences, vol. 12, no. 19, art. 10002, Oct. 2022.

[15] A. H. Mohammadkhani, C. Tantithamthavorn, and H. Hemmati, "Explainable AI for Pre-Trained Code Models: What Do They Learn? When They Do Not Work?" arXiv preprint arXiv:2211.12821, 2022.

[16] N. Rao, C. Bansal, and J. Guan, "Search4Code: Code Search Intent Classification Using Weak Supervision," in Proc. 18th Int. Conf. on Mining Software Repositories (MSR), pp. 575–579, 2021.

[17] Z. Sun, Y. Liu, C. Yang, and Y. Qian, "PSCS: A path-based neural model for semantic code search," *arXiv preprint* arXiv:2008.03042, 2020.

[18] Z. Feng *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547.

[19] C. Ling, Z. Lin, Y. Zou, and B. Xie, "Adaptive deep code search," in *Proceedings of the 28th International Conference on Program Comprehension (ICPC '20)*, Seoul, Republic of Korea, Oct. 2020.

[20] S. Kim, J. Cambronero, H. Li, and K. Sen, "When deep learning met code search," in *Proc. 27th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE '19)*, 2019, pp. 964–974.