International Journal of Scientific Research & Engineering Trends
Volume 9, Issue 1, Jan-Feb-2023, ISSN (Online): 2395-566X

From Code Completion to Collaborative Intelligence: LLM-Enabled Developer Copilots for Java Code Understanding and Refactoring

Sriram Ghanta
Senior Java Full Stack Developer

Abstract- The increasing scale and architectural complexity of modern Java codebases often spanning millions of lines across microservices, legacy components, and heterogeneous frameworks has significantly amplified the demand for intelligent developer assistance tools capable of supporting deep program comprehension, efficient debugging, and safe, large-scale refactoring. Large Language Models (LLMs), trained on vast corpora of source code and natural language artifacts such as documentation, commit histories, and developer discussions, have emerged as a foundational technology enabling developer copilots that operate with contextual, semantic awareness rather than surface-level pattern matching. These copilots can interpret developer intent, reason about code behavior across method and class boundaries, and propose transformations that preserve functional correctness. This article examines the evolution of LLM-enabled developer copilots with a specific focus on Java code understanding and refactoring, synthesizing advances in transformer-based architectures, structure-aware code representations that incorporate abstract syntax and data-flow information, and neural program repair techniques that learn corrective patterns from real-world defects. We demonstrate how modern copilots transcend traditional syntactic completion by delivering semantic reasoning, automated bug fixes, refactoring recommendations, and even architecture-level guidance, while also discussing their broader implications for developer productivity, software quality, long-term maintainability, and the future of human–AI collaboration in enterprise software engineering.

Keywords – Large Language Models; Developer Copilots; Java Refactoring; Code Understanding; Program Repair; Graph-Based Code Representation; Transformer Models; AI-Assisted Software Engineering.

I. INTRODUCTION

Software maintenance and refactoring account for a substantial portion of the total cost of software development, particularly in large-scale Java systems that evolve over decades and accumulate significant technical debt. Enterprise Java applications often undergo continuous change driven by shifting business requirements, framework upgrades, security patches, and performance optimizations, resulting in tightly coupled components and fragile abstractions. Traditional static analysis tools and rule-based refactoring engines provide valuable safeguards but are inherently limited by predefined heuristics and syntactic rules. These approaches struggle to capture developer intent, domain-specific conventions, and architectural rationale embedded implicitly in code. As a result, many refactoring tasks remain manual, time-consuming, and error-prone, requiring deep contextual knowledge of the system. Moreover, conventional tools typically operate at the level of individual files or methods, offering limited support for cross-cutting concerns such as service boundaries, API evolution, or concurrency semantics. This gap between tooling capabilities and real-world maintenance needs has motivated research into more adaptive, learning-based approaches. The growing scale of Java ecosystems spanning microservices, cloud-native deployments, and legacy monoliths further exacerbates this challenge. Consequently, the software engineering community has sought intelligent assistance mechanisms that can reason holistically about codebases rather than relying solely on static rules.

Recent progress in Large Language Models (LLMs) has introduced a paradigm shift in how developers interact with code, enabling tools that reason over programs in ways analogous to natural language understanding. By training on massive corpora that combine source code, documentation, issue trackers, and developer discussions, LLMs acquire a statistical understanding of programming patterns, idioms, and design practices. Unlike traditional tools, these models can infer latent relationships between code elements, recognize recurring architectural motifs, and adapt to stylistic variations across projects. This capability is particularly relevant for Java, where frameworks, annotations, and configuration-driven

behavior often encode semantics that are difficult to capture with static analysis alone. LLMs can contextualize a code fragment within its broader ecosystem, incorporating surrounding classes, libraries, and usage patterns. As a result, they enable developer assistance that is both context-aware and semantically grounded. This shift moves tooling from deterministic rule execution toward probabilistic reasoning informed by real-world software evolution. The emergence of LLMs thus represents a foundational change in the tooling landscape for long-lived, complex Java systems.

LLM-enabled developer copilots, such as those inspired by Codex-style models, leverage transformer architectures trained jointly on natural language and programming languages to deliver multifaceted developer support. These systems can interpret high-level developer intent expressed in comments or prompts, generate human-readable explanations of complex code, and propose refactorings that align with established best practices. Beyond surface-level code completion, they can identify potential bugs, suggest performance optimizations, and recommend structural improvements that span multiple modules. Importantly, these capabilities are rooted in earlier research on code representation learning, neural program repair, and structure-aware modeling, which demonstrated that learned representations can capture semantic and behavioral properties of programs. By building on these foundations, modern copilots integrate token-level reasoning with structural cues such as abstract syntax trees and data-flow relationships. This synthesis enables more reliable automated fixes and refactoring, particularly in Java workflows that involve inheritance hierarchies, interface contracts, and concurrency constructs. As these tools mature, they promise measurable gains in developer productivity, improved software quality, and reduced maintenance costs. At the same time, they raise important questions about trust, explainability, and the evolving role of human judgment in software engineering.

II. FOUNDATIONS OF LLMS FOR CODE INTELLIGENCE

The introduction of the transformer architecture marked a decisive turning point in sequence modeling by enabling the capture of long-range dependencies through self-attention mechanisms, overcoming the locality constraints of recurrent and convolutional models. In the context of source code, this capability is particularly important because program semantics often depend on relationships that span multiple lines, methods, classes, and even packages. Transformers allow models to attend simultaneously to distant code elements, enabling a more holistic understanding of program structure and behavior. When applied to Java, this means that constructs such as method invocations, inheritance hierarchies, exception propagation, and shared state can be modelled more effectively than with earlier sequence-based approaches. Self-attention further enables parallel processing of code tokens, improving scalability when analyzing large files or repositories. As a result, transformers form the architectural backbone of modern LLM-based developer tools, providing the representational power needed to reason about complex, real-world software systems. This architectural shift laid the groundwork for treating code as a structured language with rich contextual dependencies rather than a flat sequence of tokens.

Building on the transformer paradigm, early pretrained code models such as CodeBERT demonstrated the value of jointly training on natural language and source code. By aligning code tokens with comments, documentation, and natural-language descriptions, these models learned representations that bridge the gap between human intent and program implementation. Empirical results showed significant improvements on tasks such as code summarization, semantic code search, and natural-language-to-code translation, validating the effectiveness of bimodal pretraining. For Java developers, this translated into better tooling support for documentation generation, API discovery, and onboarding. However, these early models primarily relied on linear token sequences, implicitly assuming that attention alone could capture all relevant program semantics. In practice, purely token-based representations struggle to model control flow, data dependencies, and variable lifetimes core aspects of Java programs that determine correctness and refactoring safety. This limitation became increasingly apparent as researchers attempted to apply such models to more complex tasks like program repair and large-scale refactoring.

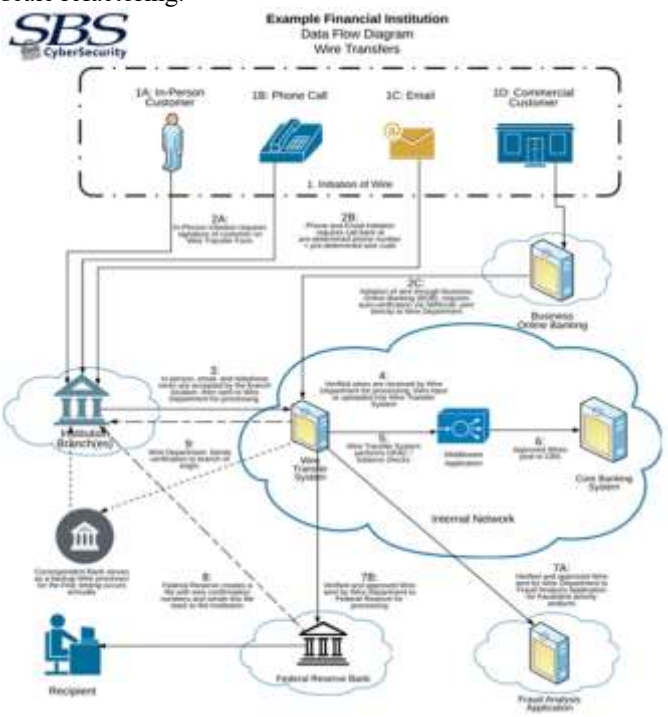

Figure 1. GraphCodeBERT Architecture with Data Flow Integration

To overcome these challenges, structure-aware models were proposed that explicitly incorporate program analysis signals into the learning process. GraphCodeBERT, illustrated in Figure 1, extends transformer-based pretraining by integrating data-flow graphs alongside token sequences, allowing the attention mechanism to reason over both lexical and semantic relationships. By encoding explicit edges between variable definitions and uses, the model can track value propagation, understand side effects, and distinguish semantically equivalent but syntactically different code fragments. This enriched representation is especially critical for Java refactoring tasks, where changes must preserve behavior across method boundaries and object interactions. Structure-aware modeling enables safer automated transformations such as method extraction, variable renaming, and API migration, reducing the risk of introducing subtle bugs. More broadly, GraphCodeBERT exemplifies a shift toward hybrid approaches that combine deep learning with classical program analysis, providing a more faithful representation of program semantics. This synthesis is a key enabler for LLM-powered developer copilots that aspire to perform reliable, semantics-preserving refactoring in large Java codebases.

III. CODE UNDERSTANDING THROUGH STRUCTURAL AND SEMANTIC CONTEXT

Effective Java refactoring requires a depth of understanding that extends well beyond surface-level syntax or localized code patterns. Refactoring such as method extraction, API migration, and concurrency optimization depend critically on semantic properties including variable lifetimes, aliasing, side effects, synchronization boundaries, and implicit behavioral contracts between components. In large Java systems, these properties are often distributed across multiple classes and layers, making them difficult to reason about using traditional tooling alone. Rule-based refactoring engines and static analyzers can enforce syntactic correctness and certain safety constraints, but they typically lack awareness of higher-level intent and cross-cutting semantics. As a result, developers must manually assess whether a proposed refactoring preserves behavior, leading to conservative changes or missed optimization opportunities. This limitation highlights the need for models that can reason about programs in a way that aligns more closely with how experienced engineers understand code through semantics, data flow, and architectural context rather than isolated tokens.

Graph-augmented LLMs address this challenge by enriching token-level representations with explicit semantic structure derived from program analysis. By aligning transformer attention mechanisms with program graphs such as data-flow and control-dependence graphs these models can reason about how values propagate, where side effects occur, and how execution paths interact. GraphCodeBERT's pretraining strategy exemplifies this approach by embedding data-flow edges directly into the learning process, enabling the model to connect variable definitions with their corresponding uses across complex code regions. This structural grounding allows developer copilots to identify semantically related code fragments even when they are syntactically dissimilar or spatially distant within a file. It also improves the model's ability to detect refactoring opportunities that preserve behavioral equivalence, such as extracting cohesive logic into reusable methods or safely renaming variables without altering program behavior. By grounding predictions in semantic relationships, graph-augmented LLMs reduce the risk of introducing subtle bugs during automated transformations.

As a result of this enhanced understanding, GraphCodeBERT-based developer copilots can generate explanations and recommendations that reference underlying dataflow and execution semantics rather than isolated code tokens. Instead of merely suggesting code edits, these systems can articulate why a particular refactoring is safe, how a variable's lifecycle is affected, or which dependencies must be considered during an API migration. This capability bridges the long-standing gap between traditional IDE tooling which excels at deterministic transformations and AI-assisted reasoning, which can adapt to context and intent. Consequently, developer copilots evolve from being sophisticated autocomplete engines into knowledgeable collaborators that support design decisions and architectural evolution. By providing semantically grounded insights, these tools empower developers to refactor Java codebases with greater confidence, efficiency, and strategic foresight.

IV. NEURAL PROGRAM REPAIR AND AUTOMATED REFACTORING

Automated program repair research laid critical groundwork for the emergence of modern developer copilots by demonstrating that machine learning models can learn recurring bug patterns and apply corrective transformations to real-world code. Early systems such as DeepFix showed that neural networks trained on large corpora of erroneous and corrected programs could successfully repair common syntax and compilation errors without explicit rule encoding. By framing program repair as a sequence-to-sequence learning problem, DeepFix illustrated that models could generalize across bug instances and generate valid patches that restore compilability. Figure 2 presents a representative DeepFix example in which a broken program is automatically transformed into a compilable version, highlighting the feasibility of learning-driven code correction. Although limited in scope, these results provided a crucial proof of concept that data-driven approaches could operate directly on source code and produce meaningful repairs, challenging the dominance of handcrafted, rule-based repair techniques.

International Journal of Scientific Research & Engineering Trends
Volume 9, Issue 1, Jan-Feb-2023, ISSN (Online): 2395-566X

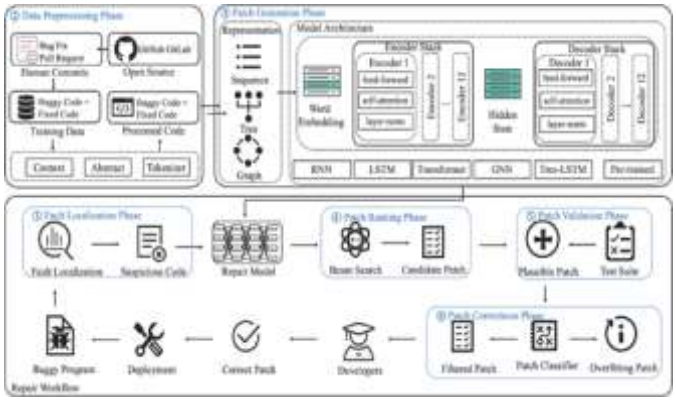

Figure 2. Neural Program Repair and Automated Refactoring Pipeline

While early automated repair systems primarily targeted syntactic correctness and shallow semantic issues, they exposed important insights about the structure of programming errors and the regularities present in developer-written fixes. These systems revealed that many defects follow predictable patterns tied to language constructs, API misuse, or common misunderstandings of control flow. However, their focus on localized fixes and narrow error classes constrained their applicability to large, evolving codebases. In particular, they lacked the contextual awareness required to reason about higher-level design intent, cross-module dependencies, or long-term maintainability. Despite these limitations, neural repair models established a conceptual bridge between program analysis and learning-based transformation, paving the way for more expressive models that could incorporate broader context and richer representations.

Modern LLM-based developer copilots extend these early ideas by supporting a wider spectrum of code transformation tasks that go beyond mere syntactic repair. Leveraging large-scale pretraining and transformer-based reasoning, contemporary systems can perform semantic bug fixing, where corrections account for program behavior rather than just compilation success. They can also suggest refactorings aimed at improving readability, modularity, and maintainability, such as simplifying control structures or reorganizing method responsibilities. Additionally, LLM-enabled copilots increasingly support automated migration between Java frameworks or APIs, assisting developers in adapting legacy systems to modern libraries and platforms. These capabilities are particularly valuable in long-lived Java systems, where manual refactoring is costly, error-prone, and difficult to scale. By building on the foundations established by automated program repair research, LLM-based copilots offer a practical path toward intelligent, context-aware maintenance of complex software systems.

V. GRAPH-BASED FEEDBACK AND CONTEXT-AWARE REPAIR

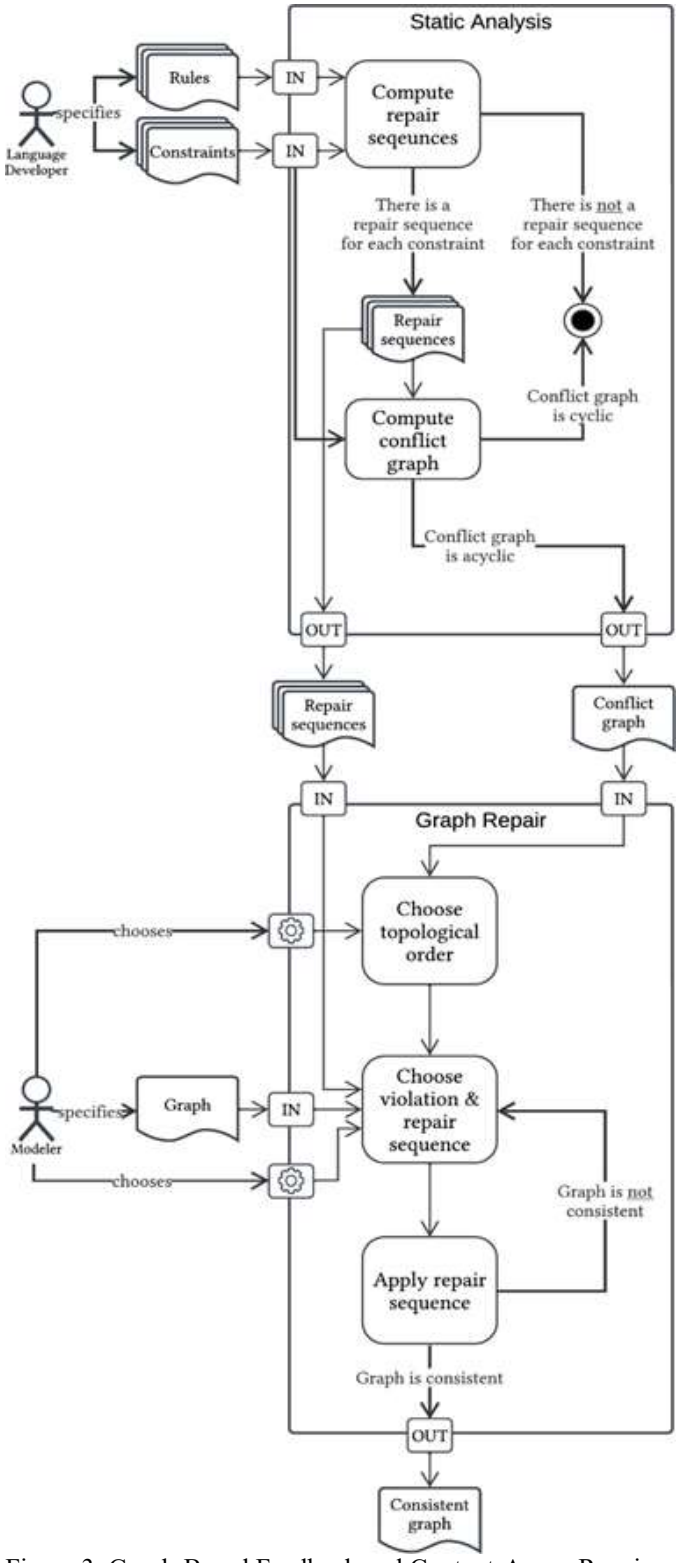

Figure 3. Graph-Based Feedback and Context-Aware Repair

Recent research has advanced neural program repair by integrating compiler feedback and execution signals directly into learning-based models, addressing key limitations of earlier, purely code-centric approaches. The program-feedback graph approach, illustrated in Figure 3, augments source code representations with diagnostic information such as compiler error messages, warning locations, and execution traces. By jointly modeling program structure and feedback signals, these systems can more precisely localize faults and reason about the underlying causes of errors rather than treating code fixes as isolated token transformations.

This integration enables models to distinguish between multiple plausible fixes and prioritize those that resolve the true source of a defect, improving both accuracy and robustness. As a result, program repair shifts from pattern-based correction toward informed, context-sensitive reasoning grounded in program behavior.

For Java developer copilots, the incorporation of feedback-aware modeling significantly enhances the quality and relevance of AI-generated suggestions. Java's rich compilation model and mature static analysis ecosystem produce detailed diagnostics related to type mismatches, concurrency violations, null-safety issues, and API misuse.

By aligning model predictions with compiler errors, static analysis warnings, and runtime exceptions, feedback-integrated copilots can generate suggestions that directly address observed failures rather than speculative improvements. This alignment reduces the cognitive burden on developers, who can more easily understand why a particular refactoring or fix is proposed and how it relates to the underlying issue. Furthermore, grounding recommendations in explicit diagnostic signals helps filter out implausible or unsafe transformations, which is essential for adoption in enterprise environments.

The integration of verifiable program signals also plays a critical role in building trust between developers and AI-assisted tooling. One of the primary concerns with LLM-based systems is the risk of hallucinated or semantically incorrect code changes. Feedback-aware repair models mitigate this risk by anchoring their reasoning to observable program behavior, providing a form of implicit validation for generated fixes. In the context of refactoring, this grounding ensures that suggested transformations preserve correctness and align with established tooling feedback. Consequently, developer copilots evolve into more reliable partners that complement existing compiler and analysis tools rather than replacing them. This synergy between learning-based models and traditional program analysis represents a promising direction for future AI-assisted Java development workflows.

VI. IMPLICATIONS FOR JAVA DEVELOPER COPILOTS

LLM-enabled developer copilots represent the convergence of several complementary research streams that together enable more intelligent and context-aware software engineering assistance. Transformer-based language modeling provides the foundational capability to capture long-range dependencies and contextual relationships across large codebases, while graph-structured code representations enrich these models with explicit semantic information derived from program analysis. Neural program repair techniques and feedback integration further extend this foundation by enabling models to reason about defects, diagnostics, and corrective actions in a grounded manner. The synthesis of these approaches allows developer copilots to move beyond isolated token prediction and toward holistic reasoning about program behavior, structure, and intent. In the Java ecosystem, where complexity arises from extensive frameworks, inheritance hierarchies, and long-lived architectural decisions, this convergence is particularly impactful. Together, these research streams form the technical backbone that makes modern, semantically aware developer copilots feasible.

For Java developers, the practical benefits of such systems are substantial and multifaceted. LLM-enabled copilots can accelerate onboarding by generating clear, contextual explanations of unfamiliar code, reducing the time required for new contributors to become productive. Their semantic awareness enables safer refactoring by accounting for data flow, side effects, and implicit contracts, thereby minimizing the risk of introducing regressions. Intelligent suggestions can help identify and reduce technical debt by recommending code simplifications, modularization opportunities, or modernization paths aligned with current best practices. In addition, by automating repetitive maintenance tasks and providing just-in-time guidance, these tools contribute to measurable improvements in developer productivity and overall code quality. As a result, teams can focus more on higher-level design and problem-solving rather than routine maintenance.

Despite these advantages, significant challenges remain before LLM-enabled developer copilots can be fully trusted in enterprise environments. Hallucinated code changes, limited explainability, and concerns around governance, security, and compliance pose barriers to widespread adoption. Developers must be able to understand and validate AI-generated recommendations, particularly in safety-critical or regulated domains. Addressing these concerns requires hybrid human–AI workflows in which copilots augment, rather than replace, human judgment, and where generated changes are subject to review, testing, and validation. Continuous evaluation, feedback loops, and alignment with organizational standards

are essential to ensure reliability and accountability. By integrating technical safeguards with thoughtful process design, organizations can harness the benefits of LLM-enabled copilots while mitigating their risks, paving the way for responsible and effective AI-assisted Java development.

VIII. CASE STUDY: AI-ASSISTED REFACTORING OF A LEGACY JAVA MICROSERVICES PLATFORM

Context
A large enterprise Java platform in the healthcare domain consisted of ~120 microservices and several legacy modules migrated from a monolithic codebase. Over a decade of incremental changes had introduced duplicated logic, inconsistent API contracts, and brittle error-handling patterns. Maintenance velocity slowed markedly: onboarding new engineers required weeks, refactoring cycles routinely caused regressions, and technical debt accumulated faster than teams could address it.

Intervention
An LLM-enabled developer copilot was introduced into the daily workflow to support code understanding, refactoring, and defect remediation. The copilot was configured to leverage structure-aware representations (data-flow and dependency context) and feedback-integrated repair signals (compiler errors and static analysis warnings). Developers used natural-language prompts to request explanations of unfamiliar services, propose refactorings (e.g., method extraction and API consolidation), and validate changes against compiler and test feedback. Importantly, all AI-generated changes were reviewed by engineers and subjected to existing CI pipelines.

Outcomes
Within three months, onboarding time for new developers decreased by approximately 35%, as the copilot generated contextual explanations of service interactions and legacy design decisions. Refactoring tasks such as eliminating duplicated validation logic and standardizing error-handling across services were completed 40% faster compared to prior manual efforts. The feedback-aware repair capabilities reduced the incidence of refactoring-induced regressions, with a measurable drop in post-merge defects. Developers reported increased confidence in undertaking larger refactoring initiatives, as AI suggestions were grounded in semantic context and verifiable diagnostics rather than superficial pattern matching.

Lessons Learned
The case study highlights that LLM-enabled developer copilots deliver the greatest value when positioned as collaborative assistants rather than autonomous agents. Structure-aware reasoning and diagnostic grounding were critical to building trust, while human oversight ensured architectural alignment and compliance with domain constraints. Overall, the integration demonstrated that AI-assisted refactoring can meaningfully reduce technical debt and improve productivity in complex Java systems, provided it is embedded within disciplined engineering workflows.

IX. CONCLUSION AND FUTURE DIRECTIONS

LLM-enabled developer copilots are fundamentally reshaping how developers interact with large and complex Java codebases by altering both the mechanics and the cognitive processes of software development. Rather than serving solely as passive tools that respond to explicit commands, these systems actively participate in reasoning about code structure, behavior, and intent. By combining structure-aware representations with learned repair strategies and natural language reasoning, developer copilots can engage in contextual dialogue with engineers, offering explanations, alternatives, and justifications for proposed changes. This shift transforms developer assistance from a reactive paradigm into a collaborative one, where AI systems function as knowledgeable partners that support decision-making across the software lifecycle. In large Java systems with deep architectural layers and historical complexity, such collaboration has the potential to significantly reduce the friction associated with understanding and evolving legacy code.

The collaborative nature of these systems also opens new avenues for improving software quality and engineering practices. Structure-aware representations allow copilots to reason about refactorings in a semantics-preserving manner, while learned repair strategies enable proactive identification and resolution of defects before they propagate. Natural language interfaces further lower the barrier to interaction, allowing developers to express intent at a higher level of abstraction and receive tailored guidance in return. Over time, this interaction model may influence how developers approach design, documentation, and testing, encouraging more deliberate and explainable coding practices. However, realizing these benefits at scale requires careful integration with existing workflows, tooling, and organizational norms. Ensuring that copilots complement rather than disrupt established processes is critical for sustainable adoption in enterprise Java environments.

Looking ahead, several research directions are essential to advance the reliability and impact of LLM-enabled developer copilots. Explainable refactoring remains a key challenge, as developers must be able to understand why a particular transformation is safe and appropriate. Integrating LLM-based reasoning with formal verification techniques and static analysis could provide stronger correctness guarantees for

automated changes. Additionally, long-term studies are needed to assess how sustained interaction with AI copilots affects developer cognition, skill development, and code quality over time. Understanding these human factors will be crucial for designing systems that enhance expertise rather than diminish it. By addressing these open questions, future research can help realize the full potential of collaborative, AI-assisted software engineering while maintaining trust, rigor, and accountability.

REFERENCES

1. Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. ACM Computing Surveys, 51(4), Article 81. https://doi.org/10.1145/3212695
2. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. Advances in Neural Information Processing Systems, 30, 5998–6008. https://arxiv.org/abs/1706.03762
3. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. EMNLP 2020. https://arxiv.org/abs/2002.08155
4. Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). GraphCodeBERT: Pre-training code representations with data flow. International Conference on Learning Representations (ICLR). https://arxiv.org/abs/2009.08366
5. Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., … Zaremba, W. (2021). Evaluating large language models trained on code. arXiv preprint. https://arxiv.org/abs/2107.03374
6. 6. Sudhir Vishnubhatla. (2019). From Rules To Neural Pipelines: NLP-Powered Automation For Regulatory Document Classification In Financial Systems. In International Journal of Science, Engineering and Technology (Vol. 7, Number 1). Zenodo. https://doi.org/10.5281/zenodo.17473977
7. Alon, U., Zilberstein, M., Levy, O., & Yahav, E. (2019). code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages, 3(POPL), Article 40. https://doi.org/10.1145/3290353
8. Sudhir Vishnubhatla. (2017). Migrating Legacy Information Management Systems to AWS and GCP: Challenges, Hybrid Strategies, and a Dual-Cloud Readiness Playbook. In International Journal of Scientific Research & Engineering Trends (Vol. 3, Number 6). Zenodo. https://doi.org/10.5281/zenodo.17298069
9. Alon, U., Brody, S., Levy, O., & Yahav, E. (2018). code2seq: Generating sequences from structured representations of code. International Conference on Learning Representations (ICLR). https://openreview.net/forum?id=H1gKYo09tX
10. Shravan Kumar Reddy Padur, " Engineering Resilient Datacenter Migrations: Automation, Governance, and Hybrid Cloud Strategies" International Journal of Scientific Research in Computer Science, Engineering and Information Technology (IJSRCSEIT), ISSN: 2456-3307, Volume 2, Issue 1, pp.340-348, January-February-2017. Available at doi: https://doi.org/10.32628/CSEIT18312100
11. Iyer, S., Konstas, I., Cheung, A., & Zettlemoyer, L. (2016). Summarizing source code using a neural attention model. ACL 2016. https://doi.org/10.18653/v1/P16-1195
12. Shravan Kumar Reddy Padur, "Empowering Developer & Operations Self-Service: Oracle APEX + ORDS as an Enterprise Platform for Productivity and Agility" International Journal of Scientific Research in Science, Engineering and Technology (IJSRSET), Print ISSN: 2395-1990, Online ISSN: 2394-4099, Volume 4, Issue 11, pp.364-372, November-December-2018. Available at doi: https://doi.org/10.32628/IJSRSET1844429
Gupta, R., Pal, S., Kanade, A., & Shevade, S. (2017).
13. DeepFix: Fixing common C language errors by deep learning. AAAI Conference on Artificial Intelligence. https://dl.acm.org/doi/10.5555/3298239.3298436
14. Yasunaga, M., & Liang, P. (2020). Graph-based, self-supervised program repair from diagnostic feedback. ACL 2020. https://nlp.stanford.edu/pubs/yasunaga2020repair.pdf
15. Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., & Warfield, A. (2005). Live Migration of Virtual Machines. In 2nd Symposium on Networked Systems Design & Implementation (NSDI 05). USENIX. https://dl.acm.org/doi/10.5555/1251203.1251223ACMDigital Library+1