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

# Prompt-Oriented Code Understanding: Towards Natural Language-Driven Debugging in IDEs

Mohan Siva Krishna Konakanchi mohansivakrishna16@gmail.com

***Abstract*—The increasing complexity of modern software systems has rendered traditional debugging methods, such as manual code inspection and breakpoint analysis, progressively inefficient. This paper introduces Prompt-Oriented Code Understanding (POCU), a novel paradigm that leverages Generative Artificial Intelligence (GenAI) to enable natural language-driven debugging directly within Integrated Development Environments (IDEs). We propose a system architecture that translates developer queries in natural language into actionable code analysis and debugging operations. To facilitate continuous model improvement without compromising intellectual property, we introduce a Trust-Metric Federated Learning (TMFL) framework. TMFL allows the underlying GenAI model to be fine-tuned across disparate, private codebases (silos) while ensuring the integrity and accountability of contributions through a novel trust metric. Furthermore, acknowledging the critical need for transparency in AI-assisted tools, we present a framework to quantify and optimize the inherent trade-off between the explainability of the AI's suggestions and its raw performance. We define metrics for both dimensions and formulate an optimization strategy to achieve a Pareto-optimal balance. Our conceptual framework and proposed methodologies lay the groundwork for a new generation of intelligent, intuitive, and trustworthy developer tools designed to significantly reduce debugging time and cognitive load.**

***Index Terms*—Generative AI, Code Understanding, Natural Language Processing, Software Debugging, Federated Learning, Explainable AI, IDE Integration**

## I. Introduction

Software development is an intricate process where debugging constitutes a significant portion of the development lifecycle, often consuming up to 50% of a developer's time [1]. Traditional debugging techniques rely heavily on developers forming hypotheses about code behavior and manually verifying them using tools like debuggers, log analyzers, and static analysis tools. This process is labor-intensive, requires deep expertise in the codebase, and scales poorly with the size and complexity of modern software projects, especially in distributed and microservices-based architectures.

The recent proliferation of Large Language Models (LLMs) and Generative AI (GenAI) has opened new frontiers in human-computer interaction, particularly in specialized domains like software engineering [2], [3]. These models have demonstrated remarkable capabilities in code generation, completion, and summarization. However, their application to the nuanced and context-heavy task of debugging remains an open and challenging research area. The primary challenge lies in bridging the gap between high-level, often ambiguous, developer intent expressed in natural language and the low-level, precise operations required for effective code analysis and fault localization.

This work was not supported by any organization.

This paper introduces **Prompt-Oriented Code Understanding (POCU)**, a novel approach that recasts debugging as a conversational, query-driven process. The core idea is to empower developers to interact with their codebase using natural language prompts directly within their IDE. For instance, a developer could ask, "Why is the user session object nullifying after the payment gateway callback?" or "Trace the execution path for request ID 'xyz-123' and highlight any potential race conditions." The POCU system would then parse this query, gather relevant context from the code, runtime data, and version control history, and provide a synthesized, human-readable explanation along with actionable suggestions.

However, building such a system presents three fundamental challenges that this paper aims to address:

1) **Contextual Intelligence and Integration:** A generic GenAI model lacks the specific context of a proprietary codebase. An effective debugging assistant must be deeply integrated with the IDE and be aware of the project's static structure, dynamic runtime behavior, and historical evolution.
2) **Data Privacy and Collaborative Learning:** To improve its accuracy, the model must learn from a wide array of codebases. However, organizations are unwilling to share their proprietary source code for centralized training. A decentralized learning paradigm is required that preserves data privacy while enabling collaborative model improvement.
3) **Trust and Transparency:** Developers will not rely on a "black box" tool for critical debugging tasks. The system's suggestions must be explainable, allowing developers to understand the reasoning behind a particular conclusion. There is an inherent trade-off between the model's performance (e.g., accuracy) and the explainability of its outputs.

To address these challenges, we propose a multi-faceted solution. First, we detail the architecture of the POCU system, which integrates a GenAI core with static and dynamic code analysis tools via a context-aware middleware. Second, we introduce a **Trust-Metric Federated Learning (TMFL)** framework. TMFL enables distributed fine-tuning of the core GenAI model across organizational silos. It incorporates a novel trust metric that weighs model updates based on factors like data quality, contribution consistency, and security posture, preventing malicious or low-quality updates from corrupting the global model. Third, we propose a formal framework to manage the **Explainability-Performance Optimization (EPO)** trade-off, enabling organizations to configure the system to their desired balance between insightful explanations and predictive accuracy.

The contributions of this paper are therefore threefold:

- A novel system architecture for Prompt-Oriented Code Understanding (POCU) that facilitates natural language-driven debugging in IDEs.
- A Trust-Metric Federated Learning (TMFL) framework designed for secure, privacy-preserving, and accountable collaborative training of code-understanding models.
- A formal methodology for quantifying and optimizing the trade-off between model explainability and performance in the context of AI-assisted software engineering.

This paper is organized as follows: Section II reviews related work. Section III presents the detailed methodology of the POCU system, the TMFL framework, and the EPO model. Section IV outlines a proposed experimental design for evaluating the system. Section V discusses potential results and their implications. Finally, Section VI concludes the paper and suggests avenues for future research.

## II. Related Work

The application of AI to software engineering, often termed AISE, has a rich history. Our work builds upon several key areas: code analysis, natural language processing for source code, and federated learning.

### *A. AI in Code Analysis and Debugging*

Automated program repair (APR) and fault localization have been long-standing goals in software engineering research. Early techniques relied on statistical methods and program slicing [4]. More recently, machine learning-based approaches have gained prominence. Systems like DeepBugs [5] used deep learning to detect bugs based on code patterns. Other research has focused on mining software repositories to learn common bug-fix patterns [6]. While powerful, these systems are typically specialized for specific bug classes and lack the interactive, general-purpose query capabilities of our proposed POCU system.

The advent of large-scale, pre-trained models like Codex [3] and AlphaCode [7] has revolutionized the field. These models excel at code generation and have been integrated into tools like GitHub Copilot. While these tools assist in ”forward-engineering” (writing new code), our work focuses on ”reverse-engineering” and diagnostics (understanding and debugging existing code), which requires a deeper level of contextual inference.

### *B. Natural Language Processing on Source Code*

The idea of treating source code as a natural language (the ”naturalness hypothesis”) has been influential [8]. This has led to the development of models that can perform tasks like code summarization, documentation generation, and code search using natural language queries. Models like CodeBERT [9] and GraphCodeBERT [10] use bimodal pre-training on parallel corpora of code and natural language text to learn rich representations. Our POCU system leverages such representations but extends them by integrating runtime context and interactive dialogue capabilities, moving from static code representation to dynamic execution understanding.

### *C. Federated Learning for Privacy-Preserving AI*

Federated Learning (FL) was introduced by Google to train models on decentralized data, such as on mobile devices, without centralizing the data itself [11]. It has become a cornerstone of privacy-preserving machine learning. In the software engineering domain, FL has been proposed for tasks like defect prediction across different organizations [12]. However, standard FL algorithms like FedAvg are vulnerable to non-IID data distributions and potential adversarial attacks from malicious clients [13]. Our TMFL framework addresses these vulnerabilities by introducing an explicit trust mechanism to vet and weight client contributions, making the collaborative learning process more robust and accountable, which is critical when dealing with valuable intellectual property like source code.

### *D. Explainable AI (XAI)*

As AI models become more complex, the need for explainability has grown. Techniques like LIME [14] and SHAP [15] provide post-hoc explanations for model predictions. Applying XAI to code-related tasks is an emerging field. The goal is to explain why a model flagged a certain piece of code as buggy or suggested a particular fix. Our EPO framework contributes to this area by moving beyond simply providing explanations to actively managing the trade-off between the quality of these explanations and the model’s core performance metrics, which has not been formally addressed in the context of AI-driven debugging tools.

## III. Methodology

This section details the three core components of our proposed solution: the POCU system architecture, the TMFL framework for collaborative training, and the EPO model for balancing explainability and performance.

### *A. Prompt-Oriented Code Understanding (POCU) System Architecture*

The POCU system is designed as an IDE plugin that mediates the interaction between the developer, the codebase, and a powerful GenAI core. Its architecture consists of four main layers.

*1) IDE Integration and Context Gathering:* This layer acts as the primary interface. It includes a chat-like UI within the IDE where developers can issue natural language prompts. Crucially, this layer is responsible for automatically gathering relevant context for each query. The context vector, $C_v$, is composed of:

- **Static Context ($C_s$):** The Abstract Syntax Tree (AST) of the current file and related files, the call graph, dependency information, and version control history (e.g., 'git blame' output).
- **Dynamic Context ($C_d$):** If the application is running in a debug session, this includes the current call stack, variable values, thread states, and recent application logs.
- **Query Context ($C_q$):** The developer's natural language prompt and the recent history of the conversation to maintain dialogue coherence.

The final context-enriched prompt, $P_{rich}$, is formulated as $P_{rich} = f(C_s, C_d, C_q)$, where $f$ is a serialization function that combines these disparate data sources into a format suitable for the GenAI core.

*2) Natural Language Processing (NLP) Front-end:* This component is responsible for prompt pre-processing and intent recognition. It parses the developer's query, $C_q$, to identify the core intent (e.g., 'find bug', 'explain code', 'trace execution'). It uses Named Entity Recognition (NER) to identify key code artifacts mentioned in the query (e.g., function names, variable names). This structured representation of the query allows the system to trigger more specific analysis tasks.

*3) GenAI Core:* The heart of the system is a large language model, pre-trained on a massive corpus of open-source code and natural language text (e.g., a foundation model similar to GPT-4 or PaLM 2). This model is then fine-tuned using the TMFL framework described below. The GenAI core receives the context-enriched prompt $P_{rich}$ and generates a response. Its task is not just to generate text but to synthesize information from the provided context to form a coherent, accurate, and helpful explanation or suggestion.

*4) Actionable Analysis Engine:* The raw output of the GenAI model might be a high-level hypothesis. The Actionable Analysis Engine translates this hypothesis into concrete operations. For example, if the model suggests, "The issue might be a race condition in the 'updateCache' function," this engine could automatically instrument the code to add more logging around that function, or suggest specific breakpoints to the developer. This feedback loop, where the model's output triggers further automated analysis, is a key feature of the POCU system.

### B. Trust-Metric Federated Learning (TMFL) Framework

To enable the GenAI core to learn from proprietary codebases without centralized data sharing, we propose the TMFL framework. It extends the standard Federated Averaging (FedAvg) algorithm by incorporating a trust metric for each participating client (e.g., an organization).

Let $N$ be the number of participating clients. In each communication round $t$, the process is as follows:

1) The central server distributes the current global model, $w_t$, to a subset of clients.
2) Each client $k$ fine-tunes the model on its local, private data $D_k$ to produce a local model update, $w^k_{t+1}$
3) Each client sends its proposed update $w^k_{t+1}$ back to the server.
4) The server calculates a trust score, $\tau^t_k \in [0, 1]$, for each client $k$.
5) The server aggregates the local updates to form the new global model, $w_{t+1}$, using a weighted average where the weights are a function of the trust scores.

The standard FedAvg update rule is:

$$w_{t+1} = \sum_{k=1}^{N} \frac{n_k}{n} w^k_{t+1}$$

where $n_k = |D_k|$ is the size of the local dataset and $n = \sum_k n_k$.

Our proposed TMFL update rule modifies this to:

$$w_{t+1} = \sum_{k=1}^{N} \frac{\alpha_k n_k}{\sum_{j=1}^{N} \alpha_j n_j} w^k_{t+1}$$

where $\alpha_k$ is the normalized trust weight for client $k$, derived from its trust score $\tau^t_k$.

*1) Defining the Trust Metric $\tau^t_k$:* The trust score $\tau^t_k$ for client $k$ at round $t$ is a composite metric calculated as:

$$\tau^t_k = \lambda_1 Q^t_k + \lambda_2 C^t_k + \lambda_3 S^t_k$$

where $\lambda_1 + \lambda_2 + \lambda_3 = 1$ are weighting hyper-parameters. The components are:

- **Quality Score ($Q^t_k$):** Measures the quality of the client's update. This is assessed by the server using a small, held-out validation set. The score is proportional to the performance improvement the client's update provides on this set.
- **Consistency Score ($C^t_k$):** Measures the similarity of a client's update to the global update trend. An update that is a significant outlier might be malicious or simply the result of a highly non-IID data distribution. This is calculated based on the cosine similarity between the client's update vector ($\Delta w^k = w^k_{t+1} - w_t$) and the mean update vector of the previous round.
- **Security Score ($S^t_k$):** An externally provided score representing the client's security posture. This could be based on audits, certifications, or historical data on security incidents. This helps prevent contributions from known bad actors.

This trust metric ensures that clients who contribute high-quality, consistent, and secure updates have a greater influence on the global model, making the federated ecosystem more robust and accountable.

### C. Explainability-Performance Optimization (EPO) Framework

Developers require transparent reasoning, not just correct answers. The EPO framework addresses the trade-off between the model's performance and the explainability of its outputs.

*1) Quantifying Explainability and Performance:* We first define metrics for each dimension.

- **Performance (P):** This is a task-dependent metric. For bug detection, it could be the F1 score. For code explanation, it could be measured by ROUGE or BLEU scores against human-written explanations. We define it as a singular metric $P \in [0, 1]$.
- **Explainability (E ):** This is more challenging to quantify. We propose a composite metric:

$$E = \theta_1 C + \theta_2 F + \theta_3 I$$

where $\theta_1 + \theta_2 + \theta_3 = 1$.
  - **Clarity (C):** The readability and simplicity of the generated explanation, measured using standard readability scores (e.g., Flesch-Kincaid).
  - **Fidelity (F):** How accurately the explanation reflects the model's internal reasoning. This can be approximated by measuring how the model's output changes when features (e.g., lines of code) identified as important by the explanation are perturbed, similar to the logic of LIME [14].
  - **Identifiability (I):** The extent to which the explanation points to specific, verifiable artifacts in the code (e.g., line numbers, variable names). An explanation like "There's a null pointer exception on line 42 because 'user.session' is null" has higher identifiability than "There might be a state issue."

*2) Formulating the Optimization Problem:* The goal is to find a model configuration $\vartheta$ from a set of possible configurations $\Theta$ that maximizes both performance and explainability. This is a multi-objective optimization problem. We aim to identify the Pareto frontier of configurations. A configuration $\vartheta_1$ Pareto-dominates $\vartheta_2$ if it is at least as good on all objectives and strictly better on at least one.

$$\text{maximize} \quad (P(\vartheta), E(\vartheta)) \quad \text{for} \quad \vartheta \in \Theta$$

The configuration space $\Theta$ can include model hyperparameters, the choice of decoding strategy (e.g., nucleus sampling vs. beam search, which affects creativity vs. precision), and the level of detail requested from the explanation generation module.

We can solve this using evolutionary algorithms or by training a meta-model that predicts P and E for a given $\vartheta$. The outcome is not a single "best" model, but a set of Pareto-optimal models. This allows an organization to choose a specific model from the frontier that aligns with its internal policies. For instance, a team working on critical financial software might choose a model with maximum explainability ($E_{max}$), even at a slight cost to performance, while a team developing a fast-moving web application might prefer a model with maximum performance ($P_{max}$).

## IV. Proposed Experiments and Evaluation

To validate the effectiveness of the proposed POCU system and its underlying frameworks, we outline a comprehensive, multi-stage evaluation plan.

### A. Datasets and Baselines

- **Datasets:** We will use a combination of public and synthetic datasets. The Defects4J benchmark [16] provides a large set of real-world bugs from Java projects, which is ideal for evaluating bug localization and explanation tasks. We will also create a synthetic dataset of code snippets and corresponding natural language queries to evaluate the code understanding capabilities in a more controlled environment. For the federated learning evaluation, we will partition existing large codebases (e.g., the CodeSearchNet dataset [17]) to simulate different organizational silos with non-IID data distributions.
- **Baselines:** The POCU system's performance will be compared against several baselines:
  1) A fine-tuned, non-federated GenAI model (e.g., CodeLlama) to measure the impact of TMFL.
  2) Existing static analysis tools (e.g., SonarQube, PMD) to compare traditional rule-based methods with AI-driven analysis.
  3) Human performance: We will conduct a user study with experienced software developers to measure the reduction in Mean Time To Resolution (MTTR) for a set of debugging tasks.

### B. Evaluation Metrics

*1) POCU System Performance:*

- **Bug Localization Accuracy:** The percentage of test cases where the system correctly identifies the faulty line(s) of code within its top-k suggestions.
- **Explanation Quality:** Evaluated using ROUGE-L for textual similarity to human-written explanations and through developer ratings in the user study (on a 1-5 Likert scale for clarity, correctness, and actionability).
- **Mean Time To Resolution (MTTR):** Measured in the user study, comparing the time taken to solve a debugging task with and without the POCU assistant.

*2) TMFL Framework Robustness:*

- **Model Convergence Speed:** The number of communication rounds required to reach a target accuracy level, compared to standard FedAvg.
- **Adversarial Resilience:** We will simulate a poisoning attack where a subset of malicious clients attempts to degrade the global model's performance. We will measure the deviation in accuracy of the TMFL-trained model versus a standard FedAvg model.
- **Trust Score Correlation:** We will measure the correlation between a client's calculated trust score $\tau_k^t$ and the actual quality of their data to validate that the metric correctly identifies high-value contributors.

*3) EPO Framework Analysis:*

- **Pareto Frontier Generation:** We will run experiments with different model configurations to plot the resulting (P, E) pairs and visualize the Pareto frontier.

- **Trade-off Analysis:** We will analyze the shape of the frontier to quantify the ”cost” of explainability. For example, ”A 10% increase in the explainability score leads to a 3% decrease in bug detection accuracy.”

### C. *User Study Design*

We will recruit 30-40 software developers with varying levels of experience. They will be divided into a control group (using their standard IDE and debugging tools) and a treatment group (using the IDE with the POCU plugin). Both groups will be assigned a series of identical debugging tasks on a moderately complex Java application. We will measure task completion time, success rate, and collect qualitative feedback through post-task surveys on cognitive load, usability, and trust in the tool.

## V. Expected Results and Discussion

We hypothesize that the experimental results will demonstrate the viability and advantages of our proposed system.

**For the POCU system**, we expect to see a significant reduction in MTTR for the treatment group in the user study. We anticipate that the system’s ability to synthesize static and dynamic context will allow it to identify complex, multi-faceted bugs that are often missed by traditional static analyzers. The qualitative feedback is expected to highlight the system’s intuitive nature, though some developers may initially express skepticism, underscoring the importance of the explainability component.

**For the TMFL framework**, we predict that in the adversarial simulation, the global model trained with TMFL will maintain a significantly higher accuracy compared to the one trained with standard FedAvg. The trust metric should effectively down-weight the contributions from the malicious clients, preserving the integrity of the model. We also expect TMFL to show slightly slower initial convergence than FedAvg in a clean environment, as the trust scores need a few rounds to stabilize, but to achieve a higher final accuracy ceiling due to better handling of non-IID data.

**For the EPO framework**, the experiments will likely reveal a non-linear trade-off between performance and explainability. We expect to find a ”sweet spot” on the Pareto curve where a small sacrifice in performance yields a large gain in explainability. This result would be highly impactful, as it would provide a clear, data-driven methodology for organizations to configure their AI tools. The analysis might also reveal that certain model architectures or training techniques are inherently more explainable without a significant performance penalty.

Collectively, these expected results would constitute strong evidence that natural language-driven, context-aware, and trustworthy AI assistants can fundamentally improve the software debugging process, shifting it from a manual, painstaking task to a more collaborative and efficient human-AI dialogue.

## VI. Conclusion and Future Work

This paper has introduced a comprehensive framework for Prompt-Oriented Code Understanding (POCU), a paradigm aimed at revolutionizing software debugging through natural language interaction. We have proposed a system architecture that integrates GenAI with IDEs, providing deep contextual awareness. To address the critical challenges of data privacy and model integrity, we designed the Trust-Metric Federated Learning (TMFL) framework, which enables secure, collaborative model training with accountability. Finally, we presented the Explainability-Performance Optimization (EPO) framework to formally manage the crucial trade-off between the accuracy of AI suggestions and their transparency.

Our work lays a theoretical and architectural foundation for a new class of intelligent developer tools. By treating debugging as a dialogue, we aim to lower the cognitive barrier for developers, accelerate fault resolution, and ultimately improve software quality. The proposed trust and explainability frameworks are crucial steps towards building AI systems that are not only powerful but also reliable and worthy of developer trust.

Future work will focus on implementing a prototype of the POCU system and conducting the extensive experiments outlined in this paper. Several challenges remain to be explored. Handling the sheer volume of context in large-scale enterprise systems will require sophisticated context-pruning and retrieval-augmentation techniques. Extending the TMFL framework to handle more complex, multi-modal data (e.g., performance traces, user bug reports) is another promising direction. Furthermore, the psychological and human-computer interaction aspects of AI-assisted debugging warrant deeper investigation. As these systems become more capable, understanding how they reshape developer workflows and skill requirements will be of paramount importance.

## References

[1] D. A. Patterson and J. L. Hennessy, ”Computer Organization and Design: The Hardware/Software Interface,” Morgan Kaufmann, 2013.
[2] V. J. Hellendoorn, C. Sutton, R. Singh, P. Maniatis, and D. Bieber, ”The Power of Many: A Study of Collaboration on GitHub,” in *Proc. Int. Conf. on Software Engineering (ICSE)*, 2014, pp. 678–689.
[3] M. Chen et al., ”Evaluating Large Language Models Trained on Code,” *arXiv preprint arXiv:2107.03374*, 2021.
[4] M. Weiser, ”Program Slicing,” in *Proc. Int. Conf. on Software Engineering (ICSE)*, 1981, pp. 439-449.
[5] M. Pradel and K. Sen, ”DeepBugs: A Learning Approach to Name-Based Bug Detection,” in *Proc. ACM on Programming Languages (OOPSLA)*, 2018, Art. 147.
[6] Y. Kim, D. Kim, T. F. Bissyande´, E. Choi, L. Li, A. Klein, and Y. Le Traon, ”An empirical study of bug-fixing patches in the wild: revisited,” *Software: Practice and Experience*, vol. 48, no. 1, pp. 73-98, 2018.
[7] Y. Li et al., ”Competition-level code generation with AlphaCode,” *Science*, vol. 378, no. 6624, pp. 1092-1097, 2022.
[8] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, ”On the naturalness of software,” in *Proc. Int. Conf. on Software Engineering (ICSE)*, 2012, pp. 837–847.
[9] Z. Feng et al., ”CodeBERT: A Pre-Trained Model for Programming and Natural Languages,” in *Proc. Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, 2020.
[10] D. Guo et al., ”GraphCodeBERT: Pre-training Code Representations with Data Flow,” in *Proc. Int. Conf. on Learning Representations (ICLR)*, 2021.

[11] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, ”Communication-Efficient Learning of Deep Networks from Decentralized Data,” in *Proc. Int. Conf. on Artificial Intelligence and Statistics (AISTATS)*, 2017, pp. 1273-1282.

[12] H. Zhang, H. Yu, S. Wang, and X. Yuan, ”Federated Defect Prediction,” *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 4930- 4945, Dec. 2022.

[13] V. Bagdasaryan, A. Veit, Y. Hua, D. Estrin, and V. Shmatikov, ”How to Backdoor Federated Learning,” in *Proc. Int. Conf. on Artificial Intelligence and Statistics (AISTATS)*, 2020, pp. 1625-1635.

[14] M. T. Ribeiro, S. Singh, and C. Guestrin, ””Why Should I Trust You?”: Explaining the Predictions of Any Classifier,” in *Proc. ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, 2016, pp. 1135- 1144.

[15] S. M. Lundberg and S. Lee, ”A Unified Approach to Interpreting Model Predictions,” in *Proc. Conf. on Neural Information Processing Systems (NeurIPS)*, 2017.

[16] R. Just, D. Jalali, and M. D. Ernst, ”Defects4J: A database of existing faults to enable controlled testing studies for Java programs,” in *Proc. Int. Symp. on Software Testing and Analysis (ISSTA)*, 2014, pp. 437-440.

[17] H. Husain, H. Wu, T. Kustner, W. Fedus, R. Liaw, A. D. H. van den Oord, and A. L. Gaunt, ”CodeSearchNet Challenge: Evaluating the State of the Art in Code Search,” *arXiv preprint arXiv:1909.09436*, 2019.

[18] S. Ji, S. Luan, J. He, Y. Lyu, Y. Su, S. Pu, Y. Zhang, and X. He, ”Survey of Explainable AI (XAI): A Technical Perspective,” *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 7, no. 1, pp. 8-28, Feb. 2023.

[19] K. Li, J. Wang, J. Wang, T. Li, Z. Wang, Z. Liu, H. Zhang, and D. Zhang, ”Federated Learning for Software Engineering: A Case Study of Defect Prediction,” *IEEE Transactions on Software Engineering*, early access, 2023.