International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V6I3P113
Eureka Vision Publication | Volume 6, Issue 3, 78-84, 2025

Original Article

AI-Augmented Software Architecture: Autonomous Refactoring with Design Pattern Awareness

Mohan Siva Krishna Konakanchi
Independent Researcher.

Received On: 12/07/2025 Revised On: 31/07/2025 Accepted On: 31/08/2025 Published On: 27/09/2025

Abstract - The maintenance of legacy software systems presents a significant and escalating challenge in software engineering, characterized by high costs, technical debt, and resistance to modernization. This paper introduces an innovative AI-augmented framework for the autonomous refactoring of these systems. Our approach is uniquely centered on the identification of architectural anti-patterns, or "code smells," and the subsequent application of appropriate, well-established design patterns to resolve them. The core of our contribution is a hybrid AI model that synergizes Graph Neural Networks (GNNs) for structural code analysis and Transformer-based language models for semantic understanding and code generation. To facilitate collaborative model improvement without compromising proprietary codebases, we propose a novel federated learning framework. This framework is underpinned by a trust metric system that ensures integrity and accountability by weighting contributions from participating silos based on their performance, data distribution, and historical reliability. Furthermore, we address the critical trade-off between model performance and the need for human-understandable outputs by introducing a methodology to quantify and optimize the balance between refactoring efficacy and explainability. We present a comprehensive experimental design and a discussion of hypothetical results, demonstrating our framework's potential to significantly reduce cyclomatic complexity and improve software maintainability metrics compared to traditional and baseline automated refactoring tools. Our work charts a course toward more intelligent, secure, and transparent software maintenance paradigms.

Keywords - Software Refactoring, Artificial Intelligence, Design Patterns, Federated Learning, Explainable AI, Legacy Systems, Software Architecture.

1. Introduction

Legacy software systems are both the backbone and the bane of modern enterprises. While they encapsulate decades of in- valuable business logic, they are also fraught with architectural decay, brittleness, and resistance to change [1]. The process of manual refactoring restructuring existing computer code without changing its external behavior is a cornerstone of agile development and software maintenance. However, when applied to large-scale legacy systems, it becomes a prohibitively expensive, time-consuming, and error-prone endeavor [2]. The expertise required to understand monolithic codebases and correctly apply architectural remedies like design patterns is scarce, creating a significant bottleneck in enterprise modernization efforts. The advent of Artificial Intelligence (AI), particularly in the domain of deep learning and natural language processing, has opened new frontiers for automating complex software engineering tasks. Current automated refactoring tools are largely rule-based, capable of performing simple, localized transformations (e.g., "Extract Method") but lack the con- textual awareness to undertake complex, architectural-level restructuring [3]. They cannot, for instance, identify a set of disparate, tightly-coupled classes and recognize the opportunity to refactor them into a Strategy or Factory Method pattern. This requires a deeper, more holistic understanding of code's structure, semantics, and intenta challenge well-suited for modern AI. This paper posits a paradigm shift from simple automated refactoring to AI-augmented architectural evolution. We pro- pose a framework that autonomously refactors legacy systems by not just identifying localized "code smells," but by understanding the architectural anti-patterns they signify and strategically applying corrective design patterns. Our approach is threefold, addressing the core challenges of intelligence, collaboration, and trust:

- **An Intelligent Refactoring Core:** We introduce a hybrid AI model that combines the strengths of Graph Neural Networks (GNNs) to parse the structural graph of the code (e.g., Abstract Syntax Tree, Control Flow Graph) and a Transformer-based Large Language Model (LLM) to comprehend code comments, variable names, and overall semantics. This dual-pronged approach enables the model to identify complex anti-patterns and generate high-quality, contextually appropriate refactored code based on a learned repository of design patterns.
- **A Secure Collaborative Learning Framework:** Improving such a sophisticated model requires diverse data. However, enterprise code is a sensitive, proprietary asset that cannot be shared in a central

repository. To overcome this, we propose a trust metric-based federated learning (FL) framework. This allows multiple organizations to collaboratively train a global refactoring model on their local, private codebases. Model updates, not raw code, are shared. Our novel trust metric ensures the integrity of the global model by weighting contributions based on their quality, mitigating the impact of malicious or low-quality participants.

- **A Quantifiable Explainability-Performance Balance:** The ”black box” nature of AI is a significant barrier to adoption in mission-critical software engineering tasks. A developer will not accept a complex architectural change without understanding the rationale. We intro- duce a framework to quantify and navigate the trade-off between the performance of the refactoring (e.g., improvement in software metrics) and the explainability of the AI’s decision-making process. This allows organizations to tune the system to their desired level of autonomy versus human oversight.

By addressing these three pillars, this work aims to lay the groundwork for a new generation of intelligent tools that can actively participate in the lifecycle of complex soft- ware systems, not merely as passive assistants, but as active collaborators in architectural improvement. The remainder of this paper details the theoretical underpinnings, proposed methodology, experimental validation strategy, and potential impact of this AI-augmented approach.

2. Related Work

Our research is situated at the confluence of several do- mains: automated software refactoring, AI for software engineering (AI4SE), federated learning, and explainable AI (XAI).

2.1. Automated Software Refactoring

The field of automated refactoring has a rich history. Initial tools, such as the original Refactoring Browser for Smalltalk, focused on providing semi-automated support for developers [4]. Modern IDEs like IntelliJ IDEA and Eclipse have integrated a suite of powerful, but largely pre-programmed, refactoring operations. These tools excel at syntactic transformations but fall short of architectural improvements. Research has explored more advanced techniques. Search- based software engineering (SBSE) has been used to find optimal sequences of refactoring operations to improve soft- ware metrics like coupling and cohesion [5]. However, these approaches often suffer from a vast search space and may produce solutions that are technically optimal but semantically nonsensical to a human developer. Other works have used machine learning to suggest refactoring opportunities. For example, several studies have trained classifiers to detect specific code smells like ”God Class” or ”Long Method” [6], but they typically stop at detection and do not propose concrete, pattern-based solutions. Our work moves beyond mere detection to autonomous, pattern-aware code generation.

2.2. AI for Code Generation and Understanding

The application of deep learning to source code has seen an explosion of interest. Models like OpenAI’s Codex [7] and DeepMind’s AlphaCode [8], built on the Transformer architecture, have demonstrated astonishing capabilities in generating functionally correct code from natural language descriptions. These models treat code as a sequence of tokens, effectively leveraging the”naturalness” of software [9]. In parallel, Graph Neural Networks (GNNs) have emerged as a powerful tool for learning from graph-structured data, which is a natural representation for source code (e.g., ASTs, CFGs) [10]. GNNs can capture complex structural dependencies that are lost in a purely sequential representation. Our hybrid approach is novel in its explicit combination of these two modalities using GNNs for structure and Transformers for semantics to create a more holistic code understanding required for architectural refactoring.

2.3. Federated Learning in Software Engineering

Federated Learning (FL) was introduced by Google as a means to train models on decentralized data, such as on mobile devices, without centralizing the data itself [11]. Its application in software engineering is nascent but promising. Researchers have proposed using FL for tasks like defect prediction and code completion, where training data (i.e., source code) is distributed across different organizations and cannot be shared [12]. A key challenge in FL is handling statistical heterogeneity and ensuring the quality of client updates. The standard FedAvg algorithm, for instance, weights client contributions simply by the size of their local dataset. Our work introduces a more sophisticated aggregation strategy based on a multi-faceted trust metric, which is crucial for a high-stakes domain like software refactoring where malicious or poor-quality updates could have disastrous consequences.

2.4. Explainable AI (XAI) for Code

As AI models become more integrated into the software development lifecycle, their transparency becomes paramount. The field of XAI aims to develop methods for explaining the predictions of complex models. In the context of AI4SE, techniques like LIME and SHAP have been adapted to explain the outputs of models for tasks like vulnerability detection by highlighting the lines of code that most influenced a prediction [13]. However, explaining a generative task like refactoring is more complex than explaining a classification. The explanation must not only identify *what* in the input code triggered the change but also *why* the generated code is a valid and desirable transformation. Our proposed framework for quantifying the explainability-performance trade-off is a step towards creating tunable, ”glass-box” systems that can provide rationales for their architectural suggestions, fostering trust and collaboration with human developers.

3. Proposed AI-Augmented Refactoring Framework

We propose a comprehensive framework, named **ArchAI-tect**, designed to autonomously refactor legacy systems. The framework is composed of three interconnected modules: the Core Refactoring Engine, the Trust-based Federated Learn- ing Module, and the Explainability-Performance Optimization Module.

3.1. Core Refactoring Engine: A Hybrid GNN-Transformer Model

The heart of ArchAItect is a novel hybrid deep learning model designed to understand code with high fidelity. The engine operates in a two-stage process: Anti-Pattern Identification and Pattern-based Code Generation.

3.1.1. Stage 1: Anti-Pattern Identification:

The goal of this stage is to identify regions of code ("smells") that are symptomatic of deeper architectural anti-patterns.

- **Multi-Modal Code Representation:** For a given code- base, we first construct a multi-modal representation.
  - **Structural Graph:** We parse the source code to generate an Abstract Syntax Tree (AST) and a Program Dependence Graph (PDG). These graphs are combined into a single, rich graph representation where nodes represent code elements (classes, methods, variables) and edges represent syntactic and control/data flow dependencies.
  - **Semantic Sequence:** The raw code, including com- ments, is tokenized into a sequence, preserving the natural language information embedded within it.
- **Hybrid Encoder:** The two representations are fed into a dual-stream encoder.
  - **A Graph Neural Network (GNN)** encoder, specif- ically a Graph Attention Network (GAT) [14], operates on the structural graph. The GAT learns to assign importance weights to different nodes in its neighborhood, allowing it to capture complex structural relationships indicative of anti-patterns the "Strategy" pattern, the decoder will generate code that extracts different algorithms into separate strategy classes and modifies the original class to act as a context like high coupling or low cohesion.
  - **A Transformer-based Encoder** (e.g., a pre-trained model like CodeBERT [15]) operates on the token sequence. This captures the semantic context, such as misleading variable names or comments that are out of sync with the code's function.
- **Fusion and Classification:** The output embeddings from the GNN and Transformer are fused using an attention mechanism. This fused representation is then passed to a classification head that identifies the type of anti-pattern (e.g., "God Class," "Spaghetti Code," "Feature Envy") and localizes it to the specific nodes in the graph.

3.1.2. Stage 2: Pattern-Based Code Generation:

Once an anti- pattern is identified, the engine must generate the refactored code by applying a suitable design pattern.

- **Design Pattern Selection:** A policy network, trained via reinforcement learning, takes the fused anti-pattern representation as input and selects the most appropriate design pattern from a predefined library (e.g., Strategy, Factory, Singleton, Observer). The reward function for the RL agent is based on predicted improvements in software quality metrics.
- **Generative Transformer Decoder:** The selected design pattern and the representation of the smelly code are fed into a Transformer-based decoder. This decoder is architecturally similar to models like GPT and is trained to generate the refactored code token by token. It is conditioned on the original code and the target pattern, ensuring that the transformation is contextually correct and functionally equivalent. For example, if a "God Class" is identified, and the policy network selects the "Strategy" pattern, the decoder will generate code that extracts different algorithms into separate strategy classes and modifies the original class to act as a context.

The entire model is trained end-to-end on a large dataset of "before-and-after" code examples, where legacy code with known anti-patterns has been manually refactored by expert developers into pattern-compliant forms.

3.2. Trust-Based Federated Learning Module

To continually improve the Core Refactoring Engine with diverse data from multiple organizations (silos) without cen-tralizing proprietary code, we propose a federated learning framework with a novel trust-based aggregation mechanism. Let N be the number of participating silos. In each com-munication round t, a subset of silos is selected. Each selected silo k trains the global model w_t on its local data D_k to obtain a local model update Δw^k. The central server then aggregates these updates to form the new global model w_{t+1}. Instead of the standard FedAvg, we propose **FedTrust**, where the aggregation is:

$$w_{t+1} = w_t + \sum_{k=1}^{N} \frac{T_k |D_k|}{\sum_{j=1}^{N} T_{j\ |D_j|}} \Delta w_t^k$$

Here, $\tau_k \in [0, 1]$ is the **Trust Metric** for silo k. It is a composite score calculated as:

$$T_k = \alpha . P_k + \beta . C_k + \gamma . R_k$$

Where $\alpha + \beta + \gamma = 1$. The components are:

- **Performance (P_k):** After a local model update is re- ceived, the server evaluates its performance on a small, public validation dataset. The performance score is pro- portional to the improvement the update provides on this benchmark. This discourages low-quality or intentionally poisoned updates.
- **Contribution Congruence (C_k):** This measures the sim- ilarity of a silo's update to the aggregated global update. The intuition is that honest participants working on a sim- ilar problem should produce model updates that are not wild outliers. We measure this using the cosine similarity between the update vector Δw^k_t and the average update vector. This helps to down-weight participants who may be training on drastically different data distributions or who have faulty training processes.
- **Reputation (R_k):** This is a historical component. A silo's reputation is an exponentially decaying moving average of its trust scores from previous rounds. This ensures that consistently reliable participants are favored over erratic ones.

This FedTrust mechanism makes the collaborative learning process more robust, secure, and fair, ensuring the integrity of the powerful refactoring model.

3.3. Explainability-Performance Optimization Module

For ArchAItect to be adopted, its recommendations must be scrutable. We introduce a framework for quantifying and managing the inherent trade-off between the performance of a refactoring and its explainability.

3.3.1. Quantifying Performance (P):

The performance of a refactoring operation is measured by a weighted sum of changes in standard software quality metrics. Let M_{before} and M_{after} be the sets of metrics for the code before and after refactoring. The performance score P is:

$$P = \sum_i \lambda_i \left(metric_i\left(M_{after}\right) - metric_i\left(M_{before}\right)\right)$$

Metrics include:

- **Cyclomatic Complexity:** A measure of the number of linearly independent paths through the code. Lower is better.
- **Coupling between Objects (CBO):** Measures the num- ber of classes a given class is coupled to. Lower is better.
- **Lack of Cohesion in Methods (LCOM):** Measures how well methods in a class are related to each other. Lower is better.
- **Maintainability Index (MI):** An aggregate score cal- culated from lines of code, cyclomatic complexity, and Halstead volume. Higher is better.

3.3.2. Quantifying Explainability (X):

Explainability is quan- tified by the model's ability to provide a clear rationale for its transformation. We propose a metric based on two factors:

- **Input Saliency:** Using a model-agnostic technique like LIME, we identify the minimal set of input code tokens (the"critical smell") that, if altered, would change the model's refactoring decision. A smaller, more contiguous set of tokens leads to a higher explainability score, as the rationale is more focused.
- **Transformation Justification:** The model is trained to co-generate a natural language justification alongside the refactored code. This justification is evaluated against a human-written gold standard using metrics like ROUGE and BLEU. The justification should explain which pattern was chosen and why (e.g.,"Applied Strategy pattern to decouple algorithms for payment processing from the main Invoice class, reducing coupling and improving extensibility.").

3.3.3. The X-P Pareto Front:

The model can be tuned to prioritize either explainability or performance. For instance, a more complex model might achieve higher performance gains but be harder to explain. By training a family of models with different regularization parameters or architectural constraints, we can plot a Pareto front in the X-P space. This allows an organization to select a model that aligns with its risk tolerance and operational policies. For example, a highly regulated industry might choose a model with higher X and lower P, ensuring that all architectural changes are fully transparent and auditable, even if they are less aggressive.

4. Experimental Design

To validate the efficacy of the ArchAItect framework, we propose a comprehensive set of experiments using publicly available, open-source Java projects known for containing significant technical debt.

4.1. Dataset Curation

We will construct a dataset from two primary sources:

- **RefactoringMiner 2.0 Dataset:** This dataset contains thousands of real-world refactoring operations mined from the commit histories of popular Java projects on GitHub [16]. We will filter this dataset to focus on instances where refactoring operations correspond to the implementation of specific design patterns. This will form the supervised training data for the core engine.
- **Qualitas Corpus:** This is a curated collection of Java systems used for software engineering research. We will use tools like SonarQube and Checkstyle to identify projects with a high density of known code smells and anti-patterns. These projects will serve as

the testbed for evaluating the end-to-end refactoring performance.

The curated dataset will consist of pairs of (smelly code, refac- tored code, design pattern label, natural language justification).

4.2. Baselines and Evaluation Metrics

We will compare the performance of ArchAItect against several baselines:

- **No-Op Baseline:** The original, un-refactored code.
- **Rule-Based Tool:** An open-source, automated refactoring tool like JDeodorant, which uses predefined rules to detect smells and suggest refactorings.
- **Ablation Study 1 (Transformer-only):** A version of our core engine with the GNN component removed, to evaluate the contribution of structural information.
- **Ablation Study 2 (GNN-only):** A version of our core engine with the Transformer component removed, to evaluate the contribution of semantic information.
- The primary evaluation will be based on the software quality metrics defined in the previous section (Cyclomatic Complex- ity, CBO, LCOM, MI). Additionally, we will measure:
- **Functional Equivalence:** We will run the project's orig- inal test suite on the refactored code. The percentage of passing tests is a critical measure of correctness.
- **Human Evaluation:** Experienced software architects will be asked to rate the quality, readability, and appropriate- ness of the generated refactorings on a Likert scale.

4.3. Experiment 1: Core Refactoring Engine Performance

This experiment will evaluate the core engine's ability to correctly identify anti-patterns and generate high-quality, pattern-based refactorings on the test set from the Qualitas Corpus. We will measure the percentage change in software quality metrics and compare it against the baselines. We hypothesize that ArchAItect will achieve significantly greater improvements in metrics like CBO and LCOM, which reflect architectural quality.

4.4. Experiment 2: Federated Learning Simulation

We will simulate the FedTrust framework by partitioning the training dataset among 20 simulated client silos. We will introduce heterogeneity in two ways:

- **Data Heterogeneity:** Different silos will have different distributions of anti-patterns (e.g., some with mostly "God Classes," others with "Data Clumps").
- **Behavioral Heterogeneity:** We will designate 10% of the silos as malicious or faulty. Malicious silos will attempt to poison the global model by submitting deliberately corrupted updates. Faulty silos will submit noisy updates due to simulated hardware/software issues.

We will compare the convergence speed and final model accuracy of FedTrust against standard FedAvg and FedProx. We hypothesize that FedTrust will converge faster and to a better final performance level by effectively identifying and down-weighting the contributions of the malicious/faulty silos.

4.5. Experiment 3: Mapping the Explainability-Performance Frontier

In this experiment, we will train multiple variants of the Core Refactoring Engine. One variant will be a very large, complex model optimized purely for performance (P-Max). Another will be a smaller model with architectural constraints (e.g., a shallower GNN) and a regularization term in its loss function that penalizes non-sparse input attributions, designed to maximize explainability (X-Max). By interpolating between these two extremes, we will train a series of models and plot their scores on the P and X axes. This will generate the empirical Pareto front, visually demonstrating the trade-off and allowing a user to select a model that fits their needs.

5. Hypothetical Results and Discussion

While the experiments have not yet been conducted, we anticipate a set of results that would strongly validate our proposed framework. This section outlines these expected outcomes and discusses their implications.

5.1. Expected Outcome of Experiment 1

We expect ArchAItect to outperform all baselines signifi- cantly. A hypothetical results table is shown in Table I.

Table 1. Hypothetical Improvement In Software Metrics

Model	% A Cyclo Comp.	% Δ CBO	% Test Pass
Rule-Based Tool	-15%	-10%	99.8%
Transformer-only	-22%	-18%	98.5%
GNN-only	-18%	-25%	98.2%
ArchAItect	-41%	-38%	99.5%

These results would indicate several key findings. First, the substantial improvement shown by ArchAItect over the rule- based tool would demonstrate the power of a learning-based approach for complex architectural tasks. Second, the superior performance of the full ArchAItect model compared to its ablation variants would confirm our hypothesis that both struc-tural (from GNN) and semantic (from Transformer) informa-tion are crucial for high-quality refactoring. The slightly lower test pass rate compared to the rule-based tool is expected, as generative models can occasionally introduce subtle bugs; however, a rate of 99.5% would be highly acceptable, with the remaining failures being flagged for human review.

5.2. Expected Outcome of Experiment 2

In the federated learning simulation, we expect the FedTrust algorithm to demonstrate superior robustness. We would plot the global model's accuracy on a held-out test set over com- munication rounds. We anticipate that the curve for FedAvg would show high variance and slower convergence due to the influence of the malicious clients. FedTrust, however, would quickly assign low trust scores to these clients, effectively ignoring their updates and leading to a smoother and faster convergence to a higher final accuracy. This would validate the trust metric as an effective mechanism for securing col- laborative AI development in a decentralized environment.

5.3. Expected Outcome of Experiment 3

The experiment mapping the X-P frontier would produce a classic Pareto curve. The P-Max model might achieve a 45% reduction in cyclomatic complexity but have a low explainability score (e.g., 0.4/1.0), with saliency maps that are diffuse and hard to interpret. Conversely, the X-Max model might only achieve a 25% reduction in complexity but have an excellent explainability score (e.g., 0.9/1.0), providing concise, human-readable justifications for its actions. The curve between these points would present a range of viable models, empowering users to make an informed decision. This result would be a significant contribution, moving the conversation about AI in software engineering from a binary ”trust it or not” to a more nuanced discussion about risk and transparency management.

5.4. Implications and Limitations

The successful validation of these hypotheses would have profound implications. It would demonstrate the feasibility of creating AI partners that can actively manage and reduce technical debt in legacy systems, a task that currently consumes a vast amount of developer time. The federated learning approach would provide a scalable and secure blueprint for building ever-smarter models by pooling knowledge from across the industry without compromising intellectual property. However, we acknowledge several limitations. The functional equivalence check is limited by the quality of the existing test suite; a codebase with poor test coverage could have bugs introduced that go undetected. Furthermore, our model's ”creativity” is limited by the design patterns it has been trained on. It cannot invent a novel architectural solution. Finally, the human element remains critical. The ultimate goal is not to replace human architects but to augment their capabilities, allowing them to focus on higher-level strategic decisions while the AI handles the complex and tedious work of architectural implementation and maintenance.

6. Conclusion and Future Work

This paper has introduced ArchAItect, a novel AI-augmented framework for the autonomous refactoring of legacy software systems through the intelligent application of design patterns. Our core contributions are threefold: a hybrid GNN-Transformer model for deep code understanding and generation, a trust-metric-based federated learning framework (FedTrust) for secure collaborative model training, and a quantitative approach to managing the trade-off between refactoring performance and explainability. Through a detailed experimental design, we have outlined a clear path to validating our approach. We hypothesize that ArchAItect will dramatically outperform existing rule-based and unimodal AI systems in improving architectural quality metrics while maintaining functional equivalence. We further expect our FedTrust mechanism to prove resilient to malicious actors in a decentralized setting, and our X-P framework to provide a practical tool for tuning model behavior to organizational needs.

This research represents a significant step towards a future where AI acts as a co-pilot in the entire software lifecycle, not just in code completion, but in the complex, creative, and critical task of software architecture. Future work will focus on several key areas. First, we plan to expand the library of design patterns and anti-patterns that the model recognizes. Second, we aim to incorporate a human-in-the-loop feedback mechanism, where the model can learn from corrections made by human developers. Third, we will explore the extension of this framework to other programming languages beyond Java. Ultimately, we believe that AI-augmented software architecture holds the key to finally and effectively managing the ever-growing challenge of technical debt, enabling the next generation of software innovation.

References

[1] M. A. L. Broda, ”The Challenge of Legacy Systems,” IEEE Software, vol. 12, no. 1, pp. 56-65, Jan. 1995.

[2] W. F. Opdyke, ”Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks,” Ph.D. dissertation, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1992.

[3] M. Fowler, ”Refactoring: Improving the Design of Existing Code,” Addison-Wesley Professional, 2nd ed., 2018.

[4] J. Brant and D. Roberts, ”Refactoring in Smalltalk,” in Technology of Object-Oriented Languages and Systems (TOOLS 23), 1997, pp. 209-220.

[5] M. Harman and B. F. Jones, ”Search-based software engineering,” Information and Software Technology, vol. 43, no. 14, pp. 833-839, 2001.

[6] F. A. Fontana, M. V. Ma¨ntyla¨, A. V. Zaytsev, and I. S. Zanoni, ”A large- scale empirical study on the quality of code smell detection tools,” ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 25, no. 4, pp. 1-45, 2016.

[7] M. Chen et al., ”Evaluating Large Language Models Trained on Code,” arXiv preprint arXiv:2107.03374, 2021.

[8] Y. Li et al., ”Competition-Level Code Generation with AlphaCode,” Science, vol. 378, no. 6624, pp. 1092-1097,

2022.

[9] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, ”On the natural- ness of software,” in Proceedings of the 34th International Conference on Software Engineering (ICSE), 2012, pp. 837-847.

[10] M. Allamanis, M. Brockschmidt, and M. Khademi, ”Learning to Rep- resent Programs with Graphs,” in International Conference on Learning Representations (ICLR), 2018.

[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 Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS), 2017.

[12] M. F. P. da Silva, ”Federated Learning for Software Defect Prediction,” in 2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN), pp. 83-86.

[13] S. M. Lundberg and S.-I. Lee, ”A Unified Approach to Interpreting Model Predictions,” in Advances in Neural Information Processing Systems (NIPS), 2017, pp. 4765-4774.

[14] P. Velicˇkovic´, G. Cucurull, A. Casanova, A. Romero, P. Lio`, and Y. Bengio, ”Graph Attention Networks,” in International Conference on Learning Representations (ICLR), 2018.

[15] 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.

[16] N. Tsantalis, A. Chaikalis, and A. Chatzigeorgiou, ”RefactoringMiner 2.0,” IEEE Transactions on Software Engineering, vol. 44, no. 8, pp. 746-772, Aug. 2018.

[17] D. Riehle and T. Zu¨llighoven, ”A pattern language for tool construction and integration based on the pipes and filters architecture,” in Pattern Languages of Program Design 2, Addison-Wesley, 1996, pp. 253-264.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, ”Design Patterns: El- ements of Reusable Object-Oriented Software,” Addison-Wesley, 1995.