Nan Jiang | Research Statement

Software powers the modern world, driving the websites, applications, and critical infrastructure essential to daily life, industries, and government. The software market is projected to keep increasing, with higher demands for new software and more developers to build and maintain them. However, software development remains challenging and resource-intensive. Large projects may take over a year to finish, or even run over budget and time yet still produce less value than predicted due to unexpected difficulties encountered during development.

My research in *Artificial Intelligence for Software Engineering* aims to build accurate and practical AI assistants to support software development. I have publications and ongoing research supporting various stages of the software development lifecycle as illustrated in Figure 1. This includes specification generation [10] and front-end interface design [11] in the "Designing" stage; source code generation [1] in the "Implementation" stage; automated program repair [2, 3, 4, 6] in the "Testing" stages; and vulnerability fixing [7] and reverse engineering [5, 8, 9] in the "Maintenance" stage. Although some of these tasks have been explored for years, AI assistants for software development are still not widely applied or accepted by developers in practice, mainly due to their limited accuracy and reliability. My research is guided by the key insight that effective AI assistants must incorporate "domain knowledge" specific to software engineering.

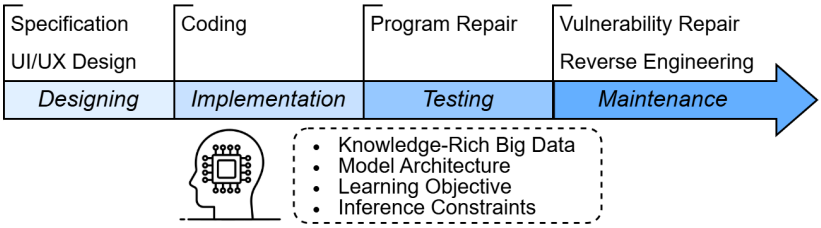

Figure 1: My work on building domain-knowledge-enhanced AI assistants for supporting various stages in the software development lifecycle.

Domain knowledge is a broad term that refers to specialized insights within a particular field. In software engineering, domain knowledge includes programming language syntax, semantics, developers' programming practices, and so on. My work identifies four approaches to equip AI models with such expertise, as illustrated in Figure 1: leveraging knowledge-rich big data to train AI models, designing domain-specific architectures for programs, tailoring learning objectives, and incorporating domain knowledge as inference constraints.

Providing More Accurate Program Repair to Developer

As developers spend nearly half, 49±39%, of their time fixing bugs during software development, there is an urgent demand for automated program repair assistants that can produce code patches given the buggy program. Yet, developers do not trust automated program repair assistants well, since they introduce new bugs to the program. My work [2] investigates the incorrect patches and reveals that more than 67% of patches produced by AI-powered program repair assistants are uncompilable due to syntax or semantics errors. As one of the earliest efforts in this area, my work [2] calls for increasing the compilation rate of patches produced by AI-powered program repair assistants. Motivated by this, I developed CURE [2] and KNOD [3] which learn the software engineering domain knowledge in various ways as illustrated in Figure 2. CURE and KNOD increase the compilation rate of patches and ultimately correctly fix 12.5% more bugs than the best previous approaches.

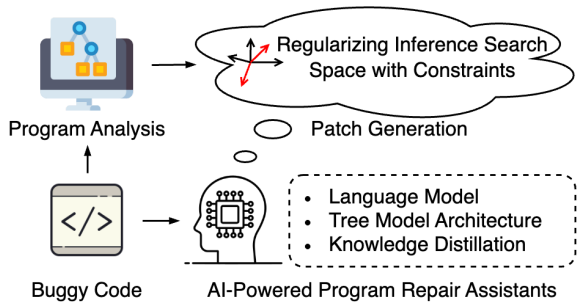

Figure 2: Building better program repair assistants that learn domain knowledge in various ways.

1. Learning programming language through pre-training. AI-powered program repair tools typically use neural networks to learn bug-fixing patterns from GitHub commit histories. However, generating syntactically correct patches remains challenging, as these models struggle to fully capture programming language syntax from limited bug-fixing data. CURE addresses this by combining a pre-trained language model, trained on large-scale source code, with a program repair model trained on bug-fixing data. The pre-trained language model captures such patterns effectively, enabling the repair model to generate syntactically correct patches [2]. CURE is one of the first approaches to integrate pre-trained language models with expert models in software engineering.

2. Guaranteeing identifiers' validity. Developers understand that before using an identifier, such as a function or variable, it must first be declared in the code—an immutable rule programmers instinctively follow in daily coding. Neural network models, however, lack this guarantee and may generate patches invoking undefined identifiers. To address this, CURE performs static analysis to identify all valid identifiers, including keywords, built-in functions, and user-defined elements. This information is used as inference constraints to guide the model, prune search space, and ensure that generated patches contain only valid identifiers [2].

3. Verifying type semantics matching. Code semantics such as typing information are much more complex than programming language syntax and identifier validity. How many arguments should I pass to a function invocation? What are the types of those arguments? What fields in this class instance I can access? These questions are all related to typing semantics and are challenging for neural networks to learn. Inspired by modern IDEs, which use language parsers to perform type-checking on code's abstract syntax tree (AST), I developed KNOD with a tree-based neural network architecture designed to learn and generate the AST of patch code. This tree-based model architecture enables KNOD to capture the structural properties of code and make full use of the static analysis information of the program to verify the type semantics matching. KNOD not only integrates type matching as inference constraints during inference but also introduces a knowledge distillation objective during training to enhance its learning of such semantic correctness [3].

Enabling AI Assistants to Work Like Developers: Coding, Reviewing, Debugging

Given the high demand for building code generation assistants, and based on my expertise in automated program repair, I started to realize that coding and program repairing are not separate tasks but deeply intertwined as iterative processes. Practical AI assistants should be able to support developers like collaborative partners, seamlessly integrating into developers' workflows that involve coding, reviewing, and debugging.

Thus, during my internship in the AWS AI Labs at Amazon, I built unified AI assistants that can write code, review the code based on execution, and debug the wrong code iteratively [1] as illustrated in Figure 3. A key challenge in this work was the lack of training data capturing the iterative process of coding, reviewing, and debugging. While developers often push their bug fixes to GitHub, they typically do so only after resolving the bugs. The intermediate trial-and-error steps, which are conducted locally and hold significant value, are not recorded or shared. My work addresses this challenge by proposing a pipeline named LeDex [1] that consists of data synthesizing and model training. The data synthesis process involves sampling programs for diverse tasks, identifying buggy ones through test case execution, generating natural language explanations for code review, and refining the code using a teacher-student framework or self-bootstrapping. This data synthesis process is designed to mirror developers' real-world workflows and practices. Using the synthesized data, LeDex fine-tunes pre-trained large language models (LLMs) through a combination of supervised fine-tuning and reinforcement learning. The fine-tuned LLMs demonstrate a significantly enhanced ability to provide more insightful code reviews and successfully refine up to 174.4% more bugs encountered during their coding assistance.

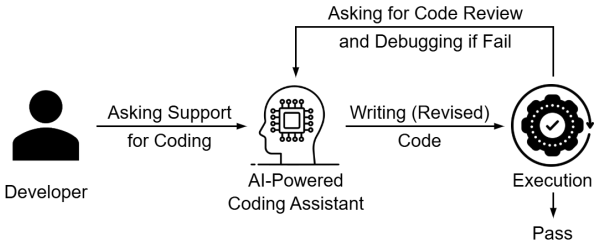

Figure 3: Building unified coding assistant capable of writing code, reviewing incorrect code, and performing debugging.

Building Reverse Engineering Assistants for Binary Code Analysis

I have also developed AI-powered assistants for binary code analysis. Binary code plays a vital role in the security domain, serving as the foundation for critical tasks such as vulnerability detection, malware analysis, binary recovery, and legacy software maintenance. However, analyzing binaries is challenging: Assembly code is difficult to interpret due to its low-level nature, and decompiled source code often lacks semantic-preserving variable and data structure names, making it less comprehensible.

Figure 4 illustrates my work on building reverse engineering assistants, which tackles the challenge in two aspects. First, developing stronger models for binary code [5] through binary-specific model design and training objectives. Second, equipping models with program analysis and posterior reasoning rules for accurate and consistent variables and data structure recovery [8]. For instance, recovering the greyed field access "`*(int`

*)(a1 + 28)" to the greened "context->type" that is easier to understand, and providing greyed identifiers "v4" with the yellowed meaningful name "total_len" according to the code context.

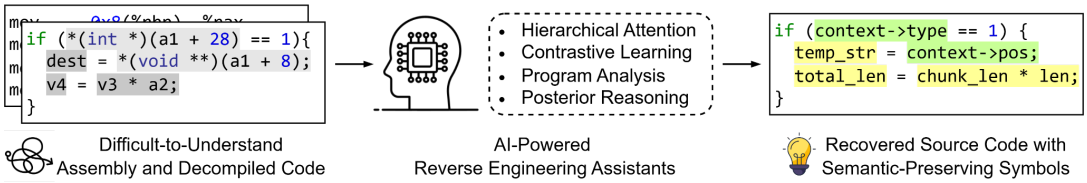

Figure 4: Building better reverse engineering assistants to recover semantic-preserving symbols of the code.

1. Developing stronger LLMs for binary code. Modern LLMs pre-trained majorly on natural language and source code have demonstrated impressive ability in source code understanding and generation tasks. Yet, they perform poorly on Assembly code, due to the lower information density of Assembly than source code and the diverse compiler optimizations. I developed Nova [5], a series of foundation LLMs tailored for Assembly code. Nova applies a novel hierarchical attention mechanism that captures the Assembly's low-density semantics at three granularity levels: within each instruction, between adjacent instructions, and across long-distance instructions. Additionally, Nova employs contrastive learning objectives to align the model's understanding of differently-optimized Assembly code that shares the same functionality. With its domain-specific model design and tailored learning objectives, Nova outperforms standard LLMs, recovering 146.5% more source code from Assembly correctly. This provides reverse engineers with valuable references for understanding Assembly code.

2. Equipping LLMs with program analysis and posterior reasoning rules. LLMs are prone to hallucination, which can lead to misleading reverse-engineering results. To address this, I collaborated on developing ReSym [8], a tool that verifies and aggregates the variable names and data structure symbols recovered by LLMs using program analysis and posterior reasoning rules. By analyzing data dependencies and function call graphs, ReSym ensures consistency in the recovered variable names and type symbols across an entire project. It also aggregates the recovery of different fields within a data structure, providing a comprehensive view. As a result, ReSym delivers trustworthy symbol recovery for binary code understanding. This work has won the ACM SIGSAC distinguished paper.

My Other Work

In addition to building AI assistants for software development, my research spans broader domains, including studying hallucinations in LLM-generated source code [13], developing practical benchmarks for repository-level code generation to evaluate LLMs' capabilities in writing code for industrial scenarios [14], designing AI models for automated recognition of formulas and tables in PDFs [12], and leveraging LLMs to guide safe and efficient planning for robotic agents [15]. These diverse research endeavors reflect my commitment to advancing AI technologies across various domains, addressing real-world challenges, and pushing the boundaries of their practical applications.

Future Vision and Research Plan

In my future work, I aim to advance the building of more accurate and practical AI assistants to support software development. My focus will be on achieving higher accuracy to ensure these tools are widely trusted and adopted by developers, while also expanding their capabilities to cover additional tasks such as unit test generation, documentation creation, and beyond.

1. "AI developer" for software engineering. I envision the future of AI assistants in software development as the so-called "AI developer" or "AI programmer" who works as virtual colleagues with human developers. Like their human counterparts, these AI developers would actively participate in every aspect of software development. To move toward this goal, I outline a high-level plan to address the research challenges that lie in the way.

First, AI assistants must be capable of managing large-scale, real-world projects. Current models, including LLMs and LLM agents, often struggle with understanding, coding, and debugging complex systems. Enhancing these capabilities is key for practical application in software development. Second, AI assistants need a deeper understanding of code execution, as software is inherently dynamic, involving changing states, system

interactions, and runtime behaviors. Understanding the code execution is essential to write functionally correct and safe programs as well as debugging and repairing code misbehavior or vulnerability. Third, AI assistants should effectively integrate with external tools commonly used in software development, such as debuggers, compilers, version control systems, and testing frameworks. Leveraging these tools can greatly improve their ability to automate workflows and deliver robust solutions. Finally, to collaborate effectively with developers, AI assistants need stronger memory and customization capabilities. Software development is a long-term process, and each developer has a unique working style that the AI must adapt to and support.

2. Democratizing programming to people without backgrounds. Beyond supporting developers, I am deeply passionate about leveraging AI assistants to democratize programming, making it accessible to people without a computer science background. In the digital information age, everyone deserves the opportunity to create and contribute, even simply building web pages for themselves or making smartphone applications.

AI-powered assistants have a great potential to break down the barrier. Unlike professional developers, individuals without a technical background require more fine-grained, step-by-step guidance and explanations. I aim to address this by creating AI assistants capable of self-planning during development—breaking tasks into manageable stages and components, implementing and testing each part individually, and seamlessly integrating them. Since these users provide more requirements but less professional collaboration, the assistants must also deliver clear, easy-to-follow explanations for each step, enabling users to adjust as needed.

3. Vision on methodology. After showing "what" I want to explore, I would like to briefly share my vision of "how". I believe domain knowledge will become increasingly critical as we tackle more complex tasks, and I plan to continue investigating ways to enable AI models to learn and apply this knowledge. Integrating domain knowledge as inference constraints aligns with the utilization of external tools, as external tools provide additional context that can be used to prune the search space of AI models. Meanwhile, designing model architecture and learning objectives should be generalizable and transferable for continuous learning and cheaper customization in the LLM era where building new models becomes increasingly costly. Creating knowledge-rich data is the most promising from my point of view, as lots of human behaviors and working practices during the development are not recorded, finding solutions to record such information or synthesize such data following human practices will greatly boost the power of AI models.

4. Collaboration and funding. I will continue collaborating with researchers in software engineering, artificial intelligence, especially natural language processing, and human-computer interaction since the research on AI assistants for software highly relies on these domains' expertise. I will also pursue collaboration with industrial tech companies, taking their input to keep pushing the build and revising my vision of practical AI assistants.

As a senior PhD, my advisor Professor Lin Tan has involved me in the proposals for the IARPA SoURCE CODE BAA[1] (Securing Our Underlying Resources in Cyber Environments)[2] program and the ARPA-H UPGRADE (Universal Patching and Remediation for Autonomous Defense) program. For both programs, I met and discussed with Professors Lin Tan and Xiangyu Zhang regularly, and contributed technically. For the IARPA program, I drafted initial sections proposing the use of contrastive learning to train LLMs to distinguish source and binary code authorship and analyze malicious attacks. For the ARPA-H program, I wrote initial draft sections about training multi-modal LLMs to understand and align the multiple-formatted knowledge of medical devices such as lifted binary code of the device, textual documentation, and GUI interfaces, which is useful for producing remediation and explanations for zero-day vulnerabilities discovered in the medical system.

For my future funding plan, I plan to apply for the National Science Foundation's funding. At the earlier stage, I will focus on programs such as NSF CAREER and CRII which are for early-career faculty. As my research progresses, I plan to apply to the NSF CISE Core Programs, particularly in the CCF and IIS divisions that align with my research interests in generating secure programs and building human-interactable AI assistants for software development. Besides, I also plan to seek industrial funding through collaborations with tech companies.

[1] https://www.iarpa.gov/newsroom/article/source-code-baa
[2] https://arpa-h.gov/research-and-funding/programs/upgrade

References:

[1] **Nan Jiang**, Xiaopeng Li, Shiqi Wang, Qiang Zhou, Soneya Binta Hossain, Baishakhi Ray, Varun Kumar, Xiaofei Ma, and Anoop Deoras. 2024. LeDex: Training LLMs to Better Self-Debug and Explain Code. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems (NeurIPS '24)*.

[2] **Nan Jiang**, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE '21)*. IEEE Press, 1161–1173.

[3] **Nan Jiang**, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, and Xiangyu Zhang. 2023. KNOD: Domain Knowledge Distilled Tree Decoder for Automated Program Repair. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, 1251–1263.

[4] **Nan Jiang**, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, 1430–1442.

[5] **Nan Jiang,** Chengxiao Wang, Kevin Liu, Xiangzhe Xu, Lin Tan, Xiangyu Zhang, and Petr Babkin. 2024. Nova: Generative Language Models for Assembly Code with Hierarchical Attention and Contrastive Learning. In *The Thirteenth International Conference on Learning Representations (ICLR '25)*

[6] Soneya Binta Hossain, **Nan Jiang**, Qiang Zhou, Xiaopeng Li, Wen-Hao Chiang, Yingjun Lyu, Hoan Nguyen, and Omer Tripp. 2024. A Deep Dive into Large Language Models for Automated Bug Localization and Repair. *Proc. ACM Softw. Eng. 1 (FSE '24)*, Article 66 (July 2024), 23 pages.

[7] Yi Wu, **Nan Jiang**, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How Effective Are Neural Networks for Fixing Security Vulnerabilities? In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23)*. Association for Computing Machinery, New York, NY, USA, 1282–1294.

[8] Danning Xie, Zhuo Zhang, **Nan Jiang**, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. 2024. ReSym: Harnessing LLMs to Recover Variable and Data Structure Symbols from Stripped Binaries. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*.

[9] Xiangzhe Xu, Zhuo Zhang, Zian Su, Ziyang Huang, Shiwei Feng, Yapeng Ye, **Nan Jiang**, Danning Xie, Siyuan Cheng, Lin Tan, and Xiangyu Zhang. 2025. Unleashing the Power of Generative Model in Recovering Variable Names from Stripped Binary. In *Proceedings of the Network and Distributed System Security Symposium 2025 (NDSS '25)*.

[10] Danning Xie, Byungwoo Yoo, **Nan Jiang**, Mijung Kim, Lin Tan, Xiangyu Zhang, and Judy S. Lee. 2023. Impact of Large Language Models on Generating Software Specifications. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '25)*.

[11] Shanchao Liang, **Nan Jiang**, Shangshu Qian, and Lin Tan. 2024. WAFFLE: Multi-Modal Model for Automated Front-End Development. *Under Review*. https://arxiv.org/abs/2410.18362

[12] **Nan Jiang**, Shanchao Liang, Chengxiao Wang, Jiannan Wang, and Lin Tan. 2024. LATTE: Improving Latex Recognition for Tables and Formulae with Iterative Refinement. In *The 39th Annual AAAI Conference on Artificial Intelligence (AAAI '25)*

[13] **Nan Jiang**, Qi Li, Lin Tan, and Tianyi Zhang. 2024. Collu-Bench: A Benchmark for Predicting Language Model Hallucinations in Code. *Under Review*. https://arxiv.org/abs/2410.09997

[14] Shanchao Liang, Yiran Hu, **Nan Jiang**, and Lin Tan. 2024. Can Language Models Replace Programmers? REPOCOD Says 'Not Yet'. *Under Review*. https://arxiv.org/abs/2410.21647

[15] Yi Wu, Zikang Xiong, Yiran Hu, Shreyash S. Iyengar, **Nan Jiang**, Aniket Bera, Lin Tan, and Suresh Jagannathan. 2024. SELP: Generating Safe and Efficient Task Plans for Robot Agents with Large Language Models. In *2025 International Conference on Robotics and Automation (ICRA '25)*