Aaditya Naik: Research Statement

A Neurosymbolic Approach to Software Engineering

Foundation models, particularly Large Language Models (LLMs), have grown increasingly powerful, demonstrating a remarkable aptitude for generating code. These capabilities have led to a surge in AI-assisted development, promising to democratize programming and accelerate development cycles. However, these models still face several fundamental issues, particularly when applied to tasks requiring rigorous planning, reasoning, and structure. One becomes painfully aware of such issues when using these models for software engineering; tasks like bug detection and repair, memory and performance optimizations, and writing code in low-resource languages still elude them. A recent report by Aikido Security [1] revealed that 1 in 5 organizations identified severe vulnerabilities directly tied to AI-generated code. The integration of these models also opens new attack vectors; research has demonstrated remote prompt injection attacks against coding agents like GitLab Duo [3], and Anthropic has highlighted how unconstrained agents can enable malicious software development and cyber-espionage [2].

Over several decades, the software engineering community has developed a rich ecosystem of tools to improve software reliability and developer productivity: static analyzers, fuzzers, symbolic execution, and formal verification methods. Yet, even if models are armed with such tools, they must plan and orchestrate their use and reason over their requirements, outputs, and side effects. These shortcomings accumulate quickly and incur high costs, leading to agents generating code that is difficult to review at best and critically vulnerable at worst.

Neurosymbolic programming promises to provide a solution to these issues by combining the best of the otherwise complementary worlds of deep learning and symbolic reasoning. My research agenda focuses on developing *neurosymbolic systems for safe and reliable AI-assisted software engineering.* I build systems that enforce sound engineering practices, ensure security, and enable rigorous verification within deep learning frameworks. To this end, I have pursued three complementary research directions.

First, I developed *neurosymbolic solutions for software engineering*, designing architectures for relational program representations, such as CodeTrek [ICLR 2022], and to tackle program verification, such as Code2Inv [CAV 2020]. Second, I developed *program synthesis techniques for neurosymbolic agents.* This includes synthesizing program safety properties, such as in Sporq [UIST 2021], and model alignment properties, such as in SQRL [ICML 2023]. Finally, I developed *general-purpose neurosymbolic frameworks* like Dolphin [ICML 2025], which integrates symbolic rules into deep learning architectures at training time, and TorchQL [OOPSLA 2024], which enables analyzing and debugging the outputs of machine learning models.

1 Neurosymbolic Agents for Software Engineering

To apply AI effectively to software engineering, we must move beyond treating programs as mere sequences of tokens. Programs are structured graphs of statements governed by control and data flows. Guided by this insight, my research addresses two fundamental questions: how to effectively represent programs within the context of coding agents, and how to generate verifiable guarantees to serve as feedback for model-generated code. To address these questions, I designed neurosymbolic solutions that integrate deep learning with formal program features, enabling robust program analysis and verification.

Neurosymbolic Program Representation. While LLMs have improved at understanding code, their limited context window remains a major bottleneck for engineering tasks involving large repositories. Current agents are woefully underpowered when identifying relevant context, often "overthinking" and wasting tokens.

In CodeTrek [ICLR 2022], I leveraged relational representations of code to extract relevant context for tasks such as bug detection and type inference. CodeTrek builds a relational database for a codebase to obtain a graphical representation. It then samples "walks" over this graph, reaching deeply nested dependencies spanning several files without the computational cost of exhaustively enumerating paths. These walks are encoded into an embedding space to guide coding agents. CodeTrek achieved up to **19%** points higher accuracy than leading LLMs on real software engineering tasks, including variable shadowing and exception prediction. It demonstrated that relational embeddings allow models to efficiently piece together meaningful context from large repositories, a critical step for scalable software analysis.

Neurosymbolic Program Verification. Even with robust representations, coding agents still lack the ability to effectively evaluate the soundness and correctness of their work until they complete their task end-to-end. Formal reasoning tools can act as an external source of feedback that can harden these agents. I demonstrated the applicability of such feedback in Code2Inv [CAV 2020], an RL-based framework for program verification.

Code2Inv formulates verification as a multi-step decision process. At each step, the RL agent iteratively constructs a proof artifact that is fed to a symbolic checker, whose feedback functions as the reward for the agent. However, standard rewards in such environments are extremely sparse, as most candidate proofs fail. To address this, I developed a mechanism that interprets counterexamples from the checker as a dense feedback signal, guiding the agent toward valid solutions step-by-step. This approach allowed Code2Inv to verify complex programs across different representations, including C programs and Constrained Horn Clauses (CHCs), outperforming state-of-the-art verifiers. Furthermore, the flexibility of this RL framework enabled the verification of programs requiring non-linear arithmetic, a domain that typically eludes conventional tools.

2 Program Synthesis for Neurosymbolic Agents

Beyond code generation, coding agents require specifications for program safety and alignment. Program safety specifications are typically hard constraints that analyze and flag programs violating safety standards, enterprise policy, and other best practices. Alignment specifications feature in programs that call deep neural models to ensure their alignment with data distribution and human intuition, and are often statistical in nature. My research in this area aims to address *how to derive these specifications* in a scalable manner.

Synthesizing Program Safety Properties. General-purpose models often struggle to catch repository-specific bugs, such as API misuse or violations of project-specific engineering standards. Detecting these patterns reliably requires custom static checkers, but writing them is non-trivial. I developed Sporq [UIST 2021], a VSCode extension that interactively synthesizes static checkers. When a developer highlights a buggy code snippet, Sporq utilizes active learning and efficient Datalog synthesis techniques I developed (GenSynth [AAAI 2021], EGS [PLDI 2021], Libra [VLDB 2024]) to generate a checker that identifies similar patterns across the repository.

Sporq operates on relational representations of code and offers transparency by showing the synthesized checker to the developer for further refining. In practice, Sporq allowed developers to synthesize checkers for complex patterns, like mutual recursion, unused variables, and unsafe function usage (defined in CWEs), in less than **10 rounds of interaction** on real repositories such as GNU coreutils and MySQL connectors.

Synthesizing Model Alignment Properties. Software engineering for AI models presents unique challenges; standard metrics aggregate failures, thus hiding rare yet critical vulnerabilities. To address this, I developed SQRL [ICML 2023], a technique to synthesize interpretable rules for testing and fixing model alignment. SQRL synthesizes *statistical quantile rules* from unstructured data that capture domain-specific constraints, such as physical dimensions or medical bounds. For instance, SQRL correctly captured medically established bounds for the *Glasgow Coma Score*, which were repeatedly violated by imputation models. Crucially, these synthesized rules serve as executable specifications. I demonstrated that using SQRL rules as feedback during test-time adaptation improved model performance on medical data imputation by **32%** and reduced rule violations by approximately **69%**.

3 General-Purpose Neurosymbolic Frameworks

Implementing the verification and synthesis tools described above requires robust infrastructure. Existing neurosymbolic frameworks were unable to scale to the complexity of modern software engineering tasks, often failing to support blackbox functions (like external linters and checkers) or slowing down trying to solve complex logical constraints. I developed general-purpose neurosymbolic frameworks specifically designed to support the rigorous requirements of engineering software with AI.

Integrating Neural Models with Symbolic Constraints. To build agents that respect syntax, traffic laws, or physics, we must incorporate symbolic reasoning into the training loop. However, prior frameworks like Scallop or Logic Tensor Networks failed to scale, often timing out on simple pathfinding tasks or running out of memory. I developed Dolphin [ICML 2025], a GPU-enabled framework that unifies discrete symbols (Python objects) and neural outputs (PyTorch tensors). Dolphin supports recursions, control flow, and, critically for SE agents, blackbox Python functions. This allows developers to integrate external tools, such as code checkers and simulators, directly into the model's training process. Dolphin scales to complex tasks over images and text, converging up to **62x** faster than sampling-driven approaches on hard pathfinding tasks that were previously untenable for other frameworks.

Testing Neural Models with Symbolic Constraints. Debugging machine learning models is an essential software engineering activity. Yet, developers often rely on ad-hoc scripts to find failure modes. I formalized this process as checking *violations of integrity constraints* and developed TorchQL [OOPSLA 2024] to enable it. TorchQL provides a database-like abstraction for specifying complex constraints over model outputs. I designed a persistent, optimized execution engine that allows TorchQL to evaluate these constraints efficiently. It achieves up to **13x** faster query execution than Pandas while being significantly more concise. I used TorchQL to rigorously test frontier LLMs, specifying constraints that captured high-value failures. By filtering outputs based on these integrity constraints, I improved LLM performance on datasets like GSM8K by **16%** points, demonstrating that SE-style constraint checking is vital for reliable AI deployment.

4 Future Research Directions

My prior research has established the foundations for *scalable neurosymbolic learning* and *relational representations of repositories* to bridge deep learning with structured reasoning. However, a significant gap remains between academic benchmarks and the reality of software engineering. While LLMs achieve high scores on benchmarks like SWE-Bench, these metrics often fail to capture the diversity of real-world tasks.

In reality, software engineering tasks involve proprietary repositories in enterprises, legacy systems written in languages like COBOL, or adversarial environments where code could be obfuscated to sidestep guardrails. The failure of current approaches stems from a fundamental disconnect: agents operate as probabilistic text generators without reliably conforming to the rigid formal semantics that govern software correctness. Standard vector embeddings lack the fidelity required to represent the intricate dependencies of enterprise software.

I intend to bridge this gap by exploring the role of **neurosymbolic programming for reliable coding agents**, investigating how sound engineering principles can be effectively used to guide off-the-shelf models.

Neurosymbolic Representations of Programs. Current agents struggle to leverage structural and semantic contexts, often resorting to retrieval-augmented generation techniques (RAG) that index code as text to find dependency chains across files. Embeddings get "thrown off" by obfuscated logic, variable renaming, or scattered dependencies. This results in token waste and hallucinated complexity. I aim to investigate how existing software engineering tools, such as static analyzers, linters, and formal verifiers, can be used to construct modular neurosymbolic representations that can be embedded in the agent's context.

Drawing on the relational embedding techniques I pioneered in **CodeTrek** [ICLR 2022], I plan to explore architectures that capture the rigorous structure of large-scale code using complex formal artifacts like inter-procedural data flow analysis, pointer analysis, and call graphs. This approach advances beyond current IDE assistants by providing models with structurally sound, semantically rich context that allows agents to reliably "reason" about program states.

Neurosymbolic Validation of Programs. Models are being increasingly integrated into security-critical workflows. Code generated by agents must therefore be checked not just against publicly studied CVEs and CWEs, but also against guidelines, security threats, and

API misuse specific to organizations and their repositories. I see significant potential in pairing generation with task-specific, ephemeral checkers.

Adapting the synthesis methods from **Sporq** [UIST 2021], I aim to research how agents can generate their own lightweight validation tools to validate subgoals on the fly. This direction promises to mitigate broken API calls and program bloat while enforcing the rigid safety checks required for trustworthy, transparent AI-assisted development.

Neurosymbolic Planning for Long-Horizon Tasks. A core component of coding agents is the "planner", whose responsibility is to orchestrate different moving parts to realize a solution. Feedback for plans regarding complex scenarios, such as patching vulnerabilities across multiple files, is extremely sparse. As a result, current agents struggle to self-correct because they lack a mechanism for modular reasoning.

I intend to explore frameworks that leverage intermediate symbolic feedback to modularize and drive planning. My goal is to investigate how the reward-shaping paradigms I established in **Code2Inv** [CAV 2020], where counterexamples from a symbolic checker guided the policy network, can be generalized to full-scale software generation. This involves researching how feedback from fuzzers and symbolic execution engines can be utilized to critique partial code plans effectively. To make this computationally feasible, I plan to build upon the GPU-accelerated infrastructure of **Dolphin** [ICML 2025] to compute symbolic rewards and their gradients efficiently. Ultimately, this line of inquiry seeks to condition models to prioritize parsimony and maintainability, moving agents beyond maximizing token output toward planning efficient, secure architectures.

References

[1] Aikido Security. *State of AI Security in Development 2026.* Accessed: 2024. 2024. URL: `https://www.aikido.dev/state-of-ai-security-development-2026`.

[2] Anthropic. *Disrupting AI Espionage.* Accessed: 2024. 2024. URL: `https://www.anthropic.com/news/disrupting-AI-espionage`.

[3] Legit Security. *Remote Prompt Injection in GitLab Duo.* Accessed: 2024. 2024. URL: `https://www.legitsecurity.com/blog/remote-prompt-injection-in-gitlab-duo`.

[AAAI 2021] Jonathan Mendelson et al. "GENSYNTH: Synthesizing Datalog Programs without Language Bias". In: *Proceedings of the AAAI Conference on Artificial Intelligence.* 2021.

[ICML 2023] Aaditya Naik et al. "Do Machine Learning Models Learn Statistical Rules Inferred from Data?" In: *International Conference on Machine Learning.* 2023.

[ICML 2025] Aaditya Naik et al. "DOLPHIN: A Programmable Framework for Scalable Neurosymbolic Learning". In: *Forty-second International Conference on Machine Learning.* 2025.

[VLDB 2024] Aaditya Naik et al. "Relational Query Synthesis $\bowtie$ Decision Tree Learning". In: *Proceedings of the VLDB Endowment, Volume 17, Issue 2.* 2024.

[UIST 2021] Aaditya Naik et al. “Sporq: An interactive environment for exploring code using query-by-example”. In: *The 34th Annual ACM Symposium on User Interface Software and Technology*. 2021.

[OOPSLA 2024] Aaditya Naik et al. “Torchql: A programming framework for integrity constraints in machine learning”. In: *Proceedings of the ACM on Programming Languages 8 (OOPSLA1)*. 2024.

[ICLR 2022] Pardis Pashakhanloo et al. “Codetrek: Flexible modeling of code using an extensible relational representation”. In: *International Conference on Learning Representations*. 2022.

[CAV 2020] Xujie Si et al. “Code2inv: A deep learning framework for program verification”. In: *International Conference on Computer Aided Verification*. 2020.

[PLDI 2021] Aalok Thakkar et al. “Example-guided synthesis of relational queries”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021.