Research Statement | Jian Wang

Software is the invisible infrastructure of modern civilization. It powers financial systems, healthcare platforms, transportation networks, and the communication fabric of daily life. Yet the way software is written is undergoing its most disruptive transformationin decades. AI-assisted coding tools — GitHub Copilot, Cursor, Claude Code — have increased developer productivity by 30–42%, enabling individuals to produce in hoursnwhat once took teams weeks. But this acceleration introduces a fundamental paradox: the faster we generate code, the less we understand what we have generated, and the harder it becomes to trust it.

My research goal is to make AI-assisted programming trustworthy. By leveraging the semantic nature of program execution and the vast availability of open-source software history, I aim to achieve this goal through two complementary directions:*detecting* unreliable AI-generated code before it enters production, and *repairing* bugs automatically once they appear — closing the loop from code generation to code assurance. As LLM-powered agents increasingly operate autonomously over entire codebases and supply chains, I extend this vision to the full attack surface of LLM infrastructure itself.

My research is situated at the intersection of Software Engineering, Large Language Models, and Trustworthy AI Systems. From a foundational perspective, I employ data-driven learning and program analysis as the core driving force, build benchmarks and evaluation frameworks grounded in real-world software artifacts, and apply these toward systems that must be provably reliable — from safety-critical C/C++ software to enterprise LLM deployments. All three components — empirical methods, semantic frameworks, and security pipelines — are systematically considered and jointly designed.My key research directions are as follows:

> **Reliable AI-Assisted Programming**: automated detection of AI-generated code and LLM-based program repair, grounded in large-scale real-world bug benchmarks.
>
> **Semantically Grounded Program Understanding**: incorporating dynamic execution traces into LLM training and inference so models reason about runtime behavior,not just surface syntax.
>
> **LLM Supply Chain Security**: end-to-end vulnerability discovery for LLM-integrated systems — from dependency governance to automated penetration testing across orchestration frameworks and agent protocols.

Reliable AI-Assisted Programming | Detecting and Fixing Unreliable Code at Scale

AI coding tools generate code at unprecedented speed, but multiple studies demonstrate that LLMs frequently produce defects that are not immediately apparent alongside raising concerns about copyright and licensing. Organizations face an expanding paradox:initial productivity gains are offset by an increased maintenance burden. How do we establish trust in code we did not fully write?

This question decomposes into two sub-problems. First, how do we *identify* whether a given piece of code is AI-generated and potentially **unreliable**? Second, once defects are identified, how do we *repair* them automatically?

For detection, I conducted the first comprehensive empirical study evaluating thirteen AIGC detectors across a large-scale dataset of 2.24 million samples — 1.08 million code-related and 1.16 million natural language. The study revealed that existing detectors suffer significant accuracy degradation on code compared to natural text.Building on this diagnosis, I developed fine-tuning-based detection methods that achieved improvements of up to 23.5% on code snippets and 15.8% on documentation.*(ASE 2024)*

For repair, the dominant benchmark in automated program repair has long been Defects4J— a Java-only dataset. C/C++ systems, which underpin critical infrastructure from operating systems to embedded devices, had no equivalent. I constructed **Defects4C**,the first large-scale executable C/C++ bug benchmark, by mining 38 million commits across 500 top GitHub repositories and 14.5K CVE-linked commits via BigQuery and GitHub API, ultimately producing 350 expert-validated bugs through three rounds of human review. Using this foundation, I conducted the first large-scale evaluation of 24state-of-the-art LLMs on C/C++ repair under both single-round and conversation-based settings, analyzing parallel and sequential test-time scaling strategies. The results demonstrate that statistical learning alone is insufficient — performance significantly lags behind Java benchmarks, motivating the semantic work below. *(ASE 2025, ISSRE 2024)*

Semantically Grounded Program Understanding | Reasoning About Behavior, Not Syntax

Code LLMs like Claude Code and Copilot have transformed programming workflows, but their core architecture has not changed: they are trained to predict the next token. A model that predicts tokens well can write syntactically plausible code that is semantically broken — it looks right but does wrong. I asked the question: *Can we teach LLMs to reason about what code does at runtime, not just what it looks like on the page?*

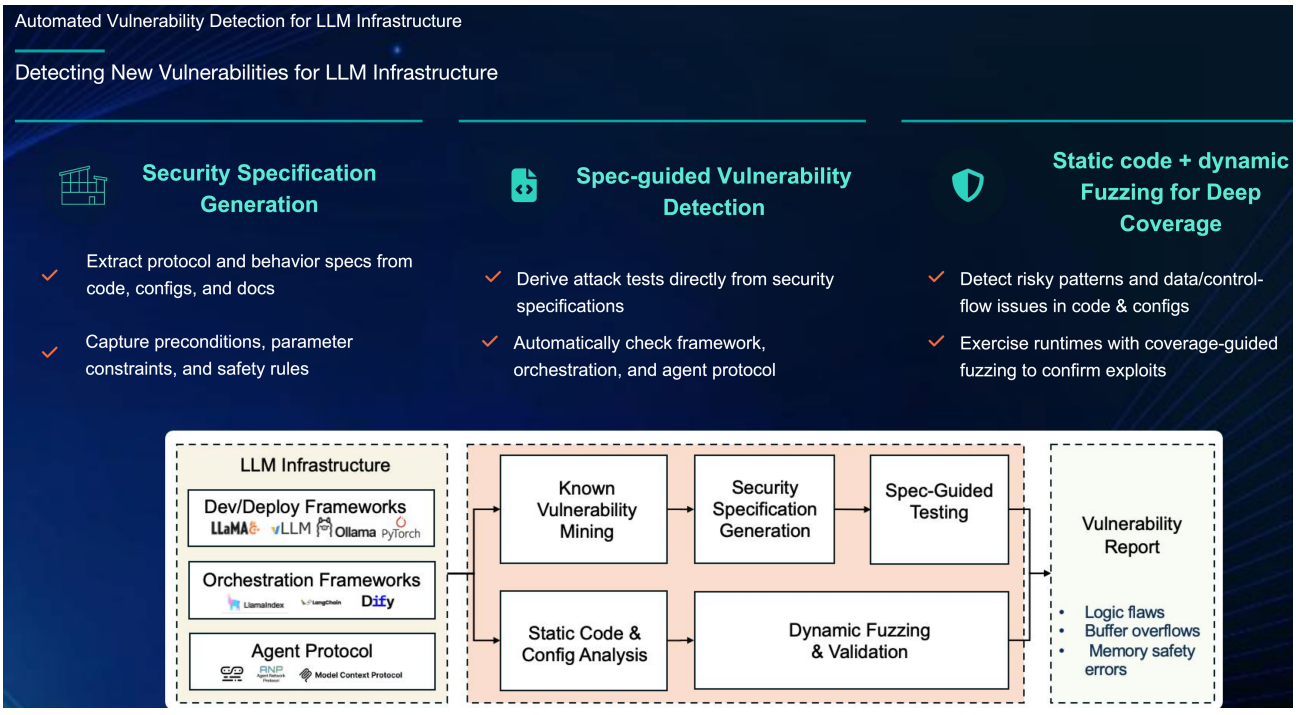
Yes. The key insight is that dynamic execution traces — the sequence of variable states,branch decisions, and memory operations that occur when code actually runs —

carry semantic information that source text alone cannot express. A bug that is invisible in source code often becomes obvious in the trace: the variable that should be 0 is 7, the branch that should be taken is skipped.

I developed a semantic enhancement framework that incorporates dynamic execution trace representations into both LLM training and inference. Rather than fine-tuning on source-to-patch pairs, the model is trained on trace-augmented inputs that expose runtime behavior. Extensive empirical evaluation demonstrates that this semantic-informed approach consistently improves patch correctness by an average of 12.3% and functional correctness by 8.7%. The framework supports parameter-efficient fine-tuning (PEFT on sub-7B models), enabling practical deployment without prohibitive compute cost.*(EMNLP 2025)*

This work establishes a new principle for code LLM improvement: the path to better reasoning is not more source code, but richer representations of program *behavior*.

LLM Supply Chain Security | A Closed-Loop Defense for LLM-Integrated Systems



The previous two directions address the code LLMs produce. This direction addresses the LLMs themselves as attack targets. As agentic systems like Claude Code operate autonomously over code bases, call external APIs, and integrate with orchestration frameworks (LangChain, LlamaIndex) and agent protocols (MCP, ANP), the attack surface of a modern software system has expanded from its source code to its entire AI supply chain.

I asked the question: *How do we systematically discover and defend against vulnerabilities that arise not from code bugs, but from how LLMs are integrated, deployed, and composed in real systems? I frame this as a system-security problem across the LLM supply chain, where risk can come from hidden dependencies, multi-vendor model usage, and complex application wiring—not just traditional implementation defects such as evalplus, defects4j Orphan bugs, incontrast, supply chain vulberalbity only activate when its context match.*

First, **supply chain transparency**: many organizations cannot precisely enumerate which models, frameworks, and dependencies their LLM applications actually rely on. To solve this, I built **AI-BOM (AI Bill of Materials)**, a static analysis system that extracts end-to-end LLM call chains using BFS traversal, identifies multi-vendor model usage (e.g., different model providers and hubs), and flags vulnerable or outdated dependencies—**without executing the project**. It supports 100+ frameworks and achieves >99% identification precision (SMU Enterprise Governance Solution).

Second, **active vulnerability discovery and runtime defense**: knowing what exists is not enough—we must determine what is exploitable and then convert findings into protection. I developed an automated pentesting pipeline that traces entry-to-sink call chains from user prompts to backend actions, adapts known PoCs to app-specific interfaces, and uses feedback-guided iterative mutation until it finds a reproducible exploit; for LLM infrastructure, it derives security specifications from code/configs and generates attack tests across common inference stacks and agent protocols. Finally, the system turns discovered attack patterns into a lightweight **Guardrail Model** fine-tuned on those artifacts for real-time input/output monitoring, closing the loop: **discover → attack → defend → learn → repeat**.

Future Directions

The convergence of AI code generation, autonomous agents, and critical infrastructure creates an urgent and open research landscape.

Verified AI-Assisted Development. Current repair methods generate patches and validate them against test suites. Test suites are incomplete. I plan to integrate formal verification guarantees — lightweight symbolic execution or SMT-based property checking — into the repair loop, so that patches can be certified correct with respect to explicit safety properties, not just passing tests.

Adaptive Detection Under Adversarial Evolution. As generation models improve, the gap between human and AI code narrows. Detection methods must evolve from pattern-based approaches toward execution-based signals that remain meaningful even as surface distributions shift. I plan to build adaptive detection frameworks that leverage runtime behavior as a generation-invariant signal.

Neurosymbolic Architectures for Semantic Repair. Bridging the semantic gap fully requires moving beyond pure neural approaches. I plan to explore hybrid neuro symbolic architectures where a formal program analysis layer provides verifiable structural facts — data flow, reachability, type constraints — and an LLMlayer interprets business logic and intent. The two layers inform each other iteratively,combining the precision of formal methods with the flexibility of learned reasoning.

LLM Infrastructure Security at Scale. As MCP and agent protocols become standardized, the attack surface becomes both larger and more uniform. I plan to build systematic, protocol-aware security analysis that can reason about multi-agent interactions, tool invocation chains, and cross-service data flows — the supply chain security problem for the agentic era.

The ultimate goal is a trustworthy AI development ecosystem: one where developers can build faster *and* safer, where reliability is not sacrificed for velocity, and where the systems that generate and deploy code are themselves understood, audited, and defended.