Loomline Technical Report

Loomline Team

https://github.com/valkor-ai/loom

Abstract

Recent advances in large language models (LLMs) have fundamentally transformed software development by enabling machines to generate code, understand complex repositories, perform debugging, and automate various programming tasks. The emergence of AI coding assistants and autonomous coding agents has significantly reduced the cost of software creation, while Vibe Coding has further lowered the barrier for users to build applications through natural language interactions. However, the rapid improvement in code generation capability does not directly translate into the ability to deliver production-ready software. The central challenge of modern software engineering is shifting from generating large volumes of code to ensuring the quality, reliability, maintainability, and deployability of AI-produced software systems.

In this project, we present **LOOMLINE**, an end-to-end AI-driven software delivery platform designed to bridge the gap between inexpensive code generation and costly software engineering processes. Unlike existing coding assistants that primarily focus on implementation, Loomline treats software delivery as a lifecycle-wide engineering problem and provides continuous quality assurance throughout requirements analysis, architecture design, implementation, testing, code review, security remediation, deployment, maintenance, and system evolution. By embedding quality governance into every development stage, **LOOMLINE** enables AI agents to produce software artifacts that are not only functionally correct but also trustworthy, secure, and aligned with long-term maintainability.

Furthermore, **LOOMLINE** transforms natural-language user requirements into fully executable and deployable software products through a unified delivery workflow. The platform integrates requirement understanding, system planning, implementation coordination, automated validation, and deployment into a coherent pipeline, enabling AI agents to move beyond isolated code generation toward autonomous software delivery. We envision **LOOMLINE** as a new generation of AI software engineering infrastructure where the ultimate objective is no longer merely generating code, but continuously delivering high-quality, verifiable, recoverable, and production-ready software systems.

1 Introduction

Over the past few years, advances in **Large Language Models** (LLMs) have fundamentally transformed the way software is created. The emergence of foundation models such as GPT, Claude, Gemini, Qwen, DeepSeek and other state-of-the-art LLMs has significantly reduced the cost of software production. These models are trained on vast corpora of source code, technical documentation, and human knowledge, enabling them to acquire increasingly sophisticated programming capabilities. Recent research on LLMs on a wide range of benchmarks demonstrates that frontier models can solve complex coding tasks, generate production-quality code, perform repository-level reasoning, and collaborate with developers across the software development lifecycle.

This progress has fueled a new wave of AI coding development. Coding assistants are evolving into autonomous coding agents capable of planning tasks, writing code, invoking tools, running tests, and interacting with development environments. Meanwhile, Vibe Coding has introduced an even more accessible model of software creation, where users describe desired functionality in natural language and AI systems generate applications with minimal human intervention. **However, despite rapid advances in code generation capabilities, critical gaps remains between generating high quality code and delivering production-ready software.** Inparticular:

- **From Code Generation to Deliverable Software Products.** Existing technologies such as AI Coding assistants and Vibe Coding platforms have dramatically increased the volume of generated code and accelerated the creation of software prototypes. However, generating code is only a small part of software delivery. Real-world software projects require requirement clarification, architectural planning, task decomposition, implementation coordination, validation, debugging, deployment, monitoring, and operational handoff. Consequently, the next stage of competition in AI Coding is not simply about which model can write more code or achieve higher benchmark scores. Instead, it is about who can provide users with a complete software delivery environment in which coding agents can produce artifacts that are *executable, verifiable, recoverable, and deployable*. The ultimate goal is not code generation, but the autonomous delivery of production-ready software systems.
- **From Code Quantity to Software Quality.** While Vibe Coding lowers the barrier to software creation, it also leads to an unprecedented explosion in generated code, prototypes, and applications. Yet the fundamental challenge of software engineering has never been merely writing code. The true challenge lies in ensuring that software systems remain correct, maintainable, secure, scalable, and aligned with evolving requirements. Current LLM-based coding systems often struggle to guarantee consistent software quality across the entire engineering lifecycle. Requirements may be ambiguous or incomplete; architectural decisions may drift during long-running tasks; generated implementations may introduce hidden defects; testing may be insufficient; and maintenance knowledge may be lost across iterations. As AI-generated software becomes increasingly prevalent, the industry requires mechanisms that continuously govern and improve software quality throughout requirements engineering, architecture design, implementation, testing, deployment, and maintenance. Without such mechanisms, faster code generation may simply accelerate the production of unreliable software at scale.

To effectively address the aforementioned challenges in the AI coding era, we proposed our **LOOMLINE** project. At its core, Loomline is built upon a simple observation: *code generation is becoming increasingly inexpensive, while delivering maintainable, verifiable, and deployable software remains expensive*. Recent advances in foundation models have dramatically reduced the cost of producing code, but they have not eliminated the complexity of software engineering. The gap between generating code and delivering production-ready software remains one of the most significant bottlenecks preventing AI agents from becoming reliable software developers. **LOOMLINE** is designed with two key characteristics:

- **Software Engineering Lifecycle Code Quality Assurance. LOOMLINE** provides a comprehensive quality assurance framework that spans the entire software engineering lifecycle. Rather than focusing solely on code generation, **LOOMLINE** continuously safeguards software quality from requirements analysis, software design, implementation, testing, code review, vulnerability remediation, deployment, maintenance, and system evolution. By integrating quality governance into every stage of the development process, **LOOMLINE** enables AI agents to generate software artifacts that are not only functionally correct, but also trustworthy, maintainable, secure, and aligned with architectural intent. The platform continuously validates requirements completeness, preserves design consistency, enforces engineering best practices, detects defects and security vulnerabilities, and supports automated repair and evolution throughout the software lifecycle.

- **End-to-End Deliverable Software Products.** LOOMLINE transforms natural-language user requirements into fully deployable software products through an integrated end-to-end software delivery workflow. Rather than focusing on producing isolated code snippets or prototypes, LOOMLINE is designed to generate complete, execution-ready software systems that directly address user needs. By bridging requirement understanding, system design, implementation, testing, and deployment within a unified pipeline, LOOMLINE ensures that the output of AI-driven development is not merely code artifacts, but production-grade software that can be immediately delivered and operated in real-world environments.

2 Techniques

2.1 Code Generation

LLM-based code generation has evolved from generating isolated code snippets to producing project-aware software artifacts. For autonomous software delivery, the key challenge is generating code that fits an existing repository, its type definitions, dependencies, coding conventions, and surrounding implementation context.

A central line of work studies repository-level and context-aware generation. CATCODER Pan et al. (2024b) investigates repository-level code generation by retrieving relevant code and type context, showing that the quality of generated code depends heavily on whether the model receives the right project context. Code search based code suggestion Chen et al. (2024) provides another perspective: rather than treating generation as a purely parametric capability of the model, it uses relevant code examples as grounding information for producing more useful suggestions.

This direction is important for Loomline because a production-oriented coding agent must operate inside a real project, not in a benchmark sandbox. It needs to find relevant files, understand local APIs, reuse existing implementation patterns, and generate code that can be integrated into the repository. Generation is therefore closely connected to retrieval, program context modeling, and execution feedback. Work such as SelfPiCo Xue et al. (2024) also suggests that partial code execution can help LLMs reason about code behavior during generation, moving generation closer to a verifiable engineering process.

2.2 Evaluation

As LLM-generated code becomes more widely used, evaluation becomes a first-class technical problem. Traditional code-generation benchmarks often measure whether a model can solve a standalone programming task, but software delivery requires broader evidence. The generated software should be correct, robust to requirement variation, consistent with runtime behavior, and realistic enough to reflect actual development scenarios.

The recent work Hu et al. (2025) systematically assesses and advances benchmarks for evaluating LLMs in software engineering tasks. RealisticCodeBench Yu et al. (2025) further argues that code generation should be evaluated under more realistic settings, rather than only through simplified benchmark tasks. Complementing this, NLPerturbator Chen et al. (2026) studies how robust code LLMs are to natural-language variations, showing that small changes in requirement descriptions can affect model behavior. Another study Chen et al. (2025a) investigates whether LLMs can reason about the runtime behavior of programs, which is crucial because passing static prompts is not equivalent to understanding how software executes.

Loomline needs evaluation mechanisms that assess not only generated code, but also requirement robustness, runtime behavior, project integration, and the reliability of intermediate development decisions. Evaluation acts as the bridge between generation and quality assurance.

2.3 Testing

For AI-generated software, testing is especially important because generated code may appear correct while hiding subtle functional errors, boundary-case failures, or integration problems. Testing serves as a practical mechanism for converting plausible code into verifiable software artifacts. The recent work Zhang et al. (2026) generates unit tests via chain-of-thought prompting and reinforcement learning from coverage feedback shows how LLMs can be guided by explicit reasoning and coverage signals. Multi-stage generation of Rust unit tests Chen et al. (2025b) studies how LLMs can generate tests in a language with strict safety and ownership constraints. Other work Yu et al. (2024) investigates practitioners' expectations on automated test generation, emphasizing that useful test generation should not only improve coverage, but also produce understandable, maintainable, and developer-acceptable

tests. The study of data quality for unit test generation Zhang et al. (2025) further shows that better training or prompting data can significantly affect the quality of generated tests.

Testing is also necessary after software changes. Automated unit test refactoring Gao et al. (2025) addresses the problem that tests must evolve together with production code. An autonomous coding agent should preserve and adapt test assets throughout the software lifecycle. In this sense, testing links generation, evaluation, and evolution into a continuous feedback loop.

2.4 Evolution

Software evolution is a necessary part of LLM-based software engineering, especially for long-running autonomous agents. The works of unit test migration Gao et al. (2024) and unit test refactoring Gao et al. (2025) study how test assets can be adapted instead of discarded. These works suggest that quality assurance must evolve together with the system. If an AI agent changes production code but fails to update tests, documentation, or related artifacts, the project may quickly become inconsistent. The work for the automated Stack Overflow post updating studies how developer-facing knowledge can be kept up to date Mai et al. (2025). These studies show that software evolution requires tracking relationships among code, tests, patches, and external knowledge sources.

For Loomline, evolution techniques are important because AI-generated software must remain maintainable after initial delivery. A coding agent should be able to understand how a change affects surrounding artifacts, update related tests or documentation, and preserve consistency across future iterations.

2.5 Security

LLM-generated code may introduce vulnerabilities, rely on unsafe APIs, hallucinate packages, or produce patches that appear correct but fail to remove the underlying risk. Therefore, security techniques must be integrated into the AI coding lifecycle rather than applied only after generation. The work of explainable vulnerability detection with LLMs Mao et al. (2025) studies how LLMs can be used not only to detect vulnerabilities, but also to provide interpretable evidence. Safe4U Li et al. (2025) focuses on identifying unsound safe encapsulations of unsafe calls in Rust using LLMs, which is particularly relevant to safe systems programming.

Security also requires assessment and remediation. Practical vulnerability assessment Pan et al. (2024a) investigates how vulnerability assessment can be automated in a way that better supports real-world usage. Automated vulnerability patch localization Zhang et al. (2024) identifies where patches should be applied. HFUZZER Zhao et al. (2025) expands the security perspective to the LLM ecosystem itself by testing LLMs for package hallucinations. This is especially relevant for AI coding agents, because hallucinated dependencies can become supply-chain risks. Loomline should detect vulnerable code, explain the evidence, assess practical impact, localize the fix, validate the patch, and monitor similar code to prevent recurrence. This makes security a core part of deliverable software rather than an optional post-processing step.

3 Long-Term Vision

As AI coding technologies continue to evolve, the industry is rapidly shifting from code generation toward autonomous software delivery. We believe that the next generation of AI-native software engineering platforms will be defined not by model capability alone, but by their ability to provide a trustworthy execution environment in which software delivery agents can reliably plan, execute, verify, repair, and deploy software systems. To realize this vision, **LOOMLINE** will continue to evolve along several strategic directions:

- **Human-Aware Autonomy: Reducing Attention Overhead.** Current coding agents often require continuous human supervision, frequent confirmations, and repeated interactions to progress through complex tasks. As software projects become larger and more autonomous, such interaction patterns become a significant bottleneck. **LOOMLINE** aims to minimize unnecessary human involvement while preserving human control over critical decisions. Future versions of **LOOMLINE** will further concentrate human participation on requirement validation, architectural reviews, risk assessment, and delivery approval gates, while routine execution and workflow progression are autonomously managed by software delivery agents.
- **Durable State and Recovery.** Software delivery is inherently long-running and interruption-prone. Existing coding agents frequently lose progress when conversations expire, contexts are compressed, or external tools fail. **LOOMLINE** will continue investing in durable state

management through persistent project memory, execution checkpoints, artifact references, workflow states, and recovery semantics.

- **Auditability and Observability** Future versions of LOOMLINE will provide comprehensive delivery observability through structured execution records, artifact lineage tracking, verification reports, review histories, and decision logs. Every significant action performed by an agent will generate auditable evidence that can be inspected, validated, and reproduced.
- **Contract-Driven Software Development.** Long-running software projects frequently suffer from scope expansion, architectural drift, inconsistent requirements, and uncontrolled modifications. To address these challenges, LOOMLINE will further develop a contract-driven software development framework that formalizes requirements, architecture decisions, task plans, acceptance criteria, and verification objectives into machine-executable contracts. These contracts serve as governance boundaries that guide agent behavior throughout the development lifecycle.
- **Dedecouples Implementation from Validation.** This separation establishes an objective quality-control mechanism that improves transparency, accountability, and reliability in AI-driven software development. By decoupling implementation from acceptance validation, Loomline reduces the risk of self-certification and ensures that software artifacts are evaluated against independently verifiable standards before delivery.
- **Model-Independent Infrastructure.** Loomline is committed to remaining model-agnostic and agent-neutral. Future development will focus on standardized delivery protocols, workflow abstractions, execution semantics, and interoperability layers that allow diverse agents and tools to participate in the same software delivery environment. The long-term value of Loomline lies not in dependence on any single model provider, but in providing durable delivery infrastructure across a multi-agent ecosystem.
- **Deployment-Oriented Workflows.** Many existing AI coding tools stop at source code generation or local demonstrations. However, software value is only realized when systems are successfully deployed and operated. Future LOOMLINE releases will deepen integration with deployment environments, runtime infrastructure, health monitoring systems, logging frameworks, and operational diagnostics. Software delivery workflows will extend beyond implementation to include deployment validation, runtime verification, incident diagnosis, and operational maintenance.

Our long-term vision is to establish LOOMLINE as the foundational infrastructure layer for AI era software engineering. In this future, autonomous software delivery agents will not simply generate code; they will understand requirements, preserve architectural intent, coordinate development activities, verify outcomes, recover from failures, deploy software systems, and continuously evolve production applications. LOOMLINE will provide the execution environment, governance framework, memory system, and delivery protocols that make this vision practical. By transforming software delivery into an observable, recoverable, verifiable, and deployable process, Loomline aims to become the operating system for the next generation of AI-powered software development.

References

Junkai Chen, Xing Hu, Zhenhao Li, Cuiyun Gao, Xin Xia, and David Lo. Code search is all you need? improving code suggestions with code search. In *Proceedings of the IEEE/ACM 46th international conference on software engineering*, pp. 1–13, 2024.

Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. Reasoning runtime behavior of a program with llm: How far are we? In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 1869–1881. IEEE, 2025a.

Junkai Chen, Zhenhao Li, Xing Hu, and Xin Xia. Nlperturbator: Studying the robustness of code llms to natural language variations. *ACM Transactions on Software Engineering and Methodology*, 35(4):1–20, 2026.

Yongqian Chen, Junwei Zhang, Xing Hu, and Xin Xia. Multi-stage generation of rust unit tests with llms. In *2025 32nd Asia-Pacific Software Engineering Conference (APSEC)*, pp. 431–442. IEEE, 2025b.

Yi Gao, Xing Hu, Tongtong Xu, Xin Xia, David Lo, and Xiaohu Yang. Mut: Human-in-the-loop unit test migration. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–12, 2024.

Yi Gao, Xing Hu, Xiaohu Yang, and Xin Xia. Automated unit test refactoring. *Proceedings of the ACM on Software Engineering*, 2(FSE):713–733, 2025.

Xing Hu, Feifei Niu, Junkai Chen, Xin Zhou, Junwei Zhang, Junda He, Xin Xia, and David Lo. Assessing and advancing benchmarks for evaluating large language models in software engineering tasks. *ACM Transactions on Software Engineering and Methodology*, 2025.

Huan Li, Bei Wang, Xing Hu, and Xin Xia. Safe4u: Identifying unsound safe encapsulations of unsafe calls in rust using llms. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):457–480, 2025.

Yubo Mai, Zhipeng Gao, Haoye Wang, Tingting Bi, Xing Hu, Xin Xia, and Jianling Sun. Towards better answers: Automated stack overflow post updating. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 591–603. IEEE, 2025.

Qiheng Mao, Zhenhao Li, Xing Hu, Kui Liu, Xin Xia, and Jianling Sun. Towards explainable vulnerability detection with large language models. *IEEE Transactions on Software Engineering*, 2025.

Shengyi Pan, Lingfeng Bao, Jiayuan Zhou, Xing Hu, Xin Xia, and Shanping Li. Towards more practical automation of vulnerability assessment. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–13, 2024a.

Zhiyuan Pan, Xing Hu, Xin Xia, and Xiaohu Yang. Catcoder: Repository-level code generation with relevant code and type context. *ACM Transactions on Software Engineering and Methodology*, 2024b.

Zhipeng Xue, Zhipeng Gao, Shaohua Wang, Xing Hu, Xin Xia, and Shanping Li. Selfpico: Self-guided partial code execution with llms. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1389–1401, 2024.

Xiao Yu, Lei Liu, Xing Hu, Jacky Keung, Xin Xia, and David Lo. Practitioners' expectations on automated test generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1618–1630, 2024.

Xiao Yu, Haoxuan Chen, Lei Liu, Xing Hu, Jacky Wai Keung, and Xin Xia. Realisticcodebench: Towards more realistic evaluation of large language models for code generation. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 3021–3033. IEEE, 2025.

Junwei Zhang, Xing Hu, Lingfeng Bao, Xin Xia, and Shanping Li. Dual prompt-based few-shot learning for automated vulnerability patch localization. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 940–951. IEEE, 2024.

Junwei Zhang, Xing Hu, Shan Gao, Xin Xia, David Lo, and Shanping Li. Less is more: On the importance of data quality for unit test generation. *Proceedings of the ACM on Software Engineering*, 2(FSE):1293–1316, 2025.

Junwei Zhang, Xing Hu, Xin Xia, Shing-Chi Cheung, and Shanping Li. Automated unit test generation via chain-of-thought prompt and reinforcement learning from coverage feedback. *ACM Transactions on Software Engineering and Methodology*, 35(4):1–30, 2026.

Yukai Zhao, Menghan Wu, Xing Hu, and Xin Xia. Hfuzzer: Testing large language models for package hallucinations via phrase-based fuzzing. *arXiv preprint arXiv:2509.23835*, 2025.