Agentic AI based Trusted Software: Viewpoint from SE

Abhik Roychoudhury[0000−0002−7127−113] and
Hironori Washizaki[0000−0002−1417−9879]

Automation with Trust

1 Historical Retrospective

Since the 1970's when programming in modern existing programming languages firmly took root, the focus has been on *programming in the large*. The start of this period can be traced back to around fifty years ago, when the first *Hello World* program was written by Brian Kerninghan in 1972-3 in the B programming language, which is the predecessor of C. Since the writing of the first "Hello World" program and the initial seminal innovations, there have been only two really major shifts in the software industry affecting *both the technology as well as the business of software*.

The first shift came 25 years after the "Hello World" program, in the late nineties. Subsequent to the birth of the internet, the software delivery model as a service took root. This led to many Software-as-a-Service (SaaS) offerings in vertical domains such as customer relationship management (SalesForce being an example company in the domain). The movement from in-house software development to cloud-based hosting of software also led to movement of storage or documents to the cloud - allowing for conveniences such as collaboratively working on documents.

In 2025-26, we are now at the second major shift, which affects both the software technology and business: the arrival of *agentic AI*. With the use of agentic AI in software, the change is beyond the hosting of the software. Instead, the engineering of the software itself is automated to a large degree. Since there is greater automation, this eases the writing of code fragments capturing different functionality. However, the correctness of the code and the overall trust in the agent output becomes a key concern that needs addressing. Thus, perhaps for the first time since the writing of the "Hello World"

Abhik Roychoudhury
National University of Singapore, School of Computing, 13 Computing Drive, Singapore 117417, Republic of Singapore
e-mail: abhik@nus.edu.sg

Hironori Washizaki
Waseda University, Tokyo, Japan, e-mail: washizaki@waseda.jp

program we face the central issue of *programming with trust*, instead of programming at large.

Figure 1 shows the timeline of some key developments in the software industry in the past fifty years, highlighting the two major developments: SaaS and agentic AI. It also shows the contrast between ”programming in the large” and ”programming with trust”.

In the rest of the chapter, we will navigate this tension between scale and trust, and also transcend the issues in programming to the core concerns in software engineering. We will also discuss how this tension between scale and trust is evolving as agentic AI solutions for software development.

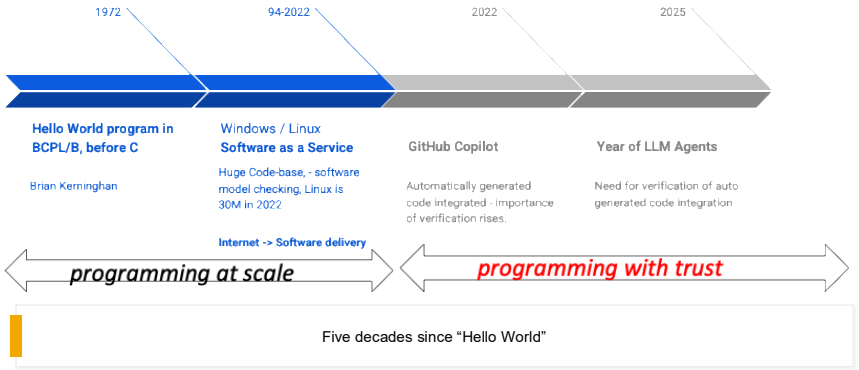


Fig. 1 *Arrival of Agentic AI in Software - more than fifty years after ”Hello World”.*

2 Software Issue Resolution

GitHub Co-pilot [1] is an early approach to AI-based automation in software engineering. It attempts to provide code completions to complete code while the human programmer is typing in an Integrated Development Environment. Instead of a human programmer going to an online community such as Stackoverflow to get help in programming, it brings the knowledge from such online communities to the programmer as part of the development environment. AI agents for coding go beyond the Co-pilot approach since the agents can exercise autonomy to make decisions and act somewhat like a (human) software engineer.

A broad structure of Large Language Model (LLM) agents for software has emerged in the past 1-2 years. An agent is autonomous and makes (informed) decisions on its own. It may be informed by invoking various external tools. In the case of LLM

agents for software, the most effective solution is found if the agents work on program representations. By making the agents interpret program representations and make micro-decisions about software-related tasks, we can have the agent conduct program analysis to make decisions. The program analysis can either be invoked as an external tool, or it can be embedded in the agent's actions as it interprets and navigates the program representation.

One of the key pivots in the growth of LLM agents for software has been the interest around automated issue resolution. A software issue is a unit of work that a (human) software engineer often needs to deal with. Such an issue is a report given to the software engineer requesting some *improvement* of the software project. The improvement can focus on a bug fix, a feature addition or even a desired efficiency improvement in a core library of the software project. In recent times, open datasets of software issues such as SWE-bench [4] have been suggested. Availability of open datasets containing open challenges has spurred the growth of agents targeting automated issue resolution in software. A schematic of an LLM agent for conducting software issue resolution is shown in Figure 2. This schematic is given at a high level, showing how an agent can invoke program analysis tools to work on program representations. At a more detailed level, an agent will use its autonomy to analyze different parts of the code-base on-the-fly in order to accomplish a software task.

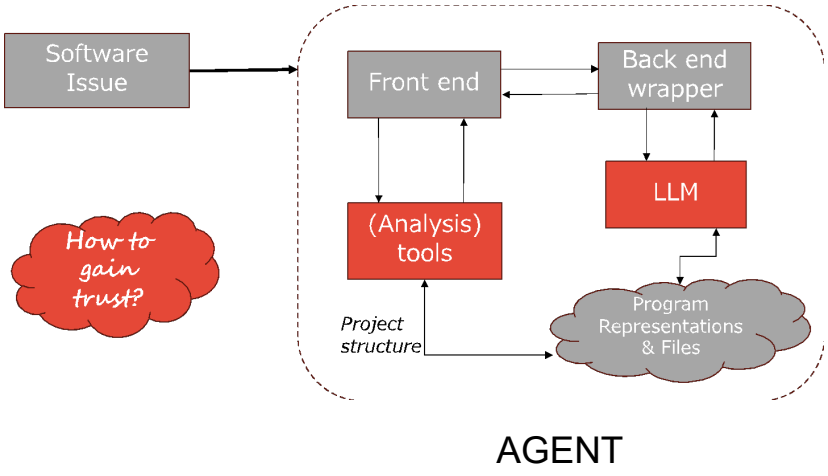


Fig. 2 *Role of analysis in issue resolution.*

Since the SWE-bench dataset suggested the resolution of software issues as a challenge, several AI-based agent approaches have emerged. Devin [14] from Cognition Labs was one of the first approaches. It suggests the use of external tools to aid the agent, thereby suggesting a first step towards an AI software engineer. Devin invokes external tools such as a shell where any command can be input. Thus, the agent can autonomously work like a software engineer inspecting files in the project and focusing

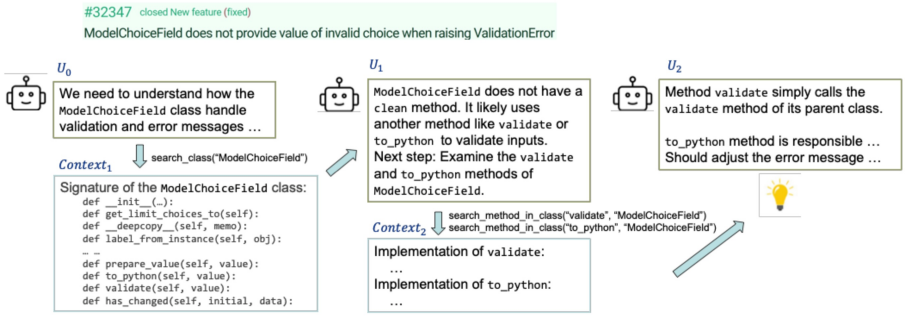


Fig. 3 *Issue Resolution by Intent Inference in AutoCodeRover.*

on certain parts of the program to resolve software issues. The SWE agent [15] provides an agentic approach that is thematically similar to Devin. It suggests the setting up of agent-computer interfaces for the agent to interact with external tools. Openhands [13] is an open-source approach towards AI agents for code. Like a human developer it can invoke a shell, web-browser or consult Stackoverflow to modify code.

3 Intent Inference: a core SE outlook

All of the aforementioned approaches, i.e., Devin, SWE-agent and OpenHands, attempt to create an agentic software engineer by endowing it with the tool capabilities of the human software engineer. However, the human software engineer is also capable of understanding and reasoning about program code semantics during the software engineering process. The process of fixing bugs or adding features in a complex code-base usually is not as simple as invoking a shell or a web-browser or looking up Stackoverflow. The human software engineer usually conducts an implicit intent inference about what is the intended behavior of the software and whether the current code-base is satisfying the intended behavior. This understanding of the intended behavior (and any deviation from it by the current code-base) allows the human software engineer to make code changes, thereby resolving software issues.

Digression: Lessons from Program Repair

Before the arrival of AI agents, the software engineering community has studied the problem of automated program repair [6]. In the problem of automated program repair, the correctness criterion is given as a set of tests. The goal of automated program repair is to automatically modify a given buggy program, so that it can pass the given set of tests. However, this risks the generated patch from overfitting the given test data, it can only pass the given tests, but mostly fail tests outside the test-suite. Trying to generalize the given tests to infer intended behavior is found to be a solution to combat the overfitting

problem. This lesson forms the heart of semantic program repair approaches, which use symbolic program analysis (specifically symbolic execution) for intent inference.

Coming back: Learning from those lessons

Resolving software issues includes the problem of automated program repair, but is a more general problem. Instead of given tests, the desired behavior is given as a natural language issue. Understanding the intended behavior of a software project and how it deviates from the intended behavior can help us resolve the software issue. In an agentic AI approach to building an AI software engineer, it is thus not sufficient to invoke external tools such as a shell or web-browser. Instead of the agent working on code as text, it needs to work on *program representations* and needs to employ program analysis for intent inference.

AutoCodeRover and followup

AutoCodeRover [16] gives a different agentic AI approach by involving program analysis. Given a software issue, its core intuition is to extract specifications (of intended behavior) from the program structure. This is achieved via code search. Starting from the code elements (classes, methods, APIs) mentioned in the issue, it progressively searches other code elements to perform *fault localization* - finding where the code changes are needed. This is then followed by a patch generation stage which considers the knowledge/specifications gained from fault localization. A follow-up work SpecRover [10] makes the implicit intent inference in AutoCodeRover more explicit by inferring the intended behavior of each of the functions (or units off code) identified by fault localization. Figure 3 shows a simple example of issue resolution via intent inference. The issue being resolved is shown at the top. The code-search starts from the `ModelChoiceField` class, which is mentioned in the natural language issue. It then progresses by looking up the signature of this class and then further looking into the implementation of the methdods. The AutoCodeRover agent thus iteratively refines its lower-level understanding of the intended program behavior from the *program structure* such as classes / methods.

Use-cases and explainability

What is the benefit of the agent working on program representations? The main benefit is the inference of intended program behavior by the agent, which can be used to provide explanations of the agent's actions. It enhances trust in the coding agent, and allows one to employ agents in more critical scenarios (that demand greater trust) than vibe-coding. We can also observe this from the current real-life usage of AutoCodeRover, which has now been integrated into the widely used SonarQube tool [2]. SonarQube is used by enterprises for code quality and security - it provides inspection of bugs, code smells and vulnerabilities. Such bug reports generated by SonarQube can be treated as issues which

are then automatically fixed by AutoCodeRover. This shows the promise of agentic AI in providing support for critical code.

A recent work [5] has shown the promise of symbolic explanations such as property-based tests for further explaining the actions of AI agents for software. Such explanations can greatly enhance the trust, but would need some changes on top of coding agents in production usage. We also note that most coding agents currently produce some natural language explanation by default though the quality / use of these explanations need further examination. Another recent work looks into the measurement of coding agent explanation quality for widely available coding agents [7].

4 Verification and Validation

Issue resolution forms the core of program improvement and maintenance of a software project. While issue resolution and prompt-based code generation have captured significant attention, the opportunities of using agentic AI for code are by no means restricted to these areas. With the growth of automation in software development, there will arguably be an explosion of AI-generated code. This will exacerbate the issue of verification and validation (V & V) of the generated code. Furthermore, as the generated code gets integrated to code-bases, the interplay of legacy human written code and the generated code - will lead to subtle errors and vulnerabilities. Even if the agentic AI that produces code (fragments) is trustworthy, the integration of AI generated code can create new trust issues in software projects further down the line. There is a long history of prominent vulnerabilities caused by (human-written) software. The latest prominent example is perhaps CrowdStrike outage in 2024, where the security system itself had a bug that caused worldwide outages. At the same time, AI generated code from prompts have been reported to have many vulnerabilities (see *e.g.*, [8]). There may be a lingering "*trust deficit*" in AI generated code.

Looking forward, one prominent direction thus could be to put agentic AI itself to use in combating the trust deficit from AI generated code! Thus, given a key critical high-level natural language requirement, agentic AI may help clarify the requirement by understanding its intent, as well as understanding how it is implemented at a low level in the code-base. This may lead to capabilities of security audits which have been recently studied [3]. In fact, it can go beyond audits by flagging violations of the policy and fixing these violations via code changes albeit by an issue resolution agent in the back-end.

Last but not the least, agentic AI can be put to use for formal verification of code [12]. Just like agents like AutoCodeRover can interpret program representations and analyze them, we can devise AI agents, which can interpret formal proof representations, analyze them, and communicate with a theorem prover via an iterative refinement loop to "improve" the proof. This will also contrast with approaches like Alphaproof[1] which help prove general mathematical theorems. Instead, a program verification agent can again leverage the program structure like classes / methods to structure the proof and

[1] https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/

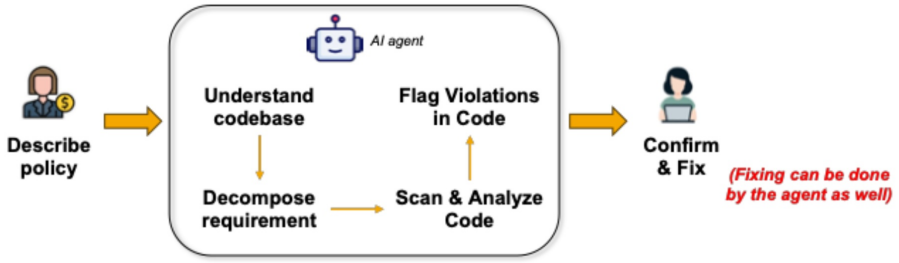


Fig. 4 *Agentic AI for Policy Enforcement in Code.*

its intermediate lemmas. This remains an attractive future direction which is currently being investigated.

Finally, for a classification of technical and human trust in using agents for software, the reader is referred to the overview article [9].

5 Patterns of Agentic AI-based Software Engineering

Researchers and practitioners have been investigating best practices for GenAI-based software engineering, leading to the identification, formalization, and documentation of software engineering patterns. Representative examples include vibe-coding patterns, which capture emerging practices for collaborating with GenAI during coding. In addition, the prompt engineering, context engineering, and harness engineering patterns introduced in Chapter 2 provide useful foundations for GenAI-based software development. As software development increasingly incorporates GenAI/LLM agents, this area has become a focus of both research and practice, creating a need to further develop and systematize the associated practices and patterns. The following Interactive Coding Paired with AI pattern is an example of a vibe-coding pattern that leverages GenAI and coding agents [11].

Interactive coding paired with AI

Intent:

Achieve reliable coding in a faster way by adopting an iterative human–AI collaboration loop in which AI coding tools/agents and humans complement each other's strengths [11].

Context:

Developers with programming and project knowledge use GenAI coding tools to generate, test, and document code. They must be able to translate abstract requirements into concrete tasks and communicate project-specific requirements, constraints, and context to AI.

Problem:

AI can rapidly generate code but may not fully understand project-specific requirements, conventions, dependencies, or contextual constraints. Unverified AI-generated code can therefore introduce defects, violate requirements, reduce maintainability, or result in costly rework. The key forces are:

- Productivity: Automating routine development tasks allows developers to focus on higher-value activities.
- Functional Suitability: Generated code must satisfy functional requirements and project-specific constraints; early verification and correction help prevent defects and mismatches from propagating.
- Maintainability: Code should remain readable, consistent, understandable, and maintainable despite AI-assisted generation and modification.
- Human Judgment: Developers must retain responsibility for decisions that require domain knowledge, contextual understanding, and engineering judgment.

Solution:

Use an iterative human–AI collaboration loop in which AI and humans complement each other's strengths:

Step 1. AI Generation: Ask an AI coding tool to generate code, tests, or documentation. Decompose complex requirements into smaller, verifiable tasks and provide relevant context and constraints.

Step 2. Human Verification and Correction: Execute and test the generated artifacts. Identify defects, requirement gaps, or inappropriate design decisions, and correct them manually or through interaction with AI.

Step 3. Human Modification: Incorporate project-specific requirements, coding conventions, architectural decisions, and domain knowledge that AI may not adequately infer.

Step 4. AI Inspection: Ask AI to inspect the resulting artifacts for potential defects, security issues, inconsistencies, and missed improvements.

Step 5. Iteration: Repeat the generation, verification, modification, and inspection cycle until the artifacts satisfy the required quality and project constraints.

Consequences:

+ *Productivity*. AI automates routine coding, testing, and documentation tasks, allowing developers to focus on higher-value activities.
+ *Functional Suitability*. Iterative human verification and AI-assisted inspection can identify defects and requirement mismatches earlier.
+ *Maintainability*. Human review combined with AI-assisted inspection and documentation can improve readability, consistency, and maintainability.

+ *Quality and Risk Control*. Alternating AI generation with human judgment reduces the risk of blindly accepting incorrect or inappropriate AI-generated artifacts.
+ *Changed Developer Role*. Human effort shifts from writing every implementation detail toward task decomposition, context provision, verification, correction, and engineering judgment.
- *Verification Overhead*. Human review, testing, and correction require time and expertise. For small or well-understood tasks, the verification effort may offset the productivity gains from AI generation.
- *Cognitive Load*. Developers must continuously switch between generating, inspecting, testing, correcting, and communicating with AI. This iterative interaction can increase cognitive workload.
- *Overreliance on AI*. Developers may accept plausible-looking AI-generated code without sufficient verification, particularly when the code is complex or outside their expertise.
- *Context Provision Cost*. Providing AI with sufficient requirements, architectural information, conventions, and project context can itself become a significant engineering task.

Acknowledgements This research / project is supported by the National Research Foundation, Singapore, under NRF Investigatorship Program, Award ID: NRF-NRFI11-2026-0001, "Agentic AI based Software of the future: from Scale to Trust".

Competing Interests Abhik Roychoudhury has been a Senior Science Advisor at SonarSource (2025-26) apart from his full-time employment at National University of Singapore. The work of AutoCodeRover has been converted into a commercial product SonarQube Remediation Agent which is available to enterprises for commercial usage. The intellectual property for AutoCodeRover has been transferred to Sonar and is owned by Sonar.

Ethics Approval Not applicable

References

1. Friedman, N.: Introducing github copilot - your ai pair programmer. The GitHub Blog (2022)
2. Gaudin, O., Mallet, F.: Sonar. Methods and tools **18**(1), 40—46 (2010), https://www.methodsandtools.com/PDF/mt201001.pdf
3. Guo, J., Wang, C., Xu, X., Su, Z., Zhang, X.: Repoaudit: An autonomous llm-agent for repository-level code auditing. In: Forty-Second International Conference on Machine Learning (ICML) (2025)
4. Jimenez, C.E., et al.: Swe-bench: Can language models resolve real-world github issues? In: International Conference on Learning Representations (ICLR) (2024), https://www.swebench.com/
5. Kang, S., Ruan, H., Roychoudhury, A.: Autocodesherpa: Symbolic explanations in ai coding agents. In: ACM International Sympsoium on Software Testing and Analysis (ISSTA) (2026)
6. Le Goues, C., Pradel, M., Roychoudhury, A.: Automated program repair. Communications of the ACM **62** (Dec 2019)
7. Pan, Z., Kang, S., Yusuf, I.N.B., Roychoudhury, A.: Explainbench: Evaluating code explanations from agents. In: 41st IEEE/ACM International Conference on Automated Software Engineering (ASE) (2026)

8. Pearce, H., et al.: Asleep at the keyboard? assessing the security of github copilot's code contributions. Communications of the ACM **68**(2) (2025)
9. Roychoudhury, A., et al.: Agentic ai software engineers: Programming with trust. Communications of the ACM (2026)
10. Ruan, H., Zhang, Y., Roychoudhury, A.: Specrover: Code intent extraction via llms. In: 47th International Conference on Software Engineering (ICSE) (2025)
11. Takai, Y., Washizaki, H., Yoshioka, N., Fukazawa, Y.: Classification of human-human and human-ai pair programming effects and expansion for ai pair programming patterns. In: Proceedings of the 30th Conference on Pattern Languages of Programs. PLoP '23, The Hillside Group, USA (2023)
12. Tu, H., Zhao, H., Song, Y., Zafar, M., Meng, R., Roychoudhury, A.: Agentic verification of software systems. In: ACM International Conference on Foundations of Software Engineering (FSE) (2026)
13. Wang, X., et al.: Openhands: An open platform for AI software developers as generalist agents. In: The Thirteenth International Conference on Learning Representations (2025), https://openreview.net/forum?id=OJd3ayDDoF
14. Wu, S.: Introducing devin, the first ai software engineer. Cognition Labs blogpost (Mar 2024), https://www.cognition.ai/blog/introducing-devin
15. Yang, J., et al.: Swe-agent: Agent-computer interfaces enable automated software engineering. Thirty-Eighth Annual Conference on Neural Information Processing Systems (NeurIPS) (2024)
16. Zhang, Y., Ruan, H., Fan, Z., Roychoudhury, A.: Autocoderover: Autonomous program improvement. In: ACM International Symposium on Software Testing and Analysis (ISSTA) (2024)