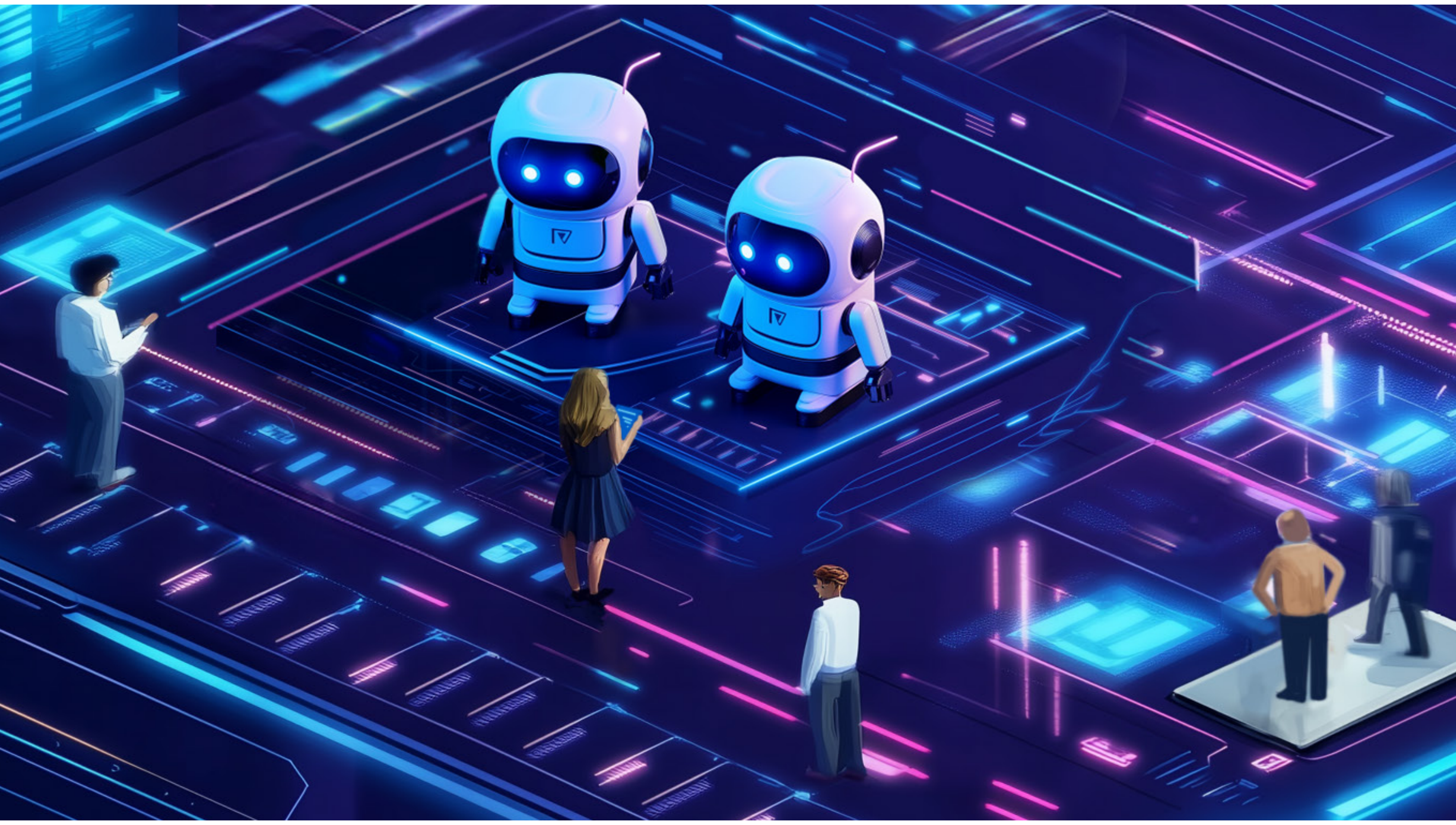
Designing AI Software Engineer of the Future

Abhik Roychoudhury, March 2025
Professor National University of Singapore, and Senior Advisor SonarSource SA



Provost's Chair Professor Abhik Roychoudhury examines AI's evolving role in software engineering, from LLM-driven automation to unified AI developers and human-AI collaboration.

Artificial intelligence (AI) technologies, particularly large language models, are transforming software engineering by automating coding tasks. However, software engineering extends beyond code generation to maintenance, architecture and collaborative development. The emergence of AI software engineers raises fundamental questions about their role, capabilities and integration into human software teams.

PROLOGUE

Software Engineering is a discipline sitting at the intersection of science and engineering. It brings in strong principles from programming languages, formal methods and other fields to provide rigor in how we design, build, maintain and evolve software systems. At the same time, and more importantly, it has over a period built a set of solid engineering principles for creating and maintaining very large codebases via continuous integration and continuous deployment (also known as CI-CD). This gradual evolution of the software engineering discipline has lasted for about half a century or fifty years. In fact, last year I had the privilege of co-chairing the 49th edition of the International Conference on Software Engineering (ICSE) and next month in April, the 50th edition of ICSE is being held. Fifty years is a long time in computing, particularly when disruptive innovations abound. As software engineering turns 50, artificial intelligence (AI) technologies like Large Language Models (LLMs) have burst into the scene with its promise around automated coding. But, software engineering is less about coding, and much more about various other aspects like design, maintenance, evolution, architecture, as well as human and social aspects. The rational question then is: with the arrival of LLMs in software engineering, what will be the role of software engineering, *beyond* coding? What is an AI software engineer [6]?

Software engineering is less about coding, and much more about various other aspects like design, maintenance, evolution, architecture, as well as human and social aspects.

AGENTS

To answer the future development of software engineering, let us start with the ongoing developments. The year 2025 is the year of agents, with NVIDIA CEO Jensen Huang, calling it a "multi-trillion dollar opportunity". Agentic AI has shown significant promise in terms of not only coding automation, but also in other coding artifacts. End-to-end web-app generation completely automatically from natural language requirements - remains a topic of active discussion. However, code components generated automatically in this fashion may not be integrated in a trustworthy fashion into a complex software project.

There has been significant attention devoted to building LLM agents for conducting software engineering tasks, such as fixing a bug, or more broadly for remediation of "issues". But then what is an agent, and to what extent does this achieve the goal of an AI Software Engineer? Agents rely on one or more Large Language Models (LLM). These go beyond the prompt engineering approach - where a prompt may be given to a LLM - and with step by step prompts, LLM may be used to solve a problem. In contrast, an agent is able to use LLMs, as well as external tools such as program analysis tools. The agent is not a fixed algorithm, but an autonomous plan to solve a problem. The autonomy in the agent remains a central design aspect of an LLM agent.

A first cut visualization of an agent is seen in the following. The depiction in Figure 1 shows a structuring of an LLM agent for issue remediation. Such an issue could refer to a program improvement needed such as bug fix, or feature addition or even performance improvement. The natural language issue could be given as a prompt to the front end of the agent which can involve an LLM (or collection of LLMs), along with a thin wrapper to invoke the backend as needed. The back end may coordinate or delegate the invocation of various tools, including program analysis tools or file navigation utilities. Depending on the tool invoked it may work on program representations or navigate the file structure of a software project. Such autonomous invocation of tools may build a patch to remediate the issue, or check a generated patch against various program artifacts like tests with the goal of producing a high quality patch.

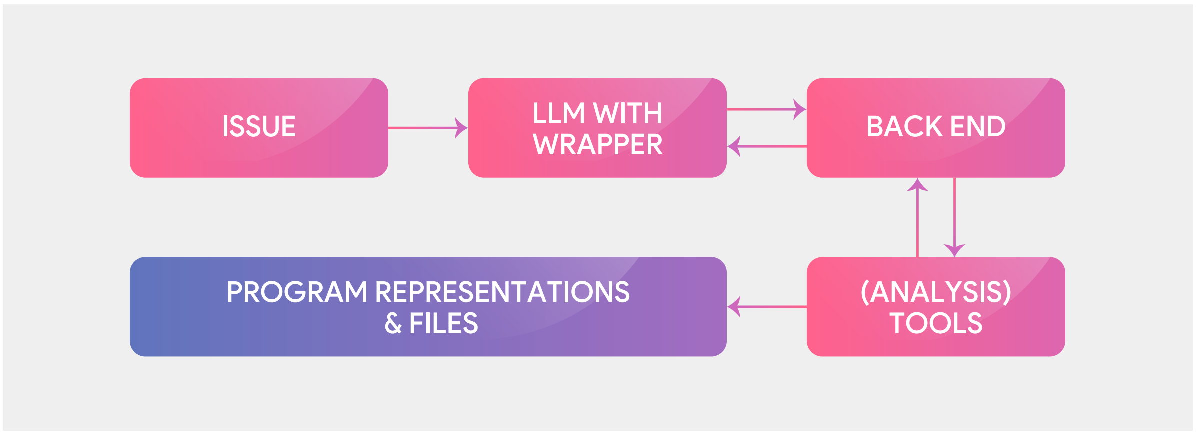


Figure 1: Thinking behind an LLM agent for issue remediation

AUTOCODEROVER

An LLM agent for software issue remediation constitutes a first draft of an AI software engineer, it truly does. A lot of human software engineers in software development companies are engaged in going over an issue tracker, handling issues assigned to them, and posting code commits into a software project to handle the issues. So if we can have an LLM agent to remediate issues, it should justifiably be considered a (junior) AI software engineer. AutoCodeRover [1,2] constitutes such an AI software engineer, a significant step in terms of greater automation in software development. Without getting into a lot of details, the following is a depiction of the high-level design of AutoCodeRover.

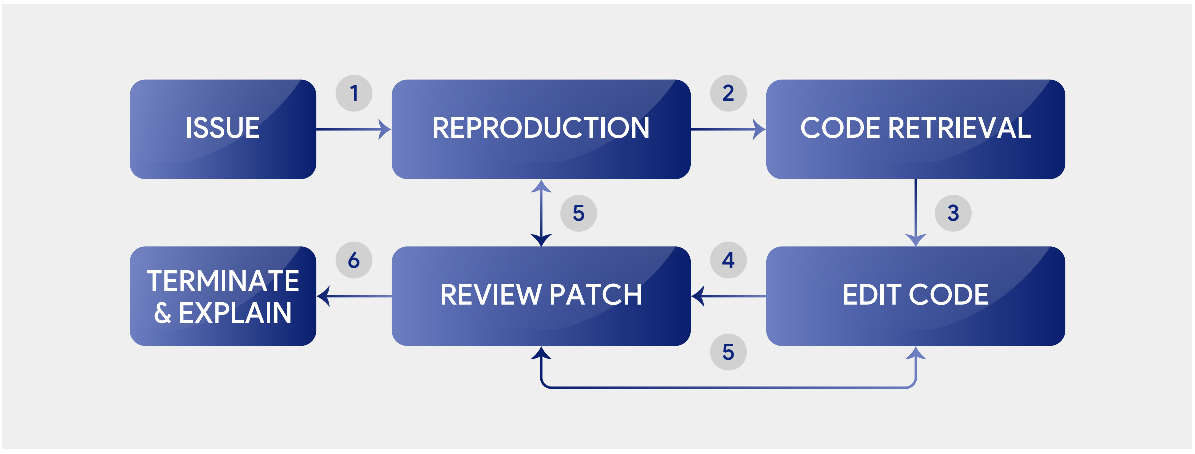


Figure 2: High-level design of AutoCodeRover

Figure 2 captures some of the key steps in the design of such an AI software engineer. Given a task, which is typically in the form of an issue, it will reproduce the issue typically via one or more reproducing tests. In the next and crucial step, it performs code retrieval via an autonomous code search with the goal of "debugging" the issue. This is where a lot of program analysis tools can play a role and can aid code search. Such a debugging of the issue eventually leads to a code edit or a patch. This patch is then cross checked or validated by additional review steps. The review may involve checking the patch against various artifacts like tests, and will typically produce an explanation to accompany the patch. Integrating such agentic AI approaches into production grade tooling is becoming a reality.

UNIFIED AGENT

Where do we go from the current burst of interest in LLM agents? Will it persist or will it morph in some ways to define the software engineering of the future in a consolidated fashion? Is this as far as we can progress the capabilities of an AI software engineer? To answer these questions, we will now need to reflect on the activities a competent human software engineer is engaged in. We will also need to reflect on the ground realities of the inherent capabilities of LLMs. Let us first reflect on the advanced capabilities of a human software engineer. A software professional today is not only doing individual activities such as testing, coding, patching. Instead a capable software engineer will handle many complex scenarios such as

Adding a feature and then taking care of any bugs it can introduce in a codebase

Adding a fix according to an issue, and if it is incomplete fix, to complete it

Take over the code of a developer who has left the organization, run tests to understand the code first, then add more tests to check the understanding

We call such software engineering tasks as “scenarios” from the software industry. In reality, handling such a scenario is like conducting a chain of primitive tasks. For example, consider the scenario of handling an incomplete fix in a large codebase. It will involve testing the code to find a “problem”, providing an initial fix, testing the code further to realize the incompleteness of the fix, and finally producing a complete fix and comprehensively testing it. What is needed in the current LLM agents so that they can handle such “scenarios”?

It is reasonable to hypothesize the prevalence of such unified agents in the near future, as junior members of software development teams, collaborating alongside human developers.

To understand the capability needed to take forward the design of an AI software engineer further --- it may be useful to settle several primitive tasks, the vocabulary of basic activities of an AI software engineer. We call this action space, as depicted in the following. What is crucial is that all the actions in such an action space can be presented to the AI Software Engineer with a unified interface. Of course each action may also introduce tests and program modifications. The unified software engineering agent can then choose and interleave actions from the action space while managing system “state”. A meta agent then takes in different actions and orchestrates them. This design would allow for (at least) two degrees of autonomy - the meta-agent can execute different actions autonomously, and the individual actions may also involve invocation of different analysis tools and program utilities which are handled autonomously.

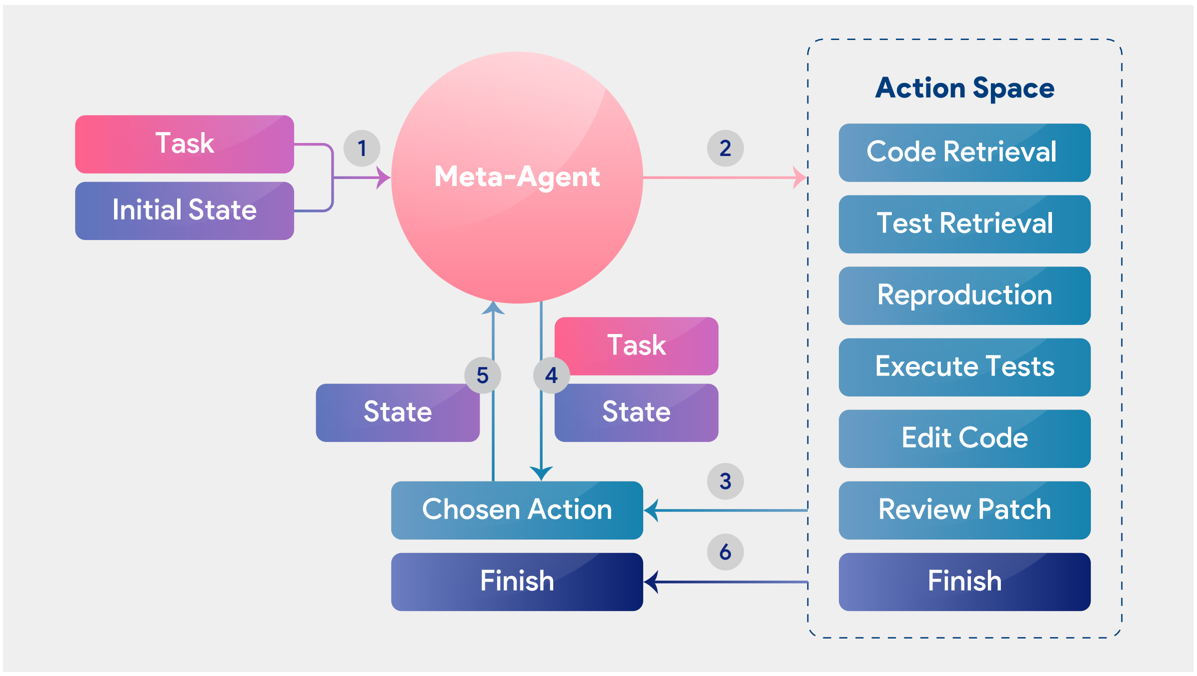


Figure 3: Thinking behind a Unified Software Engineering Agent

It is reasonable to hypothesize the prevalence of such unified agents in the near future, as junior members of software development teams, collaborating alongside human developers. For organizations to deploy them with confidence, we also need comprehensive datasets of software engineering tasks, going beyond the current popular datasets like SWE-bench [3].

AGENTS for FORMAL METHODS

So far, we have talked about the core software engineering activities like software maintenance. I have mentioned how agents may play a role there, but I would also like to emphasize that the role of humans will remain. Instead, agents will permeate any and all of software engineering activities, not only software maintenance. One of the most rigorous and formal processes in software engineering is the ability to formally and mathematically verify properties of programs. Will agents play any role in program verification? The work on Alphaproof [4] gives us some guidance in this regard, where AI has achieved silver medal standard in solving International Mathematics Olympiad problems. What implications does it have for those of us in software engineering? Is it only an interesting lunch-time conversation to lazily partake in with our colleagues? Or does it have some lessons for us in terms of program verification?

The development of Alphaproof could have significant hope for new-age program verification. In an agentic architecture on program verification, an LLM agent could construct a proof of a program property and an automated theorem prover could check the proof and provide feedback in a loop. This is already known and is not entirely novel. What could be novel is the role of lemmas in proofs. In general mathematical proofs, there is a significant inventive step in choosing the "right" lemmas to construct the proof. In program verification we can adapt the codesearch of LLM agents like AutoCodeRover to search over (program location, proof obligation) pairs intelligently and derive suitable lemmas in the process. If the software maintenance agent's code search can be adapted for lemma discovery, it would be a significant relief (and progress) towards formal program verification.

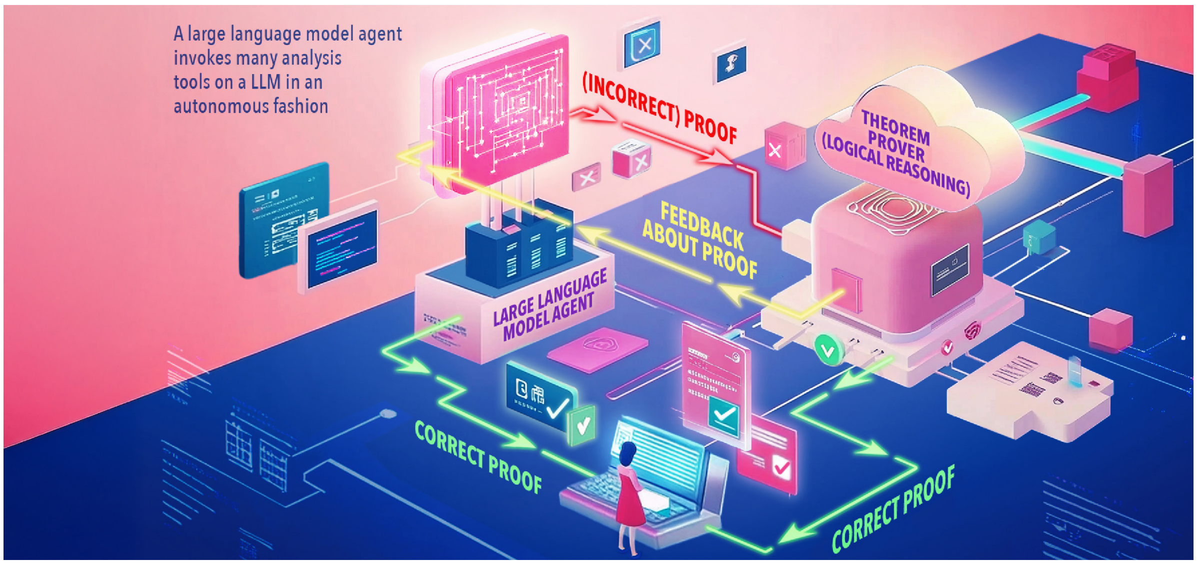
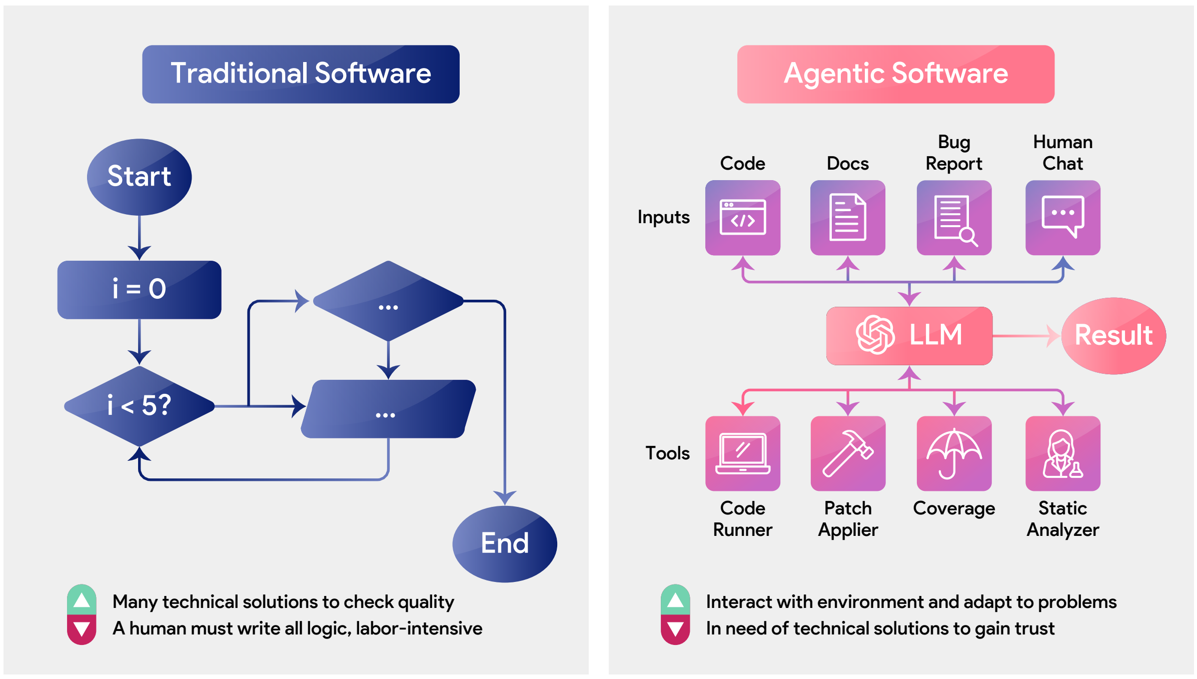

A large language model agent invokes many analysis tools on a LLM in an autonomous fashion
(INCORRECT) PROOF
THEOREM PROVER (LOGICAL REASONING)
FEEDBACK ABOUT PROOF
LARGE LANGUAGE MODEL AGENT
CORRECT PROOF
CORRECT PROOF

Figure 4: AI generated schematic for Program Verification Agents

AGENT TESTING and SAFETY

Testing and validation of LLM agents for coding is central to providing assurance about automatically generated code. Testing an agent is different from testing a LLM, in the same way as testing reactive systems is different from testing transformational systems. In a transformational system, we provide an input, check observed output and compare it against the oracle or expected output. In reactive systems, the system is in continuous interaction with the environment. Hence for testing reactive systems, we generate execution traces as sequences of actions and check them against various properties, typically safety properties which may be stated conveniently in Linear-time Temporal Logic (LTL) [7]. Similar to the difference between testing transformational and reactive systems, we would like to differentiate the testing of LLMs and agents. Testing a LLM may involve providing prompts, and examining (properties of) the output. Testing an agent will involve examining the sequences of interactions between LLMs and external tools - and checking properties of these interaction sequences.


Traditional Software
Start
i = 0
i < 5?
...
...
End
Many technical solutions to check quality
A human must write all logic, labor-intensive
Agentic Software
Inputs
Code
Docs
Bug Report
Human Chat
LLM
Result
Tools
Code Runner
Patch Applier
Coverage
Static Analyzer
Interact with environment and adapt to problems
In need of technical solutions to gain trust

In the context of agentic testing, we can envision a combination of reactive system and transformational system testing.

- For the individual LLM calls in the context of an agentic execution, we can profile the LLM call inputs/outputs and develop properties that should hold for these LLM calls. These properties can then be enforced in the LLM calls of future agentic executions.

- We can also inspect the execution traces of the agents to see temporal properties that do not hold, and suggest rectifications of the agentic executions or trajectories. The approach is to inspect agentic executions, and rectify them on-the-fly to reduce LLM hallucination and improve code quality. Extensive experiments need to be conducted on agents to see its efficacy.

- It is also possible to look beyond agentic testing, and study the security impact of the LLM being invoked. Thus, instead of enforcing properties of LLM calls, it is possible to mitigate risks from LLMs, by sanitizing the LLM outputs. This can enhance security of the automatically generated code.

TEAM DYNAMICS

So far, I have upped the ante in terms of capabilities of what an individual AI software engineer can achieve. However, as a human software engineer, my output is not entirely my output. They are also triggered by a conversation that I had with a colleague over a coffee break. It is also influenced by an architectural diagram another colleague showed me in a group meeting. So, the software development team dynamics will need to be figured out, in the longer term, when AI software engineers become prevalent. As a general goal, many of us may agree with the need to understand human-AI, and AI-AI interactions for accomplishing future development.

As a general goal, many of us may agree with the need to understand human-AI, and AI-AI interactions for accomplishing future development.

However, getting into the specifics, it is unclear whether we can accomplish this so easily either by theoretical studies or by empirical studies. The approach of using games and reward functions may not work in the software development context. A software engineer is not necessarily participating in a group meeting to get instant rewards for her contributions. Existing cognitive theories which focus on cooperative work [5] by inferring the goals of others in the team, may not also directly apply to a software development team. Designing software team dynamics could be the next frontier after we agree on the design of AI software engineers. *We are not there yet!*

Acknowledgements

I would like to acknowledge discussions with attendees at three recent meetings
(a) Dagstuhl Seminar on Automated Programming and Program Repair (Oct 2024),
(b) Shonan Meeting on Trusted Automatic Programming in Japan (Jan 2025) and
(c) IFIP Working Group 2.4 meeting (Feb 2025).

References

[1] AutoCodeRover: Autonomous Program Improvement, Zhang, Ruan, Fan, Roychoudhury, International Symposium on Software Testing and Analysis, 2024.

[2] SpecRover: Code Intent Extraction via LLMs, Ruan, Zhang, Roychoudhury, International Conference on Software Engineering, 2025.

[3] SWE-bench: Can Language Models Resolve Real world GitHub issues? Jimenez et al, International Conference on Language Representations, 2024.

[4] AI achieves silver-medal standard solving International Mathematical Olympiad problems, Alphaproof and AlphaGeometry teams, Google Deepmind, 2024.

[5] Bayesian Theory of Mind: Modeling Joint Belief Desire Attribution, Baker, Jara-Ettinger, Saxe, Tennebaum, Annual Meeting of Cognitive Science Society, 2011.

[6] Agentic AI Software Engineer: Programming with Trust, Roychoudhury, Pasareanu, Pradel, Ray, [2502.13767] AI Software Engineer: Programming with Trust, 2025.

[7] Linear-time Temporal Logic guided Greybox Fuzzing, Meng, Dong, Li, Beschastnikh, Roychoudhury, International Conference on Software Engineering, 2022.