Practical Applications of State-Of-The-Art Large Language Models to Solve Real-World Software Engineering Problems Autonomously

Yurij Mikhalevich
QA Wolf
Dubai, United Arab Emirates
email: yurij@mikhalevi.ch

***Abstract*—This paper researches the application of state-of-the-art large language models to autonomously solve real-world software engineering problems based on the problem description intended for humans. For this research, we picked 10 outstanding GitHub issues of different difficulty levels in the Aibyss project. We tasked an AI agent to autonomously solve them based solely on the GitHub Issue description intended for human software engineers. As part of this research, we compared the following large language models: Claude Sonnet 3.7, DeepSeek-V3, DeepSeek-R1, and o3-mini-high. We used the Aider agent to solve the problems. Additionally, we have evaluated the Claude Code agent as one of the best closed-source AI software engineering agents. We have found that the best performance is achieved by Claude Sonnet 3.7 with reasoning enabled – with the Aider agent and the Claude Code agent. Both of them provided working solutions to 5 out of 10 GitHub issues. We analyze the agents' behaviors, including reasoning steps, common failure modes, and the impact of reasoning tokens. The results highlight both the promise and the current limitations of autonomous LLM-based software engineering.**

Keywords-code generation; large language models; AI agents; natural language processing

I. INTRODUCTION

Recent advances in large language models (LLMs) have led to powerful code generation systems capable of assisting software developers. Models like o3-mini and DeepSeek-R1 can translate natural language specifications into code with impressive accuracy [1][2]. These models underpin tools such as GitHub Copilot and Cursor Composer, which have been rapidly adopted as "AI pair programmers" to autocomplete code and suggest solutions [3][4][5]. Studies show that such tools can improve developer productivity, but also raise questions about reliability and how developers interact with AI-generated code. So far, these AI coding assistants operate with a human in the loop: the developer guides the process, reviews suggestions, and tests or debugs the outcomes.

A growing area of interest is whether state-of-the-art LLMs can operate more *autonomously* to tackle software engineering tasks end-to-end. Inspired by agentic frameworks like ReAct [6] and the popularity of systems such as AutoGPT [7], researchers have begun treating LLMs as independent problem-solvers rather than just interactive assistants. For example, the ReAct paradigm by Yao et al. [6] enables an LLM to generate reasoning traces and act on them iteratively, and projects like AutoGPT aim to let an LLM plan and execute a sequence of steps towards a high-level objective. Meanwhile, multi-agent approaches have emerged to coordinate multiple LLMs or tools in specialized roles (e.g., planning vs. coding) to solve complex tasks [8]. Existing benchmarks of these autonomous LLM agents indicate that the choice of the underlying model has a critical impact on success: for instance, GPT-4 can substantially outperform GPT-3.5 or smaller models in autonomous decision-making tasks [9].

Recent advances in LLM-based agents open up the possibility of automating several steps in the classical software development lifecycle. Tasks such as requirement interpretation, code generation, test creation, and documentation can now be at least partially handled by these agents. Especially in the early phases-like prototyping or resolving isolated issues from natural language descriptions-LLMs show a strong capacity for autonomous operation [10]. However, phases involving architectural decisions, integration testing, and final validation still rely heavily on human expertise and oversight.

Despite the growing capabilities of these models, transitioning from generated code to a trusted, production-grade system presents significant challenges. These include ensuring correctness, robustness, maintainability, and compliance with domain-specific standards. Generated code often lacks integration context, can contain subtle bugs, or may not align with broader system constraints. Therefore, human-in-the-loop review, continuous integration pipelines, and formal verification methods are often critical to close the gap between raw LLM output and trustworthy software.

Research in software configuration management (SCM) is increasingly intersecting with AI-driven development. Some ongoing work investigates how agents can update deployment configurations, manage dependencies, and track version history intelligently. Emerging tools explore LLMs not just as code generators but as collaborative participants in the evolution of codebases, integrating with version control systems and automating routine deployment and maintenance tasks [11].

Moreover, there is a growing interest in fine-tuning or pretraining LLMs for specialized tasks such as software update management. These niche LLMs aim to support activities like patch generation, changelog summarization, and semantic versioning analysis. This area is attracting attention as organizations seek to reduce the overhead of continuous software maintenance through domain-adapted language models [10][12].

In this work, we explore the practical application of cutting-edge LLMs as autonomous software engineers on real-world tasks. We design an experiment in which an AI-driven coding agent is given only the natural-language description of a software issue (as one would find in a bug tracker or feature request) and is tasked with resolving the issue by modifying

the codebase, without human assistance. We evaluate the following state-of-the-art LLMs in this autonomous setting: Claude Sonnet 3.7 [13], DeepSeek-V3 [14], DeepSeek-R1 [2], and o3-mini-high [1]. We have used the Aider [15] agent to solve the problems – one of the best open-source AI software engineering agents. Additionally, we have evaluated the Claude Code [13] agent as one of the best closed-source AI software engineering agents. We examine not only whether the LLM-powered agent can produce a working solution, but also the quality of the solution (linting, code style, user experience) and the computational cost (API calls/tokens consumed).

The results of the research show that the best performance is achieved by Claude Sonnet 3.7 with reasoning enabled – with both the Aider agent and the Claude Code agent. Both of them provided working solutions to 5 out of 10 GitHub issues. Surprisingly, Aider paired with o3-mini-high performed the worst out of all the agents and has shown the worst understanding of the problems.

The rest of the paper is organized as follows. Section 2 provides an overview of the related works. Section 3 describes the method used in the research. Section 4 presents the experiment implementation details. Section 5 presents the evaluation results. Section 6 presents a detailed analysis of agent behaviors and failure modes. Finally, Section 7 concludes the paper.

II. Related Works

In this section, we examine the existing literature and research efforts that form the foundation for our current study. Prior investigations have established several key approaches and methodologies that inform our work.

A. LLMs for Code Generation and Assistance

The use of large neural models for code generation has rapidly progressed in recent years. OpenAI's Codex model [16], which powers GitHub Copilot, was among the first to demonstrate that an LLM trained on vast amounts of code can produce syntactically correct and often functionally correct code for given descriptions. Subsequent models have pushed these capabilities further: DeepMind's AlphaCode achieved a performance on par with average human competitors in programming contests[17], signaling the potential of LLMs to handle complex algorithmic problems.

Recent developments in the field have demonstrated significant progress in computational capabilities. Specifically, models such as OpenAI 03-mini-high, DeepSeek-R1, and Claude Sonnet 3.7 have established new performance standards [1][13]. These advancements indicate the continued rapid evolution of LLM capabilities, with potential implications for fully autonomous software engineering agents.

B. LLM-Based Autonomous Agents

Beyond single-turn code completion, the idea of an LLM-driven agent that can perform multi-step tasks has gained traction. The ReAct framework [6] pioneered the combination of *chain-of-thought* reasoning [18] with action execution, enabling an LLM to decide not only *what to think next* but also *what action to take* in a unified prompting strategy. This idea of using the LLM's own output as an intermediate state has influenced many subsequent systems.

In early 2023, a series of autonomous agent prototypes built on GPT-4 (such as AutoGPT) captured popular imagination. These systems prompt the LLM to continuously plan and execute sub-tasks towards a given objective, simulating an "AI agent" that can function without user prompts for each step.

Recent developments in AI-powered code generation have witnessed significant breakthroughs with the emergence of sophisticated agent-based systems. Notably, Courser Composer and Aider have demonstrated improved capabilities in autonomous programming tasks [5][15].

Our work can be seen as an instance of an LLM agent applied to a focused real-world task: given a specific issue in an existing software project, the agent (backed by an LLM) must understand the problem, read and modify the project's code, and submit a solution. We contribute new data on how today's strongest LLMs perform in this autonomous coding scenario, complementing prior research.

III. Method

Our research methodology is designed to evaluate each LLM's ability to autonomously resolve real software issues under controlled conditions. We selected the open-source project Aibyss, a web-based AI competition game, as our testbed. Aibyss is a TypeScript project (Nuxt/Vue frontend with a Node.js backend using Prisma ORM) where users write AI bots to compete in a game [19]. We chose Aibyss because it is a non-trivial codebase with realistic features and bugs, yet manageable in size. From Aibyss's issue tracker, we picked ten issues that were open and well-described. These issues covered a range of feature requests and bug fixes and were labeled by us based on the perceived difficulty as "easy," "medium," or "harder".

A. Task Selection

The 10 issues included 3 labeled "easy", 4 "medium", and 3 "harder". Each issue consisted of a title and a description intended for human developers. We did not provide any additional hints or test cases to the agent beyond this text.

Below is the complete list of problems that we selected and their issue titles, presented as-is:

1) easy - "feat: draggable splitter between the code and the game screen should remember its position between the page reloads"
2) easy - "feat(rating): highlight top results in k/d, kills, deaths, and food eaten columns in the rating table"
3) easy - "chore(World): double the frequency of food spawns"
4) medium - "feat: allow turning off the bots of some users by setting the "inactive" field in the database on the user object to 'true"'
5) medium - "feat: ensure that the game screen occupies all available free space to the right of the code editor"

6) medium - “feat(rating): add a new column to the rating table displaying the number of times the user submitted the code”
7) medium - “feat(sandbox): add an option to turn off sprites and replace them with circles to make debugging easier”
8) harder - “bug: fix the issue causing the bot code to submit when the user opens “API reference”
9) harder - “feat: add code versions and an option to revert to a previous version”
10) harder - “feat: surface bot execution errors to the user”

The actual GitHub issues with their descriptions can be found on the Aibyss project GitHub issues page [20].

B. Agents and LLM Variants

We evaluated six agent configurations:

- Aider 0.75.2 + o3-mini-high 2025-01-31
- Aider 0.75.2 + DeepSeek-V3
- Aider 0.75.2 + DeepSeek-R1
- Aider 0.75.2 + Claude Sonnet 3.7 20250219
- Aider 0.78.0 + Claude Sonnet 3.7 20250219 with 32k thinking tokens – in this variant, we enabled the “thinking mode” in Aider (using v0.78.0 with thinking support for Claude 3.7)
- Claude Code 0.2.35 – Anthropic’s Claude Code is a proprietary agent with a CLI interface very similar to Aider’s that uses the Claude Sonnet 3.7 model under the hood; this can be seen as a closed-source counterpart to Aider, specifically tuned for Claude [13]

C. Autonomy and Stopping Criteria

We configured the agents to operate fully autonomously. Aider was run with the `--yes-always` flag, meaning it would automatically apply its proposed actions. In the case of Claude Code, we approved all its prompts manually. Each agent was allowed to iterate until it produced no further actions.

One exception to full autonomy was with the o3-mini-high model in Aider: often this model did not automatically load files it needed, and would ask the user to add certain files to its context. Whenever *Aider+o3-mini-high* requested a file, we manually added exactly that file (and no additional help), then let it continue. No other agent required such interventions.

D. Evaluation Criteria

After each agent run, we collected the resulting code changes (if any) and deployed/tested the application to judge success. We evaluated outcomes on several criteria:

- **Works (Yes/No):** Did the changes address the issue from the end-user’s perspective? For a feature request, this meant the new functionality works as intended. For a bug, the erroneous behavior was fixed.
- **Linting Check Pass:** We ran the project’s linting scripts. If the agent’ss final code did not pass them, we marked that as a quality issue.
- **User Experience (UX):** We manually inspected if the solution introduced any noticeable UX problems (e.g., a feature works but has a confusing UI or performance lag).
- **Code Quality:** We reviewed the diffs to assess if the solution was implemented in a reasonable and maintainable way. Inefficiencies, unmaintainable code, and obvious bugs in the implementation were noted.

We selected these criteria because they mirror how work performed by a human software engineer is usually evaluated. These qualitative judgments were used to label each successful solution with additional notes (e.g., “works, but suboptimal code” or “works, except fails linting”). Finally, we measured the cost of each solution in USD.

IV. Implementation Details

All agent runs were conducted in a consistent environment. We created a fresh Docker container for each run, which checked out the Aibyss repository at commit `b4e58b2` (to ensure all models started from identical code) and installed the necessary tools (Node.js, Aider, Claude Code, etc.). The agent was then launched inside the container and given the issue text to solve. The prompt given to each agent was uniform: `"Please solve the following issue. Title: <issue title> Description: <issue body>"`. We ensured the project’s dependencies and database (SQLite for this test) were properly set up in each container so that the agent could run the app or tests if it chose to. The Aibyss codebase was about 3.5k lines of TypeScript/JavaScript. Each agent configuration was run on each of the 10 issues, yielding 60 trials in total.

After an agent completed, we committed its changes to a new branch and opened a pull request on GitHub. This allowed us to use continuous integration (CI) results as an additional datapoint. We then manually reviewed and tested the branch as described in the evaluation criteria. All of the PRs created as part of this research can be found on GitHub [21].

V. Results

Table I summarizes the performance of each agent configuration across the 10 issues. We report for each issue whether the agent produced a working solution, along with notes on linting, UX, and code quality. We also report the approximate API cost incurred for that issue’s attempt. A “doesn’t work” or “it didn’t understand the problem” indicates the agent failed to solve the issue.

Looking at the overall success rates (“solved problems” in the Total row), we see a clear ranking of the models. The Claude Sonnet 3.7 with reasoning (both with Aider and Claude Code) solved 5 out of 10 issues, the highest of any configuration. In contrast, DeepSeek-V3 and -R1 solved 2 each, and the o3-mini model solved only 1. The standard *Aider+Claude* (with no reasoning) solved 3. Enabling Claude to use “thinking” (32k tokens context for chain-of-thought) allowed it to solve two additional issues that it failed with a shorter context, showing that reasoning improved performance.

TABLE I
AGENT EVALUATION RESULTS

Problem	Aider 0.75.2 + o3-mini-high 2025-01-31	Aider 0.75.2 + DeepSeek-V3	Aider 0.75.2 + DeepSeek-R1	Aider 0.75.2 + Claude Sonnet 3.7 20250219	Aider 0.78.0 + Claude Sonnet 3.7 20250219 with 32k thinking tokens	Claude Code 0.2.35
1	cost: $0.04 doesn't work	cost: $0.0046 doesn't work	cost: $0.0092 doesn't work	cost: $0.12 ✓**works** linter check fail UX is bad code is bad	cost: $0.20 ✓**works** linter check fail UX is bad ✓**code is good**	cost: $0.2928 ✓**works** ✓**linter check pass** UX is bad ✓**code is good**
2	cost: $0.05 it didn't understand the problem	cost: $0.0037 doesn't work	cost: $0.0070 ✓**works** linter check fail ✓**UX is good** ✓**code is good**	cost: $0.04 doesn't work	cost: $0.07 ✓**works** linter check fail ✓**UX is good** ✓**code is good**	cost: $0.1175 doesn't work
3	cost: $0.03 ✓**works** ✓**linter check pass** ✓**UX is good** ✓**code is good**	cost: $0.0033 ✓**works** ✓**linter check pass** ✓**UX is good** ✓**code is good**	cost: $0.0066 ✓**works** linter check fail ✓**UX is good** ✓**code is good**	cost: $0.04 ✓**works** ✓**linter check pass** ✓**UX is good** ✓**code is good**	cost: $0.07 ✓**works** ✓**linter check pass** ✓**UX is good** ✓**code is good**	cost: $0.1151 ✓**works** ✓**linter check pass** ✓**UX is good** ✓**code is good**
4	cost: $0.07 doesn't work	cost: $0.0043 doesn't work	cost: $0.0070 doesn't work	cost: $0.06 doesn't work	cost: $0.08 doesn't work	cost: $0.4942 doesn't work
5	cost: $0.04 doesn't work	cost: $0.0042 doesn't work	cost: $0.0079 doesn't work	cost: $0.07 doesn't work	cost: $0.08 doesn't work	cost: $0.2085 doesn't work
6	cost: $0.07 doesn't work	cost: $0.0046 it didn't understand the problem	cost: $0.0092 doesn't work	cost: $0.07 it didn't understand the problem	cost: $0.10 it didn't understand the problem	cost: $0.2523 it didn't understand the problem
7	cost: $0.22 doesn't work	cost: $0.0090 doesn't work	cost: $0.02 doesn't work	cost: $0.06 doesn't work	cost: $0.09 ✓**works** linter check fail ✓**UX is good** code is bad	cost: $0.4650 ✓**works** linter check fail minor UX issues code is bad
8	cost: $0.09 doesn't work	cost: $0.0031 ✓**works** ✓**linter check pass** ✓**UX is good** code is bad	cost: $0.0063 doesn't work	cost: $0.04 ✓**works** linter check fail ✓**UX is good** code is bad	cost: $0.07 ✓**works** linter check fail ✓**UX is good** code is bad	cost: $0.1518 ✓**works** ✓**linter check pass** ✓**UX is good** ✓**code is good**
9	cost: $0.08 doesn't work	cost: $0.0062 doesn't work	cost: $0.01 doesn't work	cost: $0.10 doesn't work	cost: $0.12 doesn't work	cost: $0.53 ✓**works** linter check fail UX issues code is bad
10	cost: $0.07 doesn't work	cost: $0.0082 doesn't work	cost: $0.02 doesn't work	cost: $0.05 doesn't work	cost: $0.16 doesn't work	cost: $0.50 doesn't work
Total	cost: $0.76 1/10 solved	**cost: $0.05** 2/10 solved	cost: $0.10 2/10 solved	cost: $0.65 3/10 solved	cost: $1.04 **5/10 solved**	cost: $3.13 **5/10 solved**
Easy solved	1/3	1/3	2/3	2/3	**3/3**	2/3
Medium solved	0/4	0/4	0/4	0/4	**1/4**	**1/4**
Harder solved	0/3	1/3	0/3	1/3	1/3	**2/3**

In terms of difficulty, all agents found the easy issues more approachable: *Aider+Claude 3.7 with thinking* solved all 3 easy tasks, and even the weakest model solved one easy issue. The medium tasks proved challenging: only the Claude Sonnet 3.7 with reasoning (both with Aider and Claude Code) managed to solve 1 out of 4 medium issues. Harder tasks (8, 9, 10) saw partial success: Claude Code solved two (Issues 8 and 9), while *Aider+Claude 3.7 with thinking* and *Aider+DeepSeek-V3* each solved one (Issue 8). No model could handle Issue 10, a complex feature involving tracking and displaying bot errors.

We also observe notable differences in solution quality. For instance, Issue 3 was solved by almost all models, but DeepSeek-R1's solution failed linting due to minor format issues, whereas others passed. In Issue 1, all *Aider+Claude* solutions worked functionally, but they left some UX issues (the page loaded with a flicker in the splitter position) and non-ideal code; the Claude Code agent's solution was slightly better (fixing the linting problem) than *Aider+Claude*'s. The other models did not even reach a working solution for Issue 1. Another example is Issue 6. In Issue 6, none of the agents solved it and most (even Claude) misunderstood the requirement – they implemented a total count instead of a 7-day count for a metric. This shows that LLMs can misread context that a human developer is expected to catch. Additional context or clarification in prompts might be needed for such cases.

Cost-wise, the DeepSeek-V3 and DeepSeek-R1 are very cheap to run totalling $0.05 and $0.10 USD per 10 solutions produced respectfully. While using Claude Sonnet 3.7 was

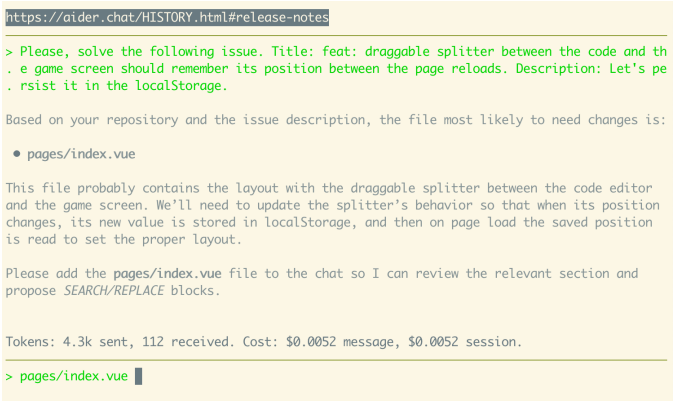

Figure 1. *Aider+o3-mini-high* prompting the user to manually add the file it needs

more expensive, these costs were fairly low in absolute terms (each issue well under $1 for Claude). Notably, Claude Code consumed roughly 3× the tokens of *Aider+Claude*, meaning it took more steps or context per issue. This aligns with Claude Code's behavior of running tests and iterating on a solution autonomously. *Aider+Claude* with reasoning achieved the same 5/10 solves at one-third the cost of Claude Code. In a scenario where API cost is a concern, the improvement in solution quality by Claude Code may not currently justify its higher cost.

VI. Discussion

To better understand the experimental outcomes, we performed a qualitative analysis of each agent's behavior and the solutions (or attempts) they produced. Here we discuss key insights and failure modes.

A. Need for Context Awareness

One striking limitation was observed with o3-mini-high. This agent often failed to load relevant files autonomously. In multiple issues it would stop and ask for a file (for example, it did not automatically open the file containing a function mentioned in the issue). We had to manually provide the file for it to proceed. This behavior is depicted in Figure 1, which shows a screenshot of the *Aider+o3-mini* agent prompting the user to add a file to the context. Notably, none of the other models exhibited this limitation — they have added needed files to the context via the Aider agent's capabilities. In addition to this, the *Aider+o3-mini-high* demonstrated lack of context awareness that significantly impeded its performance. Even after adding files, its solutions were usually incomplete or incorrect.

B. Common Reasoning Errors

Several agents made similar mistakes on certain issues, suggesting the problem might lie in the prompt or the environment rather than idiosyncrasies of one model. For example, in Issue 4 (a medium bug fix involving adding a TypeScript module import), the DeepSeek and o3-mini-high models all attempted to import a database object incorrectly (they used a wrong path), possibly picking up a pattern from elsewhere in code. This parallel behavior implies that the initial system might have led the models down a similar path, or they all latently "agreed" on a plausible but wrong solution. It highlights that autonomous agents might need better guardrails or self-checks for such predictable pitfalls. A possible remedy could be incorporating static analysis: e.g., after a code edit, have the agent verify imports or run a quick compile step (which Claude Code was usually doing).

Another pervasive issue was misunderstanding of requirements. Issue 6 (add a weekly count to the rating table) was misinterpreted by every model as a total count. This happened because the issue title was somewhat ambiguous and the description didn't explicitly mention the 7-day window (it relied on context that all other stats in that table were weekly). Our LLMs did not infer that context. This kind of mistake is hard for an agent to catch without more project knowledge. It suggests that for certain tasks, an autonomous agent might benefit from a mechanism to ask clarifying questions – something we did not allow in this study.

C. Behavior of Claude Code vs. Aider

Using the same model (Claude Sonnet 3.7), the Claude Code agent and the Aider agent exhibited different styles. Claude Code was much more thorough: it would run the project's tests when available, and even run the linter, essentially simulating what a careful developer would do. Claude Code was also the only agent that was actually creating the Prisma database migrations in addition to changing the schema. The downside was that Claude Code consumed more time and tokens.

D. Successful Case Study (Issue 7 - Sandbox Sprites Toggle)

This was a medium difficulty feature that only the Claude-based agents solved. The task was to add a user option to replace graphical sprites with simple circles in the game (for debugging). The *Aider+Claude with reasoning* agent produced a clean solution: it introduced a button to switch to and from the debug mode and a corresponding logic for the toggle. The implementation was not trivial – it required understanding how the rendering loop worked. The other models failed here likely because they got confused by the rendering code. *Aider+Claude with reasoning*'s success in Issue 7 demonstrates the benefit of reasoning. Interestingly, Claude Code's working solution for the same issue was a bit messy (the circle drawing code is unnecessary complex and is repeated in two different places). This suggests that while Claude Code's strategy of iterative refinement is helpful, it doesn't guarantee a better solution design.

E. Partial Success and Limitations (Issue 9 - Code Versioning)

One of the "harder" issues (Issue 9) was to implement code version tracking and allow reverting to previous versions. Claude Code was the only agent to achieve a working solution here. It modified the bot code storing logic to store versions, added a new API to list code versions, built a frontend component to list versions, implemented logic to restore the older

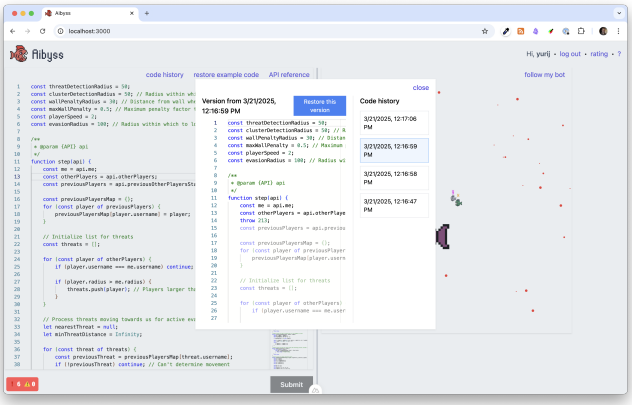

Figure 2. UI added by Claude Code for code versioning (Issue 9)

versions of the bot code, and wired it all together correctly. The resulting UI can be seen in Figure 2, which shows the new interface element listing past versions of a user's code with a revert option. While this solution technically worked (the user could revert to past code states), we discovered a serious performance problem: the agent's implementation loaded all stored code versions of all users into memory on server startup (a costly approach). This highlights a scenario where the agent solved the letter of the request but not the spirit of good software engineering – a human developer would likely avoid loading all of the source code into memory. This underscores a fundamental limitation of current LLM agents: they often don't evaluate trade-offs beyond immediate correctness. We marked this solution as not production-ready due to the RAM overhead. It needed human refactoring to be acceptable. Nonetheless, the fact Claude Code managed to implement a multi-step full-stack feature is impressive. For future agents, improving their considerations for performance might be beneficial, although that is a very difficult general problem.

VII. Conclusion

In this paper, we explored the frontier of using state-of-the-art LLMs as autonomous software engineers on real development tasks. Through experiments on ten diverse issues in an open-source project, we found that today's top models can partially fulfill the role of a developer: they wrote code that solved about half of the tested issues without any human assistance. This is a notable achievement and highlights the rapid progress in LLM capabilities for software engineering. However, our study also reveals the clear limitations and challenges that remain:

- Autonomous agents are not yet reliable across the board. They struggled especially with more complex or ambiguous tasks, and often produced suboptimal solutions even when they met the basic requirements.
- Reasoning and chain-of-thought prompting greatly influence success. Utilizing the reasoning ability of Claude Sonnet 3.7 improved outcomes in our trials.
- There is a need for built-in validation and refinement. Incorporating test execution, linting, and iterative self-correction (as Claude Code does) helped catch mistakes. Future agents should leverage all available verification tools (compilation, static analysis, tests) to ensure higher quality outputs.
- Certain errors, like misinterpreting the true intent of a requirement or making inefficient design choices, are currently beyond the examined agents' capacity to avoid. These will likely require changes to the agent's architecture or a human-in-the-loop to guide the agent.

Despite these limitations, the trend is very encouraging. We expect that with each iteration, the gap on what tasks are solvable autonomously will narrow. In practical terms, autonomous LLM agents could already take on some tedious parts of development (like writing boilerplate code, fixing simple bugs, updating configurations), freeing human developers to focus on higher-level design and complex problem-solving.

As future work, our immediate next steps include:

- **Evaluating newer models:** We plan to test open-source QwQ-32B with Aider to see if it can match Claude's performance. If successful, this could open the door to more accessible autonomous coding (not relying on closed APIs).
- **Architect-editor agent design:** We will experiment with an "architect" mode in Aider, where one model (or one prompting style) is used to outline the solution (select files to change, perhaps write pseudo-code or steps), and another model is used as the "coder" to implement those steps.
- **Scaling to more tasks and projects:** Our current test set is small. We want to expand the evaluation to include a wider variety of issues (UI-heavy issues, algorithmic challenges, integration tasks) and on different projects (perhaps some Python backend projects, mobile app issues, etc.). This will paint a fuller picture of where autonomous LLMs excel and where they fail in software engineering.

In conclusion, state-of-the-art LLMs, when coupled with a suitable agent framework, are beginning to demonstrate practical utility in automating segments of software development in a fully unsupervised manner. They function as knowledgeable but flawed junior developers: capable of writing code and solving problems in familiar contexts, yet prone to mistakes that require oversight. By continuing to improve LLM reasoning, integrating robust self-checks, and using clever orchestrations of multiple models, we move closer to a future where AI agents could handle routine programming tasks autonomously. Such a development could significantly accelerate software engineering workflows, allowing human developers to push the boundaries of innovation with the grunt work delegated to our AI collaborators.

REFERENCES

[1] OpenAI, *OpenAI o3-mini*, version 2025-01-31, 2025. [Online]. Available: https://openai.com/index/openai-o3-mini/.

[2] DeepSeek-AI et al., "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," 2025. arXiv: 2501.12948 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2501.12948.

[3] GitHub, *GitHub Copilot*, Oct. 2021. [Online]. Available: https://github.com/features/copilot.

[4] N. Nguyen and S. Nadi, "An empirical evaluation of github copilot's code suggestions," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, 2022, pp. 1–5. DOI: 10.1145/3524842.3528470.

[5] Anysphere-Inc, *Cursor composer*, 2024. [Online]. Available: https://docs.cursor.com/composer.

[6] S. Yao et al., "React: Synergizing reasoning and acting in language models," 2023. arXiv: 2210.03629 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2210.03629.

[7] Significant-Gravitas, *AutoGPT*. [Online]. Available: https://github.com/Significant-Gravitas/AutoGPT.

[8] J. He, C. Treude, and D. Lo, *Llm-based multi-agent systems for software engineering: Literature review, vision and the road ahead*, 2024. arXiv: 2404.04834 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2404.04834.

[9] H. Yang, S. Yue, and Y. He, *Auto-gpt for online decision making: Benchmarks and additional opinions*, 2023. arXiv: 2306.02224 [cs.AI]. [Online]. Available: https://arxiv.org/abs/2306.02224.

[10] X. Hou et al., "Large language models for software engineering: A systematic literature review," 2024. arXiv: 2308.10620 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2308.10620.

[11] A. F. Khan et al., "Lads: Leveraging llms for ai-driven devops," 2025. arXiv: 2502.20825 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2502.20825.

[12] M. Sapkal et al., "Ai-driven software patch management system," *SSRN Electronic Journal*, Jan. 2025. DOI: 10.2139/ssrn.5086731.

[13] Anthropic, "Claude 3.7 sonnet and claude code," Feb. 24, 2025. [Online]. Available: https://www.anthropic.com/news/claude-3-7-sonnet.

[14] DeepSeek-AI et al., "Deepseek-v3 technical report," 2025. arXiv: 2412.19437 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2412.19437.

[15] Aider-AI, *Aider*, version 0.75.2, 2025. [retrieved: Mar. 2025]. [Online]. Available: https://github.com/Aider-AI/aider.

[16] M. Chen et al., "Evaluating large language models trained on code," 2021. arXiv: 2107.03374 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2107.03374.

[17] Y. Li et al., "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, Dec. 2022, ISSN: 1095-9203. DOI: 10.1126/science.abq1158. [Online]. Available: http://dx.doi.org/10.1126/science.abq1158.

[18] J. Wei et al., *Chain-of-thought prompting elicits reasoning in large language models*, 2023. arXiv: 2201.11903 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2201.11903.

[19] Move-Fast-and-Break-Things, *Aibyss*, 2025. [retrieved: Mar. 2025]. [Online]. Available: https://github.com/move-fast-and-break-things/aibyss.

[20] Y. Mikhalevich, *Aibyss: Issues selected for the "practical applications of state-of-the-art large language models to solve real-world software engineering problems autonomously" research*, 2025. [Online]. Available: https://github.com/move-fast-and-break-things/aibyss/issues?q=is%3Aissue%20label%3Aai-agents-evaluation-2025-03.

[21] Y. Mikhalevich, *Aibyss: Prs created by ai agents as part of the "practical applications of state-of-the-art large language models to solve real-world software engineering problems autonomously" research*, 2025. [Online]. Available: https://github.com/move-fast-and-break-things/aibyss/pulls?q=is%3Apr+label%3Aai-agents-evaluation-2025-03+.