Promises, Perils, and (Timely) Heuristics for Mining Coding Agent Activity

Romain Robbes
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI
Bordeaux, France
romain.robbes@labri.fr

Théo Matricon
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
theo.matricon@inria.fr

Thomas Degueule
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI
Bordeaux, France
thomas.degueule@labri.fr

Andre Hora
Department of Computer Science, UFMG
Belo Horizonte, Brazil
andrehora@dcc.ufmg.br

Stefano Zacchiroli
LTCI, Télécom Paris, Institut Polytechnique de Paris
Palaiseau, France
stefano.zacchiroli@telecom-paris.fr

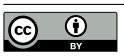
Abstract

In 2025, coding agents have seen a very rapid adoption. Coding agents leverage Large Language Models (LLMs) in ways that are markedly different from LLM-based code completion, making their study critical. Moreover, unlike LLM-based completion, coding agents leave visible traces in software repositories, enabling the use of MSR techniques to study their impact on SE practices. This paper documents the promises, perils, and heuristics that we have gathered from studying coding agent activity on GitHub.

ACM Reference Format:
Romain Robbes, Théo Matricon, Thomas Degueule, Andre Hora, and Stefano Zacchiroli. 2026. Promises, Perils, and (Timely) Heuristics for Mining Coding Agent Activity. In *23rd International Conference on Mining Software Repositories (MSR '26), April 13–14, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3793302.3793375

1 Introduction

Large Language Models (LLMs) have had a major impact on the practice of Software Engineering [19]. One of the first LLMs to be used in practice was Codex [12], which was the model powering the initial version of GitHub's coding assistant, Copilot, starting in 2021. Codex and Copilot immediately triggered a flurry of empirical studies about both the capacities and limitations of the underlying model, and how the tool impacted software development (Section 2 details these studies). Unfortunately, few of these studies used Mining Software Repositories (MSR) techniques; they relied instead on controlled experiments [44], observation studies [6], or surveys [36]. While valuable, the lack of large-scale studies of the phenomenon limits the generalizability of the findings [53]. The dearth of MSR studies is for good reason: since coding assistant are used interactively in a code editor, it is essentially impossible to know whether they were used to author code in a repository, leaving studies with very rough time-based heuristics as the only option [22], or studies of other LLMs uses [57].



MSR '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2474-9/2026/04
https://doi.org/10.1145/3793302.3793375

In 2025, a second generation of LLM-powered tools emerged and is seeing rapid adoption: coding agents. (See Section 3 for a detailed background on coding agents.) If coding assistants are used interactively to complete lines or blocks of code, the scope of coding agents is drastically broader. The key characteristic of coding agents is *autonomy*. In ideal scenarios, a developer delegates an entire task to a coding agent, which performs a sequence of actions in the code base to accomplish the task, producing a code change that the developer can review, as a human developer would. The arrival of reasoning models in late 2024, and the focus of AI industrial labs to improve tool-calling capabilities allowed agents to quickly transition from promising academic work in 2024 [10, 62, 65] all the way to an established tool category in 2025, where all major AI labs, IDE vendors, and multiple startups propose products. Adoption has quickly followed: our study of the topic, based on the heuristics presented here, found that, as of 01/11/2025, between 15.85 and 22.60% of studied GitHub projects adopted coding agents to some extent, a very large number for products that have been for the vast majority released this year [48]. Interest and adoption are very high because of the potential of coding agents. While the scope of the tasks for which they can be used in full autonomy is limited today, improvements in model capabilities could change the situation. If this promise materializes, this could represent a greater change to the software engineering practice than the first generation of LLM tools, and the arrival of CASE tools many decades ago.

Given the potential impact of coding agents, studying how they are used in practice is extremely important. MSR studies have a major role to play here, providing an alternative viewpoint to controlled experiments [7] or observational studies [31]. It turns out that coding agents, by their very nature, leave much more traces in software repositories than code completion LLMs did, making their study via MSR techniques possible. Section 4 documents the heuristics we were able to identify in order to study this phenomenon, constituting the *first contribution* of this paper.

While coding agents leave traces in repositories that we can study, these come with important caveats: based on our experience studying agents, we have identified several of these, including that the observed data is partial, comes from multiple, heterogeneous agents, and is rapidly changing. On the other hand, there is ample space for future work studying the impact of coding agents, how

they are used, how changes to the underlying LLMs affect them, and potentially changes to SE practices. In addition, as coding agent adoption is high, detecting and excluding coding agent activity will be necessary for MSR studies focusing on humans. We expand on the promises, perils and mitigations we formulated in Section 5, which constitute our *second contribution*.

Finally, as one of the perils we identify is the rapid rate of change of coding agents, no paper will remain the definitive resource for studying coding agent traces in software repositories for long. While we expect the promises and perils to remain quite stable over time, heuristics to identify agent activity will change often. This is why we call for the community to join us in documenting heuristics in a community-maintained repository, which we have bootstrapped [47]. The repository also contains sample datasets of traces found for specific coding agents, in order for researchers to investigate what coding agent traces look like and help them plan their studies before delving in massive amounts of real-world data. This heuristic and data repository is our *third contribution*; Section 6 details it. We discuss the limitations and implications of this work in Section 7, before concluding with Section 8.

2 Related work

Generative AI for SE and Coding assistants such as Github Copilot have been extensively studied, using a variety of techniques; we cite but a few, with a more specific focus on MSR studies.

2.1 Studies of coding assistants

Controlled experiments. Early studies of developer productivity showed mixed results. Peng et al. [44] found 95 programmers completed HTTP servers 55% faster using Copilot (71 vs 161 minutes). However, Imai [27] found higher productivity but lower code quality with Copilot versus pair programming, while Vaithilingam et al. [58] found participants failed tasks more often with Copilot than Intellisense due to incorrect code, despite preferring Copilot. Similarly, studies focused on code security yielded contrasting results. Sandoval et al. [49] and Asare et al. [5] found little or no differences in terms of security aspects, while Perry et al. [45] found AI-assisted code was less secure across 6 tasks.

Qualitative studies. Barke et al. [6] identified two usage modes: in *acceleration*, developers prefer small, quickly-validated suggestions; in *exploration*, they prefer larger suggestions as starting points when working with unfamiliar APIs. Mozannar et al. [40] observed 21 programmers; they spent 22% of time evaluating suggestions versus 14% writing code, suggesting acceptance rates underestimate cognitive overhead. Wang et al. [59] interviewed 17 developers who emphasized understanding Copilot's limitations and expressed concerns about negative impacts on learning.

2.2 MSR studies of generative AI

Telemetry. Telemetry-based approaches are possible in scenarios where the deployment can be carefully controlled, such as in companies developing the tools themselves. Murali et al. [41] reported Meta's CodeCompose achieved 22% acceptance rate and generated 8% of company-wide code. Ziegler et al. [66] found acceptance rate was the best predictor of perceived productivity among 2,000 Copilot users. Izadi et al. [29] developed an IDE extension and collected interactions of 1200 users to analyze model failures qualitatively.

Usage in software repositories. Tufano et al. [57] searched for explicit usage of ChatGPT in GitHub commits, pull requests, and issues. They analyzed ChatGPT usage traces in 467 GitHub instances, developing a taxonomy of 45 software engineering tasks including feature implementation, documentation, software quality, and development processes. Some usage patterns were unexpected, such as assistance in motivating proposed changes.

Xiao et al. [63] also look for usage of ChatGPT or Copilot when explicitly written by developers in software artifacts. They analyze more than 1,200 Generative AI usages in a qualitative study, focusing on the type of tasks they were used in; in addition they also analyzed instances of guidelines to regulate the usage of Generative AI in several projects, and also analyzed churn, finding overall no significant change in churn post-adoption

This result is in contrast to the white papers authored by GitClear [22, 23], where they compared code churn before and after the introduction of Copilot, taken as a marker for the first usages of Generative AI in software repositories. The studies find a higher rate of churn as time passes and (presumably) generative AI adoption increases, and a higher incidence of code duplication in the repositories they analyze. However, in the absence of specific markers to detect the usage of Generative AI it is more difficult to untangle the effects of Generative AI from other factors.

MSR mining challenges. Xiao et al. [64] curated a dataset of shared ChatGPT conversations with ChatGPT on software engineering topics. This dataset was used for the 2024 challenge, in which a variety of aspects were analyzed by 18 challenge submissions. Concurrent to this work, Li et al. [34] curated a dataset of agent pull requests, to be used for the 2026 challenge.

Our work. Based on the heuristics presented in this paper, we have investigated the adoption of coding agents on GitHub [48], and their propensity to write over-mocked tests [26].

2.3 Studies of coding agents

Given their recency, studies of coding agents are still few and far-between. Given the small amount of studies and their mixed results, we see this as an ideal ground for MSR studies to complement this emerging body of evidence.

Becker et al. [7] performed a controlled experiment of software developers using Cursor. Developers believed that using Cursor reduced completion time of tasks by 20%, but it actually increased task completion time by 19%. Many potential factors could explain this slowdown. Notable ones include less than 44% of code generations accepted, 9% of the time spent cleaning AI code. Importantly, the agent used had limited capabilities, and was used in interactive rather than autonomous mode.

Kumar et al. [31] performed an observation study of 19 developers using Cursor. There were two categories of participants: those who tried to delegate their entire task to the agent, and those that collaborated with the agent to decompose the task in subtasks, and have the agent solve each subtask. Participants provided the agent several kinds of information, notably contextual information from

the task description and the environment, and expert information based on prior repository knowledge. Indeed, the main barrier to effective agent use was when it lacked this tacit knowledge.

Bouzenia and Pradel [11] analyzed 120 agent interaction logs as they were attempting to solve bugs from the SWE-bench benchmark. While these traces come from less established agents, and are not the result of genuine human interactions but rather byproducts of a benchmark, they provide useful insights, highlighting the differences in behaviour between the agents in terms of the length of their interaction traces, the type of interactions in them, and the differences between succesful and failing attempts.

3 Coding agents

While there are multiple definitions of an agent, we use the following one: an agent is an LLM executing in a loop in order to fulfill a goal, and that is provided access to tools in order to access its environments. The agent loop is then conceptually simple:

(1) query the LLM with a sequence of interactions (initially, just the task description);
(2) parse the response for the presence of tool usages and end of task;
(3) if tool calls are present: optionally ask the user for permission before executing sensitive tools;
(4) if task has not ended: add the response to the sequence of interactions, replacing tool calls with their results.

The loop executes until the LLM determines it has completed the task, in which case its response indicates it, ending the conversation.

The first component contributing to the agent is the LLM itself. Modern LLMs have improved significantly on important capabilities necessary for agentic use cases. In particular, the capability to reliably generate structured outputs is crucial for reliable tool calling, and the abilities of reasoning models to generate and exploit long reasoning traces are critical to solve longer tasks.

The agent's harness is the second component of an agent. The harness is responsible for providing the model with the list of available tools, and detecting when the model generates a tool call (using a structured format, such as JSON, containing e.g. the name of the tool and its possible parameters).

Tool use. Access to tools is a key differentiator with coding assistants. While coding assistants are provided a context for their completion, it is computed beforehand by the IDE. In contrast, tool use enables much more autonomous workflows. For example:

- Using search tools and file reading tools, an agent can explore the code base and dynamically find the right context for its task, instead of relying on the IDE's context;
- Using file writing tools, a compiler, and executing test cases, the agent can iterate on its solution until it is correct, leveraging error messages (if any) to improve it;
- If the agent has determined that it has completed the task, it may commit changes or author a pull request, summarizing the changes in the process.

Tools range from the simple to very complex: some agents use little more than shell access (the agent writes shell commands to support a variety of tools, e.g., calling the compiler, running tests, using git), up to using e.g. the model context protocol to interface with complex systems such as controlling a web browser, querying a database, or using a ticket repository.

With these capabilities coding agents have been employed to perform a variety of tasks, ranging from updating dependencies, fixing simple bugs, or writing documentation, up to implementing larger scoped features and their tests, refactor code, or porting software from one programming language to another.

Autonomy and oversight. Agents can have varying degrees of autonomy, depending on the sensitivity of the task, the degree of oversight a developer wants to invest, the environment setup, and the individual workflow of each agent. Depending on their prompt, agent may ask developers for feedback on the task (e.g., write a plan to implement a feature and submit it for feedback). Moreover, most agents allow human oversight by interrupting the agent's activity and add additional guidance before resuming.

Agents may require human oversight when they are using tools; each tool can have individual permissions, ranging from allowing all uses of a given tool, allowing the use on a case-by-case basis (with human approval), to systematically forbidding a tool. A key advantage of allowing tool use without human intervention is that it makes the agent fully autonomous, which frees up the developer to work on other tasks, rather than closely steering the agent.

Uses. There are a breadth of uses for coding agents, ranging from "vibe coding" [17, 20] with little to no supervision, up to serious practice with much more oversight [31]. Coding agents already have the potential to help developers being more productive (although for early coding agents, the evidence is mixed [7]), and to feel less alienated in their work (e.g., by spending less time on menial tasks). Uses may evolve over time: in particular, technical solutions (e.g., development in a container [14, 16]) make full autonomy less risky as the impact of the agent is controlled. Some advocate to leverage the agent's autonomy to the maximum, such as the use of parallel agents working on multiple tasks at once.

4 Traces and heuristics

For our coding agent studies [26, 48], we have gathered a set of heuristics that detect their presence in software repositories. We focus on GitHub, the most common coding platform; furthermore, due to its widespread use, some agents such as Copilot leverage GitHub functionalities (e.g., activity in pull requests). We derive our heuristics from an extensive manual investigation of known agents, involving checking their documentation for mentions of specific artifacts, and analysis of repositories identified as using agents for visible traces. This leads to a tentative list, which we validate with targeted GitHub searches and manual checks to select heuristics that return enough results to be useful (e.g., hundreds), and do not have too many false positives upon inspection of the results.

Table 1 shows the heuristics we have detected so far, across multiple categories of GitHub artifacts. Importantly, we do not expect all of these specific heuristics to remain unchanged in the long term (see Peril 4 of velocity). This is why we distill these heuristics in a set of more general detection strategies, across several categories of GitHub artifacts, that we expand on below.

Table 1: Agent detection heuristics and approximate GitHub match counts on 20/10/25 (click to browse results; file queries require GitHub login). The table includes the top 10 agents and generic heuristics, limiting to heuristics with more than 500 matches. See a more complete and up to date table in the repository [47] (direct link: click here).

Tool	File	Commit co-author or author	Branches	Labels	Total
Generic	**AGENTS.md (37.5K)**	-	-	ai-generated (1.9K)	39.4K
Aider		**aider (44.7K) aider@aider.chat (40.8K)**	-	-	85.9K
Claude Code	**CLAUDE.md (110.0K) .claude/ (141.0K)** *.github/workflows/claude (5.4K)*	**Claude (author, 38.2K) noreply@anthropic.com (2.7M)** claude@anthropic.com (2.3K)	*claude/ (20.7K)*	-	3.0M
Cline	.clinerules (1.7K) .cline/ (558) *memory-bank/ (23.0K)* memory_bank/ (1.6K)	*cline (11.0K)* cline@example.com (3.3K)	-	-	41.1K
Codex	.codex/ (3.3K)		**codex/ (2.1M)**	**codex (2.3M)**	4.4M
Copilot	**copilot-instructions.md (45.3K)** *.github/instructions/ (13.3K)* .copilot/ (1.3K) .github/workflows/copilot/ (1.6K)	**Copilot (190.0K) copilot-swe-agent (35.5K)**	**copilot/ (338.0K)**	-	626.0K
Cursor	**.cursor/ (98.3K)** *.cursorrules (16.6K)* CURSOR.md (1.3K)	**cursor (206.0K) cursoragent@cursor.com (40.7K)**	**cursor/ (209.0K)**	-	571.9K
Devin	-	*devin-ai-integration (14.8K)*	**devin/ (49.8K)**	-	64.6K
Gemini	*GEMINI.md (8.4K)* .gemini/ (4.8K)	*gemini-code-assist (22.5K)* gemini-cli (3.5K) *Gemini 2.5 Pro (11.2K)* Gemini 2.5 Flash (4.2K)	-	-	54.6K
Kiro	**.kiro/ (42.2K)**	-	-	-	42.2K
OpenHands		**openhands@all-hands.dev (35.9K) openhands-agent (author, 34.5K)**		-	71.1K

Promise 1: Presence of Traces

Coding agents leave many visible traces in software repositories, in files, commits, issues, and pull requests, detectable by specific heuristics, and more generic detection strategies.

4.1 Files

Several types of files indicate that a coding agent is used in a repository: *Configuration files* toggle a variety of settings; *Rules* and *Guidance* files affect their behavior when working in the repository.

Configuration files. Like many tools, agents can be configured, and these settings are often found in configuration files that follow naming conventions. A repository containing such a configuration file indicates that an agent has been set up at least once in the repository. Moreover, depending on the specific agent, some of the settings inside the file can offer insights on how the agent works.

In particular, some configuration files will contain information on the permissions the agent has been granted, i.e., which actions it can take autonomously (e.g. reading files), which actions require developer approval (e.g. writing files), and which actions are not allowed (e.g. accessing the internet). Studies may use these settings to compare repositories where agents are granted high autonomy, versus repositories where agents are supervised more closely.

Other interesting settings may include the availability of MCP servers, that provide agents with additional tools. These settings provide valuable context on the usage of a given agent and its capabilities. Other settings can regulate how the agent advertises its presence via commits. In general though, the settings are highly tool specific (see the Peril 3 of diversity). Settings can be found either in a specific repository, but also, as some developers do, in repositories storing configuration specifically (dotfiles).

Rules and Guidance files are more specific to coding agents. They contain natural language instructions that the agent should comply with when working in the repository. Almost always, these files follow the markdown format. These files are concatenated to the LLM's prompt to affect its behavior. The files might have a project-level scope (applying to the entire project), or a smaller scope (e.g., a module), in which case they are included in the prompt only if the agent is working in this module. These files may be human-written, but it is also common that they are generated by the agent.

Rules provide specific natural language instructions that the agent should follow. Example include following certain coding conventions, or how to perform specific actions. Some rules might be very generic, other very specific to the project (documenting a specific convention, running a command) [1].

Guidance files also primarily contain natural language, but have a broader scope. They may document higher-level knowledge about a project or a module [2], such as its architecture and its main components, or provide rationale for certain decisions. They may contain detailed task descriptions highlighting the steps to solve a specific task [3]. In both cases, the files may be generated by the agent as summaries of the code base, or plans to implement a task. Guidance files may also describe tactics and strategies that the LLM should use to tackle problems in general [4]. The space of instructions is very large, and is a promising area of study (see Promise on Potential).

Rules and guidance file encode *tacit knowledge* about the project, or software development, or even problem solving in general. As such, they are valuable artifacts that are useful beyond a single developer. This is why they are often committed in the repository to be shared, although some developers elect not to commit them.

Detection. Many of these files follow established naming conventions, making them easy to detect (e.g., `CLAUDE.md`). In some cases, the files themselves may vary, but the directory they are in follows a naming convention (e.g., `.cursor/`). However, some naming patterns can have false positives (e.g., Aider guidance file is often called `CONVENTIONS.md`, but many repositories use this to document coding conventions for developers, not agents).

4.2 Commits

Several agents (but not all) can be given the capability to commit on behalf of the developer, and may (or may not) advertise it. Agents can either be given access to a specific tool to commit, or (more commonly), simply use their shell tool to commit on behalf of

[1] Click to browse a simple cursor rules file
[2] High-level project description example
[3] Example detailed implementation plan
[4] Example guidance with strategy

the developer. Agents authoring a commit will often compose the commit message themselves and summarize changes, rather than asking the user. They can advertise their presence in several ways:

- By adding themselves as a *co-author* of the commit, with the user as main author. In this case, they add the "`Co-authored-by:`" trailer to the commit. Of note, agents may not adhere to the convention exactly, occasionally using different casings.[5]
- Less commonly, as the author of the commit, similarly to how bots such as dependabot operate.
- Some agents such as Claude Code also add additional trailers to the commit, such as "`Generated by: Claude`".

Depending on the agent or developer workflow, the commits may be authored by the agent completely autonomously, or include developer oversight or edits.

Detection. Maintaining a list of known author/co-author names makes the detection of agent-authored commits straightforward. More advanced heuristics might be possible.

4.3 Pull requests

Some agents workflows include Pull Requests (PR): the agent will author one or more commits, and additionally author a PR that is submitted for review on GitHub, with an appropriate summary. A PR is richer than a commit; notably, it allows other GitHub users to comment on the PR, and for developers to decide to merge the PR or not. These agents may revise their work when a developer comments on the PR and indicates that additional changes should be made, often also appending a comment in response (in addition to a commit). Bots submitting PRs are not new (e.g. dependabot), but the PRs submitted by agents can have a much broader scope. Depending on user instructions and the capabilities of the agents, submitted PRs can range from simple bug fixes to entire new features.[6]

Detection. Some agents such as Codex follow patterns when working with PRs, notably by advertising their presence in the name of the branch used in the repository (e.g. `codex/branch-name`). They may also assign labels to their PR. In addition, agents may submit their PR either as the user that triggered the agent, or as a specific user that can be detected. They may, in some cases, provide links to the coding agent session that led to the PR; the links themselves are, unfortunately, often inaccessible. Finally, agents may interact with developers in the PR itself via comments.

4.4 Issues

Issues submitted by and assigned to bots are not new. Some agents support the same kind of interactions, especially when they act as a specific user or are associated with GitHub actions. The issue in this case acts as a task description for the agent. This is interesting since it gives additional visibility on the process. As documented in the Peril of Partial Observability, the initial prompt given to the agent is not necessarily documented; issues solve this in some case.[7]

Detection. Participation in issues can be detected if the user associated with the agent is mentioned or responds in the issue.

[5]Example co-authored commit
[6]Example PR with user/agent interaction, implementing a feature
[7]Example with a user assigning an issue to OpenHands, which creates a PR in response

4.5 Users

Like other bots, some agents are associated with specific GitHub users. Although users may not carry much information, their presence itself is a signal. Users allow additional functionality, such as assigning an issue to an agent, or allowing it to comment on PRs.

Detection. Via a list of known agent users.

5 Promises, perils and mitigations

5.1 Coding agent adoption is real

Promise 2: Adoption is real and significant

As of October 2025, the adoption of coding agents is already clearly visible on GitHub.

As of mid-October 2025, our study of the extent of adoption of coding agents on GitHub estimates that between 15 to 19% of GitHub projects show traces of coding agents. Furthermore, the adoption is growing steeply. Thus, we expect future adoption to grow significantly in the coming months and years, and for coding agents to take a larger place in software development.

In addition, our study identified repositories with a wide variety of coding agent use, from the experimental (a handful of commits co-authored by a coding agent), to the pervasive (repositories where the *majority* of commits are co-authored by coding agents).

To give additional insight on the wealth of information available, Table 1 provides some counts on the frequency of each heuristics via simple queries to GitHub's web interface in order to estimate their count. While this count is not precise, does not offer us information about the number of repositories in which the heuristics are found (i.e., some heuristics may match with a repository multiple times), and may contain false positives, we can already get a sense of the adoption. Notably, the most popular tools have been visibly used to author *millions* of commits and pull requests.

Promise 3: Study potential

The visible traces left by agents give us a window in their impact on software engineering.

For the first time, agent-based automation is visible, *at scale*, thanks to the traces generated by coding agents. Previous studies using MSR techniques, such as the one of Tufano et al. [57] or the one of Xiao et al. [63], relied on deliberate annotations by developers; while very valuable, these studies could not have the same level of exhaustiveness. Coding agents traces allow us to finally reliably detect LLM-generated code, unveiling a level of automation that was previously hidden. Given the wealth of traces available, we think that many MSR studies could be done on the adoption and impact of coding agents in software engineering.

For instance, while the GitClear whitepapers [22, 23], in the absence of stronger signals, used a time-based delimiter ("before Copilot" vs "after Copilot"), mining coding agent traces allow us to compare repositories that explicitly use agents with those that do not. The traces are also much more precise, as they have fine degree of granularity, down to individual commits and PRs.

Studying code produced with coding agents will allow us to have a clearer understanding of the impact of coding agents on

issues such as code quality, or code defects. Importantly, longitudinal analyses of repositories can study the longer-term impact of coding agent use. Analyzing commits and PRs over time will give us insights on the true impact of coding agents on developer productivity. Analyzing the success of PRs (in terms of merge rates, rework, etc.), can give us insights on the factors that influence the success or failure of coding agents. These may be broader than individual PRs: analyses of characteristics of repositories (e.g. how does a repository's code quality affect agent performance?) and their impact on the use and success of coding agents will be insightful. For instance, we expected to find higher adoption of coding agents in smaller repositories; we were surprised to find that larger repositories had comparable (if not higher) rates of adoption [48].

Promise 4: Mining developer–agent interactions

The visible traces left by agents offer a unique opportunity to observe how *developers* interact with AI code models.

While interactions between developers and LLMs were previously confined to private chat environments, the emergence of coding agents as active participants in shared social spaces (e.g. pull requests, issues, code reviews) presents new opportunities to systematically study human–AI collaboration. The interaction traces left by these agents capture not only their outputs but also developers' responses: accepting, rejecting, or refining their contributions. Unlike conventional LLM use, these public traces reveal how developers articulate their needs, review and correct the agent contributions, and steer their behavior. They can also reveal how developers might restructure their workflows and documentation to "make space" for agents to effectively contribute to their projects.

Mining these interactions can enhance our understanding of the task distribution between developers and agents, identifying which activities developers delegate to agents and which they retain under their control. It also enables analysis of review dynamics, including how developers critique or modify agent outputs and, conversely, how agents propose changes to human-written code. More broadly, these records provide a unique perspective on the successes and failures of coding agents in practice, exposing frictions and pain points in developer–agent interactions. This information will inform the design of new agents and allow for a better integration of their capabilities into developer spaces and workflows.

Promise 5: Study agent knowledge

The files that coding agents leverage in their tasks are a very rich source of information.

Agent rules and guidance files are critical for coding agents to work well. As mentioned in Section 3, these files can contain several kinds of valuable knowledge: 1) rules and conventions that agents should follow; 2) more general knowledge about the repository (e.g., high-level organization); 3) detailed task descriptions and plans; 4) general tactics and strategies to solve problems. While this is an early classification, we believe that a full qualitative study of the contents of these files will be extremely instructive and valuable to know more about how agents are used in practice. For instance, while most repositories have straightforward guidance files, some users go to considerable lengths to define extensive and sophisticated guidance, covering a variety of dimensions (e.g. documenting project context, technical choices, architectural patterns, etc.).

After having built a better understanding of the knowledge in these files, many other studies are possible. Possible studies include for instance studies of the effectiveness of the rules encoded in the files (e.g., to which extent do coding agents follow the conventions?), or whether they impact coding agent success (e.g., to which extent do high-quality specifications influence coding agent success?).

Finally, agent configuration could also allow studies, e.g. whether and how the availability of advanced tools via the Model Context Protocol or similar mechanisms is leveraged by agents.

5.2 Agent detection: the devil is in the details

Peril 1: Partial observability

We can not observe all agent activity, only parts of it.

Mitigations: (1) Additional heuristics (2) Gather more data

While coding agent leave valuable traces, they offer only a partial view of the activity of coding agents. Moreover, due to the Peril 3 of diversity, some of the partial information is agent-specific. The missing information can come from several dimensions.

First, different agents or developer workflows will share different types of information. For instance, while several agents sign co-authored commits, others do not. Some agents may not support committing on behalf of the developers and, even when they do, developers may not be willing to adopt this workflow, and may instead commit manually. Finally, even if a developer does use an agent that commits on their behalf, they may not be willing to advertise it. Some agent configuration files (e.g. Claude Code) will co-sign commits by default, but have some settings to disable this feature; alternatively, we have seen examples of guidance files explicitly telling the agent *not* to sign commits under any circumstances. As a consequence, our study of coding agent adoption, finds that more than 40% of projects that have markers of usage of coding agents do not have any markers at the commit level [48], making it difficult to ascertain to which degree these projects have adopted coding agents (they may, after all, simply not be using the agent).

The converse is true: due to differences in terms of workflow, agents may advertise their presence via commits and pull requests, but not via files. Increasingly, other agents may use standard files such as `AGENTS.md`, making the detection of specific agents more difficult. Finally, even if agent rules and guidance files are useful knowledge that should be shared in the repository, a significant portion of are not. Our coding agent adoption study estimates that 20% of projects that have agent guidance or configuration files choose to exclude all of them from commits via the `.gitignore` [48]. Harder to quantify, developers may store their agent configuration in separated repositories (e.g. a `dotfiles` repository).

Even when commits, pull requests and files are present, they only represent the final output of the coding agent. In general, what was the input (the initial prompt) is not present, except when the agent is explicitly assigned an issue (in which case the issue description is the prompt). Depending on their guidance, agents may be asked to write down plans before implementing changes, which is a step removed from the initial prompt, but is certainly closer.

Also missing in most cases is the amount of involvement of developers in supervising the agent. Since only the end result is shown in commits, it is hard to know if the developers closely supervised the agent (or edited the code) or not. Here, pull requests may be more informative, particularly if they contain developer-agent interactions via comments, showing part of the interactions.

Finally, several agents (e.g. Codex) do record very detailed traces of the agent's execution, including prompts, interactions, and tool calls (much like is available on SWE-bench [11]). In the case of Codex, links to these traces is even provided as part of the pull request. However, these links require authentication on OpenAI's servers, and are not publicly accessible.

Mitigation for Peril 1: Additional heuristics.

Additional heuristics can remediate parts of the Peril 1 of Partial observability. The first is to identify repositories where it is very likely to apply, by identifying know patterns that reduce observability. Examples include the presence of `.gitignore` files excluding agent configuration files, checking for known settings that affect the agent's harness (e.g. disabling commit signing), or guidance with similar goals (although this is more challenging to detect).

A second way is to detect artifacts that are likely to be made at least partially by coding agents. Several heuristics can be devised for this, exploiting for instance LLM tendencies to thoroughly document their work in commit messages or pull requests (often better than developers), or their usage of Emojis. Other heuristics might use process metrics (e.g. looking for changes in the rate of commits or pull requests [35]). Work to do this already exist for text snippets [39], or for code [54], but repository-level heuristics and information might prove useful signals. Needless to say, these heuristics should be evaluated carefully. In this regard, the availability of data with clear markers for usage of coding agents might be useful to define training, validation, or test sets to evaluate simple or elaborate heuristics to detect unlabeled coding agent activity. On the other hand, changes to model capabilities over time may change the nature of the problem, making detection an arms race between improving models and detectors.

Mitigation for Peril 1: Gather more data

Another mitigation is to collect more complete data. Similarly to how developer interaction data can be collected via telemetry [38], some agents provide functionality to record more information, via explicit telemetry [15] or user-defined hooks [13, 18]. This is however a significant undertaking.

Peril 2: Agent multiplicity

There are many agents, leaving traces in very different ways.

Mitigations: (1) Unequal adoption, (2) Knowledge sharing.

As can be seen in Table 1, there are large number of agents, with a wide range of different heuristics. This number is still growing: while the first were released in 2024, and there were a dozen in early 2025, the last few months have seen the total reach three dozen agents. The large number of agents make it more challenging to keep track of them, and to search for effective heuristics in all cases. Using only some heuristics will miss agent uses, which may be problematic for a study that values exhaustiveness.

Mitigation for Perils 2 and 3: Unequal adoption

While many different agents are available, most of the adoption is focused on a few agents. Currently, Claude, Codex, Cursor and Copilot are the most popular and captures more than 80% of the adoption, whereas agents like OpenHands or Jules see comparatively little usage. This can be seen in Table 1 by looking at the counts of each heuristic; our study of agent adoption points towards a similar conclusion. Thus, a possible strategy for studies is to focus on the subset of agents that captures the majority of the usage. This reduces the cost of carrying out the study since less heuristics have to be developed and validated, at the expense of increased false negatives. That said, studies of smaller agents are still useful, to study their specific characteristics and the impact of these characteristics on the way they work.

Previous work on JavaScript front-ends frameworks [21] or version control [43] show that tool adoption is highly dependent on popularity of the framework, creating a phenomenon of "the rich get richer". While this is still quite early to observe this effect, and this might change in the future, we see the emergence of a winner takes all phenomenon, in which a very small subset of coding agents captures most use, leading to less agents in the long term.

Peril 3: Diversity

Each agent works differently, and this may be necessary to consider in any data analysis.

Mitigations: (1) Unequal adoption, (2) Knowledge sharing.

Each agent has a different harness, exposing different capabilities. For instance, some agents such as Codex submit their changes via pull requests; others like Claude tend to use the Co-authored-by commit trailer, or other conventions (e.g. full authorship).

Beyond the need to develop specific heuristics, these changes affect the workflow developers follow while using the tools, which, depending on the study, may be important to take into account. Examples of this include the way that agents work with pull requests: PRArena [4] is a website that tracks, for a set of agents, the merge rate of their pull requests, using heuristics similar to the ones presented in this paper. Some agents, such as Codex, favor private iteration on PRs, while others, such as Copilot, will submit a PR and favor public iteration with developer via GitHub PR comments. This is clearly visible in the difference between the PR merge rates: among all PRs, Codex has a much higher merge rate (86.6% on 20/10/2025) than Copilot (63.2%); however, if one excludes "draft" PRs, Copilot's merge rate jumps to 93.1%. Even more importantly, PRArena has no information on Claude Code, despite it being one of the most prominent agents (our study finds it has the highest adoption level): this is because Claude Code's workflow did not include PRs until very recently (september 29, 2025). The problem is broader: while some agents can author PRs or commits, other agents do not (also see the Peril 1 of Partial observability).

Another dimension of variability is how agents use guidance files. Earlier agents such as Cursor put more emphasis on simpler rules (possibly reflecting the limited capacities of earlier models). Most agents now use free-form guidance files (e.g. AGENTS.md, CLAUDE.md), where the user (or the agent) can summarize information deemed important. Some agents incentivize their users to adopt sophisticated guidance systems (e.g. Cline's memory bank pattern, or Kiro's steering). As these instructions vary considerably, we expect them to influence the behavior of the agents.

This is mitigated by the previous two mitigations: only a few agents are widely used and this will keep going with less agents in the long term making it easier for data analysis.

Peril 4: High velocity

Practices evolves very quickly, with far reaching implications.

Mitigations: (1) Revisit studies, (2) Standardization, (3) Knowledge sharing.

The adoption of agents, their functionality, and their usage changes rapidly, for many reasons. First, the capacities of the underlying LLMs is still evolving, and this influence the agents capabilities. Kwa et al. estimate that the duration of tasks that LLMs can solve autonomously doubles every 7 months [32, 33]. This impacts the tasks that are delegated to coding agents over time.

This rate of change is also visible in the Peril 2 of Agent multiplicity: many agents exist; new ones are regularly introduced; and existing ones change, sometimes impacting heuristics. For instance, Cursor transitioned from storing rules in a single `.cursorrules` file to a `.cursor` directory with more general guidance. Another example is the `AGENTS.md` file: originally, this was the default file used for guidance by the Codex agent. There is now a standardization effort around this format, with a growing number of coding agents supporting it. As a consequence, this heuristic is no longer a marker for the Codex agent, but rather a general marker for coding agent usage in a given repository. In addition, since guidance files are text files, it is easy for an agent to use the guidance of other agents: for instance, Zed's agent looks for files used by other popular agents and incorporates it as its own guidance [28].

As a consequence, the detection of agent traces needs to be regularly updated to keep up with the fast pace of releases. Combined with the perils of multiplicity and diversity, this is challenging, which is why we introduce heuristic repository (see Section 6). If the trend of standardization continues, it is possible that it will become both *easier* to detect agent use in general, but *more difficult* to detect the usage of individual agents; this will make it more challenging to take into account the diversity of coding agents.

Another manifestation of the Peril 4 of velocity is the importance of data freshness. While datasets are very important in MSR research, changes to coding agents mean that datasets of their activity will age more rapidly than traditional MSR datasets, particularly if the capacities of coding agents continue to evolve rapidly.

Mitigation for Peril 4: Revisit studies

Given the changes to the capacities of LLMs and their accompanying changes to the capabilities of agents, and the use that is made of these agents, we think that some studies may benefit from being revisited regularly in order to check how their conclusions evolve with time. For instance, a study on the quality of code generated by agents may be revisited if one suspects the evolving capacities may have resulted in higher code quality. Automation helps; studies that use manual annotation will be more difficult to revisit, although partial automation with LLMs can help [2].

Mitigation for Peril 4: Standardization

Initiatives like the Agentic AI Foundation [1] host standards such as the website AGENTS.md or the MCP. Some projects adopt guidelines to recognize and standardize contributions by coding agents. Perhaps more standards will emerge, simplifying studies.

5.3 Beyond agent detection

Promise 6: Model upgrades

Coding agent traces enable the study of the impact of new model release on the use of coding agents.

As the capabilities of the underlying models drive the capabilities of agents and their usage, studies of their actual impact would be very useful. As models have clear release dates, and some agents have large amounts of data, this becomes feasible, e.g. by comparing commits done in time periods in which a newer version of a model was released, with an older time period.

Peril 5: Multiple Models

Some coding agents support multiple models, so determining which model is used is not trivial, or perhaps impossible.

Several agents provide multiple models for their users, yet the traces left do not specify which model (or model version) was used. For instance, Copilot can be used with GPT-5, Sonnet and many other, yet there is no indication of the model used. This limits the ability to study the impact of a specific model release. Coding agents specific to a model vendor are much more likely to use a model from that vendor, however, and some coding agents (such as Aider) do specify which model was used to author a commit. In other cases, the agent's configuration files may provide some clues, as the models to use can be specified there.

Peril 6: Scientific reproducibility

Coding agents are non-deterministic and often based on closed, proprietary LLMs, undermining scientific reproducibility.

Mitigations: (1) Open coding agents, (2) Replication packages.

LLMs are non-deterministic [8]: they can provide different results starting from identical inputs (project status, prompt, etc.), and even with a fixed random seed (with batch inference [24]). Furthermore, most coding agents are currently based on LLMs that are closed and/or proprietary [37], as a whole or in part [61], and accessed via remote APIs. In other cases the models are available, but only provide the final model weights. In rare cases (e.g., OLMo [42]), all LLM parts—training datasets, training pipeline, model weights, inference code—are available and released under traditional open source licenses [37, 61]. The agent themselves, implemented in software, are available under licenses with varying degrees of openness, while the inference stack is most often opaque and can affect performance [55]. The inference stack may include "model routers" that allow to switch the underlying model, based on the query.

These factors make it either impossible or very hard to replicate the activities of coding agents, potentially undermining the scientific reproducibility of empirical experiments that analyze them.

MSR research will focus on existing traces, which overwhelmingly come from "closed" systems. For studies that focus on traces only, this might not be as problematic, but specific study designs may face more limitations. Studies that require running coding agents (e.g. a study of agent success/failure factors that wishes to establish a causality link) or LLMs (e.g., to automate manual annotations) will face reproducibility issues. Closed models or inference stacks make it impossible to archive the models or even precisely

identify the used versions, as they can be silently altered by the model operators, compounding the Peril of Multiple Models. API models access can also be deprecated and suspended, such as OpenAI's original Codex model. Likewise, research interested in the impact of training data will not be possible with closed systems.

Mitigation for Peril 6: Open coding agents

Studies can trade off data volume for increased reproducibility by focusing on the most open coding agents. Using established scales such as the Modeling Openness Framework [61] or the European Open Source AI Index [37] will help. Note that, even with fully-open agents, the problem of LLM non-determinism remains.

Mitigation for Peril 6: Thorough replication packages

Empirical studies conducted using coding agents should report agent information and/or artifacts in replication packages. For open agents, the package should include all relevant artifacts (datasets, models, agent code, etc.). For closed agents, the package should document as best as possible their versions and deployment details.

Peril 7: Costs shape usage

Coding agents are expensive, and this influences their usage.

Mitigation: (1) Strategic Sampling

LLM inference is compute intensive, particularly for coding agents. For a single inference turn, processing the LLM's prompt has a quadratic complexity with respect to the prompt size, while generating tokens has a linear complexity (with respect to the prompt and the previously generated tokens) due thanks to the key-value cache. This makes inference on long prompts much more expensive than shorter prompts. Coding agents tend to work on tasks requiring large prompts, including guidance, relevant context, tool results, built over multiple turns (e.g. OpenHands routinely consumes more than a million tokens on SWE-bench tasks [11]).

As a consequence, using coding agent is expensive. The 2025 Stack Overflow survey [50] shows that the majority of interviewees agree that "the cost of using AI agent platforms is a problem". Different pricing models exist: flat-rate (with usage limits) and usage-based. This introduces a discrepancy between users depending on their resources: open-source contributors might have to buy their own subscription, opting for lower tiers than developers working for companies that pay for their access. A few testimonials advocate that budgets exceeding 1,000$ per developer per month can be justified [46]. This, in turn, can lead to radically different usages between open-source repositories and industrial repositories, where open-source repositories and contributors, vastly more prevalent on GitHub, may feature lower usage of coding agents. The effect of this is broader, with developers from different countries being more or less price-sensitive. In addition price-sensitive users may also use multiple accounts with free tiers to satisfy their usage demands. Conclusions on the usage of coding agents should be made with this context in mind. This also implies that the distribution of repositories using coding agents may be radically different than the distribution of open source repositories.

Furthermore, as mentioned in the Peril of High Velocity, the practice changes very quickly, including pricing: coding agents change their prices or usage limits over time (e.g. Cursor introducing rate limits [56]), model inference cost change, which will affect usage.

Mitigation for Peril 7: Strategic Sampling

While there are some usage differences between users that have higher resources than others (especially those coming from industry), looking for open-source repositories in which companies contribute significantly may be helpful in finding users with usages more representative of those found in industry.

Peril 8: AI coding slop

Coding agents may produce more and larger development artifacts, of lower quality than those produced by humans. This "AI coding slop" will make the work of researchers in empirical software engineering harder.

Traditionally, software development artifacts have a high signal-to-noise ratio and require significant human effort to produce. For example, it can take a full day for a developer to fix a thorny bug; at the end of the day the observable artifact might be a one-line patch, recorded in Git with an accompanying detailed commit message. Net result: significant human effort, small artifact of high quality.

"AI slop" [25] is a derogatory term that refers to AI-generated digital content characterized by its abundance, low quality, and the perceived little effort put into generating it (causing its low quality). Coding agents risk polluting the primary sources that researchers in empirical software engineering study (VCS repositories, collaborative coding platforms, etc.) with *"AI coding slop"*: abundant, verbose, and poor-quality development artifacts (commits, pull/merge requests, comments, etc.). Examples include AI-generated security reports [51], although discerning AI use is much more useful [52].

Although it is too soon to understand the magnitude of this problem, if AI coding slop is coming, it will materialize at different levels. First, the number of artifacts researchers will have to analyze will increase significantly. For example, in our experience it is already very common to observe very long interaction sequences among developers and agents in pull request discussions, publicly displaying the feedback loop needed to converge to an acceptable result. Previously, the feedback loop was in the head of (or conversations among) developers and did not pollute interactions that researchers study. Second, the size of individual artifacts will also increase, due to a natural tendency of LLMs to be more verbose than humans.

AI slop spreading to software development artifacts, will require methodological adaptations. Studies only interested in analyzing human interactions will now need to filter out agent-human and agent-agent interactions. This is not an entirely new requirement—MSR researchers already have to deal with bot detection—but will require new techniques, tools, and heightened methodological scrutiny.

6 Heuristics and data repository

Mitigation for Perils 2, 3, and 4: Knowledge sharing

The heuristics we have presented are necessarily numerous, varied and change prone due to the Perils of Multiplicity, Diversity, and Velocity. This makes keeping them up to date challenging. The best mitigation is to distribute the effort, and share the knowledge. To this extent, we started a repository to easily share this knowledge with the MSR community, and update it over time. The repository [47] is in its early stages, but it contains:

- A brief description of each coding agent (to be expanded over time), with links for further information;
- A list of heuristics to detect various markers of its presence in software repositories with their period of validity.
- For heuristics that allow it, links to GitHub queries for interactive browsing and python scripts to gather sample data;
- A list of repositories (ca. 10,000) featuring agent adoption as of October 2025, allowing targeted data collection.

We call on the MSR community to contribute to this effort, by expanding the heuristic repository with additional heuristics as the practices changes, and by giving feedback on the effectiveness of the existing heuristics, in order to continuously improve them. While a handful of researchers will have difficulties keeping up with the velocity of coding agents, we hope that the much larger MSR research community, will do so in a much more scalable way.

7 Discussion

Limitations. This paper is focused on the promises and perils of mining coding agent usage, rather than the promises and perils of coding agent themselves. Issues such as their environmental impact, impact on the workforce, or intellectual property, while extremely important, are thus not covered in this paper. Previous work on promises and perils of mining Git [9] and GitHub [30] also apply.

Our promises, perils, and heuristics are based on our personal experience conducting MSR studies of coding agents [26, 48]. While we strive for exhaustivity, we can not guarantee it. In fact, the Peril 4 of velocity implies that parts of the knowledge of this work (at least the heuristics) will need to be updated over time. The heuristic and data repository is a key mitigation to ensure this work remains current; we call on the community to contribute to this effort.

On the accuracy of heuristics. By definition, heuristics are noisy. We take this in consideration when evaluating individual heuristics. For instance, this led us to exclude a heuristic for Aider, a guidance file named `CONVENTIONS.md`, as most such files contains content aimed at developers, rather than coding agents. However, the large quantity of heuristics (more than 100 in the repository so far) makes a precise evaluation of each heuristic impractical at the scale of a single research effort. Once again, we see the involvement of the community in the heuristic and data repository as a key mechanism to improve this aspect. In the meantime, we can already provide some general lessons on the accuracy of heuristics:

- The Peril 1 of partial observability indicates that *recall* is a general concern: a significant portion of agentic use is still difficult to detect. The way each tool works influences this.
- Heuristics can have precision issues, and this varies across categories and per heuristic.
- The precision of author or co-author-based heuristics depends on the patterns. Heuristics relying on specific author emails, e.g. `<noreply@anthropic.com>` have a high precision; Git author names (e.g. `claude`) may have lower precision, while references to specific GitHub user accounts is unambiguous. When possible, relying on emails (or GitHub accounts) is thus preferable.
- We expect the most variability in precision for files, depending on how generic or specific a pattern is: a highly specific `.kiro/` directory is a precise; a generic `CONVENTIONS.md` is not.
- Similarly, heuristics for branches and labels may be more or less specific depending on the pattern; e.g., references to `cursor` may refer to the agent, or to the general concept of a cursor (searching for a `cursor/` branch prefix is more specific).

Depending on accuracy a study seeks, some heuristics may use additional filtering (such as date ranges) or aggregation to refine the data. In addition, several heuristics might identify the same artifact. We call on the MSR community to document these cases.

Implications. The principal implication of this work is that the existence of coding agent traces opens up a vast field of potential studies of this phenomenon and its impact on Software Engineering. We provided some examples of such studies, but expect many more.

The implications in terms of potential studies vary with the degree of effective adoption of coding agents. The examples we provide (e.g. studies of effects on productivity, or of the factors influencing the success of agents) assume a "business as usual" scenario, were developers use coding agents to assist them in the same tasks that they work on today. However, some have much more ambitious visions for coding agents. If the observed trend of doubling the autonomy of agents on tasks every seven months is sustained for a few years [33], the capabilities of coding agents will improve significantly, changing the tasks they are given. At the same time, techniques and tools to handle multiple agents in parallel emerge [3, 60], which would maximize this effect. If these trends materialize, then the type, the amount, and the scope of tasks delegated to agents may change rapidly, in which case software engineering practices themselves will be affected.

We stress that the impact for MSR studies is potentially broader than this: the already high level of adoption shows that agents are already used in practice extensively. As mentioned in the Peril of AI Slop, if, as we expect it to, this trend continues, research that aims at studying *human* contributions will need to take this into account, and *exclude* contributions from agents, much like bot activity should be excluded. Our heuristics are a first step for this.

8 Conclusion

In the span of a few months, coding agents have transitioned from an academic endeavor to products that are used daily by developers. While LLM-based completion was used to streamline coding activities and was difficult to observe using MSR techniques, coding agents can address more tasks, and leave explicit traces in software repositories, finally enabling the study of AI-assisted coding via MSR techniques at scale. In this work, we presented heuristics to detect coding agent activities in files, commits, issues, and pull requests. More importantly, after having used these heuristics in MSR studies, we have distilled the promises and perils of mining coding agent activities. The principal perils being the multiplicity of diverse, fast changing agents, we invite the MSR community to contribute to our repository of shared knowledge and heuristics. The main promise being the very large potential for studies of this phenomenon and its impact on SE practices, we hope the community will join us in exploring it.

Acknowledgments

This study received financial support from the French State (Investments for the Future programme, IdEx université de Bordeaux).

References

[1] [n. d.]. Agentic AI Foundation (AAIF). https://aaif.io Accessed: 2026-01-26.

[2] Toufique Ahmed, Premkumar Devanbu, Christoph Treude, and Michael Pradel. 2025. Can LLMs Replace Manual Annotation of Software Engineering Artifacts?. In *22nd IEEE/ACM International Conference on Mining Software Repositories (MSR '25)*. 526–538. doi:10.1109/MSR66628.2025.00086 Accessed 2025-10-23.

[3] DevSwarm AI. 2025. DevSwarm — Code in Parallel, Build Unstoppably. https://devswarm.ai/. Accessed 2025-10-21.

[4] PR Arena. 2025. PR Arena – AI Coding Agent Leaderboard. https://prarena.ai. Accessed: 2025-10-20.

[5] Owura Asare, Meiyappan Nagappan, and N Asokan. 2023. A User-centered Security Evaluation of Copilot. *arXiv preprint arXiv:2308.06587* (2023).

[6] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111.

[7] Joel Becker, Nate Rush, Elizabeth Barnes, and David Rein. 2025. Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity. *arXiv preprint arXiv:2507.09089* (2025).

[8] Adhithya Bhaskar and Victoria Stodden. 2024. Reproscreener: Leveraging LLMs for Assessing Computational Reproducibility of Machine Learning Pipelines. In *Proceedings of the 2nd ACM Conference on Reproducibility and Replicability, ACM REP 2024, Rennes, France, June 18-20, 2024*. ACM. doi:10.1145/3641525.3663629

[9] Christian Bird, Peter C Rigby, Earl T Barr, David J Hamilton, Daniel M German, and Prem Devanbu. 2009. The promises and perils of mining git. In *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 1–10.

[10] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 694–694.

[11] Islem Bouzenia and Michael Pradel. 2025. Understanding Software Engineering Agents: A Study of Thought-Action-Result Trajectories. *arXiv preprint arXiv:2506.18824* (2025).

[12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[13] Anthropic / Claude Code. 2025. Claude Code Hooks Guide. https://docs.claude.com/en/docs/claude-code/hooks-guide. Accessed 2025-10-21.

[14] Anthropic / Claude Code. 2025. DevContainer — Claude Code Documentation. https://docs.claude.com/en/docs/claude-code/devcontainer. Accessed 2025-10-21.

[15] OpenAI / Codex. 2025. Config — Observability and Telemetry. https://github.com/openai/codex/blob/main/docs/config.md#observability-and-telemetry. Accessed 2025-10-21.

[16] Dagger / container use. 2025. container-use — Development environments for coding agents. https://github.com/dagger/container-use. Accessed 2025-10-21.

[17] Wikipedia contributors. 2024. Vibe coding. https://en.wikipedia.org/wiki/Vibe_coding. Accessed on 2025-10-20.

[18] Cursor. 2025. Agent Hooks — Cursor Documentation. https://cursor.com/docs/agent/hooks. Accessed 2025-10-21.

[19] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. In *International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 31–53.

[20] Ahmed Fawz, Amjed Tahir, and Kelly Blincoe. 2025. Vibe Coding in Practice: Motivations, Challenges, and a Future Outlook-a Grey Literature Review. *arXiv preprint arXiv:2510.00328* (2025).

[21] Fabio Ferreira, Hudson Silva Borges, and Marco Tulio Valente. 2022. On the (un-) adoption of JavaScript front-end frameworks. *Software: Practice and Experience* 52, 4 (2022), 947–966.

[22] William Harding. 2025. AI Copilot Code Quality: Evaluating 2024's Increased Defect Rate via Code Quality Metrics. https://www.gitclear.com/ai_assistant_code_quality_2025_research Accessed on October 13th, 2025.

[23] William Harding and Matthew Kloster. 2024. Coding on Copilot: 2023 Data Suggests Downward Pressure on Code Quality. https://www.gitclear.com/coding_on_copilot_data_shows_ais_downward_pressure_on_code_quality Accessed on 03 24, 2024.

[24] Horace He and Thinking Machines Lab. 2025. Defeating Nondeterminism in LLM Inference. *Thinking Machines Lab: Connectionism* (Sept. 2025). https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/ Accessed 2025-10-21.

[25] Alex Hern and Dan Milmo. 2024. Spam, junk… slop? The latest wave of AI behind the 'zombie internet.'. https://www.theguardian.com/technology/article/2024/may/19/spam-junk-slop-the-latest-wave-of-ai-behind-the-zombie-internet. *The Guardian* (2024). Accessed 2025-10-15.

[26] Andre Hora and Romain Robbes. 2026. Are Coding Agents Generating Over-Mocked Tests? An Empirical Study. In *Proceedings of the International Conference on Mining Software Repositories (MSR 2026)*. In press.

[27] Saki Imai. 2022. Is github copilot a substitute for human pair-programming? an empirical study. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 319–321.

[28] Zed Industries. 2025. Using Rules — Zed AI Documentation. https://zed.dev/docs/ai/rules. Accessed: 2025-10-20.

[29] Maliheh Izadi, Jonathan Katzy, Tim Van Dam, Marc Otten, Razvan Mihai Popescu, and Arie Van Deursen. 2024. Language models for code completion: A practical evaluation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[30] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2014. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*. 92–101.

[31] Aayush Kumar, Yasharth Bajpai, Sumit Gulwani, Gustavo Soares, and Emerson Murphy-Hill. 2025. Why AI Agents Still Need You: Findings from Developer-Agent Collaborations in the Wild. *arXiv e-prints* (2025), arXiv–2506.

[32] Thomas Kwa, Ben West, Joel Becker, Amy Deng, Katharyn Garcia, Max Hasin, Sami Jawhar, Megan Kinniment, Nate Rush, Sydney Von Arx, et al. 2025. Measuring ai ability to complete long tasks. *arXiv preprint arXiv:2503.14499* (2025).

[33] Thomas Kwa, Ben West, Joel Becker, Amy Deng, Katharyn Garcia, Max Hasin, Sami Jawhar, Megan Kinniment, Nate Rush, Sydney Von Arx, et al. 2025. Measuring AI Ability to Complete Long Tasks (up to date web site). https://metr.org/blog/2025-03-19-measuring-ai-ability-to-complete-long-tasks/. Accessed: 2025-10-20.

[34] Hao Li, Haoxiang Zhang, and Ahmed E. Hassan. 2025. AIDev: Studying AI Coding Agents on GitHub. https://doi.org/10.5281/zenodo.16919051. Accessed 2025-10-21.

[35] Hao Li, Haoxiang Zhang, and Ahmed E Hassan. 2025. The Rise of AI Teammates in Software Engineering (SE) 3.0: How Autonomous Coding Agents Are Reshaping Software Engineering. *arXiv preprint arXiv:2507.15003* (2025).

[36] Jenny T Liang, Chenyang Yang, and Brad A Myers. 2024. A large-scale survey on the usability of ai programming assistants: Successes and challenges. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[37] Andreas Liesenfeld and Mark Dingemanse. 2024. Rethinking open source generative AI: open washing and the EU AI Act. In *The 2024 ACM Conference on Fairness, Accountability, and Transparency, FAccT 2024, Rio de Janeiro, Brazil, June 3-6, 2024*. ACM, 1774–1787. doi:10.1145/3630106.3659005

[38] Walid Maalej, Thomas Fritz, and Romain Robbes. 2013. Collecting and processing interaction data for recommendation systems. In *Recommendation Systems in Software Engineering*. Springer, 173–197.

[39] Eric Mitchell, Yoonho Lee, Alexander Khazatsky, Christopher D Manning, and Chelsea Finn. 2023. Detectgpt: Zero-shot machine-generated text detection using probability curvature. In *International conference on machine learning*. PMLR, 24950–24962.

[40] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2022. Reading between the lines: Modeling user behavior and costs in AI-assisted programming. *arXiv preprint arXiv:2210.14306* (2022).

[41] Vijayaraghavan Murali, Chandra Maddila, Imad Ahmad, Michael Bolin, Daniel Cheng, Negar Ghorbani, Renuka Fernandez, and Nachiappan Nagappan. 2023. CodeCompose: A large-scale industrial deployment of AI-assisted code authoring. *arXiv preprint arXiv:2305.12050* (2023).

[42] Team OLMo, Pete Walsh, Luca Soldaini, Dirk Groeneveld, Kyle Lo, Shane Arora, Akshita Bhagia, Yuling Gu, Shengyi Huang, Matt Jordan, Nathan Lambert, Dustin Schwenk, Oyvind Tafjord, Taira Anderson, David Atkinson, Faeze Brahman, Christopher Clark, Pradeep Dasigi, Nouha Dziri, Michal Guerquin, Hamish Ivison, Pang Wei Koh, Jiacheng Liu, Saumya Malik, William Merrill, Lester James V. Miranda, Jacob Morrison, Tyler Murray, Crystal Nam, Valentina Pyatkin, Aman Rangapur, Michael Schmitz, Sam Skjonsberg, David Wadden, Christopher Wilhelm, Michael Wilson, Luke Zettlemoyer, Ali Farhadi, Noah A. Smith, and Hannaneh Hajishirzi. 2025. 2 OLMo 2 Furious. *CoRR* abs/2501.00656 (2025). arXiv:2501.00656 doi:10.48550/ARXIV.2501.00656

[43] Tihomir Orehovački, Darko Etinger, and Snježana Babić. 2020. Modelling the adoption of the version control system: an empirical study. In *International Conference on Human Systems Engineering and Design: Future Trends and Applications*. Springer, 45–50.

[44] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590* (2023).

[45] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2023. Do users write more insecure code with AI assistants?. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2785–2799.

[46] Vincent Quigley. 2025. First attempt will be 95% garbage: A staff engineer's 6-week journey with Claude Code. *Sanity Blog* (Sept. 2025). https://www.sanity.io/blog/first-attempt-will-be-95-garbage Accessed 2025-10-21.

[47] Romain Robbes, Théo Matricon, Thomas Degueule, Andre Hora, and Stefano Zacchiroli. 2025. Agent Mining Repository. https://github.com/labri-progress/agent-mining. Accessed 2025-10-23.

[48] Romain Robbes, Théo Matricon, Thomas Degueule, Andre Hora, and Stefano Zacchiroli. 2026. Agentic Much? Adoption of Coding Agents on GitHub. *arXiv preprint* (2026). arXiv:2601.18341 [cs.SE] doi:10.48550/arXiv.2601.18341

[49] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. 2023. Lost at c: A user study on the security implications of large language model code assistants. In *32nd USENIX Security Symposium (USENIX Security 23)*. 2205–2222.

[50] Stack Overflow. 2025. AI | 2025 Stack Overflow Developer Survey: Challenges with AI Agents. https://survey.stackoverflow.co/2025/ai#3-challenges-with-ai-agents. Accessed: 2025-10-16.

[51] Daniel Stenberg. 2025. Death by a thousand slops. *daniel.haxx.se Blog* (14 July 2025). https://daniel.haxx.se/blog/2025/07/14/death-by-a-thousand-slops/ Accessed 2025-10-23.

[52] Daniel Stenberg. 2025. A new breed of analyzers. *daniel.haxx.se Blog* (10 Oct. 2025). https://daniel.haxx.se/blog/2025/10/10/a-new-breed-of-analyzers/ Accessed 2025-10-23.

[53] Klaas-Jan Stol and Brian Fitzgerald. 2018. The ABC of software engineering research. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 27, 3 (2018), 1–51.

[54] Hyunjae Suh, Mahan Tafreshipour, Jiawei Li, Adithya Bhattiprolu, and Iftekhar Ahmed. 2025. An Empirical Study on Automatically Detecting AI-Generated Source Code: How Far are We?. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 859–871.

[55] Anthropic Engineering Team. 2025. A postmortem of three recent issues. *Anthropic Engineering Blog* (17 Sept. 2025). https://www.anthropic.com/engineering/a-postmortem-of-three-recent-issues Accessed 2025-10-22.

[56] Michael Truell. 2025. Clarifying our pricing. *Cursor Blog* (4 July 2025). https://cursor.com/blog/june-2025-pricing Accessed 2025-10-21.

[57] Rosalia Tufano, Antonio Mastropaolo, Federica Pepe, Ozren Dabić, Massimiliano Di Penta, and Gabriele Bavota. 2024. Unveiling ChatGPT's Usage in Open Source Projects: A Mining-based Study. *arXiv preprint arXiv:2402.16480* (2024).

[58] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*. 1–7.

[59] Ruotong Wang, Ruijia Cheng, Denae Ford, and Thomas Zimmermann. 2023. Investigating and designing for trust in ai-powered code generation tools. *arXiv preprint arXiv:2305.11248* (2023).

[60] Warp. 2025. The Agentic Development Environment — Agents. https://www.warp.dev/agents. Accessed 2025-10-21.

[61] Matt White, Ibrahim Haddad, Cailean Osborne, Xiao-Yang Liu, Ahmed Abdelmonsef, and Sachin Varghese. 2024. The Model Openness Framework: Promoting Completeness and Openness for Reproducibility, Transparency and Usability in AI. *CoRR* abs/2403.13784 (2024). arXiv:2403.13784 doi:10.48550/ARXIV.2403.13784

[62] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489* (2024).

[63] Tao Xiao, Youmei Fan, Fabio Calefato, Christoph Treude, Raula Gaikovina Kula, Hideaki Hata, and Sebastian Baltes. 2025. Self-Admitted GenAI Usage in Open-Source Software. *CoRR* abs/2507.10422 (2025). arXiv:2507.10422 doi:10.48550/ARXIV.2507.10422

[64] Tao Xiao, Christoph Treude, Hideaki Hata, and Kenichi Matsumoto. 2024. Devgpt: Studying developer-chatgpt conversations. In *Proceedings of the 21st international conference on mining software repositories*. 227–230.

[65] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1592–1604.

[66] Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 21–29.