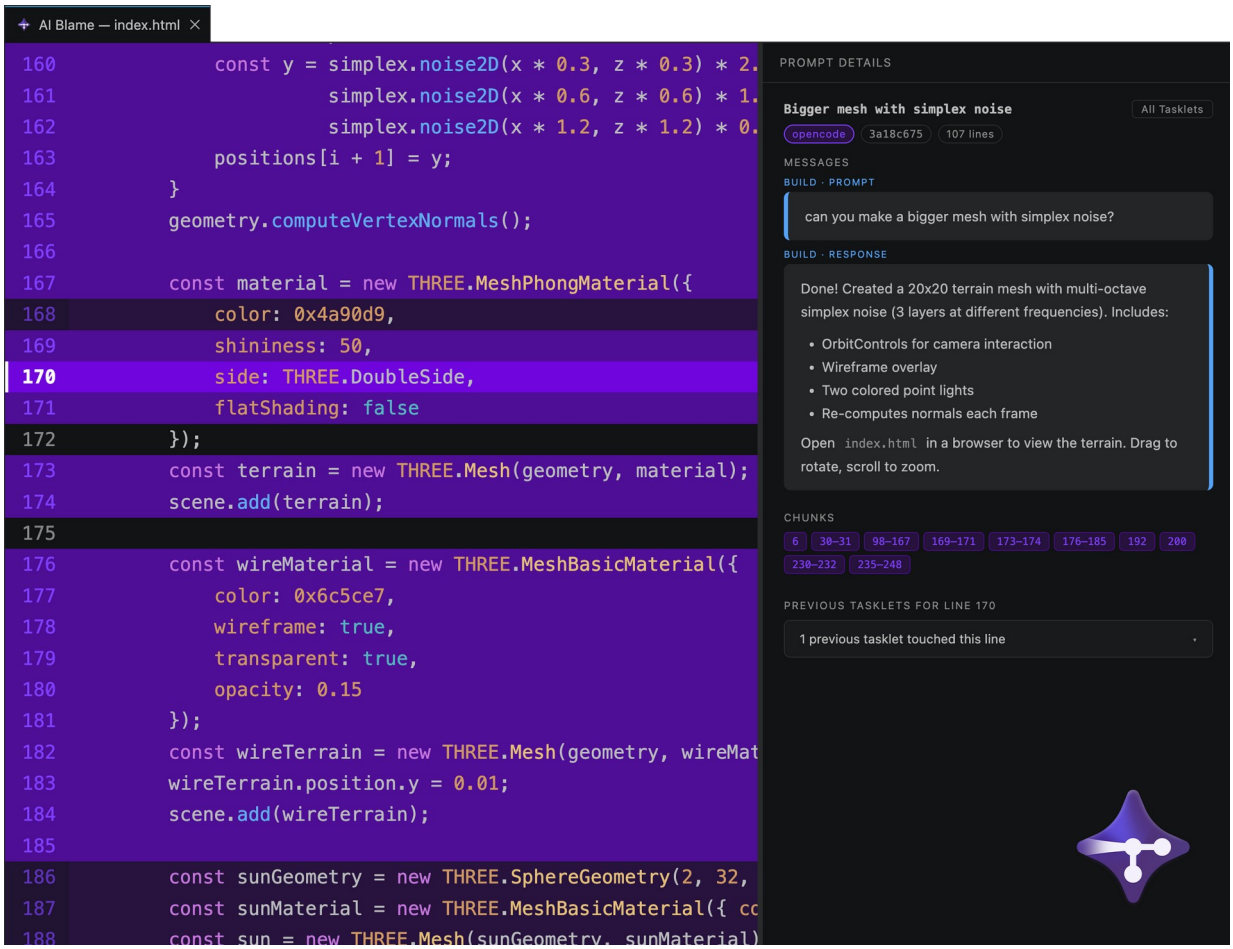
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Tracybot

Tracing LLM-Generated Code to Prompts

Bachelor of Science Thesis in Software Engineering and Management

ALIAKSEI KHVAL
RĂZVAN ALBU
MAXINE ORLOVA
DANIS MUSIC
FILIPE ARAÚJO VERAS ROSA

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026

CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Tracybot: Tracing LLM-Generated Code to Prompts
A Design Science Study on Improving Transparency and Accountability in AI-Assisted Software Engineering

Supervisor: RANIM KHOJAH
Examiner: DANIEL STRÜBER

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

[Cover: a screenshot of the Tracybot VS Code extension, showing a code example with highlighted AI attribution and traced prompts.]

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026

Tracybot: Tracing LLM-Generated Code to Prompts

Authors
Aliaksei Khval, Răzvan Albu, Maxine Orlova, Danis Music, Filipe Araújo Veras Rosa
Department of Applied IT
University of Gothenburg
Gothenburg, Sweden

Supervisor
Ranim Khojah
Interaction Design and Software Engineering
Chalmers University of Technology
Gothenburg, Sweden

***Abstract*—Large Language Models (LLMs) are now widely used in software engineering, but prompts are rarely treated as traceable artifacts, limiting transparency and accountability. To address this gap, a two-cycle Design Science Research methodology is applied to design, implement and evaluate *Tracybot*, an automated tool that integrates with *Git*, *OpenCode*, and *VS Code* to map LLM-generated code to originating prompts. To evaluate its usability, feedback was gathered via a semi-structured focus group (7 participants) and a user evaluation survey (24 participants). The results show that developers perceived *Tracybot* as useful for supporting code review workflows, particularly for improving AI usage transparency and support for better informed assessment of code correctness. However, challenges remain in managing long-term storage overhead, refining attribution between human-authored and AI-authored code, as well as mitigating organizational and ethical concerns such as workplace micromanagement and developer monitoring.**

I. Introduction

The rapid emergence of high-performance Large Language Models (LLMs) has fundamentally altered the field of software development. What began as experimental assistance has matured into a standard component of the Software Engineering workflow, with professionals increasingly relying on generative artificial intelligence (AI) for code production[1].

Although this adoption has made code generation highly efficient [2], it has also outpaced the development of the necessary ethical and legal frameworks. For example, Kraishan [3] notes that the majority of developers acknowledge the use of AI in some way, but only a few explicitly credit it with code generation. Biases such as social desirability bias lead developers to under-report AI usage, making self-reporting insufficient for accurate attribution [4], [5]. Moreover, there is legal pressure on companies to adopt clear AI crediting by government regulations such as the European Union AI Act [6].

According to a recent study [7], software companies believe that LLM-generated code needs extra verification, with many practitioners recognizing the need for additional review steps tailored specifically to LLM outputs. However, this has not yet been applied in industry. Designing an LLM-specific verification process would require the generated code to be clearly labelled. In addition, when the label includes the prompt, it also helps to understand the intention of the developer who generated the code.

In this project, we aim to bridge this gap by developing *Tracybot* - a tool that integrates with agentic AI workflows and allows for automated mapping between prompts and LLM-generated code artifacts, so that development processes do not have to rely on manual LLM usage reporting.

II. Related Work

Traceability is essential for software quality and is defined by Souali et al. [8] as "the degree to which a relationship can be established between two or more products of the development process". Gotel et al. [9] further define it as "the potential to relate data that is stored within artifacts of some kind, along with the ability to examine this relationship". Its utility includes change impact analysis, coverage analysis, and dependency analysis. Traditionally, traceability is applied to requirements, design specifications and test cases.

In AI-assisted software development, prompts serve a similar role to requirements as they drive the development of the software. However, they are not yet commonly treated as traceable artifacts [3], [4], [5].

Current AI attribution developments focus on categorising code as LLM-generated or human, but struggle with co-authored code due to the mix of human and LLM writing styles [10]. To address this challenge, post-generation approaches exploit unique LLM "fingerprints." Xu et al. [11] showed that LLM-generated code carries identifiable stylistic features. Building on this, Bisztray et al. [12] demonstrated that transformer encoders (via their CodeT5-Authorship architecture) can exploit these stylistic features to attribute fully LLM-generated code to its specific source model with up to 97.56% accuracy. However, because these attribution methods operate only after the code is written, they cannot guarantee perfect attribution accuracy. Furthermore, prompt traceability remains largely unaddressed in existing AI attribution literature. To capture this deeper context, Akgün [13] argues that authorship frameworks must shift toward emphasising concrete contributions and authorial control over prompts and outputs, while Sun et al. [14] emphasise using syntax and semantic features to better measure these developer contributions.

The existing related research, therefore, focuses on (1) conceptual foundations for traceability in software engineering, (2) emerging techniques for detecting LLM-generated and LLM-co-authored code, (3) evolving perspectives on authorship in AI-assisted development. However, there remains a gap in applying the AI attribution practices and their practical benefits. This study builds upon previous related research by treating prompts as traceable artifacts during development rather than focusing solely on the detection of LLM code after it is generated.

III. Research Method

In this study, we adopt a Design Science research methodology with two cycles by following the guidelines defined by Hevner and Chatterjee [15]. This methodology involves creating and updating an artifact to address domain-related research

questions. More specifically, according to Johannesson et al. [16], "Design science in information systems and technology aims to create novel artifacts in the form of models, methods, and systems that support people in developing, using, and maintaining IT solutions." This study's artifact is *Tracybot* - a tool responsible for tracing an LLM prompt(s) to the resulting code. The tool's usability is then evaluated through developer interviews.

Based on the identified research gap, research questions are presented to define the artifact, following the guidelines defined by Knauss [17].

> **RQ1 (Problem Space)**: What requirements are necessary to trace LLM prompts to code?
>
> ***Finding:*** *Six core requirements were identified to accommodate tracing prompts to code, including prompt capture, Plan/Build prompts grouping and associating the aforementioned with code changes. During evaluation, challenges were identified and formalised as 4 additional requirements.*
>
> **RQ2 (Solution)**: What are the necessary components for a tool to trace generated code to prompts?
>
> ***Finding:*** *Three necessary components were identified: an agent integration (OpenCode plugin), a git-based tracing backend, and an editor integration (VSCode extension). The components were designed for ease of future extensibility and straightforward integration of additional tools.*
>
> **RQ3 (Evaluation)**: To what extent is the prompt traceability feature usable?
>
> ***Finding:*** *Developers generally found Tracybot useful and non-intrusive. Challenges remain around attribution logic, storage overhead and tool misuse.*

A. Problem Space

To understand the practical challenges of tracing LLM-generated code, we conducted two initial brainstorming sessions, one of them involving the supervisor. The initial session was driven by the analysis of the related work. The latter session was driven by interview data from a recent study[1] that included feedback from 32 software practitioners about how they perceive LLM-labelled code and how they use the information on the label during code review. In addition, we brainstormed different use cases for prompt traceability. The sessions resulted in six requirements that we later used to inform the design and development of our tool artifact. In addition, four requirements were derived during the first cycle, using the feedback gathered from the focus group and internal testing.

B. Solution Candidates

Based on the elicited requirements and problem space analysis, internal discussions and design sessions were held to explore potential solutions for *Tracybot*. Major decisions were formalised as Architecture Decision Records (ADRs) (see subsection V-B). Records are also available in the project repository at https://github.com/TracyTeam/tracybot/tree/main/docs/adr.

C. Validation

To validate the core concept, manual tests of each component were conducted to confirm the technical feasibility of the design. In addition, internal discussions were held to validate design choices against the requirements. During the implementation cycles, the tool was continuously validated by manually testing new functionality on dedicated test repositories.

D. Development Process

The development process followed an agile and iterative workflow throughout both design cycles. Development activities were managed using GitHub's issue board, where feature-related issues were generally mapped to individual requirements to maintain traceability between requirements and implementation tasks. The complete and continuously updated list of requirements was documented in the project Wiki[2].

There are two design cycles, each concluding with an evaluation. The first cycle aimed to build an initial version of *Tracybot*, focusing on the core functionality of prompt tracing (2-3 weeks). At the end of the first cycle, a focus group was held to collect feedback and identify areas for improvement. During the second cycle, the findings from the focus group were consolidated (2-5 days) to identify work items for improvement of the tool, which were then implemented in the final iteration (2-3 weeks). In conclusion, an additional survey was conducted as a final evaluation of the tool and of the improvements made during the second iteration.

The implementation process adopted a feature-branching strategy[3], where each new feature was developed in a dedicated branch before integration into the main branch. To ensure code quality, all contributions were subject to a review process that required at least one approval from a team member who had not contributed to the corresponding feature branch prior to merging. Documentation-related changes were exempted from this requirement.

In addition, commit messages followed the Conventional Commits[4] specification, which provides a standardized structure for commit history by categorising changes according to their purpose (e.g., `feat`, `fix`, or `docs`).

The team conducted daily meetings to coordinate ongoing work, discuss blockers, and monitor progress throughout the implementation cycles. Overall, the project followed agile software engineering practices emphasising iterative development, continuous feedback, and collaborative decision-making.

E. Evaluation

To ensure quality feedback collection, focus groups and surveys were selected as the primary evaluation tools. The use of qualitative data is essential for this study to identify areas of improvement during iterations, as well as to evaluate the final usability of the project. Additionally, quantitative data was collected to measure the intuitiveness of *Tasklets* and of

[1] Under review in a double-blind review process

[2] https://github.com/TracyTeam/tracybot/wiki

[3] https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow

[4] https://www.conventionalcommits.org/en/v1.0.0/

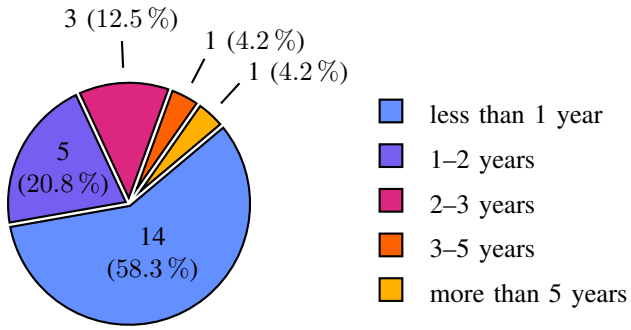


Figure 1: Professional Software Development Experience of Survey Participants

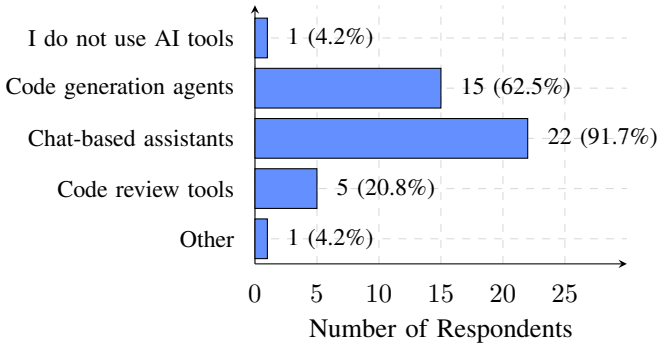


Figure 2: Types of AI Tools Used by Survey Participants (Select all that apply, $n = 24$)

the tool's UI & UX. The target demographic for data collection is software engineering practitioners and students.

The artifact was evaluated by a focus group at the end of the first iteration and using a survey at the end of the second iteration.

1) Focus Group

A semi-structured focus group (see focus group guide in Appendix A) was conducted with a mix of 5 software practitioners and 2 students, selected via convenience sampling, to evaluate the tool at the end of the first iteration. The participants' reported roles and years of professional experience are summarized in Table I.

Table I: Focus Group Participants' Demographics (Cycle 1)

ID	Role	Years of Exp.
S1	Student (Master's in Software Engineering)	<1 year
S2	Student (Master's in Game Development)	<1 year
P1	Software Engineer (Aviation)	<1 year
P2	DevOps Engineer (Telecom)	<1 year
P3	Software Engineer (Automotive)	<1 year
P4	Embedded Engineer (Automotive)	<1 year
P5	IT Engineer (Automotive, Logistics)	1-2 years

To analyse the data from the focus group, qualitative analysis was conducted. The process involved all authors of the study reading and analysing the transcript in one session, where key insights were identified, extracted, and mapped to actionable items to guide the next iteration. Disagreements were resolved through discussions on the spot during the session.

2) Survey

A survey was used to evaluate the second and final iteration. It was conducted with 24 participants selected via convenience sampling, most of whom are students. In addition, two of the participants reported involvement in the initial focus group. The professional experience of the survey participants is summarised in Figure 1.

The aim of the survey was to evaluate the intuitiveness and usability of *Tracybot*, with a focus on the perceived correctness of the attribution algorithm. The survey participants were presented with screenshots of various scenarios that can be encountered using *Tracybot*, showing the AI attribution

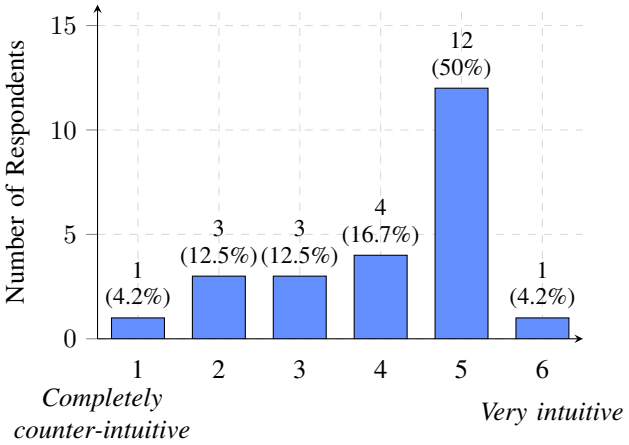


Figure 3: Likert Scale Results
Would you consider the UI intuitive?

highlighting and the Tasklet information. Qualitative questions were used to collect feedback on the examples, allowing the respondents to express disagreements with the presented AI attribution. Two quantitative Likert scale questions alongside additional qualitative questions were used to evaluate the intuitiveness of the UI&UX, and to identify potential improvements. The complete survey can be found in Appendix B.

The participants reported a high degree of engagement with AI tools. As shown in Figure 2, the most commonly used tools were chat-based assistants, used by 22 (91.7%) respondents, followed by code generation agents, used by 15 (62.5%) respondents. 5 respondents (20.6%) reported using code review and analysis tools, while only a single participant reported not using any AI tools. This confirms that the sample is largely familiar with AI-assisted development workflows, in particular agentic development, for which Tracybot is designed to be used with.

Regarding the intuitiveness of the interface, responses were mostly positive, as illustrated in Figure 3. On a six-point Likert scale ranging from "completely counter-intuitive" to "very intuitive", more than half of the participants (54.2%, 13 participants) selected a positive rating. Three participants (12.5%) rated 3, and three participants (12.5%) rated 2, while only a single respondent (4.2%) rated the interface at the lowest rating. Taken together, 71% of participants perceived the UI as intuitive.

The clarity of the Tasklet concept was perceived relatively well, as shown in Figure 4. No participants selected either

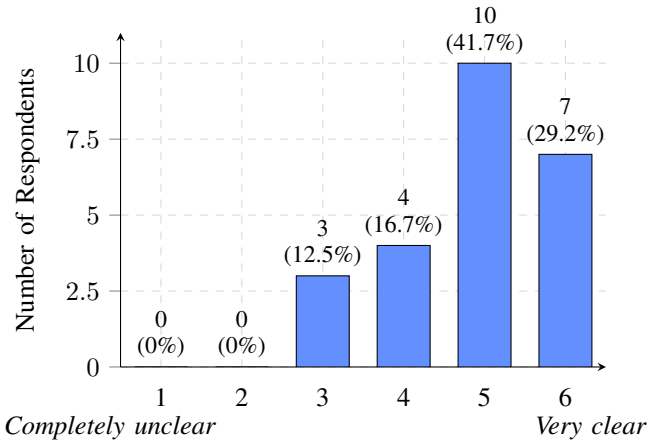


Figure 4: Likert Scale Results
Would you agree that grouping multiple relevant prompts into a Tasklet is a clear method to represent the developer's intent while prompting?

of the two lowest ratings. Responses mainly focused on the upper end of the scale, with 41.7% (10 participants) selecting the second-highest rating and 29.2% (7 participants) selecting the highest. Furthermore, 16.7% (4 participants) and 12.5% (3 participants) gave the fourth and third ratings, respectively. The results indicate a strong understanding of the Tasklet notion and a positive perception of it as an effective representation of developer intent.

IV. Tracybot: The Artifact

The artifact constructed during this research is *Tracybot*, a traceability tool designed to link LLM-generated code to its originating prompts. This section details the conceptual overview, system architecture, and practical usage of the final tool.

Tracybot's source code is publicly available at https://github.com/TracyTeam/tracybot. The *User Guide* (Appendix C) covers installation and setup, while the *Developer Guide* (Appendix D) documents local compilation and development workflows.

A. Overview and Objectives

Tracybot aims to enable developers to quickly identify LLM-generated code and help them understand the reason and motivation behind AI changes by highlighting LLM-generated code and displaying relevant prompts alongside it. The project seeks to move beyond unreliable self-reporting by automating origin tracking within the development environment in a non-disruptive way.

We argue that introducing such a tool could provide the following benefits:

- *Documentation and Code Explainability*: LLM prompts used for code generation often provide additional conceptual guidance. *Tracybot* documents how the LLM was used even when the generated code itself was modified.
- *Disambiguating Developer Intent*: Prompts capture the developer's underlying reasoning and serve as the conceptual starting point for a change. The work of Ogenrwot and Businge [18] shows that developers typically treat LLM output as a draft rather than a final implementation. Tracking the prompt, therefore, helps reviewers understand the original intent and identify any misalignment between the developer's goal and the LLM-generated result.
- *AI Use Transparency*: Transparency is essential for responsible AI adoption in software development. Because AI systems can have technical limitations such as architectural misalignment or limited repository and context awareness, teams need visibility into where and how AI contributes to the codebase to ensure appropriate rigour and review. *Tracybot* improves this transparency by making LLM-assisted changes visible and traceable, helping teams maintain high standards of quality, safety, and accountability while confidently adopting AI.
- *Model Provenance and Evaluation*: LLM metadata, such as the model and its settings, could be traced along with the prompt, which could allow analysing the performance of various models to optimize for a particular codebase.

B. Architecture

The architecture of *Tracybot* consists of 3 main components: An agent integration, the tracing backend, and an editor integration. This architecture was designed to be easily extensible, allowing easy integration with more tools in the future.

In this study, one agentic tool and one editor were selected for integration. *OpenCode*[5] was chosen due to its open-source nature, enabling deep integration and inspection of internal behaviour, as well as its existing plugin ecosystem. Similarly, *Visual Studio Code (VS Code)*[6] was selected as the editor due to its widespread adoption and mature extension framework.

The architecture is visualized in Figure 5. Technical details of each component are summarized below.

1) OpenCode Plugin

The *OpenCode* Plugin acts as the agent integration layer, implemented as a plugin for the *OpenCode* CLI tool[7]. Its primary responsibility is intercepting events sent by *OpenCode* and calling the tracing backend to save the LLM changes and metadata. To fully preserve developer intent, the plugin implements a notion of *Tasklets*, grouping relevant Planning prompts with a Build prompt that produces code changes (see paragraph V-B0b). The plugin operates silently and is transparent to the user, meaning that no additional user interaction is needed when using *Tracybot* with *OpenCode*.

2) Tracing Backend

The Tracing Backend implements the core functionality of code-to-prompt traceability. It operates via a series of Python scripts (`tracy.py` and hooks) that are invoked by a series of automated *Git* Bash hooks. To fully understand the backend's architecture, it is essential to define the underlying mechanisms it leverages:

- *State Snapshots:* Captures of the repository's working directory, including uncommitted changes.
- *Git References (Refs):* Named pointers to commits within a repository. Refs provide human-readable identifiers for commit hashes and are used to represent branches, tags, and other repository states. Common examples include `refs/heads/*` for local branches and `refs/tags/*` for tags.
- *Git Namespace:* A custom reference hierarchy. By storing snapshots under a decoupled `refs/tracy-local/*`

[5]https://opencode.ai
[6]https://code.visualstudio.com
[7]https://opencode.ai/docs/cli/

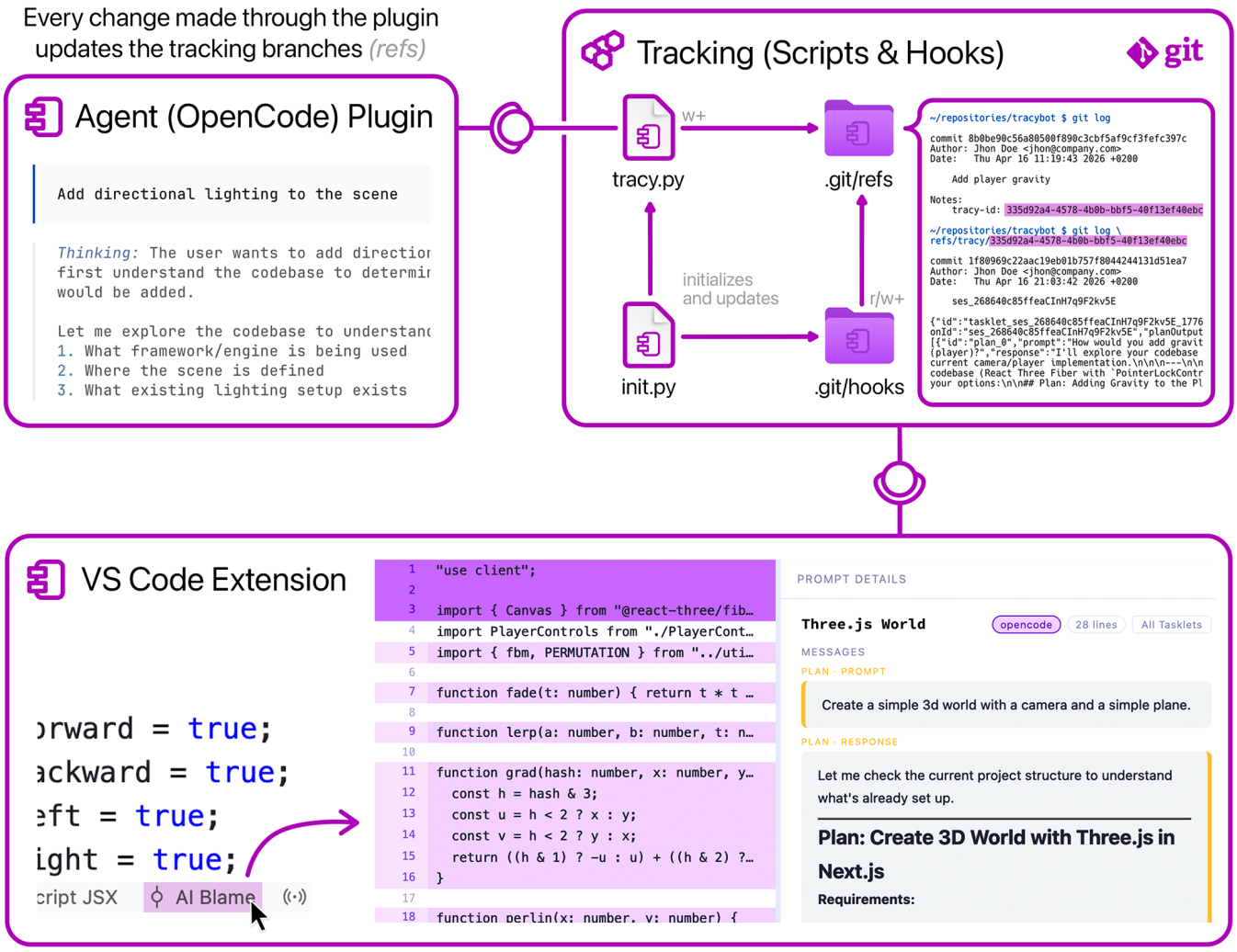


Figure 5: Architecture Diagram

namespace rather than the standard `refs/heads/*`, the backend hides these commits from the user's primary branch history.

- *Git Notes:* A native *Git* feature used to attach metadata to an existing commit without altering the commit's hash.
- *Git Hooks:* Scripts that *Git* executes automatically before or after events such as committing or pushing.

The traceability workflow consists of three primary stages: Snapshot Creation, Commit Linking, and History Rewriting.

a) Snapshot Creation

As the user interacts with the AI agent, the agent integration invokes `tracy.py` to capture state snapshots immediately after the agent inserts code changes. The backend then stores the state of tracked files as hidden snapshot commits within the `refs/tracy-local/*` namespace. As illustrated in Figure 6, this creates a decoupled chain of changes that parallels the user's working directory without impacting their standard workflow. In addition, user snapshots are created immediately before agent modifications and on each *pre-commit hook* (see paragraph IV-B2b) to ensure the AI-generated snapshots only contain code changes created by the agent.

b) Commit Linking

When the user commits their work, the hidden snapshots are scanned and linked to the commit using *pre-commit* and *post-commit* hooks. The standard commit is linked to the head of the hidden snapshot chain by attaching a *Git* note containing the ID of the chain.

To prevent potentially sensitive or untracked snapshot data from being pushed to the remote, two *Git* namespaces are maintained: `refs/tracy-local` and `refs/tracy`. In this design, `refs/tracy-local` acts as a local workspace containing raw snapshots, while `refs/tracy` contains the filtered, remote-safe representation of that data. Filtering occurs in the *post-commit* hook, which reconstructs the snapshot chain by retaining only snapshots corresponding to files present in the committed change, before promoting the result to the main `refs/tracy` namespace.

The full flow that occurs upon a user commit can be seen in Figure 8 as a sequence diagram.

c) History Rewriting

Developers frequently rewrite their local *Git* history using commands like `git commit --amend` or `git rebase`. To preserve traceability across these operations, the backend employs a *post-rewrite hook*. Whenever a commit is modified (e.g., Commit C becomes C'), the hook merges the associated tracking chains and applies a new *Git* note to the rewritten commit hash. As a result, the rewritten commit remains linked to its original snapshots. The rewrite flow is shown in Figure 7 as a sequence diagram.

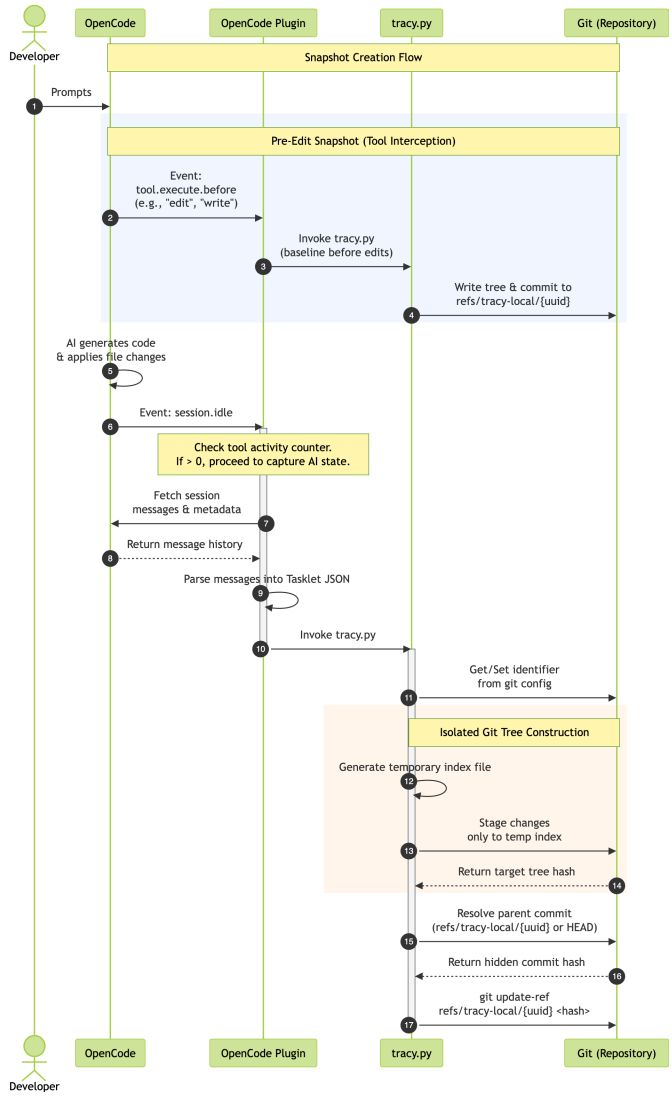


Figure 6: Snapshot Creation

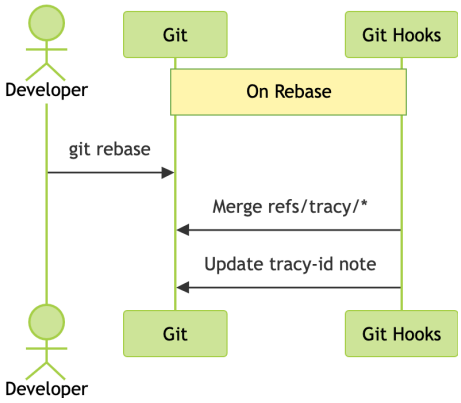


Figure 7: Rewrite Flow

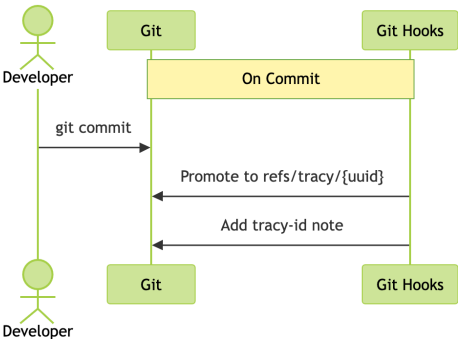


Figure 8: Commit Flow

d) Remote Synchronization

In addition to the primary flows outlined above, the backend utilizes hooks for synchronization with remote repositories. Namely, the *pre-push* and *reference-transaction* hooks ensure that local *Git* notes are successfully synchronized with the `origin` remote, by explicitly fetching remote note updates and merging them into the local notes. Note synchronization uses a union-based merge strategy to reconcile new notes from remote with local notes. The local note namespace takes priority, with the remote acting solely as a synchronization layer.

3) VS Code Extension

The *VS Code* Extension serves as the editor integration and visualisation component. Upon workspace activation, the extension runs a series of checks, including a repository initialisation check and agent integration check. If the repository has not been initialised, it displays a prompt to execute the setup script (`init.py`). If no supported agent is installed or the integration is missing, the user is prompted to install the missing component.

To visualize traced data, the extension reconstructs the repository's history by compiling data from standard commits, active *Git* notes, and the hidden `refs/tracy-local/*` snapshot namespace. The extension's history-building algorithm computes line-level diffs that track code changes over time, filtering out human modifications to accurately trace surviving LLM-authored code back to its origin prompt.

Developers interact with the history via an "AI Blame" panel, which highlights AI-authored lines in the read-only editor window and shows all Tasklets that modified those lines (see UI in Figure 9).

To ensure relevance of the displayed Tasklet, only the most recent Tasklet is displayed, while previous Tasklets that modified the same sections can be accessed via the "All Tasklets" view, including Tasklets with no surviving lines. This mechanism, along with the blame view being read-only, was implemented based on the feedback collected during the first design cycle evaluation (see subsubsection III-E1).

V. Findings

This section generalises the findings from the two design iterations and presents them in relation to the research questions.

A. RQ1: Problem Space

The initial problem investigation was based on analysis of related work and revealed a gap in current software engineering practices - the prompts used to generate code are rarely treated as traceable artifacts. Following the analysis, as well as internal brainstorming sessions, high-level requirements were elicited to define the core functionality of *Tracybot*. The requirements are presented in Table II.

In addition to the foundational requirements, practical issues were later uncovered via internal validation and the focus group. During the first cycle, the team identified critical issues with the implementation, namely: (1) some version control operations such as rebasing, squashing, or amending could

Table II: System Requirements

Requirement	Context	Source
The system shall capture all prompt interactions executed via an agentic tool that result in a code change.	User messages, model's response and any follow-up questions have to be captured in order to be shown to the user by the extension.	Initial Design
The system shall group related Plan and Build prompts into a single Tasklet.	Agents have two different prompting modes: Plan and Build. In Plan mode the model outlines changes but does not implement them. A Tasklet consists of any number of Plan prompts and exactly one Build prompt.	Initial Design
The system shall record LLM-generated code changes.	Whenever an agent makes a change, snapshots before and after the change must be kept to enable calculating code diffs that contain only the agent's changes. The snapshots must be decoupled from version control.	Initial Design
The system shall associate Tasklets with code changes.	When the user commits changes to version control, the hidden Tasklet snapshots have to be linked to the created commit.	Initial Design
The system shall display Tasklet details in the editor when "AI-blame" is enabled.	When the user clicks the AI-blame button, it should open a new tab including the Tasklet information of the latest saved version of the file	Initial Design
The system shall highlight LLM-authored blocks when "AI-blame" is enabled.	Prompt information is shown only on demand to avoid distraction, and is displayed as metadata rather than inline code artifacts; the source file remains unmodified.	Initial Design
The system shall maintain tracking integrity during complex version control operations.	Operations such as rebasing, squashing, or amending must not lead to corrupted tracking history.	Internal Testing
The system shall handle subsequent agent modifications to previously traced lines.	If multiple Tasklets have modified the same line of code, the user should be able to see all Tasklets that have changed that line previously.	Focus Group
The system shall respect repository exclusion rules for files created by the agent.	The tracing backend may unintentionally capture sensitive files generated by the agent even if they have been explicitly excluded by the user or project.	Internal Testing
The system shall ignore insignificant changes during code tracking.	Tracking minor changes, such as formatting, inflates the history and reduces usefulness.	Focus Group

lead to corrupted tracking under certain scenarios; (2) the tracing backend could capture sensitive files generated by the agent even if explicitly excluded.

Alongside the internal findings, the focus group highlighted two additional challenges. First, participants noted that associating code with only the most recent prompt erases valuable history when an agent iteratively modifies the same block of code. Second, they pointed out the sensitivity of code tracking to minor, non-semantic changes. Initially, the tracking mechanism treated minor formatting adjustments as significant modifications, which participants perceived as inflating the traceability history and reducing its usefulness. As one participant noted regarding formatting changes:

> **P5:** *"I would ignore this completely [...] if you're using agentic AI, you also most likely have some sort of tools [...] for formatting the file."*

The aforementioned issues were formalised as additional requirements, presented in Table II with the source being "internal testing" or "focus group".

B. RQ2: Necessary Components

Based on the elicited requirements, three necessary components for *Tracybot* were identified: an agent integration, a tracing backend, and an editor integration (the implementation of which is detailed in subsection IV-B). To identify the needed components, the requirements were analysed and used to devise the system design. The decisions behind the design of *Tracybot* and its required components are formalized as Architecture Decision Records (ADRs).

a) ADR1: Decouple tracing logic from tool integrations

Context: The system must intercept AI prompts, execute prompt tracing logic, and visualize this data in the developer's environment. Embedding the core tracking logic directly into the agent plugin or the editor extension would tightly couple the system to those specific tools.
Decision: Extract the core prompt tracing logic into a standalone backend that operates independently of the agent and the editor.
Rationale: This modularity separates concerns, keeping the agent and editor plugins lightweight as they only need to handle data interception and visualization, respectively. This design ensures extensibility, where adapting *Tracybot* to support new tools in the future requires minimal implementation, as new integrations simply interface with the existing tracing backend.
Consequences: Introduces additional architectural complexity by requiring a third discrete component implementing the tracing logic, and the related communication between the components.

b) ADR2: Group Plan prompts with each Build prompt

Context: Agentic tools commonly include Plan and Build modes, used for reasoning about changes and making code changes, respectively. A common workflow is to iteratively prompt the agent in Plan mode until satisfactory, then switch to Build mode.
Decision: Introduce the notion of a Tasklet—on code changes, group Plan mode prompts (if any) together with a Build mode prompt into a Tasklet.
Rationale: If *Tracybot* only tracked Build mode prompts, only short approval prompts would be tracked, and the prompts showing user intent would be lost, hindering code explain-

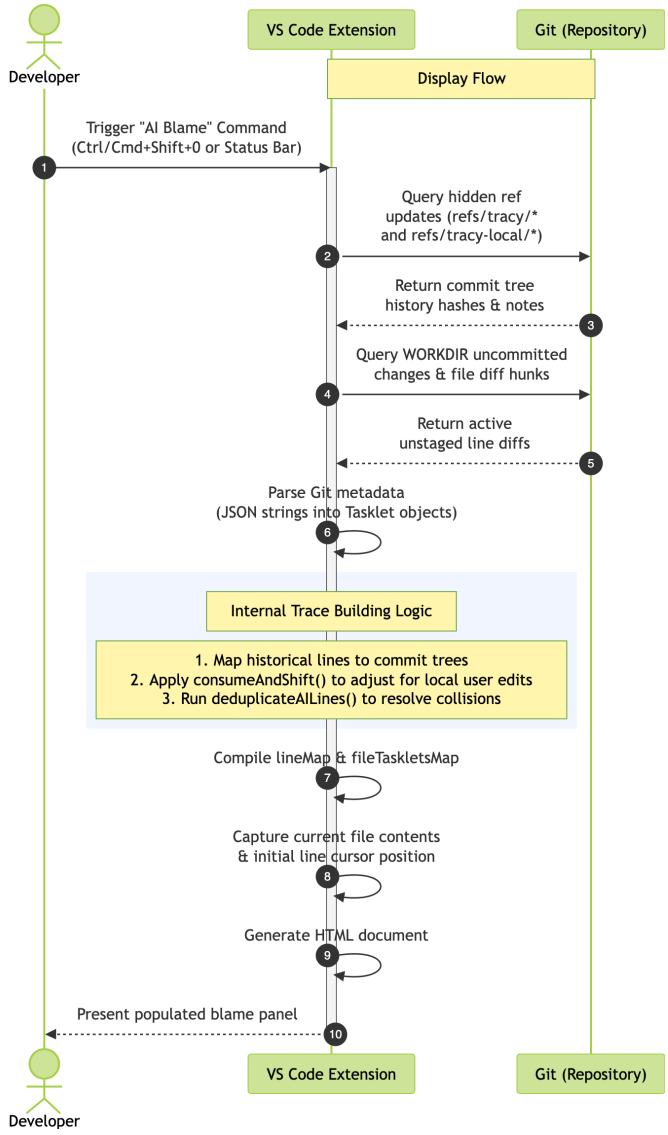


Figure 9: Display Flow

ability.

Consequences: Additional logic is needed to build and parse Tasklets, introducing processing and storage overheads compared to tracking a single prompt per change.

c) ADR3: Traceability via Git

Context: *Tracybot* has to reliably trace LLM-generated code to its source prompts while integrating with common workflows.

Decision: Traceability is implemented using hidden *Git* commits created by *OpenCode*. When the user creates a commit, the hidden chain is tracked to the user commit via *Git* notes.

Rationale: The alternative (file copies) was rejected due to incompatibility with *Git* workflows, especially merging. The accepted decision is transparent to the user and integrates well with Git.

Consequences: The tool becomes dependent on *Git*, requiring that users do not manually use *Git* notes, and must additionally handle *Git* operations such as rebasing, squashing, and amending.

d) ADR4: Non-obstructive workflow

Context: *Tracybot* should not interfere with existing workflows.

Decision: Use *Git* hooks (pre-commit, post-commit, and post-rewrite) to automatically update the tracking state in response to repository events.

Rationale: Commit hooks ensure robust tracking. The post-commit hook attaches a note linking the commit to the hidden chain, and the post-rewrite hook merges notes to preserve history during rebases or squashes. This minimizes user friction without requiring aliases or wrappers.

Consequences: Initialization must preserve pre-existing user-defined hooks to avoid disruption.

e) ADR5: Determine code change significance with BLEU

Context: Simple code changes such as formatting or renaming should not shift attribution.

Decision: Use BLEU similarity score on *Git* hunks to approximate change significance. Consider changes with similarity above a certain threshold as insignificant, and do not shift attribution for those changes.

Rationale: Several alternatives to measure change similarity were examined. Advanced similarity scores such as CodeBLEU [19] are not suitable for this use-case, given that syntax scanning cannot be applied to small hunks. CrystalBLEU [20] was rejected as an alternative due to needing language-specific corpora, and only showing insignificant improvements over BLEU in the context of *Git* hunks, according to internal testing.

Consequences: Changes with high similarity do not shift attribution. A similarity threshold must be established below which changes are considered significant.

C. RQ3: Evaluation

1) Workflow Integration and UI

In general, participants positively responded to the concept and implementation of *Tracybot*, particularly its integration into existing workflows. Focus group participants described the tool as well thought out, well integrated into the editor and free from introducing additional friction. They also appreciated the non-obstructive nature of the tracking mechanism. The use of *Git* notes and hidden references was seen as a safe and practical design choice that preserves existing version control practices.

An additional point of strong consensus was the preference for maintaining a clear separation between the traceability interface and the main code editor. In the focus group, participants favoured the existing design, where the “AI Blame” view is presented in a separate tab rather than directly embedded into the coding environment. This approach was seen as reducing visual clutter and preserving focus during development, while still allowing easy access to traceability information when needed. This was corroborated by the survey where 71% of participants deemed the UI intuitive.

Feedback also strongly emphasized improvements to the *VS Code* extension’s user interface. Focus group participants highlighted the need for better alignment with existing editor settings, such as respecting font size preferences to ensure readability across different environments, which would later be implemented during the second cycle.

Additionally, several participants proposed introducing a visual representation of Tasklets, such as a timeline or tree structure, to make the sequence of changes more intuitive and easier to follow. Participants also suggested features such as marking or saving prompts, highlighting particularly useful prompts, and filtering Tasklet history to reduce clutter

and improve the usefulness of traceability data. While these ideas were not evaluated within the scope of this study, they represent promising directions for future work.

The suggested UI improvements were implemented in the second iteration and evaluated through the survey. All focus group participants who completed the survey rated the UI&UX question (subsection B-F) at least 5 out of 6 on the Likert scale, indicating a generally positive perception of the updated interface.

2) Technical Issues

The change significance feature implemented during the second cycle (paragraph V-B0e) was evaluated in the survey using code examples. For all examples, over 83.3% of the participants agreed with the tool's code attribution except for one scenario. That scenario shows that using similarity as an approximation of code change significance can be perceived as unreliable or unintuitive. We conclude that, given the varying nature of code changes, an arbitrary similarity threshold is not a reliable way to determine the perceived significance of changes.

Participants also expressed confusion regarding the binary attribution model, where code is labelled as either human or LLM-generated. This approach was seen as overly simplistic, particularly in cases where LLM-generated code is later modified by a developer. Instead, participants advocated for a more layered or historical representation of authorship that preserves the evolution of a code segment.

Finally, infrastructure concerns were raised regarding long-term storage overhead. The accumulation of prompt history could become overwhelming, with one participant specifically noting concerns about long repository clone times in CI pipelines for large mono-repos if the traceability data is not carefully managed.

3) Socio-Technical Concerns

Participants highlighted the tool's potential value in professional environments, especially in contexts requiring auditing, accountability, or adherence to emerging AI policies. The ability to trace and potentially revert LLM-generated changes was considered particularly useful for maintaining code quality and security.

> **P4:** *"[...] we just started implementing outside AI agents. So now that everybody's kind of using them and seeing how far they can test them, it'd be really good to have this sort of blame that shows exactly what was generated [...]."*

Finally, a more critical perspective emerged from the potential misuse of the tool in workplace settings. Some participants cautioned that traceability features could be repurposed for micromanagement or performance monitoring, potentially discouraging developers for using AI tools transparently. This highlights an important socio-technical consideration: while the tool enhances transparency, its adoption must be carefully aligned with organizational culture and policies to avoid unintended negative consequences.

VI. Limitations

A. Delimitations

Tracybot does not attempt to detect LLM-authored code the origin of which is not the agent, and which is not generated while using the *Tracybot* agent integration. It focuses on integration with an agentic tool used to generate code and with an IDE for displaying the tracing information.

The code change significance mechanism (paragraph V-B0e) utilizes basic BLEU similarity and an arbitrary threshold. Identifying a reliable method to evaluate the significance of perceived code change is out of scope for this work. In addition, while *Tracybot* touches upon significance of human modifications to LLM-generated code, it does not aim to define code ownership in relation to distinguishing between Human or LLM authorship.

B. Validity threats

- *Limited Sample Size*: The 11-week time frame limits the number of iterations and opportunities to evaluate the artifact, potentially impacting the generalizability and validity of the findings and collected feedback.
- *Semantic Subjectivity*: Thresholds for what constitutes a logic change may vary between developers.
- *No user experience evaluation*: Due to time restrictions, there was no opportunity to observe users interacting with the tool. The focus group was held as a presentation, and the final cycle evaluation used screenshots of the tool instead of requiring respondents to use the tool.

VII. Conclusion and Outlook

This research explored how prompt traceability can be introduced into AI-assisted software development workflows through the design and evaluation of *Tracybot*. The resulting artifact demonstrates that LLM-generated code can be linked to its originating prompts in a way that integrates with existing development practices and version control workflows.

The evaluations showed that developers generally viewed prompt traceability positively, particularly for improving transparency, explainability, and ease of review of AI-assisted code changes. The tool supports common workflows such as rebasing, squashing, amending, and cherry-picking while remaining largely non-obstructive to developers.

At the same time, the study identified several limitations and open challenges. Participants noted that line-level attribution alone is insufficient for accurately representing collaborative human-AI authorship, especially after subsequent human modifications. Additional concerns included storage overhead, handling of non-semantic changes such as formatting, and the potential organizational misuse of traceability data.

Overall, the results suggest that prompt traceability is both technically feasible and practically useful within modern software engineering workflows. As AI-assisted development becomes increasingly common, treating prompts as traceable development artifacts may help improve transparency and accountability in collaborative human-AI software development.

A. Future Work

Several directions for future work surface from this project. First, the current implementation could be extended to support additional agentic tools and development environments, such as *Claude Code*, *Cursor*, and JetBrains IDEs. Since the tracing backend was intentionally designed as a stand-alone component, the architecture supports future integrations with relatively limited changes.

Second, more advanced attribution mechanisms should be explored. The current line-based tracing approach struggles with partial modifications and detecting insignificant changes.

Future work could investigate a finer-grained attribution algorithm that combine syntactic, semantic, and contextual analysis to better model authorship between developers and LLMs. Configurable attribution thresholds and language-aware similarity metrics could further improve usability and precision.

Third, the collected traceability data could support additional development workflows beyond attribution alone. For example, prompt histories and Tasklets could be reused as contextual input for future AI-assisted sessions, automatically generating summaries of implemented features or assisting reviewers during pull request evaluation.

In addition, integration with platforms such as *GitHub* or *GitLab* could allow reviewers to inspect AI-generated contributions directly within pull requests and potentially apply different review policies depending on the level of AI usage. Such integrations could further support code review, auditing, and AI transparency practices in software development teams.

Finally, future research should further investigate the ethical implications of AI traceability tools. Although increased transparency may improve accountability and software quality, concerns about surveillance, micromanagement, and developer trust must also be carefully considered. Understanding how such tools can be responsibly introduced within professional environments remains an important area of study.

References

[1] S. Bistarelli, M. Fiore, I. Mercanti, and M. Mongiello, "Usage of large language model for code generation tasks: A review," *SN Computer Science*, vol. 6, no. 6, Jul. 2025, ISSN: 2661-8907. DOI: 10.1007/s42979-025-04241-5. [Online]. Available: http://dx.doi.org/10.1007/s42979-025-04241-5.

[2] B. Idrisov and T. Schlippe, "Program code generation with generative ais," *Algorithms*, vol. 17, no. 2, p. 62, Jan. 2024, ISSN: 1999-4893. DOI: 10.3390/a17020062. [Online]. Available: http://dx.doi.org/10.3390/a17020062.

[3] O. Kraishan, *The ai attribution paradox: Transparency as social strategy in open-source software development*, 2025. DOI: 10.48550/ARXIV.2512.00867. [Online]. Available: https://arxiv.org/abs/2512.00867.

[4] S. M. Kashif, P. Liang, and A. Tahir, "On developers' self-declaration of ai-generated code: An analysis of practices," *ACM Transactions on Software Engineering and Methodology*, Oct. 2025, ISSN: 1557-7392. DOI: 10.1145/3771937. [Online]. Available: http://dx.doi.org/10.1145/3771937.

[5] J. He, S. Houde, and J. D. Weisz, "Which contributions deserve credit? perceptions of attribution in human-ai co-creation," in *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, ser. CHI '25, ACM, Apr. 2025, pp. 1–18. DOI: 10.1145/3706598.3713522. [Online]. Available: http://dx.doi.org/10.1145/3706598.3713522.

[6] European Parliament and Council of the European Union, *Regulation (eu) 2024/1689 of the european parliament and of the council of 13 june 2024 laying down harmonised rules on artificial intelligence (artificial intelligence act)*, Jul. 12, 2024. Accessed: Mar. 10, 2026. [Online]. Available: https://eur-lex.europa.eu/eli/reg/2024/1689/oj/eng.

[7] R. Khojah, M. Mohamad, L. Erlenhov, F. G. de Oliveira Neto, and P. Leitner, "Large language model company policies and policy implications in software organizations," *IEEE Software*, vol. 43, no. 1, pp. 64–72, 2026. DOI: 10.1109/MS.2025.3622039.

[8] K. Souali, O. Rahmaoui, and M. Ouzzif, "An overview of traceability: Definitions and techniques," in *2016 4th IEEE International Colloquium on Information Science and Technology (CiSt)*, IEEE, Oct. 2016, pp. 789–793. DOI: 10.1109/cist.2016.7804995. [Online]. Available: http://dx.doi.org/10.1109/cist.2016.7804995.

[9] O. Gotel et al., "Traceability fundamentals," in *Software and Systems Traceability*, J. Cleland-Huang, O. Gotel, and A. Zisman, Eds. London: Springer London, 2012, pp. 3–22, ISBN: 978-1-4471-2239-5. DOI: 10.1007/978-1-4471-2239-5_1. [Online]. Available: https://doi.org/10.1007/978-1-4471-2239-5_1.

[10] B. Huang, C. Chen, and K. Shu, *Authorship attribution in the era of llms: Problems, methodologies, and challenges*, 2025. arXiv: 2408.08946 [cs.CY]. [Online]. Available: https://arxiv.org/abs/2408.08946.

[11] Y. Xu, S. Huang, M. Geng, Y. Wan, X. Shi, and D. Chen, *The influence of large language models on code*, 2025. DOI: 10.48550/ARXIV.2506.12014. [Online]. Available: https://arxiv.org/abs/2506.12014.

[12] T. Bisztray et al., "I know which llm wrote your code last summer: Llm generated code stylometry for authorship attribution," in *Proceedings of the 18th ACM Workshop on Artificial Intelligence and Security*, ser. AISec '25, ACM, Oct. 2025, pp. 28–39. DOI: 10.1145/3733799.3762964. [Online]. Available: http://dx.doi.org/10.1145/3733799.3762964.

[13] M. Akgün, "Author or prompter? scientific writing, identity, and the theseus paradox," *Philosophy, Ethics, and Humanities in Medicine*, vol. 20, no. 1, Oct. 2025, ISSN: 1747-5341. DOI: 10.1186/s13010-025-00195-x. [Online]. Available: http://dx.doi.org/10.1186/s13010-025-00195-x.

[14] Y. Sun, Z. Xu, C. Liu, Y. Zhang, and Y. Liu, "Who is the real hero? measuring developer contribution via multi-dimensional data integration," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, Sep. 2023, pp. 825–836. DOI: 10.1109/ase56229.2023.00102. [Online]. Available: http://dx.doi.org/10.1109/ASE56229.2023.00102.

[15] A. Hevner and S. Chatterjee, *Design research in information systems: theory and practice*. Springer Science & Business Media, 2010, vol. 22.

[16] P. Johannesson and E. Perjons, *An Introduction to Design Science*. Springer International Publishing, 2014, ISBN: 9783319106328. DOI: 10.1007/978-3-319-10632-8. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-10632-8.

[17] E. Knauss, *Constructive master's thesis work in industry: Guidelines for applying design science research*, 2020. DOI: 10.48550/ARXIV.2012.04966. [Online]. Available: https://arxiv.org/abs/2012.04966.

[18] D. Ogenrwot and J. Businge, *Patchtrack: A comprehensive analysis of chatgpt's influence on pull request outcomes*, 2025. arXiv: 2505.07700 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2505.07700.

[19] S. Ren et al., *Codebleu: A method for automatic evaluation of code synthesis*, 2020. DOI: 10.48550/ARXIV.2009.10297. [Online]. Available: https://arxiv.org/abs/2009.10297.

[20] A. Eghbali and M. Pradel, "Crystalbleu: Precisely and efficiently measuring the similarity of code," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22, Rochester, MI, USA: Association for Computing Machinery, 2023, ISBN: 9781450394758. DOI: 10.1145/3551349.3556903. [Online]. Available: https://doi.org/10.1145/3551349.3556903.

APPENDIX A
FOCUS GROUP GUIDE: TRACEABILITY OF AI-GENERATED CODE (TRACYBOT)

1. Introduction & Consent (2 Mins)

Get participants' consent for recording the meeting's audio. Give a surface level introduction of Tracybot.

2. Problem Statement & Context (5 Mins)

Brief explanation of OpenCode and VScode integration. Mention problems tools attempt to solve:

- Untraceable LLM code
- Inconsistent declaration of AI usage
- Lack of developer intent visibility
- The authorship boundary problem (when does AI code become human written?)

3. Product Demonstration & Core Mechanics (15 Mins)

Present a guided walk-through on the tool's usage using an example repository. Show existing Tasklets, creating new Tasklets when using OpenCode, and committing code.

4. Discussion (40+ min)

Following the Demo, discuss the current implementation. Describe the technicalities behind the current implementation. Collect feedback focusing on the following questions:

- Where is the best place to view the AI blame? In a view-only window or integrated in the editable code?
- Explain the two cases:
  - Separate view, forces autosave
  - Editor view, will not autosave but will be out of sync (updates when you save)
- In the tasklets view, what information would you like to see?
- Is the current summary sufficient?

  Helper questions:
  - In what situations would you actually use this tool?
  - Can you think of a recent task where this would have helped you?
  - What would make this a 'must-have' vs 'nice-to-have'?
  - Would this be something you rely on, or only use occasionally?
  - What feels unnecessary or not very useful?
  - Is this saving you time, or just changing how you work?
- Would you like to use the context that the tool provides in separate chatbots? Maybe human-readable summaries about the decisions made?
- How much or what do humans need to modify in AI-generated code before it's no longer considered AI-generated but human-written?
- If they mention history (multiple tasklets per line), how would you want to visualize previous tasklets?
- Is this tool something you would use when developing software?
- What would hinder or discourage using it? (talk about the limitations)
- Are the current limitations unbearable? (No git notes allowed and no build prompts in parallel)
- Is the Git dependency a deal breaker for you?

APPENDIX B
SURVEY QUESTIONNAIRE

A. Data Collection Consent

1) I agree for the data submitted in this form to be processed and used for research purposes. All data collected in the survey will be anonymized.
   - Yes
   - No

B. Demographics

1) What is your current role? (eg., Student, Software Developer, DevOps Engineer) *(short answer)*
2) What industry or domain do you primarily work/study in? *(short answer)*
3) How many years of professional experience do you have in a software development or related field?
   - less than 1 year
   - 1–2 years
   - 2–3 years
   - 3–5 years
   - more than 5 years
4) How frequently do you use AI tools (eg., ChatGPT, Claude Code) in your workflow?
   - Never
   - Rarely (few times per month)
   - Occasionally (few times per week)
   - Frequently (daily)
   - Very frequently (multiple times per day)
5) Which types of AI tools do you use (Select all that apply)
   - I do not use AI tools
   - Code generation agents (e.g., Claude Code, OpenCode)
   - Chat-based assistants (e.g., ChatGPT)
   - Code review / Analysis tools (e.g., GitHub Copilot)
6) How comfortable are you with using generative AI in your development workflow? *(Likert 1 — 6)*

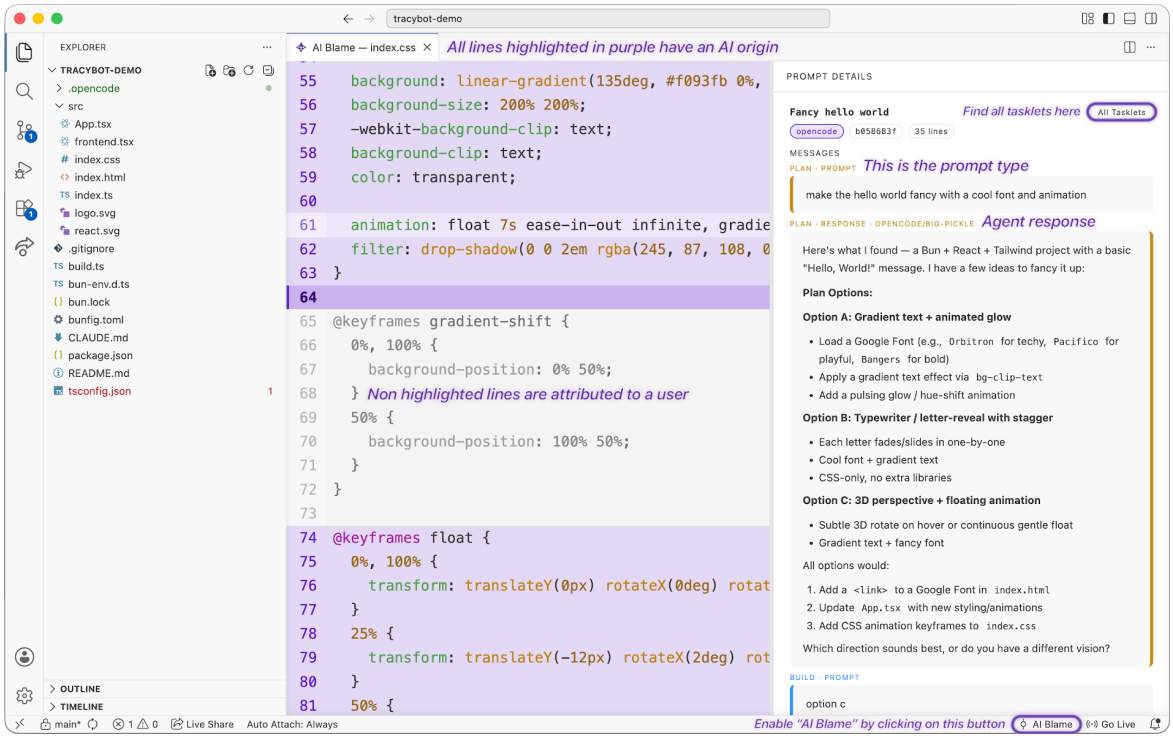
C. AI Attribution and UI Familiarization

To keep code history accurate, Tracybot doesn't blindly credit whoever last touched a file. Instead, it uses a similarity score algorithm to track and assign code ownership. This logic determines whether lines of code are categorised as human-written or AI-generated, controlling the visual highlights you see in the editor.

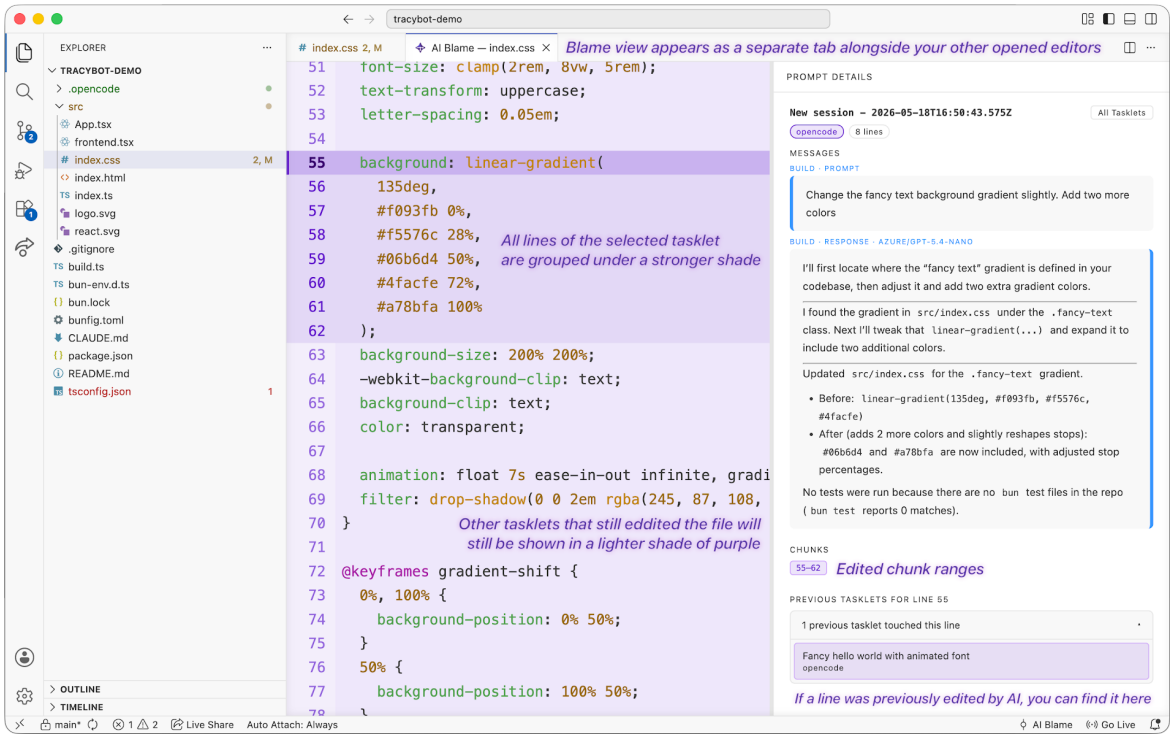
- What is a Tasklet? Every time you make a change through OpenCode, Tracybot packages a set of relevant prompts into a Tasklet. A Tasklet will contain a Build mode prompt alongside the preceding Plan mode prompts, if present. This allows to capture the developer's intent behind the resulting changes in full. Read more about Plan and Build modes here: https://opencode.ai/docs/#add-features.
- How Ownership Shifts (The Similarity Score): When humans and AI both modify the same blocks of code, attribution becomes mixed. To ensure the relevance of traced prompts and accuracy of attribution, Tracybot evaluates these edits by similarity:
  - Minor tweaks: If a human fixes a small typo in an AI-generated function, or an AI slightly formats a human-written block, the underlying attribution is considered unchanged. The similarity score remains high, and original ownership is preserved.
  - Significant rewrites: If code originally written by a human is heavily modified or entirely rewritten by AI, the structural similarity drops below a certain threshold. Ownership of those lines officially shifts to the other party, and the blame updates accordingly. If AI significantly modifies a block written by AI, the new Tasklet takes priority over the previous one, while the latter remains accessible in the history.
  - By clicking on any of these AI-highlighted lines, you can pull up its corresponding Tasklet, allowing you to trace the exact prompt and thought process that inspired the change, no matter how much the file has changed since.

Please familiarize yourself with the interface elements highlighted in purple:

- AI Blame Button (Bottom Right): Click to open the Tracybot panel.
- Snapshot Timeline (Sidebar): A chronological history of your automatically saved work. Click any line of code to see the tasklet that modified it.
- Code Diff View (Main View): Highlights the exact changes (additions/changes) made in the selected snapshot.



- Highlighted Code (Code Diff View): Lines written or modified by AI. Clicking any highlighted line selects it and updates the prompt history on the right.
- Line Chunks: Clickable range indicators (e.g., line numbers). Clicking a chip automatically scrolls the code panel to that section.
- Previous Tasklets Dropdown: Displays older historical AI edits or previous tasklets that touched the active line before the current version took over.



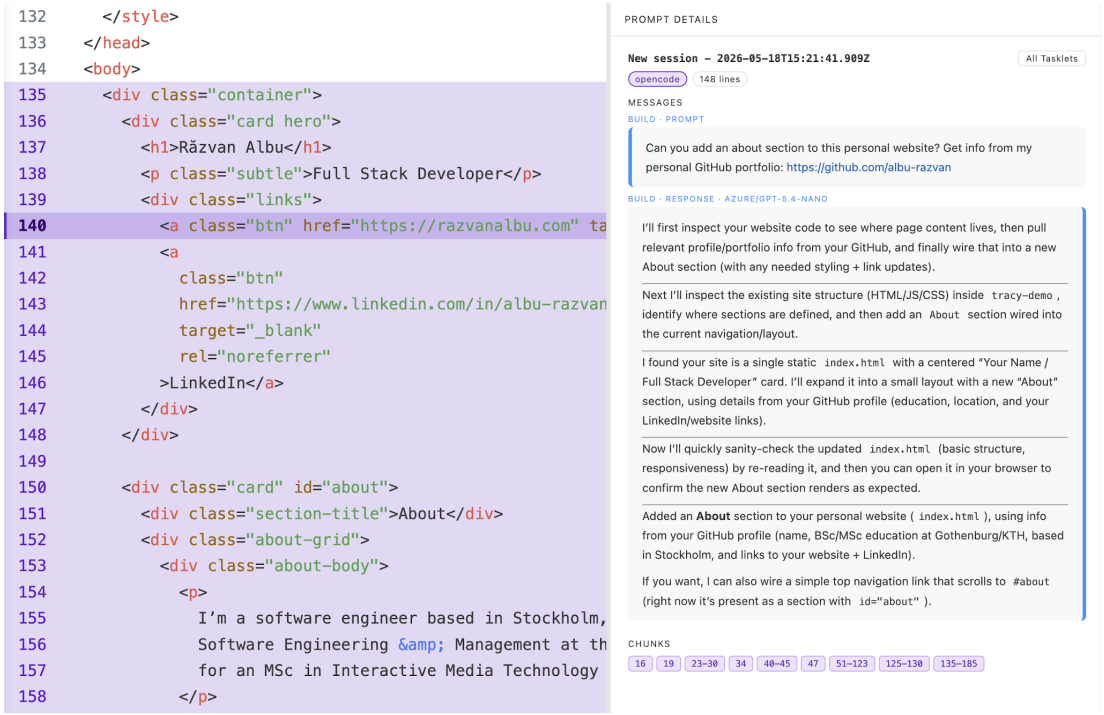
D. User Change to AI Change Behavior

In these examples, an agent (OpenCode) has modified the contents of file index.html. Previously, this content was *FULLY* written by a human.

1) Git diff of the AI-generated changes (current) against the user code (original)

```
   2   2  <html lang="en">
   3   3    <head>
41     8      <style>
42   132      </style>
43   133    </head>
44   134    <body>
45       -    <div class="card">
46       -      <h1>Your Name</h1>
47       -      <p>Full Stack Developer</p>
     135+    <div class="container">
     136+      <div class="card hero">
     137+        <h1>Răzvan Albu</h1>
     138+        <p class="subtle">Full Stack Developer</p>
     139+        <div class="links">
     140+          <a class="btn" href="https://razvanalbu.com" target="_blank"
     141+          <a
     142+            class="btn"
     143+            href="https://www.linkedin.com/in/albu-razvan/"
     144+            target="_blank"
     145+            rel="noreferrer"
     146+          >LinkedIn</a>
     147+        </div>
     148+      </div>
     149+
     150+      <div class="card" id="about">
     151+        <div class="section-title">About</div>
     152+        <div class="about-grid">
     153+          <div class="about-body">
     154+            <p>
```

This is the resulting blame after the user changes mentioned above.



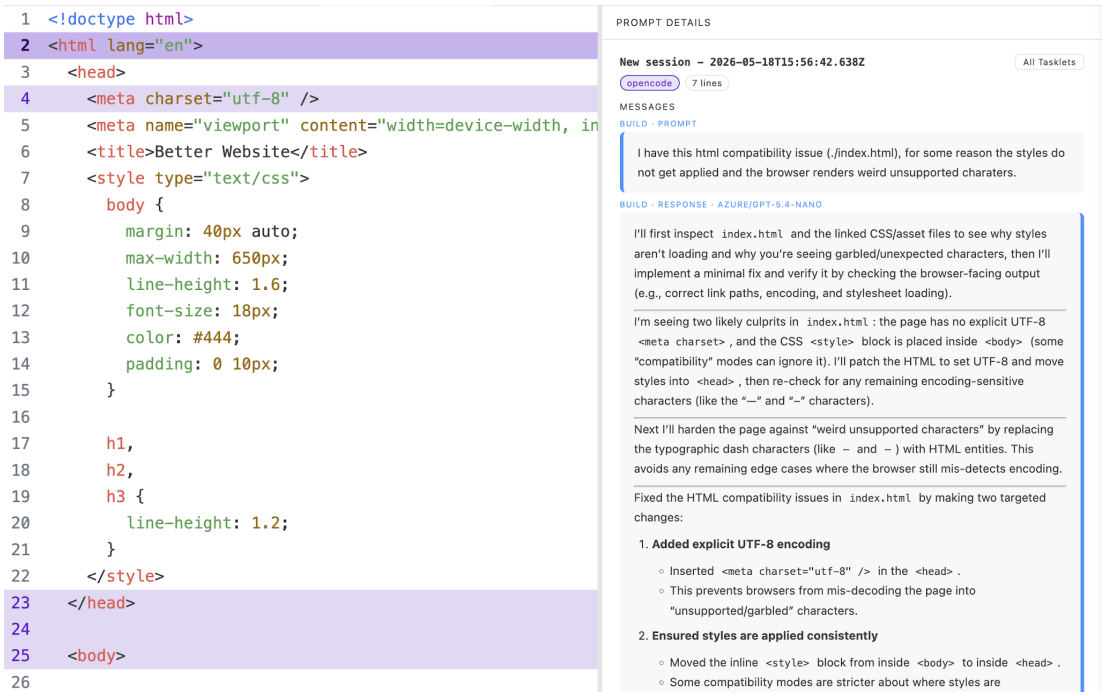
Do you agree with the resulting ownership? If not, explain why under "Other"

- I agree with the resulting code ownership
- Other: *(open ended)*

2) Git diff of the AI-generated changes (current) against the user code (original)

```
1  1  <!doctype html>
2     <html>
   2+ <html lang="en">
3  3    <head>
   4+     <meta charset="utf-8" />
4  5     <meta name="viewport" content="width=device-width, initial-scale=1" />
5  6     <title>Better Website</title>
6     -  </head>
7     -
8     -  <body>
9  7     <style type="text/css">
10 8       body {
11 9         margin: 40px auto;
12 10        max-width: 650px;
13 11        line-height: 1.6;
14 12        font-size: 18px;
15 13        color: #444;
16 14        padding: 0 10px;
17 15      }
18 16
19 17      h1,
20 18      h2,
21 19      h3 {
22 20        line-height: 1.2;
23 21      }
24 22    </style>
   23+ </head>
   24+
   25+ <body>
25 26
```

This is the resulting blame after the user changes mentioned above.



Do you agree with the resulting ownership? If not, explain why under "Other"

- I agree with the resulting code ownership
- Other: *(open ended)*

E. AI Change to User Change Behavior

In these example, a user has modified function `perlinNoise3Layers2D()`. Previously, this function was *FULLY* AI-generated.

1) Git diff of the user changes (current) against the AI-generated code (original)

```
111 102 export function perlinNoise3Layers2D(
112 121   for (let y = 0; y < height; y++) {
113 122     const ny = y * baseFrequency;
114 123     for (let x = 0; x < width; x++) {
115 124       const nx = x * baseFrequency;
116 125
117 126       let v = 0;
118 127       for (let i = 0; i < 3; i++) {
119 128         const freq = layerFreqMultipliers[i] * 1;
120 129         const amp = layerWeights[i] / wsum;
121     -         v += amp * perlin2(nx * freq, ny * freq, perm);
    130 +         const ox = (seed * (i + 1) * 0.61803398875) % 256;
    131 +         const oy = (seed * (i + 1) * 1.41421356237) % 256;
    132 +         v += amp * perlin2(nx * freq + ox, ny * freq + oy, perm);
122 133       }
123 134
124 135       // Convert from ~[-1,1] to [0,1]
125 136       out[y * width + x] = clamp01(v * 0.5 + 0.5);
126 137     }
127 138   }
128 139
129 140   return out;
130 141 }
```

This is the resulting blame after the user changes mentioned above.

```
120   // Sample in a consistent coordinate space.
121   for (let y = 0; y < height; y++) {
122     const ny = y * baseFrequency;
123     for (let x = 0; x < width; x++) {
124       const nx = x * baseFrequency;
125
126       let v = 0;
127       for (let i = 0; i < 3; i++) {
128         const freq = layerFreqMultipliers[i] * 1;
129         const amp = layerWeights[i] / wsum;
130         const ox = (seed * (i + 1) * 0.61803398875) %
131         const oy = (seed * (i + 1) * 1.41421356237) %
132         v += amp * perlin2(nx * freq + ox, ny * freq
133       }
134
135       // Convert from ~[-1,1] to [0,1]
136       out[y * width + x] = clamp01(v * 0.5 + 0.5);
137     }
138   }
139
140   return out;
141 }
```

PROMPT DETAILS

New session – 2026-05-18T13:48:19.816Z All Tasklets

opencode 2cee610a 142 lines

MESSAGES

BUILD · PROMPT

> Make a simple noise generator that generates perlin noise with 3 different layers. JS is fine

BUILD · RESPONSE · AZURE/GPT-5.4-NANO

I'll first inspect the repo to see what frontend/runtime setup you already have, then I'll add a small self-contained JS noise generator (Perlin) with 3 layers and an example canvas render.

Repo search seems to have returned nothing, so I'm going to re-run the lookup with the workspace root as the search path to confirm what files are present before adding new ones.

Repo is essentially empty (only `.gitignore` and an `.opencode` folder), so I'll add a self-contained `noise/perlin3layers.js` module plus a tiny `demo.html` that renders the noise to a canvas.

I've added a self-contained Perlin implementation (`perlin3layers.js`) plus a `demo.html` you can open to see the 3-layer noise rendered on a canvas. Next I'll quickly sanity-check syntax by running a minimal JS parse check and ensure the module exports correctly.

Added a simple **2D Perlin noise generator with 3 layers (octaves)** and a tiny canvas demo.

Files

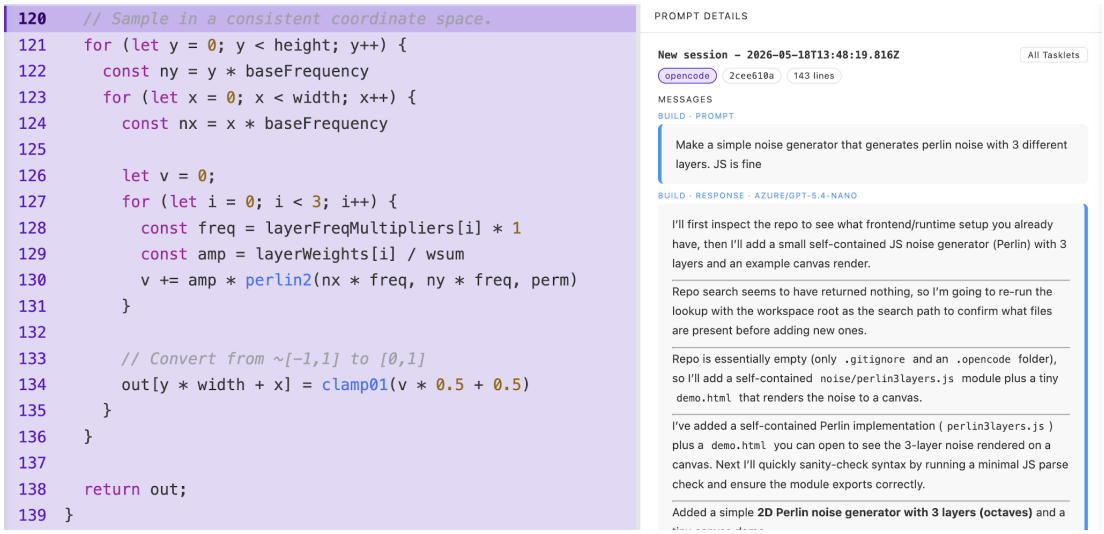
Do you agree with the resulting ownership? If not, explain why under "Other"

- I agree with the resulting code ownership
- Other: *(open ended)*

2) Git diff of the user changes (current) against the AI-generated code (original)

```
111 102 export function perlinNoise3Layers2D(
112 121   for (let y = 0; y < height; y++) {
113     -     const ny = y * baseFrequency;
    122+      const ny = y * baseFrequency
114 123       for (let x = 0; x < width; x++) {
115     -       const nx = x * baseFrequency;
    124+        const nx = x * baseFrequency
116 125
117 126         let v = 0;
118 127         for (let i = 0; i < 3; i++) {
119     -         const freq = layerFreqMultipliers[i] * 1;
120     -         const amp = layerWeights[i] / wsum;
121     -         v += amp * perlin2(nx * freq, ny * freq, perm);
    128+          const freq = layerFreqMultipliers[i] * 1
    129+          const amp = layerWeights[i] / wsum
    130+          v += amp * perlin2(nx * freq, ny * freq, perm)
122 131         }
123 132
124 133         // Convert from ~[-1,1] to [0,1]
125     -       out[y * width + x] = clamp01(v * 0.5 + 0.5);
    134+        out[y * width + x] = clamp01(v * 0.5 + 0.5)
126 135       }
127 136     }
128 137
129 138     return out;
130 139   }
```

This is the resulting blame after the user changes mentioned above.



```
120   // Sample in a consistent coordinate space.
121   for (let y = 0; y < height; y++) {
122     const ny = y * baseFrequency
123     for (let x = 0; x < width; x++) {
124       const nx = x * baseFrequency
125
126       let v = 0;
127       for (let i = 0; i < 3; i++) {
128         const freq = layerFreqMultipliers[i] * 1
129         const amp = layerWeights[i] / wsum
130         v += amp * perlin2(nx * freq, ny * freq, perm)
131       }
132
133       // Convert from ~[-1,1] to [0,1]
134       out[y * width + x] = clamp01(v * 0.5 + 0.5)
135     }
136   }
137
138   return out;
139 }
```


Do you agree with the resulting ownership? If not, explain why under "Other"

- I agree with the resulting code ownership
- Other: *(open ended)*

3) Git diff of the user changes (current) against the AI-generated code (original)

```
111  102  export function perlinNoise3Layers2D(
112  121    for (let y = 0; y < height; y++) {
113  122      const ny = y * baseFrequency;
114  123      for (let x = 0; x < width; x++) {
115  124        const nx = x * baseFrequency;
116       -
117       -        let v = 0;
     125+        let z = 0;
118  126        for (let i = 0; i < 3; i++) {
119  127          const freq = layerFreqMultipliers[i] * 1;
120  128          const amp = layerWeights[i] / wsum;
121       -          v += amp * perlin2(nx * freq, ny * freq, perm);
     129+          z += amp * perlin2(nx * freq, ny * freq, perm);
122  130        }
123  131
124       -      // Convert from ~[-1,1] to [0,1]
125       -      out[y * width + x] = clamp01(v * 0.5 + 0.5);
     132+      out[y * width + x] = clamp01(z * 0.5 + 0.5);
126  133      }
127  134    }
128  135
129  136    return out;
130  137  }
```

This is the resulting blame after the user changes mentioned above.

```
120    // Sample in a consistent coordinate space.
121    for (let y = 0; y < height; y++) {
122      const ny = y * baseFrequency;
123      for (let x = 0; x < width; x++) {
124        const nx = x * baseFrequency;
125        let z = 0;
126        for (let i = 0; i < 3; i++) {
127          const freq = layerFreqMultipliers[i] * 1;
128          const amp = layerWeights[i] / wsum;
129          z += amp * perlin2(nx * freq, ny * freq, perm);
130        }
131
132        out[y * width + x] = clamp01(z * 0.5 + 0.5);
133      }
134    }
135
136    return out;
137  }
```

PROMPT DETAILS

New session – 2026-05-18T13:48:19.816Z All Tasklets

opencode 2cee610a 139 lines

MESSAGES

BUILD · PROMPT

> Make a simple noise generator that generates perlin noise with 3 different layers. JS is fine

BUILD · RESPONSE · AZURE/GPT-5.4-NANO

I'll first inspect the repo to see what frontend/runtime setup you already have, then I'll add a small self-contained JS noise generator (Perlin) with 3 layers and an example canvas render.

Repo search seems to have returned nothing, so I'm going to re-run the lookup with the workspace root as the search path to confirm what files are present before adding new ones.

Repo is essentially empty (only `.gitignore` and an `.opencode` folder), so I'll add a self-contained `noise/perlin3layers.js` module plus a tiny `demo.html` that renders the noise to a canvas.

I've added a self-contained Perlin implementation (`perlin3layers.js`) plus a `demo.html` you can open to see the 3-layer noise rendered on a canvas. Next I'll quickly sanity-check syntax by running a minimal JS parse check and ensure

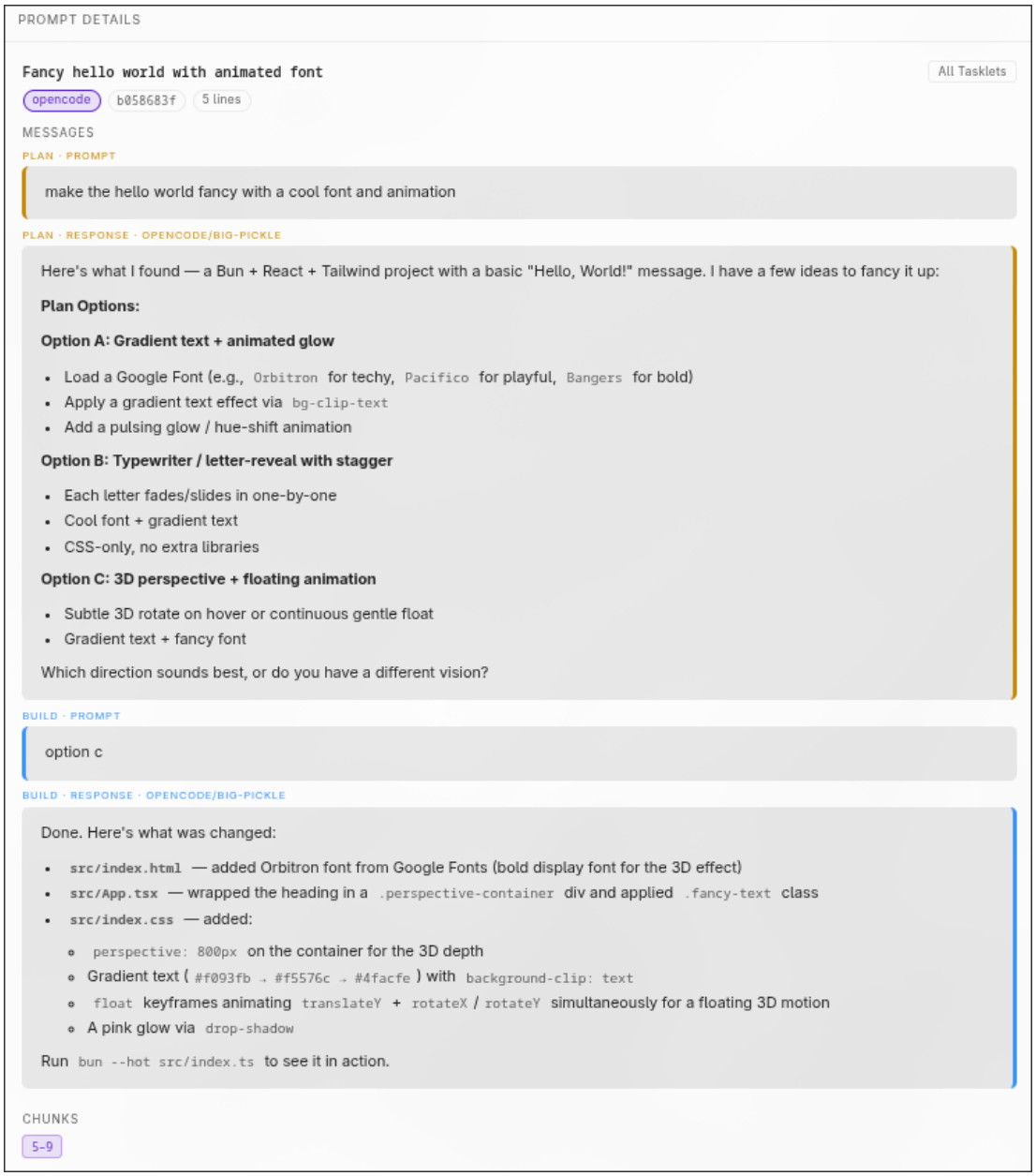
Do you agree with the resulting ownership? If not, explain why under "Other"

- I agree with the resulting code ownership
- Other: *(open ended)*

F. UI & UX

1) Have you tried using Tracybot?
   - Yes
   - No
2) Would you consider the UI intuitive? *(Likert 1 — 6)*
3) If you would like to elaborate on your answer above, please provide feedback here. *(long answer)*
4) Would you agree that grouping multiple relevant prompts into a Tasklet is a clear method to represent the developer's intent while prompting? *(Likert 1 — 6)*
   - What is a Tasklet? Every time you make a change through OpenCode, Tracybot packages a set of relevant prompts into a Tasklet. A Tasklet will contain a Build mode prompt alongside the preceding Plan mode prompts, if present.

   An example Tasklet with multiple prompts is found below.



5) If you would like to elaborate on your answer above, please provide feedback here. *(long answer)*

G. Additional Questions and Suggestions

1) Do you use git notes in your workflow?
   - Yes
   - No
2) What would hinder you from using the tool? In what scenarios you wouldn't want to use the tool? *(long answer)*
3) Would you consider this kind of tracing intrusive on your privacy?
   For example, if a use of this tool were to be mandatory in a work or study environment, and your prompts would be viewed by a manager/teacher.
   If you believe this is intrusive, explain why. *(long answer)*
4) The current version stores all Tasklets in the git history. Do you believe that Tasklets that no longer map to any code should still be stored? If no, then at which point should they be dropped? *(long answer)*
5) Was there anything that you expected to find that was missing? *(long answer)*

APPENDIX C
USER GUIDE

This section describes the setup and usage of *Tracybot* from the User's perspective, which can also be found in the project repository[8].

A. Install the VS Code Extension

This is the entry point to *Tracybot*. The extension can open AI Blame, prompt to initialize *Tracybot* in the current repository, and offer to install the *OpenCode* plugin.

You can install the extension directly within VS Code:

i. Open VS Code and go to the **Extensions** view (`Ctrl+Shift+X` or `Cmd+Shift+X`).
ii. Search for `Tracybot`.
iii. Click **Install**.

Alternatively, visit the Tracybot VS Code Marketplace page [9] and follow the installation instructions on the marketplace page.

B. Open a Repository in VS Code

When the extension activates, it adds an `AI Blame` status bar item on the bottom-right side of *VS Code* (see Figure 5).

If Tracybot has not been initialized in the open repository yet, the extension offers to run initialization for you.

C. Install the OpenCode Plugin (Optional)

The *OpenCode* plugin is required to trace changes made with *OpenCode*. If the *VS Code* extension is installed, it will prompt you to install the *OpenCode* plugin when it is missing. Alternatively, it can be installed from the terminal:

- Windows:

```
irm https://raw.githubusercontent.com/
    TracyTeam/tracybot/main/opencode-
    plugin install.ps1 | iex
```

- Linux & macOS:

```
curl -fsSL https://raw.
    githubusercontent.com/TracyTeam/
    tracybot/main/opencode-plugin/
    install.sh | bash
```

D. Start Using AI Blame

After the repository is initialised, has AI changes (either created by *OpenCode* or pre-existing), and opened in *VS Code* the AI changes can be viewed in the `AI Blame` mode.

APPENDIX D
DEVELOPER GUIDE

This section provides instructions for development work on *Tracybot*.

Tracybot source code is available at https://github.com/TracyTeam/tracybot.

The repository is a mono-repo containing all the components of *Tracybot* in separate directories:

- `opencode-plugin` contains the source code for the *OpenCode* integration
- `tracking` contains the *Git* hook scripts implementing the tracing logic
- `vscode-extension` contains the source code for the *VS Code* integration

A. Initialize a Repository Manually

To initialize a repository from the terminal, use `python ./init.py /path/to/target-repository`, or from within the target repository: `python /path/to/tracybot/init.py`.

B. Work on the VS Code Extension

The *VS Code* extension component of *Tracybot* is maintained using a set of Node.js-based development scripts that automate common workflows:

- `npm run clean`: removes all build artifacts and generated assets to ensure a clean build state.
- `npm run copy-assets`: copies required runtime assets (e.g., `init.py` and `tracking`) into the build output.
- `npm run compile`: compiles the TypeScript source code into JavaScript using the project configuration.
- `npm run package`: builds a distributable *VS Code* extension package (`.vsix`).
- `npm run deploy`: packages the extension and installs it into the local VS Code environment.

To be able to compile the extension, the dependencies must first be installed using `npm install`.

If working from *VS Code*, the extension can be built using `npm run copy-assets && npm run compile`, then started by pressing `F5` in the editor. Alternatively, `npm run deploy` builds and installs the extension into *VS Code*.

C. Work on the OpenCode Plugin

The *OpenCode* plugin component is maintained using a set of Bun development scripts.

To be able to compile the extension, the dependencies must first be installed using `bun install`.

To build and install the extension into *OpenCode*, use `bun run deploy`. Alternatively, use `bun run build` to build the extension without installing.

To inspect the plugin logs from the latest *OpenCode* session, use `bun run logs`.

[8] https://github.com/TracyTeam/tracybot#quick-start

[9] https://marketplace.visualstudio.com/items?itemName=TracyTeam.tracybot-extension