CodeWatcher: IDE Telemetry Data Extraction Tool for Understanding Coding Interactions with LLMs

Manaal Basha[1], Aimeê M. Ribeiro[2], Jeena Javahar[1], Cleidson R. B. de Souza[2]
Gema Rodríguez-Pérez[1], [1]University of British Columbia, Kelowna [2]Federal University of Pará
manaals@student.ubc.ca, aimee@ufpa.br, jeenajavahar@gmail.com,
cleidson.desouza@acm.org, gerope@mail.ubc.ca

***Abstract*—Understanding how developers interact with code generation tools (CGTs) requires detailed, real-time data on programming behavior which is often difficult to collect without disrupting workflow. We present *CodeWatcher*, a lightweight, unobtrusive client-server system designed to capture fine-grained interaction events from within the Visual Studio Code (VS Code) editor. *CodeWatcher* logs semantically meaningful events such as insertions made by CGTs, deletions, copy-paste actions, and focus shifts, enabling continuous monitoring of developer activity without modifying user workflows. The system comprises a VS Code plugin, a Python-based RESTful API, and a MongoDB backend, all containerized for scalability and ease of deployment. By structuring and timestamping each event, *CodeWatcher* enables post-hoc reconstruction of coding sessions and facilitates rich behavioral analyses, including how and when CGTs are used during development. This infrastructure is crucial for supporting research on responsible AI, developer productivity, and the human-centered evaluation of CGTs. Please find the demo, diagrams, and tool here.**

***Index Terms*—Code Generation, Telemetry, VS Code, Plugin**

I. Introduction

Understanding how programmers write code, beyond just a correct final product has been a challenge in both computer science education and in industry [1]. This understanding is essential for improving tool support, enhancing productivity, and identifying usability challenges within IDEs. Traditional methods of evaluating code such as test cases or static analysis typically checks correctness or style after the code is written. These approaches although helpful, offer limited insight into the process of writing code and can struggle to support developers during moments of confusion, experimentation, or problem solving [2].

In educational contexts, novice programmers frequently struggle to identify and correct mistakes as they write code, particularly when using code generation tools (CGTs) [1] such as GitHub Copilot[1] or Amazon Q[2] which generate code within the IDE. Providing timely, formative feedback during the coding process can improve learning outcomes, retention, and engagement in programming [3], [4]. However, large class sizes or online settings often make it infeasible for instructors to deliver personalized feedback at scale. Existing tools such as AutoGrader[3], Coding Rooms[4], and CodeWorkout[5] automate assessments, but typically focus on correctness or syntax, missing deeper behavioural indicators like confusion, excessive rewriting, or blind copy-pasting. While process-aware tools such as OverCode [5] attempt to visualize student code patterns, they often lack real-time monitoring and rely on batch analysis.

In industry, software engineering teams also benefit from understanding how developers code in relation to tasks such as during onboarding [6], debugging [7], and code reviews [8]. Prior research has shown that developer telemetry, which is data about interactions with editors, compilers, and version control systems, can identify productivity bottlenecks and support developer success [9], [10], [11], [12]. However, most industrial tooling is proprietary, emphasizes aggregate metrics, and lacks detailed or customizable feedback suitable for educational use or behavioral analysis. Furthermore, current CGTs are based on LLMs that offer single and multiline suggestions and have been affecting developers' workflow. More studies about these tools rely on surveys, video recordings, and interviews [13], [14], [15], [16], [17] (with two exceptions [11], [18]). These studies primarily focus on users' reactions rather than capturing real-time CGT interaction data within the IDE, an area where *CodeWatcher* offers a unique contribution.

CodeWatcher addresses the lack of detailed and scalable interaction telemetry data for CGTs used in the IDE. *CodeWatcher* is a client-server system that captures detailed programming events (e.g., insertions, deletions, copy/paste, focus shifts), is compatible across multiple CGTs and programming languages, and can support real-time feedback through a rule-based engine. Instructors or team leads can define custom rules that trigger alerts or suggestions based on coding behavior data. Unlike traditional tools that focus solely on code correctness, *CodeWatcher* highlights how the code was written with CGTs, offering process-level visibility like frequent rework, use of LLMs, or idle time.

II. Related Work

That development of tools that capture and analyze user interactions within IDEs is crucial for understanding programmer behaviour and improving software development pro-

[1]https://github.com/features/copilot
[2]https://aws.amazon.com/q/
[3]https://autograder.io/
[4]https://www.codingrooms.com/
[5]https://codeworkout.cs.vt.edu/

cesses. Existing research highlights the need for tools that can track various aspects of developer activity, including code changes, IDE commands, and cognitive processes. For instance, TaskTracker, a plugin for IntelliJ-based IDEs, collects code snapshots and user actions during programming tasks [19]. Similarly, DFLOW records interaction data, enabling analysis of time spent on different activities like editing and navigating code. These tools demonstrate the value of tracking IDE interactions for understanding developer behavior. However, they often lack the ability to simultaneously capture other forms of behavioral data, such as eye-tracking data, which can provide insights into cognitive processes [20].

To address this limitation, CodeGRITS was developed as a plugin for JetBrains IDEs to track both IDE interactions and eye gaze data [18]. This tool allows researchers to combine these complementary approaches for a more comprehensive understanding of developer behavior. Furthermore, the emergence of LLMs in software development necessitates tools that can track the use and impact of AI-generated code. For instance, WaitGPT transforms AI-generated code into an interactive visual representation, enabling users to monitor and steer data analysis performed by LLMs [21].

Techniques to assist with distinguishing between human-written and CGT code across multiple coding languages with high accuracy such as through AI code stylometry classifiers [22] could help to better understand user interactions in CGTs. However, this method can be tedious as it requires having and utilizing a relevant dataset to train a transformer based encoder classifier for such a task.

While these tools or methods offer valuable insights into developer behavior and the use of LLMs, a gap remains in tools that can specifically track data or metrics related to AI code suggestions, acceptance rates, and code reuse patterns. The *CodeWatcher* tool aims to fill this gap by providing researchers with the ability to extract and analyze data related to AI-generated code within the IDE, enabling a deeper understanding of the impact of AI on software development practices.

III. Event Capture and Representation

A core capability of *CodeWatcher* lies in its ability to unobtrusively capture detailed programmer activity by logging semantically meaningful events from the developer's IDE. It specifically logs user interactions with CGTs and not general coding behavior. The system is designed to balance fidelity with minimal intrusiveness, enabling continuous, real-time data extraction of code authoring behaviors without requiring changes to user workflows.

The *CodeWatcher* client plugin monitors eight distinct event types that reflect key aspects of interaction during a coding session: **Start**, **End**, **Insertion**, **Deletion**, **Focus**, **Unfocus**, **Copy**, and **Paste**. Each event is recorded as a structured object containing up to four fields:

- **Type** – the event category (e.g., *Insertion*, *Focus*);
- **Time** – a timestamp (ms precision) marking the event occurrence;
- **Text** – the relevant code snippet or user input (when applicable); and
- **Line** – the full line of code where the event occurred (for insertions and deletions).

Not all fields are applicable to every event type. Table I summarizes the semantics of each event and the associated fields.

TABLE I
Event Types and Captured Fields in CodeWatcher

Event Type	Description	Captured Fields
Start	Triggered when a new file is opened and becomes active in the editor.	**Time** - records the moment the document is opened and editing begins
End	Triggered when the file is removed from focus.	**Time** - captures the moment the editing session ends.
Insertion	Logged when code is pasted from outside the IDE (including IDE chats) or code suggestion acceptances that are more than 3 characters, reducing noise from keystroke input.	**Time** - captures the time when the increase in text is detected, **Text** - the newly inserted text, **Line** - the entire line of code where the insertion occurred
Deletion	Logged when any content is removed, whether single-character or multi-line.	**Time** - the time when the text is deleted, **Text** - the text of the removed content, **Line** - the line where the deletion took place
Unfocus	Recorded when the user switches away from the IDE.	**Time** - records the time when the IDE loses focus
Focus	Recorded when the IDE regains focus from another application.	**Time** - records the time when the IDE regains focus
Copy	Detected upon executing a copy command (e.g., `CTRL+C`) within the IDE.	**Time** - the time when the copy command is executed, **Text** - the copied content
Paste	Triggered on paste command (e.g., `CTRL+V`); the subsequent *Insertion* captures the actual content.	**Time** - the time when the paste command is executed **Text** - the newly pasted text

The plugin is optimized for minimal overhead, relying on native editor hooks to track text manipulation and focus changes. To reduce noise, it filters out character-level inputs and aggregates small edits into higher-level *Insertion* and *Deletion* events. Importantly, the system disambiguates between typed input, specifically collecting code generation acceptance data and pasted content. By capturing this rich sequence of temporally ordered, context-aware events, *CodeWatcher* enables both real-time intervention and post-hoc reconstruction of development sessions involving CGTs. This data forms the basis for CGT usage analysis, behavioral modeling, and personalized feedback mechanisms.

IV. Tool Architecture

CodeWatcher is implemented as a modular, client-server system composed of three primary components: a VS Code plugin (client), a RESTful API (server), and a backend database. This architecture supports ease of maintenance, deployment scalability, and flexible integration with workflows.

A. Client-Server Communication

The front-end component of *CodeWatcher* is a VS Code plugin written in JavaScript. Operating locally within the developer's environment, the plugin is responsible for unobtrusively capturing interaction events within the editor such as code insertions, deletions, and focus changes. This is then transmitted to the backend API over HTTP when there is an *end* event type.

Communication between the plugin and the server adheres to RESTful principles. Event data is encoded in JSON format and sent via HTTP requests to predefined API endpoints. The API, implemented in Python using the FastAPI framework, is designed around the service-repository architectural patterns to promote separation of concerns.

The API exposes a variety of endpoints that support key operations including user account creation, login, interaction log registration, user deletion, and permission management. Input validation is handled via Pydantic models, ensuring data consistency and integrity prior to storage. All communication is routed through a remote server running Ubuntu, where the API and the database are deployed using Docker containers.

B. Database Schema

The backend database is implemented using MongoDB, selected for its schema-less structure and flexibility in handling diverse interaction log formats. Two primary collections are part of the storage layer: `user`, and `interaction_logs`. The `user` collection stores all user credentials, metadata, and access-level permissions. Whereas `interaction_logs` contain structured records of all IDE events, each annotated with metadata like those seen in Table I.

Event documents are stored as JSON-like objects, enabling dynamic schema evolution as the set of tracked events expands. MongoDB Compass is used for monitoring and querying the database during development and analysis of phases.

V. Validation and Testing

To validate *CodeWatcher*, we confirmed the accuracy and completeness of the data it collects and transmits to the server. This includes ensuring the tool captures the correct event types, attributes, and timestamps during development activity.

A. Validation Methodology and Results

We validated *CodeWatcher*'s ability to accurately and reliably log user interactions by designing a set of controlled test scenarios (Table II) aligned with the following key objectives:

- **Correct Event Detection:** Each user interaction (e.g., insertions, deletions, copy-paste, file start/end) is correctly identified and logged.
- **Field-Level Accuracy:** Captured metadata (e.g., time, text, line content) precisely reflects the editor state.
- **Transmission Integrity:** Ensures events are transmitted to the server without data loss or corruption.
- **System Robustness:** Operates reliably under realistic and edge-case IDE usage patterns.

TABLE II
Validation Metrics Across Scenarios Summary

Scenario	Event Type(s)	Precision	Recall
Insertion Accuracy	Insertion	1.00	1.00
Deletion Detection	Deletion	1.00	1.00
Copy-Paste Behavior	Copy, Paste		
Focus Switching	Focus, Unfocus	0.87	1.00
Back-and-Forth Use Patterns	Insertion, Deletion	1.00	1.00
Multiple File Sessions	Start, Insertion (across tabs/file), end	1.00	1.00
Event Logging Under Load (1 user)	All events	0.93	1.00
Concurrent Users	All events (stress test)	1.00	1.00

To verify these goals, we ran 150 trials with 2 users simulating realistic developer workflows. For each event type, we triggered interactions manually and checked server-side logs for accuracy. Our validation process used:

- **Exact Match Verification:** For event types involving code changes (e.g., Insertion, Deletion, Paste), we confirmed that the logged `Text` and `Line` fields exactly matched the actual editor content at the time of the event.
- **Timestamp Validation:** We ensured that all logged events reflected the correct temporal order by comparing timestamps to externally recorded wall-clock timings (e.g., from screen recordings).

Where applicable, we also validated correct identification of event types (e.g., Copy, Focus, Start) and ensured they appeared in the correct sequence. Table II summarizes the precision and recall metrics obtained during these tests. These results demonstrate *CodeWatcher*'s robustness in capturing fine-grained user behaviors across a variety of realistic usage patterns.

B. Robustness Under Diverse Use Cases

We also validated behavior under more realistic usage scenarios such as:

- **Rapid Back-and-Forth Use Patterns:** Simulated high-speed editing and undo-redo operations.
- **Multi-line Suggestion Acceptance:** Accepted multiple lines from code suggestions to validate correct grouping and event generation (part of insertion validation).
- **Multiple File Sessions:** Utilized multiple files in parallel to test session consistency and file-scoped logging.
- **Concurrent Users:** Ran parallel sessions to ensure isolated logging and user separation with users.

C. Summary of Findings

Across all scenarios, on average, *CodeWatcher* accurately detected and logged user interactions in accordance with the specification in Table I. Captured data was successfully transmitted to the server, and timestamps and content fields aligned with observed behavior in the IDE. This validation confirms that *CodeWatcher* reliably captures raw behavioral

data necessary for downstream analyses, without relying on assumptions about specific analytics workflows or post-processing interpretations.

VI. Empirical Use Case Example: Identifying AI Generated Code

Algorithm 1 Algorithm for Categorizing Coding Interactions

Input: *CodeWatcher* logs (JSON), Final submitted code (text file)
Output: Line-by-line labels: *AI-Generated*, *AI-Modified*, or *User-Written*

Step 1: Load and Preprocess Data
Load the JSON file and extract historical code lines. Read the final code file and split it into individual lines, normalize, and strip whitespace.

Step 2: Compare Lines and Compute Similarity
foreach *line* L_f *in final code* **do**
  foreach *line* L_h *in historical data* **do**
    Compute fuzzy similarity score $S(L_f, L_h)$
  Let L_{best} be the historical line with highest score
  Step 3: Label Line Based on Similarity Score
  if $S(L_f, L_{best}) \geq 95$ **then**
    Label L_f as *AI-Generated*
  else
    if $80 \leq S(L_f, L_{best}) < 95$ **then**
      Label L_f as *AI-Modified*
    else
      Label L_f as *User-Written*

Step 4: Save Output
Store labeled lines into a structured JSON output file

The *CodeWatcher* tool can enable researchers to analyze the authorship of code submissions by linking keystroke-level interaction data with the final program output. To illustrate this, we implemented a post-processing script to be run in the server side, that determines whether each line of code in a final code submission was generated by a CGT, CGT generated but modified by the user, or entirely written by the user. This classification process begins with *CodeWatcher*'s JSON log files, which record the code inserted from the CGT during the programming session. The final submitted code file is then compared to the historical logs (i.e., data from *CodeWatcher* logs) using a fuzzy string matching algorithm (described in 1) that computes similarity scores between corresponding lines. Based on these scores, the script labels lines as *AI-Generated* (score $\geq$ 95), *AI-Modified* (score between 80 and 94), or *User-Written* (score $<$ 80). This labeling allows researchers to quantify the extent of AI assistance, identify instances of user engagement or editing, and evaluate how students or developers interact with generative coding tools. The output can be further validated using precision, and recall, and F1-score metrics against manually annotated ground truth data. The methodology is reproducible, and adaptable across diverse datasets of coding activity.

The results on three developers completing easy tasks show on average, 68% of the code written by participants was directly generated by an AI tool, 7% was AI-generated but modified by the user, and 25% was entirely user-written. The classification algorithm demonstrated high precision (77%) but moderate recall (41.5%). These results suggest that our use of *CodeWatcher* data provides a promising foundation for reliably distinguishing between AI- and user-written code, though further improvements are needed to enhance sensitivity.

VII. Use Cases

This section presents a set of use cases in which we envision *CodeWatcher* being used.

Educational Feedback and Assessment: In academic settings instructors can use the data from *CodeWatcher* to understand how students approach assignments beyond grading final code submissions. *CodeWatcher* data can help identify when students overly on AI-generated completions or copy-paste code without meaningful engagement, supporting a better assessment of student effort and understanding. It can also inform curriculum refinement by revealing recurring pain points across peers.

Research on Programming Behaviour: *CodeWatcher* offers researchers a robust platform for studying how novice and expert programmers interact with code, IDEs, and CGTs. It enables the collection of rich temporal and behavioral data, allowing for analyses of strategies such as incremental development, test-driven coding, or (over-)reliance on AI completions. Researchers can conduct controlled studies to compare coding behaviors across tools, environments, or demographics (e.g., native vs. non-native English speakers, gender groups) to assess how these factors influence learning and performance. Such data is critical for understanding equity in CS education and the role of AI in shaping different learners' experiences.

CGT Evaluation: *CodeWatcher* can be used to assess the utility, limitations, and impacts of CGTs on user interactions and outcomes. By logging all interactions with these tools such as when completions are accepted or edited, researchers can better explore where CGTs are helpful, confusing, or misused. These insights can inform refinements to AI suggestion engines, particularly in how suggestions are presented or triggered. *CodeWatcher* data can also be used for retrospective attribution of which code segments were likely AI-generated versus hand-written, helping to disentangle human and machine contributions in collaborative development.

Industry Applications: Development teams can use *CodeWatcher* data to evaluate productivity based on interaction richness, and task-switching patterns. This can help identify blockers, improve tooling, or streamline workflows. For onboarding, *CodeWatcher* allows teams to monitor how new developers learn internal systems and identify areas where additional support is needed. Engineering leads can also use *CodeWatcher* to evaluate the adoption and real-world utility of new IDE features or plugins, identifying friction points or unused capabilities. In regulated industries or high-stakes collaborations, *CodeWatcher* logs may serve as provenance data for audit trails, particularly when code authorship must be clarified.

Ethics, Fairness, and AI Accountability: *CodeWatcher* contributes to responsible AI practices by enhancing transparency and traceability in code authored with CGTs. As AI-generated code becomes increasingly prevalent in both educational and industrial contexts, distinguishing between

human and machine contributions is crucial for fair grading, authorship attribution, and intellectual property claims. It can also help identify potential biases in CGTs by revealing disparities in suggestion acceptance or effectiveness across user groups. *CodeWatcher* empowers educators, researchers, and organizations to promote fairness, accountability, and informed decision-making in AI-assisted development.

IDE Personalization and Smart Interventions: Finally, *CodeWatcher* lays the groundwork for intelligent, personalized IDEs. Real-time data on user interaction can be used to trigger context-sensitive interventions—such as hints, documentation links, or debugging suggestions such as when a user appears to be struggling. Over time, this data can also inform IDE adaptations that suggest shortcuts, identify common errors, or recommend tools tailored to individual developer workflows. By supporting these smart, adaptive features, *CodeWatcher* can transform the static IDE into a more responsive and supportive learning and development environment.

VIII. Limitations

CodeWatcher may not work as expected with different IDEs or CGTs, and does require a server for the user to connect to the tool so that the data is uploaded in real-time. However, the tool can be setup to run on a local server or locally without a server upload. Web-inserted code logs as unfocus and then insertion not paste, hinting it's not AI, but remains error-prone. The tool intentionally does not capture text that is not part of the coding language of the file. For example if a user inserts text that is not considered code, or a comment, the tool will not capture this as to avoid irrelevant data. This can make the plugin data less useful for those interested in such data.

IX. Conclusion

We presented *CodeWatcher*, a lightweight plugin that captures coding interactions in the presence of CGTs. Our empirical use case shows how this data supports classification of AI-generated, AI-modified, and user-written code. *CodeWatcher* offers a practical foundation for studying CGT usage and developer behavior with minimal overhead.

Acknowledgments

We thank GOOGLECA (PWPU GR028067) and CNPq (420406/2023-9 and 442779/2023-2) for funding this research.

References

[1] S. Barke, M. B. James, and N. Polikarpova, "Grounded copilot: How programmers interact with code-generating models," *Proceedings of the ACM on Programming Languages*, vol. 7, pp. 85–111, 2023.

[2] S. H. Edwards, "Using test-driven development in the classroom," in *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, 2004.

[3] R. Shen, D. Y. Wohn, and M. J. Lee, "Comparison of learning programming between interactive computer tutors and human teachers," in *Proceedings of the ACM Conference on Global Computing Education*, ser. CompEd '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 2–8. [Online]. Available: https://doi.org/10.1145/3300115.3309506

[4] A. K. Veerasamy, M.-J. Laakso, and D. D'Souza, "Formative assessment tasks as indicators of student engagement for predicting at-risk students in programming courses," *Informatics in education*, vol. 21, no. 2, pp. 375–393, 2022.

[5] E. L. Glassman, J. Scott, R. Singh, P. J. Guo, and R. C. Miller, "Overcode: Visualizing variation in student solutions to programming problems at scale," *ACM Trans. Comput.-Hum. Interact.*, vol. 22, no. 2, Mar. 2015. [Online]. Available: https://doi.org/10.1145/2699751

[6] A. Ju, H. Sajnani, S. Kelly, and K. Herzig, "A case study of onboarding in software teams: Tasks and strategies," in *43rd International Conference on Software Engineering*. IEEE, 2021, pp. 613–623.

[7] T. Hirsch and B. Hofer, "What we can learn from how programmers debug their code," in *2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*. IEEE, 2021, pp. 37–40.

[8] A. K. Turzo and A. Bosu, "What makes a code review useful to opendev developers? an empirical investigation," *Empirical Software Engineering*, vol. 29, no. 1, p. 6, 2024.

[9] M. Beller, A. Park, K. Nakad, A. Patel, S. Mohanty, F. Garberson, I. G. Malone, V. Garg, H. Verroken, A. Kennedy *et al.*, "What's dat? three case studies of measuring software development productivity at meta with diff authoring time," *arXiv preprint arXiv:2503.10977*, 2025.

[10] K. Z. Cui, M. Demirer, S. Jaffe, L. Musolff, S. Peng, and T. Salz, "The productivity effects of generative ai: Evidence from a field experiment with github copilot," 2024.

[11] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Measuring github copilot's impact on productivity," *Commun. ACM*, vol. 67, no. 3, p. 54–63, Feb. 2024. [Online]. Available: https://doi.org/10.1145/3633453

[12] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann, "Software developers' perceptions of productivity," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 19–29. [Online]. Available: https://doi.org/10.1145/2635868.2635892

[13] S. Barke, M. James, and N. Polikarpova, "Grounded copilot: How programmers interact with code-generating models," *Association for Computing Machinery*, vol. 7, April 2023.

[14] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Conference on human factors in computing systems extended abstracts*, 2022, pp. 1–7.

[15] F. F. Xu, B. Vasilescu, and G. Neubig, "In-ide code generation from natural language: Promise and challenges," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 2, pp. 1–47, 2022.

[16] W. Mendes, S. Souza, and C. De Souza, ""you're on a bicycle with a little motor": Benefits and challenges of using ai code assistants," in *IEEE/ACM 17th International Conference on Cooperative and Human Aspects of Software Engineering*, ser. CHASE '24. New York, NY, USA: ACM, 2024, p. 144–152. [Online]. Available: https://doi.org/10.1145/3641822.3641882

[17] R. Wang, R. Cheng, D. Ford, and T. Zimmermann, "Investigating and designing for trust in ai-powered code generation tools," in *Proceedings of the 2024 ACM Conference on Fairness, Accountability, and Transparency*, ser. FAccT '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1475–1493. [Online]. Available: https://doi.org/10.1145/3630106.3658984

[18] N. Tang, J. An, M. Chen, A. Bansal, Y. Huang, C. McMillan, and T. J.-J. Li, "Codegrits: A research toolkit for developer behavior and eye tracking in ide," in *46th International Conference on Software Engineering Companion (ICSE-Companion '24)*. ACM, 2024.

[19] E. Lyulina, A. Birillo, V. Kovalenko, and T. Bryksin, "Tasktracker-tool: A toolkit for tracking of code snapshots and activity data during solution of programming tasks," in *52nd ACM Technical Symposium on Computer Science Education*, 2021, pp. 495–501.

[20] R. Minelli, A. Mocci, and M. Lanza, "I know what you did last summer - an investigation of how developers spend their time," in *IEEE 23rd International Conference on Program Comprehension*, 2015, pp. 25–35.

[21] L. Xie, C. Zheng, H. Xia, H. Qu, and C. Zhu-Tian, "Waitgpt: Monitoring and steering conversational llm agent in data analysis with on-the-fly code visualization," in *37th Annual ACM Symposium on User Interface Software and Technology*, 2024, pp. 1–14.

[22] A. Gurioli, M. Gabbrielli, and S. Zacchiroli, "Is this you, llm? recognizing ai-written programs with multilingual code stylometry," *arXiv preprint arXiv:2412.14611*, 2024.