Coding Process Visualizations for Improved Learning and Academic Integrity in CS1

Badri Adhikari
Department of Computer Science
University of Missouri-St. Louis
St. Louis, USA
adhikarib@umsl.edu

Nirala Lamichhane
Department of Electronics & Computer Engineering
Tribhuvan University (ACEM)
Kathmandu, Nepal
nirala.075bct034@acem.edu.np

***Abstract*—While novice programmers often find it challenging to learn coding in CS1 courses, teachers also face the daunting task of facilitating an environment that encourages students to become original thinkers and develop meta-thinking skills. A significant reason for these teaching and learning challenges is the focus on outcome rather than process. Merely reviewing and assessing a student's submitted code is often insufficient to provide insightful and personalized feedback to students. This work suggests a potential solution: quantitative summaries and data visualizations of a student's coding process that can serve as powerful tools for highlighting students' coding habits. By presenting process data—such as breaks taken, code execution history, AI usage, copy-paste events, and playback of the code evolution process—in easy-to-understand and interactive visualizations, students can reflect on and learn from their process. An added benefit of this process-focused approach to teaching and learning is the automatic promotion of academic integrity. This paper elaborates on these process visualizations and discusses strategies for their effective implementation in CS1 courses.**

***Index Terms*—programming feedback; AI-assisted coding; self-directed learning; coding fundamentals; academic integrity**

I. Introduction

In computer programming education, the easy accessibility of generative AI tools has added challenges for instructors in helping students master fundamental coding skills. These basic skills are essential for most careers related to coding, regardless of whether AI copilots are utilized. Students learning to code in CS1 courses often encounter frustration when confronted with cryptic error messages from compilers [1]. During independent coding assignments, these cryptic messages can lead to potential disengagement. At these moments of frustration and disengagement, it is common for many students to seek alternative methods to complete their tasks. In response to these challenges, this work proposes shifting the focus from the outcome to the student's coding process. With quantitative summaries and data visualizations that summarize a student's coding process, they can self-learn from their process and utilize the process data to acquire feedback.

Shifting the focus from the outcomes to the underlying process holds promise for guiding students in utilizing AI effectively (if they use it) and empowering instructors to teach coding fundamentals more efficiently. Providing access to the coding process also helps students develop a deeper understanding of fundamental concepts by encouraging them to self-reflect on their work. Ultimately, this can lead to more effective teaching and learning practices. One example of such a process-revealing tool is Process Feedback [2].

The rest of the paper is organized as follows. First, the Process Feedback online compiler is described. Next, several quantitative summaries and data visualizations that can display a student's coding process are explained. Finally, in the discussion section, the proposed idea of focusing on the process is discussed in relation to two recent studies.

II. Process Feedback Online Compiler

Process Feedback (PF) [2], available at processfeedback.org, is an innovative and free online coding (and writing) tool designed to reveal students' process behind their work. The platform encourages students to view their coding process, reflect on it, and self-learn. When students code on the platform, they see their process represented through several summaries and visualizations, including playback of the entire coding journey, breaks taken during coding, total time taken to complete the task, typing fluency, copy-paste events, time spent on each code block, code execution history, and AI usage.

The PF platform also allows students to use AI innovatively. Anytime during coding, students can click on the "Explain Code" button to receive a plain-English explanation of the selected code. Similarly, when students encounter a compiler error message, they can click on the "Explain Error" button to receive an explanation of the error message with hints to fix the code in an easy-to-understand language. This *guard-railed* way of using AI, as shown by other recent studies [3], is known to engage students in learning coding without providing them full solutions.

PF enables students to self-explore and self-learn from their processes while also allowing them to download all data and visuals as a comprehensive PDF report to share with their instructor. Equipped with such a *process report*, instructors can offer personalized feedback to their students, focusing on both the students' outcomes and their underlying processes.

III. Coding Process Visualizations

Using the visualizations and summaries implemented in the Process Feedback platform as a reference, this section

describes various unique data summaries and visualizations. Individually and collectively, these process summaries and visualizations display a student's coding process, providing insights into their coding habits.

A. Playback of the Coding Journey

As shown in Figure **1A**, allowing students to play back their coding journey is an engaging technique that displays the entire coding process like a film. With options to pause, stop, replay, and control the speed of playback, such an interactive feature allows students to observe the steps taken during their coding process and see the evolution of the code's structure. During playback, characters added are shown in green highlights and those deleted as red strike-through text, enabling students to spot changes during playback. Besides Process Feedback, other tools such as CodeProcess [4] also discuss playback as a dynamic view of the coding process.

B. Continuous Streak of Typing and Breaks Taken

Displaying a timeline with segments of continuous active typing and thinking (or inactive) time durations can help analyze how engaged a student was when coding. Such a timeline, as shown in Figure **1B**, can also help students realize how often they were distracted while working on the task. Observing the durations of active typing and thinking (or inactive) periods can help both students and instructors gauge the level of engagement in the task. For instance, if a student completes a challenging coding task in a single continuous active typing duration of 5 minutes, reviewing the process data can make it easy to discern if the student was typing the code from another source or is indeed a proficient programmer.

C. Location of Code Edit at Various Time Points

When coding, students often spend most of their time on a code block that has the core logic for the task at hand. For instance, implementing a function that contains arrays and loops can be challenging and time-consuming for novice programmers and they could spend much of their time in these code blocks. The Edit Location chart can visually reveal such struggles by showing which part of the code a student was actively working on at any given time in their coding journey. As shown in Figure **1C**, it is a stacked bar diagram where the active code block is colored and highlighted. This chart can be particularly helpful to see how often a student *revised* a piece of code after initially typing it.

D. How Often AI was Used (to Explain Code or Error)

Tools like Process Feedback have "Explanatory AI" built into them, allowing students to use features such as "Explain Code" and "Explain Error." How often a student uses such AI features can also be useful data to review. As shown in Figure **1D**, AI usage can be visualized as a lollipop chart with green dots showing the instances of "Explain Code" usage and red dots showing "Explain Error" usage.

E. How Often Code Executions Failed or Passed

A code execution history chart, as shown in Figure **1E**, can visually display how often a student's code execution failed or passed. It is also a timeline chart with points indicating successful and unsuccessful code executions, allowing students to observe the frequency and timing of successful (green dots) and failed (red dots) code executions. In the interactive version of the report, hovering over the dots shows the actual code error or output. While it may be quite common for students to have many failed executions and at least one successful execution at the end, it may be uncommon for students to have no successful executions.

F. Number of Characters Added or Removed Over Time

The number of characters added or removed over time can be shown as a timeline bubble chart, as shown in Figure **1F**, where the sizes of the bubbles represent the number of characters added or deleted at specific time points. Such a chart can help students see how common it is to delete text during the coding process. Additionally, if there are no deletions at all during a long coding process, which is uncommon, it can prompt the instructor to investigate further to understand why the student was typing so fluently without deletions. For instance, such a timeline bubble chart, with no deletions, can often be a sign of unrefined work.

G. Typing Speed and Copy-paste Events

Visualizing the typing speed over time, as shown in Figure **1G**, can reveal periods where a student is fluently typing, typing slowly (possibly thoughtfully), or copy-pasting. While records of copy-pasting may not be very useful for the students themselves, they can be informative for their instructors to discuss (if needed) why it was necessary to paste code from outside. Such a chart can also help students establish that they did not simply copy-paste from elsewhere to complete their work.

H. Comparing the Code at a Time Point with Any Other

Implementing a dual-editor view, with left and right editors, allows students to compare their code at any given time point with another version, helping to identify specific changes of interest (see Figure **1H**). This feature can be particularly beneficial when combined with the visualizations discussed earlier. For instance, clicking on a typing speed dot indicating copy-paste actions can instantly scroll to the comparator, displaying precisely what was pasted.

I. The Final Output of the Code

A common practice in coding assignments is to have students submit screenshots of the output of their code. The final output of the code (or the most recent code output) can be shown along with other process visuals for completeness. While this makes grading easier for instructors, it also helps students ensure that they run and test the code before submitting it.

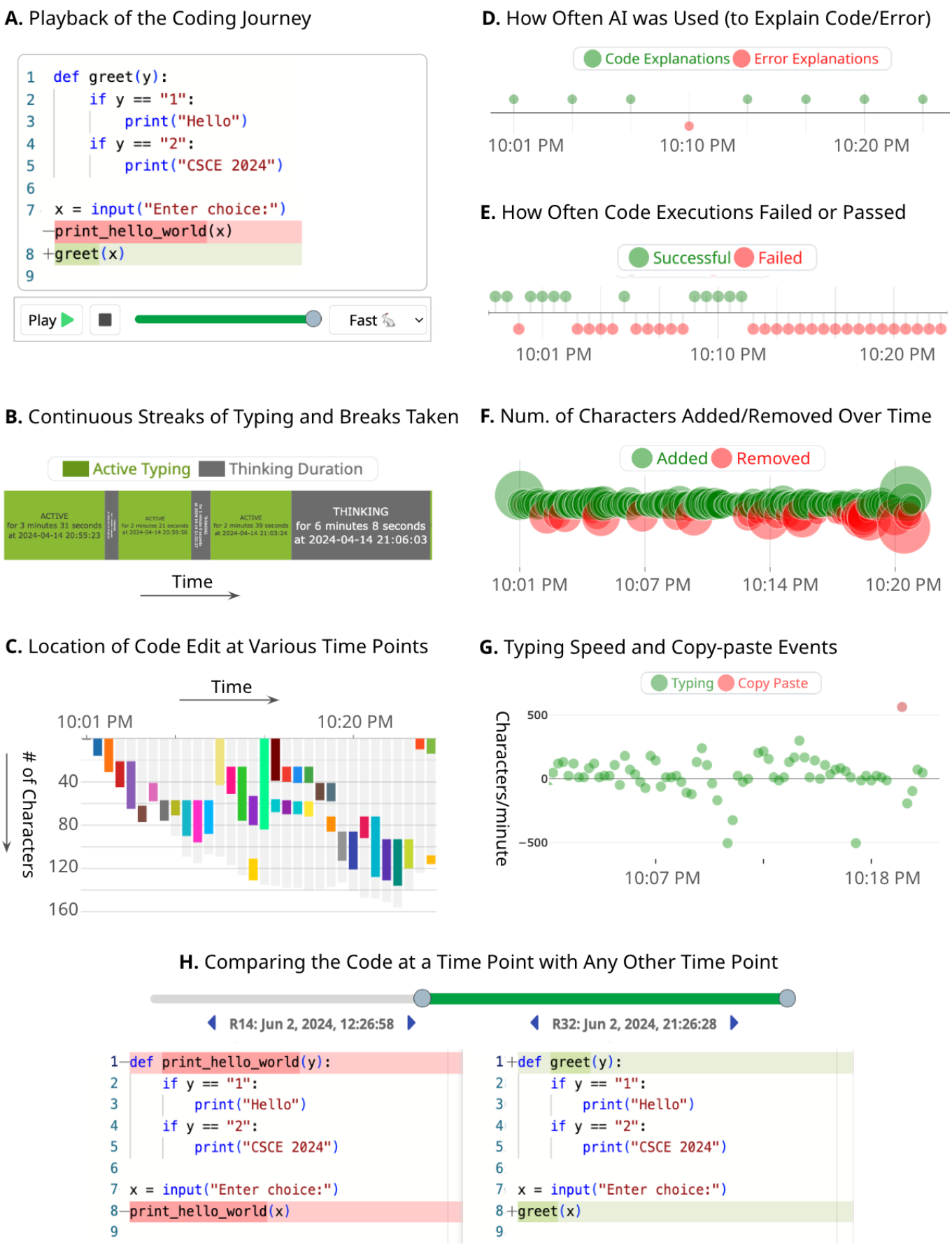

Fig. 1. Data visualizations for summarizing and displaying the coding process of a student.

J. The Most Frequently Observed Error

Students often spend a lot of their time fixing similar kinds of bugs, unaware that they are doing so. Along with other process summaries and visuals, showing the most frequent error message that the student received during the coding process can help the student understand what bugs they spend most of their time fixing. Looking at the most frequent errors of all students, instructors can also find the top errors that most students in the class faced.

K. Coding Time Summary

As a summary of the coding process, several key time-related data points can be shown. These may include the total time taken to finish a coding task (including the breaks taken) and the total active typing time. For instructors, in particular, this single piece of data—the time taken to complete the task—can be a key to determining if the student's work needs appreciation or further attention. For example, if a student takes more than an hour to complete an easy programming task, it can be revealing to explore further and find out what took so long.

IV. Discussion

This section relates the proposed approach of using visualizations to capture a student's coding process with two recent studies that capture a student's process.

In one study involving a CS1 course, the authors required participating students to record their keystroke data (coding process information) and submit the keystroke data along with their assignments [5]. Students were asked to install an additional plugin that automatically recorded every keystroke they typed while coding. The results of the experiment showed that requiring students to submit the process data deterred them from plagiarizing. However, a significant percentage of students surveyed expressed anxiety about plagiarism detection through keystroke data. One student, in particular, mentioned significant stress and anxiety throughout the semester due to concerns about being accused of plagiarism. This anxiety is understandable, as false positives in plagiarism detection can lead to unwarranted accusations.

Although the results of the plagiarism deterrence study were not encouraging from the perspective of student acceptance, given the research design this was completely expected. When an instructor requires students to install an additional plugin as a "recorder" where only the instructor analyzes the recorded data, it is natural for some students to feel anxious. To the students installing the plugin, the only purpose of the plugin is so they can be monitored.

A better way of introducing such process-recording tools is to respect student privacy as much as possible, *let the student be in control of the process data*, and use the process data primarily to enhance learning and not to deter plagiarism. If students are engaged in learning and *are required to reflect on their process*, plagiarism can automatically be less of a concern. In a recent work where students using such process tools were surveyed, students expressed that if such tools are introduced as learning enhancers that can help improve their grades, they are more receptive to these process-revealing tools [6].

Another related study focused on capturing and visualizing a student's writing process. Having 65 high-school students write in InputLog, a process recording tool similar to Process Feedback, the authors explored how showing students personalized process reports can affect their writing skills compared to the national baseline [7]. The study concluded that the implementation of the process feedback approach over just one week resulted in a notable improvement in the text quality of the participants. This improvement was on par with what could typically be achieved with one full year of regular schooling.

The writing study highlights the significant effectiveness of process-oriented tools and the impact of process feedback. The notable improvements in text quality observed among writing students suggest a potential for similar improvements in code quality for students learning to code. Introducing this practice early could revolutionize coding education by helping students identify habits and address their key pain points.

In today's context, where AI presents both challenges and opportunities in education, tools like Process Feedback can play a pivotal role in enriching students' coding skills by empowering instructors to assess student work more effectively and promoting academic integrity. By providing students with insights into their coding processes and encouraging self-reflection through process summaries and visualizations, these tools can significantly enhance learning outcomes.

V. Conclusion

This paper discussed how a CS1 student's coding process can be effectively summarized using various data visualization techniques. The insights derived from these visualizations can help students enhance their learning by reflecting on the challenges faced during the coding process. Additionally, students can share their process reports with instructors to receive targeted feedback. With these process summaries and visuals, instructors can quickly and accurately assess students' work. Overall, such a process-focused approach promotes transparency and academic integrity in coding education.

References

[1] James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani Peters, John Homer, Nevan Simone, and Maxine Cohen. On novices' interaction with compiler error messages: A human factors approach. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, pages 74–82, 2017.

[2] Badri Adhikari. Thinking beyond chatbots' threat to education: Visualizations to elucidate the writing or coding process. *Education Sciences*, 13(9):922, 2023.

[3] Ha Nguyen and Vicki Allan. Using gpt-4 to provide tiered, formative code feedback. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, pages 958–964, 2024.

[4] Raj Shrestha, Juho Leinonen, Arto Hellas, Petri Ihantola, and John Edwards. Codeprocess charts: Visualizing the process of writing code. In *Proceedings of the 24th Australasian Computing Education Conference*, pages 46–55, 2022.

[5] Kaden Hart, Chad Mano, and John Edwards. Plagiarism deterrence in cs1 through keystroke data. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 493–499, 2023.

[6] Kate Arendes, Shea Kerkhoff, and Badri Adhikari. Engaging students to learn coding in the ai era with emphasis on the process. *Edukasiana: Jurnal Inovasi Pendidikan*, 3(2):257–268, 2024.

[7] Nina Vandermeulen, Elke Van Steendam, Sven De Maeyer, and Gert Rijlaarsdam. Writing process feedback based on keystroke logging and comparison with exemplars: Effects on the quality and process of synthesis texts. *Written Communication*, 40(1):90–144, 2023.