SPARK: Real-Time Monitoring of Multi-Faceted Programming Exercises

Yinuo Yang*
College of Engineering
University of Notre Dame
Notre Dame, IN USA
yinooyang@nd.edu

Ashley Ge Zhang
School of Information
University of Michigan
Ann Arbor, MI USA
gezh@umich.edu

Steve Oney
School of Information
University of Michigan
Ann Arbor, MI USA
soney@umich.edu

April Yi Wang
Department of Computer Science
ETH Zurich
Zurich, Switzerland
april.wang@inf.ethz.ch

***Abstract*—Monitoring in-class programming exercises can help instructors identify struggling students and common challenges. However, understanding students' progress can be prohibitively difficult, particularly for multi-faceted problems that include multiple steps with complex interdependencies, have no predictable completion order, or involve evaluation criteria that are difficult to summarize across many students (e.g., exercises building interactive web-based user interfaces). We introduce SPARK, a coding exercise monitoring dashboard designed to address these challenges. SPARK allows instructors to flexibly group substeps into checkpoints based on exercise requirements, suggests automated tests for these checkpoints, and generates visualizations to track progress across steps. SPARK also allows instructors to inspect intermediate outputs, providing deeper insights into solution variations. We also construct a dataset of 40-minute keystroke coding data from N=22 learners solving two web programming exercises and provide empirical insights into the perceived usefulness of SPARK through a within-subjects evaluation with 16 programming instructors.**

***Index Terms*—programming education**

I. INTRODUCTION

Programming instructors often use *in-class exercises*—short hands-on coding tasks conducted during class time—to actively engage students and reinforce the concepts being taught [1]–[6]. However, ensuring students gain meaningful learning outcomes from these exercises is not easy, given the variability in coding abilities, paces, and problem solving approaches [2]. This variation can make it challenging for instructors to provide timely and personalized feedback. Without such assistance, students may struggle to develop essential metacognitive skills, such as formulating effective problem solving strategies, tracking their progress, and assessing whether goals have been met [2], [7]–[9]. This can lead to frustration and a potential loss of confidence in their abilities [10]. Therefore, it is essential for instructors to effectively monitor students' progress and promptly recognize the difficulties they encounter.

However, successfully monitoring students can be challenging, particularly for problems that are *multi-faceted*. We refer to "multi-faceted" problems as those involving non-sequential workflow paths with nested substeps—where some steps are interdependent, others independent, and where uniform evaluation criteria cannot be easily applied across many students. For example, in a web programming exercise, students may need to: (1) create the HTML layout, (2) add CSS for styling, and (3) implement JavaScript for interactivity. Adding CSS may involve edits to both the CSS and HTML files and depends on the layout being complete, but is independent of the JavaScript. Students can choose their own order and often alternate between tasks as they work.

Prior work has emphasized the importance of real-time monitoring tools, but existing solutions struggle to effectively summarize student progress for complex, multi-faceted programming problems. Code clustering tools (e.g., [11], [12]) and progress visualizations (e.g., [13]) can summarize many code samples but do not give instructors control over which aspects to group by or summarize. Techniques for monitoring code in real-time (e.g., [14], [15]) give instructors real-time feedback but do not summarize students' progress and can be overwhelming in large classes. Further, most prior work does not address additional difficulties of monitoring in-class exercises. Implementations for features often span multiple files or modules [16], [17] but most prior work is focused on short, one-file snippets [11], [13], [14]. Further, instructors should be able to *explore* variations in students' code output and intermediate states to gain deeper insights into students' approaches and challenges.

To address these challenges, we introduce SPARK, a coding exercise monitoring dashboard designed for multi-faceted programming exercises. SPARK enables instructors to customize multi-level checkpoints with testing code suggestions, allowing them to track student progress for individual tasks. At each checkpoint, the testing code evaluates the intended outcomes, ensuring that students meet the specific objectives of the exercise. A progress visualization diagram that summarizes students' progress across tasks is generated using evaluation data from the testing code. Additionally, SPARK allows instructors to customize inspections of intermediate variables and outputs, providing active engagement and deeper insights into students' program state.

While SPARK is adaptable to various types of multi-faceted programming exercises, its implementation is specifically tailored for web programming. It includes features such as

*Work done as an undergraduate student at the University of Michigan.

1943-6106/25/$31.00 ©2025 IEEE
DOI 10.1109/VL-HCC65237.2025.00018

customized real-time inspections of output variations, as well as viewing DOM attributes and clustered previews of selected elements for evaluating web element performance across different students. Additionally, it allows for the simulation of element interactions within the testing code before conducting inspections, accommodating the event-driven nature of web programming tasks [18]–[20].

We created a dataset consisting of 22 students' keystroke data for two web programming problems in a 40-minute session. Using this data, we simulated a real-time classroom setting and conducted a within-subject user study with 16 participants to evaluate SPARK's effectiveness in helping instructors monitor students' programming progress. We found that SPARK helps participants 1) identify students' challenges more accurately and 2) feel more confident in their understanding of students' programming progress. Participants also reported that SPARK provides more detailed information and valuable, customizable insights into variations in students' code states. This work can help instructors improve real-time teaching by deepening their understanding of students' mental models and encouraging active engagement in the monitoring process. This work makes the following contributions:

- A pipeline that uses customized checkpoints with nested steps to visualize student progress and inspect immediate output variations in real-time.
- SPARK, a system based on this pipeline, designed to monitor student progress for web programming exercises.
- A dataset containing coding keystroke data from 22 students for two web programming exercises.
- A within-subject user study (N=16) involving 16 participants validating the effectiveness of SPARK.

II. RELATED WORK

A. Understanding Students' Programming Progress

Prior research has introduced various methods to visualize students' programming progress and support real-time monitoring, clustering, and runtime inspection. Many tools used 2D maps to track code changes and similarities [21], [22], and tools like VizProg [13] helped track different programming approaches. While these maps reflect the relative proximity of code states based on edit distance, they do not convey absolute positioning or capture the nuances of non-linear workflows in multi-faceted programming tasks. Such tasks involve interdependent subgoals that students often tackle out of sequence, making traditional linear visualizations insufficient. To address this, SPARK introduces a checkpoint-based framework for tracking progress across varied sequences.

Real-time monitoring is essential for timely feedback and maintaining student engagement [1]–[5], [10]. Tools like Codeopticon [14], RIMES [23], and VizProg [13] offer dashboards for live observation, yet they often present fixed data views. PuzzleMe [15] provides insights via peer-generated test cases, but it is designed for peer support, not instructor control. These systems fall short when instructors need to customize what they monitor, leading to either data overload or insufficient detail. SPARK overcomes this by letting instructors selectively view runtime outputs and variables, tailoring the feedback to immediate teaching needs.

To manage growing class sizes, many studies have employed clustering to summarize student solutions. Techniques such as AST edit distances [22], Overcode [11], and CFlow [12] group similar code to surface common patterns. Others, like [24]–[26], focus on functional patterns or errors. However, these methods often assume structural similarity, which is not always present in open-ended, multi-section exercises like web development. SPARK supplements clustering with test case results, enabling instructors to evaluate functional correctness regardless of divergent implementation paths.

Finally, understanding the runtime behavior of students' code is critical, especially when students struggle to articulate their issues [27]. Systems like Callisto [28] link questions to code, but real-time classroom scenarios demand more scalable solutions. Visualizing runtime values helps comprehension [29], yet with many students, it becomes hard to decide where to focus [30], [31]. Tools like RunEx [32] and TeachNow [33] provide scalable inspection and assistance. SPARK integrates workflow visualization with variable inspection, allowing instructors to first identify students with unusual progress patterns, and then drill down into variable-level details, offering a guided path from overview to diagnosis.

B. Runtime Variable Visualization

Variable visualizations play a crucial role in code comprehension, as inspecting variable states is essential to understanding how a program behaves [34]. Prior works have visualized variable values adjacent to code [35]–[38], while tools like Omnicode [39] and Theseus [40] present runtime behavior through scatterplot matrices or inline displays. CrossCode [41] extends this by visualizing multi-level execution traces. However, in classroom settings, instructors face significant cognitive load when trying to inspect runtime variables for every student [30]. To address this, SPARK enables clustered runtime value visualizations, helping instructors see variable states, outputs, and program behavior at scale. In event-driven, interactive web programming tasks, visual outputs are often key to understanding runtime behavior. Tools like Colaroid [42], CoCapture [43], and InterState [44] demonstrate the power of visual representations in supporting comprehension and communication. SPARK integrates this approach, allowing instructors to view students' rendered output directly, improving their understanding of dynamic interface behavior.

In the context of AI-generated code, research highlights the importance of building trust through runtime feedback [45], with visualizing intermediate values shown to help validate AI outputs [46]. Beyond AI, comparisons of runtime states also aid in understanding and debugging student or unfamiliar code. For instance, DITL helps data scientists compare datasets [47], and Doppio visualizes changes in UI flows [48]. These findings support the value of runtime comparison. In SPARK, instructors can monitor runtime behavior as students write code, and when creating test cases, they can combine

AI-generated suggestions with reference validation to verify both intermediate and final states—offering a reliable path to ensure code correctness.

III. System Design

A. System Design Goals

SPARK aims to support instructors in effectively monitoring students' progress during multi-faceted programming exercises in real time. Its design goals stem from the reflective analysis of instructional challenges around the difficulty of tracking diverse learning paths [13], [49].

1) DG1: Customizable Structured Progress Monitoring: Effectively monitoring individual student progress during programming exercises is critical for classroom management and student success [2], [15], [49], but it remains challenging. First, the diversity of teaching contexts requires flexible monitoring approaches [50], as classroom needs and instructional goals vary [51], [52]. This highlights the limitations of one-size-fits-all solutions and underscores the need for customizable tools. Second, interpreting student progress data can impose significant cognitive load on instructors [14], particularly in exercises involving complex workflows and varied learning paths. Even with real-time data, making sense of it remains difficult. Tools like Glancee [53] and VizProg [13] help visualize progress to reduce cognitive demands, but Lee et al. [54] emphasize that flexibility and customizability are essential to avoid information overload. This supports Dillenbourg's [55] argument that loosely structured activities are hard to manage without checkpoints.

These insights motivate DG1: *customizable structured progress monitoring*, calling for tools that provide customizable, organized insights tailored to the teaching contexts.

2) DG2: Gain a Holistic Understanding of Class Progress: While individual code submissions offer detailed insights, instructors need efficient tools to detect broader patterns that indicate conceptual misunderstandings across the class [56]. This need arises in two key contexts: in real-time teaching, instructors must balance individual support with class-wide awareness [54], [57]; in lab sessions, they often rely on large scale of retrospective reviews such as recordings or edit histories due to limited real-time visibility [58].

These challenges underscore the need for tools that aggregate and simplify synchronous and asynchronous programming data to reveal meaningful patterns while minimizing information loss [13], [21], [22]. Prior work demonstrates this principle: Taniguchi et al. [21] and Huang et al. [22] used 2D maps to visualize code evolution and similarity, and Zhang et al. [13] showed the value of tracking student progress at multiple granularities for classroom management. These align with visualization principles aimed at simplifying complex data without sacrificing essential information [59]–[61], and with Tissenbaum's call for real-time visualization to support instructional orchestration [62].

From these insights, we derive DG2: *aggregated progress tracking*, enabling instructors to efficiently detect class-wide patterns and bottlenecks.

3) DG3: Query Multiple Properties of Students' Code: DG3 builds on research in program comprehension [35]–[40], [46], [48] and instructional needs in classroom settings [30], [51], [52]. Studies show that understanding program behavior requires more than reviewing source code or final outputs—it involves examining intermediate runtime states [46], [48]. To support this, prior work has introduced techniques such as displaying variable values next to code [35]–[38] and embedding runtime visualizations in code editors [39], [40].

Instructors face additional challenges in classrooms, where they must assess knowledge mastery [51], [52], manage limited time and cognitive resources [30], and monitor many students simultaneously. Real-time visualization of intermediate code behavior can support this process, enhancing teaching efficiency.

Thus, DG3: *code querying capabilities* promotes tools that go beyond passive code review, allowing instructors to actively query checkpoint correctness, runtime state, and output. This empowers them to apply their expertise in diagnosing and supporting student learning.

B. Overview of SPARK

Informed by our design goals, we developed SPARK as a dashboard to help instructors monitor students' programming progress in real-time for multi-faceted programming problems, with its implementation focus on web programming exercises. SPARK consists of five panels:

- **Reference** panel (Fig.1.a), where instructors can enter their reference code answers to *Reference Code* board. The related preview (in *Reference Page Preview* board) and DOM tree (in *DOM Tree Reference* board) would be automatically generated.
- **Checkpoints** panel (Fig.1.b), which allows the creation and display of nested-task checkpoints. Each task includes a description and testing code for assessment.
- **Progress Visualization** panel (Fig.1.c), which features a progress visualization diagram.
- **Components Inspector** panel (Fig.1.d), which allows instructors to inspect students' output variations.
- **My Classroom** panel (Fig.1.e), which contains classroom statistics and student code boxes.

To illustrate the experience of using SPARK, we describe how a hypothetical instructor, Emily, conducts a multi-faceted programming exercise in class in real-time. Emily wishes to monitor students' programming progress, understand their programming progress, and provide timely assistance. For instance, she wants to identify common issues students are facing, as well as those who are falling behind, and offer support accordingly. Below, we describe how SPARK can help Emily monitor the classroom, highlighting both its features and implementation. In the scenario below, descriptions of SPARK's key features are integrated with screenshots and

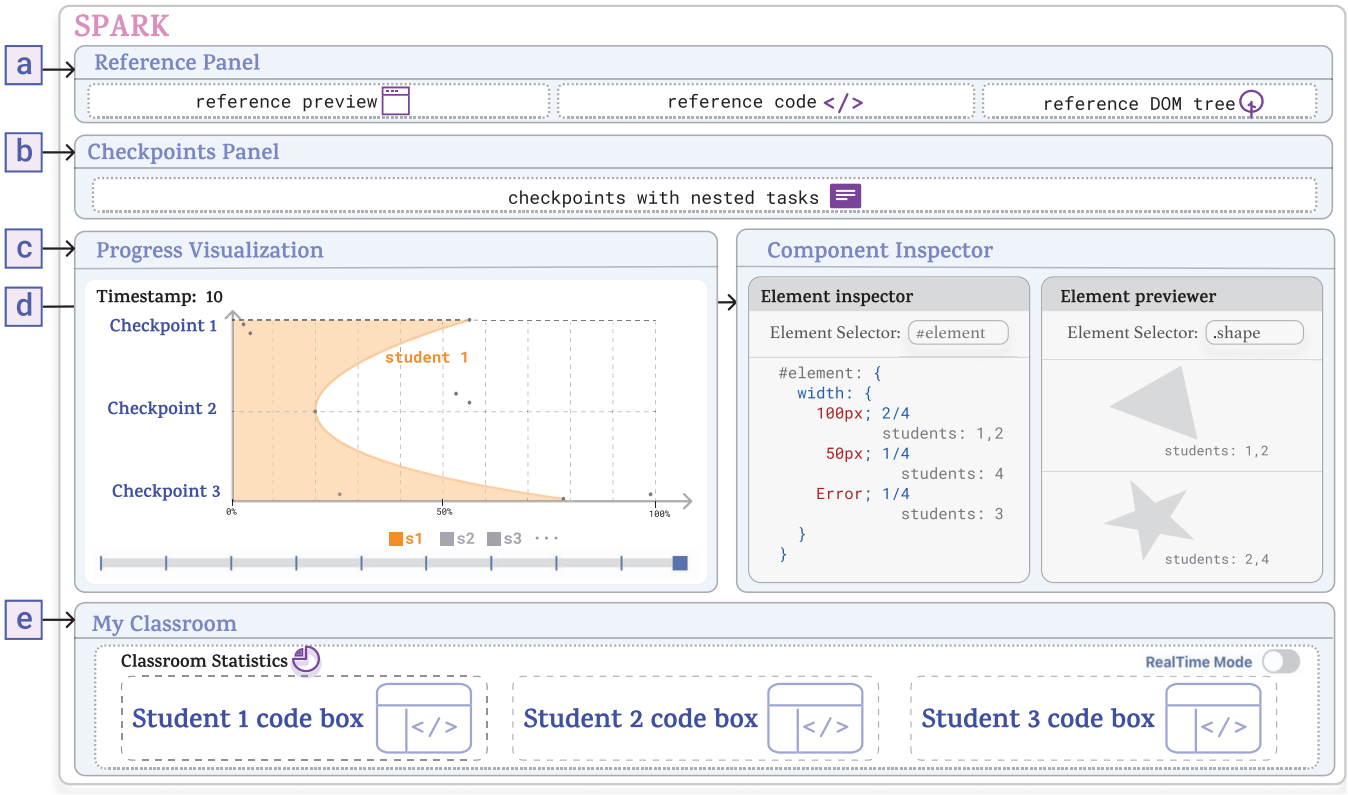

Fig. 1. SPARK consists of five panels. Here's an overview of the SPARK dashboard: the *Reference Panel* (a), which provides instructors with the expected code answer, the webpage, and a DOM tree preview for the programming exercise; the *Checkpoints Panel* (b), which allows for the creation and display of nested-task checkpoints; the *Progress Visualization* (c), which presents a visualization of students' programming progress; the *Component Inspector* (d), which enables instructors to customize inspections of students' output variations; and the *My Classroom* (e), which contains student code and statistics on overall task performance.

implementation details for each feature. Only instructors can see the features of SPARK.

C. Creating Checkpoints with Nested Tasks

Before the class begins, to use SPARK to monitor students' programming progress, Emily first creates checkpoints with grouped steps in the *Checkpoints* panel (Fig.2) (DG1). The process of creating checkpoints with nested tasks involves three steps. First, Emily inputs the reference code answers into the *Reference Code* board in the *Reference* panel, enters the task description into the task box (Fig.2.a), and clicks the *Generate Test* button (Fig.2.b) to use AI for generating testing code based on the task description and the reference code. Next, she could review the AI-generated testing code to ensure it meets her expectations, making any necessary modifications. Finally, to verify the accuracy of the testing code, Emily clicks Verify Checkpoint (Fig.2.c) to check if the testing code successfully passes against the reference code, displaying a success message (Fig.2.e). If the test fails, the system allows instructors to retrieve information through the return statement (Fig.2.d), which aids in debugging and identifying the issue. Throughout the editing and verification process, Emily could continually use the *Reference Page Preview*, *Reference Code*, and *DOM Tree Reference* boards to preview the expected programming exercise outcome, assisting in the creation and verification of checkpoints.

The testing code serves two primary functions: first, it evaluates whether a student's code meets the step requirements by assessing the behavior of specific elements; second, it simulates interactions before performing the evaluation. Additionally, the testing code is used in element inspections, as will be explained in Section III-F, allowing instructors to observe output variations with the required interactions simulated.

Implementation: SPARK uses the OpenAI API [63] to provide testing code suggestions[1] and employs Puppeteer [64] to simulate and evaluate code execution. The *Reference Page Preview* is implemented using an iframe, and the *DOM Tree Reference* is generated based on the *Reference Code*.

D. Real-time Monitoring of Students' Progress

Once Emily creates the checkpoints, she simply shares a folder with setup files and starter code. When students open it in VS Code with the required extension, SPARK begins receiving real-time programming data. This data is reorganized and displayed in code boxes within the *My Classroom* panel (Fig.3), similar to Codeopticon [14].

[1]More details could be found in the supplementary material: link.

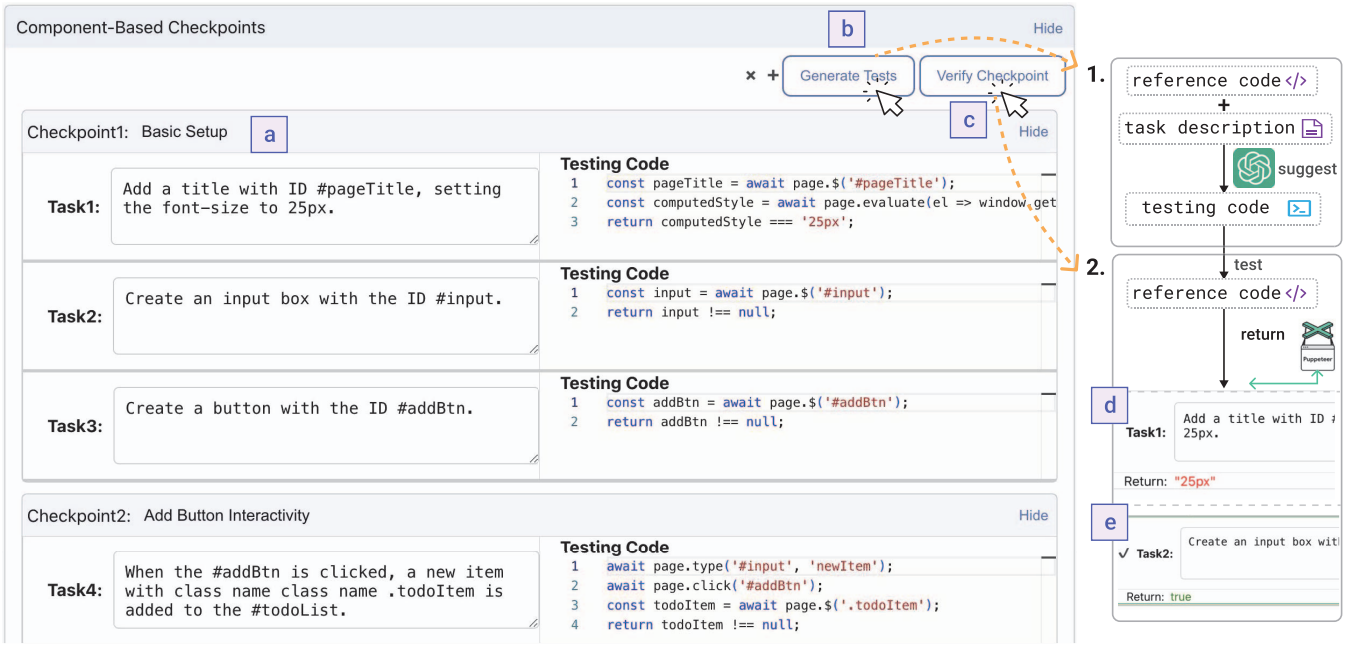

Fig. 2. *Checkpoints* panel. In the *Checkpoints* panel, instructors can freely create nested-task checkpoints. Each checkpoint consists of multiple tasks, and each task (a) is made up of two parts: *Task Description* and *Testing Code*. Instructors can click the *Generate Tests* (b) button to view AI-suggested testing code, which can be manually modified. They can then click the *Verify Checkpoint* (c) button to test the reference code to see if it passes (e) or fails (d).

Each code box (Fig.3.e) shows a student's live code and task completion status (Fig.3.a) across checkpoints. The *Classroom Statistics* board (Fig.3.b) offers a high-level view of task progress (DG2). The panel supports two modes: real-time (live keystroke updates) and timestamp (minute-by-minute snapshots), allowing Emily to review students' code history via a slider (Fig.3.c). A blue-highlighted file name indicates the currently active file.

By default, code boxes are ordered by student ID. Emily can rearrange them in the *Progress Visualization* or *Components Inspector* panels, and reset the layout via the *Reset Order* button (Fig.3.d). While students work, Emily monitors progress using the *Progress Visualization* panel (auto-updated every minute) and the *Components Inspector* panel for more detailed inspection (Figs.4, 5).

Implementation: SPARK uses a custom VS Code extension to capture and transmit keystroke-level data (edit content, location, and timestamp). Only this lightweight edit data is sent to SPARK, which organizes and displays it in real time within the *My Classroom* panel (Fig.3).

E. Progress Visualization View

In the *Progress Visualization* panel (Fig.4), each student's progress within a checkpoint is represented by a dot (Fig.4.a), placed left to right from 0% to 100% task completion (Fig.4.b). A student's overall progress is visualized as a shaded area; overlapping areas indicate similar progress levels, with darker shades showing higher student density (DG2).

Hovering over a dot highlights the student's trajectory line across checkpoints and changes their shaded area to a unique color while hiding others (Fig.4.c) (DG1). Emily can also use the brush tool to select and view groups of students (Fig.4.d) (DG2) or use the slider (Fig.4.f) to explore progress over time.

To investigate a specific student who struggled with checkpoint 2, Emily can click their dot or label to lock the highlight (Fig.4.e), then use the slider to trace their progress over time (Fig.4.g). The student's code box is also brought to the top for direct inspection.

Implementation: SPARK uses Puppeteer [64] to run predefined test cases on students' code, using the results to generate visualizations of checkpoint completion.

F. Component Inspector View

With a general view of student progress, Emily turns to the *Components Inspector* panel (Fig.5.a) for deeper analysis of specific elements. This panel offers two customizable inspection features: DOM property inspection (Fig.5.d) and visual previews (Fig.5.f), which can be used separately or together.

Emily selects the relevant task (Fig.5.b), enters the element selector, and uses the *Element Inspector* to check property variations via the *Property Selector* (Fig.5.d, g), with matching student counts shown (DG3). Alternatively, she can preview how elements render across students using the *Element Previewer* (Fig.5.e, f). Clicking *Inspect* runs the analysis, and she can use the magnifier button to bring students with matching issues to the top of the *My Classroom* panel for closer review.

The inspector panel mirrors the checkpoint structure, with each task linked to an inspector board. Results reflect simulated interactions from the checkpoint's testing code (e.g., Fig.5.h shows interaction with the "add todo item" element).

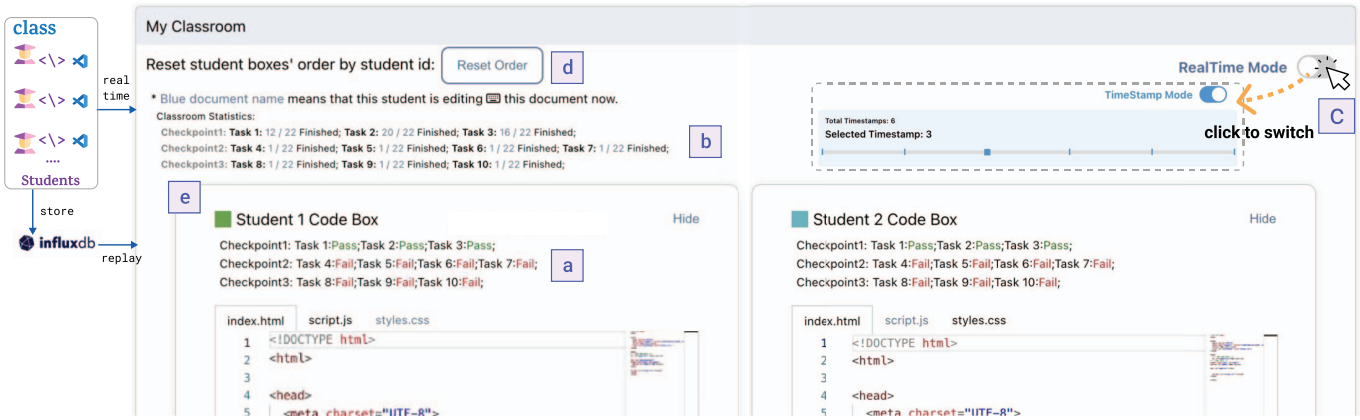

Fig. 3. *My Classroom* panel. Each student working on the programming problem has a *Student Code Box* (e), which includes the student's code and *task completion status* (a) indicating whether they have passed the task. There is also a *Classroom Statistics* board (b) showing the class's performance across different tasks. The *My Classroom* panel can switch between two modes (c): *Real-Time Mode*, where instructors can view students' code in real-time, and *Timestamp Mode*, which records students' code history every minute. In *Timestamp Mode*, Instructors can use the slider to review the students' code history. If instructors change the order of the student code boxes in the *Progress Viz* panel or *Components Inspector* panel, they can use the *Reset Order* button (d) to revert the Code Boxes to their default order.

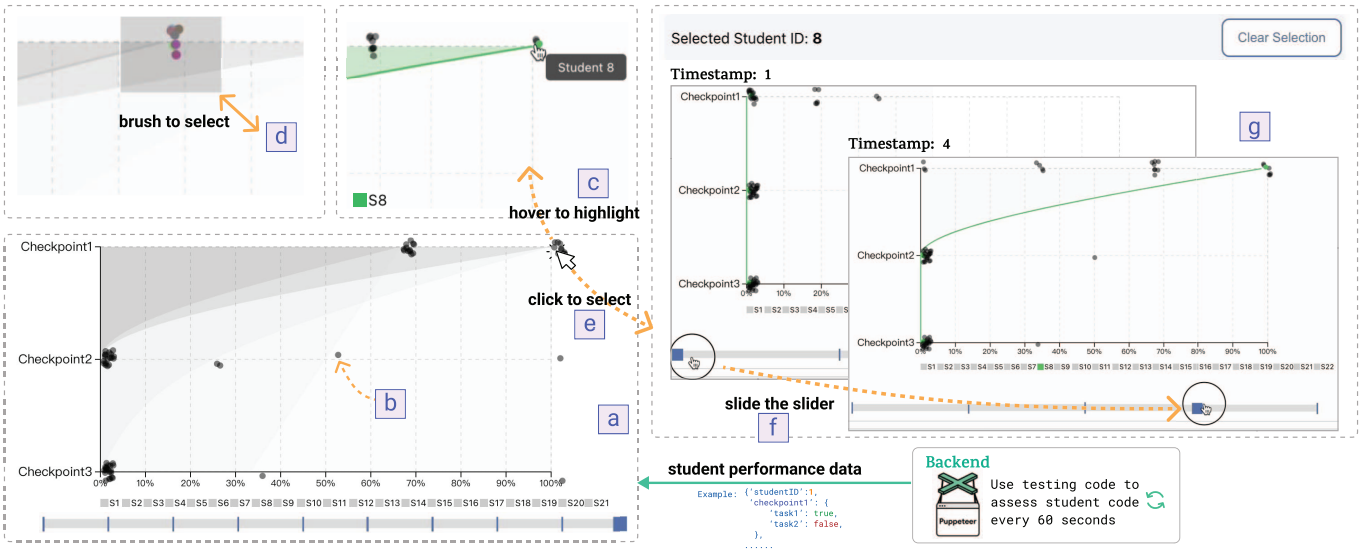

Fig. 4. *Progress Visualization* panel. The *Progress Visualization* panel includes a diagram (a) that displays students' progress across checkpoints. Each dot along the checkpoint line represents a student's task completion rate for that checkpoint at a particular timestamp (b). Instructors can hover over a dot to highlight a student (c), use the brush tool to select multiple students within that area (d), and click to select an individual student (e), with the selected student remaining highlighted. Additionally, instructors can adjust the slider to view the visualization diagrams at different timestamps (f).

Implementation: SPARK sends students' code to Puppeteer [64], which simulates test interactions before inspection. For the *Element Previewer*, visual similarities are clustered using Resemble.js [65].

G. Recording and Replaying

SPARK supports both real-time monitoring and replay via keystroke recording. The replay feature helps address key challenges: the cognitive load of real-time tracking, unequal attention to students, and oversight during TA-led sessions. For example, when instructors like Emily step away to assist students, they may miss critical moments. With SPARK, they can review class data afterward or merge asynchronous session recordings (DG2), enabling retrospective analysis to identify common struggles and provide targeted support. This ensures no student progress is overlooked.

Implementation: SPARK logs keystroke data to a database (e.g., InfluxDB). In replay mode, it retrieves and chronologically replays this time-series data for simulation.

H. Example Usage Contexts

SPARK supports both in-person and online classrooms, including asynchronous settings like MOOCs, enabling real-time progress monitoring and learning support. This flexibility is especially valuable for reserved students who may hesitate

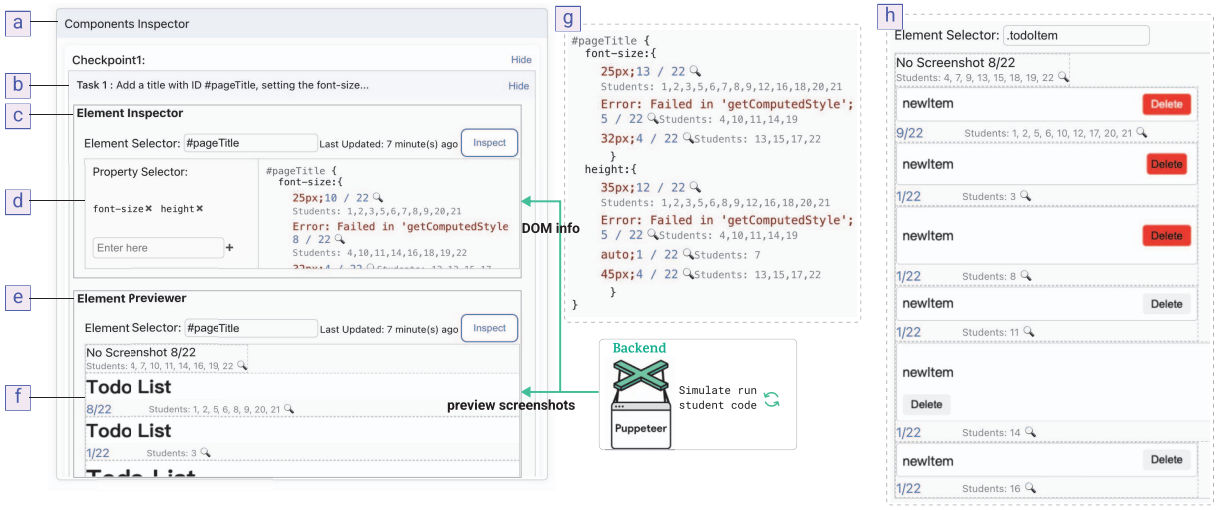

Fig. 5. *Components Inspector* Panel. The *Components Inspector* (a) is structured according to checkpoints. Each task is linked to a corresponding inspector board (b) that includes two key features: the *Element Inspector* (c) and the *Element Previewer* (e). Instructors can view the inspection results in (d) (a full view of the results is shown in (g)) and preview the element in (f). The inspector automatically simulates interactions before performing the inspection. For example, (h) displays clustered screenshots of the newly added `.todoitem` following the interaction *"Add new item after clicking the add button (Task 4)"*. Screenshots of identical elements are grouped together to facilitate easier analysis.

to seek help [66]–[68], allowing instructors to track and assist learners regardless of when or how they engage.

SPARK can also scale beyond intermediate web programming by adjusting checkpoint granularity—for example, supporting finer-grained steps in beginner machine learning tasks like building a digit recognition pipeline. With shared rubrics and replay features, SPARK promotes consistent evaluation and coordinated instruction across teaching teams.

IV. EVALUATION

A. Dataset of Real-Time Programming Data

To simulate a real-time classroom environment for web programming exercises, we conducted a data collection session prior to the user study [69]. We recruited 22 students with programming experience, including 10 beginners, 11 intermediate, and 1 advanced in web programming.

Sessions were held via Zoom, with each participant completing two same 20-minute introductory web programming tasks. A research team member collected data using a customized VS Code extension. Participants could use resources like Google and Copilot [70], but not LLMs to generate code, balancing realistic usage with data reliability.

Keystroke-level data—averaging 810 keystrokes per student per task—was stored in InfluxDB with timestamps, edit locations, and anonymized IDs. SPARK then replayed this data chronologically to simulate real-time progress. This dataset offers a fine-grained view of coding behavior, addressing the lack of real-time detail in traditional datasets and providing a valuable benchmark for future research on programming learning and problem-solving.

B. User Study

To evaluate SPARK's effectiveness, we conducted a within-subjects study with 16 participants experienced in teaching and web programming. Using the dataset from Section IV-A, participants simulated real-time classroom monitoring by observing replays and answering quiz questions about student progress and challenges.

1) Recruitment: We recruited 16 participants (9F, 6M) from Computer Science and Information Science departments based on their teaching background and web programming experience. Participants included instructors, tutors, TAs, and experienced graduate students. Fifteen had prior teaching experience, with 1 to 6+ years of web programming experience.

2) Study Protocol: The user study used a within-subjects design with three sessions. In S1 and S2, participants used the Baseline and SPARK systems to monitor student progress. And in S3, participants evaluated the preparation phase by creating checkpoints and test code using SPARK.

- **Baseline:** A simplified version of SPARK with *Reference* and *My Classroom* panels; real-time and timestamped code views only, without performance metrics.
- **SPARK:** Full version of SPARK with all the features.

Tasks were consistent across S1 and S2, with the order counterbalanced. S3 was always used as the final activity. Each session included a 5-minute tutorial. Four participants joined in person, while 12 participated remotely.

In S1 and S2, participants observed replays of 22 students' progress and answered quiz questions during and after the session. The quiz questions included nine questions combining fact-based queries, diagnostic tasks, and open-ended reflections. Each session included with a questionnaire (5-point Likert scale) and a brief interview. S1 and S2 were

limited to 25 minutes for comparability. In S3, after a tutorial, participants created a checkpoint and generated corresponding test code, then completed a usability questionnaire.

3) Data Collection and Analysis: During screening, we collected participants' teaching and web programming experience. In each session, a researcher took observation notes and graded quiz responses. S1 and S2 used nearly identical quizzes tailored to their respective exercises, each containing 5 multiple-choice and 4 open-ended questions. We recorded quiz accuracy and time spent per question using screen recordings.

Data analysis included questionnaire ratings, quiz performance, and self-reported confidence. A mixed-effects linear regression model revealed significant effects of system type (SPARK vs. Baseline). The order in which tools were used also had a significant effect in most cases, with lower ratings observed when SPARK was used first—likely due to comparison effects. In contrast, problem type showed no consistent influence. Full results are shown in Table I and Fig. 6.

We also conducted a thematic analysis of 16 semi-structured interviews. Transcripts were coded and iteratively clustered into themes to extract key insights.

C. Results

1) SPARK *helps instructors identify students' challenges more accurately:* In the quiz, participants assessed students' performance, identified issues, and recognized shared challenges. We counted the number of correct answers participants provided in each session, and there was a significant difference between SPARK and Baseline ($p < .001$), with SPARK showing a marked improvement in answer accuracy.

We observed that in the Baseline session, when asked to assess students' programming performance, 11 out of 16 participants only glanced at the top half of the student code box, making observations based on this limited view. In contrast, 15 out of 16 participants in the SPARK session used the *Component Inspector* to gain insights into students' code behavior. Several participants noted that SPARK has a learning curve, but once they became familiar with the system and the checkpoints, it significantly improved the quality of monitoring, making the effort worthwhile (P1, P5, P9-P10). P9 mentioned that *"..., it is kind of hard (to learn)...but it gives good overview of students' states."*.

2) SPARK *improves instructors' confidence in understanding students' programming progress:* We found that participants gained significantly more confidence in the monitoring results with SPARK, as shown in Fig.6. There was a notable difference in the number of unsure answers in the quiz between SPARK and the Baseline system ($p < .001$). In the Baseline system, many participants made incorrect assessments of students' performance (P1, P4-6, P11-14), often misinterpreting syntax they believed to be correct but that was actually incorrect. Additionally, some participants (P4, P12) were confident in their observations and assessments while using the Baseline system, yet still made inaccurate assessments.

When comparing SPARK to the Baseline system, many participants highlighted the value of test cases for evaluating specific tasks. As P8 noted, *"Using test cases to show inter-student progress...that's a really good idea."* P10 added, *"They provide a more intimate understanding...showing real-time progress and highlighting issues."*

In post-session interviews (S1 and S2), all 16 participants expressed interest in using SPARK in real classrooms, while only 7 were open to using the Baseline system—2 of whom would do so only in small classes. As P12 commented, *"It's better than nothing...but with many students, I won't have time."* P7 remarked, *"Looking at too many students' code is exhausting. I'd rather see nothing."*

D. System Usability and Study Insights

1) SPARK *enables more detailed programming progress monitoring:* In the post-session questionnaire, 15 of 16 participants agreed that SPARK provided detailed insights into student progress. Compared to the Baseline, SPARK enabled faster, more accurate checkpoint assessments—14 of 16 answered correctly in a quiz using SPARK, versus 7 with the Baseline. Baseline users also took twice as long on average.

Participants (P2–6, P11–15) praised the *Progress Visualization* for its clarity and intuitive interactions (e.g., hover and brush), with P2 noting, *"Grouping tasks into checkpoints saves unnecessary effort reviewing similar code."*

All participants used the visualization to identify struggling students and reviewed code histories via the timestamp feature. Several (e.g., P13, P16) noted students followed varied workflow paths to reach correct solutions.

SPARK offered deeper insights beyond task progress. P6 pointed out that abrupt code trajectory changes could indicate copying. In an open-ended quiz, 12 participants used SPARK to decide which topics to revisit, and 9 used classroom statistics to identify difficult concepts. P2 noted, *"Seeing how long students struggled helps decide what to emphasize next time."*

2) SPARK *enables a better understanding of variations among students' code output:* In the post-session questionnaire, participants rated SPARK significantly higher than the Baseline system for inspecting variations in students' output ($p < .001$). Many found the *Component Inspector* helpful for understanding performance, with P6 and P8 noting it *"provides a direct way to understand students' performance"*, and P2 highlighting its value for visualizing layout and properties.

The checkpoint structure also reduced cognitive load and made tracking progress more intuitive. P15 noted, *"Organizing tasks into checkpoints is intuitive and allows for more detailed insights."* While Baseline users struggled to recall common issues, 15 of 16 participants using SPARK successfully identified at least one issue faced by over half the class.

3) SPARK *enables instructors to active engage with the monitoring process:* During the SPARK condition, participants used the *Components Inspector* an average of 3.4 times during the 20-minute session. Many appreciated its customization, with P2 noting, *"It's great that I could inspect only one element—much easier to compare."* In exploring issues in *checkpoint 2*, 15 of 16 participants used the inspector, most selecting multiple task boards. As P12 observed, *"...the add*

TABLE I
MIXED-EFFECTS LINEAR REGRESSION MODEL RESULTS FOR TOOL EFFECT (SPARK VS. BASELINE)

Statement	b	SE	z	p	95% CI Low	95% CI High
Monitor all students' programming progress comprehensively.	2.44	0.31	7.94	<.001	1.84	3.04
Gain detailed insights into students' programming trajectories.	2.00	0.29	6.80	<.001	1.42	2.58
Identify students who are falling behind.	2.06	0.33	6.35	<.001	1.43	2.70
Identify challenges faced by students.	2.06	0.35	5.87	<.001	1.37	2.75
Assess the element functionality among students.	2.75	0.27	10.37	<.001	2.23	3.27
Inspect variations in element behavior output among students.	3.00	0.27	10.93	<.001	2.46	3.54
Number of correct answers in the quiz (5 in total)	2.19	0.26	8.35	<.001	1.67	2.70
Number of unsure answers in the quiz (5 in total)	-2.25	0.38	-5.93	<.001	-2.99	-1.51

Statement	Condition	N	M	SD
Monitor all students' programming progress comprehensively.	Baseline	16	2.19	1.28
	SPARK	16	4.63	0.62
Gain detailed insights into students' programming trajectories.	Baseline	16	2.63	1.15
	SPARK	16	4.63	0.62
Identify students who are falling behind.	Baseline	16	2.81	1.42
	SPARK	16	4.88	0.34
Identify challenges faced by students.	Baseline	16	2.06	1.06
	SPARK	16	4.13	1.09
Assess the element functionality among students.	Baseline	16	1.69	0.79
	SPARK	16	4.44	0.96
Inspect variations in element behavior output among students.	Baseline	16	1.69	1.01
	SPARK	16	4.69	0.48
Number of correct answers in the quiz (5 in total)	Baseline	16	2.50	0.97
	SPARK	16	4.69	0.70
Number of unsure answers in the quiz (5 in total)	Baseline	16	2.56	1.46
	SPARK	16	0.31	0.79

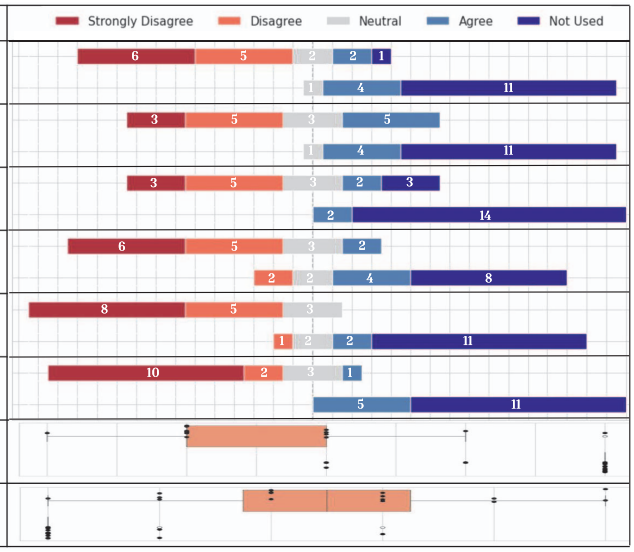

Fig. 6. Perceptions of the Baseline and SPARK system. Participant rated on a 5-point scale. (M: mean. SD: standard deviation).

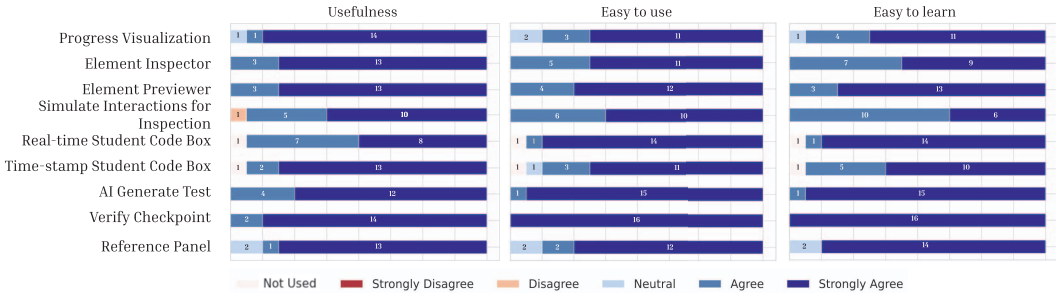

Fig. 7. Results from the questionnaire of the Likert-scale responses to "usefulness", "easy to use", and "easy to learn" after each session.

button interactivity is more difficult, so I'd take a look at this." SPARK's flexible inspection tools enabled participants to focus on specific problem areas, boosting their confidence in identifying issues and offering targeted feedback.

4) SPARK makes creating step-nested checkpoints as well as their test cases easy: In the third session (S3), participants used SPARK to create a checkpoint with one task. Features were highly rated for usefulness, ease of use, and ease of learning (Fig.7). Most participants strongly agreed that *AI Generate Test* (12/16), *Verify Checkpoint* (14/16), and *Reference Panel* (13/16) supported easy test case creation and clarified assessment goals.

Participants found the AI-generated tests *"super convenient and time-saving (P10, P11)"*, while the verification and reference features gave them *"confidence [they] could use these in real classrooms (P13)"*. All 16 participants expressed willingness to use these features for creating step-nested checkpoints.

User Challenges and Feedback. Participants identified several challenges when using SPARK. First, they raised concerns about the scalability of the scatter plot, which became increasingly cluttered and difficult to interpret as the number of students grew. Second, some participants noted that the system's rich features and modular interface, while powerful, occasionally introduced additional visual and cog-

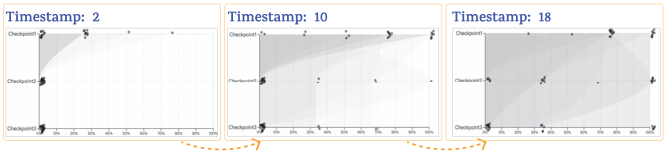

Fig. 8. How the shaded areas in the visualization diagram change over time in the *Image Carousel* example.

nitive load—particularly during real-time monitoring. These observations suggest the need for systems that can better support large-scale classrooms while maintaining usability and minimizing cognitive effort.

V. Discussion

A. Visualizing Multi-facet Programming at Scale

The expansion of programming education has led to larger class sizes, making it difficult for instructors to monitor student progress—especially in multi-faceted tasks with loosely structured workflows, such as data analysis pipelines, GUI applications, or hardware programming. These tasks involve complex codebases and exploratory coding behaviors that traditional clustering techniques struggle to capture. For instance, in data analysis, students may experiment with features that don't impact final outputs, which runtime- or AST-based clustering may overlook or misrepresent.

Our study shows that checkpoint-based visualizations provide a more effective solution. By allowing instructors to define key stages of the task, they can track how students iteratively approach each checkpoint over time. This design accommodates non-linear workflows and highlights meaningful progress. Future work may explore expanding checkpoint mechanisms using AI-assisted techniques [46] or peer assessment [15] to provide richer insights into student performance.

Our user study confirms that SPARK performs well in small-to-medium-sized classrooms. In testing with 22 students, a single Puppeteer server maintained sub-30-second computation times. These benchmarks suggest the system can scale further for low-latency, real-time tracking in larger classrooms.

B. Visualizing Student Code with Spatial Meanings

By visualizing test case results across checkpoints, SPARK provides a reliable way to track student progress in multi-faceted programming exercises. This approach reveals students' workflow sequences and clarifies task interdependencies, enabling instructors to quickly assess progress across different code sections. For example, in the *Image Carousel* exercise (Fig.8), some students completed *Checkpoint 2* first, while most followed the expected sequence. At *timestamp 18*, performance was highest on *Checkpoint 2* and lowest on *Checkpoint 3*. The spatial layout helps instructors understand both overall progress and specific challenges at each stage.

While VizProg [13] uses absolute code positions for progress, SPARK aligns progress with individual workflow sequences and using shaded spatial regions to signal differences in progress. This enhances instructors' situational awareness and offers a more intuitive understanding of student behavior. Although this method scales well to large classes, it may lead to information overload in very large cohorts. Future work could explore adaptive filtering and summary views to surface key trends and outliers.

Engagement with Instructors

Our findings show that participants actively engaged with SPARK's interactive features, finding them effective for managing information and retrieving key details. Unlike prior systems [11], [13], [14] that support mostly passive monitoring, SPARK enables a hands-on approach, allowing instructors to create custom checkpoints and inspect student work in-session—reducing cognitive load and boosting confidence in instructional decisions.

As AI tools become more common, it's essential to balance automation with human oversight. While AI can assist with predictions or evaluations, meaningful instruction relies on active engagement. Effective learning analytics should go beyond correctness metrics to offer insights into student reasoning and strategies. SPARK supports this by combining AI-generated test suggestions with instructor-led exploration.

Future work could incorporate features like automated feedback [57], predictive analytics, and inactivity tracking, while ensuring instructors remain central to the monitoring process.

C. Limitations

A current limitation of SPARK is the preparation effort. While it is currently tailored for intermediate web programming, future work could explore AI-assisted preparation workflows to balance customization with efficiency, and extend SPARK to advanced domains such as machine learning.

The user study also has two main limitations. First, the initial sessions focused on using pre-generated checkpoints and test code, with only the third session evaluating their creation. This offers limited insight into how instructors might generate and use custom tests in authentic teaching contexts. Future studies could examine instructor interactions when designing their own tests to better understand SPARK's support for real-time monitoring. Second, since SPARK can inform teaching pace and concept review, future research could explore its long-term classroom use to understand how instructors adapt and integrate its features into everyday teaching.

VI. Conclusion

This paper presents a real-time visualization approach for multi-faceted programming exercises in classroom settings. We developed SPARK, a dashboard that lets instructors define checkpoints, suggest automated tests, and visualize student progress across varied workflow sequences. SPARK also supports inspection of intermediate outputs, offering deeper insight into students' code states. Our evaluation shows that SPARK facilitates easy checkpoint creation, detailed progress monitoring, and active, customizable engagement. By visualizing progress and variables, SPARK helps instructors better understand students' mental models, reduce cognitive load, and deliver more effective, personalized feedback at scale.

ACKNOWLEDGMENT

We thank Xinyi Chen for the valuable feedback on the SPARK project. We are also grateful to all study participants and collaborators for their time and contributions.

REFERENCES

[1] E. De Graaf and A. Kolmos, "Characteristics of problem-based learning," *International journal of engineering education*, vol. 19, no. 5, pp. 657–662, 2003.

[2] C. E. Hmelo-Silver, "Problem-based learning: What and how do students learn?" *Educational psychology review*, vol. 16, pp. 235–266, 2004.

[3] J. Michael, "Where's the evidence that active learning works?" *Advances in physiology education*, 2006.

[4] R. M. Felder and R. Brent, "Active learning: An introduction," *ASQ higher education brief*, vol. 2, no. 4, pp. 1–5, 2009.

[5] M. Prince, "Does active learning work? a review of the research," *Journal of engineering education*, vol. 93, no. 3, pp. 223–231, 2004.

[6] W. Peng, Y. Yang, Z. Zhang, and T. J.-J. Li, "Glitter: An ai-assisted platform for material-grounded asynchronous discussion in flipped learning," 2025. [Online]. Available: https://arxiv.org/abs/2504.14695

[7] D. H. Jonassen and W. Hung, "All problems are not equal: Implications for problem-based learning," *Essential readings in problem-based learning: Exploring and extending the legacy of Howard S. Barrows*, vol. 1741, 2015.

[8] D. W. Schön, "Teaching artistry through reflection-in-action," 2015.

[9] H. A. Simon, "The structure of ill structured problems," *Artificial intelligence*, vol. 4, no. 3-4, pp. 181–201, 1973.

[10] E. J. Huang, D. Rees Lewis, S. Gaudani, M. Easterday, and E. Gerber, "Intelligent coaching systems: Understanding one-to-many coaching for ill-defined problem solving," *Proc. ACM Hum.-Comput. Interact.*, vol. 7, no. CSCW1, apr 2023. [Online]. Available: https://doi-org.proxy.lib.umich.edu/10.1145/3579614

[11] 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 Transactions on Computer-Human Interaction (TOCHI)*, vol. 22, no. 2, pp. 1–35, 2015.

[12] A. G. Zhang, X. Tang, S. Oney, and Y. Chen, "Cflow: Supporting semantic flow analysis of students' code in programming problems at scale," in *Proceedings of the Eleventh ACM Conference on Learning@ Scale*, 2024, pp. 188–199.

[13] A. G. Zhang, Y. Chen, and S. Oney, "Vizprog: Identifying misunderstandings by visualizing students' coding progress," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, ser. CHI '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi-org.proxy.lib.umich.edu/10.1145/3544548.3581516

[14] P. J. Guo, "Codeopticon: Real-time, one-to-many human tutoring for computer programming," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, ser. UIST '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 599–608. [Online]. Available: https://doi-org.proxy.lib.umich.edu/10.1145/2807442.2807469

[15] A. Y. Wang, Y. Chen, J. J. Y. Chung, C. Brooks, and S. Oney, "Puzzleme: Leveraging peer assessment for in-class programming exercises," *Proceedings of the ACM on Human-Computer Interaction*, vol. 5, no. CSCW2, pp. 1–24, 2021.

[16] L. Welling and L. Thomson, *PHP and MySQL Web development*. Sams publishing, 2003.

[17] M. Grinberg, *Flask web development*. " O'Reilly Media, Inc.", 2018.

[18] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman, "Understanding javascript event-based interactions," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 367–377. [Online]. Available: https://doi-org.proxy.lib.umich.edu/10.1145/2568225.2568268

[19] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst, "Interactive record/replay for web application debugging," in *Proceedings of the 26th annual ACM symposium on User interface software and technology*, 2013, pp. 473–484.

[20] J. Hibschman, D. Gergle, E. O'Rourke, and H. Zhang, "Isopleth: Supporting sensemaking of professional web applications to create readily available learning experiences," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 26, no. 3, pp. 1–42, 2019.

[21] Y. Taniguchi, T. Minematsu, F. Okubo, and A. Shimada, "Visualizing source-code evolution for understanding class-wide programming processes," *Sustainability*, vol. 14, no. 13, p. 8084, 2022.

[22] J. Huang, C. Piech, A. Nguyen, and L. Guibas, "Syntactic and functional variability of a million code submissions in a machine learning mooc," in *AIED 2013 Workshops Proceedings Volume*, vol. 25. Citeseer, 2013.

[23] J. Kim, E. L. Glassman, A. Monroy-Hernández, and M. R. Morris, "Rimes: Embedding interactive multimedia exercises in lecture videos," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, ser. CHI '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1535–1544. [Online]. Available: https://doi-org.proxy.lib.umich.edu/10.1145/2702123.2702186

[24] S. Kaleeswaran, A. Santhiar, A. Kanade, and S. Gulwani, "Semi-supervised verified feedback generation," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 739–750.

[25] A. Head, E. Glassman, G. Soares, R. Suzuki, L. Figueredo, L. D'Antoni, and B. Hartmann, "Writing reusable code feedback at scale with mixed-initiative program synthesis," in *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*, 2017, pp. 89–98.

[26] A. Nguyen, C. Piech, J. Huang, and L. Guibas, "Codewebs: scalable homework search for massive open online programming courses," in *Proceedings of the 23rd international conference on World wide web*, 2014, pp. 491–502.

[27] J. M. Markel and P. J. Guo, "Inside the mind of a cs undergraduate ta: A firsthand account of undergraduate peer tutoring in computer labs," in *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, 2021, pp. 502–508.

[28] A. Y. Wang, Z. Wu, C. Brooks, and S. Oney, "Callisto: Capturing the" why" by connecting conversations with computational narratives," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020, pp. 1–13.

[29] M.-A. Storey, F. D. Fracchia, and H. A. Müller, "Cognitive design elements to support the construction of a mental model during software exploration," *Journal of Systems and Software*, vol. 44, no. 3, pp. 171–185, 1999.

[30] G. A. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information." *Psychological review*, vol. 63, no. 2, p. 81, 1956.

[31] T. W. Jackson and P. Farzaneh, "Theory-based model of factors affecting information overload," *International Journal of Information Management*, vol. 32, no. 6, pp. 523–532, 2012.

[32] A. G. Zhang, Y. Chen, and S. Oney, "Runex: Augmenting regular-expression code search with runtime values," in *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2023, pp. 139–147.

[33] A. Malik, J. Woodrow, C. Wang, and C. Piech, "Teachnow: Enabling teachers to provide spontaneous, realtime 1:1 help in massive online courses," in *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*, ser. ITiCSE 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 708–714. [Online]. Available: https://doi-org.proxy.lib.umich.edu/10.1145/3649217.3653629

[34] B. W. Kernighan, *The practice of programming*. Pearson Education India, 1999.

[35] S. Lerner, "Projection boxes: On-the-fly reconfigurable visualization for live programming," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020, pp. 1–7.

[36] P. Guo, "Ten million users and ten years later: Python tutor's design guidelines for building scalable and sustainable research software in academia," in *The 34th Annual ACM Symposium on User Interface Software and Technology*, 2021, pp. 1235–1251.

[37] P. J. Guo, "Online python tutor: embeddable web-based program visualization for cs education," in *Proceeding of the 44th ACM technical symposium on Computer science education*, 2013, pp. 579–584.

[38] P. Jiang, F. Sun, and H. Xia, "Log-it: Supporting programming with interactive, contextual, structured, and visual logs," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–16.

[39] H. Kang and P. J. Guo, "Omnicode: A novice-oriented live programming environment with always-on run-time value visualizations," in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, 2017, pp. 737–745.

[40] T. Lieber, J. R. Brandt, and R. C. Miller, "Addressing misconceptions about code with always-on programming visualizations," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2014, pp. 2481–2490.

[41] D. Hayatpur, D. Wigdor, and H. Xia, "Crosscode: Multi-level visualization of program execution," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–13.

[42] A. Y. Wang, A. Head, A. G. Zhang, S. Oney, and C. Brooks, "Colaroid: A literate programming approach for authoring explorable multi-stage tutorials," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–22.

[43] Y. Chen, S. W. Lee, and S. Oney, "Cocapture: Effectively communicating ui behaviors on existing websites by demonstrating and remixing," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, ser. CHI '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi-org.proxy.lib.umich.edu/10.1145/3411764.3445573

[44] S. Oney, B. Myers, and J. Brandt, "Interstate: a language and environment for expressing interface behavior," in *Proceedings of the 27th annual ACM symposium on User interface software and technology*, 2014, pp. 263–272.

[45] R. Wang, R. Cheng, D. Ford, and T. Zimmermann, "Investigating and designing for trust in ai-powered code generation tools," in *The 2024 ACM Conference on Fairness, Accountability, and Transparency*, 2024, pp. 1475–1493.

[46] K. Ferdowsi, R. Huang, M. B. James, N. Polikarpova, and S. Lerner, "Validating ai-generated code with live programming," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–8.

[47] A. Y. Wang, W. Epperson, R. A. DeLine, and S. M. Drucker, "Diff in the loop: Supporting data comparison in exploratory data analysis," in *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, 2022, pp. 1–10.

[48] P.-Y. Chi, S.-P. Hu, and Y. Li, "Doppio: Tracking ui flows and code changes for app development," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018, pp. 1–13.

[49] N. Gil Fonseca, L. Macedo, and A. J. Mendes, "Supporting differentiated instruction in programming courses through permanent progress monitoring," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 209–214. [Online]. Available: https://doi-org.proxy.lib.umich.edu/10.1145/3159450.3159578

[50] C. Deed, D. Blake, J. Henriksen, A. Mooney, V. Prain, R. Tytler, T. Zitzlaff, M. Edwards, S. Emery, T. Muir *et al.*, "Teacher adaptation to flexible learning environments," *Learning Environments Research*, vol. 23, pp. 153–165, 2020.

[51] K. R. Koedinger, A. T. Corbett, and C. Perfetti, "The knowledge-learning-instruction framework: Bridging the science-practice chasm to enhance robust student learning," *Cognitive science*, vol. 36, no. 5, pp. 757–798, 2012.

[52] D. L. Schwartz, J. M. Tsang, and K. P. Blair, *The ABCs of how we learn: 26 scientifically proven approaches, how they work, and when to use them*. WW Norton & Company, 2016.

[53] S. Ma, T. Zhou, F. Nie, and X. Ma, "Glancee: An adaptable system for instructors to grasp student learning status in synchronous online classes," in *Proceedings of the 2022 CHI conference on human factors in computing systems*, 2022, pp. 1–25.

[54] H. Y. Lee, S. Park, E. H. Kim, J. Seo, H. Lim, and J. Lee, "Investigating the effects of real-time student monitoring interface on instructors' monitoring practices in online teaching," in *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, ser. CHI '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi-org.proxy.lib.umich.edu/10.1145/3613904.3642845

[55] P. Dillenbourg, "Design for classroom orchestration," *Computers & education*, vol. 69, pp. 485–492, 2013.

[56] S. K. Card, J. Mackinlay, and B. Shneiderman, *Readings in information visualization: using vision to think*. Morgan Kaufmann, 1999.

[57] M. Kazemitabaar, R. Ye, X. Wang, A. Z. Henley, P. Denny, M. Craig, and T. Grossman, "Codeaid: Evaluating a classroom deployment of an llm-based programming assistant that balances student and educator needs," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–20.

[58] E. Riese, "Teaching assistants' experiences of lab sessions in introductory computer science courses," in *2018 IEEE Frontiers in Education Conference (FIE)*. IEEE, 2018, pp. 1–5.

[59] U. Demšar, "Data mining of geospatial data: combining visual and automatic methods," Ph.D. dissertation, KTH, 2006.

[60] U. M. Fayyad, G. G. Grinstein, and A. Wierse, *Information visualization in data mining and knowledge discovery*. Morgan Kaufmann, 2002.

[61] G. Costagliola, V. Fuccella, M. Giordano, and G. Polese, "Monitoring online tests through data visualization," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 6, pp. 773–784, 2008.

[62] M. Tissenbaum, C. Matuk, M. Berland, L. Lyons, F. Cocco, M. Linn, J. L. Plass, N. Hajny, A. Olsen, B. Schwendimann *et al.*, "Real-time visualization of student activities to support classroom orchestration." Singapore: International Society of the Learning Sciences, 2016.

[63] OpenAI, "Chatgpt," 2024. [Online]. Available: https://openai.com/chatgpt/

[64] Puppeteer, "Puppeteer — puppeteer," 2024. [Online]. Available: https://pptr.dev/

[65] rsmbl, "Resemble.js," https://github.com/rsmbl/Resemble.js, Sep. 2024.

[66] F. Draxler, L. Hirsch, J. Li, C. Oechsner, S. T. Völkel, and A. Butz, "Flexibility and social disconnectedness: Assessing university students' well-being using an experience sampling chatbot and surveys over two years of covid-19," in *Proceedings of the 2022 ACM Designing Interactive Systems Conference*, ser. DIS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 217–231. [Online]. Available: https://doi-org.proxy.lib.umich.edu/10.1145/3532106.3533537

[67] A. G. Picciano *et al.*, "Beyond student perceptions: Issues of interaction, presence, and performance in an online course," *Journal of Asynchronous learning networks*, vol. 6, no. 1, pp. 21–40, 2002.

[68] A. Y. Ni, "Comparing the effectiveness of classroom and online learning: Teaching research methods," *Journal of public affairs education*, vol. 19, no. 2, pp. 199–215, 2013.

[69] Y. Yang and S. Oney, "Vizcode: A practical real-time tool for in-class computer programming tutoring," in *Proceedings of the Eleventh ACM Conference on Learning @ Scale*, ser. L@S '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 544–546. [Online]. Available: https://doi-org.proxy.library.nd.edu/10.1145/3657604.3664716

[70] GitHub, "Github copilot: Your ai pair programmer," 2024. [Online]. Available: https://github.com/features/copilot/