TutorTrace: A Dataset and Taxonomy for Classifying Learner Behavioral States during AI-Assisted Programming Education

David Barron
Virginia Tech
Blacksburg, Virginia, USA
dbarron410@vt.edu

Xiaohang Tang
Virginia Tech
Blacksburg, Virginia, USA
xiaohangtang@vt.edu

Rezky Dwisantika
Sepuluh Nopember Institute of Technology
Surabaya, Jawa Timur, Indonesia
rezkysantika21@gmail.com

Minsun Kim
Virginia Tech
Blacksburg, Virginia, USA
minsunkim@vt.edu

David H. Smith IV
Virginia Tech
Blacksburg, Virginia, USA
dhsmith4@vt.edu

Jiaming Cui
Virginia Tech
Blacksburg, Virginia, USA
jiamingcui@vt.edu

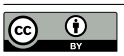
Yan Chen
Virginia Tech
Blacksburg, Virginia, USA
ych@vt.edu

Abstract

AI programming tutors provide scalable support, yet lack the behavioral context human tutors rely on to adapt support to learners' needs. We present TutorTrace, a dataset and behavioral abstraction pipeline that makes learners' behavioral context visible and computable in real time from low-level IDE telemetry. Across four deployments in two introductory Python courses (N=480), TutorTrace captures approximately 180K telemetry events, 13,633 behavioral segments, and 27 continuously computed metrics. From this foundation, we derive a taxonomy of learner activity before the first AI query, between consecutive queries, and across the full session, enabling systems to respond not just to what learners say, but to what they have done leading up to the help-seeking moment. In a preliminary classroom evaluation, behavior-aware prompts were associated with a decrease in intervals between queries with no independent work from 50.0% to 20.7%. As an additional demonstration of downstream utility, we evaluate TutorTrace on two held-out prediction tasks: whether a learner will query within the next 60 seconds (AUROC=.726) and whether an upcoming query reflects guided or dependent help-seeking (AUROC=.717). Together, these findings show how behavioral context can enable adaptive AI tutoring at scale.

Keywords

behavioral analysis, intelligent learning environment, educational technology, student behavior classification, LLM-assisted programming, IDE telemetry, human-AI interaction



UIST '26, Detroit, MI, USA

ACM ISBN 979-8-4007-2856-3/2026/11
https://doi.org/10.1145/3830398.3830712

ACM Reference Format:
David Barron, Xiaohang Tang, Rezky Dwisantika, Minsun Kim, David H. Smith IV, Jiaming Cui, and Yan Chen. 2026. TutorTrace: A Dataset and Taxonomy for Classifying Learner Behavioral States during AI-Assisted Programming Education. In *The 39th Annual ACM Symposium on User Interface Software and Technology (UIST '26), November 02–05, 2026, Detroit, MI, USA.* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/3830398.3830712

1 Introduction

Large Language Models have improved personalized computing education by providing learners with near-instant, in-depth responses built on a knowledge base no single tutor can compete with [34]. Yet, one-on-one human tutoring remains the gold standard for improving learner outcomes [5]. One key reason is that human tutors respond not only to what learners explicitly say, but also to what their observable behaviors suggest they implicitly need [7, 36]. Consider what this could look like at scale.

> Imagine if every student in a 400-person programming course had a tutor sitting beside them, observing what they do and where they struggle. One who knows when to push them harder and when to slow down and offer support. One who could tell the difference between a student who is struggling and one who hasn't put in the effort. Achieving this vision requires systems that can infer learners' evolving behavioral context in real time from fine-grained interaction data, a capability and data resource not yet available to our community.

The human tutor's comparative advantage lies in the behavioral context that precedes the learner's help-seeking moment. Current AI tutoring systems are typically limited to the question itself, missing the struggle or lack thereof that preceded it. To bridge this gap, systems must be able to programmatically detect the behavioral patterns that human tutors observe intuitively.

Table 1: Comparison of TutorTrace with existing CS education and AI-assisted programming datasets: the 2nd CSEDM Data Challenge [9, 10], ProgSnap2 [28], CodeAid [20], and CodeWatcher [3].

Property	CSEDM	ProgSnap2*	CodeAid	CodeWatcher†	TutorTrace (ours)
Real classroom deployment	✓	✓	✓	✗	✓
Longitudinal coverage (semester)	✓	✓	✓	✗	✗
Raw IDE telemetry	✗	✓	✗	✓	✓
AI query & response logs	✗	✗	✓	✗	✓
Task outcome data	✓	✓	✗	✗	✓
Behavioral classification (expert-validated)	✗	✗	✗	✗	✓
N students	819	varies	563	3†	480
N AI interactions	—	—	7,003	—	1,386

*ProgSnap2 is a data format specification; student counts vary by adopting institution.

†CodeWatcher is a telemetry-collection tool (demonstration); N reflects an illustrative use case with three developers, not a released dataset.

This requires datasets that expose fine-grained student programming behavior at scale. Most existing datasets capture only subsets of the learner's programming process, such as static code submissions or AI interaction logs. Few provide the continuous, fine-grained interaction telemetry required to enable full-fidelity replays of learner programming sessions (Table 1). Systems that do capture interaction at this granularity [27, 38] were not built to release classroom-scale datasets.

Prior work shows that the behavioral context surrounding a help-seeking moment is associated with learning outcomes [24], and specific interaction patterns distinguish effective from ineffective AI use [24, 31]. Students who invest independent effort before receiving instruction learn more from it [19, 32], and a student's behavioral context directly informs the tutor's pedagogical strategy for intervening [13]. Yet current AI tutoring systems typically respond to the information learners explicitly provide rather than the behavioral process that preceded it. As a result, two learners with similar code, errors, and questions may receive similar support despite arriving at that moment through fundamentally different processes.

We present TutorTrace, a dataset and automated behavioral abstraction infrastructure designed to make this context visible and computable. The dataset comprises fine-grained IDE telemetry from 480 students across four deployments in two introductory Python courses. An automated pipeline transforms this telemetry into continuously computed observable metrics and behavioral sequences. Unlike existing datasets that capture only what learners submit or ask, TutorTrace captures the story of how they got there, making each learner's evolving behavioral context programmatically accessible in real time.

To demonstrate the utility of TutorTrace for future research and system design, we examine how its representations of behavioral context can support adaptive AI tutoring. First, we derive a three-window behavioral taxonomy that organizes learner activity before the first AI query, between consecutive queries, and through recurring re-querying patterns across the session. Next, we conduct a preliminary comparison examining whether behavior-aware prompting is associated with changes in learners' observable activity between queries. Finally, we demonstrate additional predictive utility through two tasks: predicting whether a learner will initiate an AI interaction within the next 60 seconds and whether an upcoming query will reflect guided or dependent help-seeking. Together, these capabilities enable systems to detect, respond to, and anticipate learner behavior surrounding AI interactions. We make the following contributions:

- **TutorTrace Dataset:** A publicly available dataset[1] from 480 learners across four deployments in two introductory Python courses, comprising approximately 180K fine-grained IDE telemetry events mapped to 13,633 automatically classified behavioral segments, along with 27 continuously computed observable metrics surrounding 1,386 AI interactions.
- **TutorTrace Classifier:** An automated behavioral classifier that transforms raw IDE telemetry into labeled behavioral sequences in real time. The classifier is grounded in expert-developed rules and validated against expert labels and student self-reports, with overall pairwise agreement ranging from 78% to 87%.
- **TutorTrace Taxonomy and Downstream Utility:** We demonstrate how the representations provided by TutorTrace can support adaptive AI tutoring through:
  - A three-window behavioral taxonomy comprising ten profiles that characterize activity before the first AI query, between consecutive queries, and through recurring re-querying patterns across the session.
  - A preliminary classroom comparison in which behavior-aware prompting was associated with a decrease in Passive inter-query windows from 50.0% to 20.7% and greater observable activity between queries.
  - Two held-out prediction tasks demonstrating that observable metrics support prediction of query imminence within 60 seconds (AUROC=.726) and guided versus dependent help-seeking (AUROC=.717).

[1] https://github.com/divadbaroon/TutorTrace_dataset_and_benchmark/tree/uist

2 Related Work

2.1 Prior Taxonomies in CS Education

Taxonomies typically organize distinct, non-overlapping categories across one or more dimensions while also revealing relationships among the taxa [16]. Past studies in CS education have employed educational taxonomies to design learning objectives and assess learning effectiveness [33], though researchers have argued that general-purpose frameworks such as Bloom's taxonomy and SOLO do not fully capture the distinctive characteristics of computer science learning, particularly in programming-related contexts [12].

After the rise of LLMs, behavioral analysis of programming interaction has become more important and complex. Copilot, an LLM-based code generation tool, can produce correct solutions to many introductory programming problems, raising questions about how its presence reshapes student work [35]. In CS education specifically, LLM-based programming assistants support students during coding tasks while also reshaping their interaction patterns, problem-solving processes, and reliance on AI assistance [14, 20, 22]. As a result, programming sessions generate large amounts of human-AI interaction traces not easily described by traditional educational taxonomies alone [2, 27]. To address this gap, Mozannar et al. [27] proposed a taxonomy representing AI-assisted coding sessions as timelines of transitions across programmer activities, while Barke et al. [2] showed that Copilot use also reflects broader interaction modes, such as acceleration and exploration.

Despite these developments, taxonomies specifically designed to characterize student behavior in LLM-assisted programming contexts remain rare. In this work, we introduce a taxonomy of learner behavioral profiles for AI-assisted programming education, aiming to make student–AI interaction patterns more visible and interpretable in educational settings.

2.2 Help-Seeking Quality and AI Dependency

With the advancement of AI, students increasingly rely on them to complete assignments rather than as learning aids, raising concerns about over-reliance and reduced independent problem-solving [17, 23, 37]. Recent work in CS education has developed tools that leverage LLMs for scaffolded hint generation while incorporating guardrails to prevent solution delegation [14, 20, 22]. However, even when system-level constraints limit the model's output, students bypass them when given the option: Kapoor et al. [18] deployed an AI TA with an optional "See Solution" control that disabled the guardrails, and 50% of 885 students used it on at least one problem, with 14% using it on all three.

To mitigate these behaviors, several approaches shape or qualify student queries before they reach the underlying LLM. CodeHelp [22] replaces a single free-form box with structured fields for language, code, error message, and issue description, and runs an LLM-based sufficiency check that returns a clarification request when a query lacks critical information. CodeAid [20] similarly offers feature-specific input templates that scaffold how students frame a request. However, these approaches intervene based on the content of the query itself, not the behavioral context that preceded it. A student who spent five minutes debugging independently before asking and a student who re-queried immediately after the last response may submit identical questions, yet require fundamentally different pedagogical responses. TutorTrace addresses this gap by making the behavioral context surrounding each query visible and computable, enabling interventions grounded not in what the student says, but in what they did before saying it.

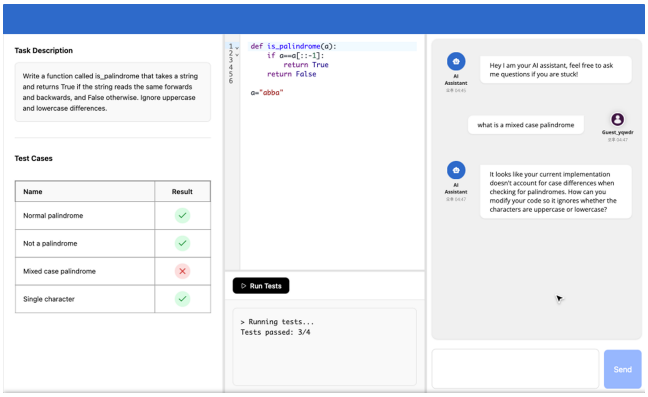


Figure 1: The student interface with task description, code editor, terminal, test cases, and AI chat window.

3 System

TutorTrace is a task-based IDE platform with LLM support, similar in structure to LeetCode[2] and HackerRank[3].These platforms share a fixed layout containing a task description, test cases, code editor, terminal, and AI chat window, constraining student activity to a well-defined interaction space, making fine-grained behavioral observation within this environment both feasible and reproducible. As students complete programming tasks, TutorTrace passively captures and translates low-level telemetry into behavioral sequences and observable metrics in real time, providing programmatic access to the behavioral context that human tutors observe intuitively but that existing AI tutoring systems do not see.

3.1 Telemetry Capture

TutorTrace captures fine-grained user interactions within the interface, recording 37 event types across six source regions (Table 2). Events are captured continuously on the client side and batched to the back end every five seconds with no impact to the student's workflow. Each recorded event includes a millisecond timestamp, source region, and payload. A student's full stream of telemetry events enables a full-fidelity replay of their session in the Behavioral Labeling Interface (Appendix A, Figure 4) and serves as the foundational input to our behavioral abstraction pipeline (Figure 2).

A complete schema of all 37 event types with example payloads is provided in Appendix C, Table 10

3.2 LLM Integration

Our system is integrated with GPT-4o[4], accessible via the chat window panel. The prompt is grounded in prior literature on pedagogical prompting strategies for AI tutoring and is structured

[2]https://leetcode.com/
[3]https://www.hackerrank.com/
[4]https://openai.com/index/hello-gpt-4o/

Table 2: Telemetry events captured by TutorTrace, grouped by source region.

Region	Example Events	Example Payload
Code Editor (11 types)	TYPE, DELETE, PASTE, COPY, CUT, UNDO, REDO, SELECT, INDENT	Characters, cursor position, selection range
Terminal (6 types)	RUN, OUTPUT, ERROR, RESULT, SELECT, COPY	stdout, error type and message, test pass/fail counts
Chat (9 types)	QUERY, RESPONSE, TYPE, DELETE, PASTE, SELECT, COPY	Message content, length, latency, sender ID
Task/Tests (4 types)	SELECT, COPY	Selected text, source region
Global (5 types)	MOUSE_CLICK, TAB_STATE, WINDOW_RESIZE, PANEL_RESIZE, MOUSE_MOVE	Coordinates, active region, tab visibility, panel dimensions
Session (2 types)	START, END	Window dimensions, duration

around three escalating levels of scaffolding [19, 29, 36]. It was iteratively refined through pilot testing until the LLM consistently adhered to the following principles (Appendix D):

(1) **Socratic Questioning:** Guide the student toward the answer through targeted questions rather than direct explanation. Avoid revealing solutions [20].
(2) **Conceptual Hint:** Provide a high-level conceptual nudge identifying the relevant concept or approach without specifying implementation [22].
(3) **Concrete Scaffolding:** Provide direct, specific guidance using pseudocode and blanks when the student has demonstrated sustained effort without progress [19].

4 Behavioral Classification

While prior work often derives behavioral labels from heuristic rules alone, we ground ours in the judgment of four domain-expert annotators, each with teaching and research experience in CS education. To label learner sessions, annotators used the Behavioral Labeling Interface (Appendix A), which presents a replay of the learner's session, a session timeline, and a behavioral codebook for annotating observed behavior directly onto the timeline. Annotators independently coded replays of learner sessions and met weekly to reconcile disagreements in their classifications.

4.1 Codebook Development

We derived our initial behavioral categories from prior literature on programming behavior and help-seeking [4, 27]. We iteratively refined the codebook through 10 pilot labeling sessions drawn from a preliminary deployment in an intermediate Python course at our institution. During this phase, four domain-expert annotators independently segmented and labeled 10 pilot sessions (each 5–13 minutes of learner activity, mean 9 minutes) using the Behavioral Labeling Interface (Appendix A). The annotators met weekly to reconcile disagreements over segment boundaries and behavioral classifications, updating the codebook as needed (Appendix B, Table 9). The finalized codebook, presented in Table 3, served as the basis for the segmentation and classification rules used by our automated behavioral classification pipeline.

Table 3: Behavioral codebook used by domain-expert annotators to label learner sessions. The table lists each behavior, its subtypes and definition, and the corresponding automated segmentation and classification rule derived through annotator consensus except where noted.

Behavior	Subtype	Definition	Derived rule
Implementing	—	Writing new code	`CODE_EVENT` with no unresolved terminal error
Debugging	—	Fixing an error	`CODE_EVENT`s while an unresolved error or failed test exists
Testing†	—	Executing tests	Automatically marked at each `TERMINAL_RUN` event
Thinking	Task	Reading the task	≥3s inactivity before any CODE, TERMINAL, or CHAT EVENT
	Code	Reviewing code	≥6s gap between code edit events
	Error	Reading an error	≥3s inactivity after `TERMINAL_ERROR`
	Pre-query	Formulating query	≤5s pause before `CHAT_QUERY` (stored as metadata)
	Response	Reading AI reply	`CHAT_RESPONSE` when the preceding segment was chat input
Seeking Help	—	Typing query to AI	`CHAT_TYPE`, `CHAT_QUERY` events
Idle	—	No activity	No events of any type for >15s
Off-Topic	—	Unrelated activity	Input unrelated to task
Unknown	—	Cannot classify	Unknown telemetry event

† *Testing was not manually annotated. Because test execution is directly observable, each* `TERMINAL_RUN` *event was automatically inserted as a run marker on the Behavioral Labeling Interface timeline.*

4.2 Behavioral Segmentation

The most challenging aspect of manual annotation was determining precise segment boundaries. Across the pilot sessions, annotators generally agreed on which behavior was occurring but differed in the exact timing of its start and end. The mean discrepancy in boundary placement across annotators was 2.3 seconds. During

Table 4: Pairwise agreement among learner self-reports, expert annotations, and the automated classifier across six user-study sessions. Cohen's κ [8] and raw percentage agreement are reported for each behavior and overall.

	Self vs Expert		Expert vs Auto		Self vs Auto	
Behavior	κ	Agr.	κ	Agr.	κ	Agr.
Implementing	0.93	97%	0.95	98%	0.98	99%
Debugging	0.76	96%	0.88	98%	0.71	94%
Seeking Help	0.90	98%	1.00	100%	0.90	98%
Thinking:						
Task	0.92	96%	0.96	98%	0.95	98%
Code	0.64	87%	0.34	82%	0.32	78%
Error	0.54	87%	0.42	84%	0.42	79%
Pre-query	0.50	97%	1.00	100%	0.50	97%
Response	0.50	89%	1.00	100%	0.50	89%
Overall	0.79	83%	0.83	87%	0.73	78%

weekly reconciliation meetings, we found this stemmed from the cognitive load of continuously watching session replays rather than any conceptual disagreement on what constituted a behavioral transition. Annotators reported that identifying the behavior itself was straightforward, whereas pinpointing the exact time of transition required sustained attention that naturally degraded over longer sessions.

To address this, the first author reviewed all independently annotated boundaries and synthesized them into proposed consensus segment boundaries. To verify these, all annotators then rewatched each session while the first author verbally announced each boundary as it occurred. This reconciliation process yielded consensus on all segment boundaries. We used the resulting consensus boundaries to derive the automated segmentation rules described in Appendix F, Algorithm 1.

4.3 Validation

We validated the pipeline in two stages. First, we established inter-rater reliability among human annotators across the 10 pilot sessions to confirm the codebook was sufficiently stable before scaling. Second, we conducted a user study with 13 participants who completed two Python programming tasks and then self-labeled their behavior using the Behavioral Labeling Interface. Our domain-expert annotators independently labeled six of these sessions, allowing us to triangulate agreement across three sources: learner self-reports, expert annotations, and the automated classifier. Testing was excluded from the agreement analysis because test executions were directly observed from `TERMINAL_RUN` events rather than independently annotated. Across the three pairwise comparisons, overall Cohen's κ ranged from 0.73 to 0.83, and overall raw agreement ranged from 78% to 87% (Table 4).

We also found several recurring edge cases that raw IDE telemetry could not differentiate alone. For instance, after receiving an error, a learner might ignore it and continue implementing a separate feature, making *Implementing* difficult to distinguish from *Debugging*. In the same vein, *Thinking about Code* may be difficult to distinguish from *Thinking about Error*. For these situations, semantic understanding is needed to differentiate the behavior. We discuss further in Section 9.

Table 5: The 35 candidate observable metrics grouped by observable area, drawn from prior IDE-based learning analytics literature [4, 27]. Metrics marked † were excluded during pruning.

Observable Area	Metrics	
Code activity	`code_edits`	`code_edit_rate`
	`chars_inserted`	`chars_deleted`
	`code_deletes`	`net_code_growth`
	`delete_type_ratio`	`code_pastes`†
Terminal activity	`terminal_runs`	`terminal_errors`
	`max_consecutive_errors`	`mean_time_between_runs_s`
Error recovery	`error_self_fix`	`error_reading_time_s`
	`error_to_edit_s`	`failed_test_to_edit_s`
	`failed_test_self_fix`	
Time distribution	`time_in_editor_s`	`time_in_terminal_s`
	`time_in_chat_s`	`time_in_task_s`
	`time_in_tests_s`	`longest_idle_s`
	`tab_hidden_time_s`†	
Chat behavior	`thinking_time_s`	`seeking_help_time_s`
	`duration_s`	`response_reading_time_s`
	`chat_to_code_latency_s`	
Interface events	`tab_switches`†	`copy_events`†
	`paste_events`†	`undo_events`†
	`redo_events`†	`select_events`†

† Excluded during pruning: sparse metrics (> 80% zeros) and metrics capturing activity outside the instrumented workspace.

5 Observable Metrics

While behavioral sequences capture the temporal progression of learner behavior, they do not provide insights into the learners' aggregate activity within these temporal windows, such as the frequency of code edits, terminal runs, and character deletions. To capture this aggregate activity, we compute observable metrics that convert raw IDE telemetry into intuitive measures of the amount, frequency, and distribution of learner activity within a specific temporal window. Together, behavioral sequences and observable metrics provide complementary perspectives on the temporal evolution and aggregate characteristics of learner behavior within the programming environment.

We compute 35 candidate observable metrics drawn from prior IDE-based learning analytics literature (Table 5). These metrics capture complementary dimensions of code activity, terminal activity, error recovery, time distribution, chat behavior, and interface events. After excluding sparse metrics (> 80% zeros) and metrics capturing activity outside the instrumented workspace, 27 observable metrics remain. Together, the retained metrics provide general-purpose aggregate representations of learner activity.

6 Deployments and Dataset

We deployed TutorTrace across four deployments in two introductory Python courses at our institution (Table 6). Deployments 1 and 2 were conducted with Instructor A using a task focused on list indexing, slicing, and in-place manipulation with a 15-minute time limit. Deployments 3 and 4 were conducted with Instructor B

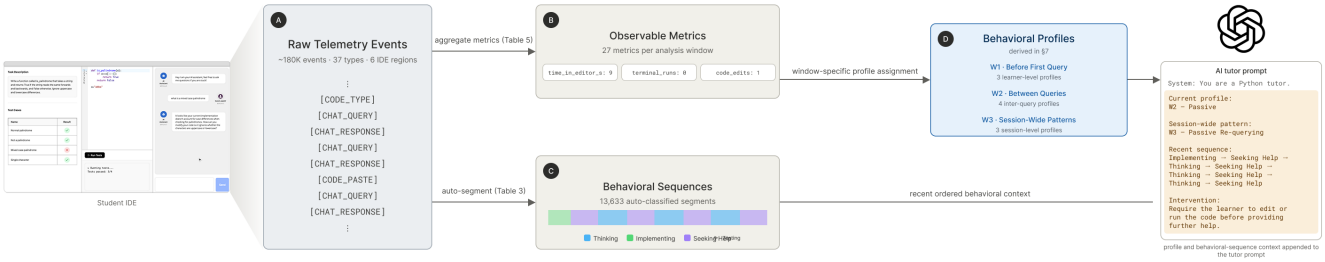


Figure 2: TutorTrace behavioral abstraction pipeline. Raw telemetry events (A) are aggregated into window-specific observable metrics (B) and auto-segmented into behavioral sequences (C). Observable metrics support assignment to the behavioral profiles derived in the TutorTrace Taxonomy (D), while behavioral sequences preserve recent ordered activity. These representations can be incorporated into an AI tutor prompt to support behavior-aware adaptation.

using a task focused on nested lists and iteration. Full task descriptions and test cases are provided in Appendix I. Due to scheduling constraints in the second course with Instructor B, students in Deployments 3 and 4 completed the task under a 10-minute time limit rather than the 15 minutes used in Deployments 1 and 2, which may have contributed to the lower completion rates in those sessions.

The predictive models reported in Section 7.4.5 were trained on Deployment 1 (Instructor A, morning, $n = 190$) and evaluated on held-out Deployment 2 (Instructor A, afternoon, $n = 113$), providing a cross-cohort evaluation under the same instructor and task but at a different time of day. Deployments 3 and 4 supported a preliminary classroom evaluation examining whether behaviorally aware prompts could support changes in learner behavior at scale (Section 7.3). Deployment 3 (Instructor B, morning, $n = 70$) served as the baseline condition, while Deployment 4 (Instructor B, afternoon, $n = 107$) served as the intervention condition, in which the AI tutor's prompts were informed by each learner's behavioral profile.

Table 6: Deployment context across four sessions.

Deployment	1	2	3	4
Instructor	A	A	B	B
Time limit	15 min	15 min	10 min	10 min
Task	Playlist	Playlist	Grade Book	Grade Book
Total students	190	113	70	107
Used AI	94	90	48	85
AI Interactions	428	540	190	228
Task completion	87.2%	82.3%	34.7%	44.3%

Together, these deployments yield the TutorTrace foundation dataset, comprising approximately 180K raw telemetry events, 27 continuously computed observable metrics, and 13,633 automatically classified behavioral sequences (Layers A–C in Figure 2).

7 The TutorTrace Taxonomy

While our real-time behavioral abstraction pipeline enables systems to be behaviorally aware, making this awareness actionable requires identifying the recurring behavioral signatures that are both distinct and associated with differences in task outcomes. We're interested in both distinct behavioral signatures and task outcomes because learners who are performing poorly with high effort, and those that are performing well with low effort, each require fundamentally different pedagogical responses. We therefore derive the TutorTrace Taxonomy, which organizes learner activity surrounding AI help-seeking into recurring behavioral profiles, their respective outcomes, and expert-informed interventions that are provided to the AI tutor's prompt.

7.1 Taxonomy Construction

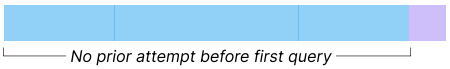
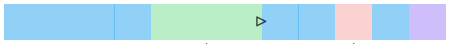
We organize the analysis across three temporal windows, each capturing a distinct stage of the help-seeking cycle: learner activity before the first AI query (Window 1), activity between consecutive queries (Window 2), and recurring inter-query patterns across the session (Window 3). We truncate each session at its first all-pass test result, which marks the end of task-directed activity and prevents post-completion behavior from influencing the profiles. For learners who do not complete the task, we retain activity through the observed session end. We also exclude individual windows containing more than 30 seconds of tab-hidden time to limit the influence of unobserved off-platform activity

Windows 1 and 2 follow an outcome-guided clustering procedure intended to identify behaviorally distinct profiles that also differ meaningfully in task outcomes. We first separate observations with no observable programming effort, defined as zero code edits and zero terminal runs, into rule-defined profiles. These observations represent a qualitatively distinct state rather than simply a low-activity version of active behavior, and including them in K-means could cause the zero-inflated observations to dominate the resulting clusters.

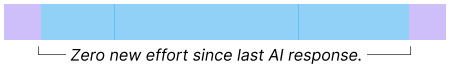
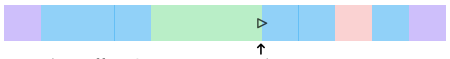
For the remaining active observations, we construct a window-specific candidate metric pool so that clustering uses only measures meaningful within that stage of the help-seeking cycle. We exclude metrics that are inapplicable to the window or zero-valued in more than 80% of observations because they provide little discriminative information and may produce unstable or artifact-driven clusters.

We exhaustively evaluate every three-metric combination from the resulting candidate pool. For each combination, we standardize the metrics and apply K-means [26] with $K \in \{2, 3\}$ using 10

Window 1. Before First Query

Profile Description	Intervention	N(%)	Think	Runs	Err.	Comp.
Cold Start  *No prior attempt before first query*	Scaffold task entry. Offer pseudocode with blanks. End with a "Try:" starter.	31 (20%)	72 s	0.0	0.0	97%
Oriented  *Made a genuine attempt, moderated code and effort*	Acknowledge the attempt. Unblock the specific error or gap.	100 (66%)	98 s	1.2	0.7	92%
Struggling  *Extensive effort but no resolution with high runs, errors, edits*	Diagnose the misconception. Explain why the approach fails. Give a precise redirect.	21 (14%)	165 s	4.2	2.5	67%

Window 2. Between Queries

Profile Description	Intervention	N(%)	Think	Runs	Err.	Comp.
Passive  *Zero new effort since last AI response.*	Try a different angle. Require student action before answering.	318 (46%)	18 s	0.0	0.0	81%
Iterating  *Light effort One run, some edits, low error count. Steady progress.*	Validate the approach. Nudge toward the next step	290 (42%)	31 s	1.0	0.5	84%
Debugging  *Multiple runs and errors*	Recognize the effort. Name the error. Escalate scaffolding	56 (8%)	57 s	3.7	2.8	79%
Spinning  *High thinking and edits but barely tests. Writing without executing.*	Interrupt the loop. Direct to run code now. Explain why and how testing reveals the issue.	22 (3%)	149 s	1.8	0.6	50%

Window 3. Session-Wide Behavioral Patterns

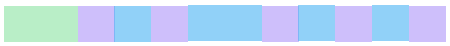
Profile Description	Intervention	N(%)	Passive	Tested	Comp.
Passive Re-querying  *Repeatedly re-queries with little observable coding or execution.*	Require a concrete action before another response, such as editing or running the code.	47 (42%)	70%	22%	79%
Active Testing  *Regularly edits and executes code between queries.*	Use the latest test result to identify the specific error and guide the next step.	31 (28%)	20%	77%	81%
Untested Editing  *Writes code but often skips testing before asking*	Prompt the learner to run the current code and inspect the result before further explanation.	34 (30%)	35%	23%	88%

Figure 3: The TutorTrace three-window behavioral taxonomy. Each window captures a distinct moment in the help-seeking cycle, characterized by different behavioral signals and requiring different intervention strategies.

initializations. We discard solutions containing fewer than 15 observations in any cluster. For each remaining solution, we calculate its silhouette score [30] and the spread between its highest and lowest task-completion rates. Candidate solutions are ranked according to:

$$\text{score} = \text{silhouette} \times \text{completion-rate spread}.$$

We perform clustering using behavioral metrics only. Completion rates are used to select among candidate solutions that produce behaviorally distinct groups. Because completion informs the selection of the final solution, the resulting differences in completion rates across clusters are descriptive rather than confirmatory. After selecting a solution, we assign profile names based on the behavioral patterns represented by each cluster centroid.

7.1.1 Window 1: Before the First Query. In Window 1 each learner contributes one set of aggregate metrics capturing their activity before their first query. Learners with zero code edits and zero terminal runs during this period are assigned to the rule-defined Cold Start profile (n=31). The remaining 121 active learners are included in the exhaustive search.

The highest-ranked solution uses `time_in_editor_s`, `time_in_terminal_s`, and `time_in_chat_s` at K=2 (s=0.420; completion-rate spread=25.3 percentage points). Based on their behavioral centroids, the resulting clusters are labeled Oriented (n=100) and Struggling (n=21). Together with Cold Start, these profiles distinguish initial help-seeking that follows no observable programming attempt, a moderate active attempt, or an extended period of unsuccessful effort.

7.1.2 Window 2: Between Consecutive Queries. In Window 2, each interval between consecutive AI queries contributes one set of aggregate metrics capturing the learner's activity after one AI response and before their next query. Learners may therefore contribute multiple Window 2 observations and may exhibit different profiles at different points in a session.

Inter-query windows with zero code edits and zero terminal runs following the preceding AI response are assigned to the rule-defined Passive profile (n=318). We combine eligible post-response and subsequent-effort metrics into one candidate pool and apply the shared exhaustive search to the remaining 368 active windows. Each interval inherits the task-completion outcome of its corresponding session.

The highest-ranked solution uses `time_in_editor_s`, `thinking_time_s`, and `error_self_fix` at K=3 (s=0.547; completion-rate spread=33.8 percentage points). Based on their centroids, the resulting clusters are labeled Iterating (n=290), Debugging (n=56), and Spinning (n=22). Together with Passive, these profiles distinguish repeated queries made without observable programming activity from queries following brief iteration, active error recovery, or prolonged activity with limited execution.

7.1.3 Window 3: Session-Wide Re-querying Patterns. Whereas Window 2 characterizes individual inter-query moments, Window 3 summarizes whether particular forms of inter-query behavior recur across the session. We aggregate each learner's valid Window 2 intervals and retain learners with at least two such intervals, ensuring that the representation captures a recurring pattern rather than a single observation. This yields 112 eligible learners.

We assign each inter-query interval to one of three mutually exclusive behavioral categories: Passive, containing no code edits or terminal runs; Tested, containing at least one terminal run; or Active Untested, containing code edits but no terminal run. We then represent each learner using four session-level measures: the proportion of Passive intervals, the proportion of Tested intervals, the proportion of Active Untested intervals, and the longest consecutive Passive streak normalized by the learner's number of valid intervals.

We standardize these four measures and apply K-means for $K \in \{2, 3, 4, 5, 6\}$ using 100 initializations. As in the preceding windows, we exclude solutions containing fewer than 15 learners in any cluster. Unlike Windows 1 and 2, task completion is not used for either clustering or model selection; we select the eligible solution with the highest silhouette score.

The highest silhouette score occurs at K=3 (s=0.399), producing the Passive Re-querying (n=47), Active Testing (n=31), and Untested Editing (n=34) profiles. The assignments are identical across 20 random seeds (ARI [15]=1.000).

7.2 Characteristics of Behavioral Profiles

The three windows capture complementary aspects of AI help-seeking: activity before a learner's first query, activity after an AI response and before the learner's next query, and recurring re-querying patterns across the session. Because task completion informed the exploratory selection of the Window 1 and Window 2 clustering solutions, their completion rates are reported as descriptive characteristics rather than as independent evidence of profile validity. Task completion did not inform the construction or selection of the Window 3 profiles; its completion rates are likewise reported descriptively.

Window 1: Before the First Query. Cold Start learners comprised 20% of the Window 1 population (n=31). These learners made no code edits or terminal runs before their first query, although their pre-query periods averaged 72 seconds. The corresponding sessions had a 97% completion rate. Oriented learners (n=100, 66%) exhibited moderate pre-query activity and had a 92% completion rate. In contrast, Struggling learners (n=21, 14%) spent substantially longer before querying, averaging 165 seconds, 4.2 terminal runs, and 2.5 errors. Their corresponding sessions had a 67% completion rate.

These profiles complicate the assumption that greater pre-query effort necessarily signals better progress. In which, Cold Start learners completed the task at high rates despite making no observable programming attempt, whereas Struggling learners invested the most time and encountered the most errors but completed at the lowest rate. Rather than implying that low effort is beneficial, this contrast suggests different pedagogical needs: Cold Start learners may require prompting to engage independently, while Struggling learners may have exhausted their current strategies and require more direct scaffolding. An adaptive tutor should therefore respond not only to the amount of prior activity, but also to what that activity indicates about the learner's progress.

Window 2: Between Queries. Passive behavior accounted for 46% of valid inter-query windows (n=318). In these windows, learners issued another query without editing or executing their code after

receiving the preceding AI response. The corresponding sessions had an 81% completion rate.

Among active windows, Iterating was the most common profile (n=290, 42%). These windows involved relatively brief thinking periods and modest execution activity, and their corresponding sessions had an 84% completion rate. Debugging windows (n=56, 8%) involved more frequent execution and error recovery, averaging 3.7 terminal runs and 2.8 errors. Their corresponding sessions had a 79% completion rate. Spinning windows (n=22, 3%) involved substantially longer thinking periods, averaging 149 seconds, but comparatively limited execution activity. Their corresponding sessions had the lowest completion rate among the Window 2 profiles at 50%.

Because learners may contribute multiple inter-query windows, these percentages characterize help-seeking moments rather than fixed learner types. Together, the profiles distinguish repeated queries made without observable programming activity from queries following iteration, active debugging, or prolonged activity with limited execution. These contexts could help an adaptive tutor determine whether to prompt independent work, support error recovery, or help a learner move beyond an unproductive pattern.

Window 3: Session-Wide Re-querying Patterns. Passive Re-querying learners comprised 42% of the Window 3 population (n=47). On average, 70% of their valid inter-query intervals contained no code edits or terminal runs, 22% included code execution, and 8% included editing without execution. The corresponding sessions had a 79% completion rate.

Active Testing learners comprised 28% of the population (n=31). Their sessions were characterized by regular code execution: on average, 77% of their inter-query intervals included at least one terminal run, while 20% were Passive and 4% involved editing without execution. The corresponding sessions had an 81% completion rate.

Untested Editing learners comprised the remaining 30% (n=34). On average, 43% of their inter-query intervals included code edits without a subsequent terminal run, compared with 35% Passive intervals and 23% Tested intervals. The corresponding sessions had an 88% completion rate.

Whereas Window 2 characterizes behavior within individual inter-query intervals, Window 3 summarizes which forms of inter-query activity recur across a learner's session. Together, the three windows provide complementary context about when learners request AI assistance, what they do between requests, and whether those behaviors persist across the session.

7.3 Preliminary Evaluation

TutorTrace enables systems to detect and understand behavioral context in real time. The natural next question is whether acting on this information is associated with changes in learners' observable behavior. We therefore deployed the baseline system in a morning session (Deployment 3, n=48 eligible AI users) and a treatment version in an afternoon session (Deployment 4, n=85 eligible AI users) for the same course and task.

In the treatment condition, the system appended a behavior-aware intervention block selected from the learner's recent activity to the AI tutor's prompt at each interaction (Appendix G). We subsequently applied the revised Window 2 taxonomy to both deployments to examine whether their profile distributions and observable activity differed. This preliminary evaluation does not measure learning gains or establish a causal effect.

Passive inter-query windows accounted for 50.0% of valid windows in the baseline deployment and 20.7% in the intervention deployment (−29.3 pp). Iterating windows accounted for 39.1% and 56.0%, respectively (+16.9 pp). The overall Window 2 profile distribution differed between deployments ($\chi^2(3)$=27.55, p<.0001). Because learners could contribute multiple windows, we also compared each learner's proportion of Passive windows. This learner-level comparison was likewise significant (U=1533, p<.0001).

The intervention deployment also exhibited more code edits per valid inter-query window (23.7 vs. 11.7), more terminal runs (2.1 vs. 1.0), and longer periods of activity between queries (74.9 s vs. 47.0 s). Completion among eligible AI users was 33.3% in the baseline deployment and 43.5% in the intervention deployment. This difference was not statistically significant (p=.273) and remains confounded by the between-session design.

These exploratory findings indicate that behavior-aware prompting was associated with less Passive re-querying and greater observable activity between queries. However, the nonrandomized, between-session design does not establish that the intervention caused these differences.

7.4 Demonstration of Downstream Utility: Prediction Tasks

The preliminary evaluation demonstrates how behavioral profiles can support reactive adaptation during an AI interaction. We next examine whether the telemetry and behavioral abstractions provided by TutorTrace support two additional predictive decisions: *query imminence*, whether a learner will submit an AI query within the next 60 seconds, and *help-seeking type*, whether an upcoming query reflects guided or dependent help-seeking. Guided queries articulate a specific need, question, or concern, whereas dependent queries offload identifying the problem or determining the next step to the AI.

Together, predicting when a learner is likely to query and whether that query will be dependent could allow systems to intervene preemptively and encourage behaviors associated with more guided help-seeking. For example, 96.7% of Cold Start queries were dependent, compared with 30.9% and 28.6% for Oriented and Struggling learners, respectively (Table 8).

7.4.1 Prediction Tasks. For **query-imminence prediction**, the system maintains a 30-second observation window that advances in 5-second steps. Each window is labeled according to whether the learner submits an AI query within the following 60 seconds.

For **help-seeking-type prediction**, each query is paired with the learner's activity during the 15 seconds before submission. The model predicts whether the query is guided, articulating a specific goal, concept, problem, or misunderstanding, or dependent, offloading identification of the problem or next step through a vague request, completion directive, or code without a specific question. This distinction parallels adaptive and unproductive help-seeking in prior tutoring research [1, 25].

We use GPT-4o to label all queries offline as guided or dependent. For each query, the model receives the query text, the learner's

Table 7: Held-out AUROC for query imminence (query within 60 seconds) and help-seeking type (guided vs. dependent within 15 seconds) across nested feature representations. Models were trained on Deployment 1 and evaluated on Deployment 2 (n=113). Δ is relative to raw telemetry; bold indicates the best result.

		Raw Telemetry		+Observable Metrics		+Behavioral Sequences	
Prediction Task	**Description**	AUROC	Δ	AUROC	Δ	AUROC	Δ
Query imminence	Will the learner submit an AI query within the next 60 seconds?	0.689	—	**0.726**	+0.037	0.719	+0.030
Help-seeking type	Will the learner's upcoming query reflect guided or dependent help-seeking?	0.690	—	**0.717**	+0.027	0.705	+0.015

Table 8: Guided and dependent help-seeking by behavioral profile. Percentages are calculated within profiles.

Profile	Labeled n	Guided	Dependent
Window 1: Before the First Query			
Cold Start	30	3.3%	**96.7%**
Oriented	97	**69.1%**	30.9%
Struggling	21	**71.4%**	28.6%
Window 2: Between Queries			
Passive	318	25.8%	**74.2%**
Iterating	290	32.4%	**67.6%**
Debugging	56	35.7%	**64.3%**
Spinning	22	36.4%	**63.6%**

code at submission, and the preceding chat history. The full labeling prompt is provided in Appendix H. Two human raters independently classified 97 queries and achieved substantial agreement (κ=.897) [21]. Agreement between GPT-4o and the two raters was κ=.709 and κ=.690, respectively.

For the prediction tasks, semantic labeling inputs were excluded. Both tasks omit query-composition events (`CHAT_TYPE`, `CHAT_DELETE`, `CHAT_PASTE`, and `CHAT_QUERY`), query text, source-code content, chat history, and the subsequent AI response. This prevents query-imminence models from detecting query composition, while help-seeking-type windows end immediately before submission. The models therefore rely only on preceding behavioral telemetry to predict when help-seeking will occur and what form it will take.

7.4.2 Feature Representations. For both tasks, we compare three nested feature representations to evaluate whether the higher-level abstractions produced by TutorTrace provide predictive value beyond raw telemetry. Features are computed strictly within each task's observation window: 30 seconds for query imminence and 15 seconds for help-seeking type.

- **Raw telemetry**: counts of each telemetry event type within the task-specific window.
- **+Observable metrics**: adds the observable metrics of Section 5, computed within the same window.
- **+Behavioral sequences**: adds features derived from the auto-classified behavioral sequences: time in each behavior, the current and preceding behavior, and counts of transitions between behaviors.

7.4.3 Experimental Protocol. For each task and feature representation, we train a Random Forest classifier using Deployment 1 (Instructor A, morning, n=190) and evaluate it on held-out Deployment 2 (Instructor A, afternoon, n=113; Table 6). Deployment 2 instances are excluded from model fitting and hyperparameter selection.

We report area under the receiver operating characteristic curve (AUROC) [11] as the primary measure of predictive discrimination. AUROC measures how consistently a model ranks positive instances above negative instances across classification thresholds.

7.4.4 Profiles and Help-Seeking Type. Table 8 reports guided and dependent help-seeking across behavioral profiles. In Window 1, 96.7% of labeled first queries from Cold Start learners were dependent. In contrast, first queries from Oriented and Struggling learners were more often guided (69.1% and 71.4%, respectively). Learners who queried without first editing or executing code therefore exhibited a markedly different help-seeking distribution from those who engaged in observable programming activity before their first query.

Differences were less pronounced in Window 2. Queries following all four inter-query profiles were more often dependent than guided, with rates ranging from 63.6% for Spinning to 74.2% for Passive. Because learners may contribute multiple Window 2 observations, these percentages describe profile–query pairs rather than independent learner groups. We therefore interpret them as descriptive associations between the activity preceding a query and the form of help-seeking that follows.

7.4.5 Prediction Results. Table 7 reports held-out AUROC across the three nested feature representations. For query-imminence prediction, raw telemetry achieved an AUROC of 0.689, while the full representation including behavioral-sequence features achieved 0.719. Observable metrics alone performed best at 0.726, an absolute improvement of 0.037 over raw telemetry.

A similar pattern emerged for help-seeking-type prediction. Raw telemetry achieved an AUROC of 0.690, while the full representation including behavioral-sequence features achieved 0.705. Observable metrics alone performed best at 0.717, an absolute improvement of 0.027 over raw telemetry.

Across both tasks, observable metrics provided the largest gain over raw event counts. Behavioral-sequence features remained above raw telemetry but did not improve on observable metrics in this Random Forest [6] evaluation. Together, these results suggest

that window-level summaries of learner activity capture useful signal about both the timing and form of help-seeking; Appendix J reports window-size sensitivity, complete feature contributions, and profile distributions.

8 Discussion

Our findings motivate a central design question: when an AI tutor can see that a learner has been consistently passive, anticipate that a query may be approaching, and estimate that the request will reflect dependent help-seeking, how should it respond? TutorTrace provides a foundation for educational system designers to build around this question. By making learner behavior observable and computable in real time, TutorTrace enables a new class of support that guides learners toward the behavioral states where productive AI use can emerge as a natural byproduct rather than as an explicit goal.

Prior approaches to AI overreliance have emphasized AI literacy [24], restrictions on system use, and guardrails on model responses [14, 18, 20, 22]. Our findings suggest a complementary direction: supporting the learner's behavioral process. In which, rather than asking only how students should be taught to use AI, we ask how can adaptive systems support the behaviors that make productive help-seeking a likely outcome.

8.1 Design Implications

Interpret a query through the behavior that preceded it. In Window 1, 96.7% of labeled first queries from Cold Start learners were dependent, compared with 30.9% for Oriented and 28.6% for Struggling learners. Although these associations do not establish causality, they demonstrate that the query alone provides an incomplete account of the learner's needs. Tutors should incorporate evidence of prior editing, execution, thinking, and error recovery when deciding how to respond.

Scaffold the learners behaviors. Learners who repeatedly query without editing or executing code may benefit from an intervention that prompts a concrete action before further assistance, whereas learners who edit without testing may benefit from being prompted to run and inspect their code. In our preliminary comparison, behavior-aware prompting was associated with a reduction in Passive windows from 50.0% to 20.7% and increased observable activity between queries. Because this comparison was nonrandomized, it provides initial evidence rather than a causal estimate.

Match support to the learner's demonstrated effort. The taxonomy suggests that uniform scaffolding is unlikely to serve every learner equally. A Cold Start learner may require orientation toward an initial step, a Passive learner may need encouragement to act independently, and a learner engaged in extended unsuccessful effort may require more direct support. The goal is therefore not simply to restrict assistance, but to provide the level and form of support appropriate to the learner's current behavioral context.

8.2 Broader Implications

Toward behaviorally aware learning systems. This work enables a new class of behaviorally aware systems that respond not only to what learners say, but also to how they have worked leading up to a help-seeking moment. By incorporating this process-level context, such systems can create conditions for learners to succeed by preserving productive struggle, supporting learner agency, and providing scaffolding appropriate to their recent behavior.

9 Limitations and Future Work

Our data were collected during short introductory programming tasks at a single institution, limiting generalizability across courses, tasks, populations, and learning environments. The preliminary evaluation compared a morning baseline deployment with an afternoon intervention deployment without random assignment. Cohort composition, time of day, or other unmeasured factors may therefore contribute to the observed behavioral differences. Future work should evaluate the taxonomy and interventions across institutions using randomized studies.

The Window 1 and Window 2 profiles were selected through an exploratory procedure partly informed by task-completion differences. Their completion rates should therefore be interpreted as descriptive characteristics rather than independent validation. Guided and dependent help-seeking labels were generated by GPT-4o and validated against two human raters on a subset of queries. Despite substantial agreement, these labels remain approximations rather than direct measurements of learners' cognitive engagement.

Our system also uses a single tutor model, GPT-4o. Behavioral patterns and intervention effects may differ across models with different capabilities, response styles, or scaffolding strategies. Future work should test whether the profiles and prediction tasks remain stable across models.

Finally, the rule-based auto-segmentation pipeline lacks semantic understanding of learner code and intent. For example, it may classify edits as Debugging while an unresolved error remains even when the learner is implementing an unrelated feature. Future work should combine behavioral telemetry with lightweight semantic code analysis and test whether reduced Passive re-querying improves knowledge retention, transfer, and long-term help-seeking behavior.

10 Conclusion

We presented TutorTrace, a dataset and real-time behavioral abstraction pipeline that makes the context surrounding learner–AI interactions computable. Across 480 learners in four introductory Python deployments, TutorTrace captures approximately 180K telemetry events, 13,633 auto-classified behavioral segments, and 27 observable metrics validated against expert labels and student self-reports.

We derived a three-window taxonomy characterizing activity before the first AI query, between consecutive queries, and across the session. In a preliminary between-deployment evaluation, behavior-aware prompting was associated with a reduction in Passive inter-query windows from 50.0% to 20.7%. Observable metrics also improved held-out AUROC from 0.689 to 0.726 for query imminence and from 0.690 to 0.717 for help-seeking type.

We release the TutorTrace dataset, taxonomy, classifier, and code through the repository linked in the Introduction to support AI tutors that respond not only to what learners say, but also to what they do.

References

[1] Vincent Aleven, Ido Roll, Bruce M McLaren, and Kenneth R Koedinger. 2016. Help helps, but only so much: Research on help seeking with intelligent tutoring systems. *International Journal of Artificial Intelligence in Education* 26, 1 (2016), 205–223.

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

[3] Manaal Basha, Aimeê M Ribeiro, Jeena Javahar, Cleidson RB De Souza, and Gema Rodríguez-Pérez. 2025. Codewatcher: Ide telemetry data extraction tool for understanding coding interactions with llms. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 935–939.

[4] Paulo Blikstein. 2011. Using learning analytics to assess students' behavior in open-ended programming tasks. In *Proceedings of the 1st International Conference on Learning Analytics and Knowledge*. 110–116.

[5] Benjamin S Bloom. 1984. The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational researcher* 13, 6 (1984), 4–16.

[6] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[7] Michelene TH Chi, Stephanie A Siler, Heisawn Jeong, Takashi Yamauchi, and Robert G Hausmann. 2001. Learning from human tutoring. *Cognitive Science* 25, 4 (2001), 471–533.

[8] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46.

[9] CSEDM Data Challenge Organizers. 2021. The 2nd CSEDM Data Challenge. https://sites.google.com/asu.edu/csedm-ws-lak-2019/home?authuser=0 CodeWorkout CS1 dataset, Spring and Fall 2019 semesters, in ProgSnap2 format.

[10] Stephen H. Edwards and Krishnan Panamalai Murali. 2017. CodeWorkout: Short Programming Exercises with Built-in Data Collection. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) *(ITiCSE '17)*. Association for Computing Machinery, New York, NY, USA, 188–193. doi:10.1145/3059009.3059055

[11] Tom Fawcett. 2006. An introduction to ROC analysis. *Pattern recognition letters* 27, 8 (2006), 861–874.

[12] Ursula Fuller, Colin G Johnson, Tuukka Ahoniemi, Diana Cukierman, Isidoro Hernán-Losada, Jana Jackova, Essi Lahtinen, Tracy L Lewis, Donna McGee Thompson, Charles Riedesel, et al. 2007. Developing a computer science-specific learning taxonomy. *ACm SIGCSE Bulletin* 39, 4 (2007), 152–170.

[13] Emma Harvey, Allison Koenecke, and Rene F Kizilcec. 2025. " Don't Forget the Teachers": Towards an Educator-Centered Understanding of Harms from Large Language Models in Education. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. 1–19.

[14] Xinying Hou, Zihan Wu, Xu Wang, and Barbara J Ericson. 2024. Codetailor: Llm-powered personalized parsons puzzles for engaging support while learning programming. In *Proceedings of the Eleventh ACM Conference on Learning@ Scale*. 51–62.

[15] Lawrence Hubert and Phipps Arabie. 1985. Comparing partitions. *Journal of Classification* 2, 1 (1985), 193–218. doi:10.1007/BF01908075

[16] Jeff Irvine. 2021. Taxonomies in education: Overview, comparison, and future directions. *Journal of Education and Development* 5, 2 (2021), 1.

[17] Gregor Jošt, Viktor Taneski, and Sašo Karakatič. 2024. The impact of large language models on programming education and student learning outcomes. *Applied Sciences* 14, 10 (2024), 4115.

[18] Amanpreet Kapoor, Paul Denny, Leo Porter, Stephen MacNeil, and Marc Diaz. 2026. Exploring Student Behaviors and Motivations when using AI Teaching Assistants with Optional Guardrails. In *Proceedings of the 28th Australasian Computing Education Conference*. 22–31.

[19] Manu Kapur. 2014. Productive failure in learning math. *Cognitive Science* 38, 5 (2014), 1008–1022.

[20] Majeed Kazemitabaar, Runlong Ye, Xiaoning Wang, Austin Zachary Henley, Paul Denny, Michelle Craig, and Tovi Grossman. 2024. Codeaid: Evaluating a classroom deployment of an llm-based programming assistant that balances student and educator needs. In *Proceedings of the 2024 chi conference on human factors in computing systems*. 1–20.

[21] J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *biometrics* (1977), 159–174.

[22] Mark Liffiton, Brad E Sheese, Jaromir Savelka, and Paul Denny. 2023. Codehelp: Using large language models with guardrails for scalable support in programming classes. In *Proceedings of the 23rd Koli calling international conference on computing education research*. 1–11.

[23] Chang Liu, Qinyi Zhou, Xinjie Shen, Xingyu Bruce Liu, Tongshuang Wu, and Xiang 'Anthony' Chen. 2026. Behavioral Indicators of Overreliance During Interaction with Conversational Language Models. In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems* (Barcelona, Spain) *(CHI '26)*. Association for Computing Machinery, New York, NY, USA, Article 790, 23 pages. doi:10.1145/3772318.3790332

[24] Qianou Ma, Kenneth R Koedinger, and Tongshuang Wu. 2026. Not Everyone Wins with LLMs: Behavioral Patterns and Pedagogical Implications for AI Literacy in Programmatic Data Science. In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems (CHI '26)*. Association for Computing Machinery, New York, NY, USA, Article 139, 22 pages. doi:10.1145/3772318.3791283

[25] Samiha Marwan, Anay Dombe, and Thomas W. Price. 2020. Unproductive Help-seeking in Programming: What it is and How to Address it. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (Trondheim, Norway) *(ITiCSE '20)*. Association for Computing Machinery, New York, NY, USA, 54–60. doi:10.1145/3341525.3387394

[26] James B McQueen. 1967. Some methods of classification and analysis of multivariate observations. In *Proc. of 5th Berkeley Symposium on Math. Stat. and Prob.* 281–297.

[27] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2024. Reading between the lines: Modeling user behavior and costs in AI-assisted programming. In *Proceedings of the 2024 CHI conference on human factors in computing systems*. 1–16.

[28] Thomas W Price, David Hovemeyer, Kelly Rivers, Ge Gao, Austin Cory Bart, Ayaan M Kazerouni, Brett A Becker, Andrew Petersen, Luke Gusukuma, Stephen H Edwards, et al. 2020. Progsnap2: A flexible format for programming process data. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. 356–362.

[29] Ido Roll, Vincent Aleven, Bruce M McLaren, and Kenneth R Koedinger. 2011. Improving students' help-seeking skills using metacognitive feedback in an intelligent tutoring system. *Learning and Instruction* 21, 2 (2011), 267–280.

[30] Peter J Rousseeuw. 1987. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *J. Comput. Appl. Math.* 20 (1987), 53–65.

[31] Judy Hanwen Shen and Alex Tamkin. 2026. How AI Impacts Skill Formation. arXiv:2601.20245 [cs.CY] https://arxiv.org/abs/2601.20245

[32] Tanmay Sinha and Manu Kapur. 2021. When problem solving followed by instruction works: Evidence for productive failure. *Review of Educational Research* 91, 5 (2021), 761–798.

[33] Julie M. Smith, Jacob Koressel, Sofía De Jesús, Joe Kmoch, and Bryan Twarek. 2025. Comparing Learning Taxonomies With Computer Science K-12 Standards. In *Annual Meeting of the American Educational Research Association*. American Educational Research Association. doi:10.3102/2190401

[34] Xiaohang Tang, Sam Wong, Marcus Huynh, Zicheng He, Yalong Yang, and Yan Chen. 2025. SPHERE: Supporting Personalized Feedback at Scale in Programming Classrooms with Structured Review of Generative AI Outputs. In *Proceedings of the Extended Abstracts of the CHI Conference on Human Factors in Computing Systems (CHI EA '25)*. Association for Computing Machinery, New York, NY, USA, Article 467, 17 pages. doi:10.1145/3706599.3720203

[35] Michel Wermelinger. 2023. Using github copilot to solve simple programming problems. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 172–178.

[36] David Wood, Jerome S Bruner, and Gail Ross. 1976. The role of tutoring in problem solving. *Journal of Child Psychology and Psychiatry* 17, 2 (1976), 89–100.

[37] Yuankai Xue, Hanlin Chen, Gina R Bai, Robert Tairas, and Yu Huang. 2024. Does ChatGPT help with introductory programming? An experiment of students using ChatGPT in CS1. In *Proceedings of the 46th International conference on software engineering: software engineering education and training*. 331–341.

[38] Ashley Ge Zhang, Yan-Ru Jhou, Yinuo Yang, Shamita Rao, Maryam Arab, Yan Chen, and Steve Oney. 2026. Editrail: Understanding AI Usage by Visualizing Student-AI Interaction in Code. *arXiv preprint arXiv:2601.20085* (2026).

A Behavioral Labeling Interface

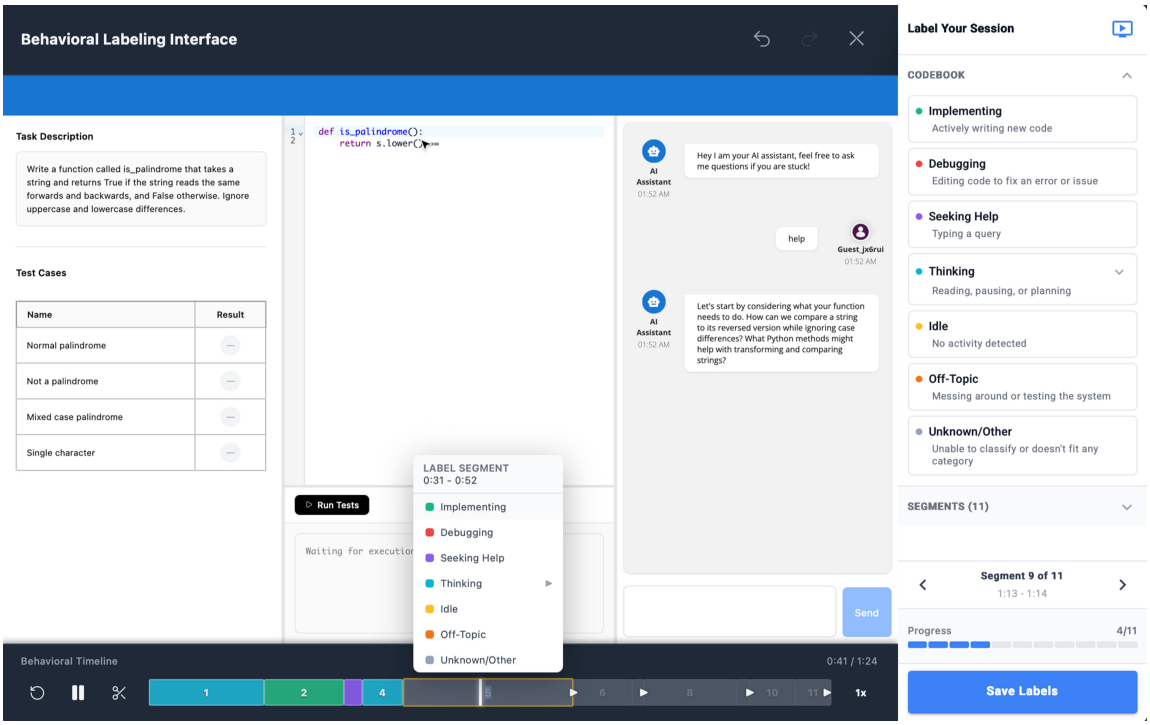


Figure 4: The Behavioral Labeling Interface used by expert labelers to annotate student sessions. The interface presents a session replay, a timeline for placing segment boundaries, and the behavioral codebook for classification.

B Codebook Change History

The behavioral codebook was iteratively refined across 10 pilot labeling sessions through weekly reconciliation meetings. Table 9 documents each revision with the corresponding date and specific change.

Table 9: Codebook revision history across the pilot labeling phase, derived from platform configuration changes.

Date	Change
Feb 2	Initial codebook: Implementing, Debugging, Thinking (with subtypes: Task, Code, Error, LLM Response), Idle, Off-Topic, Unknown.
Feb 3	Added *Seeking Help* ("Typing a query or reading LLM response").
Feb 9	Revised *Seeking Help* description to "Typing a query" only, removing "reading LLM response" to resolve overlap with Thinking: LLM Response subtype.
Feb 10	Added *Testing* ("Running code and reviewing results").
Mar 11	Removed *Testing* from manual labeling codebook (captured by auto-segmentation rules). Added *Thinking: Pre-query* subtype ("Asking LLM") to capture formulation pauses before chat queries.
Mar 11	Codebook finalized. All labelers confirmed agreement on category definitions and segmentation rules.

C Telemetry Event Schema

Table 10: Full telemetry event schema with example payloads from live deployments. Each event includes a millisecond timestamp, source region, and event-specific payload. Long event names and payloads are abbreviated for space.

Event Type	Description	Example Payload
Code Editor (11 types)		
CODE_TYPE	Typed characters	{code:"l", changes:[{to:0, from:0, text:"l"}], raw_action:"input.type"}
CODE_DELETE	Backspace/Delete	{code:"...grade_boo", changes:[{to:101, from:100, text:""}], raw_action:"delete.backward"}
CODE_DELETE_SEL.	Deleted selection	{code:"", changes:[{to:145, from:0, text:""}], raw_action:"delete.selection"}
CODE_PASTE	Pasted content	{code:"...append([78,84,91])", changes:[{to:75, from:75, text:"[78,84,91]"}], raw_action:"input.paste"}
CODE_COPY	Copied to clipboard	{selected_text:"grade_book = [[88,92,75],...]"}
CODE_CUT	Cut to clipboard	{code:"", changes:[{to:186, from:0, text:""}], raw_action:"delete.cut"}
CODE_UNDO	Ctrl+Z	{code:"...append()", changes:[{to:88, from:76, text:""}], raw_action:"undo"}
CODE_REDO	Ctrl+Y	{changes:[...], raw_action:"redo"}
CODE_SELECT	Highlighted text	{selected_text:"print(grade_book)"}
CODE_INDENT	Tab key	{code:"...for grade in grade_book:\n ", changes:[{to:96, from:96, text:" "}], raw_action:"input.indent"}
CODE_UNKNOWN	Unrecognized edit	{code:"...li", changes:[{to:57, from:57, text:"i"}], raw_action:"input.type.compose"}
Terminal (6 types)		
TERMINAL_RUN	Clicked Run Tests	{code_length:180}
TERMINAL_OUTPUT	stdout printed	{output:"95"}
TEST_CASE_RESULT	Test results	{passed_count:2, total_tests:3, passrate:66.7, detailed_results:[{passed:true, test_name:"Test Case 1"}, ...]}
TERMINAL_ERROR	Runtime error	{type:"UnknownError", message:"Traceback...IndexError: list assignment index out of range"}
TERMINAL_SELECT	Highlighted output	{selected_text:"IndexError: list assignment index out of range"}
TERMINAL_COPY	Copied output	{selected_text:"IndexError: list assignment index out of range"}
Chat (9 types)		
CHAT_TYPE	Typed in chat	{text:"h"}
CHAT_DELETE	Deleted in chat	{key_used:"Backspace", text_before:"...command fr", text_after:"...command f", cursor_position:34}
CHAT_PASTE	Pasted into chat	{pasted_text:"IndexError: list assignment index out of range"}
CHAT_QUERY	Sent message to AI	{text:"hi", length:2}
CHAT_RESPONSE	AI response	{content:"Great start! Let's tackle this step by step...", length:322, latency_ms:2495}
CHAT_SELECT_INPUT	Highlighted input	{selected_text:"what would code be"}
CHAT_SELECT_HIST.	Highlighted history	{selected_text:"what should I do?"}
CHAT_COPY_INPUT	Copied from input	{selected_text:"..."}
CHAT_COPY_HIST.	Copied from history	{selected_text:"what should I do?"}
Task / Tests (4 types)		
TASK_SELECT	Highlighted task	{selected_text:"[78, 84, 91"}
TASK_COPY	Copied from task	{selected_text:"[78, 84, 91]"}
TEST_SELECT	Highlighted test	{selected_text:"New"}
TEST_COPY	Copied from test	{selected_text:"Print all scores"}
Global (5 types)		
MOUSE_CLICK	Clicked in workspace	{x:0.588, y:0.183, region:"CODE_EDITOR", tag:"DIV", className:"cm-content"}
MOUSE_MOVE	Mouse position (50ms)	{x:0.384, y:0.272, region:"CODE_EDITOR"}
TAB_STATE	Browser tab switch	{visible:false}
WINDOW_RESIZE	Browser resized	{width:1588, height:901}
PANEL_RESIZE	Panel resized	{panel:"left", newSize:456}
Session (2 types)		
SESSION_START	Entered workspace	{window_width:1588, window_height:901}
SESSION_END	Left workspace	{duration_ms:542000}

D LLM System Prompt

The system prompt provided to GPT-4o for every student interaction is shown in Figure 5. The prompt implements the three-level scaffolding framework described in Section 3.2. The behavioral context blocks referenced in its final paragraph are reproduced in Appendix G. These blocks could override the default escalation order by directing the model to begin at a higher scaffolding level, but did not override the prohibition against providing runnable code.

Role: You are a supportive programming tutor embedded in an educational coding environment. Your role is to help students develop problem-solving skills through guided inquiry, not to solve problems for them.

Task Context: You will be provided with the task description, the student's current code, their test results, and the conversation history. Use the conversation history to track escalation level and whether the student has made progress.

Core Approach — Three-Level Scaffolding:

Level 1 — Socratic Questioning (default): Ask one targeted question that directs the student's attention to the relevant part of their code or problem. Do not provide hints at this level.

Level 2 — Conceptual Hint: Name the relevant concept or method without showing syntax, then ask a follow-up question.

Level 3 — Concrete Scaffolding: Provide a pseudocode skeleton with blanks for the current step only. Never show the structure of the full solution. All specific values, indices, and strings must be replaced with ___.

Absolute Rules:

(1) Never write runnable code in any programming language. When showing patterns, use only pseudocode with blank placeholders (___).
(2) Never scaffold more than one step at a time.
(3) Never skip escalation levels. Escalate exactly one level at a time.
(4) When the student moves to a new step or concept, always reset to Level 1.

Escalation Rules: Start at Level 1 for each new concept. After each response, assess whether the student made progress. If the current level did not result in progress, escalate one level. Only one escalation per response. If the student has no code and expresses uncertainty, begin at Level 2 with a conceptual hint about the first step only. When the student completes a step and moves to a new concept, reset to Level 1.

Response Guidelines: Keep responses concise (2–4 sentences plus one question). Ask only one question per response. Reference specific lines, variables, or test failures from the student's actual code.

Behavioral Context Override: If a behavioral context block is present, it describes what the student was doing before they asked. Use it to adjust scaffolding approach and escalation pace. The behavioral context never overrides the rule against writing runnable code; it only changes how scaffolding is delivered (e.g., starting at a higher level, trying a different angle, acknowledging effort).

Figure 5: The LLM tutoring system prompt provided to GPT-4o for every student interaction, implementing three escalating levels of scaffolding with a behavioral context override.

E Foundation Dataset Structure

Table 11 summarizes the abstraction layers of the released foundation dataset and its overall scale.

Table 11: Foundation dataset structure. Layers A–C comprise the TutorTrace foundation; Layer D is derived in the TutorTrace Taxonomy.

Layer	Representation	Scale
A: Raw Telemetry	Timestamped IDE events	~180K
B: Observable Metrics	Effort intensity per window	27 features
C: Behavioral Sequences	Auto-classified segments	13,633
D: Behavioral Profiles	Clustered profiles	10 profiles
Student sessions		480
AI interactions		1,386
Queries		840

F Auto-Segmentation Algorithm

Algorithm 1 presents the seven-step auto-segmentation pipeline that translates raw telemetry events into labeled behavioral sequences. Each step corresponds to a rule derived from expert-labeler consensus during codebook development (Section 4.1).

Algorithm 1 Auto-Behavioral Classification Pipeline

Require: Events $E = [e_1, \ldots, e_n]$ with timestamps relative to session start; session duration D
Ensure: Ordered list of behavioral segments S, each with behavior label and thinking subtype

Step 1: Build Major Segments
Classify each event by category: Code, Terminal, ChatInput, ChatResponse
Group consecutive same-category events into segments
if two consecutive Code events are separated by $\geq$ 6s, split into two segments
Assign initial behavior: Code $\rightarrow$ Implementing, Terminal $\rightarrow$ Testing, ChatInput $\rightarrow$ Seeking Help, ChatResponse $\rightarrow$ Thinking
Step 2: Fill Gaps with Thinking
for each gap between consecutive segments **do**
    if gap $\geq$ 3s **then**
        Insert Thinking segment spanning the gap
    else
        Extend the preceding segment to close the gap
    end if
end for
Apply same logic to gaps before the first segment and after the last segment
Step 3: Merge Short Testing Segments
for each Testing segment with duration $<$ 1.5s **do**
    Absorb into the adjacent segment (prefer next; fall back to previous)
end for
Step 4: Absorb Pre-Query Pauses
for each Thinking segment of duration $\leq$ 5s **do**
    if preceded by Implementing or Debugging **and** followed by Seeking Help **then**
        Store pause duration as metadata on the Seeking Help segment
        Extend the preceding segment to cover the pause
        Remove the Thinking segment
    end if
end for
Step 5: Apply Error State
Maintain flag *hasUnresolvedError* $\leftarrow$ FALSE
for each segment in chronological order **do**
    if segment contains `TERMINAL_ERROR` or a failed `TEST_CASE_RESULT` **then**
        hasUnresolvedError $\leftarrow$ TRUE
    end if
    if segment contains a `TEST_CASE_RESULT` where all tests pass **then**
        hasUnresolvedError $\leftarrow$ FALSE
    end if
    if segment is Implementing **and** *hasUnresolvedError* **then**
        Relabel segment as Debugging
    end if
end for
Step 6: Classify Thinking Subtypes
for each Thinking segment in chronological order **do**
    if no code, terminal, or chat activity has occurred yet **then**
        Label as *Thinking: Task* ▷ Reading the task description
    else if preceding segment was Seeking Help **then**
        Label as *Thinking: Response* ▷ Reading AI reply
    else if most recent terminal run produced an unresolved error **then**
        Label as *Thinking: Error* ▷ Reading an error message
    else
        Label as *Thinking: Code* ▷ Reviewing own code
    end if
end for
Step 7: Post-Process
Fix any remaining unlabeled segments (terminal events $\rightarrow$ Testing; otherwise $\rightarrow$ Thinking)
Merge consecutive segments with the same behavior label
Re-index all segment IDs
return S

G Behavior-Aware Intervention Prompts

In the treatment condition (Section 7.3), the system appended one of five behavior-aware context blocks to the student's query. Cold Start was used for a learner's first query when no code edit or terminal run had occurred. For subsequent queries, the system selected among Passive, Iterating, Debugging, and Spinning using online activity rules derived during system development. These deployment-time rules used the same profile names as the later taxonomy but represented precursor heuristic states rather than assignments from the revised clustering procedure. The prompts below reproduce the exact intervention text used during deployment. The revised taxonomy was subsequently applied to both deployments to compare their Window 2 profile distributions.

Cold Start

This is the student's first time asking for help. They have not written or run any code yet. Give them something concrete to start with — a pseudocode scaffold with blanks to fill in. End your response with a single sentence starting with "Try:" that gives them that pseudocode starting point so they know what to write next.

Passive

This student received help but has not written or run any code since. Try a different angle from your last response — do not repeat the same explanation. Do NOT escalate your scaffolding level — stay at the same level but approach it differently. If the student asks "how to fix" or requests the answer directly, acknowledge, redirect, and re-ask with pseudocode. If the student has just completed a step and is asking what to do next, start at Level 1 for the new step. If the question is purely conceptual, answer it naturally. Otherwise, end your response with a single sentence starting with "Try:" that gives them one concrete next action using pseudocode with blanks if code is involved.

Iterating

This student wrote some code but has not tested it. Do not give more explanation. If their current code would produce meaningful output (e.g. contains a print statement, an expression, or enough logic to show a result), end your response with a single sentence starting with "Try:" telling them to run it. If running it would produce no output, end with a "Try:" that asks them to add a specific print statement first so they can see what their code is doing.

Debugging

This student has been coding, testing, and hitting errors. They are engaged and struggling. Recognize their effort. Name the specific error and the misconception behind it so they can read errors independently. Escalate to Level 2 or 3 immediately. End your response with a single sentence starting with "Try:" giving them the targeted fix as a pseudocode pattern with blanks.

Spinning

This student has run their code many times and keeps hitting errors. They do not need encouragement — they need to be unblocked. Name the concept or method they need, explain briefly why their current approach fails, and escalate to Level 3 immediately. End your response with a single sentence starting with "Try:" that gives them the precise pseudocode pattern they need, with blanks for the specific values.

H Help-Seeking Type Labeling Prompt

The system prompt shown in Figure 6 was provided to GPT-4o to classify each student query as guided or dependent. The model received the student's query, current code, and preceding chat history. These semantic inputs were used only to generate the offline help-seeking labels and were excluded from the prediction models described in Section 7.4.

You are an educational assistant analyzing student help requests in introductory programming courses.

Given a student query, their current code, and chat history, classify the query as either `GUIDED` or `DEPENDENT`.

GUIDED — The student demonstrates independent thinking. They have identified what they need help with and are actively steering their learning. This includes:

- Asking a specific question about a concept ("What is a nested loop?")
- Identifying a specific problem or confusion ("I'm not sure what to put in the print statement")
- Describing what they tried and what went wrong ("I tried using a for loop but it only prints the first item")
- Asking how to approach a specific step ("How do I iterate through each student's grades?")
- Requesting clarification on a specific point from a prior AI response ("What do you mean by iterating over the inner list?")
- Answering the AI's question with specific information ("The error is IndexError on line 5")

The key indicator: the student has done some cognitive work to formulate what they need. The query communicates a specific need, question, or confusion, even if brief.

DEPENDENT — The student is offloading cognitive work to the AI with minimal independent effort. This includes:

- Pasting code with no question or description of the problem (implicit "fix this for me")
- Vague requests with no specifics ("help," "it doesn't work," "idk")
- Pure acknowledgments that delegate next steps ("ok do that," "yeah," "sure," "go ahead")
- Empty or near-empty messages ("?," stray characters)
- Requests that ask the AI to do the work ("can you just write it for me," "give me the code")
- Repeating the assignment prompt or pasting instructions without any attempt or question
- Answering the AI's question with no effort ("idk," "I don't know," "you tell me")

The key indicator: the student has not done cognitive work to identify what they need. They are asking the AI to do the thinking for them.

Boundary cases — apply these rules:

- "I don't know how to do X" → `GUIDED` (they identified what they do not know)
- "I don't know" or "I'm stuck" alone, with no specifics → `DEPENDENT`
- "Is this right?" with code → `DEPENDENT`
- "Is this right? I'm not sure if my loop handles the last element" → `GUIDED`
- Code pasted with "Why does this print the whole list instead of individual scores?" → `GUIDED`
- Code pasted with no question → `DEPENDENT`
- "What about edge cases?" in the context of an ongoing conversation → `GUIDED`
- "Ok," "Thanks," or "Got it" alone → `DEPENDENT`
- "Ok, but how does that work with nested lists?" → `GUIDED`

Confidence:

- `high`: clearly guided or clearly dependent
- `medium`: leans one way but has some ambiguity
- `low`: genuinely on the boundary

Return only a JSON object:

```
{
"queryEngagement": "guided" or "dependent",
"rationale": "1 sentence explaining why",
"confidence": "high/medium/low"
}
```

Figure 6: The help-seeking type labeling prompt provided to GPT-4o to classify each student query as guided or dependent from the query text, the student's code at submission time, and the preceding chat history.

I Task Descriptions

Full task descriptions and test cases are reproduced below as presented to students.

Students were given a pre-populated list of song names and asked to complete four list manipulation steps within 15 minutes.

Deployments 1–2: Playlist (Python, 15 min)

You are given a list of song names called `playlist`. Complete the following steps:

(1) The song at index 4 was added by mistake. Replace it with "Purple Rain".
(2) Remove the last 3 songs from the playlist using slicing.
(3) Reverse the playlist in place.
(4) Print every other song in the playlist using slicing with a step value.

Test cases:

1. `"Purple Rain" in playlist` → `True`
2. `len(playlist)` → `5`
3. `playlist[0]` → `"Purple Rain"`
4. `playlist[::2]` → `["Purple Rain", "Imagine", "Bohemian Rhapsody"]`

Students were given a nested list of test scores and asked to complete three list manipulation steps within 10 minutes.

Deployments 3–4: Grade Book (Python, 10 min)

You are given a list of lists called `grade_book`, where each inner list contains a student's test scores. Complete the following steps:

(1) A new student joined the class with scores `[78, 84, 91]`. Add their scores to `grade_book`.
(2) The first student's third score was entered incorrectly. Update it to `80`.
(3) Using a nested for loop, print every individual score in `grade_book`, one per line.

Test cases:

1. `len(grade_book)` → `4`
2. `grade_book[3]` → `[78, 84, 91]`
3. `grade_book[0][2]` → `80`
4. Nested loop prints all 12 scores, one per line.

J Prediction Breakdowns: Observation Windows, Feature Contributions, and Profiles

Table 12: Held-out AUROC across observation-window sizes for both prediction tasks and all three feature layers. The 30-second query-imminence row and 15-second help-seeking-type row correspond to Table 7; bold marks the configuration reported in the subsequent tables.

Task	Window	Test n	Positive	Raw	+Obs.	+Seq.
Query imminence	15 s	11,160	37.4%	0.657	0.690	0.707
	30 s	10,825	38.2%	0.689	**0.726**	0.719
	45 s	10,492	38.7%	0.701	0.734	0.720
	60 s	10,165	39.1%	0.700	0.730	0.713
Help-seeking type	15 s	499	33.1%	0.690	**0.717**	0.705
	30 s	499	33.1%	0.696	0.680	0.694
	45 s	498	33.1%	0.691	0.674	0.696
	60 s	495	33.3%	0.678	0.649	0.667

Table 13: Complete feature-contribution list (Random-Forest importances, all features with importance > 0) for query imminence at its reported configuration (30 s window, observable layer).

Feature	Imp.	Feature	Imp.
metric__time_in_chat_s	0.1465	metric__error_to_edit_s	0.0060
metric__longest_idle_s	0.0761	raw__TERMINAL_OUTPUT	0.0058
raw__event_density_per_s	0.0575	raw__CODE_SELECT	0.0057
metric__time_in_editor_s	0.0573	metric__failed_test_to_edit_s	0.0053
raw__event_count	0.0573	raw__TEST_CASE_RESULT	0.0043
raw__MOUSE_MOVE	0.0525	metric__max_consecutive_errors	0.0040
raw__CHAT_RESPONSE	0.0507	raw__TERMINAL_ERROR	0.0038
metric__time_in_terminal_s	0.0452	metric__terminal_errors	0.0037
raw__MOUSE_CLICK	0.0435	raw__CODE_PASTE	0.0028
metric__thinking_time_s	0.0393	raw__CODE_UNKNOWN	0.0027
raw__TAB_STATE	0.0278	raw__CODE_COPY	0.0026
metric__net_code_growth	0.0276	raw__PANEL_RESIZE	0.0026
metric__chars_inserted	0.0263	raw__CODE_DELETE_SELECTION	0.0020
metric__time_in_task_s	0.0248	raw__SESSION_START	0.0020
raw__CODE_TYPE	0.0247	metric__failed_test_self_fix	0.0020
metric__delete_type_ratio	0.0240	raw__CODE_INDENT	0.0019
metric__code_edit_rate	0.0236	metric__error_self_fix	0.0017
metric__code_edits	0.0230	raw__TASK_COPY	0.0016
metric__chars_deleted	0.0218	raw__TASK_SELECT	0.0014
metric__time_in_tests_s	0.0198	raw__CHAT_SELECT_HISTORY	0.0013
raw__CODE_DELETE	0.0152	raw__CHAT_SELECT_INPUT	0.0011
metric__code_deletes	0.0148	raw__TERMINAL_SELECT	0.0007
raw__TERMINAL_RUN	0.0072	raw__CODE_UNDO	0.0007
metric__error_reading_time_s	0.0070	raw__TERMINAL_COPY	0.0003
metric__terminal_runs	0.0069	raw__CODE_CUT	0.0002
raw__WINDOW_RESIZE	0.0069	raw__TEST_SELECT	0.0001
metric__mean_time_between_runs_s	0.0062		

Table 14: Complete feature-contribution list (Random-Forest importances, all features with importance > 0) for help-seeking type at its reported configuration (15 s window, observable layer).

Feature	Imp.	Feature	Imp.
metric__longest_idle_s	0.1442	raw__TERMINAL_OUTPUT	0.0064
metric__time_in_editor_s	0.0984	metric__max_consecutive_errors	0.0054
raw__event_count	0.0922	raw__WINDOW_RESIZE	0.0050
raw__event_density_per_s	0.0881	raw__TERMINAL_ERROR	0.0049
metric__time_in_chat_s	0.0868	raw__TERMINAL_SELECT	0.0048
raw__MOUSE_MOVE	0.0786	metric__terminal_errors	0.0045
raw__MOUSE_CLICK	0.0625	raw__CODE_SELECT	0.0042
metric__time_in_terminal_s	0.0501	raw__CODE_COPY	0.0039
raw__CHAT_RESPONSE	0.0422	raw__TERMINAL_COPY	0.0036
metric__thinking_time_s	0.0361	raw__TASK_COPY	0.0025
metric__time_in_task_s	0.0173	raw__TASK_SELECT	0.0025
metric__code_edit_rate	0.0170	raw__TEST_CASE_RESULT	0.0024
metric__code_edits	0.0154	metric__error_reading_time_s	0.0022
raw__CODE_TYPE	0.0137	metric__mean_time_between_runs_s	0.0010
metric__net_code_growth	0.0132	raw__CHAT_SELECT_INPUT	0.0009
metric__chars_inserted	0.0123	raw__CHAT_SELECT_HISTORY	0.0005
metric__chars_deleted	0.0118	raw__PANEL_RESIZE	0.0004
metric__code_deletes	0.0109	raw__CODE_DELETE_SELECTION	0.0003
raw__CODE_DELETE	0.0109	metric__failed_test_self_fix	0.0002
metric__time_in_tests_s	0.0100	raw__CODE_PASTE	0.0002
metric__terminal_runs	0.0084	metric__error_to_edit_s	0.0002
metric__delete_type_ratio	0.0083	metric__error_self_fix	0.0001
raw__TERMINAL_RUN	0.0082	metric__failed_test_to_edit_s	0.0001
raw__TAB_STATE	0.0071		

Table 15: Distribution of behavioral contexts among held-out query-imminence observation windows at the reported window (30 s), with each context's positive rate. Profiles are evaluation strata from the deployment-1–2 taxonomy models.

Behavioral context	Windows	Share	Positive rate
W2: Iterating	1,837	17.0%	80.8%
After last query	1,787	16.5%	0.0%
No query this session	1,600	14.8%	0.0%
W2 context (invalid interval)	1,289	11.9%	21.5%
W1: Oriented	1,072	9.9%	52.6%
W2: Passive	1,061	9.8%	91.3%
W2: Spinning	607	5.6%	25.5%
W2: Debugging	566	5.2%	55.1%
W1: Struggling	436	4.0%	27.5%
W1 context (excluded from population)	327	3.0%	14.7%
W1: Cold Start	243	2.2%	82.7%

Table 16: Distribution of behavioral profiles among held-out help-seeking-type queries at the reported window (15 s), with each profile's guided rate.

Preceding profile	Queries	Share	Guided
W2: Passive	186	37.3%	22.6%
W2: Iterating	166	33.3%	33.7%
W1: Oriented	47	9.4%	72.3%
W2: Debugging	28	5.6%	35.7%
W2 (invalid interval)	25	5.0%	36.0%
W1: Cold Start	20	4.0%	0.0%
W2: Spinning	13	2.6%	46.2%
W1: Struggling	10	2.0%	60.0%
W1 (excluded from population)	4	0.8%	50.0%