

Artificial Programming Students as Tool-Calling Agents

Charles Koutcheme¹, Arto Hellas¹ and Juho Leinonen^{1,*}

¹Aalto University, Espoo, Finland

Abstract

Artificial programming students are a promising direction for computing education research, as they could enable controlled experimentation around tutoring or feedback without requiring repeated classroom deployments. However, existing models treat artificial students as program generators, predicting sequences of submitted or executed programs without modeling the actions that produced them. As a result, much of the iterative process that characterizes real student behavior, the editing, the local execution, and the integration of summative feedback, remains unmodeled. In this work, we propose to model artificial students as agents interacting with a programming environment through tool calls. We present a framework that derives a vocabulary of student actions from event-stream process data and casts those actions as tool calls in the agentic conversational format. We formalize student modeling under this framework as a next-action prediction task and define corresponding evaluation protocols. We demonstrate the feasibility of the agentic formulation by instantiating the resulting setting on raw keystroke-level interaction logs from a large-scale CS1 dataset and fine-tuning small open-weight language models on real student trajectories. We also discuss how modeling students at the level of fine-grained actions is a promising direction for building more faithful computational models of student programming behavior. We release our code to support further research.

Keywords

Artificial Students, Language Models, Fine-tuning, Process Data

1. Introduction

Understanding how students learn to program, and how they respond to different forms of support, is central to computing education research [1]. This includes studying how students progress through programming tasks, how they react to feedback or hints, and whether given problems or solutions are within their reach [2]. However, investigating such questions in real settings remains challenging: controlled studies are slow to iterate on, difficult to design without confounds, and constrained by the need to avoid disrupting real learners.

Developing artificial students, simulated computational representations of learners, has long been explored as a way to address this challenge [3, 4, 5]. Recent advances in language modeling now make this vision more attainable. Language models are strong pattern learners: they can reproduce distributions observed in provided examples through prompting, or internalized from training data through fine-tuning [6]. When aligned with the characteristics of specific student populations, such models could therefore enable controlled and scalable studies tailored to each classroom setting, without the practical constraints of real-world deployment.

These qualities make them a natural candidate for modeling how students progress through programming tasks. In programming education, artificial learners based on language models are already being explored. While early approaches primarily relied on prompting proprietary LLMs, more recent work instead fine-tunes smaller open models directly on student traces [7, 8]. This approach offers practical advantages in adaptability, cost, privacy, and deployability over proprietary systems [9].

Despite this progress, most if not all approaches do not depart from a common assumption: that students can always be modeled as sequential code generators. Students' coding processes are complex. They incrementally *edit* code, execute it locally, interpret the resulting output, and refine their solutions before ever submitting. Upon submission, they receive summative feedback from a grader and continue

CSEDM'26: 10th Educational Data Mining in Computer Science Education (CSEDM) Workshop, June 28, 2026, Seoul, South Korea

*Corresponding author.

✉ charles.koutcheme@aalto.fi (C. Koutcheme); arto.hellas@aalto.fi (A. Hellas); juho.2.leinonen@aalto.fi (J. Leinonen)

ORCID 0000-0002-2272-2763 (C. Koutcheme); 0000-0001-6502-209X (A. Hellas); 0000-0001-6829-9449 (J. Leinonen)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

the process. In other words, students do not simply *emit* programs; they iteratively *refine* their solutions while interacting with a programming environment that continuously shapes their actions through feedback. Modern learning environments capture and store these interactions in fine-grained details, including both student actions and environment responses [10]. Yet, although this data is often available, current artificial student models largely collapse this rich process into sparse sequences of submissions. As a result, key aspects of student problem-solving dynamics remain unmodeled. We argue that addressing this limitation requires more than richer data – it requires a shift in the modeling objective itself: student behavior should be modeled at the level of actions, not outputs.

In this work, we therefore shift from predicting programs to predicting the fine-grained actions students take while solving assignments, conditioned on the feedback they receive from the learning environment. This shift aligns the modeling granularity with the structure of the underlying process, enabling us to capture both the incremental evolution of student code and the feedback signals that guide it. To operationalize this conceptual shift, we propose a novel framework that represents an artificial student as an agent [11], where student actions correspond to tool calls and environment feedback corresponds to the resulting tool responses [12]. We equip these artificial students with tools for incremental code modification, local execution, and code submission and task them to solve the given programming assignments.

We test our framework on CodeBench [13], a large-scale dataset of Python programming assignments with fine-grained student interaction data. We train small language models from the Qwen family [14] at 0.8B, 2B, and 4B parameter sizes on serialized trajectories and evaluate their ability to predict students fine-grained actions, comparing trained models against prompted baselines. Interestingly, our results suggest that increasing model size does not systematically improve performance.

Contributions. In summary, our contributions are:

- **Framework.** We propose a framework that derives a vocabulary of student actions from event-stream process data and casts those actions as tool calls in the agentic conversational format. To the best of our knowledge, we are the first to reposition artificial student modeling as an action-prediction problem rather than a program-generation one.
- **Evaluation and empirical study.** We define evaluation protocols for measuring action-level alignment with real student trajectories, and validate the framework on large-scale programming interaction data. Our code:  [KoutchemeCharles/csedm26-aps](https://github.com/KoutchemeCharles/csedm26-aps)

2. Related Work

A complementary line of work focuses on Knowledge Tracing (KT), the task of estimating students’ latent knowledge states over time, typically by predicting performance on future tasks. In programming education, several extensions of KT have incorporated code and program execution outcome prediction to improve prediction accuracy [15, 16, 17]. Despite these advances, these models are designed primarily for knowledge estimation and response prediction, and are not intended to model the fine-grained, within-assignment process by which students iteratively construct solutions through interaction with the learning environment.

Our work draws on other approaches that focus exclusively on within-assignment simulation. Closest to our work is Zhan et al. [11], who propose *CoderAgent*, a prompted LLM agent architecture comprising various modules designed to predict subsequent student submissions to programming assignments. In a similar spirit, *CoderAgent* integrates compiler feedback to guide generation and structures revision by identifying a problematic code segment, generating a localized fix, and then producing the full next program.

However, our work differs in three key aspects. First, in *CoderAgent*, behavior is induced through prompting GPT-4o models, whereas our work trains small models directly on authentic student traces.

Second, while *CoderAgent* models edits implicitly through structured reasoning, we represent edits explicitly as actions through tool calls. Third, our execution harness incorporates not only compiler feedback, but also local execution signals and grader-provided summative feedback, more closely matching the information available to real students.

The idea of training models of student behavior from data has already been explored. Miroyan et al. [7] fine-tune open-weight language models to predict a student’s next submission given a limited context of prior attempts, and report improvements over prompted baselines. Similarly, Ross et al. [8] demonstrate that learning from large-scale sequences of program snapshots yields richer representations of student behavior than learning from final programs alone. Rather than predicting a student’s next program or next local attempt as a monolithic unit, we model the individual actions they take as they work through an assignment.

Outside of education, environmental feedback has been increasingly integrated into language models to improve their ability to solve general coding tasks, for instance by allowing models to observe code execution outputs [18]. This paradigm has been extended in agent-based settings, where models are equipped with tools and interfaces to support iterative problem solving [12]. In particular, closest to us in that domain are Yang et al. who provide language model agents with editing, navigation, and execution tools through a dedicated agent interface, enabling them to solve real-world software engineering tasks [12]. They instantiate a ReAct-style loop [19] of thought, action, and observation, where the model iteratively interacts with the environment and incorporates feedback from tool execution, without modifying model weights.

This line of work provides a natural abstraction for modeling programming as an interactive process grounded in environmental feedback, and our work is partly inspired by it. Yet our goal diverges from prior approaches in a key aspect. Rather than designing agentic behavior through prompting and system architecture with the goal of solving complex software engineering problems, we define tools that map to student actions and learn the process directly from authentic student interaction traces. The goal is to model accurately, not to solve efficiently. This allows us to capture not only how solutions are constructed through iterative feedback, but how this process unfolds for real learners in educational settings.

3. Framework

In this section, we introduce our framework for simulating student behavior within programming assignments using language models. We focus on modeling behavior within a single programming assignment, and leave the integration of prior assignment history to future work.

We build the framework in two steps. We first derive a vocabulary of student actions from the structure of process data, then map that vocabulary onto the agentic conversational format used by tool-calling models.

3.1. From Process Data to Actions

Our framework starts from the analysis of programming process data. Prior work has analyzed this data at multiple levels of granularity, from submission logs and compilation snapshots to edit logs and full keystroke traces [20, 21, 22, 23]. We find that process data can be categorized into two regimes: snapshot and stream. *Snapshot* type data record the state of the program at specific moments, such as saves, compilations, or submissions; here the program is explicit at each step, and the student actions between snapshots are *implicit*. *Event-stream* data (e.g., line level or keystroke) on the other hand record the changes themselves as they happen; here the actions are *explicit*, and the program state is implicit, recoverable only by replaying the stream.

Prior work in artificial programming students has largely operated on snapshot data, and has consequently modeled student programming as a sequence of program states, predicting the next submission [7] or next program state [8] as a monolithic unit. Our framework instead operates on event-stream data, and we therefore model student behavior as a sequence of actions. This shift emphasizes the process of construction and revision central to how students learn to program [20, 21].

From this stream type data, two classes of student actions naturally emerge: modifications and executions. **Modification actions** are everything the student does to the program between these execution events. We treat each class in turn. **Execution actions** are explicit interactions with the programming environment: running the current program, or submitting it for grading.

3.1.1. Modification Actions

We categorize modification events in stream data in three primitive operations on the program: *additions* that insert new content without removing existing content, *deletions* that remove existing content, and *edits*, a general case combining both, that replace existing content with new content. These three operations are easy to identify at a low level in a log, but defining what *should* count as a single instance of *action* is not. Raw stream events arrive at a finer grain than the level at which students actually decide what to write: a student modifying a region of the program typically has a **programming plan** in mind [24, 25], such as fixing a specific behavior, and the stream records the keystrokes that follow from that plan rather than the plan itself. Since we cannot observe these plans directly, we need proxies that aggregate stream events into units closer to a single coherent decision. Our two modeling choices below each address one dimension of this aggregation: granularity and grouping.

The first dimension is granularity. Stream data arrives at different resolutions: keystroke logs preserve every character event, while line-level edit logs record only the lines that change between two consecutive states. We operate at the line level rather than the character level. Character-level edits are too fine-grained to be stable action units, as they include transient corrections, partial tokens, and other local fluctuations that do not reliably correspond to coherent steps in problem solving. Line-based operations provide a more interpretable unit of change while remaining expressive enough to capture substantive program revisions [26, 27]. This choice is also compatible with current language models, which continue to be weaker on character-level manipulation tasks because tokenization makes such operations harder to represent and execute reliably [28, 29]. A consequence of this commitment is that even a single-character change is recorded as a line-level action, in which the affected line is replaced in full by its modified version.

The second dimension is grouping. In this first version of our framework, we group line-level events that fall within a *contiguous* region of the program into a single action, regardless of the order in which the lines were touched. Adjacent lines typically implement a common local structure, such as a condition body or a short loop, so edits within a contiguous region are likely serving the same immediate local goal. Although this is not a perfect reconstruction of a plan step boundary, which do not necessarily need to be spatially contiguous [24, 25], the contiguous region is our best proxy for the spatial scope of a single editing decision for a student attempt at improving their program.

We label the net change (i.e., the cumulation of all changes) within each region as an *addition* if new lines are inserted without altering existing content, a *deletion* if lines are removed without replacement, and an *edit* otherwise, with *edit* as the general case. For example, a student who modifies lines 1, 2, and 3 in any order is working in one contiguous region, and we record this as a single action spanning those lines. A student who modifies line 2, then line 6, then line 5 is working in two distinct regions and produces two actions: one on line 2 and one on lines 5–6.

3.1.2. Execution Actions

Students perform two types of execution actions: running their program locally, producing command-line outputs, and submitting their work for evaluation, which produces autograder feedback. These actions are the main external signals through which students observe whether their approach works.

3.2. Casting Actions as Tool Calls

Recent advances in language modeling have produced *agentic* models, that is, models trained to interact with external systems through *tools*. A tool is a function the model can call to perform some action outside its own forward pass and receive a result back (e.g., a search query that returns documents, a Python interpreter that returns the output of executing a code snippet). These calls take place inside an extended conversational format: alongside the system, user, and assistant turns of standard chat models [30], agentic models consume and emit *tool* turns that carry tool calls and their responses. The model interleaves natural-language reasoning with tool calls, and tool results re-enter its context as new information to act on [12, 19].

Our action vocabulary maps directly onto this paradigm. In our instantiation, assistant turns represent the artificial student, user turns represent any instructional agent, tool turns represent the programming environment, and the system turn provides contextual information that conditions the interaction as a whole. Each modification or execution action a student takes corresponds to a tool call from the assistant; each environment response corresponds to the result returned in the following tool turn. Modeling student behavior in the agentic setting therefore amounts to defining the set of tools available to the artificial student, the inputs each tool accepts, and the result each returns.

We define five tools for the artificial student, following the primitives identified in Section 3.1¹. Our three modification tools update the program state: EDIT takes a contiguous line range and a block of replacement content; ADD takes an insertion point and a block of new content; DELETE takes a contiguous line range. Our two execution tools probe the program functionality: EXECUTE takes an optional list of standard-input values to feed the program at runtime, and SUBMIT takes no arguments. What each tool returns follows from the nature of the underlying action. For execution tools, the return is the externally produced feedback that students already observe in their IDE: the program’s standard output and standard error for EXECUTE, and autograder feedback for SUBMIT. For modification tools, no such external feedback is intrinsically produced. We therefore return a window of numbered lines around the affected region of the program. This window approximates the local view of the code a student keeps on screen while editing, and parallels prior agentic coding work that returns an explicit observation of the edit so that the model can confirm the effect of its action and maintain an updated view of the code state [12, 31].

The system turn instructs the model that it is a student learning how to program and lists the available tools; in principle, it can also carry learner-specific information such as background statistics [32] or knowledge component scores from prior assignments [33], following prior works. The first user turn carries the assignment description; in principle, if the dataset used contains such information, subsequent user turns can carry tutor interventions (e.g. socratic hint) aligning with the standard dialogue framing in the tutoring literature [34, 35]. Assistant turns contain only tool calls in our instantiation; extending them with natural-language reasoning before each call, in the style of ReAct [19], would require reasoning traces that are not directly visible in process data, and we leave this to future work. Figure 2 (Appendix B) shows an example trajectory under this serialization, situated next to a related agentic formulation from the software engineering literature.

4. Training Artificial Students

In this section, we introduce how to train a language model to act as an artificial student under our framework. Specifically, we present student modeling as a next-action prediction task (Section 4.1) and adopt supervised fine-tuning as a first training strategy (Section 4.2).

¹We also define a tool called “GIVE_UP” to make it explicit to language models when students abandon solving the assignment, but leave the analysis of such cases for future work.

4.1. Next-Action Prediction

A **trajectory** is a sequence of tool calls and their responses recorded over the full problem-solving session: $\xi = (a_1, o_1, \dots, a_T, o_T)$, where each a_t is a tool call issued by the student at step t and o_t is the environment’s response. Given a partial trajectory $\xi_{<t} = (a_1, o_1, \dots, a_{t-1}, o_{t-1})$ up to step t , the **next-action prediction** task is to predict a_t .

4.2. Supervised Fine-Tuning

Given a dataset $\mathcal{D} = \{(d^i, \xi^i)\}_{i=1}^N$ of N student trajectories, each for an assignment d^i , our goal is to fine-tune a language model π_θ for next-action prediction. We adopt supervised fine-tuning as a first training strategy. Each trajectory is serialized into the conversational format as described in Section 3.2: the assignment description d is placed in the first user turn, and the tool calls and responses are interleaved as alternating assistant and tool turns. We then minimize the negative log-likelihood of student actions over the training trajectories:

$$\mathcal{L}(\theta) = - \sum_{(d, \xi) \in \mathcal{D}} \sum_{t=1}^{T_\xi} \log \pi_\theta(a_t \mid d, \xi_{<t}),$$

where T_ξ is the length of trajectory ξ . Following standard practice for instruction-tuned models [36], the loss is computed *only* on assistant turns; this excludes environment responses o_t appearing in the conditioning context, as they are deterministic outputs of the execution environment rather than student behavior to be modeled.

5. Experiments

In this section, we present a set of experiments training small language models as programming students for next action prediction under our framework. We aim to answer the two following research questions:

RQ1 How does model size affect action-type prediction and edit-quality metrics?

RQ2 How does supervised fine-tuning improve next-action prediction compared with prompted base models?

5.1. Dataset

We use data from CodeBench [13], a large-scale publicly available programming dataset containing real-life student programs. CodeBench is an online judge that has been deployed since 2016 as part of a CS1 course reform at the Federal University of Amazonas. The dataset, provides fine-grained logs of student interactions with the integrated IDE, including code executions, submissions, test-cases and grading. The current release spans more than over 375,000 student solutions over multiple course offerings. Due to space limitations, we describe the dataset preprocessing and details to obtain training and test sets in Appendix A.1.

5.2. Models

We finetune three small open-weight language models from the Qwen3.5 family [14]: Qwen3.5-0.8B, Qwen3.5-2B, and Qwen3.5-4B. We focus on small models for their potential to be deployed in low-resource environments, and select this family for its strong performance on coding benchmarks. All models are finetuned on the training split of CodeBench; hyperparameters are reported in Appendix A.2. As baselines, we evaluate the same three models under in-context learning, i.e., without finetuning. We evaluate each model on sampled trajectories from the test set; Appendix A.3 details the setup.

5.3. Evaluation

Each predicted action $\hat{a}_t$ has two evaluable components: its **type** and its **arguments**. We first evaluate action type prediction performance as multiclass classification problem and report per-class and aggregate weighted F1 scores [37]. We also evaluate modification actions quality by measuring the similarity between the program states induced by $\hat{a}_t$ and a_t . Let c_{t-1} denote the program state at step t and $\delta(c_{t-1}, a_t)$ the state obtained by applying a_t . We compare $\delta(c_{t-1}, \hat{a}_t)$ against $\delta(c_{t-1}, a_t)$ using: (a) **CodeBLEU** [38], measuring token-level syntactic similarity [16, 17, 15]; (b) **grade proximity**, defined as $1 - |v^{(d)}(\delta(c_{t-1}, \hat{a}_t)) - v^{(d)}(\delta(c_{t-1}, a_t))|$ where $v^{(d)}(\cdot) \in [0, 1]$ is the grading function for assignment d , measure functional alignment. These two metrics are computed, for each model, on steps where both the ground truth and the model predict a modification action. We report this as N_{match} metric. This means that models with different N_{match} values have their quality metrics computed on different subsets of predictions, which should be kept in mind when comparing across methods. For completeness, we also report N_{inv} , the number of N_{match} for which the input arguments are invalid (e.g., model deletes 6 lines in a 3 lines program). We leave the evaluation of execution arguments for future work.

6. Results

Table 1 shows the results of our experiments. We can make the following observations.

Base model performance does not scale meaningfully with size. Increasing model size from 0.8B to 4B parameters yields no clear improvement in action type prediction. The weighted F1 score remains essentially flat (0.452, 0.437, 0.453), with the smallest model slightly outperforming the 2B variant. Edit content quality follows a similar pattern: CodeBLEU varies non-monotonically (0.777, 0.784, 0.747) and grade proximity stays within 0.003 across sizes. Yet two modification metrics do scale: matched modification pairs (N_{match}) increase from 3460 to 3829, and invalid edits (N_{inv}) drop from 193 to 117. Larger base models are therefore better at recognizing that a modification step is occurring and at staying within the scope of the program, but no better at deciding which action type to take or at producing higher-quality edits.

Supervised fine-tuning improves performance across most metrics. Fine-tuning on the training set trajectories raises weighted F1 by 0.137 to 0.148 points across the three model sizes, with the largest per-class gains on EDIT (e.g., 0.521 to 0.761 for the 4B model) and SUB (e.g., 0.285 to 0.413 for the 0.8B model). Edit content also improves: CodeBLEU rises across all sizes, and N_{inv} drops substantially (e.g., 117 to 50 for the 4B model). Two metrics behave differently: (i) DEL F1 decreases slightly (e.g., 0.127 to 0.120 for the 4B model), which we attribute to deletion being the rarest action in the dataset (Figure 1); models likely overfit to the more frequent action types, and (ii) Grade proximity drops by roughly 0.02 across all sizes. We argue this drop does not reflect a degradation in edit quality; since N_{match} rises by roughly 1500 pairs after fine-tuning, models are evaluated on a substantially larger and likely harder subset of modification steps that the base models miss entirely.

Supervised fine-tuned performance does not scale reliably with model size. We expected gains to grow with model size, since LoRA capacity scales with the base model. The trend holds only partially. EDIT F1 increases monotonically (0.723, 0.737, 0.761), and the edit matching metrics continue to scale (N_{match} : 4957 to 5340; N_{inv} : 114 to 50). Yet weighted F1 is non-monotonic (0.589, 0.581, 0.601), and SUB F1 actually drops from 0.413 at 0.8B to 0.344 at 2B before recovering only to 0.349 at 4B. CodeBLEU and grade proximity remain flat across sizes (within 0.002 and 0.009 respectively). The scaling pattern after fine-tuning therefore mirrors the base model finding: model size helps with detecting and bounding modification actions, but does not translate into better classification of action types or measurably better edit content.

Table 1

Action type prediction and edit quality (2023-1 train $\rightarrow$ 2024-1 test, bin sampling). **Mode:** Base = raw checkpoint, SFT = fine-tuned. Type columns: per-class F1 and aggregate weighted F1 ($\uparrow$); Edit columns: CodeBLEU ($\uparrow$), matched modification pairs (N_{match}), invalid edits (N_{inv} , $\downarrow$), grade proximity ($\uparrow$). For each model, **bold** = better of Base / SFT per column.

Model	Mode	Type (F1)						Edit quality			
		EDIT	ADD	DEL	EXEC	SUB	weighted	CodeBLEU	N_{match}	N_{inv}	GradeProx
Qwen3.5-0.8B	Base	0.535	0.167	0.122	0.515	0.285	0.452	0.777	3460	193	0.822
	SFT	0.723	0.190	0.102	0.613	0.413	0.589	0.794	4957	114	0.801
Qwen3.5-2B	Base	0.524	0.164	0.131	0.500	0.256	0.437	0.784	3698	127	0.824
	SFT	0.737	0.174	0.129	0.612	0.344	0.581	0.796	5198	84	0.797
Qwen3.5-4B	Base	0.521	0.196	0.127	0.534	0.287	0.453	0.747	3829	117	0.821
	SFT	0.761	0.315	0.120	0.624	0.349	0.601	0.795	5340	50	0.806

Models struggle to predict infrequent and terminal actions. Our results reveal a consistent performance gap between frequent and infrequent action types. Within modification actions, ADD and DELETE lag behind EDIT across all models and conditions, partly explained by their lower frequency in the training data (Figure 1). Beyond frequency, we also hypothesize that each action type reflects a structurally distinct situation: ADD requires that the student omitted an entire line in a prior edit, while DELETE requires that a previously written line contributes nothing and must be fully removed; both situations are rarer than the general case of writing incorrect content that needs replacement, which EDIT covers. Within execution actions, SUBMIT is consistently harder to predict than EXECUTE, suggesting that models learn students’ local Edit-Execute loop but struggle to determine when they consider their solution complete enough to submit.

7. Concluding Discussion

In this work, we presented a framework for modeling artificial programming students as tool-calling agents, grounded in event-stream process data. Rather than predicting programs as monolithic outputs, our framework derives a vocabulary of student actions from keystroke-level logs and casts those actions as tool calls in the agentic conversational format. We then formalized student modeling under this framework as a next-action prediction task, and validated the resulting setting on a large-scale dataset of CS1 student trajectories on Python programming assignments.

Our initial experiments addressed two research questions. On RQ1, we found that increasing base model size from 0.8B to 4B parameters yields no measurable improvement in next-action prediction (weighted F1 stays within 0.016), although larger models do recognize modification steps more reliably and stay within the scope of the program. On RQ2, we found that supervised fine-tuning consistently improves performance across most metrics (weighted F1 gains of 0.137 to 0.148 across sizes), yet these gains themselves do not scale reliably with model size.

Taken together, these findings suggest that model size alone is not the bottleneck for action-level student modeling, and motivate the training extensions discussed below. Importantly, we also note that sequential approaches focused on next-program prediction remain well suited when only snapshot data is available, and we do not aim to replace them entirely.

Limitations. Two main types of limitations apply to the current work.

First, on the theory side, our framework rests on several design choices that may not always hold. Notably, we group contiguous line-level events into a single action as a proxy for unobservable student intent; this choice also limits the length of the serialized trajectories. We acknowledge that the framework cannot accurately represent what a student is working toward at a given moment, nor capture the cognitive unit a student treats as a single editing decision. The framework is, however, flexible: practitioners can adapt the level of granularity and the grouping rule to their own understanding of

their classroom or their modeling objectives. As such, the next-action prediction task is bounded by the framework’s design choices, not by an intrinsic notion of what a student “does” at a single step. Additionally, our framework keeps the program state implicit: it is recoverable from the conversation history through tool outputs, but the model has no *explicit* representation of it as a structured object. A student’s next action is therefore implicitly conditioned on the program state, with prediction quality depending on how well the model attends to the relevant parts of its context window – something that may degrade across long trajectories.

Second, on the empirical side, our experiments cover a single dataset from one institutional context and a small set of language models from the same family. Moreover, we excluded outlier-length trajectories from training and evaluation, which means that students with the longest sessions, those most likely to struggle and most in need of intervention, are underrepresented in our study. We therefore cannot yet assess how findings generalize across programming environments, student populations, or the full range of student ability levels. Moreover, evaluation should extend beyond single next-action prediction to multi-step rollouts that assess whether artificial students remain behaviorally coherent over longer horizons. More fundamentally, we did not experimentally compare what the next-action and next-program prediction approaches each bring to downstream student-modeling applications.

Future work. Our future work will start by refining the framework and identify how to more meaningfully identify student-specific “action” intents from low-level stream data. We will then pursue training extensions along two directions. First, we will train models to produce short natural language thoughts before emitting the next tool call, following the ReAct framework [19]. Such thoughts would serve two purposes: improving prediction performance by making the reasoning process explicit, and giving practitioners a window into the strategies the model attributes to the simulated student. We will then investigate whether reinforcement learning with verifiable rewards techniques, specifically GRPO [39], can further improve both prediction quality and the interpretability of generated thoughts. We will also explore how to meaningfully integrate low-level keystroke data (e.g. pauses, cursor position, etc.) into the training pipeline. We believe for instance that such low-level behaviors could condition where model attentions should go during training, forcing not only the model outputs to align with the student “outputs” but also the model internal mechanisms. Finally, we will address the challenge of incorporating student process history into the prediction setting. A deployed artificial student must condition future action predictions on all prior assignment attempts, yet practical context window limits make naive concatenation infeasible. We will study various strategies to make such prediction possible, such as including KC-estimate summaries [16], building memory architectures [11], or learning summary embeddings [8].

Acknowledgments

This work was supported by Research Council of Finland grants #356114 and #367787. We acknowledge the computational resources provided by the Aalto Science-IT project.

Declaration on Generative AI

During the preparation of this work, the authors used Claude, ChatGPT, and Grammarly for grammar and spelling check. Further, the authors used Claude for generating (the Python code for) Figures 1 and 2. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication’s content.

References

- [1] S. A. Fincher, A. V. Robins (Eds.), The Cambridge Handbook of Computing Education Research, Cambridge University Press, Cambridge, UK, 2019.

- [2] A. Luxton-Reilly, Simon, I. Albluwi, B. A. Becker, M. Giannakos, A. N. Kumar, et al., Introductory programming: A systematic literature review, in: *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, ACM, New York, NY, USA, 2018, pp. 55–106.
- [3] K. VanLehn, S. Ohlsson, R. Nason, Applications of simulated students: An exploration, *Journal of Artificial Intelligence in Education* 5 (1994) 135–175.
- [4] A. H. Brown, Simulated classrooms and artificial students: The potential effects of new technologies on teacher education, *Journal of Research on Computing in Education* 32 (1999) 307–318.
- [5] N. Matsuda, W. W. Cohen, J. Sewall, G. Lacerda, K. R. Koedinger, Evaluating a simulated student using real students data for training and testing, in: *International Conference on User Modeling*, Springer, 2007, pp. 107–116.
- [6] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, et al., Language models are few-shot learners, in: *Proceedings of the 34th International Conference on Neural Information Processing Systems, NIPS '20*, Curran Associates Inc., Red Hook, NY, USA, 2020.
- [7] M. Miroyan, R. Niousha, J. E. Gonzalez, G. Ranade, N. Norouzi, ParaStudent: Generating and evaluating realistic student code by teaching LLMs to struggle, 2025. Preprint.
- [8] A. Ross, M. Srivastava, J. Blanchard, J. Andreas, Modeling student learning with 3.8 million program traces, *arXiv preprint arXiv:2510.05056* (2025).
- [9] B. C. Das, M. H. Amini, Y. Wu, Security and privacy challenges of large language models: A survey, *ACM Comput. Surv.* 57 (2025). URL: <https://doi.org/10.1145/3712001>. doi:10.1145/3712001.
- [10] F. D. Pereira, E. H. T. d. Oliveira, D. B. F. Oliveira, A. I. Cristea, L. E. R. Fernandes, L. Cunha, et al., Using learning analytics in the Amazonas: understanding students' behaviour in introductory programming, *British Journal of Educational Technology* 51 (2020) 955–972.
- [11] Y. Zhan, Q. Liu, W. Gao, Z. Zhang, T. Wang, S. Shen, et al., Coderagent: Simulating student behavior for personalized programming learning with large language models, in: *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2025*, Montreal, Canada, August 16–22, 2025, *ijcai.org*, 2025, pp. 293–301. URL: <https://doi.org/10.24963/ijcai.2025/34>. doi:10.24963/IJCAI.2025/34.
- [12] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. R. Narasimhan, et al., SWE-agent: Agent-Computer interfaces enable automated software engineering, in: *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.
- [13] F. J. M. Coelho, E. H. T. Oliveira, F. D. Pereira, D. B. F. Oliveira, L. S. G. Carvalho, E. J. P. Souto, et al., Learning analytics em cursos de introdução à programação: Uma mostra da universidade federal do Amazonas, *Revista Brasileira de Informática na Educação* 31 (2023) 1089–1127. URL: <https://doi.org/10.5753/rbie.2023.3334>. doi:10.5753/rbie.2023.3334.
- [14] Qwen Team, Qwen3.5-omni technical report, 2026. URL: <https://arxiv.org/abs/2604.15804>. arXiv:2604.15804.
- [15] N. Liu, Z. Wang, R. Baraniuk, A. Lan, Open-ended knowledge tracing for computer science education, in: Y. Goldberg, Z. Kozareva, Y. Zhang (Eds.), *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, Association for Computational Linguistics, Abu Dhabi, United Arab Emirates, 2022, pp. 3849–3862. doi:10.18653/v1/2022.emnlp-main.254.
- [16] Z. Duan, N. Fernandez, A. B. L. Narayanan, M. Hassany, R. S. de Alencar, P. Brusilovsky, et al., Automated knowledge component generation for interpretable knowledge tracing in coding problems, 2025. URL: <https://arxiv.org/abs/2502.18632>. arXiv:2502.18632.
- [17] Z. Duan, N. Fernandez, A. Hicks, A. Lan, Test case-informed knowledge tracing for open-ended coding tasks, in: *Proceedings of the 15th International Learning Analytics and Knowledge Conference, LAK '25*, Association for Computing Machinery, New York, NY, USA, 2025, p. 238–248. URL: <https://doi.org/10.1145/3706468.3706500>. doi:10.1145/3706468.3706500.
- [18] T. Zheng, G. Zhang, T. Shen, X. Liu, B. Y. Lin, J. Fu, et al., Opencodeinterpreter: Integrating code generation with execution and refinement, 2025. URL: <https://arxiv.org/abs/2402.14658>. arXiv:2402.14658.
- [19] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, Y. Cao, ReAct: Synergizing reasoning and

- acting in language models, in: The Eleventh International Conference on Learning Representations, 2023. URL: https://openreview.net/forum?id=WE_vluYUL-X.
- [20] P. Ihantola, A. Vihavainen, A. Ahadi, M. Butler, J. Börstler, S. H. Edwards, et al., Educational data mining and learning analytics in programming: Literature review and case studies, in: Proceedings of the 2015 ITiCSE on Working Group Reports, ACM, 2016, pp. 41–63. doi:10.1145/2858796.2858798.
 - [21] A. Carter, C. D. Hundhausen, D. Olivares, Leveraging the integrated development environment for learning analytics, in: S. A. Fincher, A. V. Robins (Eds.), The Cambridge Handbook of Computing Education Research, Cambridge University Press, 2019, pp. 679–706. doi:10.1017/9781108654555.024.
 - [22] J. Edwards, K. Hart, R. Shrestha, Review of cshedm data and introduction of two public cs1 keystroke datasets, Journal of Educational Data Mining 15 (2023) 1–31. doi:10.5281/zenodo.7646659.
 - [23] J. Leinonen, Keystroke data in programming courses (2019).
 - [24] E. Soloway, K. Ehrlich, Empirical studies of programming knowledge, IEEE Transactions on Software Engineering SE-10 (1984) 595–609. doi:10.1109/TSE.1984.5010283.
 - [25] R. S. Rist, Schema creation in programming, Cognitive Science 13 (1989) 389–414. doi:10.1207/s15516709cog1303_3.
 - [26] A. Vihavainen, M. Luukkainen, P. Ihantola, Analysis of source code snapshot granularity levels, in: Proceedings of the 15th Annual Conference on Information Technology Education, ACM, 2014, pp. 21–26. doi:10.1145/2656450.2656473.
 - [27] M. M. Villamor, A review on process-oriented approaches for analyzing novice solutions to programming problems, Research and Practice in Technology Enhanced Learning 15 (2020) 8. doi:10.1186/s41039-020-00130-y.
 - [28] Z. Xiong, Y. Cai, B. Hooi, N. Peng, K.-W. Chang, Z. Li, et al., Enhancing llm character-level manipulation via divide and conquer, arXiv preprint arXiv:2502.08180 (2025). arXiv:2502.08180.
 - [29] A. Cosma, S. Ruseti, E. Radoi, M. Dascalu, The strawberry problem: Emergence of character-level understanding in tokenized language models, in: Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, Suzhou, China, 2025, pp. 28252–28263. URL: <https://aclanthology.org/2025.emnlp-main.1434/>. doi:10.18653/v1/2025.emnlp-main.1434.
 - [30] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, et al., Training language models to follow instructions with human feedback, Advances in neural information processing systems 35 (2022) 27730–27744.
 - [31] J. Wei, G. Durrett, I. Dillig, Coeditor: Leveraging repo-level diffs for code auto-editing, arXiv preprint arXiv:2305.18584 (2024).
 - [32] S. P. Neshaei, R. L. Davis, A. Hazimeh, B. Lazarevski, P. Dillenbourg, T. K. Aaser, Towards modeling learner performance with large language models, in: B. P. Aÿen, C. D. Epp (Eds.), Proceedings of the 17th International Conference on Educational Data Mining, International Educational Data Mining Society, Atlanta, Georgia, USA, 2024, pp. 759–768. doi:10.5281/zenodo.12729942.
 - [33] Z. Duan, N. Fernandez, A. Lan, Kaser: Knowledge-aligned student error simulator for open-ended coding tasks, 2026. URL: <https://arxiv.org/abs/2601.06633>. arXiv:2601.06633.
 - [34] J. Macina, N. Daheim, S. Chowdhury, T. Sinha, M. Kapur, I. Gurevych, et al., MathDial: A dialogue tutoring dataset with rich pedagogical properties grounded in math reasoning problems, in: Findings of the Association for Computational Linguistics: EMNLP 2023, Association for Computational Linguistics, Singapore, 2023, pp. 5602–5621. URL: <https://aclanthology.org/2023.findings-emnlp.372/>. doi:10.18653/v1/2023.findings-emnlp.372.
 - [35] A. Scarlatos, R. S. Baker, A. Lan, Exploring knowledge tracing in tutor-student dialogues using LLMs, in: Proceedings of the 15th International Learning Analytics and Knowledge Conference (LAK 2025), ACM, Dublin, Ireland, 2025. URL: <https://dl.acm.org/doi/10.1145/3706468.3706501>. doi:10.1145/3706468.3706501.
 - [36] T. Dettmers, A. Pagnoni, A. Holtzman, L. Zettlemoyer, Qlora: efficient finetuning of quantized llms, in: Proceedings of the 37th International Conference on Neural Information Processing

Systems, NIPS '23, Curran Associates Inc., Red Hook, NY, USA, 2023.

- [37] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, et al., Scikit-learn: Machine learning in Python, *Journal of Machine Learning Research* 12 (2011) 2825–2830.
- [38] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, et al., Codebleu: a method for automatic evaluation of code synthesis, 2020. [arXiv:2009.10297](https://arxiv.org/abs/2009.10297).
- [39] D. Guo, D. Yang, H. Zhang, J. Song, P. Wang, Q. Zhu, et al., Deepseek-r1 incentivizes reasoning in llms through reinforcement learning, *Nature* 645 (2025) 633–638. URL: <https://doi.org/10.1038/s41586-025-09422-z>. doi:10.1038/s41586-025-09422-z.
- [40] F. D. Pereira, S. C. Fonseca, S. Wiktor, D. Oliveira, A. Cristea, A. Benedict, et al., Towards supporting cs1 instructors and learners with fine-grained topic detection in online judges (2022).
- [41] C. Leys, C. Ley, O. Klein, P. Bernard, L. Licata, Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median, *Journal of Experimental Social Psychology* 49 (2013) 764–766. doi:10.1016/j.jesp.2013.03.013.
- [42] D. Han, M. Han, Unsloth team, Unsloth, 2023. URL: <http://github.com/unslothai/unsloth>.
- [43] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, et al., Lora: Low-rank adaptation of large language models, in: *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022, OpenReview.net*, 2022. URL: <https://openreview.net/forum?id=nZeVKeeFYf9>.
- [44] Unsloth AI, Lora hyperparameters guide, 2024. URL: <https://docs.unsloth.ai/get-started/fine-tuning-llms-guide/lora-hyperparameters-guide>, accessed: 2025-12-23.
- [45] E. Worden, C. Heffernan, N. Heffernan, S. Sonkar, FoundationalASSIST: An educational dataset for foundational knowledge tracing and pedagogical grounding of LLMs, 2026. URL: <https://arxiv.org/abs/2602.00070>. [arXiv:2602.00070](https://arxiv.org/abs/2602.00070).

A. Experiment Details

A.1. Dataset Details

The CodeBench dataset is released at <https://codebench.icomp.ufam.edu.br/dataset/>. It spans multiple years, each with one or more semesters of data; we arbitrarily selected the first semester of 2023 (2023-1) as the training set and the first semester of 2024 (2024-1) as the test set to form a cross-semester evaluation protocol, following real-world machine learning practices. Each semester contains sessions recording the full keystroke log for one student solving one exercise, alongside grader feedback from test-case submissions.

Raw data. Problem descriptions are not available for all exercises; our analysis focuses on sessions of assignments for which problem descriptions were made available [40]. The two selected semesters contain 8,850 valid student-exercise sessions in total: 5,128 from 2023-1 and 3,722 from 2024-1, involving 448 unique students across 157 exercises and 15 classes. Exercises span eight topic areas including conditional structures, sequential programs, strings, loops, and arrays, all solved in Python. The grade distribution is heavily right-skewed: 91.3% of sessions end with a perfect score of 100, 8.2% with a score of 0, and the remainder with partial credit. We note that this is typical in CS1 datasets, as students often seek help, and most students who do not finish assignments drop out of courses early.

Serialization. Sessions were conducted using CodeMirror (<https://codemirror.net/>). We wrote a script, available in our released codebase, that converts those raw CodeMirror logs to tool-call sequences using the process described in Section 3.1. We additionally remove outlier-length trajectories using the median absolute deviation [41], keeping only sessions with fewer than median + $3 \times \text{MAD}$ tool calls. The resulting dataset contains 124,133 tool calls in total. Session lengths range from 1 to 41 tool calls (mean 13.2, median 11, std 9.3). Roughly, we have the following distribution of tools: EDIT: 38.5%, EXECUTE: 23.1%, SUBMIT: 17.5%, ADD: 12.9%, GIVE_UP²: 5.6%, DELETE: 2.3%. Figure 1 shows the distribution of tool calls across session length.

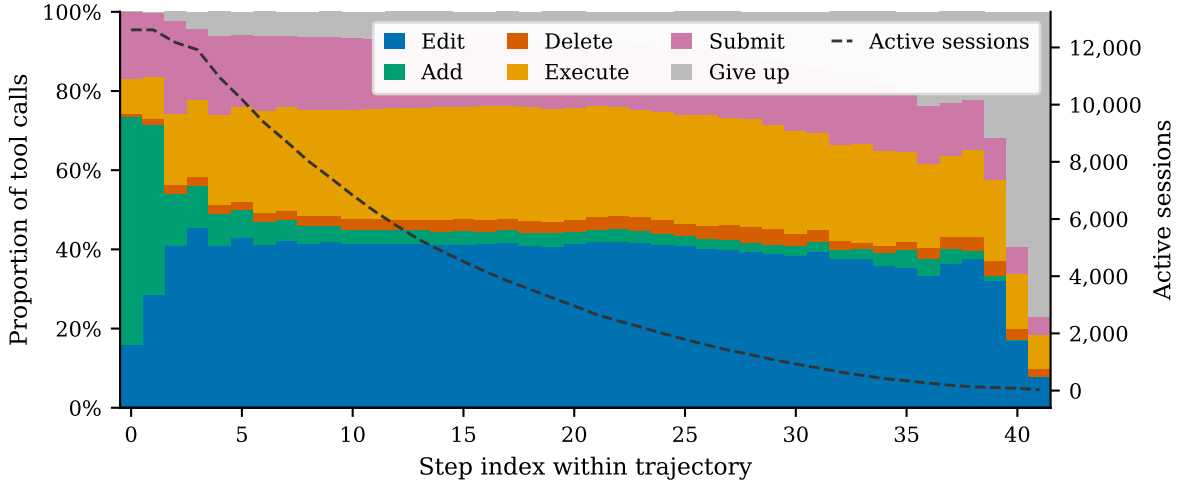


Figure 1: Distribution of tool-call types across trajectory positions (all three semesters, $N = 14,963$ sessions). Each bar shows the proportion of each action type at step i , normalised by the number of sessions still active at that step (3-step rolling average for visual clarity). The dashed curve (right axis) shows how many sessions remain active.

²We created a special tool “giving_up” to help the model understand when a student abandons the task, which stabilizes training. We leave the prediction and analysis of this action for future work.

Train/test split. Models are fine-tuned on the 2023-1 semester (5,128 sessions, 239 students, 76,383 tool calls) and evaluated on the 2024-1 semester (3,722 sessions, 227 students, 47,750 tool calls). Because semesters are conducted independently, student sets are disjoint by construction: no student appears in both the training and evaluation cohorts.

A.2. Training Details

We use Unsloth [42] with the TRL SFTTrainer to fine-tune all models on our institution’s research cluster (NVIDIA A100/H100, 80 GB, bfloat16).

All models are fine-tuned with LoRA [43], using unsloth default hyperparameter recommendations [44]: rank $r = 32$, $\alpha = 32$, dropout = 0, applied to all attention and feed-forward projection matrices (q, k, v, o, gate, up, down). Models are trained with a maximum context length of 8192 tokens; conversations exceeding this limit are dropped from the training and test sets rather than truncated.

All models are trained for one epoch with sequence packing enabled, an effective batch size of 16 (achieved via gradient accumulation), a peak learning rate of 2×10^{-4} with cosine decay and 10% linear warm-up, and gradient clipping at norm 1.0. We use Unsloth gradient checkpointing to reduce peak VRAM. A student-level 80/20 held-out split of the training data (no overlap with the test set) is used for in-training validation and best-model selection. Table 2 summarises the per-model effective batch size configuration.

Table 2

Per-model SFT configuration. Effective batch = batch size $\times$ gradient accumulation steps.

Model	Batch size	Grad. accum.	Peak VRAM (est.)
Qwen3.5-0.8B	4	4	~10 GB
Qwen3.5-2B	2	8	~18 GB
Qwen3.5-4B	2	8	~30 GB

A.3. Inference Details

We evaluate in *teacher-forcing* mode: for each assistant turn in a conversation, we feed the ground-truth prefix (all prior messages) to the model and ask it to predict the next tool call.

Bin-based sampling. Evaluating every turn in every test conversation is expensive and skews metrics toward long trajectories and the majority EDIT action. We instead use the bin-based sampling protocol of [45]: each trajectory of n assistant turns is divided into fixed windows of $B=5$ steps. The first window is skipped, as it is required as warm-start to provide enough context to the model to understand the student behavior. Within each remaining window, we sample at most two steps: the first modification-type step (i.e., EDIT/ADD/DELETE) and the first execution-type step (EXECUTE/SUBMIT), if each exists. This yields balanced coverage across action types and trajectory positions at approximately 60–70% lower cost than exhaustive evaluation.

Decoding. All evaluation runs use greedy decoding (temperature = 0) for reproducibility.

B. Illustrations

Figure 2 illustrates our serialization approach and compares it to the SWE-agent [12] architecture.

SYSTEM

You are a software engineering agent.

Expert framing; no persona or tool manifest

USER

ENVIRONMENT (ISSUE)

Fix the GitHub issue in the repository.
Bug: `compute_average()` crashes on empty list.
File: `utils/stats.py`, line 14

Issue = sole task context; no code provided

ASSISTANT

THOUGHT

The bug is in `stats.py` at line 14.
Let me inspect the function.

ACTION

`open_file("utils/stats.py", line=14)`

TOOL

FILE VIEWER

[`utils/stats.py` (20 lines total)]
...
14: `def compute_average(nums):`
15: `return sum(nums) / len(nums)`
...

ACI returns full file with line numbers

ASSISTANT

THOUGHT

No empty-list guard. I'll add one before the division.

ACTION

`edit(start=14, end=15,`
 `"if not nums: return 0`
 `return sum(nums) / len(nums)")`

TOOL

AFTER EDIT

[File updated. Lint: OK]
14: `def compute_average(nums):`
15: `if not nums: return 0`
16: `return sum(nums) / len(nums)`

Updated file + linting guardrail

ASSISTANT

THOUGHT

The edit looks correct. Let me run the test suite to confirm.

ACTION

`run_tests("tests/test_stats.py")`

TOOL

TEST RUNNER

.... 4 passed in 0.12s

ASSISTANT

All tests pass. I have fixed the issue.

No explicit submit; resolution declared in natural language

SYSTEM

You are a first-year Python student.
Tools: `add`, `edit`, `execute`, `submit`

Novice framing + tool manifest declared upfront

USER

ASSIGNMENT

Write `compute_average(nums)` that returns the mean of a non-empty list of integers.

Problem description in user turn; IDE starts empty

ASSISTANT

ACTION

`add(0,`
 `"def compute_average(nums):`
 `total = sum(nums)`
 `return average/len(nums)`

`if __name__ == '__main__':`
 `ns = list(map(int, input().split()))`
 `print(compute_average(ns))"`

TOOL

EDITOR WINDOW

[lines 1-7 | 7 total]
1: `def compute_average(nums):`
2: `total = sum(nums)`
3: `return average/len(nums)`
4:
5: `if __name__ == '__main__':`
6: `ns = list(map(int, input().split()))`
7: `print(compute_average(ns))`

Window view of the modified region; no linting guardrail

ASSISTANT

ACTION

`execute(inputs=[5, 3, 2])`

TOOL

OUTPUT

`NameError: name 'average' is not defined`

Student tests with own inputs, not a formal suite

ASSISTANT

ACTION

`edit(3, 3, " return total/len(nums)")`

TOOL

EDITOR WINDOW

[lines 1-3 | 7 total]
1: `def compute_average(nums):`
2: `total = sum(nums)`
3: `return total/len(nums)`

ASSISTANT

ACTION

`submit()`

TOOL

GRADER

Tests passed: 8/8 | Grade: 100%

Student may resubmit multiple times within the same session

Figure 2: Comparison of the SWE-agent ACI [12] and our proposed student agent framework on an illustrative Python debugging task. Background colours denote the four conversation roles: `system`, `user`, `assistant`, `tool`. **Left:** an expert agent navigates a repository, applies a targeted edit, verifies with a test suite, and declares the issue resolved in natural language. **Right:** an artificial student adds an initial draft, tests locally, fixes the bug, and submits — each action immediately followed by its tool turn.