Teaching Language Models How to Code Like Learners: Conversational Serialization for Student Simulation

Charles Koutcheme
Aalto University
Espoo, Finland
charles.koutcheme@aalto.fi

Juho Leinonen
Aalto University
Espoo, Finland
juho.2.leinonen@aalto.fi

Arto Hellas
Aalto University
Espoo, Finland
arto.hellas@aalto.fi

ABSTRACT

Artificial students—models that simulate how learners act and respond within educational systems—are a promising tool for evaluating tutoring strategies and feedback mechanisms at scale. However, most existing approaches rely on prompting large, proprietary language models, limiting adaptability to specific courses and raising concerns around privacy, cost, and dependence. In this work, we propose a framework for training open-weight artificial programming learners directly from authentic student process data. Our approach serializes temporal log traces into a conversational format, representing each student's problem-solving process as a dialogue between the learner and their automated assessment system. Student code submissions and environment feedback, such as test outcomes, grades, and error traces, form alternating conversational turns, enabling models to learn from the iterative debugging process. We additionally introduce a training pipeline combining supervised fine-tuning with preference optimization to align models with authentic student debugging behavior. We evaluate our framework by training Qwen models at 4B and 8B scales on a large-scale dataset of real student submissions to Python programming assignments. Our results show that incorporating environment feedback strengthens models' ability to replicate student debugging behavior, improving over both prior code-only approaches and prompted large language models baselines in functional alignment and code similarity. We release our code to support reproducibility.

Keywords

generative AI, language models, simulated students, process data, preference optimization

1. INTRODUCTION

To better support learners at scale, computing education research has long relied on models of students built from the rich log data collected as they solve programming assignments [17, 46]. Much of this work focuses on capturing what students know, such as estimating mastery over concepts through knowledge tracing [7, 18, 44], or identifying misconceptions that explain incorrect outcomes [40]. These models provide valuable insights into learning progress and have informed the design of successful interventions [3]. However, while such approaches are effective for analyzing student cognition and predicting outcomes, operationalizing interventions based on these models remains challenging. In many settings, what is needed is not only an estimate of what a student knows, but a model of how a student behaves: how they act, react, and progress when interacting with learning systems. This has motivated growing interest in *artificial students*—models that act as stand-ins for real learners by generating plausible sequences of actions [43, 26, 5]. When realistic, artificial students could enable the simulation of student–system interactions and support controlled experimentation, such as comparing tutoring strategies, evaluating feedback policies, or assessing the difficulty of programming tasks, as already done in prior works [32, 12, 29], without requiring repeated classroom deployments. Despite this promise, existing artificial student models remain limited, particularly in programming education. Recent work using large language models has shown that it is possible to generate student-like code [23, 25], but these approaches often rely on prompting alone and tend to reproduce surface-level error patterns rather than the dynamic nature of real student debugging. In particular, existing methods remain limited in their ability to realistically simulate how learners tackle programming assignments.

In this work, we leverage the rich process data already collected by programming learning environments to train small and medium size open-weight models for *within-assignment student simulation* [27, 35]. Since student data is sensitive and institutional, this offers a more practical path than prompting proprietary systems [35], keeping data local while learning the iterative debugging patterns that prompting alone cannot reliably elicit. Our core innovation is to represent each problem-solving trajectory as a dialogue between a student and their learning environment. Student code submissions are encoded as assistant turns, while outputs from automated assessment systems (e.g., unit-test outcomes, or grader messages) form the corresponding user turns. By adopting the conversational format commonly used to interact with language models, we turn raw log traces into structured sequences that capture how a student's code evolves in response to feedback. This representation enables chat-capable language models to learn from iterative debugging sequences without requiring architectural modifications.

To train artificial students from this conversational representation, we propose a pipeline combining supervised fine-tuning with offline preference optimization via Direct Preference Optimization [33]. We additionally explore online preference optimization methods based on Group Relative Preference Optimization [50]. We evaluate our framework on FalconCode [9], a large-scale dataset of student submissions to Python programming assignment, training Qwen models [41] at 4B and 8B scales. Our results show that incorporating environment feedback strengthens models' ability to replicate student debugging behavior, improving over both prior code-only baselines and prompted large language models in functional alignment and code similarity.

Summary contributions. Our main contributions are:

- **Student-environment serialization.** We introduce a conversational formulation of student–environment interactions that enables language models to learn students iterative debugging processes.
- **A training methodology for artificial students.** We propose a pipeline combining supervised fine-tuning and preference optimization to align language models with authentic student coding trajectories.
- **Empirical evaluation on real student data.** We validate our framework on a large-scale dataset of student programming submissions, showing that environment feedback and preference optimization yield complementary gains in simulation realism. We also release our code: `KoutchemeCharles/edm-conv-ser`

2. RELATED WORK

A complementary line of work focuses on *knowledge tracing* (KT), whose primary goal is to estimate a student's mastery of knowledge components across exercises. While early approaches focused exclusively on predicting success on future assignments, more recent methods leverage language-model–based architectures to predict students' first submission to following assignments and the associated outcomes [24, 13, 15]. While such methods may predict student programs, this prediction serves mainly as an intermediate signal for updating knowledge estimates across programming tasks, not as a goal in itself. In contrast, we aim to obtain models that can be unrolled over multiple steps within a single assignment to generate and simulate problem-solving trajectories. Closer to our work in the educational domain is Miroyan et al. [27], who train open-weight language models using supervised fine-tuning to predict a student's next submission to Python programming assignments given a limited context of prior attempts. Their work shows that trained language models perform better than prompted LLMs. Ross et al. [35] go further by fine-tuning pre-trained language models on 3.8 million traces from a block-based educational programming platform, showing that models trained on real edit sequences yield richer representations of student behavior than those trained on final programs alone. However, neither approach incorporates the learning environment's responses to student submissions. Yet, serializing structured data into conversational format for language model training is well-established in NLP, with extensive work on synthetic dialogue generation [28, 11, 49] and math-informed dialogues [1].

Closest to our approach is OpenCodeInterpreter [51], which fine-tunes open-weight language models on multi-turn code-execution dialogues to improve code generation and refinement abilities. While their approach also serializes execution feedback into multi-turn dialogs, our work differs in two key aspects. First, our goal is to reproduce student problem-solving behavior, not to improve code generation; we train on authentic student trajectories rather than synthetic expert-generated interactions refined through execution outputs. Second, we develop preference optimization methods specifically adapted to student programming trajectories, leveraging infrastructure tied to learning environments, in particular the autograder, to construct preference pairs for training.

3. FROM LOGS TO DIALOGS

In this section, we introduce our approach for transforming student log data into suitable data for student simulation. Our work assumes that students interact with an automated assessment system returning summative feedback [6].

3.1 Assumptions

We assume access to a deterministic grading function which, for a submitted program a to a programming assignment d, can return two observable components: $v_r^{(d)}(a) = r$ and $v_f^{(d)}(a) = o$. The numerical score $r \in [0, 1]$ measures the correctness of the executed program (e.g., proportion of passed test cases). The textual feedback o contains the textual outcome of executing that submission: summaries of passed tests, numerical grades, and/or even runtime error traces, depending on what each dataset provides. We do not assume any internal structure of the (summative) feedback, only that it is consistently produced and directly observable in the logs. We also assume access to a dataset of institutional log data containing sequences of graded submissions and their outcomes: $\mathcal{D} = \{(d^i, s^i)\}_{i=1}^N$, where d^i denotes an assignment textual description and $s^i = (u_1^i, u_2^i, \ldots, u_{T_i}^i)$ denotes the sequence of graded interactions for a single student on that assignment. Each entry $u_t^i = (a_t^i, o_t^i)$ contains the submitted program a_t^i and its outcome $o_t^i = v_f^{(d^i)}(a_t^i)$. The trajectory length T_i equals the number of graded submissions recorded for assignment d^i.

3.2 Serialization

We represent student programming logs using the standard conversational format used to train and interact with chat-capable language models [31]. We structure each trajectory as a dialogue where the `user` role represents the automated assessment system and the `assistant` role represents a student learning how to program. The `system` prompt establishes the context: the model is a novice student learning to program, solving assignments while interacting with a learning environment that provides summative feedback. The initial `user` message provides the assignment description d_i and any relevant context. Each logged interaction $u_t^i = (a_t^i, o_t^i)$ then maps to two conversational turns: the student's code submission a_t^i appears as an `assistant` turn, followed by the grading environment's feedback o_t^i as the next `user` turn. This alternating structure naturally captures the iterative debugging process: students write code, receive feedback, then revise based on that feedback. Below is an illustration using a simple Python assignment:

System

```
You are a first-year novice student learning
programming in Python. Solve the given programming
assignment(s). You will be interacting with a learning
environment which will provide you with summative feedback.
```

User

```
Write a Python function called "compute_average".
The function should take as argument a (non-empty) list of
integers and returns the mean over those elements.
```

Assistant

```
def compute_average(nums):
    total = nums[0]
    for i in nums:
        total += i
    return average / len(nums)
```

User

```
Runtime error: undefined variable "average".
```

Assistant

```
def compute_average(nums):
    total = nums[0]
    for i in nums:
        total += i
    return total / len(nums)
```

User

```
Tests passed: 1/8
```

Assistant

```
def compute_average(nums):
    total = 0
    for i in nums:
        total += i
    return total / len(nums)
```

User

```
Tests passed: 8/8
```

4. TRAINING ARTIFICIAL LEARNERS

In this section, we present our training pipeline for training a language model π_θ to simulate how programming students solve assignments. Our core pipeline combines supervised fine-tuning with offline preference optimization on a serialized dataset $\mathcal{D}$. We additionally explore online preference optimization as an alternative to the offline stage.

4.1 Supervised Fine-tuning

The first method is to simply supervised finetune our model π_θ using the negative log likelihood (i.e., SFT) objective:

$$\mathcal{L}_{\text{SFT}}(\theta, \mathcal{D}) = -\sum_{s^i \in \mathcal{D}} \sum_{t=1}^{T_i} \log \pi_\theta\left(a_t^i \mid u_{<t}^i\right).$$

Following prior work, we only backpropagate the loss on assistant turns [10] containing student code submissions. The supervised fine-tuning step enables models to quickly learn patterns of student problem-solving.

4.2 Offline Preference Optimization

We use Direct Preference Optimization [33] (DPO), an offline preference optimization algorithm that aligns language models using pairwise preference datasets. This algorithm has shown much success in applied AI in education [21, 48, 38]. In this work, we construct preference datasets by forming contrastive pairs of candidate continuations from the same student trajectory. Let $u_{\leq t}^i$ denote a sampled partial trajectory of a student up to step t. The preferred continuation is the student's immediate next submission a_{t+1}^i, while the dispreferred continuation is the first following submission with a different grade. Formally, if we define:

$$k^\star(i,t) = \min\left\{k \in [2, T_i - t] : v_r^{(d)}(a_{t+k}^i) \neq v_r^{(d)}(a_{t+1}^i)\right\}.$$

then the resulting preference dataset is :

$$\mathcal{D}_{\text{DPO}} = \left\{\left(u_{\leq t}^i, a_{t+1}^i, a_{t+k^\star(i,t)}^i\right)\right\}_{i=1,\ldots,N;\ t \in \mathcal{T}_i}.$$

We believe $a_{t+k^\star(i,t)}^i$, the nearest semantically distinct code the student actually wrote, provides a high-quality contrast that teaches the model *why* students submit specific solutions at each step, preventing models from jumping prematurely to fully correct solutions, a pattern observed in prior work [27]. Using the resulting dataset, we optimize our model using the DPO loss [33]:

$$\mathcal{L}_{\text{DPO}}(\theta) = -\mathbb{E}_{(u_{\leq t}^i, a_{t+1}^i, a_{t+k}^i) \sim \mathcal{D}_{\text{DPO}}} \left[\log \sigma\big(r_\theta(u_{\leq t}^i, a_{t+1}^i) - r_\theta(u_{\leq t}^i, a_{t+k^\star(i,t)}^i)\big)\right] \quad (1)$$

where σ is the sigmoid function and

$$r_\theta(u, a) \triangleq \beta \log \frac{\pi_\theta(a \mid u)}{\pi_{\text{ref}}(a \mid u)}. \quad (2)$$

with π_θ being the model being optimized and π_{ref} being the reference policy (the model before the start of the optimization (often π_{SFT})). In essence, Equation (1) penalizes the model based on how much more it prefers the dispreferred generation over the preferred one using an implicit reward (Equation (2)), defined by the model probabilities, with β controlling how much the trained model weights θ can deviate from their initialization point (e.g., the SFT weights).

4.3 Online Preference Optimization

We additionally explore whether online preference optimization can exceed offline methods for student simulation. Specifically, we use a variant of Group Relative Preference Optimization (GRPO) [16], an online algorithm that iteratively (1) samples multiple candidate next-step programs from the current policy, (2) evaluates them through a reward function, and (3) updates the model according to their relative quality. At each iteration, given a trajectory prefix $u_{\leq t}^i$, we sample $G = 4$ candidate next-step programs using top_p sampling. Each candidate is scored using a discrete reward that captures two complementary aspects [19] of next-step program prediction: functional alignment and syntactic similarity. Candidates whose abstract syntax tree exactly matches the student's actual next submission receive the highest reward (+2.0). Candidates that do not match exactly but achieve the same grade as the student's next attempt receive a smaller reward (+1.0). Candidates that fail to compile are penalized (-1.0), while all remaining candidates receive no reward. This tiered design prioritizes reproducing the student's specific solution while still rewarding functionally equivalent behavior, and aligns with outcome-based reward practices in code generation [22]. We update model parameters using DAPO [50], a more stable version of the original GRPO algorithm. Due to space constraints, we refer the reader to the original paper and our code base for the full loss formulation.

5. EXPERIMENTS

In this section, we detail the components of our experiments aimed at evaluating the utility of our framework.

5.1 Dataset

We evaluate our framework using FalconCode [9], a large-scale CS1 Python programming dataset from the United States Air Force Academy. The dataset includes student submissions, the associated grades, and the course autograder, which we used to regenerate the same detailed summative feedback students received during the course, showing pass/fail and expected outputs per test case. FalconCode contains three assignment difficulty levels: "skill", "lab", and "project". We excluded project assignments (lacking automated tests) and skill assignments (1–10 lines of code). Lab assignments represent medium-sized programs (10–50 lines). To manage the costs of running our experiments, we arbitrarily selected the assignment with the highest number of submissions from each lesson regrouping assignments targetting similar concepts. We retained trajectories where students passed at least one test, as all-failing trajectories often reflect non-serious attempts. We further removed non-compiling submissions but retained runtime errors and traces as environment feedback. We leave the handling of syntax errors to future work. FalconCode spans three semesters. To mimic realistic deployment, we use data from Spring 2021 for training, and evaluate on a randomly sampled susbet of 1000 trajectories from Spring 2022.

5.2 Models

We fine-tune two open-weight language models from the Qwen family [41]: `Qwen3-4B-Instruct` and `Qwen3-8B`. We selected these models for their strong performance on coding benchmarks and to study model size effects on student simulation quality.

Model variants. We evaluate variants to isolate the contribution of our framework components. As a baseline to prior work [27], we train models *without* environment feedback (**PARA**). Within our framework, we compare **SFT** (supervised fine-tuning), **DPO** (offline preference optimization), and **DAPO** (online preference optimization). For both preference optimization, we start training from the SFT weights.

Training setup. We use `unsloth` [8] to enable long-context fine-tuning on NVIDIA V100 GPUs with 32 GB of VRAM, using `triton`, our institution research cluster. All models are fine-tuned with QLoRA [10], following established hyperparameter recommendations [39, 42]: rank $r = 8$, $\alpha = 16$. Models are trained with a maximum context length of 4096 tokens. When trajectories exceed this limit, we remove earlier assistant turns until the sequence fits, creating a dynamic sliding window over recent submissions. For PARA models, unlike [27] who use a fixed window of $k = 4$ prior submissions, we dynamically limit prior submissions based on the context window size such that all trained models operate under the same computational constraints. We use a cosine learning rate of 1×10^{-4} for SFT variants (including PARA) (one epoch) and 1×10^{-5} for preference optimization, with $\beta = 0.5$ for DPO following [38]. For SFT variants and DPO, we select the best checkpoint based on validation loss from 20% of training data. For DPO, we sample all positions from all available trajectory prefixes from the training set. For GRPO, we sample two positions from each trajectory due to computational costs, generating $G = 4$ possible attempts per trajectory.

In-context learning baselines. To assess the value of fine-tuning, we include in-context learning (ICL) baselines using **BASE** model variants without training. We also evaluate the proprietary `GPT-5-mini` [30] model under the *minimal* reasoning configuration (see OpenAI model documentation).

5.3 Evaluation

For each test trajectory, we evaluate model predictions at all possible starting positions $t > 2$. Given a partial history $u^i_{\leq t}$, we autoregressively roll out the model for up to 5 steps using greedy-decoding. At each rollout step $j \in \{1, \ldots, 5\}$, the model generates a submission $\hat{a}^i_{t+j}$ conditioned on the original history and all previously generated rollout steps. Each generated submission is evaluated by the grading environment to produce the outcome $\hat{o}^i_{t+j} = v_f^{(d^i)}(\hat{a}^i_{t+j})$, which is then appended to the context for subsequent predictions. Rollout terminates early if there is no corresponding ground truth student generation or if the model generates a fully correct solution.

Metrics. We report three complementary metrics averaged across all predictions. First, **coverage** measures the proportion of ground-truth student submissions with model predictions. Coverage is less than 1.0 when models prematurely submit a correct solution (at step k where $v_r^{(d^i)}(\hat{a}^i_{t+k}) = 1$) while the student has not yet finished ($v_r^{(d^i)}(a^i_{t+k}) < 1$), a known challenge in student simulation [27]. For matched pairs where models did generate, we evaluate generation quality using two metrics: (1) **CodeBLEU** [34], measuring token-level syntactic and semantic similarity [15, 13, 24]; and (2) **grade proximity**, computed as $1 - |v_r^{(d)}(\hat{a}^i_{t+j}) - v_r^{(d)}(a^i_{t+j})|$, where higher values indicate closer functional alignment. These quality metrics are computed only on matched pairs where models did generate, to avoid biasing results against models that correctly stop at perfect grades. Conversely, this means models with different coverage levels have their quality metrics computed on different susbet of predictions, which should be kept in mind when comparing across methods. To assess robustness over rollouts, we also report **average degradation** Δ for each metric, which we compute as $\Delta m = \frac{1}{K-1} \sum_{k=2}^{K} (m_k - m_1)$, where m_k denotes metric performance at rollout step k and $K = 5$. More negative values indicate faster degradation from the first step.

6. RESULTS

Table 1 summarizes our training and test split statistics. Table 2 reports coverage and generation quality metrics averaged across all rollout steps. Figure 1 details performance at each rollout step, and Figure 2 illustrates how model-generated grades evolve compared to students ground truth grades. We highlight several key findings below.

Table 1: FalconCode dataset statistics. Summary of training and test splits. #Traj: total trajectories; #Stud: unique students; #Succ/#Fail: trajectories with final grade 100%/below 100%; #Asg: unique programming assignments; Avg. Len: mean submissions per trajectory;

Split	#Traj	#Stud	#Succ	#Fail	#Asg	Avg. Len
Train	1762	448	1712	50	17	22.4
Test	1000	384	981	19	15	20.1

Table 2: Rollout performance metrics. Legend: Cov (Coverage), GP (Grade Proximity), CB (CodeBLEU). Metrics averaged across k=1 to k=5 steps ahead, with degradation (Δ) showing the average drop from k=1 to k=2..5. Negative Δ values indicate performance worsens over longer rollouts. Bold: best performance within each model.

		Average			Degradation		
Model	Method	Cov	GP	CB	ΔCov	ΔGP	ΔCB
Qwen3-4B	BASE	0.528	0.690	0.491	-0.640	**0.278**	**-0.063**
	PARA	0.918	0.763	0.695	-0.118	-0.071	-0.137
	SFT	0.922	0.782	**0.704**	-0.107	-0.060	-0.138
	DPO	**0.982**	**0.797**	0.700	**-0.024**	-0.079	-0.137
	DAPO	0.859	0.781	0.700	-0.203	-0.068	-0.138
Qwen3-8B	BASE	0.570	0.733	0.581	-0.581	**0.287**	**-0.046**
	PARA	0.921	0.785	0.714	-0.112	-0.059	-0.121
	SFT	0.921	0.793	**0.718**	-0.110	-0.062	-0.126
	DPO	**0.957**	**0.796**	0.714	**-0.061**	-0.074	-0.129
	DAPO	0.919	**0.796**	0.709	-0.116	-0.078	-0.133
GPT-5-mini	BASE	**0.340**	**0.549**	**0.433**	**-0.910**	**0.184**	**-0.078**

Prompted baselines cannot simulate realistic student behavior. BASE Qwen models achieve substantially lower coverage (0.528 for 4B, 0.570 for 8B), grade proximity (0.690, 0.733), and CodeBLEU (0.491, 0.581) than any fine-tuned model. GPT-5-mini performs worst overall, with a coverage of only 0.340 and grade proximity of 0.549. Its low coverage indicates that it frequently generates fully correct solutions that terminate trajectories after one or two steps, while its low CodeBLEU (0.433) confirms that the code it does produce has little resemblance to student submissions. Interestingly, BASE Qwen models outperform GPT-5-mini, possibly because their smaller capacity leads to genuine mistakes that incidentally resemble student errors, whereas GPT-5-mini's stronger coding ability defaults to expert-like solutions despite the explicit personification prompt.

Environment feedback improves student simulation. SFT performs as well as or better than PARA on all three metrics for both models (Table 2). The margins are modest but the pattern is consistent. This is notable because PARA has a structural advantage: without environment feedback in the context, it can fit more prior student submissions within the same token budget. Despite this, environment feedback proves more informative than additional code history. We hypothesize that earlier submissions, particularly from already-passed test stages, carry diminishing signal about the student's current debugging strategy, whereas grader feedback directly conditions the next revision.

Offline preference optimization yields the strongest results. DPO achieves the highest coverage across both model sizes (0.982 for 4B, 0.957 for 8B), producing valid code at more rollout steps than any other method. It also achieves the highest grade proximity for 4B (0.797 vs. 0.782 for SFT). SFT achieves slightly higher CodeBLEU (0.704 vs. 0.700 for 4B; 0.718 vs. 0.714), although this gap may partly reflect DPO's higher coverage: by generating at more rollout steps, DPO is evaluated on a broader and likely harder subset of predictions. DAPO shows mixed results: it matches DPO on grade proximity for 8B but achieves the lowest coverage among fine-tuned models on 4B (0.859), and does not consistently improve over SFT.

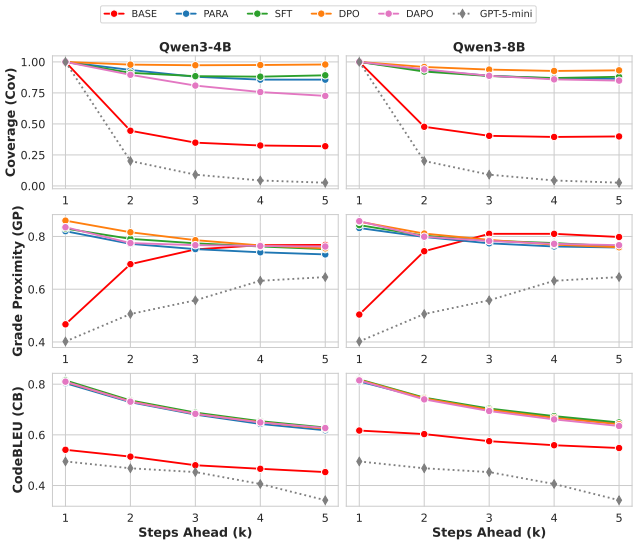

Figure 1: Performance across rollout steps. Coverage (top), Grade Proximity (middle) and CodeBLEU (bottom) as a function of autoregressive steps ahead (k=1 to k=5).

Performance over steps and trajectory position. Looking at Figure 1, BASE model coverage drops sharply ($\Delta = -0.640$ for 4B, -0.581 for 8B), as they frequently generate solutions that terminate the trajectory prematurely. Trained models degrade less, with DPO maintaining the highest coverage throughout (-0.024 for 4B, -0.061 for 8B), while DAPO coverage reduces more substantially over time (-0.203 for 4B). All fine-tuned models exhibit similar grade proximity degradation across rollout steps, with drops of approximately 0.06–0.08 from k=1 to k=5. The pattern over size is nuanced: for the 4B model, PARA degrades faster than SFT on grade proximity (-0.071 vs. -0.060), but for 8B, PARA degrades the least (-0.059) while preference-optimized models degrade the most (DPO: -0.074, DAPO: -0.078). Despite this, DPO maintains a higher absolute grade proximity compared to PARA at every rollout depth. Figure 2 shows that DPO-trained models most closely follow real student grade progression throughout normalized trajectories. DAPO models in contrast struggle to track students progressions. BASE models exhibit the opposite pattern: their grade proximity *improves* over rollout steps (+0.278 for 4B, +0.287 for 8B), and GPT-5-mini shows the same trend (+0.184). Figure 2 reveals an interesting pattern: untrained models generate solutions that achieve similar grades independently of trajectory position. We hypothesize that these models attempt to generate correct solutions, rather than adapting to the student's current progress. The difference is that GPT-5-mini largely succeeds, producing near-perfect grades throughout, while BASE Qwen models fail enough test cases that the corresponding grades happen to align with students grades at later trajectory positions. Neither reproduces the progressive improvement characteristic of real students. CodeBLEU shows more uniform downward trends across all trained models (≈ -0.13), suggesting that syntactic divergence accumulates at a similar rate regardless of training method. BASE model CodeBLEU remains comparatively stable (-0.063 for 4B, -0.046 for 8B).

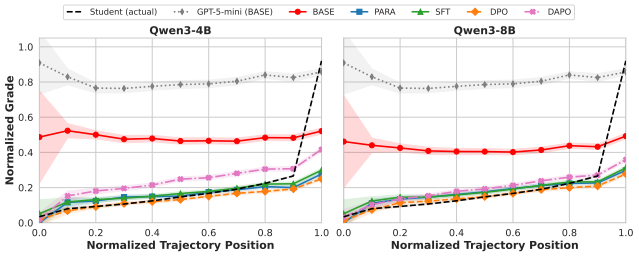

Figure 2: Grade progression across normalized trajectory position. Comparison of predicted grades between artificial student models and real students (black dashed, the average). The x-axis represents normalized position (by length) within a student's problem-solving trajectory (0 = start, 1 = end). Shaded regions indicate 95% confidence intervals. The ground truth line does not reach 1.0 as some students failed. The abrupt jump reflects the proportion of students who resolve multiple failing test cases in a single revision.

7. CONCLUDING DISCUSSION

Our experiments show that conversational serialization of student–environment interactions, combined with preference optimization, produces artificial students that more closely track real learners' debugging behavior than prompted baselines and models trained without feedback. Performance improvements compared to baselines are consistent in direction across both model sizes and all rollout steps. We believe coverage and grade proximity are particularly important dimensions for student simulation: models that generate code at every trajectory step, passing and failing the same tests at the same stages as real students, capture the incremental nature of real problem-solving more faithfully than models optimizing for surface-level code similarity alone. Among our findings, offline preference optimization via DPO provides the largest gains on these dimensions, achieving the highest coverage and grade proximity across both model sizes. Online preference optimization (DAPO) on the other hand does not consistently improve over SFT. We hypothesize that DAPO's weaker results may stem from (a) our computational constraint of sampling only 2 prefixes per trajectory compared to DPO's exhaustive coverage, and (b) the absence of chain-of-thought reasoning [47] prior to code generation, which online methods typically rely on for effective optimization. Our results thus suggest that without explicit reasoning traces, offline methods may remain more practical than recent online alternatives for student simulation.

While our experiments focus on within-assignment simulation, we hope the proposed framework will support several existing research directions.

Bridge with knowledge tracing and tutoring. Knowledge tracing (KT) models estimate a learner's latent proficiency across assignments, often predicting whether a student will solve a future task on the first attempt. Our framework operates at a complementary granularity: modeling the fine-grained debugging steps and intermediate attempts that precede success within a single task. We envision connecting these two levels by using cross-problem KC estimates as conditioning inputs to our simulation prompt [14], allowing our artificial students to exhibit more realistic learner behavior.

Bridge with tutoring. Moreover, our conversational serialization treats the automated assessment system as a dialogue partner that provides summative feedback. This is structurally identical to how a tutor agent would interact with a student. By representing student–environment interactions in this format, we could seamlessly extend sequences to include student–tutor exchanges, combining debugging traces with tutoring interactions [2] in a single sequence. This opens several directions: our models could be adapted to predict student success conditioned jointly on debugging attempts and tutor exchanges [37], or integrated into conversational knowledge tracing frameworks [36]. More broadly, artificial students trained with our approach could serve as scalable simulators for reinforcement-learning–based tutor training [12], where student success provides a training signal of tutoring strategy effectiveness.

Limitations. Our work is not free of limitations. We experimented with a single Python programming dataset and one model family across two sizes. Validation on additional datasets from different institutional contexts (with different autograder feedback styles) and programming languages is needed. Our evaluation data is heavily skewed toward successful students: 95% of trajectories end with a perfect score, meaning our results primarily reflect simulation quality for learners who ultimately succeed. This skew is partly inherent to programming courses, where most students eventually pass [45] and those who do not often abandon the course entirely [4, 20]. However, struggling students are precisely those most in need of intervention; modeling their behavior, including trajectory abandonment, remains important future work. We also report results using greedy decoding on a single semester of test data without statistical testing; evaluating under varied sampling strategies and data splits would strengthen confidence in our findings. Finally, while our metrics demonstrate that models replicate student grade trajectories and code similarity, one needs to validate whether these new artificial students will be effectively useful for the downstream applications we motivate, such as tutor training or intervention testing. Establishing this educational validity is important future work.

Future work. Beyond addressing the highlighted limitations, there are three direct next steps for student simulation. First, we will explore agent-based artificial students equipped with execution, submission, and editing tools, mirroring the interactive environments real students use when programming. Second, we will improve our online preference optimization method. GRPO-style methods are known to improve model performance when elicited to produce chain-of-thought reasoning. By prompting models to generate explicit student-like thought traces in between tool operations, we hope to improve such models' ability to follow students' processes while potentially yielding post-analysis insights into why students code the way they do. Lastly, we will explore methods to address the challenge of efficiently integrating students' prior history (i.e., their attempts at prior assignments) to enable better personalization.

8. REFERENCES

[1] S. N. Akter, S. Prabhumoye, J. Kamalu, S. Satheesh, E. Nyberg, M. Patwary, et al. MIND: Math informed synthetic dialogues for pretraining LLMs. In *The*

Thirteenth International Conference on Learning Representations, 2025.

[2] N. Ashok Kumar and A. Lan. Improving socratic question generation using data augmentation and preference optimization. In *Proceedings of the 19th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2024)*, pages 108–118, Mexico City, Mexico, June 2024. Association for Computational Linguistics.

[3] D. Azcona and A. F. Smeaton. Targeting at-risk students using engagement and effort predictors in an introductory computer programming course. In *European Conference on Technology Enhanced Learning*, pages 361–366. Springer, 2017.

[4] J. Bennedsen and M. E. Caspersen. Failure rates in introductory programming: 12 years later. *ACM Inroads*, 10(2):30–36, Apr. 2019.

[5] 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(2):307–318, 1999.

[6] D. L. Butler and P. H. Winne. Feedback and self-regulated learning: A theoretical synthesis. *Review of Educational Research*, 65:245–281, 1995.

[7] A. T. Corbett and J. R. Anderson. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User Modeling and User-Adapted Interaction*, 4(4):253–278, 1994.

[8] M. H. Daniel Han and U. team. Unsloth, 2023.

[9] A. de Freitas, J. Coffman, M. de Freitas, J. Wilson, and T. Weingart. Falconcode: A multiyear dataset of python code samples from an introductory computer science course. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2023, page 938–944, New York, NY, USA, 2023. Association for Computing Machinery.

[10] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer. Qlora: Efficient finetuning of quantized llms. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, page 441, Red Hook, NY, USA, 2023. Curran Associates Inc.

[11] N. Ding, Y. Chen, B. Xu, Y. Qin, S. Hu, Z. Liu, et al. Enhancing chat language models by scaling high-quality instructional conversations. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3029–3051, 2023.

[12] D. Dinucu-Jianu, J. Macina, N. Daheim, I. Hakimi, I. Gurevych, and M. Sachan. From problem-solving to teaching problem-solving: Aligning LLMs with pedagogy using reinforcement learning. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 272–292, Suzhou, China, Nov. 2025. Association for Computational Linguistics.

[13] Z. Duan, N. Fernandez, A. Hicks, and 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, page 238–248, New York, NY, USA, 2025. Association for Computing Machinery.

[14] Z. Duan, N. Fernandez, and A. Lan. Kaser: Knowledge-aligned student error simulator for open-ended coding tasks, 2026.

[15] Z. Duan, N. Fernandez, A. B. L. Narayanan, M. Hassany, R. S. de Alencar, P. Brusilovsky, B. Akram, et al. Automated knowledge component generation for interpretable knowledge tracing in coding problems, 2025.

[16] D. Guo, D. Yang, H. Zhang, J. Song, P. Wang, Q. Zhu, et al. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645:633–638, 2025.

[17] M. C. Jadud. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the second international workshop on Computing education research*, pages 73–84, 2006.

[18] J. Kasurinen and U. Nikula. Estimating programming knowledge with bayesian knowledge tracing. *ACM SIGCSE Bulletin*, 41(3):313–317, 2009.

[19] C. Koutcheme, N. Dainese, and A. Hellas. Direct repair optimization: Training small language models for educational program repair improves feedback. In *Proceedings of the 20th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2025)*, pages 564–581, Vienna, Austria, July 2025. Association for Computational Linguistics.

[20] C. Koutcheme, S. Sarsa, A. Hellas, L. Haaranen, and J. Leinonen. Methodological considerations for predicting at-risk students. In *Proceedings of the 24th Australasian Computing Education Conference*, ACE '22, page 105–113, New York, NY, USA, 2022. Association for Computing Machinery.

[21] C. Koutcheme, J. Woodrow, and C. Piech. Aligning small language models for programming feedback: Towards scalable coding support in a massive global course. In *Proceedings of the 57th ACM Technical Symposium on Computer Science Education V.1*, SIGCSE TS 2026, page 610–616, New York, NY, USA, 2026. Association for Computing Machinery.

[22] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning, Nov. 2022. arXiv:2207.01780 [cs].

[23] J. Leinonen, P. Denny, O. Kiljunen, S. MacNeil, S. Sarsa, and A. Hellas. Llm-itation is the sincerest form of data: Generating synthetic buggy code submissions for computing education. In *Proceedings of the 27th Australasian Computing Education Conference*, ACE '25, page 56–63, New York, NY, USA, 2025. Association for Computing Machinery.

[24] N. Liu, Z. Wang, R. Baraniuk, and A. Lan. Open-ended knowledge tracing for computer science education. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 3849–3862, Abu Dhabi, United Arab Emirates, Dec. 2022. Association for Computational Linguistics.

[25] S. MacNeil, M. Rogalska, J. Leinonen, P. Denny, A. Hellas, and X. Crosland. Synthetic students: A comparative study of bug distribution between large language models and computing students. In *Proceedings of the 2024 on ACM Virtual Global Computing Education Conference V. 1*, SIGCSE Virtual 2024, page 137–143, New York, NY, USA,

2024. Association for Computing Machinery.

[26] N. Matsuda, W. W. Cohen, J. Sewall, G. Lacerda, and K. R. Koedinger. Evaluating a simulated student using real students data for training and testing. In *International Conference on User Modeling*, pages 107–116. Springer, 2007.

[27] M. Miroyan, R. Niousha, J. E. Gonzalez, G. Ranade, and N. Norouzi. Parastudent: Generating and evaluating realistic student code by teaching llms to struggle, 2025.

[28] S. Mukherjee, A. Mitra, G. Jawahar, S. Agarwal, H. Palangi, and A. Awadallah. Orca: Progressive learning from complex explanation traces of GPT-4. *arXiv preprint arXiv:2306.02707*, 2023.

[29] M. H. Nguyen, V.-A. Pădurean, A. Gotovos, S. Tschiatschek, and A. Singla. Synthesizing high-quality programming tasks with llm-based expert and student agents. In A. I. Cristea, E. Walker, Y. Lu, O. C. Santos, and S. Isotani, editors, *Artificial Intelligence in Education*, pages 77–91, Cham, 2025. Springer Nature Switzerland.

[30] OpenAI. GPT-5 System Card. Technical report, OpenAI, 2025.

[31] 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*, 35:27730–27744, 2022.

[32] T. Phung, V.-A. Pădurean, A. Singh, C. Brooks, J. Cambronero, S. Gulwani, A. Singla, et al. Automating human tutor-style programming feedback: Leveraging gpt-4 tutor model for hint generation and gpt-3.5 student model for hint validation. In *Proceedings of the 14th Learning Analytics and Knowledge Conference*, LAK '24, page 12–23, New York, NY, USA, 2024. Association for Computing Machinery.

[33] R. Rafailov, A. Sharma, E. Mitchell, S. Ermon, C. D. Manning, and C. Finn. Direct preference optimization: your language model is secretly a reward model. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

[34] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, et al. Codebleu: a method for automatic evaluation of code synthesis, 2020.

[35] A. Ross, M. Srivastava, J. Blanchard, and J. Andreas. Modeling student learning with 3.8 million program traces, 2025.

[36] A. Scarlatos, R. S. Baker, and A. Lan. Exploring knowledge tracing in tutor-student dialogues using llms. In *Proceedings of the 15th Learning Analytics and Knowledge Conference, LAK 2025, Dublin, Ireland, March 3-7, 2025*. ACM, 2025.

[37] A. Scarlatos, N. Liu, J. Lee, R. Baraniuk, and A. Lan. Training llm-based tutors to improve student learning outcomes in dialogues. In A. I. Cristea, E. Walker, Y. Lu, O. C. Santos, and S. Isotani, editors, *Artificial Intelligence in Education*, pages 251–266, Cham, 2025. Springer Nature Switzerland.

[38] A. Scarlatos, D. Smith, S. Woodhead, and A. Lan. *Improving the Validity of Automatically Generated Feedback via Reinforcement Learning*, page 280–294. Springer Nature Switzerland, 2024.

[39] J. Schulman and T. M. Lab. Lora without regret. *Thinking Machines Lab: Connectionism*, 2025. https://thinkingmachines.ai/blog/lora/.

[40] T. Sirkiä and J. Sorva. Exploring programming misconceptions: an analysis of student mistakes in visual program simulation exercises. In *Proceedings of the 12th Koli calling international conference on computing education research*, pages 19–28, 2012.

[41] Q. Team. Qwen3 technical report. Technical report, 2025.

[42] Unsloth AI. Lora hyperparameters guide, 2024. Accessed: 2025-12-23.

[43] K. VanLehn, S. Ohlsson, and R. Nason. Applications of simulated students: An exploration. *Journal of Artificial Intelligence in Education*, 5(2):135–175, 1994.

[44] L. Wang, A. Sy, L. Liu, and C. Piech. Deep knowledge tracing on programming exercises. In *Proceedings of the fourth (2017) ACM conference on learning@ scale*, pages 201–204, 2017.

[45] C. Watson and F. W. Li. Failure rates in introductory programming revisited. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, ITiCSE '14, page 39–44, New York, NY, USA, 2014. Association for Computing Machinery.

[46] C. Watson, F. W. Li, and J. L. Godwin. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *2013 IEEE 13th international conference on advanced learning technologies*, pages 319–323. IEEE, 2013.

[47] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, et al. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837, 2022.

[48] J. Woodrow, C. Piech, and S. Koyejo. Improving generative ai student feedback: Direct preference optimization with teachers in the loop. In *Proceedings of the 18th International Conference on Educational Data Mining*, pages 442–449, Palermo, Italy, July 2025. International Educational Data Mining Society.

[49] C. Xu, Q. Sun, K. Zheng, X. Geng, P. Zhao, J. Feng, et al. WizardLM: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations*, 2024.

[50] Q. Yu, Z. Zhang, R. Zhu, Y. Yuan, X. Zuo, Y. Yue, et al. Dapo: An open-source llm reinforcement learning system at scale, 2025.

[51] T. Zheng, G. Zhang, T. Shen, X. Liu, B. Y. Lin, J. Fu, et al. Opencodeinterpreter: Integrating code generation with execution and refinement. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 12834–12859. Association for Computational Linguistics, 2024.