Generating Planning Feedback for Open-Ended Programming Exercises with LLMs

Mehmet Arif Demirtas[1], Claire Zheng[1], Max Fowler[1], and Kathryn Cunningham[1]

University of Illinois Urbana-Champaign, Champaign IL 61820, USA
{mad16,clairez5,mfowler5,katcun}@@illinois.edu

Abstract. To complete an open-ended programming exercise, students need to both plan a high-level solution and implement it using the appropriate syntax. However, these problems are often autograded on the correctness of the final submission through test cases, and students cannot get feedback on their planning process. Large language models (LLM) may be able to generate this feedback by detecting the overall code structure even for submissions with syntax errors. To this end, we propose an approach that detects which high-level goals and patterns (i.e. *programming plans*) exist in a student program with LLMs. We show that both GPT-4o and GPT-4o-mini can detect these plans with remarkable accuracy, outperforming baselines inspired by conventional approaches to code analysis. We further show that the smaller, cost-effective model GPT-4o-mini achieves results on par with state-of-the-art GPT-4o after fine-tuning, creating promising implications for smaller models for real-time grading. These smaller models can be incorporated into autograders for open-ended code-writing exercises to provide feedback for students' implicit planning skills, even when their program is syntactically incorrect. Furthermore, LLMs may be useful in providing feedback for problems in other domains where students start with a set of high-level solution steps and iteratively compute the output, such as math and physics problems.

Keywords: large language models · autograders · computer science education · programming plans

1 Introduction

Learning programming requires applying several skills at the same time, which may be overwhelming for novice students [44]. To solve a problem, a student will likely *interpret* a problem statement, *decompose* it into smaller chunks, *plan* how these chunks will connect, and *implement* it using the correct syntax. In psychology of programming literature, the chunks in this planning process have been termed 'programming plans': common code snippets that have a clear goal [48]. Experts in programming recall and implement these plans as they decompose a problem for efficient problem-solving [49]. Implicit or explicit instruction about plans is a common approach to support students' problem-solving processes [21, 11].

Programming courses commonly include open-ended code writing exercises to practice this end-to-end process of planning and implementation. Moreover, these exercises can be autograded with test cases, making them easily scalable to hundreds of students. However, test-case based autograders do not necessarily identify the source of errors, whether it is starting with the wrong plan for the problem or making a small syntax error in the final submission. With autograders that rely on test cases, students only see the *outcome* of the executed code, and they do not get feedback for the skills required in the *process* for solving programming problems. If we can develop autograders to detect how a student has decomposed a problem, *independent of how well they implemented their solution*, we can provide feedback on their planning skills.

Prior literature on large language models (LLM) for programming suggest that LLMs can interpret code and extract high-level semantic information [3, 47, 13, 17, 56, 12], making them appropriate for inferring intermediate planning skills from a final program. However, LLMs may generate hallucinations, where the output is incorrect or inaccurate in subtle ways [23]. To prevent hallucinations in a specialized domain, some researchers have suggested constraining the LLM to generate *structured output* within a framework that is informed by that domain [33]. To this end, programming plan literature from computing education research can constitute a framework for understanding common planning activities in open-ended programming exercises.

In this work, we utilize large language models (LLMs) to provide feedback on planning skills based on a student's final code submission, regardless of test case correctness. Specifically, we formulate a classification task where the student submission to an open-ended code-writing exercise is classified into a set of predefined programming plans to answer the following research question:

RQ: To what extent can a student's *intended* programming plan be detected from their (possibly incorrect) code submission?

Through an evaluation on student submissions from a CS1 course, we show that LLMs (GPT-4o [38] and GPT-4o-mini [1]) are remarkably better at plan classification compared to traditional methods for code analysis, even with no fine-tuning. More importantly, we find that fine-tuning GPT-4o-mini with a small dataset can improve performance significantly, with fine-tuned GPT-4o-mini achieving a higher micro-F1 score at (.781) than GPT-4o (.778). These findings suggest that LLMs enable feedback on implicit skills in programming, and small and cost-effective models can be used to provide feedback on programming plans in open-ended code-writing exercises.

2   Related Works

2.1   Programming Plans

Although programming requires thinking at varying levels of granularity, a core cognitive unit in programming problem-solving processes is identified as the programming plan [44]. A programming plan is a common code pattern for

achieving a goal, such as counting items in a list. Structures similar to plans have been reviewed under many names including "programming patterns" [22], "templates" [9], "algorithmic patterns" [36], and "plan-schemata" [26]. The ability to notice these underlying patterns appears to be a key part of programming expertise [49]. It has also been shown that experts' recall of plan structures is associated with being more efficient at problem-solving compared to novices [41].

Programming plans are taught in many introductory CS courses, either explicitly or implicitly [21, 55, 37]. Evidence from student submissions in CS1 courses suggests that novice learners improve at applying these plans with more practice [11]. There have been several instructional interventions leveraging plan-like structures to support students in planning stages, without necessarily requiring code-writing [52, 10, 40, 43]. For instance, Jigsaw [40] presented students with a set of programming plans to help them compose their solution to practice planning skills before writing code. Rivera et al. [43] designed a planning workflow that asks students to describe their solution in natural language. They evaluated the workflows by using LLMs to generate code based on these descriptions but reported difficulties in providing high-level feedback with this approach due to the quality and structure of the LLM outputs. Moreover, when planning activities are distinct from code-writing exercises, some students may find them less authentic or frustrating [42]. Thus, detecting a student's planning logic from the output of a code-writing exercise can help students practice programming plan knowledge without losing the benefits of open-ended code-writing exercises.

2.2   Detecting Structures in Code

While the problem of inferring plan usage from code submissions has not been directly addressed, problems related to detecting structures and patterns in code have been explored in different contexts. For instance, software engineering and artificial intelligence communities have explored mining *code idioms*: a piece of code that has a semantic purpose and appears across projects [4]. These studies focused on identifying emergent patterns in large codebases rather than using a pedagogically verified set of patterns [4, 46, 25, 45]. Thus, their utility in educational contexts is limited. An alternative approach was using canonical graph representations to recognize a set of patterns in programs [53]. While this approach identifies coding patterns that are similar to programming plans, the system was limited to functional languages and could not process some data abstractions. We hypothesize that LLMs can avoid these limitations as they are trained on large corpora of code obtained from many languages.

A similar task in computing education research is detecting subgoals for problems. Similar to programming plans, subgoals break down larger problems into chunks with clear objectives. Providing formative feedback on subgoals can improve student motivation and support problem-solving skills [34]. However, expert-authored feedback for student progression is unfeasible due to the large number of possible solutions [50]. To address this, Marwan et al. proposed a data-driven approach with expert constraints to identify subgoals a student intends to implement as they are writing code in real-time [34]. However, their approach

identifies subgoals on a problem basis and requires a set of submissions on the given problem for the initial training. In contrast, we identify problem-agnostic programming plans that can be extended to new problems with no additional data, supporting the generalizability of the autograder to new problems.

2.3   Autograding Programming Problems with LLMs

LLMs have created promising opportunities for autograding open-ended assessment items, as they achieve remarkable results even with no fine-tuning on additional data [54, 8]. These models have proven to be particularly useful in programming domains, potentially due to being trained on a large number of open-source projects available online [15, 16, 51]. Studies have shown that LLMs can explain code more accurately than students [30] and describe code on different levels of abstraction for learners [24]. Furthermore, autograders have incorporated LLMs for various purposes: generating test cases for programming exercises [3], evaluating short answer questions [47, 13], applying rubrics on student assignments [17, 56], and summarizing student code [12]. Thus, LLMs might be appropriate for inferring high-level structure and underlying patterns from student submissions to detect programming plans.

3   Methods

We formulate the plan detection task as a classification problem where student submissions are classified as containing zero or more programming plans. We evaluate the models against human labels on 1616 student submissions. In this section, we describe our dataset, our baseline approaches, our approaches using LLMs, and an ablation study for testing the robustness of our methods.

3.1   Dataset

Our dataset is collected in an introductory Python course at a large public US university with IRB approval. The course data was collected over 7 semesters between 2019 and 2022, with the course primarily serving first and second-year undergraduate students from Business and Liberal Arts and Sciences majors. 42.6% of students in the class identified as female. 41.8% of students were White, 21.3% were Asian, 16.6% were International, 11.9% were Hispanic, 4.4% were Black/African American, 3.2% were Multi-race, and 0.9% did not report.

The dataset contains 116 short coding homework problems, one instructor-written solution per problem featuring at least one programming plan (adapted from [21]), and up to 30 student submissions per problem.

For each problem, we classified student submissions into four categories: completely correct (i.e. passing all test cases), partially correct (i.e. passing at least one test case), semantically incorrect (i.e. passing no test cases), and syntactically incorrect (i.e. failing at compilation). For the first two categories, we started by collecting 10 submissions per problem. To capture diverse implementations in

Table 1: Programming plans and their goals, adapted from [21]

Plan	Goal
`processAllItems`	Iterate over the items in a collection
`filterACollection`	Select items from a collection that satisfy a condition
`findBestInCollection`	Find the value that has the greatest value by an arbitrary measure
`sum`	Compute the total of items from a collection
`evennessCheck`	Check if a number is even or odd using the modulus operator
`counting`	Compute the number of items from a collection
`booleanOperatorChaining`	Combine multiple logic expressions to make a decision
`multiWayBranching`	Split into three different branches based on a logic expression
`linearSearching`	Find the first matching item that satisfies a condition

this process, we computed the abstract syntax trees (AST) for each submission to filter out structurally identical submissions. For the latter two categories, we collected 3 submissions per problem and did not apply any filtering, resulting in 1616 submissions total.

Each student submission was annotated by the plan(s) included in the submission in an iterative process, using the list adapted from [21] (Table 1). A submission might also be labeled as including multiple plans, or none of the known plans (denoted as *UNKNOWN*). A codebook was iteratively developed during the annotation process by the first two authors. Initially, the first author and second author achieved a percent agreement of 90% on completely correct submissions and 66% on partially correct submissions after independently annotating 50 examples of each category. After reconciling disagreements through discussion and refining the codebook, both authors annotated 50 more examples in each of the four submission categories independently, achieving 93.9% agreement.

3.2   Baseline Approaches

As prior work did not provide a directly applicable method for detecting a predefined set of plans across many problems (Sec 2.2), we propose two baselines to compare the performance of large language models against: *AST-Rules* and *CodeBERT-kNN*.

Baseline 1: AST-Rules. We designed a rule-based classifier using abstract syntax trees (**ASTs**) of student submissions, inspired by [53]. An AST is a tree representation of the student code where each syntax element corresponds to a node in the tree. For example, an idea such as 'increment variable inside for loop' can be represented as a subtree in the AST. Thus, by traversing the AST for a student submission with the right set of rules, we can detect whether or not a programming plan is implemented in the code. Thus, we formulated a set of syntax structures (*rules*) that are associated with each plan by a manual review of instructor solutions. Then, we implemented a rule-based classifier that checks

these structures in the AST and detects the plans whose rules are satisfied. While this approach is less computationally expensive compared to LLMs, making it more feasible for autograding systems at scale, it also requires more instructor effort to identify rules. Moreover, it is more sensitive to small errors. For instance, incorrect indentation can lead to a drastically different AST.

Baseline 2: CodeBERT-kNN. We designed a k-Nearest-Neighbors (kNN) classifier using the code embeddings generated by CodeBERT model [14]. Code-BERT is a transformer architecture that creates numerical embeddings for representing code snippets. It has been shown to excel at tasks that require a high-level understanding of code, such as code search from natural language descriptions, and was the leading choice for code understanding tasks prior to the introduction of LLMs. We used CodeBERT to generate embeddings for both instructor solutions and student submissions. Then, we used a kNN classifier (k=3), classifying each student submission by comparing the labels of the instructor solutions with the most similar embeddings.

This approach is similar to that of LLMs as it relies on pretrained code embeddings, but it employs a more interpretable classification step due to comparing embeddings of code snippets directly. However, these embeddings may be sensitive to variable names and other surface-level details from the program, rather than summarizing the high-level structure of the plan the student used.

3.3   LLM Approaches

We used 2 models: GPT-4o [38] and GPT-4o-mini [1]. GPT-4o is a state-of-the-art model for code generation in Python [7, 32], whereas GPT-4o-mini [1] is a smaller and more cost-effective variant. However, GPT-4o-mini may be more feasible to deploy at a large scale due to its size. We evaluated these models in two settings: *few-shot prompting* and *fine-tuning*.

Few-shot Prompting. In few-shot prompting, examples of the target task are provided to the pretrained LLM as part of the prompt [5]. Our prompt defined the nine relevant plans with three examples each and introduced the *UNKNOWN* category for solutions where no known plan could apply. These examples are from the instructor solution set to prevent data leakage from student submissions.[1]

Fine-tuning. Fine-tuning is a technique for improving the performance of a pretrained LLM by further training it on examples from a particular task. As the pretrained models have already learned efficient embeddings, prior work has shown that a small number of examples can be sufficient to fine-tune these models and alter their behavior significantly, e.g. for automated scoring of constructed response problems [28]. We fine-tuned both models on the dataset of 116 instructor solutions, for three epochs over the full dataset with a batch size of two at the same learning rate as the pretraining. For fine-tuned models, we used an alternative prompt for classification that does not include the few-shot examples.

---

[1] The full prompts are available at ANON_FOR_REVIEW...

3.4   Ablation Study

In addition to our full classification results, we conducted an *ablation study* by replacing variable names and function signatures in programs with non-descriptive identifiers (e.g. var1) to evaluate the robustness of our models. We designed this ablation study to assess whether the models could infer structure from the code, or they were only using contextual cues (such as detecting the 'counting' plan when there is a variable called 'counter'). As novice students may not be using best practices when writing programs, we believe that this might result in a more realistic setting for our models.

For this study, we used the submissions passing at least one test and used abstract syntax trees to remove all user-generated variable names and function signatures, replacing them with random identifiers. Due to the nature of the AST-Rules baseline, this ablation did not affect the results for that approach. We ran all of the remaining approaches with the modified data.

4   Results

4.1   Exploratory Data Analysis

The dataset had 10 classes including *UNKNOWN* and the class distribution was imbalanced with most common being *filterACollection* (24.5%), and least common being *linearSearching* (3.2%). To address the imbalance, we computed per-class F1 scores and weighted averages in addition to overall accuracy scores.

Table 2 contains an analysis of the submissions by their test case success and by whether they use programming plans. Success on test cases is categorized into four groups as explained in Sec 3.1. Programming plan use is reviewed under three categories: submissions that use the same set of plans as the instructor (*Instructor Set*), submissions that use any other plan taught in the class (*Class Set*), and submissions with no known plans (*UNKNOWN*). We observed that a higher percentage of successful submissions use the same plans as instructor solutions ($\sim 77\%$ vs $\sim 62\%$), supporting our assumption that some students fail at these exercises due to a mistake in the plan selection stage. Similarly, we see that less than 5% of successful submissions omit plans, and the percentage of submissions with no known plans increases as the success of submissions decreases,

Table 2: Performance on test cases by usage of programming plans

| Success on Test Cases | % Submissions Using Plans From | | | #**Total** |
	Instructor Set	Class Set	UNKNOWN	
Passing All Tests	77.17	18.21	4.62	**692**
Passing Some Tests	61.40	27.63	10.96	**456**
Passing No Tests	62.39	20.94	16.67	**234**
Compilation Error	63.68	17.52	18.80	**234**
Overall	68.63	21.16	10.21	**1616**

shown by almost 19% of non-compiling submissions missing any plans from the class. While these results are not conclusive, they reinforce the idea that students who can select correct plans create correct submissions at higher rates, pointing to the potential benefits of plan-based feedback.

4.2   Main Study: Success on Plan Classification

Table 3 shows the overall classification accuracy of each approach, averaged over all submissions. We use three metrics for quantifying classification accuracy. Exact match ratio is a strict accuracy metric that measures the ratio of *submissions that are correctly classified*, penalizing partial cases where the model produced an incorrect classification for one of the several plans involved in a single submission. Micro-F1 score provides a more lenient accuracy metric that measures the ratio of *plans that are correctly classified*, providing credit to cases with partial success where the model classified at least one plan correctly in a submission with multiple plans. Finally, the weighted F1 score provides a more nuanced view by calculating the F1 scores for each plan individually and averaging per-plan F1 scores weighted by the number of submissions including each plan to adjust for data imbalance.

LLM approaches outperform baseline approaches significantly in all metrics. We observed that both baseline approaches perform similarly, with KNN-Clustering performing slightly worse than the AST-Rules approach. LLM-based approaches provide remarkably better results even with no fine-tuning applied, with GPT-4o and GPT-4o-mini improving the baseline by .20 and .10 points in micro-F1 scores. A series of Wilcoxon signed-rank tests with Bonferroni corrections indicated that the F1 scores for all GPT approaches were significantly different than both baselines ($p < .001$), with no significant difference between the baselines AST and KNN ($S = 174138, p = .27$). For a system where fine-tuning is not possible, GPT-4o with few shot prompting could be a viable model for providing feedback.

Promisingly, fine-tuning GPT-4o-mini improves its performance by another .10 points, with the best approach by micro-F1 score overall being GPT-4o-mini

Table 3: Comparison of all approaches by three evaluation metrics

Approach	Exact Match Ratio	Micro-F1 Score	Weighted-F1 Score
Baseline			
AST-Rules	0.5259	0.5880	0.6035
CodeBERT-kNN	0.4965	0.5496	0.5315
With Prompting			
GPT-4o	0.7157	0.7779	0.7647
GPT-4o-mini	0.6070	0.6721	0.7016
With Finetuning			
GPT-4o	0.7016	0.7713	0.7395
GPT-4o-mini	0.7157	0.7816	0.7442

Table 4: Micro-F1 scores compared for four types of student submissions

| Approach | Micro-F1 Score for Submissions | | | |
	Passing All Tests	Passing Some Tests	Passing No Tests	With Syntax Error
Baseline				
AST-Rules	0.7419	0.5874	0.5325	0.1778
CodeBERT-kNN	0.6500	0.5138	0.4218	0.4561
With Prompting				
GPT-4o	0.8373	0.7597	0.6891	0.7258
GPT-4o-mini	0.8255	0.6031	0.4775	0.5310
With Finetuning				
GPT-4o	0.8320	0.7391	0.7371	0.6892
GPT-4o-mini	0.8410	0.7805	0.6970	0.6940

(.782). Even with a relatively small dataset and a short post-training process, this small and cost-effective model can outperform the larger, state-of-the-art model. We note that GPT-4o's performance is slightly hindered by fine-tuning, potentially implying that the larger models do not benefit from being fine-tuned on small datasets. There were no significant differences between prompted GPT-4o and fine-tuned GPT-4o ($S = 10133.5, p = .61$), or fine-tuned 4o-mini ($S = 9564, p = .33$), as well as no differences between fine-tuned GPT-4o and 4o-mini ($S = 6432, p = .07$).

We show that LLM approaches are especially valuable for providing feedback for code with errors in Table 4, where micro-F1 scores for each approach on four types of submissions are provided. These results show that all approaches achieve worse prediction accuracy on submissions with errors. However, we also see that the performance gap between the baselines and the LLM approaches widens in submissions with errors. While LLM's performance decreases by approximately 10% in failing submissions, we see that the baseline models are almost unusable, experiencing drops in predictive performance that range from 30% to 70%.

We note that the model performance can vary by plan as shown in Figure 1, with the F1 scores calculated for each plan. The heatmap highlights two important insights that are not obvious from the aggregate metrics. First, submissions with no known plans (represented with label *UNKNOWN*) seem to be classified more poorly than other plans. This indicates that all our approaches are biased towards predicting *any* plan rather than predicting the *UNKNOWN* token. Second, fine-tuned models, especially fine-tuned GPT-4o-mini, perform especially worse at predicting *UNKNOWN*s. Note that the fine-tuning is done on the instructor solutions, which do not include any datapoints with *UNKNOWN* label. Thus, the fine-tuned models seem to be biased towards repeating labels from the data distribution rather than following the rules laid out in the prompt. Otherwise, there are no notable exceptions to the trends observed in the previous tables.
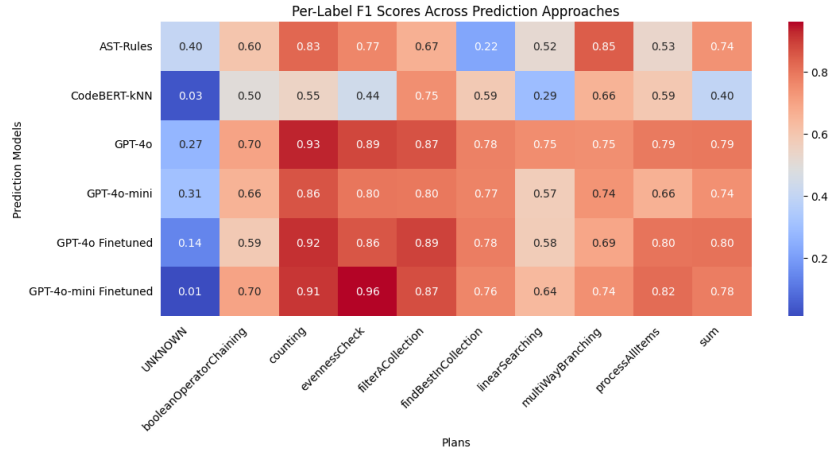
Fig. 1: F1 scores for each of the ten programming plans. Scores are generally reasonably distributed, save for classifying solutions that are labeled as UNKNOWN.

4.3   Ablation Study: Impact of Obfuscation

Table 5 shows the results of our ablation study, testing all models on completely/partially correct submissions with obfuscated variable names. For each approach, micro-F1 scores and the change from the original results are shown. Note that the results for the AST-Rules baselines are unchanged, as ASTs discard variable names during canonicalization.

LLM approaches seem to be robust to these changes on the surface-level features. However, we see that obfuscation hurts the performance of the CodeBERT-KNN baseline by around 20%, A Wilcoxon signed-rank test indicated that that the decrease for KNN was statistically significant ($S = 18981.5, p < .001$), whereas no significant differences were observed for GPT models (for prompt-

Table 5: Micro-F1 scores for obfuscated data and differences from original scores

Approach	Micro-F1 Score for Submissions	
	Passing All Tests (Δ)	Passing Some Tests (Δ)
Baseline		
AST-Rules	0.7419 (0.0000)	0.5874 (0.0000)
CodeBERT-kNN	0.5304 (-0.1197)	0.3677 (-0.1460)
With Prompting		
GPT-4o	0.8419 (0.0046)	0.7543 (-0.0054)
GPT-4o-mini	0.8363 (0.0108)	0.6504 (0.0472)
With Finetuning		
GPT-4o	0.8379 (0.0059)	0.7576 (0.0184)
GPT-4o-mini	0.8479 (0.0069)	0.7813 (0.0008)

ing: 4o $S = 2667.0, p = .97$, mini $z = 5106.5, p = .02$; for fine-tuning: 4o $S = 2,100, p = .07$, mini $S = 1466.5, p = .35$), suggesting that CodeBERT embeddings are sensitive to the surface-level features that the ablation study removes, as we initially hypothesized.

5   Discussion

Large language models are far more accurate in providing feedback on programming plans compared to conventional baselines using graph representations or code similarity measures. We achieve remarkable micro-F1 scores by prompting larger models such as GPT-4o with a few example submissions, in line with earlier studies on few-shot prompting [54]. In grading tasks where prior submission data is limited, LLM-based autograders provide a reasonable starting point.

Promisingly for classroom deployments, the smaller GPT-4o-mini model achieved and surpassed the performance of the state-of-the-art model after being fine-tuned on the small subset of the data. GPT-4o-mini is less computationally expensive, more cost-effective, and could be deployed in real-time grading scenarios at 1/16th of the cost of GPT-4o for the same performance. While we focused on GPT models, these findings also motivate fine-tuning open-source language models. There are some ethical and privacy concerns associated with using third-party APIs as part of a grading pipeline. Open-source models can be hosted on institution servers or even student devices, allowing the instructors or students to retain control of their data by processing them in secure environments.

In addition to overall higher performance, one main advantage of LLMs over baseline methods is to provide planning feedback on code submissions that are incorrect or incomplete (Table 4). Prior studies on LLM autograders have reported that LLMs are prone to generating correct code from incorrect student artifacts, such as traces of planning activities or handwritten pseudocode [43, 20]. In our work, we use this overcorrection tendency to provide high-level feedback on the structure the student *intended* to implement by ignoring implementation errors. Thus, LLMs not only improve the accuracy of plan feedback on correct submissions compared to baselines, but also make it possible to generate feedback for submissions with syntactic or semantic errors.

Teaching implications. There are multiple potential benefits to providing feedback on programming plans. First, our exploratory data analysis shows that when students' selected plans differ from the instructor's plans, submissions are more likely to be incorrect. Targeting the development of planning skills directly may improve students' programming exercise performance. Furthermore, feedback that aims to help students identify plans could potentially lead to more reflection on the problem-solving process. The literature suggests that reflective feedback [2] and feedback that nudges as opposed to providing direct next steps [57] in autograding systems may improve students' performance and interaction with feedback. Additionally, immediate feedback may improve student performance and increase students' willingness to submit assignments [35]. In the programming context, recent work augmenting SQL feedback with hints

generated by comparing model solutions and student queries saw students require fewer submissions to construct correct solutions [27], suggesting faster learning.

However, deployment of automatic feedback does not come without some risk, particularly when an autograder is *incorrect*. One study investigating an NLP autograder for students' short descriptions of code found that false positives (that is, saying a student is correct when they are not) reduced student learning, potentially due to reducing reflection [31]. However, false negatives were not harmful, as students were more able to reflect upon the feedback. Given that our plan feedback approaches can get relatively low F1 scores on some classes (e.g. *UNKNOWN* token), plan feedback can be presented in ways to encourage reflection rather than to generate final grades. For example, feedback could present students with worked examples that incorporate plans the system *thinks* the student was trying to use. At worst, a mismatch between a worked example and a plan a student was using may be ignored by the student.

Limitations and future work. Our work provides a first step for feedback on planning in code-writing problems by identifying programming plans in short programs. Future work can explore ways to generalize this approach to larger programming projects, where students may need to modify and combine multiple plans. Furthermore, evaluating this detection technique in a real-time environment with students can yield a greater understanding of how getting feedback on the problem-solving process can shape the student experience.

Due to the rapid nature of LLM research, we left several models out of our scope, including OpenAI's reasoning models (o3 [39]) as well as DeepSeek's R1[18]. While these models may outperform GPT-4o, we found that Chain-of-Thought generation significantly increases the inference time in preliminary experiments, with these reasoning models taking over 30 seconds per submission (compared to <1 second in our models). Thus, these models may not be appropriate for real-time grading at large scale due to computational limitations.

One exciting finding from our ablation study was the implication that LLMs classify submissions based on their structure and not on surface-level context cues like keywords. Therefore, future work could explore the use of LLMs to process code for analyzing structure and subgoal-level information. Moreover, the strength of LLMs in pattern recognition can motivate their application in non-programming domains where students practice selecting and applying patterns, including proof techniques in math [29], system analysis problems in physics [19], or schema acquisition for language learning [6].

6   Conclusion

In this work, we propose a framework for generating high-level planning feedback for open-ended programming exercises. By analyzing data from a CS1 course, we show that large language models can provide this feedback at higher accuracy compared to traditional code analysis methods. Moreover, even small models can be fine-tuned to provide accurate feedback at a lower cost. This approach can support students as they develop intermediate planning skills in programming

problems, as well as students in other domains where recognizing and applying common patterns plays an important role.

References

1. Gpt-4o mini: Advancing cost-efficient intelligence, https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence, accessed: 2025-02-15
2. Abu Deeb, F., Hickey, T.: Impact of reflection in auto-graders: an empirical study of novice coders. Computer science education **34**(3), 473–494 (2024)
3. Alkafaween, U., Albluwi, I., Denny, P.: Automating Autograding: Large Language Models as Test Suite Generators for Introductory Programming (Nov 2024). https://doi.org/10.48550/arXiv.2411.09261
4. Allamanis, M., Sutton, C.: Mining idioms from source code. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. p. 472–483. FSE 2014, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2635868.2635901, https://doi.org/10.1145/2635868.2635901
5. Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D.M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., Amodei, D.: Language models are few-shot learners (2020), https://arxiv.org/abs/2005.14165
6. Carrell, P.L.: Schema theory and esl reading: Classroom implications and applications. The modern language journal **68**(4), 332–343 (1984)
7. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.d.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F.P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W.H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A.N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., Zaremba, W.: Evaluating Large Language Models Trained on Code (Jul 2021). https://doi.org/10.48550/arXiv.2107.03374
8. Chen, S., Lan, Y., Yuan, Z.: A Multi-task Automated Assessment System for Essay Scoring. In: Olney, A.M., Chounta, I.A., Liu, Z., Santos, O.C., Bittencourt, I.I. (eds.) Artificial Intelligence in Education. pp. 276–283. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-64299-9_22
9. Clancy, M.J.: Designing Pascal Solutions: Case studies using data structures. WH Freeman & Co. (1996)
10. Cunningham, K., Ericson, B.J., Agrawal Bejarano, R., Guzdial, M.: Avoiding the turing tarpit: Learning conversational programming by starting from code's purpose. In: Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems. CHI '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3411764.3445571, https://doi.org/10.1145/3411764.3445571
11. Demirtaş, M.A., Fowler, M., Hu, N., Cunningham, K.: Validating, refining, and identifying programming plans using learning curve analysis on code writing data.

In: Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1. p. 263–279. ICER '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3632620.3671120, https://doi.org/10.1145/3632620.3671120

12. Dong, D., Liang, Y.: Grading Programming Assignments by Summarization. In: Proceedings of the ACM Turing Award Celebration Conference - China 2024. pp. 53–58. ACM-TURC '24, Association for Computing Machinery, New York, NY, USA (Jul 2024). https://doi.org/10.1145/3674399.3674426

13. Duong, T.N.B., Meng, C.Y.: Automatic Grading of Short Answers Using Large Language Models in Software Engineering Courses. In: 2024 IEEE Global Engineering Education Conference (EDUCON). pp. 1–10 (May 2024). https://doi.org/10.1109/EDUCON60312.2024.10578839

14. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al.: Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155 (2020)

15. Finnie-Ansley, J., Denny, P., Becker, B.A., Luxton-Reilly, A., Prather, J.: The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In: Proceedings of the 24th Australasian Computing Education Conference. pp. 10–19. ACE '22, Association for Computing Machinery, New York, NY, USA (Feb 2022). https://doi.org/10.1145/3511861.3511863

16. Finnie-Ansley, J., Denny, P., Luxton-Reilly, A., Santos, E.A., Prather, J., Becker, B.A.: My AI Wants to Know if This Will Be on the Exam: Testing OpenAI's Codex on CS2 Programming Exercises. In: Proceedings of the 25th Australasian Computing Education Conference. pp. 97–104. ACE '23, Association for Computing Machinery, New York, NY, USA (Jan 2023). https://doi.org/10.1145/3576123.3576134

17. Grandel, S., Schmidt, D.C., Leach, K.: Applying Large Language Models to Enhance the Assessment of Parallel Functional Programming Assignments. In: Proceedings of the 1st International Workshop on Large Language Models for Code. pp. 102–110. LLM4Code '24, Association for Computing Machinery, New York, NY, USA (Sep 2024). https://doi.org/10.1145/3643795.3648375

18. Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., et al.: Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. arXiv preprint arXiv:2501.12948 (2025)

19. Hewson, P.W., Posner, G.J.: The use of schema theory in the design of instructional materials: A physics example. Instructional Science **13**(2), 119–139 (1984)

20. Islam, M.S., Doumbouya, M.K.B., Manning, C.D., Piech, C.: Handwritten code recognition for pen-and-paper cs education. In: Proceedings of the Eleventh ACM Conference on Learning @ Scale. p. 200–210. L@S '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3657604.3662027, https://doi.org/10.1145/3657604.3662027

21. Iyer, V., Zilles, C.: Pattern Census: A Characterization of Pattern Usage in Early Programming Courses. In: Proceedings of the 52nd ACM Technical Symposium on Computer Science Education. pp. 45–51. SIGCSE '21, Association for Computing Machinery, New York, NY, USA (Mar 2021). https://doi.org/10.1145/3408877.3432442

22. Iyer, V.N.: An Analysis of Pattern Usage in CS1 Courses. Thesis, University of Illinois at Urbana-Champaign (Jul 2020)

23. Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y.J., Madotto, A., Fung, P.: Survey of hallucination in natural language generation. ACM Computing Surveys **55**(12), 1–38 (Mar 2023). https://doi.org/10.1145/3571730, http://dx.doi.org/10.1145/3571730

24. Jury, B., Lorusso, A., Leinonen, J., Denny, P., Luxton-Reilly, A.: Evaluating LLM-generated Worked Examples in an Introductory Programming Course. In: Proceedings of the 26th Australasian Computing Education Conference. pp. 77–86. ACM, Sydney NSW Australia (Jan 2024). https://doi.org/10.1145/3636243.3636252
25. Karanikiotis, T., Symeonidis, A.L.: Towards extracting reusable and maintainable code snippets. In: International Conference on Software Technologies. pp. 187–206. Springer (2022)
26. Kather, P., Duran, R., Vahrenhold, J.: Through (tracking) their eyes: Abstraction and complexity in program comprehension. ACM Transactions on Computing Education (TOCE) **22**(2), 1–33 (2021)
27. Kleiner, C., Heine, F.: Enhancing feedback generation for autograded sql statements to improve student learning. In: Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1. p. 248–254. ITiCSE 2024, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3649217.3653579, https://doi.org/10.1145/3649217.3653579
28. Latif, E., Zhai, X.: Fine-tuning chatgpt for automatic scoring. Computers and Education: Artificial Intelligence **6**, 100210 (2024)
29. Lee, K.: Students' proof schemes for mathematical proving and disproving of propositions. The Journal of Mathematical Behavior **41**, 26–44 (2016)
30. Leinonen, J., Denny, P., MacNeil, S., Sarsa, S., Bernstein, S., Kim, J., Tran, A., Hellas, A.: Comparing Code Explanations Created by Students and Large Language Models. In: Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1. pp. 124–130. ACM, Turku Finland (Jun 2023). https://doi.org/10.1145/3587102.3588785
31. Li, T.W., Hsu, S., Fowler, M., Zhang, Z., Zilles, C., Karahalios, K.: Am i wrong, or is the autograder wrong? effects of ai grading mistakes on learning. In: Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1. p. 159–176. ICER '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3568813.3600124, https://doi.org/10.1145/3568813.3600124
32. Liu, J., Xia, C.S., Wang, Y., ZHANG, L.: Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In: Oh, A., Neumann, T., Globerson, A., Saenko, K., Hardt, M., Levine, S. (eds.) Advances in Neural Information Processing Systems. vol. 36, pp. 21558–21572. Curran Associates, Inc. (2023)
33. Liu, M.X., Liu, F., Fiannaca, A.J., Koo, T., Dixon, L., Terry, M., Cai, C.J.: "we need structured output": Towards user-centered constraints on large language model output. In: Extended Abstracts of the CHI Conference on Human Factors in Computing Systems. CHI EA '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3613905.3650756, https://doi.org/10.1145/3613905.3650756
34. Marwan, S., Shi, Y., Menezes, I., Chi, M., Barnes, T., Price, T.W.: Just a Few Expert Constraints Can Help: Humanizing Data-Driven Subgoal Detection for Novice Programming. Tech. rep., International Educational Data Mining Society (2021)
35. Mitra, J.: Studying the impact of auto-graders giving immediate feedback in programming assignments. In: Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1. p. 388–394. SIGCSE 2023, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3545945.3569726, https://doi.org/10.1145/3545945.3569726

36. Muller, O.: Pattern oriented instruction and the enhancement of analogical reasoning. In: Proceedings of the First International Workshop on Computing Education Research. pp. 57–67. ICER '05, Association for Computing Machinery, New York, NY, USA (Oct 2005). https://doi.org/10.1145/1089786.1089792

37. Muller, O., Ginat, D., Haberman, B.: Pattern-oriented instruction and its influence on problem decomposition and solution construction. In: Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education. pp. 151–155. ITiCSE '07, Association for Computing Machinery, New York, NY, USA (Jun 2007). https://doi.org/10.1145/1268784.1268830

38. OpenAI, :, Hurst, A., Lerer, A., Goucher, A.P., Perelman, A., Ramesh, A., Clark, A., Ostrow, A., Welihinda, A., Hayes, A., Radford, A., Mądry, A., Baker-Whitcomb, A., Beutel, A., Borzunov, A., Carney, A., Chow, A., Kirillov, A., Nichol, A., Paino, A., Renzin, A., Passos, A.T., Kirillov, A., Christakis, A., Conneau, A., Kamali, A., Jabri, A., Moyer, A., Tam, A., Crookes, A., Tootoochian, A., Tootoonchian, A., Kumar, A., Vallone, A., Karpathy, A., Braunstein, A., Cann, A., Codispoti, A., Galu, A., Kondrich, A., Tulloch, A., Mishchenko, A., Baek, A., Jiang, A., Pelisse, A., Woodford, A., Gosalia, A., Dhar, A., Pantuliano, A., Nayak, A., Oliver, A., Zoph, B., Ghorbani, B., Leimberger, B., Rossen, B., Sokolowsky, B., Wang, B., Zweig, B., Hoover, B., Samic, B., McGrew, B., Spero, B., Giertler, B., Cheng, B., Lightcap, B., Walkin, B., Quinn, B., Guarraci, B., Hsu, B., Kellogg, B., Eastman, B., Lugaresi, C., Wainwright, C., Bassin, C., Hudson, C., Chu, C., Nelson, C., Li, C., Shern, C.J., Conger, C., Barette, C., Voss, C., Ding, C., Lu, C., Zhang, C., Beaumont, C., Hallacy, C., Koch, C., Gibson, C., Kim, C., Choi, C., McLeavey, C., Hesse, C., Fischer, C., Winter, C., Czarnecki, C., Jarvis, C., Wei, C., Koumouzelis, C., Sherburn, D., Kappler, D., Levin, D., Levy, D., Carr, D., Farhi, D., Mely, D., Robinson, D., Sasaki, D., Jin, D., Valladares, D., Tsipras, D., Li, D., Nguyen, D.P., Findlay, D., Oiwoh, E., Wong, E., Asdar, E., Proehl, E., Yang, E., Antonow, E., Kramer, E., Peterson, E., Sigler, E., Wallace, E., Brevdo, E., Mays, E., Khorasani, F., Such, F.P., Raso, F., Zhang, F., von Lohmann, F., Sulit, F., Goh, G., Oden, G., Salmon, G., Starace, G., Brockman, G., Salman, H., Bao, H., Hu, H., Wong, H., Wang, H., Schmidt, H., Whitney, H., Jun, H., Kirchner, H., de Oliveira Pinto, H.P., Ren, H., Chang, H., Chung, H.W., Kivlichan, I., O'Connell, I., O'Connell, I., Osband, I., Silber, I., Sohl, I., Okuyucu, I., Lan, I., Kostrikov, I., Sutskever, I., Kanitscheider, I., Gulrajani, I., Coxon, J., Menick, J., Pachocki, J., Aung, J., Betker, J., Crooks, J., Lennon, J., Kiros, J., Leike, J., Park, J., Kwon, J., Phang, J., Teplitz, J., Wei, J., Wolfe, J., Chen, J., Harris, J., Varavva, J., Lee, J.G., Shieh, J., Lin, J., Yu, J., Weng, J., Tang, J., Yu, J., Jang, J., Candela, J.Q., Beutler, J., Landers, J., Parish, J., Heidecke, J., Schulman, J., Lachman, J., McKay, J., Uesato, J., Ward, J., Kim, J.W., Huizinga, J., Sitkin, J., Kraaijeveld, J., Gross, J., Kaplan, J., Snyder, J., Achiam, J., Jiao, J., Lee, J., Zhuang, J., Harriman, J., Fricke, K., Hayashi, K., Singhal, K., Shi, K., Karthik, K., Wood, K., Rimbach, K., Hsu, K., Nguyen, K., Gu-Lemberg, K., Button, K., Liu, K., Howe, K., Muthukumar, K., Luther, K., Ahmad, L., Kai, L., Itow, L., Workman, L., Pathak, L., Chen, L., Jing, L., Guy, L., Fedus, L., Zhou, L., Mamitsuka, L., Weng, L., McCallum, L., Held, L., Ouyang, L., Feuvrier, L., Zhang, L., Kondraciuk, L., Kaiser, L., Hewitt, L., Metz, L., Doshi, L., Aflak, M., Simens, M., Boyd, M., Thompson, M., Dukhan, M., Chen, M., Gray, M., Hudnall, M., Zhang, M., Aljubeh, M., Litwin, M., Zeng, M., Johnson, M., Shetty, M., Gupta, M., Shah, M., Yatbaz, M., Yang, M.J., Zhong, M., Glaese, M., Chen, M., Janner, M., Lampe, M., Petrov, M., Wu, M., Wang, M., Fradin, M., Pokrass, M., Castro, M., de Castro, M.O.T., Pavlov, M., Brundage, M., Wang, M., Khan, M., Murati, M.,

Bavarian, M., Lin, M., Yesildal, M., Soto, N., Gimelshein, N., Cone, N., Staudacher, N., Summers, N., LaFontaine, N., Chowdhury, N., Ryder, N., Stathas, N., Turley, N., Tezak, N., Felix, N., Kudige, N., Keskar, N., Deutsch, N., Bundick, N., Puckett, N., Nachum, O., Okelola, O., Boiko, O., Murk, O., Jaffe, O., Watkins, O., Godement, O., Campbell-Moore, O., Chao, P., McMillan, P., Belov, P., Su, P., Bak, P., Bakkum, P., Deng, P., Dolan, P., Hoeschele, P., Welinder, P., Tillet, P., Pronin, P., Tillet, P., Dhariwal, P., Yuan, Q., Dias, R., Lim, R., Arora, R., Troll, R., Lin, R., Lopes, R.G., Puri, R., Miyara, R., Leike, R., Gaubert, R., Zamani, R., Wang, R., Donnelly, R., Honsby, R., Smith, R., Sahai, R., Ramchandani, R., Huet, R., Carmichael, R., Zellers, R., Chen, R., Chen, R., Nigmatullin, R., Cheu, R., Jain, S., Altman, S., Schoenholz, S., Toizer, S., Miserendino, S., Agarwal, S., Culver, S., Ethersmith, S., Gray, S., Grove, S., Metzger, S., Hermani, S., Jain, S., Zhao, S., Wu, S., Jomoto, S., Wu, S., Shuaiqi, Xia, Phene, S., Papay, S., Narayanan, S., Coffey, S., Lee, S., Hall, S., Balaji, S., Broda, T., Stramer, T., Xu, T., Gogineni, T., Christianson, T., Sanders, T., Patwardhan, T., Cunninghman, T., Degry, T., Dimson, T., Raoux, T., Shadwell, T., Zheng, T., Underwood, T., Markov, T., Sherbakov, T., Rubin, T., Stasi, T., Kaftan, T., Heywood, T., Peterson, T., Walters, T., Eloundou, T., Qi, V., Moeller, V., Monaco, V., Kuo, V., Fomenko, V., Chang, W., Zheng, W., Zhou, W., Manassra, W., Sheu, W., Zaremba, W., Patil, Y., Qian, Y., Kim, Y., Cheng, Y., Zhang, Y., He, Y., Zhang, Y., Jin, Y., Dai, Y., Malkov, Y.: Gpt-4o system card (2024), https://arxiv.org/abs/2410.21276

39. OpenAI: Openai o3-mini system card (2025), https://cdn.openai.com/o3-mini-system-card-feb10.pdf

40. Reichert, H., Tabarsi, B.T., Price, T., Barnes, T.: Jigsaw: A Tool for Decomposing and Planning Programming Problems. In: 2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 236–247 (Sep 2024). https://doi.org/10.1109/VL/HCC60511.2024.00034

41. Rist, R.S.: Program Structure and Design. Cognitive Science **19**(4), 507–562 (1995). https://doi.org/10.1207/s15516709cog1904_3

42. Rivera, E., Fisler, K., Krishnamurthi, S.: Observations on the Design of Program Planning Notations for Students. In: Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1. pp. 1133–1139. ACM, Portland OR USA (Mar 2024). https://doi.org/10.1145/3626252.3630901

43. Rivera, E., Steinmaurer, A., Fisler, K., Krishnamurthi, S.: Iterative Student Program Planning using Transformer-Driven Feedback. In: Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1. pp. 45–51. ITiCSE 2024, Association for Computing Machinery, New York, NY, USA (Jul 2024). https://doi.org/10.1145/3649217.3653607

44. Robins, A.V.: Novice Programmers and Introductory Programming. In: Fincher, S.A., Robins, A.V. (eds.) The Cambridge Handbook of Computing Education Research, p. 355. Cambridge Handbooks in Psychology, Cambridge University Press (2019). https://doi.org/10.1017/9781108654555.013

45. Shin, R., Allamanis, M., Brockschmidt, M., Polozov, O.: Program synthesis and semantic parsing with learned code idioms. arXiv preprint arXiv:1906.10816 (2019)

46. Sivaraman, A., Abreu, R., Scott, A., Akomolede, T., Chandra, S.: Mining idioms in the wild. In: Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice. p. 187–196. ICSE-SEIP '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3510457.3513046, https://doi.org/10.1145/3510457.3513046

47. Smith, D.H., Denny, P., Fowler, M.: Prompting for Comprehension: Exploring the Intersection of Explain in Plain English Questions and Prompt Writing. In: Proceedings of the Eleventh ACM Conference on Learning @ Scale. pp. 39–50. ACM, Atlanta GA USA (Jul 2024). https://doi.org/10.1145/3657604.3662039

48. Soloway, E.: Learning to program= learning to construct mechanisms and explanations. Communications of the ACM **29**(9), 850–858 (1986)

49. Soloway, E., Ehrlich, K.: Empirical Studies of Programming Knowledge. IEEE Transactions on Software Engineering **SE-10**(5), 595–609 (Sep 1984). https://doi.org/10.1109/TSE.1984.5010283

50. Toll, D., Wingkvist, A., Ericsson, M.: Current state and next steps on automated hints for students learning to code. In: 2020 IEEE Frontiers in Education Conference (FIE). pp. 1–5. IEEE (2020)

51. Wang, T., Díaz, D.V., Brown, C., Chen, Y.: Exploring the Role of AI Assistants in Computer Science Education: Methods, Implications, and Instructor Perspectives. In: 2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 92–102. IEEE, Washington, DC, USA (Oct 2023). https://doi.org/10.1109/VL-HCC57772.2023.00018

52. Weinman, N., Fox, A., Hearst, M.A.: Improving instruction of programming patterns with faded parsons problems. In: Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems. CHI '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3411764.3445228, https://doi.org/10.1145/3411764.3445228

53. Wills, L.M.: Automated Program Recognition (Feb 1987)

54. Wu, X., He, X., Liu, T., Liu, N., Zhai, X.: Matching Exemplar as Next Sentence Prediction (MeNSP): Zero-Shot Prompt Learning for Automatic Scoring in Science Education. In: Wang, N., Rebolledo-Mendez, G., Matsuda, N., Santos, O.C., Dimitrova, V. (eds.) Artificial Intelligence in Education, vol. 13916, pp. 401–413. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-36272-9_33

55. Xie, B., Loksa, D., Nelson, G.L., Davidson, M.J., Dong, D., Kwik, H., Tan, A.H., Hwa, L., Li, M., Ko, A.J.: A theory of instruction for introductory programming skills. Computer Science Education **29**(2-3), 205–253 (Jul 2019). https://doi.org/10.1080/08993408.2019.1565235

56. Yousef, M., Mohamed, K., Medhat, W., Mohamed, E.H., Khoriba, G., Arafa, T.: BeGrading: Large language models for enhanced feedback in programming education. Neural Computing and Applications (Oct 2024). https://doi.org/10.1007/s00521-024-10449-y

57. Zamprogno, L., Holmes, R., Baniassad, E.: Nudging student learning strategies using formative feedback in automatically graded assessments. In: Proceedings of the 2020 ACM SIGPLAN Symposium on SPLASH-E. p. 1–11. SPLASH-E 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3426431.3428654, https://doi.org/10.1145/3426431.3428654