Koh, E., et al. (Eds.) (2026). Proceedings of the 2nd International Conference on Learning Evidence and Analytics. Asia-Pacific Society for Computers in Education.

Deriving Common Novice Programming Error Patterns from Student Submissions using LLMs

Boxuan MA[a*] & Huiyong LI[b]
[a]*Faculty of Arts and Science, Kyushu University, Japan*
[b]*Research Institute for Information Technology, Kyushu University, Japan*
*boxuan@artsci.kyushu-u.ac.jp

Abstract: Identifying common difficulties in novice programming is crucial for helping instructors understand where students struggle and for designing targeted interventions. However, extracting actionable insights from large collections of student code submissions typically requires substantial manual review, which can be time-consuming. In this paper, we investigate whether Large Language Models (LLMs) can derive course-specific summaries of common novice programming error patterns directly from historical code submissions. Results showed that the LLM achieved high accuracy in identifying error patterns and generating useful hints, indicating its potential to support instructor-facing analysis of novice programming difficulties.

Keywords: Large Language Models, code submissions, programming errors

1. Introduction

Novice programmers often struggle with recurring misconceptions, including errors in syntax, function structure, variable use, return values, and output formatting (Li & Ma, 2025). For instructors, identifying such patterns at the cohort level is valuable for prioritizing lecture content, designing targeted feedback, and creating debugging activities (Altadmri & Brown, 2015). However, although modern programming courses collect large volumes of student submissions through automated assessment systems, extracting instructor-ready summaries of common misconceptions from these logs remains labor-intensive.

Recent advances in LLMs offer a potential solution. LLMs can analyze student code, describe how it diverges from task requirements, and help identify recurring error patterns and their likely underlying misconceptions across many submissions (Li et al., 2025). Rather than fixing individual programs, LLMs may support cohort-level analysis by grouping similar failed attempts, labeling common difficulties, and providing representative examples for instructional planning and debugging activities (Fwa, 2024).

In this paper, we examine whether an LLM-based approach can derive common novice programming difficulties from historical student submissions. Using data from an introductory programming context, we prompt the LLM to extract error patterns in failed attempts.

2. Method

2.1 Context and Dataset

The data used in this study were collected in a laboratory-based study conducted in a Moodle course space configured for the experiment. A total of 22 participants (14 male and 8 female) were recruited from a national university in Japan. Informed consent was obtained from all participants prior to data collection. Participants enrolled in an introductory Python programming course on Moodle, which provided the study materials and tasks. They first reviewed introductory materials covering basic Python concepts, then solved four beginner-level coding tasks using CodeRunner, an open-source Moodle plugin for automated

Prompt:

You are an expert programming instructor and error analyst.

[Task]: {task_description}

[Solution code for reference: {solution_code}

[Test case(s)]: {test_case(s)}

[Expected output(s)]: {expected_output(s)}

Identify all errors in the student code below that is intended to perform the same function, and provide concise hints to fix them.

[Student codes]: {student_code}

Figure 1. The prompt template used in our implementation.

assessment. The tasks focused on function definitions and output requirements. For each task, participants read the problem statement, entered their code, and submitted solutions for automatic checking against predefined test cases. The system logged each submission, including the submitted code and timestamps.

2.2 Workflow and Evaluation

We first organized the data by question and extracted all failed submissions. We analyzed 171 failed submissions in total. Across the 22 participants, this corresponds to an average of 7.77 failed submissions per student. For each failed attempt, we prompted an LLM to generate error signatures that capture the recurring error pattern and brief feedback for each error. Note that many submissions contained multiple errors. We therefore treated error identification as a multi-label task at the submission level. In this study, we used GPT-5.2 to analyze student submissions, extract observable error patterns, and generate corrective hints. The prompt template used in our implementation is shown in Figure 1.

We evaluated the LLM outputs for two aspects: (1) correctness of the extracted error labels and (2) usefulness of the generated fix hints. Two human raters independently reviewed all 171 failed submissions and assigned binary labels (yes/no) to the LLM outputs. For each submission, the raters checked whether the extracted error labels accurately captured the observable issues in the submitted code and whether the generated hints were appropriate and actionable for correcting the identified errors. Disagreements were resolved through adjudication after the independent rating process. Inter-rater agreement on error extraction was high (98.25%). Inter-rater agreement for hint evaluation was also high (98.24%).

3. Results

The human evaluation showed that the LLM achieved 96.5% correctness in error extraction and 95.9% correctness in generated feedback (hints). Overall, these findings suggest that the LLMs can provide useful support for identifying observable error patterns and generating hints in novice programming submissions. The remaining incorrect cases were primarily due to course-context mismatches rather than to a complete misunderstanding of the code. In several cases, the LLM generated technically plausible hints that were not aligned with the course's intended instructional scope or included unnecessary suggestions beyond the course-specific evaluation criteria. Table 1 shows the LLM-identified common error patterns for each task, along with representative student code examples. The task-level summaries reveal recurring issues across all four tasks, including both single-error and multi-error submissions, which is important because many failed attempts contained co-occurring problems rather than isolated mistakes.

4. Conclusion

Table 1. *LLM-Identified Common Error Patterns and Representative Examples by Task.*

Task	Error Pattern	Representative Example
Task-1	Missing colon after def	`def print_morning()` `  print("おはようございます")`
	Fullwidth punctuation	`def print_morning():` `  print （"おはようございます"）`

	Extra test code in submission	`def print_morning():`
		`  print("おはようございます")`
		`print_morning()`

Task-2	Wrong newline escape slash n	`def print_greetings():` `  print("おはよう/n こんにちは/n こんばんは/n おやすみ/n")`
	Wrong function name Comma print inserts separators	`def print_greeting():` `  print("おはよう","こんにちは","こんばんは","おやすみ")`

	Extra function call after definition	`def print_greetings():`
		`  print("おはよう\n こんにちは\n こんばんは\n おやすみ\n")`
		`print_greetings()`

Task-3	Missing day factor	`def convert_days_to_seconds(d):` `  return d*60*60`
	Print instead of return	`def convert_days_to_seconds(d):` `  print(d*86400)`

	Extra test code in submission	`def convert_days_to_seconds(d):`
		`  print(d*86400)`
		`result = convert_days_to_seconds(d)`

Task-4	Prints with newlines default end	`def repeat_five_times(s):`
		`  print(s)`
		`  print(s)`
		`  print(s)`
		`  print(s)`
		`  print(s)`

	Undefined identifier	`def repeat_five_times(s):` `  print(sssss)`
	Comma print inserts spaces	`def repeat_five_times(s):` `  print(s,s,s,s,s)`

This study examined whether an LLM can derive summaries of novice programming error patterns from student submission logs. The findings suggest that LLMs can reduce the manual effort required to review large volumes of submissions while helping instructors identify common learning obstacles and prepare targeted teaching interventions. Future work should validate the approach on larger datasets and in additional programming languages, and further examine how instructors use the generated summaries in real teaching practice.

Acknowledgements

This work is supported by JSPS KAKENHI Grant Numbers JP24K20903 and JP25K17078.

References

Li, H., Ma, B., & Yin, C. (2025). Examining Metacognitive Difficulties in Learning Programming: Analysis of Student Behavior and Strategy. In *Proceedings of the International Conference on Learning Evidence and Analytics*.

Fwa, H. L. (2024). Experience report: Identifying common misconceptions and errors of novice programmers with ChatGPT. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training* (pp. 233-241).

Altadmri, A., & Brown, N. C. (2015). 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM technical symposium on computer science education* (pp. 522-527).

Li, H., & Ma, B. (2025). CodeRunner Agent: Integrating AI Feedback and Self-Regulated Learning to Support Programming Education. In *Proceedings of the International Conference on Computers in Education*.