

# From Fixes to Learning: Knowledge-Augmented Hierarchical Feedback for Programming Practice

Yuchen CAO

Department of Computer Science  
City University of Hong Kong  
Hong Kong, China  
cynthcao2-c@my.cityu.edu.hk

Yifei Wang

Department of Computer Science  
City University of Hong Kong  
Hong Kong, China  
ywang4748-c@my.cityu.edu.hk

Jacky Keung

Department of Computer Science  
City University of Hong Kong  
Hong Kong, China  
Jacky.Keung@cityu.edu.hk

Jingyu Zhang\*

Department of Electronic Engineering  
and Computer Science  
Hong Kong Metropolitan University  
Hong Kong, China  
fzhang@hkmu.edu.hk

**Abstract**—Large language models (LLMs) are increasingly used as on-demand assistants for programming practice. Yet the support they provide is often fix-oriented rather than learning-oriented: models may produce corrected solutions too quickly, or generate comments that remain only loosely connected to the learner's underlying concept gap. This paper presents a lightweight framework for programming exercises that combines pedagogical diagnosis, knowledge-grounded retrieval, and hierarchically structured feedback. Given a problem and an erroneous submission, the framework first identifies a likely misconception-oriented error category, then retrieves relevant instructional knowledge from a curated programming knowledge base, and finally produces feedback at three levels of explicitness: a hint, a concept explanation, and a targeted repair suggestion. The framework is implemented using GPT-5.4 as the common underlying language model across all LLM-based stages and is designed to support revision without defaulting to full-solution disclosure. We evaluate the approach on a curated set of 30 LeetCode-style tasks and 150 erroneous solutions constructed with LLM assistance and reviewed by three computer-science graduate research students for plausibility and concept alignment. The results suggest that concept-grounded and hierarchically structured feedback is associated with stronger concept alignment and pedagogical usefulness while preserving useful support for second-attempt correction.

**Keywords**—programming education, intelligent tutoring systems, large language models, automated feedback, retrieval-augmented generation

## I. INTRODUCTION

Large language models (LLMs) are becoming a routine part of programming education. Students use them to ask for hints, debug code, interpret error messages, and obtain explanations of algorithmic ideas during self-practice [1]–[4]. This is especially visible in LeetCode-style exercises, where learners often turn to AI assistance after failing hidden tests, misapplying a familiar pattern, or getting stuck between a local implementation error and a broader algorithmic idea.

At the same time, the educational value of this support remains unsettled. In many cases, current LLM-based assistance is more fix-oriented than learning-oriented. A model may directly rewrite the solution, or it may offer generic advice such as “check your loop condition” or

“consider edge cases” without connecting the failure to the actual concept the learner is missing. Work in computing education has already shown both the promise of LLM-based help and the risks of inaccurate, overly explicit, or pedagogically weak assistance [1], [3], [4]. From a broader educational perspective, feedback is most useful when it helps learners understand what is wrong, why it is wrong, and what a productive next step looks like, rather than simply exposing the final answer [5], [6].

This tension points to a design problem in current AI support for programming practice. Much existing work on automated feedback has focused on correctness, grading, or static and dynamic analysis pipelines [7], [8]. More recent systems using LLMs have explored guardrails, next-step hints, code explanations, and error-message enhancement [3], [9], [10], [4], but relatively less attention has been given to feedback that is simultaneously grounded in explicit instructional knowledge and structured as scaffolded tutoring. In practice, learners do not only need to know that a solution is wrong; they also need help connecting a local mistake to a reusable concept such as a sliding-window invariant, a prefix-sum counting relation, or a state transition in dynamic programming.

To address this issue, we propose a knowledge-augmented hierarchical feedback framework for programming practice. Instead of asking an LLM to comment on an erroneous solution in one shot, we separate the process into three connected stages. The framework first performs a pedagogical diagnosis that maps the failure to a likely misconception-oriented category. It then retrieves concept-level information from a curated programming knowledge base organized around common problem types, key skills, standard solution patterns, and typical bug patterns. Finally, it produces feedback in three progressively more explicit forms: a hint, a concept explanation, and a targeted repair suggestion. The goal is to make the support more educationally aligned and less likely to collapse into immediate answer substitution.

This paper presents a compact framework for LLM-supported programming feedback that integrates pedagogical diagnosis, knowledge-grounded retrieval, and hierarchical response generation in a lightweight tutoring pipeline. We show how these components can be combined to support programming revision in a more concept-aware way. We also define an initial evaluation setting centered on

---

\*Corresponding author

educational feedback quality rather than correctness alone, using a curated set of 30 representative tasks and 150 erroneous submissions. Together, these elements shift the focus from whether LLMs can solve programming tasks to how they can better support people who are still learning how to solve them.

## II. BACKGROUND AND RELATED WORK

Feedback has long been recognized as a central component of effective learning, and its value depends not only on correctness but also on timing, specificity, and how well it supports the learner's next step [5]. Work on intelligent tutoring systems has similarly emphasized stepwise guidance, adaptive support, and instructional structure rather than simple answer delivery [6], [11], [12]. These ideas remain highly relevant in the era of LLM-based programming support. If a system responds to every incorrect attempt with an immediate fix, it may improve short-term completion while doing little to support conceptual understanding or transfer.

In programming education specifically, a substantial literature has examined automated feedback, grading, and assessment tools. Keuning et al. found that many automated feedback systems focus more on error identification than on concept-aware explanation [7]. Earlier work on automated assessment similarly shows that programming feedback pipelines have often been organized around test-based correctness, scoring, and assignment management rather than around the learner's evolving conceptual model [8]. This background helps explain why newer LLM-based systems are both promising and potentially problematic: they can produce highly fluent explanations, but fluency alone does not ensure pedagogical usefulness.

Recent work has explored a range of LLM applications in computing education, including the generation of programming exercises and code explanations [13], the enhancement of programming error messages [4], comparisons between student-written and model-generated explanations [10], and the handling of beginner programmers' help requests [1]. Other studies have demonstrated that modern models perform surprisingly well on programming course assessments, which intensifies concerns about over-reliance and answer substitution [2], [14]. Systems such as CodeHelp respond to this challenge by introducing guardrails so that support remains available without immediately exposing full solutions [3], [15]. Work on next-step hint generation points in a similar direction by asking not how a model can solve a task, but how it can support a learner who has not solved it yet [9].

At the same time, recent studies show that the educational role of LLMs is broader than answer generation. LLM-produced code explanations have been integrated into course materials [16], while AI code generators have been studied as support tools for novice learners [17]. Other work has proposed new programming exercises designed around prompting and code evaluation [18] and has argued that generative code tools require changes in how introductory programming is taught and assessed [19]–[21]. These studies support the view that LLMs can be valuable learning infrastructure, but they also expose a recurring design risk: students may receive fluent responses that solve or patch a program without strengthening the underlying concept. Work on formative programming feedback and syntax-error feedback makes this concern especially concrete, since high-quality feedback needs to be accurate, bounded, and

pedagogically useful rather than merely corrective [22], [23]. Broader education-oriented analyses likewise emphasize the need for teacher oversight, critical use, and pedagogical framing when LLMs are deployed in learning contexts [24].

Retrieval-augmented generation provides a complementary route for making LLM outputs more grounded. The original RAG formulation combines parametric language-model knowledge with retrieved non-parametric evidence for knowledge-intensive generation [25], and later surveys describe retrieval as a practical way to improve factuality, updateability, and source traceability in LLM systems [26], [27]. In education, retrieval has also been explored as a way to tie programming feedback to lecture content and course-specific terminology [28]. Our work is situated in this emerging space. We investigate a middle ground in which retrieved instructional knowledge and hierarchical response design are used together to make LLM feedback more concept-aware and more tutor-like.

## III. METHODOLOGY

The framework proposed in this paper is motivated by a simple observation: when learners submit incorrect solutions, what they often need is not an immediate replacement solution, but help that connects local failure to reusable conceptual understanding. For that reason, we organize the feedback process into three stages that are deliberately separated from one another. The system first interprets the likely nature of the failure, then grounds that interpretation in a small body of instructional knowledge, and only after that generates feedback intended for the learner.

Fig. 1 illustrates the overall pipeline. The framework takes as input a programming problem, an erroneous submission, and, when available, supplementary execution signals such as failed tests or runtime errors. It then performs what we call pedagogical diagnosis. This stage identifies the kind of conceptual misunderstanding that may be driving the error. In practice, diagnosis labels correspond to recurring categories such as boundary-condition mistakes, incorrect pointer movement, invalid state transitions in dynamic programming, missing prefix-sum counting logic, backtracking state restoration failures, or incorrect monotonic-stack conditions. These categories function as bridges between an incorrect program and the instructional knowledge needed to respond to it.

Once a likely diagnosis has been formed, the framework retrieves concept-level knowledge from a curated programming knowledge base derived from LeetCode Hot100-style tasks. This resource serves as a structured instructional layer between raw problem statements and the eventual feedback. Each entry in the knowledge base stores a paraphrased account of problem intent, the main topic of the task, the key idea or invariant typically used to solve it, a standard solution pattern, common bug patterns, misconception-oriented tags, and a ladder of increasingly explicit hints. A sliding-window entry, for example, may describe the condition under which a left pointer should move, what it means for a window to remain valid, and what kinds of mistakes learners make when they update an answer too early. A prefix-sum entry may record the relation between the current cumulative sum and the historical counts that need to be remembered. Retrieval is guided jointly by the diagnosed category and the task topic so that feedback is anchored in explicit instructional content rather than generated as generic commentary over the code alone.

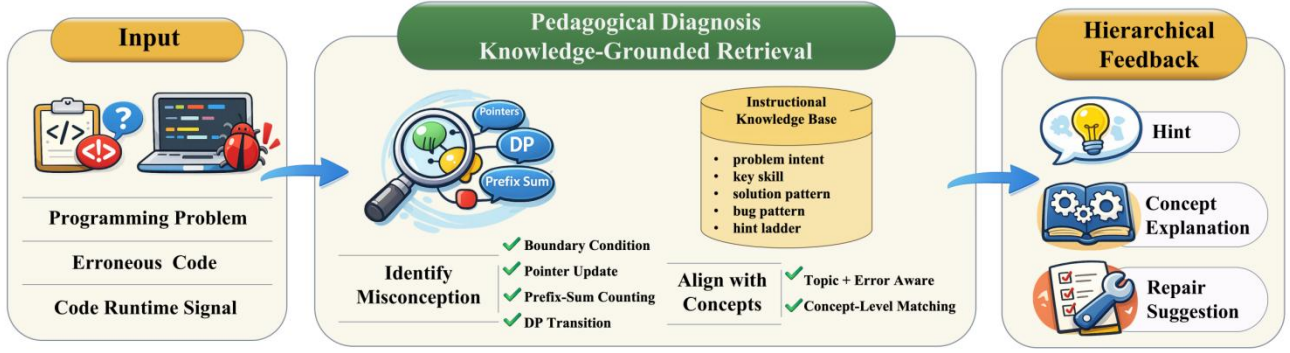


Fig. 1. Overview of the proposed knowledge-augmented hierarchical feedback framework. The system takes a programming problem, erroneous code, and runtime signals as input, performs pedagogical diagnosis and knowledge-grounded retrieval, and then generates hierarchical feedback in the form of hints, concept explanations, and targeted repair suggestions.

Using retrieval as an intermediate step also makes the feedback process easier to audit. A direct model response may be difficult to interpret because it blends code reading, diagnosis, explanation, and repair into a single generation step. By contrast, the retrieved knowledge entry gives the system an explicit pedagogical anchor that can be inspected separately from the final wording of the response. This design follows the broader motivation of retrieval-augmented generation: external knowledge can reduce reliance on unsupported parametric recall and can make generated content more specific to a task context [25]–[27]. In our setting, the retrieved unit is not a long document passage but a compact concept card, which is better aligned with the needs of programming practice feedback.

The final stage turns the diagnosed error and the retrieved knowledge into feedback. Instead of producing a single undifferentiated response, the framework organizes the output hierarchically. The first level is a hint that points the learner toward a productive direction without resolving the problem for them. The second level is a concept explanation that names and clarifies the relevant idea, such as an invariant, counting relation, or transition rule. The third level is a targeted repair suggestion that indicates which part of the logic likely requires revision. This hierarchy makes the feedback more tutor-like. It supports a progression from low explicitness to higher explicitness while avoiding full-solution disclosure in the default setting. In this way, the framework preserves the learner’s role in the revision process rather than replacing it outright.

All LLM-based stages in this study use GPT-5.4 as a common backbone model. This choice is kept fixed across the compared methods so that observed differences come primarily from prompting structure and retrieval design rather than from variation in the underlying model.

#### IV. KNOWLEDGE BASE CONSTRUCTION

The programming knowledge base used in our framework is intentionally lightweight. It is built around LeetCode Hot100-style practice problems because those tasks cover a broad range of recurring algorithmic patterns that are familiar in self-paced programming study. Rather than storing only problem statements and accepted solutions, we organize each entry around what a tutoring-oriented system actually needs in order to generate educationally useful responses. This includes a paraphrase of the task’s intent, the main topic and key skill it exercises, the core invariant or conceptual structure of a standard solution, recurrent bug patterns, misconception tags, and several levels of hints.

The current corpus is constructed in two stages. First, erroneous solutions are produced with GPT-5.4 assistance

under topic-specific misconception prompts. Second, three graduate research students in computer science review these generated cases and refine them from the perspective of plausibility, novice-like realism, and concept alignment. Their review acts as a focused knowledge-quality check on the constructed error set rather than a full user study. Cases are revised whenever the reviewers identify an unnatural error pattern, a concept mismatch, or a student rationale that does not fit the intended misconception. The resulting data therefore forms a curated evaluation set for controlled analysis of feedback quality and revision support.

This representation serves two purposes. First, it makes the retrieved material easier to align with learner errors. A diagnosis such as “incorrect window shrink condition” or “missing count update in prefix-sum logic” can be matched to concept entries in a more transparent way than if the system were simply retrieving arbitrary chunks of code or text. Second, it gives the generation stage access to pedagogically interpretable content. Because the knowledge base is small and curated, it functions as a practical intermediate layer for a tutoring system that remains focused on concept-level guidance.

The knowledge base is therefore not intended to be a repository of full accepted solutions. Instead, it stores the instructional material that is useful before a solution is revealed: the concept being exercised, the invariant or relation that should be maintained, the symptom that often appears when the concept is misunderstood, and the smallest hint that can move the learner forward. This distinction is important because a tutoring system that retrieves only finished code can easily reproduce the same answer-substitution problem that direct LLM prompting creates. By retrieving concept cards rather than full solutions, the framework keeps the retrieved evidence aligned with learning support and makes it easier for reviewers to check whether a generated response is grounded in the intended misconception.

#### V. EXPERIMENT

The experiment evaluates whether the framework improves educational feedback quality while still supporting useful revision. We therefore organize the evaluation around three questions. First, we ask whether the proposed framework improves the concept alignment of generated responses. Second, we examine whether the resulting feedback is more pedagogically useful than direct prompting baselines. Third, we consider whether this more restrained, learning-oriented feedback can still support successful revision in a second attempt without defaulting to full-solution disclosure.

To study these questions, we construct a curated evaluation set of 30 LeetCode-style programming tasks spanning major algorithmic categories including hash maps, two pointers, sliding windows, arrays and binary search, monotonic stacks, linked lists, backtracking, trees, graphs, and dynamic programming. The task set is grounded in the final LeetCode Hot100-style subset used in this project. Each task is paired with a structured problem card containing a paraphrased task description, key skill, canonical solution pattern, common bug patterns, misconception tags, and three levels of hints.

For each of the 30 tasks, we construct exactly five erroneous submissions, yielding a total of 150 cases. Each case contains a flawed Python solution, an error-type label, a short student-style rationale, and a representative failure case. Typical patterns include wrong boundary handling, incorrect pointer updates, misuse of prefix sums, missing backtracking-state restoration, invalid monotonic-stack conditions, incorrect dynamic-programming initialization, and confusion between local and global invariants. Table I summarizes the composition of the evaluation set.

TABLE I  
Evaluation-set composition by topic.

| Topic                         | Tasks | Cases | Typical Misconception      |
|-------------------------------|-------|-------|----------------------------|
| Hash / Prefix Sum             | 4     | 20    | Missing count updates      |
| Two Pointers / Sliding Window | 5     | 25    | Invalid shrink condition   |
| Stack / Monotonic Stack       | 3     | 15    | Wrong pop condition        |
| Linked List                   | 3     | 15    | Pointer update errors      |
| Tree / Graph                  | 6     | 30    | Incorrect traversal logic  |
| Binary Search / Arrays        | 4     | 20    | Boundary handling mistakes |
| Backtracking                  | 2     | 10    | Missing state restoration  |
| Dynamic Programming           | 3     | 15    | Invalid state transition   |
| Total                         | 30    | 150   | –                          |

The topic distribution in Table I is designed to cover different forms of reasoning difficulty rather than only different surface-level problem names. Hash and prefix-sum tasks are included because they frequently expose learners' difficulties in maintaining historical counts and recognizing invisible state carried across iterations. Two-pointer and sliding-window tasks capture mistakes in local update logic and invariant maintenance, especially when the correctness of the algorithm depends on when to expand or shrink a window. Stack and monotonic-stack tasks are important because novice errors there often stem from misunderstanding ordering conditions rather than syntax. Linked-list problems contribute pointer-manipulation mistakes that are locally simple but cognitively fragile. Tree and graph tasks represent traversal-based reasoning, where errors often arise from recursive structure, visited-state handling, or incomplete exploration logic. Binary-search and array tasks are particularly useful for boundary-condition analysis. Backtracking tasks make state restoration errors explicit, while dynamic-programming tasks capture more abstract misconceptions involving state definition, initialization, and recurrence transitions. In this sense, the dataset is intended to reflect a spectrum of concept types rather than a mere catalog of algorithms.

We compare three methods, all built on the same base model, GPT-5.4. The first is a direct-feedback baseline in which the model responds to the problem and erroneous submission without retrieval or additional structuring. The second is a direct-fix baseline that prompts the same model in a more explicitly repair-oriented way. The third is our framework, which inserts pedagogical diagnosis and knowledge-grounded retrieval before generating hierarchical

feedback. This setup makes the comparison model-controlled: the main difference between methods lies in how the model is scaffolded, not in which model is used.

The evaluation centers on three metrics. Concept measures how well the generated feedback aligns with the underlying knowledge point of the error. Pedagogy measures how useful the feedback is as learning-oriented guidance rather than direct answer substitution. For both metrics, three graduate research students in computer science independently assign 1–5 scores using a shared annotation rubric, and GPT-5.4 is separately prompted to score the same feedback under the same rubric. The reported value for each case is the average of these four scores, and the numbers in Table II are averages over all 150 cases. Repair (%) measures the success rate of second-attempt revision under each feedback condition. For each case, a second-attempt revision is produced under the corresponding feedback condition, and correction success is then judged independently by the same three research students and by GPT-5.4 with reference to the standard answer and expected behavior. The reported Repair (%) value is the average positive judgment rate across all cases.

The annotation rubric is designed to separate educational value from immediate correctness. A high Concept score requires the feedback to name or clearly imply the relevant algorithmic idea, such as an invariant, counting relation, or recurrence transition, rather than only pointing to a local line of code. A high Pedagogy score requires the response to be actionable, appropriately scoped, and supportive of learner agency. For example, feedback that supplies a full corrected solution too early may improve Repair (%) but receives a lower Pedagogy score because it reduces the learner's opportunity to reason through the misconception. This distinction follows the broader formative-feedback principle that useful feedback should guide the next learning step [5], and it is consistent with recent LLM-feedback studies that emphasize accuracy control and pedagogical framing [22], [23].

TABLE II  
Comparison of feedback quality and revision outcome across baselines.

| Method         | Concept | Pedagogy | Repair (%) |
|----------------|---------|----------|------------|
| Direct-LLM     | 3.04    | 3.25     | 56.4       |
| Direct-LLM-Fix | 2.84    | 3.64     | 79.3       |
| Ours           | 4.20    | 4.28     | 70.4       |

The results reveal a meaningful trade-off. Direct-LLM-Fix achieves the highest repair rate, suggesting that repair-oriented feedback is effective for short-term correction. However, our framework performs substantially better on concept alignment and pedagogical usefulness, indicating that knowledge-grounded hierarchical feedback is more suitable for learning-oriented support. This pattern is consistent with the design goal of the framework: to provide feedback that better supports conceptual understanding and guided revision. In other words, the framework preserves useful correction performance while making the feedback itself more educationally meaningful.

This outcome is useful because it makes the evaluation less dependent on a single success criterion. If repair rate alone were used, a full-solution baseline would appear preferable even when it reduces the learner's need to inspect and revise their own reasoning. Conversely, a feedback system that is conceptually rich but too vague would not be practically useful. The proposed framework occupies an intermediate position: it reduces answer substitution relative to a direct-fix prompt while retaining enough specificity to support second-attempt correction. This is the behavior

expected from a scaffolded programming assistant rather than from an automated solver.

## VI. CONCLUSION

This paper presented a knowledge-augmented hierarchical feedback framework for programming practice. It shifts LLM support from direct answer correction toward concept-oriented tutoring by separating pedagogical diagnosis, knowledge-grounded retrieval, and hierarchical feedback generation. This links an erroneous submission to a teachable misconception before feedback, helping the system generate more concept-aware, scaffolded, and actionable responses. Thus, LLM assistance should be evaluated not only by whether it leads to a correct answer, but also by whether it preserves learners' opportunity to understand why their solution fails and how to revise it.

The study reports an initial evaluation with curated tasks and GPT-5.4-assisted erroneous submissions refined through student review. Results suggest that instructional knowledge and progressively explicit feedback can improve concept alignment and pedagogical usefulness while supporting revision. However, the dataset and revision setting remain limited, and the results should not be interpreted as evidence of long-term learning gains. Future work will evaluate the framework with authentic student submissions, larger learner groups, delayed learning measures, and classroom use of hint, explanation, and repair levels. This keeps the framework lightweight, inspectable, and adaptable to practice or assessment settings.

## REFERENCES

- [1] A. Hellas, J. Leinonen, S. Sarsa, C. Koutchme, L. Kujanpää, and J. Sorva, "Exploring the responses of large language models to beginner programmers' help requests," in *Proceedings of the 2023 ACM Conference on International Computing Education Research-Volume 1*, 2023, pp. 93–105.
- [2] J. Savelka, A. Agarwal, M. An, C. Bogart, and M. Sakr, "Thrilled by your progress! large language models (gpt-4) no longer struggle to pass assessments in higher education programming courses," in *Proceedings of the 2023 ACM Conference on International Computing Education Research-Volume 1*, 2023, pp. 78–92.
- [3] M. Liffiton, B. E. Sheese, J. Savelka, and P. Denny, "Codehelp: Using large language models with guardrails for scalable support in programming classes," in *Proceedings of the 23rd Koli calling international conference on computing education research*, 2023, pp. 1–11.
- [4] J. Leinonen, A. Hellas, S. Sarsa, B. Reeves, P. Denny, J. Prather, and B. A. Becker, "Using large language models to enhance programming error messages," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, 2023, pp. 563–569.
- [5] J. Hattie and H. Timperley, "The power of feedback," *Review of educational research*, vol. 77, no. 1, pp. 81–112, 2007.
- [6] K. VanLehn, "The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems," *Educational psychologist*, vol. 46, no. 4, pp. 197–221, 2011.
- [7] H. Keuning, J. Jeuring, and B. Heeren, "A systematic literature review of automated feedback generation for programming exercises," *ACM Transactions on Computing Education (TOCE)*, vol. 19, no. 1, pp. 1–43, 2018.
- [8] K. M. Ala-Mutka, "A survey of automated assessment approaches for programming assignments," *Computer science education*, vol. 15, no. 2, pp. 83–102, 2005.
- [9] L. Roest, H. Keuning, and J. Jeuring, "Next-step hint generation for introductory programming using large language models," in *Proceedings of the 26th Australasian Computing Education Conference*, 2024, pp. 144–153.
- [10] J. Leinonen, P. Denny, S. MacNeil, S. Sarsa, S. Bernstein, J. Kim, A. Tran, and A. Hellas, "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*, 2023, pp. 124–130.
- [11] V. Aleven, B. M. McLaren, J. Sewall, and K. R. Koedinger, "A new paradigm for intelligent tutoring systems: Example-tracing tutors," *International Journal of Artificial Intelligence in Education*, vol. 19, no. 2, pp. 105–154, 2009.
- [12] B. D. Nye, A. C. Graesser, and X. Hu, "Autotutor and family: A review of 17 years of natural language tutoring," *International Journal of Artificial Intelligence in Education*, vol. 24, no. 4, pp. 427–469, 2014.
- [13] S. Sarsa, P. Denny, A. Hellas, and J. Leinonen, "Automatic generation of programming exercises and code explanations using large language models," in *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*, 2022, pp. 27–43.
- [14] J. Savelka, A. Agarwal, C. Bogart, Y. Song, and M. Sakr, "Can generative pre-trained transformers (gpt) pass assessments in higher education programming courses?" in *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, 2023, pp. 117–123.
- [15] B. Sheese, M. Liffiton, J. Savelka, and P. Denny, "Patterns of student help-seeking when using a large language model-powered programmingassistant," in *Proceedings of the 26th Australasian computing education conference*, 2024, pp. 49–57.
- [16] S. MacNeil, A. Tran, A. Hellas, J. Kim, S. Sarsa, P. Denny, S. Bernstein, and J. Leinonen, "Experiences from using code explanations generated by large language models in a web software development e-book," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, 2023, pp. 931–937.
- [17] M. Kazemitabaar, J. Chow, C. K. T. Ma, B. J. Ericson, D. Weintrop, and T. Grossman, "Studying the effect of ai code generators on supporting novice learners in introductory programming," in *Proceedings of the 2023 CHI conference on human factors in computing systems*, 2023, pp. 1–23.
- [18] P. Denny, J. Leinonen, J. Prather, A. Luxton-Reilly, T. Amarouche, B. A. Becker, and B. N. Reeves, "Prompt problems: A new programming exercise for the generative ai era," in *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, 2024, pp. 296–302.
- [19] B. A. Becker, P. Denny, J. Finnie-Ansley, A. Luxton-Reilly, J. Prather, and E. A. Santos, "Programming is hard-or at least it used to be: Educational opportunities and challenges of ai code generation," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, 2023, pp. 500–506.
- [20] P. Denny, J. Prather, B. A. Becker, J. Finnie-Ansley, A. Hellas, J. Leinonen, A. Luxton-Reilly, B. N. Reeves, E. A. Santos, and S. Sarsa, "Computing education in the era of generative ai," *Communications of the ACM*, vol. 67, no. 2, pp. 56–67, 2024.
- [21] J. Prather, P. Denny, J. Leinonen, B. A. Becker, I. Albluwi, M. Craig, H. Keuning, N. Kiesler, T. Kohn, A. Luxton-Reilly et al., "The robots are here: Navigating the generative ai revolution in computing education," in *Proceedings of the 2023 working group reports on innovation and technology in computer science education*, 2023, pp. 108–159.
- [22] N. Kiesler, D. Lohr, and H. Keuning, "Exploring the potential of large language models to generate formative programming feedback," in *2023 IEEE frontiers in education conference (FIE)*. IEEE, 2023, pp. 1–5.
- [23] T. Phung, J. Cambronero, S. Gulwani, T. Kohn, R. Majumdar, A. Singla, and G. Soares, "Generating high-precision feedback for programming syntax errors using large language models," *arXiv preprint arXiv:2302.04662*, 2023.
- [24] E. Kasneci, K. Seßler, S. K"uchemann, M. Bannert, D. Dementieva, F. Fischer, U. Gasser, G. Groh, S. G"unnemann, E. H"ullermeier et al., "Chatgpt for good? on opportunities and challenges of large language models for education," *Learning and individual differences*, vol. 103, p. 102274, 2023.
- [25] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. K"uttler, M. Lewis, W.-t. Yih, T. Rockt"aschel et al., "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in neural information processing systems*, vol. 33, pp. 9459–9474, 2020.
- [26] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, H. Wang, H. Wang et al., "Retrieval-augmented generation for large language models: A survey," *arXiv preprint arXiv:2312.10997*, vol. 2, no. 1, p. 32, 2023.
- [27] Y. Huang and J. X. Huang, "A survey on retrieval-augmented text generation for large language models," *ACM Computing Surveys*, 2024.
- [28] S. Jacobs and S. Jaschke, "Leveraging lecture content for improved feedback: Explorations with gpt-4 and retrieval augmented generation," in *2024 36th International Conference on Software Engineering Education and Training (CSEE&T)*. IEEE, 2024, pp. 1–5.