Investigating How Quizzes Shape Problem-Solving Strategies in Elementary Programming

Nina Quach[1][0009−0003−1078−3248], Ahana Ghosh[2][0000−0002−0967−5886], Man Su[3][0000−0001−5696−8414], Liina Malva[2,4][0000−0002−9731−8247], Danial Hooshyar[4][0000−0002−9143−6648], Tomohiro Nagashima[1][0000−0003−2489−5016], and Adish Singla[2][0000−0001−9922−0668]

[1] Universität des Saarlandes, Saarland Informatics Campus, Saarbrücken, Germany
s8niquac@uni-saarland.de, nagashima@cs.uni-saarland.de
[2] Max Planck Institute for Software Systems, Saarbrücken, Germany
{gahana, adishs}@mpi-sws.org
[3] Leibniz-Institut für Wissensmedien, Tübingen, Germany
m.su@iwm-tuebingen.de
[4] Tallinn University, Tallinn, Estonia
{liina.malva, danial.hooshyar}@tlu.ee

Abstract. Instructional support is widely used in K-8 programming education, yet its effectiveness is typically evaluated using aggregate performance metrics instead of the problem-solving processes it fosters. To examine how instructional support with varying levels of cognitive engagement influences these processes, we study higher- and lower-order instructional support, grounded in Bloom's revised taxonomy and instantiated as quizzes. Specifically, we conduct a mixed-methods analysis on K-8 classroom study data across three elementary programming curricula: no quiz-based support (Base), lower-order fill-in-the-gap quizzes (Fill) targeting applying and analyzing levels, and higher-order quizzes (Ace) targeting analyzing, evaluating, and creating levels. Focusing on novel post-learning transfer tasks, we analyze failure and recovery modes or patterns, and characterize problem-solving strategies through expert annotation and Affinity Diagramming. Our results show that Ace exhibits higher iterative debugging and stronger transfer outcomes, with qualitative patterns suggesting greater metacognitive engagement. In contrast, Fill shows weaker independent recovery, while Base supports transfer but risks persistent misconceptions. From a design perspective, these findings suggest combining unscaffolded exploration with targeted higher-order supports to balance independent strategy formation and conceptual accuracy.

Keywords: elementary programming · computational thinking · quizzes

1 Introduction

Programming education for K-8 learners increasingly incorporates a range of instructional supports, including curriculum design [13,26], worked examples [8,32],

adaptive feedback [16,19], and interleaved practice [11] to promote the development of computational thinking skills [17,24]. Prior work has shown that such support can improve learning outcomes, with evidence drawn from performance metrics such as task success rates, quiz scores, and task completion rates [13,14]. However, these metrics offer limited insight into the underlying learning processes that such support fosters. In particular, they do not reveal how learners reason about their problem-solving approaches, recover from unsuccessful attempts, or develop effective problem-solving strategies over time [23,31].

Understanding the problem-solving and recovery processes is especially important once instructional supports are withdrawn, as learners must ultimately self-regulate in novel, unscaffolded contexts. Engaging learners in transfer tasks after scaffolded practice is therefore essential for assessing whether learned strategies generalize beyond the instructional setting [7,21]. Performance in such settings reflects not only what the learners know, but also how they internally regulate their problem-solving processes. For example, whether they engage in systematic debugging [31], adapt strategies in response to failure, or rely on shallow heuristics [13,27]. The *nature* of non-completion, whether characterized by disengagement or persistent failed attempts (hereafter, *failure mode*), is itself informative about self-regulatory processes. However, the relationship between instructional design and these process-level behaviors during transfer remains underexplored.

In this work, we focus on elementary programming curricula interleaving write-code tasks with quizzes of varying cognitive demand to investigate how different levels of cognitive support shape learners' problem-solving, particularly with respect to failure and recovery modes. Specifically, we study how the design of interleaved quizzes during the learning phase, grounded in Bloom's cognitive levels [1,4], influences learners' problem-solving behavior on novel, unscaffolded post-learning tasks (see Figure 1). To this end, we pose the following research questions (RQs): **(RQ1)** *How do different quiz designs shape learners' patterns of failure and recovery on novel transfer tasks?* and **(RQ2)** *How do problem-solving strategies differ across quiz designs when learners face novel transfer tasks?*

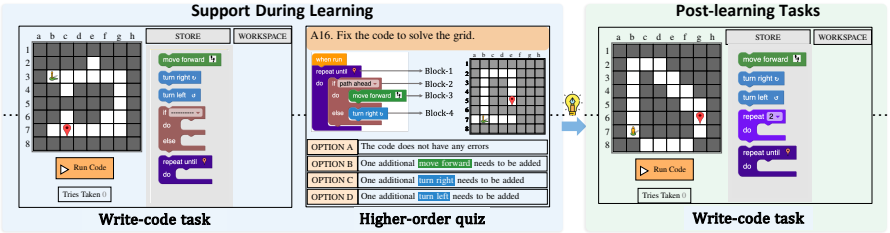


Fig. 1: Instructional pipeline in elementary-level visual programming. Learners complete write-code tasks interleaved with quizzes of varying levels of cognitive demand in the learning phase. Illustrated here is a higher-order debugging quiz. Later, they complete unscaffolded write-code tasks in the post-learning phase.

2 Related Work

Instructional support in K-8 programming. Structured instructional support, ranging from worked examples [32] and adaptive feedback [16] to interleaved practice exercises such as *Parsons problems* [19] and *fill-in-the-gap tasks* [11,14], is widely used in K-8 programming curricula to scaffold computational thinking [26,29]. These formats improve learning-phase performance by reducing cognitive load and guiding learners toward correct solutions [30,32], yet evaluations rely on aggregate outcome measures offering limited insight into the learning processes these supports foster, particularly once scaffolding is removed.

Cognitive levels of support, Bloom's taxonomy, and quizzes. A key dimension along which supports differ is the level of required cognitive engagement. Bloom's revised taxonomy [1,4] organizes cognitive skills from lower-order levels (remembering, understanding, applying) to higher-order levels (analyzing, evaluating, creating), providing a principled basis for designing instructional supports. Interleaved quizzes operationalize this spectrum within programming curricula: fill-in-the-gap quizzes target recognition and lower-order application, while quizzes requiring learners to debug code, compare solutions, or reason about task design engage higher-order analysis, evaluation, and creation [15]. The desirable difficulties framework [2,3] explains why these distinctions matter: activities requiring effortful retrieval under challenge strengthen transfer more effectively than fluent or less demanding practice [22,28]. Consistent with this framework, prior work finds that higher-order quiz formats produce stronger post-learning outcomes than lower-order formats [13], though the behavioral mechanisms underlying this advantage remain unclear.

Process-level behaviors and self-regulated learning. Effective programming problem-solving requires self-regulated learning (SRL), including monitoring understanding, systematically debugging, and adapting strategies when attempts fail [33]. Novices frequently resort to trial-and-error or disengage rather than diagnosing errors systematically [23,25]. As a result, metacognitive behaviors such as (deliberate) task skipping, self-monitoring, and selective re-engagement are less common without instructional supports that prompt reflection and reasoning [25,27]. Supports that prompt reasoning are hypothesized to build error-monitoring habits and foster SRL [3,6], whereas those directly providing correct solutions may leave learners without self-regulation strategies for independent transfer [20]. The relationship between instructional support design and these process-level behaviors fostered during transfer remains underexplored, which is a gap this study directly addresses.

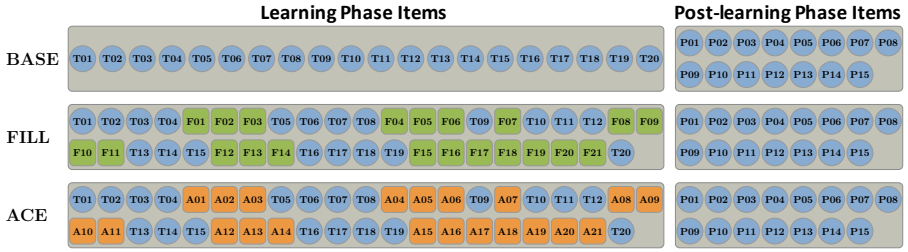
3 Method

3.1 Study Context and Participants

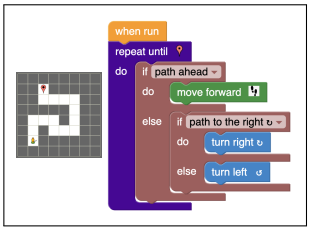
The data used in this work were collected in a prior study [13]. The study followed a two-stage quasi-experimental design in an introductory block-based programming curriculum for Grades 6–7 across 22 schools in Estonia. In the learning

phase, students were randomly assigned to one of three curriculum conditions at the class level, spanning up to two 45-minute lessons, with 20 common write-code tasks (T01–T20) from the *Hour of Code: Maze Challenge* [10], differing only in interleaved quizzes: Base (no quizzes), Fill (21 fill-in-the-gap quizzes F01–F21, targeting lower-order cognitive levels), and Ace (21 higher-order quizzes A01–A21, involving code analysis, debugging, and task design). Quiz designs were informed by Bloom's revised cognitive levels [1,4]: Fill targets applying and analyzing, while Ace targets analyzing, evaluating, and creating, adapted from the work of [15]. One week later, all three curriculum conditions completed an unscaffolded post-learning phase of 15 write-code tasks (P01–P15) on the same platform in a 45-minute class lesson. An overview of the curriculum is shown in Figure 2a and specific examples of write-code tasks and quizzes are shown in Figure 6 in Appendix A.

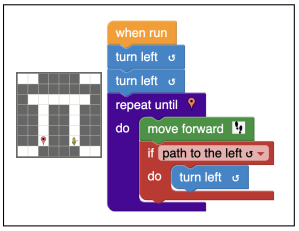
The dataset comprised $N = 575$ students between the ages of 10 and 15 (245 female, 330 male) across 53 classes in 22 schools, with an average class size of 16 students. Each of the classes was randomly assigned to one of three curricula as follows: Base ($n = 171$ students, 16 classes), Fill ($n = 175$ students, 18 classes), and Ace ($n = 229$ students, 19 classes). There were no significant group differences in prior programming experience or academic background, assessed



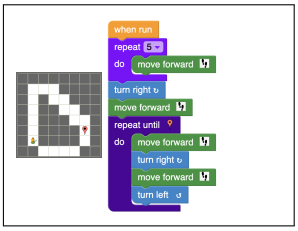
(a) Sequence of items in each curriculum group. T01–T20 represent common write-code tasks during learning and the green F01–F21 and orange A01–A21 squares represent interleaved quizzes in Fill and Ace respectively. P01–P15 represent write-code tasks from the post-learning phase.



(b) P07



(c) P10



(d) P12

Fig. 2: Overview of curriculum groups and post-learning tasks used for analysis. (a) shows the sequence of items across the three curriculum groups. (b)-(d) illustrate the selected post-learning tasks along with their representative solution codes.

via self-reported questions on demographics, programming interest and skill, prior programming environments, and familiarity with block-based constructs and code comprehension; about 50% of participants in each curriculum reported having prior programming experience, and only 3% were familiar with the *Hour of Code: Maze Challenge* domain.

To analyze process-level behaviors in depth, we focus on three post-learning tasks, P07, P10, and P12, selected to span near-transfer and novel-transfer demands. Tasks P01–P09, P15 are either direct replicas or mild variants of learning-phase tasks, providing limited diagnostic signal; of the novel post-learning tasks (P10–P14), we select P10 and P12 for their distinct reasoning demands, and retain P07 as a near-transfer reference. Specifically, P07 is structurally identical to learning-phase task T20, the most complex task encountered during learning, involving nested conditionals. Having P07 makes it possible to compare performance on a near-transfer task. In contrast, P10 and P12 are novel transfer tasks whose solution structures were not encountered during the learning phase: P10 requires sequencing of turns prior to a `repeatUntilGoal` loop, while P12 emphasizes efficient problem decomposition under a tight block constraint. All three tasks require non-trivial path planning and debugging, making them well-suited for analyzing differences in learner strategies and failure modes. Figure 2(b)–(d) illustrates the task grids along with their optimal solutions.

3.2 Analysis Framework

We address RQ1 through quantitative analysis of failure and recovery patterns. For RQ2, we conduct qualitative analysis of students' problem-solving strategies.

Quantitative analysis (RQ1). We analyze the complete dataset of student trajectory logs ($N = 575$) to characterize failure and recovery behaviors on post-learning tasks P07, P10, and P12. Here, *failure* refers to a trajectory-level outcome, i.e., the unsuccessful final state of a student's attempt sequence on a task or the absence of any attempt, rather than a judgment of individual ability. Failure is decomposed into three categories: failure with at least one attempt (students attempted but did not succeed), failure by active skipping where the time spent on a task is greater than zero (i.e., the student viewed the task but decided to skip), and failure with no engagement where a task was not even viewed. To characterize success and debugging behavior, we first distinguish between first-try success and recovery from failure. Recovery refers to eventual success following one or more incorrect attempts, and is further decomposed into immediate recovery (success on the first re-attempt), and delayed recovery after multiple re-attempts. To evaluate differences across curriculum conditions, we perform pairwise comparisons using chi-squared tests [9] with Bonferroni correction factor of 3 for multiple comparisons [12].

Qualitative analysis (RQ2). To examine process-level strategies, we conduct a qualitative analysis on a stratified random sample of 75 students (25 per curriculum group), matched on performance and demographics. This covers 225 student attempt sequences or trajectories (75 students $\times$ 3 tasks: P07, P10, P12). Figure 5 illustrates example student trajectories on task P10. Two authors with programming education expertise independently analyzed trajectories using a

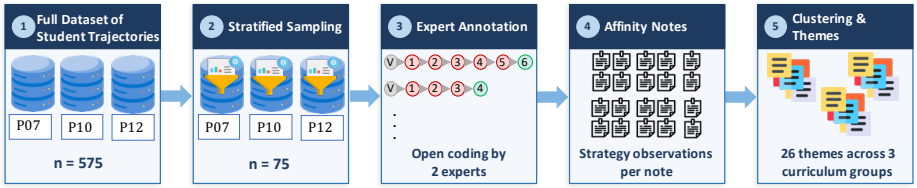


Fig. 3: Qualitative analysis pipeline applied to a stratified sample of 75 students. A total of $75 \times 3 = 225$ student attempt sequences (trajectories) across tasks P07, P10, and P12 are processed through a five-stage Affinity Diagramming [18] pipeline to derive strategy themes across the three curricula.

custom annotation platform, recording open-ended observations across five dimensions: task decomposition, code execution, debugging behavior, metacognitive engagement, and code-level misconceptions. Annotations were synthesized using Affinity Diagramming [18] (see Figure 3), where individual observations were iteratively clustered into low-level patterns and then consolidated into 26 themes across tasks. Theme frequencies were computed per curriculum. To ensure consistency in annotation, both authors independently coded a shared subset of 15 students' trajectories in a blind pilot phase. Disagreements were discussed and resolved iteratively until full agreement was reached, aligning interpretations across the five dimensions and refining the coding scheme prior to the remaining annotations.

4 Results

4.1 Descriptive Statistics

In the learning phase, success rates on the common 20 write-code tasks (T01–T20) were high across all groups, with Fill (88.1%) and Ace (88.5%) significantly outperforming Base (80.8%). In the post-learning phase (P01–P15), this pattern shifts: Ace (79.9%) and Base (76.4%) significantly outperform Fill (70.4%). Performance on quizzes was high for both quiz-based groups (Ace quizzes: 93.1%; Fill quizzes: 91.7%) with no significant differences between them. Focusing on the three post-learning tasks (P07, P10, P12), we observe variation in solution strategies, with 20 unique solutions for P07, 7 for P10, and 2 for P12. Success rates averaged across these three tasks differ significantly across conditions, with Ace (67.8%) achieving the highest performance, followed by Base (61.4%), and Fill (52.6%), consistent with overall post-learning trends. Next, we analyze these differences with regard to failure and recovery modes.

4.2 RQ1: Failure and Recovery Modes

To address RQ1, we analyze students' behavior on post-learning tasks, P07, P10, and P12. Results are summarized in Figure 4, aggregated across Grades 6 and 7; no significant grade-specific differences in performance were found.

Metric	Base n=171	Fill n=175	Ace n=229
Failure Rate	38.6 (2.15)^{b}	47.4 (2.18)^{ac*}	**32.2** (1.78)^{b*}
▶ With ≥1 Attempt	18.9 (1.73)^{b}	26.3 (1.92)^{ac*}	**18.5** (1.48)^{b*}
▶ Active Skipping	7.2 (1.14)	7.2 (1.13)	8.6 (1.07)
▶ No Engagement	12.5 (1.46)^{c*}	13.9 (1.51)^{c*}	**5.1** (0.84)^{a*b*}
Success Rate	61.4 (2.15)^{b}	52.6 (2.18)^{ac*}	**67.8** (1.78)^{b*}
▶ First-try Success	14.2 (1.54)	12.8 (1.46)^{c}	**18.5** (1.48)^{b}
▶ Recovery	47.2 (2.20)	39.8 (2.14)^{c*}	**49.3** (1.91)^{b*}
▶ Immediate	11.9 (1.43)	10.3 (1.33)	10.3 (1.16)
▶ >1 Re-attempt	35.3 (2.11)	29.5 (1.99)^{c*}	**39.0** (1.86)^{b*}

Fig. 4: Hierarchical decomposition of task outcomes averaged across three novel tasks P07, P10, and P12 for each curriculum group. We report the mean percentage and standard error (SE) per curriculum and metric. Superscripts denote pairwise significance: ^{a} differs from Base; ^{b} differs from Fill; ^{c} differs from Ace with $p < \frac{0.05}{3}$; $*$ indicates $p < \frac{0.01}{3}$.

Failure modes. Fill has the highest overall failure rate, significantly above both Base and Ace. The composition of failures also differs across conditions. Fill shows the highest rate of failures involving at least one attempt, suggesting unsuccessful engagement. Active skipping rates, where a student views a task but does not attempt it, are similar across conditions. A key difference lies in no-engagement failures, where tasks are never viewed: Ace shows significantly fewer cases than both Base and Fill. Together, these findings suggest that students in Ace exhibit a more engagement-oriented profile and tend to interact with tasks even when those interactions are unsuccessful rather than bypassing them entirely.

Recovery modes. Ace has the highest success rate, driven by both a higher first-try success rate and a higher recovery rate after initial unsuccessful attempts. Fill performs worst on both metrics. Recovery rates after a single immediate fix are similar across conditions, but Ace students recover more often through multiple re-attempts. This suggests that the advantage of Ace lies in sustained, iterative debugging rather than quick corrections.

4.3 RQ2: Problem-Solving Strategies and Misconceptions

To examine strategies underlying the recovery patterns observed in RQ1, we conducted Affinity Diagramming of 225 annotated trajectories, resulting in 26 sub-categories across six high-level categories (see Table 1). Given the small counts per sub-category, we treat our findings as emergent qualitative patterns rather than definitive evidence. Sub-categories within each high-level category are mutually exclusive per trajectory. Below, we discuss each high-level category.

– **Metacognitive engagement** captures how students plan, monitor, and regulate their approach to tasks. Fill shows higher counts of *path planning* and *no attempts*. *Transfer learning* is more frequent in Base and Ace.

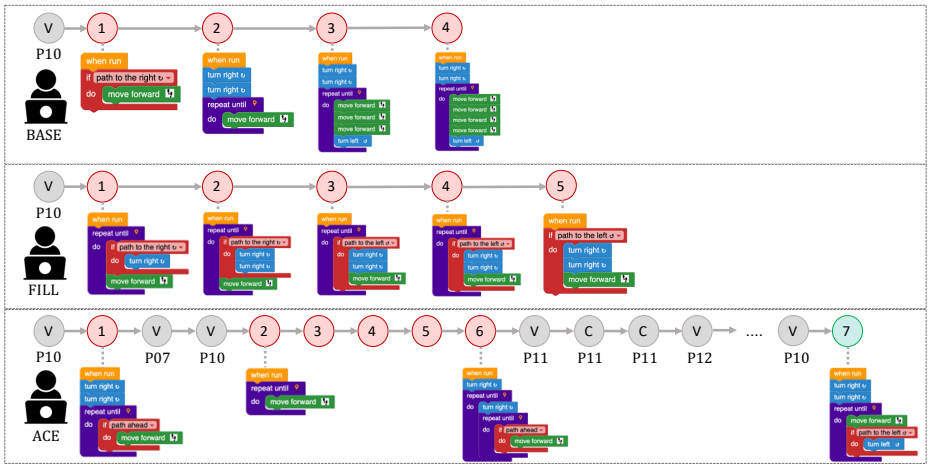


Fig. 5: Example student trajectories on task P10, one per curriculum, selected as characteristic illustrations of observed strategy differences. Timelines read left to right; "V" marks a task-viewing event and "C" a code attempt on a task other than P10. Red and green circles indicate incorrect and correct attempts, respectively. Observed sub-categories (defined in Table 1) are: [Base] *bottom-up programming misconceptions; pragmatic debug*; [Fill] *path planning; top-down perfectionist construction; random substitution*; [Ace] *bottom-up construction; selective try*, as they diverted to other tasks before ultimately completing P10.

- **Solution construction** captures how students build their solution to a task. Ace shows the highest *first-try perfect* count, consistent with RQ1, while Base shows more *almost perfect* and *bottom-up construction* patterns.
- **Debugging** captures how students recover from incorrect attempts. The clearest contrast is *no debug after error*, highest in Fill, suggesting lower-order scaffolding may not foster debugging strategies. *Hybrid debugging* counts are same across all conditions. Base shows the highest count of *pragmatic debug*.
- **Ad-hoc substitution** captures unstructured code changes without clear reasoning. *Random substitution* is more in Ace and Fill than in Base.
- **Unique strategies** covers distinctive one-off behaviors observed across trajectories. *Cognitive click*, *alternative solutions*, and *hybrid execution* appear at low, broadly similar counts across conditions.
- **Programming misconceptions** captures errors reflecting flawed mental models of programming constructs. Base shows the highest counts of *bottom-up misconceptions* and *sub-optimal solutions*, suggesting unscaffolded practice carries a higher risk of reinforcing incorrect programming concepts. Fundamental programming misconceptions are still observed across all three conditions.

An important observation is that neutral sub-categories (see Table 1) do not operate in isolation, and instead their outcome depends on co-occurring strategies. The P10 Ace trajectory in Figure 5 illustrates this: bottom-up construction

Table 1: Occurrence counts of 26 qualitative sub-categories across six high-level categories after Affinity Diagramming. **Bold** values indicate the maximum count per row when the margin over the next-highest is ≥ 2. $^{+}$ productive theme associated with success on task; $^{-}$ non-productive theme associated with failure on task; $^{\|}$ neutral theme where task outcome depends on co-occurring strategies.

Category	Sub-Category	Description	Base	Fill	Ace
Metacognitive engagement	Selective try^{+}	Strategic deferral; later solved immediately	0	1	2
	Path planning$^{\|}$	Visible solution path planning	9	**11**	3
	Zero-duration task^{-}	Never opened the task	5	10	10
	No attempts$^{\|}$	Did not attempt after inspection	2	**7**	5
	Transfer learning$^{\|}$	Inspects prior tasks and applies concepts	8	3	7
Solution construction	First-try perfect^{+}	Correct first-attempt solution	15	8	**21**
	Almost perfect^{+}	Minor fix after near-correct solution	**11**	9	5
	Top-down perfectionist$^{\|}$	Top-down solution with minimal iteration	1	3	2
	Hybrid decomposition$^{\|}$	Top-down and bottom-up decomposition	3	2	2
	Bottom-up$^{\|}$	Builds solution incrementally	**11**	9	3
Debugging	Perfectionist^{+}	Near-perfect attempt with structured fixes	4	3	3
	Top-down systematic$^{\|}$	Top-down approach with structured fixes	3	4	3
	Hybrid debugging$^{\|}$	Multiple strategies when debugging	2	2	2
	No debug after error^{-}	No debugging after encountering errors	1	**6**	0
	Trial-and-error$^{\|}$	Trial-and-error with partial reasoning	1	1	0
	Pragmatic debug$^{\|}$	Small step debugging approach	**8**	5	4
Ad-hoc substitution	Random substitution^{-}	Random substitutions in code	4	5	**7**
	Overcomplicating^{-}	Uses excess or unnecessary blocks	1	3	4
	Basic blocks only$^{\|}$	Traces the task grid with basic blocks	1	1	2
Unique strategies	Hybrid execution$^{\|}$	Hybrid use of code execution strategies	4	3	1
	Cognitive click^{+}	Sudden insight after initial wrong attempts	1	2	1
	Alternative solutions^{+}	Uses workaround optimized solutions	4	3	3
Programming misconceptions	Severe^{-}	Fundamentally wrong mental model	**4**	2	2
	Top-down^{-}	Errors during top-down decomposition	4	2	4
	Bottom-up^{-}	Errors during bottom-up decomposition	**7**	3	0
	Sub-optimal solutions^{-}	Uses workaround sub-optimized solutions	**6**	4	1

(neutral) succeeds in combination with selective task engagement, as the student strategically diverts to P07, P11, and P12 before returning to solve P10. The P12 trajectories in Figure 7(d–f) (in Appendix B) reinforce this: both Fill and Ace students employ *path planning*, yet the Fill student requires substantially more attempts, suggesting path planning alone does not determine outcome. Similarly, for P07 trajectories in Figure 7(a–c), both Fill and Ace students use *pragmatic debug*, yet only the Ace student arrives at an optimal solution. Overall, curriculum differences are most pronounced in *no debug after error* and *first-try perfect*, which are sub-categories that directly link to the recovery patterns in RQ1; while *programming misconceptions* are most concentrated in Base trajectories.

5 Discussion

5.1 Higher-Order Scaffolding and Iterative Debugging

Our results show that Ace consistently outperforms Fill not only in overall success but in how learners engage with unsuccessful tasks. The clearest contrast is in no-engagement failures (tasks which were never viewed) where Ace rates

are significantly lower, and in recovery through multiple re-attempts, where Ace outperforms both curricula. Fill shows the opposite pattern: the highest failure rate, the highest rate of no debug after error, and the weakest recovery after multiple unsuccessful attempts.

These patterns are consistent with the desirable difficulties framework [2,3], which predicts that cognitively challenging scaffolds promote deeper processing and more robust transfer. Higher-order quizzes, grounded in Bloom's levels of analyzing, evaluating, and creating [4,1], may foster the error-monitoring and iterative correction behaviors that support self-regulated problem-solving [23,33]. In contrast, lower-order scaffolding may reduce cognitive load during learning but foster reliance on external guidance, limiting independent problem-solving once support is removed [30,20]. We note, however, that these interpretations are grounded in behavioral patterns rather than direct measures of cognitive processes.

5.2 Unscaffolded Learning and Misconception Risk

Our results further show that, while Base and Ace achieve comparable overall success rates in the post-learning phase, they differ in the problem-solving processes underlying that performance. Base learners show the highest rates of *bottom-up misconceptions* and *sub-optimal solutions* in the qualitative analysis, consistent with unscaffolded practice risking reinforcing incorrect mental models, a documented challenge in novice programming education [25,5]. At the same time, Base shows reasonable recovery rates, consistent with prior work showing that unscaffolded exploration can promote independent strategy formation [2,20].

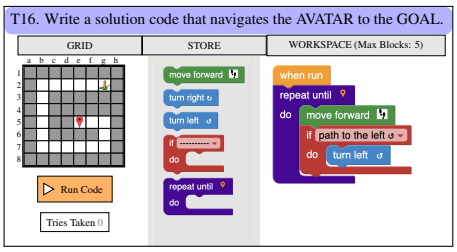
Together, these patterns point to a trade-off: unscaffolded learning may support transfer for learners who construct correct models independently, while leaving others at risk of persistent errors without corrective feedback. From a design perspective, these findings suggest that combining periods of unscaffolded exploration with targeted higher-order scaffolds could balance independent strategy formation while preventing misconceptions.

6 Conclusion

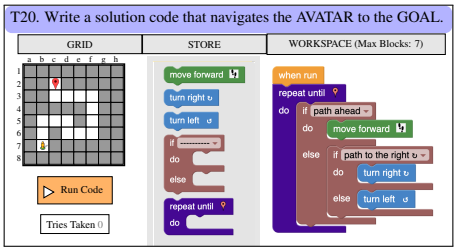
In this work, we investigated how different levels of quiz-based support in elementary programming curricula shape learners' problem-solving processes during the post-learning phase. Our findings suggest that K-8 programming curricula should combine periods of unscaffolded exploration with targeted higher-order scaffolds to balance independent strategy formation with misconception prevention. While the qualitative stratified sample of 75 students is sufficient for emergent themes, larger per-condition samples would strengthen cross-condition comparisons. Similarly, the three high-diagnostic post-learning tasks provide focused insight, though a broader task set would help assess the generalizability of the observed strategy patterns. Future work could explore adaptive scaffolding that dynamically adjusts quiz type and difficulty to balance cognitive support with independent strategy formation. Generalizing these findings across programming domains and learner populations also remains an open direction.

A Examples of Learning Phase Tasks and Quizzes

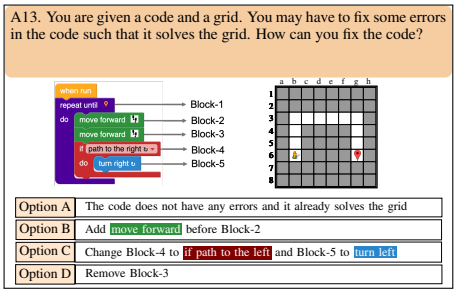
Figure 6 illustrates two write-code tasks common to all three curriculum groups, as well as representative quizzes from the FILL and ACE curricula.



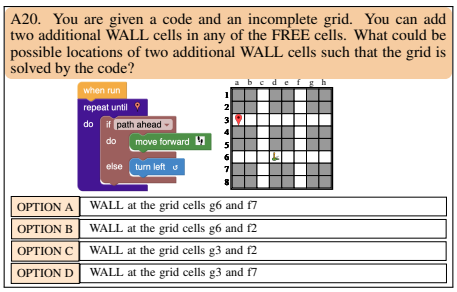
(a) Task T16 with its solution code



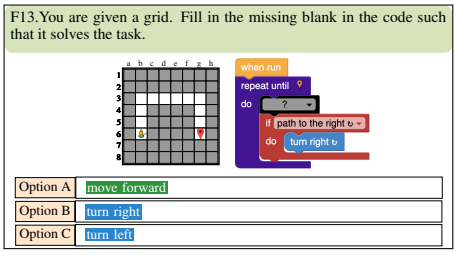
(b) Task T20 with its solution code



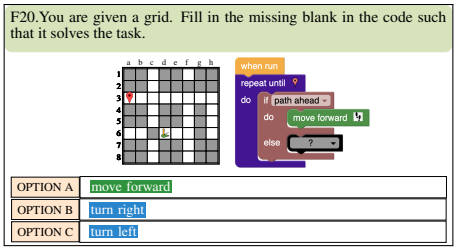
(c) Quiz A13 (Solution: Option D)



(d) Quiz A20 (Solution: Option B)



(e) Quiz F13 (Solution: Option A)

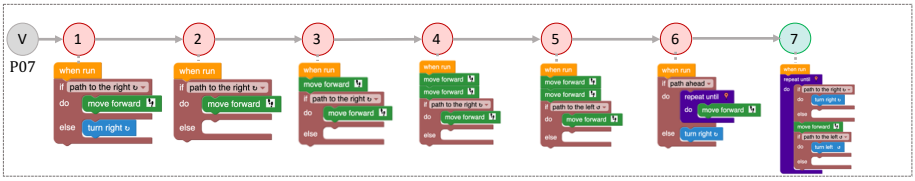


(f) Quiz F20 (Solution: Option C)

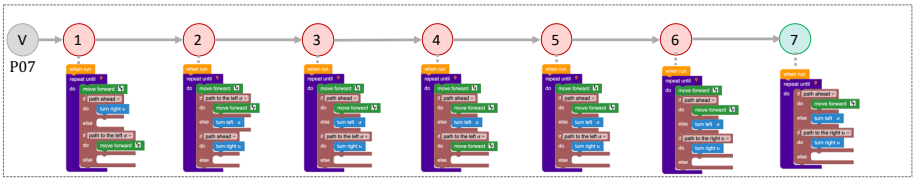
Fig. 6: (a) and (b) show examples of write-code tasks from the learning phase. (c) and (d) show quizzes based on code debugging and task design from ACE. (e) and (f) show corresponding quizzes based on fill-in-the-gap questions from FILL.

B Illustrative Student Trajectories

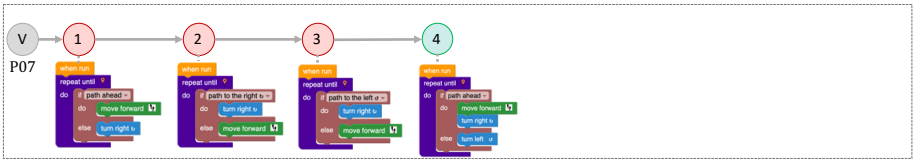
Figure 7 shows one illustrative trajectory per curriculum group for tasks P07 and P12, annotated with qualitative themes (see Table 1 for definitions). Subcaptions name the sub-categories exhibited by each trajectory.



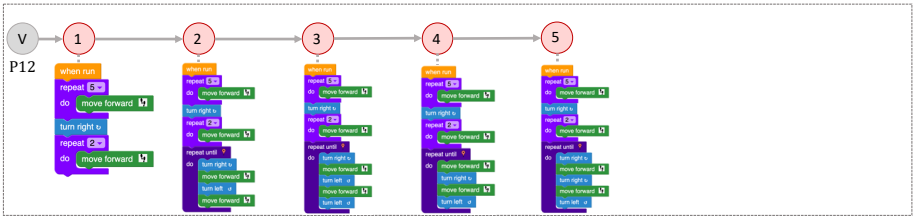
(a) P07 BASE: Bottom-up construction; cognitive click; sub-optimal solutions



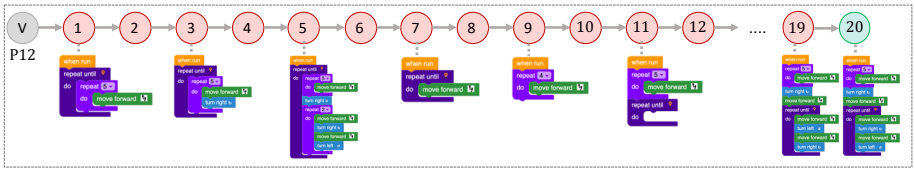
(b) P07 FILL: Pragmatic debug; sub-optimal solutions



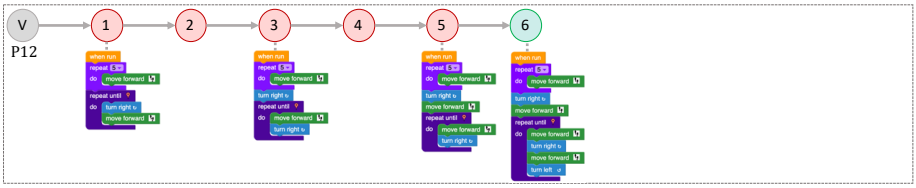
(c) P07 ACE: Pragmatic debug; alternative solutions



(d) P12 BASE: Top-down perfectionist construction; top-down systematic debugging



(e) P12 FILL: Path planning; bottom-up construction



(f) P12 ACE: Path planning; pragmatic debug

Fig. 7: Example student trajectories for tasks P07 (a–c) and P12 (d–f), one per curriculum condition, selected as characteristic illustrations of condition-specific strategy differences. Timeline and symbol conventions follow Figure 5. Subcaptions state the primary observed qualitative sub-categories; all six trajectories are from distinct students.

References

1. Anderson, L.W., Krathwohl, D.R.: A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives: complete edition. Addison Wesley Longman, Inc. (2001)
2. Bjork, E.L., Bjork, R.A.: Making things hard on yourself, but in a good way: Creating desirable difficulties to enhance learning. In: Psychology and the Real World, pp. 56–64. Worth Publishers (2011)
3. Bjork, R.A.: Desirable Difficulties Perspective on Learning. In: Pashler, H. (ed.) Encyclopedia of the Mind. SAGE Reference, Thousand Oaks, CA (2013)
4. Bloom, B.S., Engelhart, M.D., Furst, E.J., Hill, W.H., Krathwohl, D.R., et al.: Taxonomy of educational objectives: The classification of educational goals. Handbook 1: Cognitive domain. Longman New York (1956)
5. Boulay, B.D.: Some Difficulties of Learning to Program. Journal of Educational Computing Research **2**(1), 57–73 (1986)
6. Chen, M., Feng, C.: Goal-Setting and Self-Explanation in Elementary Programming: A Self-Regulated Learning Study. Journal of Computer Assisted Learning **42**(1) (2026)
7. Chen, O., Kalyuga, S., Sweller, J.: When Instructional Guidance is Needed. Educational and Developmental Psychologist **33**(2), 149–162 (2016)
8. Chi, M.T., Bassok, M., Lewis, M.W., Reimann, P., Glaser, R.: Self-Explanations: How Students Study and Use Examples in Learning to Solve Problems. Cognitive Science **13**(2), 145–182 (1989)
9. Cochran, W.G.: The χ^2 Test of Goodness of Fit. The Annals of Mathematical Statistics **23**(3), 315 – 345 (1952)
10. Code.org: Hour of Code: Classic Maze Challenge. https://studio.code.org/s/hourofcode (2022)
11. Code.org: Express Course. https://code.org/express-2024 (2024)
12. Dunn, O.J.: Multiple Comparisons among Means. Journal of the American Statistical Association **56**(293), 52–64 (1961)
13. Ghosh, A., Malva, L., Gotovos, A., Hooshyar, D., Singla, A.: Exploring the Impact of Quizzes Interleaved with Write-Code Tasks in Elementary-Level Visual Programming. In: Proceedings of the Technical Symposium on Computer Science Education V. 1, SIGCSE TS. pp. 381–387. ACM (2025)
14. Ghosh, A., Malva, L., Gotovos, A., Hooshyar, D., Singla, A.: The Right Kind of Help: Evaluating the Effectiveness of Intervention Methods in Elementary-Level Visual Programming. CoRR **abs/2512.11735** (2025)
15. Ghosh, A., Malva, L., Singla, A.: Analyzing-Evaluating-Creating: Assessing Computational Thinking and Problem Solving in Visual Programming Domains. In: Proceedings of the Technical Symposium on Computer Science Education, SIGCSE. pp. 387–393. ACM (2024)
16. Ghosh, A., Tschiatschek, S., Devlin, S., Singla, A.: Adaptive Scaffolding in Block-Based Programming via Synthesizing New Tasks as Pop Quizzes. In: Artificial Intelligence in Education: International Conference, AIED, Proceedings, Part I. p. 28–40. Springer-Verlag, Berlin, Heidelberg (2022)
17. Grover, S., Pea, R.: Computational Thinking in K–12: A Review of the State of the Field. Educational Researcher **42**(1), 38–43 (2013)
18. Holtzblatt, K., Beyer, H.: Contextual Design: Evolved. Morgan & Claypool Publishers (2014)

19. Hou, X., Ericson, B.J., Wang, X.: Using Adaptive Parsons Problems to Scaffold Write-Code Problems. In: ICER: ACM Conference on International Computing Education Research, Volume 1. pp. 15–26. ACM (2022)
20. Kapur, M., Bielaczyc, K.: Designing for Productive Failure. Journal of the Learning Sciences **21**(1), 45–83 (2012)
21. Kirschner, P.A., Sweller, J., Clark, R.E.: Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experiential, and Inquiry-Based Teaching. Educational Psychologist **41**(2), 75–86 (2006)
22. Kornell, N., Hays, M.J., Bjork, R.A.: Unsuccessful retrieval attempts enhance subsequent learning. Journal of Experimental Psychology: Learning, Memory, and Cognition **35**(4), 989–998 (2009)
23. Loksa, D., Ko, A.J.: The Role of Self-Regulation in Programming Problem Solving Process and Success. In: Proceedings of the Conference on International Computing Education Research, ICER. pp. 83–91. ACM (2016)
24. Lye, S.Y., Koh, J.H.L.: Review on teaching and learning of computational thinking through programming: What is next for K-12? Computers in Human Behavior **41**, 51–61 (2014)
25. McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L.C., Simon, B., Thomas, L., Zander, C.: Debugging: A review of the literature from an educational perspective. Computer Science Education **18**(2), 67–92 (2008)
26. Parsons, D., Haden, P.: Parson's programming puzzles: a fun and effective learning tool for first programming courses. In: Proceedings of the Australasian Conference on Computing Education - Volume 52. p. 157–163 (2006)
27. Pechorina, Y., Anderson, K., Denny, P.: Metacodenition: Scaffolding the Problem-Solving Process for Novice Programmers. In: Proceedings of the 25th Australasian Computing Education Conference. p. 59–68. ACE '23, ACM (2023)
28. Roediger III, H.L., Karpicke, J.D.: Test-Enhanced Learning: Taking Memory Tests Improves Long-Term Retention. Psychological Science **17**(3), 249–255 (2006)
29. Ruf, A., Berges, M., Hubwieser, P.: Classification of Programming Tasks According to Required Skills and Knowledge Representation. In: International Conference on Informatics in Schools: Situation, Evolution, and Perspectives, ISSEP. vol. 9378, pp. 57–68. Springer (2015)
30. Sweller, J.: Cognitive Load Theory, Learning Difficulty, and Instructional Design. Learning and Instruction **4**(4), 295–312 (1994)
31. Yang, S., Baird, M., O'Rourke, E., Brennan, K., Schneider, B.: Decoding Debugging Instruction: A Systematic Literature Review of Debugging Interventions. ACM TOCE **24**(4), 1–44 (2024)
32. Zhi, R., Price, T.W., Marwan, S., Milliken, A., Barnes, T., Chi, M.: Exploring the Impact of Worked Examples in a Novice Programming Environment. In: Proceedings of the Technical Symposium on Computer Science Education, SIGCSE. pp. 98–104. ACM (2019)
33. Zimmerman, B.J.: Becoming a Self-Regulated Learner: An Overview. Theory Into Practice **41**(2), 64–70 (2002)