Volume: 11 Issues: 85 Special Issue [August, 2026] pp. 614 - 632
Journal of Islamic, Social, Economics and Development (JISED)
eISSN: 0128-1755
Journal website: academicinspired.com/jised
DOI: 10.55573/JISED.118544

NOVICE PROGRAMMERS' PERCEPTIONS OF A SELF-QUESTIONING GUIDELINE TO METACOGNITIVELY SCAFFOLD PROGRAMMING PROBLEM-SOLVING

Rosni Ramle[1*]
Hannes Tegelbeckers[2]
D'oria Islamiah Rosli[3]
Frank Büning[4]

[1] Centre for Diploma Studies, Universiti Tun Hussein Onn Malaysia, Malaysia
Email: rosni@uthm.edu.my
[2] Faculty of Humanities, Otto von Guericke University Magdeburg, Germany
Email: hannes.tegelbeckers@ovgu.de
[3] Faculty of Technical and Vocational Education, Universiti Tun Hussein Onn Malaysia, Malaysia
Email: doria@uthm.edu.my
[4] Faculty of Humanities, Otto von Guericke University Magdeburg, Germany
Email: frank.buenning@ovgu.de
* Corresponding Author

Article history
Received date : 19-6-2026
Revised date : 20-6-2026
Accepted date : 25-7-2026
Published date : 1-8-2026

To cite this document:
Ramle, R., Tegelbeckers, H., Rosli, D. I., & Büning, F. (2026). Novice programmers' perceptions of a self-questioning guideline to metacognitively scaffold programming problem-solving. *Journal of Islamic, Social, Economics and Development (JISED)*, 11 (85), 614 – 632.

Abstract: *Introductory programming is a foundational course in computing and technical education, yet many novice programmers struggle to understand problems, plan solutions, monitor their coding process, debug errors, and evaluate completed programs. These difficulties are not only technical but also metacognitive, as students often lack structured strategies for regulating their thinking during programming tasks. Self-questioning has been proposed as a metacognitive scaffold that can guide learners through problem-solving; however, less is known about how novice programmers perceive the usability and practicality of such a guideline after using it in programming tasks. This article reports the qualitative findings of a post-intervention survey administered to treatment-group students who used a self-questioning guideline during introductory programming tasks. The study addressed one research question: How do novice programmers perceive and experience the usability and practicality of the self-questioning guideline during programming tasks? A total of 41 survey responses were analysed thematically. The survey examined students' experiences of using the guideline, its useful and difficult parts, its perceived effects before, during, and after coding, its influence on awareness and confidence, its transferability beyond the study, and possible improvements. Findings show that students generally perceived the guideline as practical, usable, and helpful for structuring their thinking. Reported benefits included pre-coding planning, requirement analysis, metacognitive monitoring during coding, systematic debugging, post-task checking, and confidence-building. However, some students found particular programming constructs difficult, experienced higher-order prompts as overwhelming, reported limited change in their usual problem-solving approach, or suggested both more examples and a shorter format. The article concludes that the guideline was*

Volume: 11 Issues: 85 Special Issue [August, 2026] pp. 614 - 632
Journal of Islamic, Social, Economics and Development (JISED)
eISSN: 0128-1755
Journal website: academicinspired.com/jised
DOI: 10.55573/JISED.118544

perceived as a useful metacognitive scaffold, but its practical value depends on careful phasing, examples, adaptive fading, and integration with foundational programming instruction.

Keywords: *Novice programmers, Metacognitive scaffolding, Programming education, Self-regulated learning, Self-questioning*

Introduction

Introductory programming remains a persistent challenge in computing education. The difficulty is not limited to remembering syntax; novice programmers must understand problem requirements, plan a solution, select appropriate constructs, trace program behaviour, locate errors, and evaluate whether the final output satisfies the task. These activities place strong demands on metacognitive regulation because learners must plan, monitor, and evaluate their thinking while managing unfamiliar programming concepts and tools. In this sense, programming is both a technical and regulatory task, as successful programming problem-solving requires learners to regulate their planning, monitoring, and evaluation processes rather than merely produce syntactically correct code (Loksa & Ko, 2016).

The research on which this article is based developed a self-questioning guideline intended to metacognitively scaffold novice programmers during programming tasks. The guideline was designed around the assumption that beginners need more than correct answers or worked examples. They also need prompts that help them ask useful questions at appropriate moments, such as before coding, while writing or modifying code, while debugging, and after completing a task. This approach is aligned with metacognitive theory, which conceptualises learning as involving awareness and regulation of cognition (Flavell, 1979), and with self-regulated learning theory, which positions forethought, performance control, and self-reflection as important phases of learning (Zimmerman, 2002).

This article focuses specifically on the student-perception component of the study. After the intervention and post-test, students in the treatment groups answered an online open-ended survey about their opinions and experiences of the guideline. This component corresponds to the research question: How do novice programmers perceive and experience the usability and practicality of the self-questioning guideline during programming tasks? The focus is therefore not on whether the guideline statistically improved programming performance. Rather, it examines how students experienced the guideline as a learning support tool.

This distinction is important. An instructional tool may be theoretically sound but practically weak if students find it too long, confusing, burdensome, or irrelevant to actual coding. Conversely, positive perceptions alone cannot prove effectiveness. For this reason, this article treats the survey results as evidence of perceived usability, perceived practicality, and perceived metacognitive value, not as conclusive evidence of performance improvement. The aim is to provide a balanced, evidence-based account of what students found helpful, what remained difficult, and what should be refined before wider implementation.

Literature Review

Novice Programming and The Metacognitive Demand of Coding

Learning to program requires more than memorising syntax or reproducing examples. Novice programmers must interpret problem requirements, plan possible solutions, translate those

Volume: 11 Issues: 85 Special Issue [August, 2026] pp. 614 - 632
Journal of Islamic, Social, Economics and Development (JISED)
eISSN: 0128-1755
Journal website: academicinspired.com/jised
DOI: 10.55573/JISED.118544

plans into code, test program behaviour, locate and correct errors, and evaluate whether the final solution satisfies the original task. These activities place substantial cognitive and metacognitive demands on beginners because programming requires learners to coordinate conceptual knowledge, procedural knowledge, strategic decision-making, and self-monitoring during problem-solving. Recent research therefore increasingly treats programming not only as a technical skill, but also as a self-regulated problem-solving activity in which learners must manage their own thinking before, during, and after coding (Loksa et al., 2022; Shin et al., 2023).

A central difficulty for novice programmers is that they often lack sufficient awareness of their own problem-solving processes, including difficulty identifying where they are stuck, judging the adequacy of their solutions, and deciding what strategy to apply next (Prather et al., 2018). They may not recognise when they have misunderstood a problem, selected an unsuitable strategy, or produced code that runs but does not fully meet the requirements. Prather et al. (2024), for example, found that novice programmers working with generative AI tools could complete tasks while still experiencing metacognitive difficulties, including overestimating their own understanding and developing an illusion of competence. This suggests that successful task completion does not necessarily indicate deep understanding or effective self-regulation. For novice programmers, therefore, one metacognitive demand of coding is the need to judge not only whether code executes, but whether the solution is conceptually appropriate, logically correct, and aligned with the task.

Programming also requires continuous planning, monitoring, and evaluation. Before coding, learners must analyse the problem, identify inputs and outputs, decompose the task, and decide which programming constructs are appropriate. During coding, they must monitor whether the code logic matches the planned solution, whether variables are being used correctly, and whether each block of code contributes to the intended outcome. After coding, they must test, debug, and evaluate whether the program works across relevant cases. Loksa et al. (2022) argue that metacognition and self-regulation are important constructs in programming education because they help explain how learners manage these processes. This is particularly relevant for beginners because expert programmers often perform such regulatory behaviours implicitly, whereas novices may not yet know what questions to ask themselves or what aspects of their code to monitor.

The metacognitive demand of coding is also evident in debugging. Debugging is not merely a technical activity of removing errors; it requires learners to diagnose the source of failure, compare expected and actual outputs, generate hypotheses about possible causes, and test corrections systematically. Without metacognitive regulation, novices may rely on trial-and-error changes, copy previous code without understanding it, or stop debugging once the program appears to run. Bubnic et al. (2024) found that metacognitive behaviour was more strongly related to programming performance than metacognitive awareness alone, suggesting that knowing about strategies is insufficient unless learners actively apply regulatory behaviours during problem-solving. This distinction is important because students may report awareness of planning or checking strategies but still fail to use them effectively while coding.

High cognitive load further intensifies the difficulty of novice programming. Beginners must often manage new syntax, unfamiliar programming constructs, problem requirements, compiler feedback, and logical reasoning at the same time. When learners lack strategies for organising these demands, they may become overwhelmed and focus narrowly on producing any working

Volume: 11 Issues: 85 Special Issue [August, 2026] pp. 614 - 632
Journal of Islamic, Social, Economics and Development (JISED)
eISSN: 0128-1755
Journal website: academicinspired.com/jised
DOI: 10.55573/JISED.118544

code rather than understanding the problem or evaluating solution quality. Shin et al. (2023) showed that worked-out examples and metacognitive scaffolding can influence problem-solving programming, cognitive load, and self-regulation. Their findings support the view that novice programmers need structured support that helps them manage both the technical and regulatory aspects of programming tasks.

Recent studies also suggest that metacognitive support should be explicit rather than assumed to develop naturally. Ubaidullah et al. (2021) emphasised the value of combining problem-solving and metacognitive techniques to improve novice students' computational thinking skills. Similarly, Pechorina et al. (2023) proposed Metacodenition, a learning environment designed to scaffold the problem-solving process for novice programmers. These studies indicate that beginners benefit when programming instruction makes thinking processes visible and provides learners with prompts or structures that guide planning, monitoring, and reflection. This is relevant because many introductory programming courses focus heavily on concepts, syntax, and task completion, while giving less explicit attention to how students should regulate their thinking during problem-solving.

Collectively, recent literature shows that novice programming difficulties are not only caused by weak technical knowledge, but also by underdeveloped metacognitive regulation. Beginners may struggle to understand problem requirements, select appropriate strategies, monitor code logic, debug systematically, and evaluate completed solutions. Therefore, learning support in programming should address not only what students know, but also how they regulate their thinking while applying that knowledge.

Metacognition, Self-Regulated Learning, and Scaffolding

Metacognition and self-regulated learning provide an important theoretical basis for understanding why novice programmers need structured support during programming problem-solving. In programming education, metacognition refers to learners' awareness and regulation of their own thinking while interpreting problems, planning solutions, writing code, debugging errors, and evaluating completed programs. Self-regulated learning is closely related because it concerns how learners plan their actions, monitor their progress, adjust strategies, and reflect on outcomes during learning tasks. In introductory programming, these processes are not secondary to coding; they are central to successful problem-solving because students must make repeated decisions about what a problem requires, which programming construct to use, whether their code logic is correct, and how to respond when the program does not behave as expected.

Recent computing education research has emphasised that metacognition and self-regulation should not be treated as general learning skills that automatically transfer into programming. Loksa et al. (2022), in a review of metacognition and self-regulation in programming education, argued that these constructs are important for understanding how students learn to program and how programming instruction can be designed more effectively. Their work is relevant because programming requires learners to regulate both technical and cognitive processes. A student may know the syntax of a loop, function, or array, but still struggle to decide when the construct is appropriate, how to test whether it has been used correctly, or how to revise the solution when an error occurs. Therefore, metacognition and self-regulation help explain why novice programmers may fail even when they have been exposed to the relevant programming concepts.

Volume: 11 Issues: 85 Special Issue [August, 2026] pp. 614 - 632
Journal of Islamic, Social, Economics and Development (JISED)
eISSN: 0128-1755
Journal website: academicinspired.com/jised
DOI: 10.55573/JISED.118544

In programming, self-regulation is visible across the task cycle. Before coding, students plan and interpret requirements; during coding, they monitor logic and strategy use; and after coding, they test, debug, and evaluate the solution. These regulatory processes are difficult for novices to perform independently, which explains the need for scaffolding.

Scaffolding is relevant because novice learners may not yet be able to perform these regulatory processes without support. In programming education, scaffolding can take the form of prompts, worked-out examples, structured problem-solving stages, reflective questions, or learning environments that guide students through the task. For example, Prather et al. (2019) showed the relevance of metacognitive scaffolding at the problem-interpretation stage, where students must first understand what a programming prompt requires before attempting to write code. The purpose of scaffolding is not to remove the difficulty of programming, but to make complex thinking processes more manageable and visible. Pechorina et al. (2023) proposed Metacodenition as a tool that scaffolds the programming problem-solving process for novice programmers. Their work illustrates the value of making the stages of problem-solving explicit so that students can better understand where they are in the process and what they should do next. This is important because novices often experience programming as an unstructured activity of trying code until something works, whereas scaffolding can guide them towards a more deliberate process of understanding, planning, implementing, testing, and evaluating.

Metacognitive scaffolding is particularly useful because it supports both learning process and problem-solving behaviour. Shin et al. (2023) examined worked-out examples and metacognitive scaffolding in programming problem-solving and found that metacognitive scaffolding, especially when combined with faded worked-out examples, supported problem-solving programming and self-regulation skills. This suggests that scaffolding should not remain static. As students gain experience, support can be gradually reduced or adapted so that learners increasingly take responsibility for planning, monitoring, and evaluating their own work. This is relevant to the design of self-questioning guidelines because such guidelines can initially provide explicit prompts, but students may later internalise, skip, or adapt prompts once the regulatory process becomes more familiar.

Recent research also indicates that metacognitive support should be integrated with problem-solving instruction rather than treated as a separate activity. Ubaidullah et al. (2021) argued that problem-solving and metacognitive techniques each contribute to the development of computational thinking skills, but either technique alone may be insufficient. This point is important for programming education because students need both technical knowledge and regulatory strategies. Metacognitive prompts may help students ask better questions about their work, but they cannot replace instruction in programming concepts, syntax, logic, and debugging practices. Similarly, teaching programming concepts without helping students regulate their thinking may leave novices dependent on memorisation, imitation, or trial and error. A combined approach is therefore needed, where programming instruction develops conceptual and procedural knowledge while scaffolding helps students manage their thinking during problem-solving.

In this article, metacognitive scaffolding provides the theoretical basis for the self-questioning guideline. The guideline is treated as a learning scaffold because it makes regulatory processes explicit through questions that students can apply before, during, and after coding. However, such scaffolding must be usable, task-specific, and adaptable; otherwise, students may perceive it as burdensome or disconnected from authentic coding practice.

Volume: 11 Issues: 85 Special Issue [August, 2026] pp. 614 - 632
Journal of Islamic, Social, Economics and Development (JISED)
eISSN: 0128-1755
Journal website: academicinspired.com/jised
DOI: 10.55573/JISED.118544

Self-questioning as a Metacognitive Scaffold

Self-questioning is a prompt-based metacognitive strategy in which learners ask themselves purposeful questions to guide comprehension, planning, monitoring, debugging, and evaluation. In programming, it is especially relevant because novice programmers must regulate their thinking while interpreting a problem, selecting constructs, writing code, checking logic, and evaluating output. This positioning is consistent with research that treats programming as a self-regulated problem-solving activity, where learners need to manage their own cognitive and regulatory processes during complex tasks (Loksa et al., 2022; Xie et al., 2023). Self-questioning therefore acts as a metacognitive scaffold by making these otherwise implicit regulatory processes visible and actionable.

In a programming task, self-questioning can be aligned with the main phases of problem-solving. Before coding, students may ask what the problem requires, what the inputs and outputs are, and which construct is most suitable. During coding, they may ask whether each statement contributes to the intended logic and whether variables, conditions, loops, arrays, or functions are being used appropriately. During debugging, they may ask where the error begins, what type of error is involved, and whether the correction addresses the actual cause of the problem. After coding, they may ask whether the output matches the requirements and whether the solution can be improved. This phased use of prompts is consistent with studies showing the value of explicit guidance, metacognitive scaffolding, and structured problem-solving support in programming education (Loksa et al., 2016; Pechorina et al., 2023; Prather et al., 2019; Shin et al., 2023).

Loksa et al. (2022) emphasised that metacognition and self-regulation are important constructs in programming education because learners must manage their own cognitive processes during complex problem-solving. Xie et al. (2023) further identified self-questioning during programming as a form of self-instruction used in the performance phase of self-regulation. These studies support the view that self-questioning is not merely a post-task reflection strategy. It can also guide real-time monitoring while students are writing, modifying, and checking code.

The broader literature on programming scaffolds also supports this rationale. Pechorina et al. (2023) proposed Metacodenition as a way to scaffold the programming problem-solving process by making its stages more explicit for novices. Although Metacodenition is not identical to a self-questioning guideline, both approaches share the assumption that beginners benefit when programming is structured as a deliberate process rather than as trial-and-error coding. Similarly, Shin et al. (2023) and Choi et al. (2023) showed that metacognitive scaffolding and reflection prompts can support programming-related learning when they are integrated into the learning process. For the present study, this supports the design decision to embed self-questioning prompts at meaningful points in the programming task rather than treating reflection as an activity only after coding.

However, self-questioning should not be treated as a substitute for programming knowledge. Ubaidullah et al. (2021) argued that problem-solving and metacognitive techniques need to be integrated because either approach alone may be insufficient for developing computational thinking. This boundary is important for the present study because a prompt can help students notice that they do not understand a function, array, loop, or condition, but it cannot by itself teach the concept. In addition, because novice programming is already cognitively demanding, poorly understood or overly complex prompts may add to students' load rather than reduce it

Volume: 11 Issues: 85 Special Issue [August, 2026] pp. 614 - 632
Journal of Islamic, Social, Economics and Development (JISED)
eISSN: 0128-1755
Journal website: academicinspired.com/jised
DOI: 10.55573/JISED.118544

(Shin et al., 2023). Therefore, self-questioning is best understood as a scaffold that complements explicit instruction, worked examples, guided practice, and lecturer support.

Self-questioning is also relevant in current programming environments where students may receive external support from automated tools or generative AI. Prather et al. (2024) found that novice programmers may experience an illusion of competence when code is produced without sufficient understanding. Self-questioning can help counter this risk by prompting learners to justify code, check alignment with task requirements, and evaluate whether they understand the solution.

Overall, the literature supports self-questioning as a metacognitive scaffold for novice programming. Its contribution is not that it teaches syntax directly, but that it helps students regulate how they apply programming knowledge during problem-solving. Its usefulness, however, depends on careful design. Prompts need to be clear, task-specific, appropriately timed, supported by examples, and gradually faded as students internalise the questioning process. This provides the rationale for examining how novice programmers perceive the usability and practicality of a self-questioning guideline after using it during introductory programming tasks.

Practicality and User Acceptance of Learning Scaffolds

The practicality of a learning scaffold is important because a theoretically sound intervention may have limited value if students cannot use it meaningfully during authentic learning tasks. In introductory programming, this issue is particularly relevant because novice programmers already face high cognitive demands when interpreting problems, selecting constructs, writing code, debugging errors, and evaluating solutions. A scaffold that adds too much complexity may increase rather than reduce learner burden. Therefore, the usefulness of a metacognitive scaffold should not be judged only by its theoretical alignment with self-regulated learning, but also by whether students perceive it as understandable, manageable, relevant, and usable while solving programming tasks.

Recent work in programming education suggests that scaffolds are most useful when they are embedded in the problem-solving process rather than treated as separate learning activities. Loksa et al. (2022) emphasised that metacognition and self-regulation are central to programming education because learners must plan, monitor, and evaluate their thinking while solving programming problems. From a practicality perspective, this implies that scaffolds should support the actual phases of programming activity, including problem interpretation, coding, debugging, and post-task evaluation. If prompts or guidelines are too general, students may not know how to apply them to specific programming situations. Conversely, if they are too detailed or lengthy, students may skip them or treat them as an additional burden.

The design of practical scaffolds also requires attention to cognitive load. Shin et al. (2023) examined worked-out examples and metacognitive scaffolding in programming problem-solving and showed that such scaffolding is connected to learners' knowledge performance, cognitive load, and self-regulation skills. This is relevant because students' acceptance of a scaffold may depend partly on whether it reduces confusion and organises thinking, rather than making the task feel more complicated. For novice programmers, a practical scaffold should therefore simplify the regulation of problem-solving by making key thinking steps visible, such as asking what the problem requires, whether the selected construct is appropriate, where an error begins, and whether the output satisfies the task.

Volume: 11 Issues: 85 Special Issue [August, 2026] pp. 614 - 632
Journal of Islamic, Social, Economics and Development (JISED)
eISSN: 0128-1755
Journal website: academicinspired.com/jised
DOI: 10.55573/JISED.118544

User acceptance is also related to whether students perceive the scaffold as useful during real task performance. Choi et al. (2023) found that reflection prompts combined with hints in an online programming course affected learning outcomes, including perceived learning and enjoyment. Although reflection prompts are not identical to a self-questioning guideline, the study is relevant because it shows that prompt-based support can be examined not only in terms of performance, but also in terms of students' learning experience. This distinction is important for the present article because students' perceptions provide evidence of perceived usefulness and acceptability, but they should not be treated as direct proof of actual effectiveness.

Practicality also depends on whether a scaffold can be integrated into students' normal problem-solving routines. This is important because novice programmers' reflections occur within the flow of programming activity, and scaffolds that support such in-situ reflection may help students become more aware of their own process while coding (Loksa et al., 2020). Pechorina et al. (2023) proposed Metacodenition as a tool to scaffold the problem-solving process for novice programmers. Their work illustrates the importance of making the stages of programming problem-solving explicit for beginners. However, any scaffold that structures learners' thinking must still be usable in the flow of coding. If students perceive a scaffold as interrupting progress, repeating obvious steps, or failing to match the task, they may abandon it. Therefore, examining students' experience of a self-questioning guideline is necessary to understand whether the scaffold is not only conceptually appropriate, but also practically adoptable.

In current programming learning environments, the issue of acceptance is further complicated by the availability of automated and generative tools. Prather et al. (2024) found that novice programmers using generative AI may still experience metacognitive difficulties and may develop an illusion of competence when they produce code without adequate understanding. This reinforces the need for scaffolds that encourage learners to remain cognitively active and evaluative. However, such scaffolds must be perceived as helpful rather than punitive or remedial. Students are more likely to accept a guideline if it is framed as a professional problem-solving support that helps them reason about code, rather than as a sign that they are weak programmers.

For a self-questioning guideline, practicality and user acceptance can therefore be understood through several dimensions; clarity, relevance, timing, workload, perceived usefulness, adaptability, and transferability. Clarity concerns whether students understand the questions. Relevance concerns whether the prompts match the programming task. Timing concerns whether the questions are placed at appropriate stages, such as before coding, during coding, debugging, and after coding. Workload concerns whether the guideline supports thinking without becoming too long or repetitive. Perceived usefulness concerns whether students believe the guideline helps them plan, monitor, debug, or evaluate more effectively. Adaptability concerns whether students can internalise, skip, or modify prompts as they become more confident. Transferability concerns whether they see the questioning strategy as useful beyond the intervention.

Accordingly, this article treats students' perceptions as an important but bounded form of evidence. Positive perceptions indicate perceived usefulness and acceptability, while negative or mixed perceptions identify design limitations. However, perceived practicality should not be confused with actual effectiveness. Students may feel more confident or organised without necessarily producing more correct code. Within this scope, examining practicality and user

Volume: 11 Issues: 85 Special Issue [August, 2026] pp. 614 - 632
Journal of Islamic, Social, Economics and Development (JISED)
eISSN: 0128-1755
Journal website: academicinspired.com/jised
DOI: 10.55573/JISED.118544

acceptance is necessary because a metacognitive scaffold can only support learning if students are willing and able to use it during programming problem-solving.

Research Gap and Rationale for the Study

Recent programming education research has established that novice programmers face substantial metacognitive demands and may struggle to plan, monitor, debug, and evaluate their solutions independently. Studies have also shown that self-regulation, explicit guidance, metacognitive scaffolding, reflection prompts, and structured problem-solving supports are relevant to programming learning. However, much of this work focuses on learners' regulation, task performance, or scaffold design, rather than on how novice programmers experience the usability and practicality of a structured self-questioning guideline after using it across programming tasks.

This gap matters because metacognitive support can remain too abstract if it is presented only as general advice, such as "plan before coding" or "check your work". Novice programmers need concrete and task-specific guidance that translates metacognitive regulation into actions they can apply before coding, during coding, while debugging, and after completing a solution. Self-questioning provides such a mechanism by turning planning, monitoring, debugging, and evaluation into explicit prompts. Nevertheless, a scaffold that is too long, too abstract, or poorly aligned with students' programming difficulties may be ignored or used superficially, even if it is theoretically sound.

The present study addresses this gap by examining novice programmers' perceptions of the usability and practicality of a self-questioning guideline used during introductory programming tasks. By focusing on students' experiences, the study contributes evidence on whether self-questioning is adoptable as a metacognitive scaffold in programming education and how such a guideline may be refined for future teaching practice.

Methodology

Research Design and Focus of the Article

This article employed a qualitative descriptive design based on open-ended survey responses. The data reported in this article were drawn from a broader mixed-methods study that developed and evaluated a self-questioning guideline for novice programmers. However, the present article focuses only on the qualitative post-intervention survey component, which was administered to treatment-group students after they had used the guideline during a series of introductory programming tasks and completed the post-test. The purpose of this qualitative component was to examine students' perceptions of the guideline's usability and practicality during programming problem-solving. The quantitative strand of the broader study, which examined metacognitive awareness and programming performance, is reported separately and is not the focus of this article. Therefore, the findings reported in this article should be interpreted as qualitative evidence of students' perceived usability and practicality, rather than as the full mixed-methods evaluation of the guideline.

Participants and Context

Participants were novice undergraduate programmers enrolled in introductory programming. The survey was administered to students from the treatment groups after they had experienced the self-questioning guideline during a series of programming tasks and completed the post-test. 41 responses were included in the dataset.

Volume: 11 Issues: 85 Special Issue [August, 2026] pp. 614 - 632
Journal of Islamic, Social, Economics and Development (JISED)
eISSN: 0128-1755
Journal website: academicinspired.com/jised
DOI: 10.55573/JISED.118544

The intervention involved a self-questioning guideline used during introductory programming tasks. The guideline prompted students to ask questions related to understanding the problem, planning, coding, checking, debugging, and reviewing their solutions. The overall aim was to externalise metacognitive regulation so that students could practise asking productive questions before, during, and after coding.

Instrument

The survey was an open-ended instrument administered bilingually in English and Malay to support students' comprehension and designed to gather feedback about the self-questioning guideline used during programming tasks. The instrument was piloted with three students not involved in the experiment to check the clarity, wording, and suitability of the open-ended questions. The pilot responses were not included in the thematic analysis. Students were instructed to answer honestly based on their experience, with no right or wrong answers, and to provide details and examples where relevant. The survey contained 14 open-ended questions grouped into perceptions of the guideline, impact on problem-solving and metacognition, and suggestions for improvement (Table 1). A standardised quantitative usability scale was not used because the purpose of this qualitative strand was not to produce a usability score, but to explore students' perceived practicality, usability, difficulties, adaptations, and suggestions after using the guideline during programming tasks.

Table 1: Components in Instrument

Section	Focus	Items
A	Perception of the guideline	Experience using the guideline; useful parts; difficult parts; perceived support for learning concepts
B	Impact on problem-solving and metacognition	Understanding problems before coding; thinking while coding; finding/fixing errors; reviewing solutions; awareness of thought processes; confidence; comparison with usual approach; usefulness beyond the study
C	Suggestions for improvement	Suggested improvements; recommendation to other beginner programmers

Data analysis

The open-ended survey responses were analysed thematically. Before coding, the bilingual responses were organised for analysis. Coding was performed on the translated English text to ensure consistency across responses, while original-language responses were retained for checking meaning where required. The analysis involved familiarisation with responses, initial coding, development of recurring themes, review of themes, and definition of themes with illustrative participant quotations. Frequencies were used to indicate how often a theme appeared in the coded responses. These frequencies are useful for identifying dominant patterns, but they should not be interpreted as inferential statistics. Since one response could contain more than one idea, thematic counts may overlap. In total, 874 code instances were assigned across the 41 participants and 14 survey items, reflecting the fact that many responses contained multiple ideas and could be coded under more than one theme.

To strengthen analytic rigour, ambiguous or flagged responses were manually reviewed before final coding. These referred to responses that were unclear, contained multiple meanings, included negative or contradictory views, or required checking against the original-language

Volume: 11 Issues: 85 Special Issue [August, 2026] pp. 614 - 632
Journal of Islamic, Social, Economics and Development (JISED)
eISSN: 0128-1755
Journal website: academicinspired.com/jised
DOI: 10.55573/JISED.118544

wording. The audit trail documented 29 manual-review flags, which were resolved through checking response meaning, coding consistency, and fit with the emerging themes. A formal negative-case analysis was also conducted to identify responses that contradicted, qualified, or complicated the dominant positive patterns. Accordingly, the results reported here include both dominant positive themes and divergent or negative cases. This is important because a purely positive summary would overstate the guideline's practicality. The analysis therefore reports cases where students found parts difficult, felt overwhelmed, perceived little change, were unsure about transferability, or suggested simplification.

Results and Discussion

The findings (Table 2) are organised into five interpretive themes; adaptive use and usability, support for programming concepts and task phases, metacognitive awareness and confidence, transferability and recommendation, and areas for refinement. These themes synthesise the 14 survey questions while preserving item-level evidence.

Table 2: Summary of Findings

Main finding	Supporting evidence from the survey analysis	Interpretation
Adaptive use rather than mechanical use	Most participants modified the guideline over time (n=28), usually by skipping redundant questions or changing order; some moved from strict adherence to more fluent use (n=8).	Students did not merely follow the tool passively; many personalised it as they became familiar with it.
Useful for concrete constructs and checking	Useful parts included programming constructs such as functions (n=18), metacognitive checking and verification (n=17), planning and comprehension (n=6), and examples (n=4).	The guideline served both as a conceptual reference and as a metacognitive checklist.
Generally usable, but not difficulty-free	Most reported no difficulty (n=25), but challenges remained for functions, arrays, and recursion (n=5), advanced metacognitive questions (n=3), and clarity/detail issues (n=4).	Usability was high, yet beginners still needed clearer examples and topic-specific support.
Changed the programming process	Students reported stronger pre-coding analysis (n=18), planning (n=16), monitoring during coding (n=19), systematic debugging (n=15), and post-task verification (n=18).	The guideline appears to have structured the full before-during-after cycle of programming.
Raised awareness and confidence for most	Awareness increased for most participants (n=39); confidence was supported by structure (n=19), learning/internalisation (n=10), and verification/error reduction (n=8).	Students perceived the guideline as supporting self-regulation and self-efficacy.
Refinement requires balancing detail and brevity	Some wanted no change (n=13), some wanted examples/supporting materials (n=13), some wanted more or clearer content (n=12), and some wanted simplification (n=3).	The next version should be adaptive; concise core prompts with optional examples and extensions.

Volume: 11 Issues: 85 Special Issue [August, 2026] pp. 614 - 632
Journal of Islamic, Social, Economics and Development (JISED)
eISSN: 0128-1755
Journal website: academicinspired.com/jised
DOI: 10.55573/JISED.118544

Adaptive Use and Perceived Usability

A central finding is that many students did not use the guideline in a fixed or mechanical way. Instead, the majority reported some form of adaptation (n=28). The most common modifications were skipping questions that were perceived as redundant or already mastered and changing the order of questions to fit the task or personal workflow. This suggests that students gradually moved from dependence on an external scaffold towards selective, strategic use.

This movement is consistent with the idea of scaffolding as temporary support. Some students initially relied on the guideline because they lacked confidence or needed structure, but later used it more fluently. P12 reported using it strictly when 'not very confident', while P26 provided a more nuanced example of adjustment over time. In response to the influence of the guideline while coding, P26 stated that it 'caused doubt and confusion at the beginning', suggesting that the prompts initially made the coding process feel more effortful. However, P26 also described moving beyond initial 'self-doubt', indicating that the guideline became more manageable as familiarity increased. This before-and-after pattern shows that metacognitive scaffolding may not be experienced as immediately easy by all novices. For some students, the guideline may first expose uncertainty before supporting more fluent and strategic use.

The usability results were strongly positive but not perfect. A significant majority reported no difficulty in understanding or applying the guideline (n=25), describing it as 'user-friendly' (P4), 'straight forward' (P32), and 'arranged and made neatly' (P7). These responses suggest that the final form of the guideline was broadly accessible to the treatment-group students. However, several participants still experienced difficulty with specific constructs such as functions, arrays, and recursion (n=5), and a smaller group found advanced questions about optimisation, assumptions, or better solutions overwhelming (n=3). P12's comment that thinking about 'better ways' was 'overwhelming at times' is important because it shows that higher-order prompts may exceed some novices' immediate cognitive capacity when they are still trying to make the code work.

The negative cases provide an important qualification to the generally positive findings. Although most students perceived the guideline as useful, several responses suggest that its usefulness was conditional rather than universal. Some students reported that particular programming constructs, such as functions and arrays, remained difficult despite the guideline, while others experienced higher-order prompts as overwhelming when they were still focused on producing working code. These responses suggest that the guideline's perceived usefulness depended partly on students' existing conceptual and procedural knowledge. For students who lacked confidence with core constructs, self-questioning may have helped them become more aware of what they did not understand, but this awareness did not automatically resolve the technical difficulty. Therefore, the guideline should be understood as a metacognitive scaffold that supports planning, monitoring, debugging, and reflection, rather than as a substitute for explicit instruction, worked examples, practice, modelling, and lecturer support.

These findings indicate perceived usefulness rather than direct evidence of performance effectiveness. The survey data show how students experienced the guideline, while claims about actual improvement require evidence from programming performance, code quality, observation, or other behavioural data.

Volume: 11 Issues: 85 Special Issue [August, 2026] pp. 614 - 632
Journal of Islamic, Social, Economics and Development (JISED)
eISSN: 0128-1755
Journal website: academicinspired.com/jised
DOI: 10.55573/JISED.118544

Support for Programming Concepts and Task Phases

Students perceived the guideline as useful at two levels; as support for specific programming constructs and as support for thinking processes. The most frequently reported useful component was scaffolding for specific programming constructs (n=18). P23 stated that 'The function part helped me a lot to learn function', while P32 said it was useful because 'I am weak in that part'. These comments show that the guideline was not experienced only as an abstract metacognitive tool. It was also used as a practical reference for difficult topics.

The second major useful component was metacognitive checking and verification (n=17). Students valued prompts that helped them check whether their code worked correctly and whether they had missed important details. P13 said the guideline 'helped me ensure the program runs correctly', while P24 valued it because it 'gave me new questions that I had not thought of before'. This finding is important because novices often finish coding once the program appears to run, without systematically checking logic, requirements, or edge cases. The guideline appears to have encouraged a more evaluative stance.

Before coding, students reported stronger problem analysis and requirement identification. Enhanced focus on problem analysis and requirements was the most frequent pre-coding theme (n=18), followed by pre-coding planning and systematic approach (n=16). P11 described moving from 'jump straight into coding' to analysing input, output, and goals. P12 said the guideline helped them 'break the problem into smaller parts'. These responses show that the guideline addressed one common novice tendency; beginning to code before fully understanding the problem.

During coding, the main reported influence was active metacognitive monitoring and real-time evaluation (n=19). Students described using the prompts to check logic, evaluate whether a line of code was needed, and monitor the purpose of a function. P12 reported thinking about why a line of code should exist rather than merely trying what might work. The guideline also acted as a practical scaffold and reference (n=12), making the coding process 'easier' (P6) and helping students remember details. This indicates that the tool supported both cognitive action and metacognitive control.

For debugging, the guideline encouraged a more systematic and structured approach (n=15). P11 described approaching errors 'step by step', while P33 used the questions to locate faults in program logic. Some students used it as a reference and verification tool (n=9); P8 said it 'helped me realise my mistakes', and P25 used it to resolve doubts about 'coding format'. However, not all students experienced a change in debugging. P9 answered 'No', and P16 stated that it did not change the fundamental troubleshooting method but 'definitely makes us more aware'. This distinction is crucial. The guideline may improve awareness of debugging, but awareness does not automatically guarantee a different or more effective debugging strategy.

After completing tasks, the guideline supported systematic review and verification (n=18) and reflective practice or quality improvement (n=12). Students used checking sections to verify code against requirements and syntax, and some moved beyond correctness to consider whether the solution was understandable, efficient, or a 'best way' solution. P11's reflection on whether the solution was the 'best way' indicates a shift from basic completion to quality-oriented review. Yet one divergent case, P19, reported, 'I did not feel proud because I used help.' This reminds educators that scaffolding can sometimes affect learners' sense of autonomy. Practical

Volume: 11 Issues: 85 Special Issue [August, 2026] pp. 614 - 632
Journal of Islamic, Social, Economics and Development (JISED)
eISSN: 0128-1755
Journal website: academicinspired.com/jised
DOI: 10.55573/JISED.118544

implementation should therefore frame the guideline not as a crutch, but as a professional problem-solving strategy to be internalised over time.

Metacognitive Awareness, Confidence, and Self-Efficacy

The survey provides strong evidence that students perceived the guideline as increasing awareness of their own thinking. Most participants reported increased awareness (n=39), with many offering general affirmation and some identifying specific changes in their thinking patterns. P11 said the guideline made them aware of 'patterns in my thinking', while P12 said it made them 'more self-aware as a learner'. P25 stated that it made them think about 'how I solve problems, not just the final answer'. These responses align directly with the intended metacognitive function of the guideline.

Confidence was also affected positively for most respondents. The most common source of confidence was structured guidance (n=19). Students valued having a clear checklist that reduced the feeling of being lost. P11 referred to the 'clear checklist' as helping them move forward when stuck, and P31 described a shift from doubt to trusting their process. Confidence also came from learning and internalisation (n=10), as students felt they were acquiring correct practices and remembering what to check. P9 linked the guideline to 'reducing feelings of confusion or fear', while P10 connected confidence to the ability to 'learn again and remember'. Verification and error reduction (n=8) further strengthened confidence because students could check their answers more reliably.

These findings are encouraging, but they must be interpreted within the limits of self-report data. Students' confidence and perceived awareness are valuable because they suggest that the guideline was meaningful and acceptable to learners. However, perceived awareness is not the same as actual behavioural regulation or improved programming performance. The broader study found a more complex relationship between metacognitive awareness and programming performance, suggesting a possible knowing-doing gap; students may become more aware of planning, monitoring, and checking strategies without consistently applying those strategies in ways that improve code correctness. This interpretation is consistent with Bubnic et al. (2024), who found that metacognitive behaviour predicted programming performance more strongly than metacognitive awareness alone. Therefore, the present findings should be read as evidence of perceived usefulness and perceived metacognitive value, while claims about actual effectiveness require supporting evidence from performance scores, code quality, observation, or other behavioural data.

Comparison With Usual Practice, Transferability, and Recommendation

When comparing the guideline experience with their usual approach, students mainly described a shift towards a structured and intentional process (n=16). P12 summarised this shift as moving from 'start coding immediately' to 'plan more, think first, and code second', which was experienced as less stressful. Participants also described enhanced metacognitive awareness and deliberate practice (n=10), improved confidence, accuracy, and efficiency (n=9), and the guideline as a valuable external aid (n=8). P23 contrasted the new approach with previous 'feeling-based' methods, suggesting that the guideline made problem-solving more explicit and analytical.

The transferability results were broadly positive. The most frequent suggested application was educational contexts and further learning (n=22), including programming classes, exams, projects, and other subjects such as mathematics and science. Professional and practical

Volume: 11 Issues: 85 Special Issue [August, 2026] pp. 614 - 632
Journal of Islamic, Social, Economics and Development (JISED)
eISSN: 0128-1755
Journal website: academicinspired.com/jised
DOI: 10.55573/JISED.118544

applications were also common (n=13), including real-world coding jobs, internships, collaborative projects, code reviews, and technical interviews. P7 linked the guideline to use 'when I am working', while P12 saw value in 'debugging in real work situations'. Some students also saw it as useful for general problem-solving and life skills (n=9), with P10 referring to 'daily problems' and P38 stating it could be useful 'everywhere'. Nonetheless, transferability was not universal. Four participants were sceptical or unsure, including P8 and P14 who answered 'No', and P28 who was 'Not sure'. This indicates that not all novices recognised the broader value of self-questioning beyond the immediate study context.

The recommendation item was highly positive, with all responses expressing willingness to recommend the guideline to other beginner programmers. However, this should be interpreted as evidence of perceived acceptability rather than proof of actual effectiveness. Because the survey was administered immediately after the post-test and involved only treatment-group students, the responses may have been influenced by recent experience, perceived obligation, or social desirability. Nevertheless, the consistency of the affirmative responses suggests that students generally regarded the guideline as a useful and acceptable learning support. Reasons included skill development and habit formation (n=14), practical utility and ease of use (n=13), error prevention and code quality (n=6), and suitability for novices (n=6). P11 said it helps beginners 'develop a clear and structured way of thinking', P3 said it helps them become 'more careful', and P1 described programming as 'much easier' with the guideline. This pattern reinforces the finding that students generally perceived the guideline as acceptable and practically useful, while still requiring cautious interpretation because the data are self-reported and post-intervention.

Areas for Refinement

Students' suggestions for improvement are especially useful because they show that practicality does not mean the guideline should remain unchanged. A substantial group said no changes were needed (n=13), using terms such as 'perfect' (P6) and 'sufficient' (P33). However, the same number requested expanded examples and supporting materials (n=13). P11 specifically asked for 'simple examples or sample answers for each question'. This suggests that beginners may understand a prompt better when it is paired with a concrete programming example.

Another group wanted content expansion and clarification (n=12), including more questions or deeper support for specific topics such as functions and arrays. P2 specifically suggested adding questions 'in the function and array sections'. Others wanted structural improvements (n=6), such as grouping questions into clear phases like 'Before/During/After Coding', as suggested by P12. These suggestions are consistent with the guideline's metacognitive purpose because phase-based organisation maps onto planning, monitoring, and evaluation.

At the same time, a smaller group requested simplification and reduction (n=3). P19 suggested removing 'less insignificant questions' to prevent boredom. This creates a practical design tension; some novices want more guidance, while others want a shorter and less repetitive tool. However, this request was less frequent than the demand for additional support, as more students asked for examples and supporting materials (n = 13) or clearer and expanded content (n = 12). This shows that the main refinement needed was not simply to shorten the guideline, but to balance sufficient guidance with manageable length. The data therefore support an adaptive scaffolding approach. A concise core checklist should be provided for all students, while optional examples, explanations, and topic-specific extensions can be made available for

Volume: 11 Issues: 85 Special Issue [August, 2026] pp. 614 - 632
Journal of Islamic, Social, Economics and Development (JISED)
eISSN: 0128-1755
Journal website: academicinspired.com/jised
DOI: 10.55573/JISED.118544

students who need additional support. Over time, students should be encouraged to fade prompts they have internalised and retain prompts for areas where they still struggle.

Conclusion

This article examined novice programmers' perceptions of a self-questioning guideline after its use in introductory programming tasks. The findings indicate that students generally perceived the guideline as usable, practical, and helpful for structuring their programming problem-solving. It supported planning before coding, monitoring during coding, debugging, reviewing completed solutions, awareness of thought processes, and confidence in problem-solving. Many students adapted the guideline over time, suggesting that it functioned not only as a fixed worksheet but as a scaffold that could become part of students' personal problem-solving routines.

At the same time, the evidence should not be overstated. The findings demonstrate perceived usefulness and perceived practicality, not definitive performance effectiveness. Some students experienced difficulty with specific programming constructs, some found advanced prompts overwhelming, and a few reported limited change in their usual approach or uncertainty about transferability. Improvement suggestions also showed a clear tension between the need for more examples and the risk of making the guideline too long.

The main contribution of the findings is therefore practical and pedagogical. The self-questioning guideline appears to be acceptable to novice programmers and capable of making metacognitive regulation more visible during programming tasks. For future implementation, the guideline should be refined into a phased and adaptive format; a concise core set of before-during-after prompts, optional examples, topic-specific supports for difficult constructs, and guidance for fading prompts as students gain fluency. Used in this way, self-questioning can become a learning innovation that supports learner-centred programming education by helping beginners regulate their thinking rather than leaving them to rely mainly on trial and error.

The findings suggest several implications for teaching practice. First, self-questioning should be introduced explicitly before students are expected to use the guideline independently, so that they understand its purpose and how it supports programming problem-solving. The prompts should also be organised according to the main phases of programming, namely before coding, during coding, debugging, and after coding, to help students apply them at appropriate points in the task. In addition, the prompts should be paired with short examples, particularly for functions, arrays, and other constructs that students commonly find difficult. To encourage positive acceptance, the guideline should be framed as a professional problem-solving strategy rather than as remedial support or a sign of weakness. Finally, lecturers should encourage gradual fading, whereby students may skip prompts they have already internalised while continuing to use prompts that address their persistent weaknesses.

Overall, the findings suggest that programming problem-solving among novice programmers is influenced by several interacting factors, including the ability to analyse task requirements before coding, plan an appropriate solution, monitor logic and syntax during coding, debug errors systematically, evaluate completed solutions, and maintain confidence when facing difficulty. In this study, students perceived the self-questioning guideline as useful because it supported several of these factors, particularly planning, monitoring, debugging, and post-task checking. However, the findings also indicate that metacognitive prompts alone are insufficient when students lack foundational understanding of specific programming constructs, suggesting

Volume: 11 Issues: 85 Special Issue [August, 2026] pp. 614 - 632
Journal of Islamic, Social, Economics and Development (JISED)
eISSN: 0128-1755
Journal website: academicinspired.com/jised
DOI: 10.55573/JISED.118544

that self-questioning should complement, rather than replace, explicit instruction in programming concepts.

This study has several limitations. The evidence is based on open-ended self-report responses from treatment-group students, meaning that it captures students' perceptions rather than direct behavioural observation during every stage of coding. Because the survey was administered after the intervention, the responses may also have been influenced by recent experience, memory, social desirability, or students' feelings about their post-test performance. In addition, the thematic frequencies reported in the analysis are descriptive rather than inferential, and should not be treated as evidence of causal impact. Finally, the findings are most applicable to introductory programming contexts similar to the study setting and should be further tested in other institutions, programming languages, and course formats.

Future studies should examine how students use the guideline in real time through screen recordings, think-aloud protocols, or log data. Further work should also compare paper-based and digital versions of the guideline, test adaptive versions that fade prompts over time, and investigate whether perceived improvements in planning, monitoring, debugging, and review translate into stronger programming performance across multiple tasks and course contexts.

Acknowledgements
The authors would like to acknowledge and extend special gratitude to the Centre for Diploma Studies, Universiti Tun Hussein Onn Malaysia who approved the data collection for this research.

Volume: 11 Issues: 85 Special Issue [August, 2026] pp. 614 - 632
Journal of Islamic, Social, Economics and Development (JISED)
eISSN: 0128-1755
Journal website: academicinspired.com/jised
DOI: 10.55573/JISED.118544

References

Bubnic, B., Kovačević, Ž., & Kosar, T. (2024). Can metacognition predict your success in solving problems? An exploratory case study in programming. In *Proceedings of the 24th Koli Calling International Conference on Computing Education Research*. New York, NY: Association for Computing Machinery. doi:10.1145/3699538.3699593

Choi, H., Jovanovic, J., Poquet, O., Brooks, C., Joksimović, S., & Williams, J. J. (2023). The benefit of reflection prompts for encouraging learning with hints in an online programming course. *The Internet and Higher Education, 58*, 100903. doi:10.1016/j.iheduc.2023.100903

Flavell, J. H. (1979). Metacognition and cognitive monitoring: A new area of cognitive developmental inquiry. *American Psychologist, 34*(10), 906–911.

Loksa, D., & Ko, A. J. (2016). The role of self-regulation in programming problem-solving process and success. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (pp. 83–91). New York, NY: Association for Computing Machinery. doi:10.1145/2960310.2960334

Loksa, D., Ko, A. J., Jernigan, W., Oleson, A., Mendez, C. J., & Burnett, M. M. (2016). Programming, problem solving, and self-awareness: Effects of explicit guidance. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (pp. 1449–1461). New York, NY: Association for Computing Machinery. doi:10.1145/2858036.2858252

Loksa, D., Margulieux, L., Becker, B. A., Craig, M., Denny, P., Pettit, R., & Prather, J. (2022). Metacognition and self-regulation in programming education: Theories and exemplars of use. *ACM Transactions on Computing Education, 22*(4), Article 39. doi:10.1145/3487050

Loksa, D., Xie, B., Kwik, H., & Ko, A. J. (2020). Investigating novices' in situ reflections on their programming process. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (pp. 149–155). New York, NY: Association for Computing Machinery. doi:10.1145/3328778.3366823

Pechorina, Y., Anderson, K., & Denny, P. (2023). Metacodenition: Scaffolding the problem-solving process for novice programmers. In *Proceedings of the 25th Australasian Computing Education Conference* (pp. 59–68). New York, NY: Association for Computing Machinery. doi:10.1145/3576123.3576130

Prather, J., Pettit, R., Becker, B. A., Denny, P., Loksa, D., Peters, A., Albrecht, Z., & Masci, K. (2019). First things first: Providing metacognitive scaffolding for interpreting problem prompts. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (pp. 531–537). New York, NY: Association for Computing Machinery. doi:10.1145/3287324.3287374

Prather, J., Pettit, R., McMurry, K., Peters, A., Homer, J., & Cohen, M. (2018). Metacognitive difficulties faced by novice programmers in automated assessment tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* (pp. 41–50). New York, NY: Association for Computing Machinery. doi:10.1145/3230977.3230981

Prather, J., Reeves, B. N., Leinonen, J., MacNeil, S., Randrianasolo, A. S., Becker, B. A., . . . Briggs, B. (2024). The widening gap: The benefits and harms of generative AI for novice programmers. In *ICER 2024: ACM Conference on International Computing Education Research* (pp. 469–486). New York, NY: Association for Computing Machinery. doi:10.1145/3632620.3671116

Shin, Y., Jung, J., Zumbach, J., & Yi, E. (2023). The effects of worked-out example and metacognitive scaffolding on problem-solving programming. *Journal of Educational Computing Research, 61*(6), 1312–1331. doi:10.1177/07356331231174454

Volume: 11 Issues: 85 Special Issue [August, 2026] pp. 614 - 632
Journal of Islamic, Social, Economics and Development (JISED)
eISSN: 0128-1755
Journal website: academicinspired.com/jised
DOI: 10.55573/JISED.118544

Ubaidullah, N. H., Mohamed, Z., Hamid, J., Sulaiman, S., & Yussof, R. L. (2021). Improving novice students' computational thinking skills by problem-solving and metacognitive techniques. *International Journal of Learning, Teaching and Educational Research, 20*(6), 88–108. doi:10.26803/ijlter.20.6.5

Xie, B., Lim, J. O., Pham, P. K. D., Li, M., & Ko, A. J. (2023). Developing novice programmers' self-regulation skills with code replays. In *Proceedings of the 2023 ACM Conference on International Computing Education Research V.1* (pp. 298–313). New York, NY: Association for Computing Machinery. doi:10.1145/3568813.3600127

Zimmerman, B. J. (2002). Becoming a self-regulated learner: An overview. *Theory Into Practice, 41*(2), 64–70.