Students' Feedback Requests and Interactions with the SCRIPT Chatbot: Do They Get What They Ask For?

Andreas Scholl
Computer Science
Nuremberg Tech
andreas.scholl@th-nuernberg.de

Natalie Kiesler
Computer Science
Nuremberg Tech
natalie.kiesler@th-nuernberg.de

Abstract

Building on prior research on Generative AI (GenAI) and related tools for programming education, we developed SCRIPT, a chatbot based on ChatGPT-4o-mini, to support novice learners. SCRIPT allows for open-ended interactions and structured guidance through predefined prompts. We evaluated the tool via an experiment with 136 students from an introductory programming course at a large German university and analyzed how students interacted with SCRIPT while solving programming tasks with a focus on their feedback preferences. The results reveal that students' feedback requests seem to follow a specific sequence. Moreover, the chatbot responses aligned well with students' requested feedback types (in 75%), and it adhered to the system prompt constraints. These insights inform the design of GenAI-based learning support systems and highlight challenges in balancing guidance and flexibility in AI-assisted tools.

1. Introduction

Ever since the broad availability of GenAI, we have seen an increasing number of application contexts in computing education, and particularly introductory programming (Prather et al., 2023, 2025). This development is not surprising, as GenAI is capable of passing introductory programming exercises and courses (Geng, Zhang, Pientka, & Si, 2023; Kiesler & Schiffner, 2023a; Savelka, Agarwal, Bogart, & Sakr, 2023). They can generate programming error messages (Leinonen et al., 2023; Sarsa, Denny, Hellas, & Leinonen, 2022; MacNeil et al., 2023), exercises (Jacobs, Peters, Jaschke, & Kiesler, 2025), and formative feedback (Bengtsson & Kaliff, 2023; Kiesler, Lohr, & Keuning, 2024; Jacobs & Jaschke, 2024; Roest, Keuning, & Jeuring, 2023; Lohr, Keuning, & Kiesler, 2025). For example, it has been shown that it is possible to elicit certain types of feedback for programming novices by using specific prompts (Lohr et al., 2025). Moreover, recent GenAI tools have advanced rapidly, and some of them can generate precise, structured, and personalized feedback for novice programmers (Azaiz, Kiesler, & Strickroth, 2024; Phung et al., 2023; Scholl & Kiesler, 2024).

Due to GenAI's potential for learners, several prototypes and tools have been developed to integrate GenAI into programming education. Context-specific tools with feedback for novice learners of programming comprise, for example, CodeAid (Kazemitabaar et al., 2024), Codehelp (Liffiton, Sheese, Savelka, & Denny, 2024), LLM Hint Factory (Xiao, Hou, & Stamper, 2024), and the StAP-tutor (Roest et al., 2023). Other chatbots, such as the CS50 Duck (Liu et al., 2024, 2025) and the CodeTutor (Lyu, Wang, Chung, Sun, & Zhang, 2024) can provide context-aware feedback. Tutor Kai (Jacobs, Peters, et al., 2025) can generate comprehensive programming tasks, including problem descriptions, code skeletons, unit tests, and model solutions (Jacobs, Peters, et al., 2025) to help students practice and gain experience while also getting feedback (Jacobs, Kempf, & Kiesler, 2025). Generally speaking, all of these tools are designed to support novice programmers struggling with various challenges. Among them are cognitively complex tasks in introductory programming (e.g., problem understanding, designing and writing algorithms, debugging, and understanding error messages (Kiesler, 2024; Luxton-Reilly et al., 2018; Ebert & Ring, 2016; Spohrer & Soloway, 1986; Du Boulay, 1986)). Educators' high and unrealistic expectations (Luxton-Reilly, 2016; Luxton-Reilly et al., 2018; Whalley, Clear, & Lister, 2007; Kiesler, 2022), a low student-educator ratio (Petersen, Craig, Campbell, & Tafliovich, 2016), and an increasingly diverse student body in higher education (e.g., due to diverse learners and educational biographies) add to the list of novices' challenges.

Considering the potential of GenAI and the challenging nature of introductory programming education, it is crucial to keep advancing tutorial systems capable of providing the feedback computing students need and want. Despite the availability of certain tools for novice programmers (e.g., CodeAid, etc.), there is no one-size-fits-all solution and no consensus on how to design a chatbot with pedagogical guardrails yet. Prior research analyzed students' interactions with GenAI tools, such as ChatGPT (Scholl, Schiffner, & Kiesler, 2024). It revealed diverse usage patterns, e.g., superficial engagement, such as seeking quick solutions, but also extensive and iterative dialogues. The evaluation of the student perspective indicates their appreciation of GenAI's continuous availability and immediate responses (Scholl & Kiesler, 2024). At the same time, students expressed their concerns about overreliance, inconsistent responses, and a lack of pedagogical guidance (Scholl & Kiesler, 2024). Motivated by these insights and in line with the lessons learned from the design of similar tools, we developed a *Supportive Chatbot for Resolving Introductory Programming Tasks* (SCRIPT) for novice learners of programmers. It is based on ChatGPT-4o-mini, and integrates task description and context into the system prompt. Computing students are offered predefined prompts for specific feedback types (Keuning, Jeuring, & Heeren, 2018; Narciss, 2006) based on related work (Lohr et al., 2025). In addition, students can enter free-form input, and receive step-by-step responses (Liffiton et al., 2024) without giving away (model) solutions.

In this research paper, the **goal** is to investigate students' interactions with SCRIPT in terms of requested feedback types. In addition, we evaluate to what extent the generated responses match students' requests and whether the output adheres to the system prompt constraints. The results **contribute** to increasing our understanding of GenAI-based chatbots for introductory programming education and how to design them to align with learners' needs. They particularly inform us of learners' needs in terms of AI-generated feedback for programming tasks. These findings thus have implications for tool developers and educators, but also computing students applying respective tools.

2. Related Work

In the past few years, numerous application contexts were evaluated to show the potential of GenAI for computing education, and introductory programming in particular. For example, GenAI and related tools (e.g., ChatGPT, Codex, Copilot, or Llama) have been shown to successfully solve introductory programming tasks and pass respective exams (Wermelinger, 2023; Geng et al., 2023; Kiesler & Schiffner, 2023a; Savelka et al., 2023). Other studies revealed they can enhance programming error messages (Leinonen et al., 2023), and effectively generate code explanations (MacNeil et al., 2023; Sarsa et al., 2022). Explanations by GenAI were found to be capable of analyzing student code and providing instruction on how to fix errors (Phung et al., 2023; Zhang et al., 2022). Even elaborate feedback types were identified in GenAI output in response to student code (Kiesler et al., 2024; Azaiz et al., 2024). In the following, we present relevant research on GenAI feedback, and customized GenAI tools that informed the development of our system (SCRIPT), and the research design.

2.1. Feedback Potential of GenAI

The generation of feedback is considered a huge potential of GenAI, especially with the rapid advancements of the underlying large language models. A study by Phung et al. (Phung et al., 2023) investigated the use of LLMs to fix syntax errors in Python programs. They developed a technique to receive high-precision feedback from Codex. By using qualitative methods, several other studies explored the feedback characteristics of GenAI in response to student solutions containing errors (Kiesler et al., 2024; Azaiz et al., 2024). For example, Kiesler et al. (Kiesler et al., 2024) identified the following feedback elements: stylistic advice, textual explanations of the cause of errors and their fix, examples, metacognitive and motivational elements. However, they recognized misleading information and uncertainty in the model's output, depending on the task. ChatGPT-3.5 also requested more information in a few cases. A qualitative evaluation of GPT4 Turbo's feedback showed notable improvements (Azaiz et al., 2024). The outputs were more structured, consistent, and always personalized (Azaiz et al., 2024).

Lohr et al. (Lohr et al., 2025) investigated whether they can generate specific feedback for introduc-

tory programming tasks using GenAI (i.e., ChatGPT). Following an iterative approach, they designed prompts to elicit specific feedback types, such as knowledge about mistakes, or knowledge on how to proceed (cf. (Narciss, 2008; Keuning et al., 2018)). They qualitatively evaluated the generated output with human intelligence and determined its feedback type to check if the prompts had elicited the expected feedback. As a result, they present prompts for the generation of different types of feedback suitable for introductory programming tasks and student submissions (Lohr et al., 2025). They also noted that misleading information occurred less frequently compared to related work (Kiesler et al., 2024) (Lohr et al., 2025).

2.2. Customized GenAI Tools for Programming Education

Due to their feedback potential, GenAI models are being integrated into educational tools and systems to offer novice-friendly explanations tailored to individual student errors. The dcc - -help tool, for example, integrates ChatGPT 3.5 into the Debugging C Compiler (DCC). It produces context-aware explanations in response to compiler errors in DCC (Taylor, Vassar, Renzella, & Pearce, 2024). A more recent conversational AI extension to the GenAI-enhanced C/C++ compiler DCC is DCC Sidekick (Renzella, Vassar, Lee Solano, & Taylor, 2025). It generates pedagogical programming error explanations without integrating student input, and by applying the Socratic method.

Another example of a programming assistant is CodeAid (Kazemitabaar et al., 2024). Its implementation avoids the generation of code solutions to support students' thinking and learning. CodeAid offers help with general questions, inline code explorations, questions from code, help to fix code, code explanations, and help in writing code. Yet, the developers did not distinguish or implement any well-known and context-specific feedback typology (cf. (Keuning et al., 2018)). Kazemitabaar et al. (2024) summarize four design considerations for AI coding assistants: (1) exploiting unique advantages of AI, (2) designing the AI querying interface, (3) balancing the directness of AI responses, and (4) supporting trust, transparency and control.

CodeHelp is another GenAI-powered tool designed to provide guardrails and on-demand programming assistance (Liffiton et al., 2024). Its features comprise a simple interface for students' help requests, a system prompt avoiding the generation of complete code solutions, and a sufficiency check to catch ambiguous or incomplete student queries. CodeHelp was evaluated by 52 students over a 12-week period. The results show students value the availability and help of the tool, and that it complements the efforts of the instructor (Liffiton et al., 2024).

Xiao et al. (2024) followed a somewhat different approach by investigating whether and how different hints support students' problem-solving and learning processes in the LLM Hint Factory (Xiao et al., 2024). The latter is a system providing four levels of hints, from natural language to concrete code assistance. Via a think-aloud study with 12 programming novices, they identified recommendations for feedback design. For example, hints about the next step or how to solve syntax issues should be concise and personalized to students' requests to meet their needs. Otherwise, students may switch to ChatGPT to receive full solutions according to the authors of the study (Xiao et al., 2024). Similarly, Roest et al. 2023 developed the StAP-tutor; a GenAI-based tutoring system for next-step hints (Roest et al., 2023). The system was designed by exploring various prompts and evaluating the feedback. The best output was generated for inputs with a problem description and the words "student" and "hint". Another recommendation is to increase the temperature parameters of the used model. Through a student and expert evaluation, the feedback was evaluated as concise and personalized, but it still contained misleading information for OpenAI's GPT-3.5-turbo model. Thus, future work is required to evaluate more recent models (Roest et al., 2023).

There are some other recent applications and studies on the use of GenAI in computing. For example, Yeh et al. (2025) investigated an interactive approach to teach students how to write better prompts so they can eventually generate working code (Yeh et al., 2025). Bhowmick and Li (2025) experimented with LANTERN to teach students relational query processing as part of a database course (Bhowmick & Li, 2025). Similarly, Riazi and Rooshenas (2025) explored GenAI feedback to support students' concep-

tual design competencies (e.g., via entity-relationship diagrams) in a database systems course (Riazi & Rooshenas, 2025). Forden et al. (2025) developed an automated assessment tool trying to cultivate students' coding style and foster timely submissions (Forden, Schneider, Gebhard, Islam Molla, & Brylow, 2025). A last example is SENSAI, an AI-powered tutoring system for teaching cybersecurity (Nelson, Doupé, & Shoshitaishvili, 2025). It utilizes the learner's workspace, i.e., active terminals and edited files as input, highlighting the importance of context in the system prompt.

2.3. Research Gap

It is crucial to keep investigating newly emerging large language models (e.g., ChatGPT 4o, 4.5, and OpenAI o1, o3), which is due to the fast pace of this technology (Prather et al., 2023). This is particularly relevant as research data (e.g., for benchmarking) are usually not available (Kiesler & Schiffner, 2023b; Prather et al., 2023). Few studies utilize the same dataset for evaluating or benchmarking recent models. Moreover, the general limitations of GenAI models are well-known. Among them are inaccuracies, hallucinations, misleading information (especially for novice learners), data and privacy concerns, reproduction of bias and stereotypes, lack of accessibility, and their in-transparency by design (Gill et al., 2024; Prather et al., 2023; Zhai, 2022; Kiesler et al., 2024, 2025; Alshaigy & Grande, 2024). Hence, we need to critically reflect upon the use of GenAI, and how to provide pedagogical instruction and guardrails for students. The same applies to the development and design of tools based on GenAI models. A major challenge in this design is finding a balance between individual student support and avoiding the extensive generation of model solutions. This can be addressed by restricting the chatbot from generating any code at all (cf. (Liffiton et al., 2024)).

Our goal, however, is to enable more flexible interactions, where the chatbot can produce helpful code snippets without giving away complete solutions. We also postulate that students can benefit from individual feedback and help, and code snippets may be helpful for some of them. In addition, we want to provide exemplary prompts leading to specific types of feedback, e.g., indicating and explaining error(s), or the next steps to solve them, as successful prompting can be challenging for novices as well. Moreover, we need to integrate students into the discussion to qualitatively evaluate how students apply GenAI tools (customized or not). Therefore, a critical aspect of this work is whether AI-generated feedback addresses students' informational needs.

3. Methodology

In this study, we present SCRIPT, a chatbot based on ChatGPT-4o-mini. It is designed to support novice learners of programming seeking feedback. The evaluation of SCRIPT is guided by the following research questions:

RQ1 *How do students interact with SCRIPT in the context of introductory programming tasks?*

RQ2 *To what extent does the generated output match students' requests?*

RQ3 *To what extent do the generated outputs adhere to the system prompt constraints?*

To address these RQs, we utilize empirical data from students (n=136) enrolled in an introductory programming class at a large German university. Students were asked to solve programming exercises at home while having access to the SCRIPT chatbot. The chat protocols were automatically recorded, and students were asked to rate the GenAI-powered responses and leave comments (all voluntarily). In the following subsections, we introduce SCRIPT, the course context, selected tasks, and data analysis methods. Our research data (e.g., tasks, system prompt, interactions, etc.) are available in an online repository (Scholl & Kiesler, 2025a).

3.1. Introducing SCRIPT

SCRIPT (Scholl & Kiesler, 2025b) is implemented as a standalone web application, which is accessible via a link in the respective university's Moodle course (no additional sign-in needed). The setup enables anonymous data gathering (via anonymous IDs). The user interface (UI) is depicted in Figure 1. It is similar to the structure of ChatGPT's interface. Each chat session is displayed at the very left

of the UI. It is task-specific, meaning a conversation is dedicated to solving one programming problem. Students can initiate multiple conversations, including revisiting the same task. The chatbot receives both the task description (displayed on the right) and a reference solution as context (i.e., as part of the system prompt (Scholl & Kiesler, 2025a)), eliminating the need for students to manually copy and paste these elements into the chat, which is supposed to ease its use (Kazemitabaar et al., 2024). Students can rate each GenAI response with a thumbs-up or down feedback, and via textual comments (all optional). The backend architecture relies on a Node.js environment with an Express/Socket.io server. The data management is handled via an SQL database (MariaDB). Student interactions and their feedback (thumbs-up/down ratings, and open input) are logged.

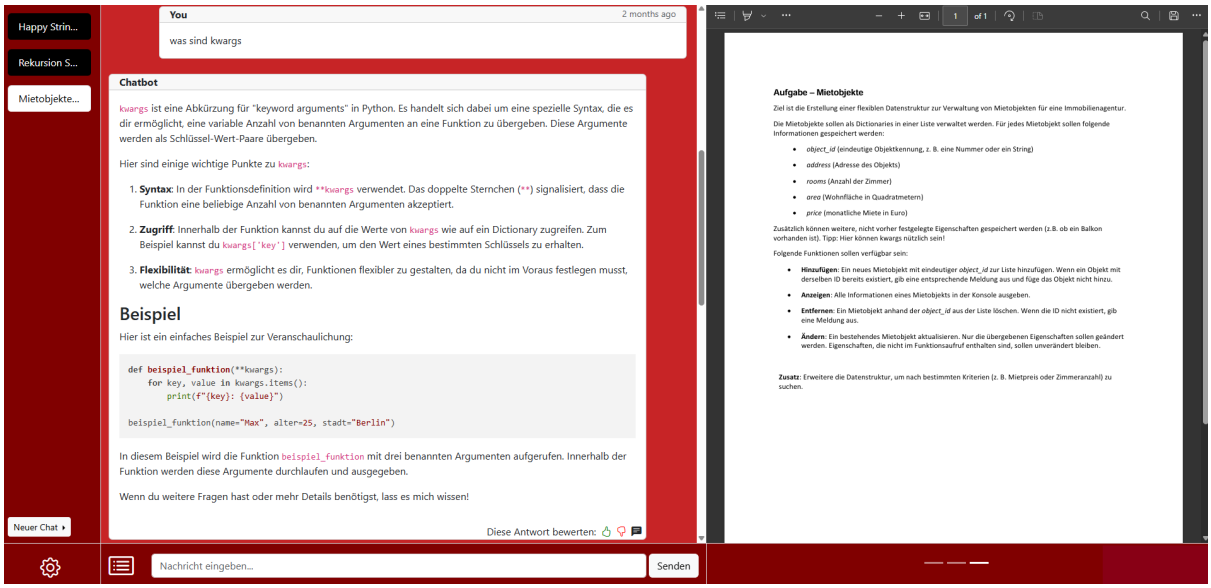

Figure 1 – Overview of the SCRIPT User Interface

SCRIPT provides two distinct prompting options, which are designed to exploit the unique advantages of GenAI while balancing guidance and flexibility (Kazemitabaar et al., 2024; Yeh et al., 2025). *Closed prompts* are predefined based on recent research on AI-generated feedback (Lohr et al., 2025) and an established feedback typology for programming contexts (Keuning et al., 2018; Narciss, 2006). These closed prompts guide students through problem-solving steps and issues, including understanding task constraints (KTC), identifying required programming concepts (KC), recognizing mistakes (KM), determining how to proceed (KH), assessing performance levels (KP), and evaluating code correctness relative to a reference solution (KR). These closed prompts are available for students next to the input field. They are supposed to encourage students' structured engagement with the chatbot, and reduce the challenges of starting with the problem-solving process, or formulating a prompt (cf. (Scholl & Kiesler, 2024; Scholl et al., 2024)).

Open prompts are common when using a chatbot, and are comparable to those expected by CodeHelp (Liffiton et al., 2024). They denote students' free-form textual input. When students enter text, SCRIPT prevents the generation of complete code as output. Instead, it fosters step-by-step problem-solving by identifying errors in student code without directly correcting them. Thus, the tool provides stepwise hints and no complete solutions in the form of code. It also offers examples (unrelated to the specific task) and templates (incomplete code structures with only comments in key sections). SCRIPT is available at any time, so students can work at their own pace, with all tasks being accessible. This is supposed to support mastery learning where learners can repeatedly engage with a task until achieving proficiency (Szabo et al., 2025; Keller, 1968). The dual-mode interaction structure with closed and open prompts allows students to choose between guided problem-solving and a more exploratory learning approach. The system prompts for both modes are available in the online repository (Scholl & Kiesler, 2025a).

3.2. Course Context and Task Selection

SCRIPT was designed to support students in an introductory programming course for first-year computing students ($N = 666$) at Goethe University Frankfurt (Germany). The present study was conducted in the winter term 2024/25. The majority of students were enrolled in the bachelor's degree program in Computer Science (CS). Only some were enrolled in other disciplines or chose CS as a minor. Prior programming experience was not required to participate. A Moodle course provided access to learning materials, including SCRIPT. The course structure included a weekly two-hour lecture for all students. Tutorial sessions (20 students each) accompanied the course. Tutorial sessions typically last two hours every week. A key component of the tutorials is the weekly or biweekly exercises, so students can gain hands-on programming experience. Students work individually on tasks and earn exercise points for their submissions. These contribute to the final exam score.

For this study, we designed three programming tasks for students to be completed using SCRIPT. The tasks were supposed to be solved within one week, starting on January 13, 2025. Students had the opportunity to engage with one or more tasks. Students were not instructed on how to use SCRIPT. Participation was voluntary, but students could earn four exercise points as an incentive. Furthermore, the purpose of the study and its procedures were introduced to students during the lecture preceding their tutorial sessions. Table 1 summarizes the tasks used in this study and the concepts they address. The given tasks were aligned with the course curriculum and the student's assumed/expected level of expertise. The course's facilitator also ensured that the tasks were relevant and adequate. The full task descriptions are available along with the other research data (Scholl & Kiesler, 2025a).

Task	Description	Concepts
Happy Strings	Compute the number of "happy" strings within all sub-strings of a given string of digits. A string is considered "happy" if the digits can be rearranged to repeat twice. The following steps are required: (a) Test for "happy" property; (b) iterate and test all substrings.	recursion, functions, lists, conditionals, string manipulation
Recursion Snippets	Determine recursion type, return value and number of function calls of 4 functions: (a) sum of digits; (b) list reversal; (c) multiplication (d) Ackermann function.	
Rental Properties	Implement a data structure to manage rental properties using dictionaries in a list. Provide functions to add, display, remove, and update properties based on unique IDs. Extend functionality to allow searching by criteria.	functions, lists, dictionaries, keyword parameters, data structures

Table 1 – Selected tasks with short description and required programming concepts.

3.3. Data Analysis

To examine students' interactions with SCRIPT, we collected and analyzed chat sessions from 136 students with their input and SCRIPT's output. The chat logs were stored as individual sessions (without any personal data or identifier). They also include students' ratings of SCRIPTs' responses, and textual feedback (if any). The chat logs constitute the data basis for RQ1, RQ2, and RQ3.

For RQ1 and RQ2, we particularly focused on the student prompts and SCRIPTs responses to them. All of these questions and answers were analyzed and categorized regarding the requested and generated feedback type(s): knowledge of result (KR), knowledge of correct result (KCR), knowledge of performance (KP), knowledge about task constraints (KTC), knowledge about concepts (KC), knowledge about mistakes (KM), knowledge on how to proceed (KH), and knowledge about meta-cognition (KMC) (Keuning et al., 2018). The analysis followed a structured coding approach. We started with approximately 15% of the chat sessions to confirm the adequacy of the deductive coding scheme based on the literature (Keuning et al., 2018). It applied to almost all feedback requests and responses. However, some additional requests and responses were noticed and coded inductively based on the material. For example, students' prompts were categorized as: technical (TEC), social interaction (SoI), answer to GenAI question (ANS), clarification question (WHAT), off-topic (OFT), new task request (TR), incomprehensive (IN), and prompt injection (HACK). Similarly, SCRIPTs' responses were coded with these additional categories: marked as offensive (OFF), denied request (DENY), social (SoI), technical

(TEC), off-topic (OFT), technical error (TE), and new task (TR). (Definitions and anchor examples of the additional categories are available in the data publication (Scholl & Kiesler, 2025a).) As part of this rule-guided coding process, multiple codes (a maximum of 3) were applied to a coding unit (i.e., a student input and a generated response each was considered a single coding unit). A student's entire chat session was considered a context unit, in case of uncertainties (e.g., regarding the intentions of a student trying to elicit feedback or have a nice chat). After the initial round, deductive and inductive categories were applied to the remaining material. All of the collected chat data were coded twice by the same coder (first author of this work) to ensure internal consistency and reliability. The coder had prior experience in qualitative coding.

To answer RQ1, the student requests and chatbot responses were aggregated and processed into a flowchart to represent the interactions and their frequencies. Regarding RQ2, we compared the feedback type request of every student question with the feedback category/categories evident in SCRIPTs' responses (both in an aggregated and pair-wise form). We present the number of matches, over-matches (i.e., a match of requested and generated feedback types, plus additional feedback categories in the response), partial matches, or mismatches.

The last research question (RQ3) had the goal of evaluating the quality of SCRIPTs' outputs in terms of correctness, and step-wise hints. We addressed those aspects through the following indicators (cf. (Azaiz et al., 2024; Azaiz, Kiesler, Strickroth, & Zhang, 2025; Roest et al., 2023)): (1) *Number of problem-solving steps* SCRIPT provides in a response. Per system prompt, a single step per response should be provided. (2) *Solution* provided to the task (partial or complete), whereas no full solution should be generated. (3) *Code examples*; SCRIPT is instructed to provide only simple examples. (4) *Code templates*; SCRIPT may provide code templates to students, consisting of structural frameworks for a function, loop, or conditional headers. (5) *Code Corrections*; SCRIPT should correct students' code by pointing out the mistake and providing hints. Hence, we evaluate whether SCRIPT provides corrected code. (6) *Correctness of Response*; we determine if a response from SCRIPT is correct, partially correct, or incorrect.

4. Results

In total, 136 students engaged with SCRIPT, generating 241 chat sessions. Across these interactions, students submitted 1,409 prompts, and SCRIPT generated 1,409 responses. The distribution of prompts per student varied, though, with a mean of 10.36 (SD = 10.86) and a median of 7. Among the student prompts were 207 predefined "closed" prompts, requesting the following feedback types: KC (54), KTC (48), KR (49), KM (26), KP (17), and KH (13).

4.1. Students' Interactions with SCRIPT (RQ1)

To answer RQ1, we analyzed the feedback requests and generated responses w.r.t. their feedback types. The resulting feedback (and other) categories of inputs and outputs were aggregated in a flowchart, as displayed in Figure 2. It illustrates common interaction patterns between students and SCRIPT. Nodes indicate the feedback types of students' prompts, and edges represent the feedback generated by SCRIPT. The flowchart includes loops, such as for KH, which illustrate repeated requests for the same feedback type. Next to the feedback type, we always provide the number of occurrences. Figure 2 only represents those interaction patterns observed at least ten times, and thus were more common. It is important to note that the figure represents the results in an aggregated form and does not display individual student paths. A more detailed flowchart and the raw data are available online (Scholl & Kiesler, 2025a).

In the following, we describe the most frequent requests and responses as depicted in Figure 2. After a default greeting by SCRIPT, students began their session by asking questions about the task constraints (KTC) in 71 cases. This was often followed by questions (19) regarding the necessary programming concepts (KC). Next, students (46) engaged in multiple follow-up inquiries, requesting further explanations or clarifications of concepts. Then they submitted their (partial) solution and asked SCRIPT to evaluate its correctness (KR, 14). This process often occurred iteratively, with students refining their

code based on SCRIPT's feedback (28). Instead of simply providing a binary correct/incorrect response, SCRIPT typically offered reasoning about the student's performance (KP, 16), clarifying key concepts (KC, 12), or suggesting how to approach corrections (KH, 16). However, when students used predefined *closed prompts*, SCRIPT adhered to a stricter response format, delivering only KR feedback (i.e., correct/incorrect). This often led students to seek additional clarification about their mistakes through *open prompts*. Another interaction pattern was students requesting the correct solution (KCR) at the beginning of the session. In 18 of the 23 cases, SCRIPT explicitly denied these solution requests.

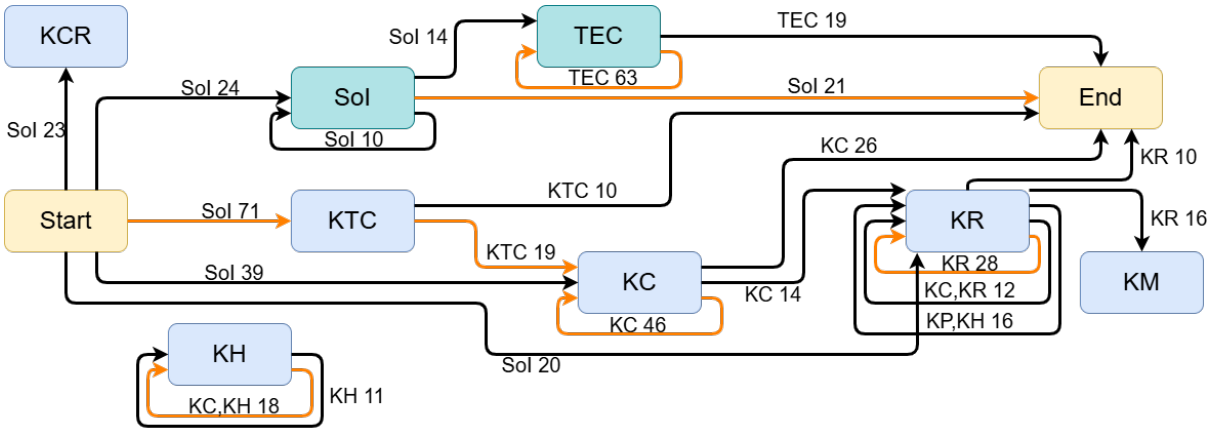

Figure 2 – Flowchart of students' interactions. Nodes mark students' feedback requests, edges represent SCRIPTs' feedback categories. Only interactions that occurred at least 10 times are represented.

In 24 chat sessions, students initiated conversations with social interactions, sometimes (10) followed by additional social exchanges. In 14 sessions, this led students to ask technical questions about SCRIPT before engaging with the task. Some sessions (19) ended at this step without students attempting to solve the problem. Regardless of how a student started or concluded their chat, students asked how to proceed (KH) at different stages of the problem-solving process (see Figure 2). KH loops have occurred at least 10 times (KH, 11; KC, KH, 18). Thus, the number of outgoing requests from KH to other feedback types was less than 10, which is why the KH node is isolated in the aggregated flowchart.

SCRIPT typically responded with step-by-step explanations, often adding conceptual knowledge (e.g., KC, KH 18) to guide students through the task and next steps. This iterative exchange generally continued until the student reached a solution. However, a few students relied solely on requesting the next steps without adding their code as input. In such cases, SCRIPT constructed the solution incrementally, guiding the student through each step of the process.

4.2. Matches of Student Requests and SCRIPT's Responses (RQ2)

To understand the extent to which the generated output matched students' requests, we first of all aggregated all requested and provided feedback types in Table 2a. It shows the total number of occurrences of requested and generated feedback types across all student prompts and responses. For most feedback categories, the numbers align closely, indicating that SCRIPT generally generated what students requested. For some requests, SCRIPT supplemented its responses with additional conceptual explanations (KC, 111) and guidance on how to proceed (KH, 106), particularly when students asked for feedback for their code (i.e., had requested KM (27), or KR (37)). For this reason, the number of feedback types in the responses exceeds the number of requests for KC, and KH (see Table 2a).

We also evaluated to what extent the student feedback requests and the immediate responses from SCRIPT were aligned (i.e., in the question-response pairs). It should be noted though that not all student inputs requested feedback, which is why we only evaluated 891 question-response-pairs. In 47% of cases (417 out of 891), the chatbot's feedback directly matched what students requested. We also observed so-called "over-matching". That is, SCRIPT provided additional feedback types, and not

Feedback Type	No. Requests	No. Responses
KTC	158	147
KC	295	424
KH	159	352
KM	84	81
KP	36	71
KR	209	149
KCR	100	120
KMC	6	6

(a) Feedback Types

Additional Cat.	No. in Prompts	No. in Responses
TEC	165	152
SoI	101	106
ANS	141	-
WHAT	71	-
OFT	48	45
TR	15	15
IN	14	-
HACK	26	-
DENY	-	42
OFF	-	9
TE	-	58

(b) Additional Categories

Table 2 – Total number of requested and generated (a) feedback types and (b) additional categories (both not pair-wise); prompts and responses may contain multiple types each.

just those that were explicitly requested. Considering these, the total alignment increased to 75%, i.e., an additional 255 out of the 891 pairs were over-matching. Overmatching primarily occurred for KC and KH categories, where SCRIPT offered extra explanations or next-step guidance beyond what was explicitly requested. In 22% of question-response pairs (195 out of 891), the response did not match the intended feedback type (mismatch). Notably, 25% of these mismatches (49 out of 195) involved students requesting direct solutions (KCR), which SCRIPT was designed to reject. This indicates that while mismatches occurred, a quarter resulted from SCRIPT correctly adhering to its constraints – rather than failing to provide the desired feedback. 58 technical errors (TE) occurred when SCRIPT refused to answer requests for KTC and KC feedback, due to missing student code (KTC and KC did not require student code to be generated, though).

Students themselves provided 151 thumbs-ratings (130 up, 21 down) and 29 short written comments. Positive remarks highlighted SCRIPT's clear explanations and helpful guidance ("Providing a template without giving everything away is helpful"). Critical comments mostly addressed incorrect evaluations, overly confident, or brief responses ("Here, the AI was confidently wrong.").

4.3. SCRIPT's Adherence to System Prompt Constraints (RQ3)

To evaluate SCRIPT's adherence to the system prompt constraints, we evaluated (1) the number of problem-solving steps, the generation of (2) solutions, (3) code examples, (4) templates, (5) code corrections, and (6) the correctness of responses of all 1409 generated responses (see Table 3). After restricting the responses from SCRIPT to problem-related answers (removing the categories TEC, SoI, DENY, OFF, OFT, TE, and TR), 1016 responses remained for analysis.

(1) In 51%, SCRIPT adhered to the constraint of providing only one step at a time in the problem-solving process. 21% of the responses comprised an overview containing multiple steps but included a clear starting point or next step (see Table 3). (2) Across all tasks, students made 100 KCR requests. SCRIPT provided solutions in 42 cases. These were usually granted after a stepwise generation of problem-solving steps. The success rate of solution requests varied across tasks, with "Recursion Snippets" having the highest proportion of fulfilled requests (71%), followed by "Happy Strings" (38%) and "Rental Properties" (25%). (3) Simple code examples were generated 104 times, and complex examples were observed in 3 responses. (4) Regarding template generation, SCRIPT correctly provided 167 templates in the intended format, offering general code structures without implementations. However, in 95 responses, it eventually provided completed templates, after progressively building up the solution and providing several partial templates. (5) SCRIPT corrected the code of students 34 times and displayed it, despite the constraint not to do so. (6) Overall, the correctness of SCRIPT's responses was high, with 81% of answers being fully correct. 14% of responses were at least partially correct, leaving only 5% of answers classified as completely incorrect.

(1) Number of solving steps in response			(6) Correctness of response		
Single Step	514	51%	Correct	822	81%
Multiple Steps	289	28%	Partially Correct	141	14%
Multiple Steps, Explicit Next-Step	213	21%	Incorrect	53	5%

(2) Solution given		(3) Given Examples		(4) Code templates		(5) Code corrections	
Partial	102	Simple	104	Provided	167	Corrected	34
Complete	129	Complex	3	Completed	95		

Table 3 – Indicators for SCRIPT's adherence to the system prompt constraints (for 1016 responses).

5. Discussion

Our findings revealed interaction patterns of introductory programming students using SCRIPT. When using the chatbot, students seemed to follow a certain (problem-solving) sequence, that has not been revealed in previous research: understanding the task constraints (KTC), identifying relevant programming concepts (KC), formulating a solution approach (KH), debugging errors (KM), verifying results (KP, KR), and ultimately comparing with the correct solution (KCR). While most interactions focused on problem-solving, some students also engaged in technical inquiries (TEC) and social exchanges, highlighting a broader spectrum of engagement beyond the mere task. Notably, KH inquiries and follow-up questions from the chatbot led to more student interactions (i.e., follow-up questions), with SCRIPT guiding students step by step.

Related to RQ2, the chatbot's responses aligned well with students' needs in most cases, particularly in clarifying task constraints (KTC), which were addressed with good accuracy. Concept explanations (KC) were generally good but occasionally too complex or lacking specific details. This suggests future improvements of the chatbot in tailoring explanations more precisely to the task. Performance feedback (KP) showed a high degree of variation and seemed random. KR feedback in *closed prompts* was mostly binary, so students usually continued with additional *open prompts* to elicit KM feedback. This may indicate that students perceived it as insufficient or had different expectations towards the generated response. Solution requests (KCR) resulted in the highest mismatch rate. This was expected as the chatbot was designed to reject such queries. Respective students tried to circumvent this via step-wise questions and gradually reconstructing the solution.

While SCRIPT generally adhered to system prompt constraints, some inconsistencies were observed. Especially for the *Recursion Snippets* task, we found that solutions were often provided too early. Code examples were consistently brief and simple. Code templates initially adhered to the guidelines but tended to be complete after several student inquiries. Direct solution requests were blocked as expected. Yet, in step-wise interactions, partial solutions emerged sooner or later, depending on the task. From a pedagogical perspective, this may not necessarily be a problem – as long as students actively process the feedback. In general, it seems recommendable to balance direct AI responses (Kazemitabaar et al., 2024), and prevent the generation of complete code solutions (Liffiton et al., 2024). Importantly, SCRIPT demonstrated robustness against prompt injection attempts, successfully resisting nearly all manipulation efforts, except for one.

An additional observation concerns the chatbot's responses to *closed prompts* (based on (Lohr et al., 2025)), which were more formal, often writing in the third-person. In contrast, students' *open* inputs exhibited a more conversational tone. We perceived *closed* and *open prompts* as two distinct conversational modes, with some abrupt transitions between the two. This observation may be attributed to the varying perspectives SCRIPT is required to adopt: *closed prompts* request structured, restricted guidance, while *open prompts* may shift the chatbot's role to a personal assistant – depending on the student input.

In general, the *closed prompts* from prior work seemed useful (Lohr et al., 2025), which is reflected in students' use of them (207 times) and the general match between student requests and generated responses. It may also be helpful for students to get started or formulate a specific prompt, e.g., due to language barriers, anxiety, or insecurity about how to use technical terms. At the same time, it seems

advisable to offer users multiple options, i.e., both *closed* and *open prompts*, thereby bridging guidance and flexibility, as suggested in related work (Yeh et al., 2025).

Finally, we noted that the feedback typology from prior work (Keuning et al., 2018) was sufficient to describe the requested and generated feedback items. This is worth discussing, as it was constructed based on the existing learning environment at the time, not chatbots and/or GenAI. However, due to the new presentation (i.e., modality, adaptation (Narciss, 2006)) of the feedback via the chatbot, we identified additional categories, such as social interactions (SoI), and many others. As GenAI and related tools advance rapidly, we expect to see new feedback categories, or changed presentation modes, etc., in the near future.

5.1. Limitations

Some of the study's limitations should be noted. For example, students knew their interactions were being analyzed, potentially influencing their behavior (Roethlisberger & Dickson, 1939). SCRIPT was also used in an unsupervised setting, allowing students to engage freely. Few students experimented with the tool, using social or technical prompts without attempting to solve the task, which may have affected interaction patterns. The study also relies on a researcher-driven analysis. Thus, students were not explicitly asked to categorize their feedback types. They might have had slightly other intentions or informational needs than those resembled in their prompts. Yet, we tried to mitigate this limitation by allowing student feedback (via thumbs, and open input field) to each generated response. Finally, this study was conducted in a single university course in one country, limiting the generalizability of the findings. While the number of participants and responses strengthens its representativeness (Boddy, 2016), results may not be transferable to different curricula or institutions.

6. Conclusions and Future Work

In this study, we investigated students' interactions with SCRIPT, a GenAI-based chatbot developed to support problem-solving in introductory programming education. The evaluation involved 136 students, who solved dedicated programming tasks using SCRIPT in an unsupervised, self-paced setting. Chat sessions were analyzed regarding students' interactions and feedback requests, prompt-response alignment, and the system's adherence to constraints. Our analysis showed that students seemed to follow a certain sequence of feedback requests: KTC, KC, KH, KM, KP and KR, and, finally, KCR. Overall, SCRIPT provided correct responses in alignment with students' requests in 75% of its responses. Moreover, the system prompt was suitable for guiding students through step-by-step problem-solving without revealing full solutions. The chatbot remained robust and followed the instructional design.

The study's findings can inform the design of better prompts and scaffolded feedback sequences to help students work more independently with such tools. Future work should explore the use of GenAI tools with a broader range of task types and learning contexts, and continuously improve these early tools and prompts. In particular, closing the gap between *open* and *closed prompts* remains a key challenge, as it requires balancing structure and flexibility while managing the shift between the educator's role of restricting responses and supporting student exploration. This will be one of the next steps in advancing SCRIPT, to make the conversation flow more naturally. Moreover, we will use the students' feedback to improve the technical implementation, User Interface, and interaction. Continuing this work will help support novice programmers seeking feedback by SCRIPT. We also encourage other CS researchers, educators, and tool developers to keep exploring educational tools so we can leverage the potential of GenAI for good.

7. References

Alshaigy, B., & Grande, V. (2024). Forgotten again: Addressing accessibility challenges of generative ai tools for people with disabilities. In *Adjunct proceedings of the 2024 nordic conference on human-computer interaction.* New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3677045.3685493

Azaiz, I., Kiesler, N., & Strickroth, S. (2024). *Feedback-generation for programming exercises with gpt-4.* New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3649217.3653594

Azaiz, I., Kiesler, N., Strickroth, S., & Zhang, A. (2025). *Open, small, rigmarole – evaluating llama 3.2 3b's feedback for programming exercises.* (accepted to the International Journal of Engineering Pedagogy (iJEP; eISSN: 2192-4880)) doi: 10.48550/arXiv.2504.01054

Bengtsson, D., & Kaliff, A. (2023). *Assessment accuracy of a large language model on programming assignments.* Retrieved from `https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-331000`

Bhowmick, S., & Li, H. (2025). Experience report on using lantern in teaching relational query processing. In *Proceedings of the 56th acm technical symposium on computer science education v. 1* (p. 123–129). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3641554.3701812

Boddy, C. R. (2016). Sample size for qualitative research. *Qualitative market research: An international journal*, *19*(4), 426–432.

Du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, *2*(1), 57–73. doi: 10.2190/3LFX-9RRF-67T8-UVK9

Ebert, M., & Ring, M. (2016). A presentation framework for programming in programing lectures. In *Proc. educon* (pp. 369–374).

Forden, J., Schneider, M., Gebhard, A., Islam Molla, M. T., & Brylow, D. (2025). Unlocking student potential with ta-bot: Timely submissions and improved code style. In *Proceedings of the 56th acm technical symposium on computer science education v. 1* (p. 346–352). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3641554.3701955

Geng, C., Zhang, Y., Pientka, B., & Si, X. (2023). *Can ChatGPT Pass An Introductory Level Functional Language Programming Course?*

Gill, S. S., Xu, M., Patros, P., Wu, H., Kaur, R., Kaur, K., ... Buyya, R. (2024). Transformative effects of chatgpt on modern education: Emerging era of ai chatbots. *Internet of Things and Cyber-Physical Systems*, *4*, 19-23. doi: 10.1016/j.iotcps.2023.06.002

Jacobs, S., & Jaschke, S. (2024, May). Evaluating the Application of Large Language Models to Generate Feedback in Programming Education. In *2024 IEEE Global Engineering Education Conference (EDUCON)* (pp. 1–5). New York: IEEE. doi: 10.1109/EDUCON60312.2024.10578838

Jacobs, S., Kempf, M., & Kiesler, N. (2025). That's not the feedback i need! – student engagement with genai feedback in the tutor kai.. Retrieved from `https://arxiv.org/abs/2506.20433`

Jacobs, S., Peters, H., Jaschke, S., & Kiesler, N. (2025). *Unlimited practice opportunities: Automated generation of comprehensive, personalized programming tasks.* (accepted at ITiCSE 2025, https://doi.org/10.1145/3724363.3729089) doi: 10.48550/arXiv.2503.11704

Kazemitabaar, M., Ye, R., Wang, X., Henley, A. Z., Denny, P., Craig, M., & Grossman, T. (2024). Codeaid: Evaluating a classroom deployment of an llm-based programming assistant that balances student and educator needs. In *Proceedings of the chi conference on human factors in computing systems.* New York, USA: ACM. doi: 10.1145/3613904.3642773

Keller, F. S. (1968). Good-bye, teacher... *Journal of applied behavior analysis*, *1*(1), 79.

Keuning, H., Jeuring, J., & Heeren, B. (2018, 9). A systematic literature review of automated feedback generation for programming exercises. *ACM Trans. Comput. Educ.*, *19*(1). doi: 10.1145/3231711

Kiesler, N. (2022). *Kompetenzförderung in der programmierausbildung durch modellierung von kompetenzen und informativem feedback* (Dissertation). Johann Wolfgang Goethe-Universität, Frankfurt am Main. (Fachbereich Informatik und Mathematik)

Kiesler, N. (2024). *Modeling programming competency: A qualitative analysis*. Cham: Springer

International Publishing. doi: 10.1007/978-3-031-47148-3

Kiesler, N., Lohr, D., & Keuning, H. (2024). Exploring the potential of large language models to generate formative programming feedback. In *2023 ieee frontiers in education conference (fie)* (p. 1-5). doi: 10.1109/FIE58773.2023.10343457

Kiesler, N., & Schiffner, D. (2023a). *Large language models in introductory programming education: Chatgpt's performance and implications for assessments.* doi: 10.48550/arXiv.2308.08572

Kiesler, N., & Schiffner, D. (2023b). Why We Need Open Data in Computer Science Education Research. In *Proceedings of the 2023 conference on innovation and technology in computer science education v. 1* (p. 348–353). New York: ACM. doi: 10.1145/3587102.3588860

Kiesler, N., Smith, J., Leinonen, J., Fox, A., MacNeil, S., & Ihantola, P. (2025). *The role of generative ai in software student collaboraition.* New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3724363.3729040

Leinonen, J., Hellas, A., Sarsa, S., Reeves, B., Denny, P., Prather, J., & Becker, B. A. (2023, March). Using large language models to enhance programming error messages. In *Proc. sigcse.* ACM. doi: 10.1145/3545945.3569770

Liffiton, M., Sheese, B. E., Savelka, J., & Denny, P. (2024). Codehelp: Using large language models with guardrails for scalable support in programming classes. In *Proceedings of the 23rd koli calling international conference on computing education research.* New York: ACM. doi: 10.1145/3631802.3631830

Liu, R., Zenke, C., Liu, C., Holmes, A., Thornton, P., & Malan, D. J. (2024). Teaching cs50 with ai: Leveraging generative artificial intelligence in computer science education. In *Proceedings of the 55th acm technical symposium on computer science education v. 1* (p. 750–756). doi: 10.1145/3626252.3630938

Liu, R., Zhao, J., Xu, B., Perez, C., Zhukovets, Y., & Malan, D. J. (2025). Improving ai in cs50: Leveraging human feedback for better learning. In *Proceedings of the 56th acm technical symposium on computer science education v. 1* (p. 715–721). New York, NY, USA: ACM. doi: 10.1145/3641554.3701945

Lohr, D., Keuning, H., & Kiesler, N. (2025). You're (Not) My Type – Can LLMs Generate Feedback of Specific Types for Introductory Programming Tasks? *Journal of Computer Assisted Learning*. doi: 10.1111/jcal.13107

Luxton-Reilly, A. (2016). Learning to Program is Easy. In *Proc. ITiCSE* (pp. 284–289). doi: 10.1145/2899415.2899432

Luxton-Reilly, A., Simon, Albluwi, I., Becker, B. A., Giannakos, M., Kumar, A. N., … Szabo, C. (2018). Introductory Programming: A Systematic Literature Review. In *Proc. ITiCSE* (pp. 55–106). New York: ACM. doi: 10.1145/3293881.3295779

Lyu, W., Wang, Y., Chung, T. R., Sun, Y., & Zhang, Y. (2024). Evaluating the effectiveness of llms in introductory computer science education: A semester-long field study. *arXiv:2404.13414*.

MacNeil, S., Tran, A., Hellas, A., Kim, J., Sarsa, S., Denny, P., … Leinonen, J. (2023). Experiences from Using Code Explanations Generated by Large Language Models in a Web Software Development E-Book. In *Proc. sigcse ts* (p. 931–937). doi: 10.1145/3545945.3569785

Narciss, S. (2006). *Informatives tutorielles feedback: Entwicklungs- und evaluationsprinzipien auf der basis instruktionspsychologischer erkenntnisse*. Münster: Waxmann Verlag.

Narciss, S. (2008). Feedback strategies for interactive learning tasks. *Handbook of research on educational communications and technology*, *3*, 125–144.

Nelson, C., Doupé, A., & Shoshitaishvili, Y. (2025). Sensai: Large language models as applied cybersecurity tutors. In *Proceedings of the 56th acm technical symposium on computer science education v. 1* (p. 833–839). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3641554.3701801

Petersen, A., Craig, M., Campbell, J., & Tafliovich, A. (2016). Revisiting why students drop cs1. In *Proc. koli calling* (pp. 71–80). doi: 10.1145/2999541.2999552

Phung, T., Cambronero, J., Gulwani, S., Kohn, T., Majumdar, R., Singla, A., & Soares, G. (2023). *Gen-*

erating High-Precision Feedback for Programming Syntax Errors using Large Language Models.

Prather, J., Denny, P., Leinonen, J., Becker, B. A., Albluwi, I., Craig, M., ... Savelka, J. (2023). The robots are here: Navigating the generative ai revolution in computing education. In *Proceedings of the 2023 working group reports on innovation and technology in computer science education* (p. 108–159). New York: ACM. doi: 10.1145/3623762.3633499

Prather, J., Leinonen, J., Kiesler, N., Gorson Benario, J., Lau, S., MacNeil, S., ... Zingaro, D. (2025). Beyond the hype: A comprehensive review of current trends in generative ai research, teaching practices, and tools. In *2024 working group reports on innovation and technology in computer science education* (p. 300–338). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3689187.3709614

Renzella, J., Vassar, A., Lee Solano, L., & Taylor, A. (2025). Compiler-integrated, conversational ai for debugging cs1 programs. In *Proceedings of the 56th acm technical symposium on computer science education v. 1* (p. 994–1000). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3641554.3701827

Riazi, S., & Rooshenas, P. (2025). Llm-driven feedback for enhancing conceptual design learning in database systems courses. In *Proceedings of the 56th acm technical symposium on computer science education v. 1* (p. 1001–1007). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3641554.3701940

Roest, L., Keuning, H., & Jeuring, J. (2023). Next-Step Hint Generation for Introductory Programming Using Large Language Models. In *Proceedings of the 26th Australasian Computing Education Conference* (pp. 144–153). Sydney, Australia: ACM. doi: 10.1145/3636243.3636259

Roethlisberger, F. J., & Dickson, W. J. (1939). *Management and the Worker*. Cambridge: Harvard University Press.

Sarsa, S., Denny, P., Hellas, A., & Leinonen, J. (2022, August). Automatic generation of programming exercises and code explanations using large language models. In *Proc. icer.* ACM. doi: 10.1145/3501385.3543957

Savelka, J., Agarwal, A., Bogart, C., & Sakr, M. (2023). *Large language models (gpt) struggle to answer multiple-choice questions about code.*

Scholl, A., & Kiesler, N. (2024). How novice programmers use and experience chatgpt when solving programming exercises in an introductory course. In *2024 ieee frontiers in education conference (fie)* (p. 1-9). doi: 10.1109/FIE61694.2024.10893442

Scholl, A., & Kiesler, N. (2025a, Jul). *Data: Students' feedback requests and interactions with the script chatbot - do they get what they ask for?* OSF. doi: 10.17605/OSF.IO/TG5R3

Scholl, A., & Kiesler, N. (2025b). Script - supportive chatbot for resolving introductory programming tasks. In *Proceedings of the 30th acm conference on innovation and technology in computer science education v. 2* (p. 759). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3724389.3730786

Scholl, A., Schiffner, D., & Kiesler, N. (2024). Analyzing chat protocols of novice programmers solving introductory programming tasks with chatgpt. In *Proc. delfi 2024* (pp. 63–79). doi: 10.18420/delfi2024_05

Spohrer, J. C., & Soloway, E. (1986). Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM*, *29*(7), 624–632. doi: 10.1145/6138.6145

Szabo, C., Parker, M. C., Friend, M., Jeuring, J., Kohn, T., Malmi, L., & Sheard, J. (2025). Models of mastery learning for computing education. In *Proceedings of the 56th acm technical symposium on computer science education v. 1* (p. 1092–1098). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3641554.3701868

Taylor, A., Vassar, A., Renzella, J., & Pearce, H. (2024). dcc –help: Transforming the role of the compiler by generating context-aware error explanations with large language models. In *Proceedings of the 55th acm technical symposium on computer science education v. 1* (p. 1314–1320). New York: ACM. doi: 10.1145/3626252.3630822

Wermelinger, M. (2023). Using github copilot to solve simple programming problems. In *Proceedings*

of the 54th acm technical symposium on computer science education v. 1 (p. 172–178). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3545945.3569830

Whalley, J., Clear, T., & Lister, R. (2007). The many ways of the Bracelet project. *BACIT*.

Xiao, R., Hou, X., & Stamper, J. (2024). Exploring how multiple levels of gpt-generated programming hints support or disappoint novices. In *Extended abstracts of the 2024 chi conference on human factors in computing systems.* New York, USA: ACM. doi: 10.1145/3613905.3650937

Yeh, T. Y., Tran, K., Gao, G., Yu, T., Fong, W. O., & Chen, T.-Y. (2025). Bridging novice programmers and llms with interactivity. In *Proceedings of the 56th acm technical symposium on computer science education v. 1* (p. 1295–1301). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3641554.3701867

Zhai, X. (2022). Chatgpt user experience: Implications for education. doi: http://dx.doi.org/10.2139/ssrn.4312418

Zhang, J., Cambronero, J., Gulwani, S., Le, V., Piskac, R., Soares, G., & Verbruggen, G. (2022). Repairing bugs in python assignments using large language models. *arXiv preprint arXiv:2209.14876*.