Noname manuscript No.
(will be inserted by the editor)

"Ok Pal, We Have to Code That Now": Interaction Patterns of Programming Beginners with a Conversational Chatbot

Alina Mailach[⋆] · **Dominik Gorgosch**[⋆] ·
Norbert Siegmund · **Janet Siegmund**

Received: date / Accepted: date

Abstract Context: Chatbots based on large language models are becoming an important tool in modern software development, yet little is known about how programming beginners interact with this new technology to write code and acquire new knowledge. Thus, we are missing key ingredients to develop guidelines on how to adopt chatbots for becoming productive at programming.
Objective: With our research, we aim at identifying these ingredients. Specifically, we want to understand how programming beginners use conversational chatbots when writing source code.
Method: To this end, we study programming beginners' interaction with a chatbot in a CS2 course while they were solving programming assignments. Additionally, we evaluate the correctness of submitted solutions and compare them to solutions of beginners who did not use a conversational chatbot.
Findings: We analyzed 756 prompts of 129 conversations, most of them focusing on code generation. Interestingly, conversations that contain prompts asking for debugging or testing of code are linked with higher success rates, indicating that deeper engagement with code leads to higher quality code. Moreover, prompts without sufficient context often lead to unsatisfying results.

[⋆] Both authors contributed equally

A. Mailach
ScaDS.AI Dresden/Leipzig, Leipzig University, Leipzig, 04109, Germany
E-mail: alina.mailach@cs.uni-leipzig.de

D. Gorgosch
Chemnitz University of Technology, Chemnitz, 09107, Germany

N. Siegmund
ScaDS.AI Dresden/Leipzig, Leipzig University, Leipzig, 04109, Germany

J. Siegmund
Chemnitz University of Technology, Chemnitz, 09107, Germany

Implications: While not surprising, this underpins the importance that programming beginners need to know how to use chatbots, instead of merely using it as code generators without investing time in code quality. Moreover, companies should employ prompt guidelines, in which code quality prompts might be enforced after a code generation prompt has been stated.

Keywords AI chatbot · programming beginners · prompting · interaction patterns

1 Introduction

Code assistants, such as Copilot[1], ChatGPT[2], and CodeWhisperer[3] based on *Large language Models (LLMs)* are becoming a central tool in modern software development [20]. The astonishing speed at which this technology is conquering industry leaves many important aspects unanswered. For instance, studies show that the use of such tools may increase security risks [24] or could even degrade developers' productivity despite the hype around them [21].

A further important aspect that has received only little attention are programming beginners. So far, the majority of studies on code assistants and chatbots with human participants either concentrate on professional developers or recruit students for a specific task, such as studying security-related programming tasks [29]. However, surprisingly little is known about *how programming beginners interact with code assistants*, despite some suggestions [3, 18, 6], work in progress [25] and investigations [8, 36, 14] on how AI code generation could transform teaching.

In essence, we do not know typical interaction or conversation patterns that programming beginners apply to solve a given task or whether these patterns differ from professional developers. And, we do not know in which way programming beginners may struggle with this new technology. Do they use chatbots similar to a search engine? Do they search for explanations, or do they just want to generate code to complete a task? And more fundamentally, do chatbots in fact improve programming beginners coding? Answering these questions is not only of paramount importance for new teaching concepts, but also for industry when on-boarding inexperienced developers, be it for novel technologies and languages or for first-time employment. Moreover, career changers and no-code applications are becoming more and more prevalent in industry. It is, thus, necessary to learn more about how these inexperienced new programmers interact with conversational chatbots and how they can be effectively integrated into the development process.

In this study, we set out to answer these questions with a large-scale human study on 2nd semester CS students. Our key research goal is to identify interaction and communication patterns of such programming beginners (i.e., having

[1] https://github.com/features/copilot

[2] https://openai.com/blog/chatgpt

[3] https://aws.amazon.com/codewhisperer/

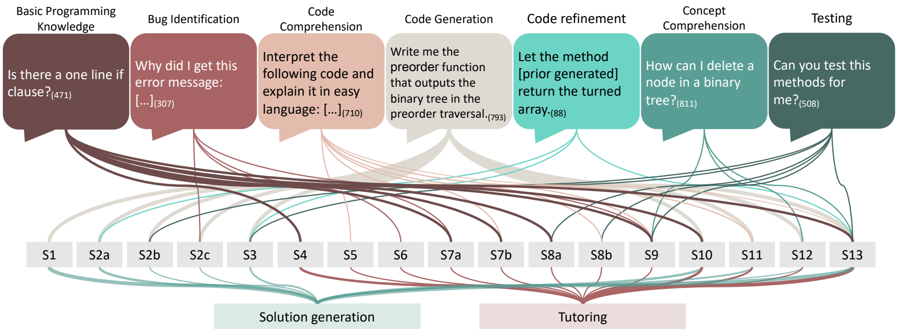

Basic Programming Knowledge
Is there a one line if clause?(471)
Bug Identification
Why did I get this error message: […](307)
Code Comprehension
Interpret the following code and explain it in easy language: […](710)
Code Generation
Write me the preorder function that outputs the binary tree in the preorder traversal.(793)
Code refinement
Let the method [prior generated] return the turned array.(88)
Concept Comprehension
How can I delete a node in a binary tree?(811)
Testing
Can you test this methods for me?(508)
S1
S2a
S2b
S2c
S3
S4
S5
S6
S7a
S7b
S8a
S8b
S9
S10
S11
S12
S13
Solution generation
Tutoring

Fig. 1 Different prompt purposes of programming beginners with a conversational chatbot

already received basic programming classes), as well as possible struggles when using chatbots. In a nutshell, we found that programming beginners who use a chatbot score 21 % higher on implementation tasks than programming beginners who do not use a chatbot. When analyzing further the interactions of participants with the chatbot, we observe distinct conversation structures at different levels of granularity, as shown in Figure 1: At the level of individual prompts, we identified repeating prompting intentions, for example, to generate code or to test code; at the level of entire conversations, we identified distinct structures (labeled S1 to S13) consisting of multiple prompts with the overarching goal to *generate a solution* to complete a task or *tutoring* to gain knowledge (highest level of granularity). Some of these interactions are more successful (e.g., when testing or code refinement is contained), whereas others are less successful for solving a programming task (e.g., when code generation is the only purpose of a conversation).

Such a diversity of usage patterns opens up novel research opportunities for software engineering education and practice: Our study identified code investigation activities as clear indicators of higher performance and code quality, which might help to develop guidelines and tutorials supporting beginners in efficiently utilizing chatbots. By contrast, prompts focused on learning basic programming knowledge and concepts showed lower performance, which indicates potential predictors for an early warning system, identifying struggling beginners and providing tailored support.

In summary, we make the following contributions:

- A detailed study on how programming beginners use LLMs to learn programming and the benefit and drawbacks they have from it.
- A definition of seven prompt purposes when programming beginners solve programming tasks: code generation, code comprehension, concept comprehension, bug identification, testing, and requesting basic programming knowledge.
- An overview of successful conversation patterns that are linked with higher scores on programming assignments in a grading evaluation.

- Causes for struggles when interacting with chatbots, that is, missing knowledge on prompt engineering and missing familiarity with the capabilities of an LLM.
- Identification of a need for teaching prompt design and engineering, corroborating findings of previous studies [32, 16], which count even for professional developers, indicating the impact of our results also for an industry setting.
- We provide material and collected data of our study in a supplementary repository[4].

Our results have impact far beyond the teaching context alone, as they can guide the development of the next generation of conversation-based code assistants. Furthermore, we identified main sources of struggles and key ingredients of successful conversations that are valuable in a professional setting. This way, we can potentially improve developer productivity by recognizing conversation patterns and guiding them toward a successful outcome.

2 Related Work

In the short time period in which LLMs have become powerful and accurate enough to be used in real settings, there have been numerous studies involving human participants to analyze different aspects of LLMs when it comes to programming and software development. We focus on papers with human participants, separated into professional and educational context. In professional contexts, studies evaluate how LLMs affect typical development activities, and in educational context, studies evaluate the impact of LLMs on learning. We can infer and relate insights from both settings to our findings. Finally, we provide a brief comparison of our work to research endeavors for educational chatbots prior to the rise of LLMs.

2.1 Professional Setting

Several studies find that developers evaluate programming assistants positively, including a study by Ross and others, in which developers liked specifically the high quality of responses and the potential impact on their productivity [28]. Vaithilingam and others find that programmers appreciate assistants to provide starting points for development tasks [32]. This observation is included in a grounded theory developed by Barke and others, and is called exploration mode, in which the assistant is used primarily to find the next steps in the implementation [1]. The complementary part of Barke and others' theory describes developers in acceleration mode, in which they already know exactly the following steps, and the assistant is expected to generate what the developer has in mind.

[4] https://github.com/mailach/Ok-pal-we-have-to-code-that-now

Furthermore, Weisz and others report that, for translating code from Java to Python, developers with AI assistance produced code with fewer errors than when working alone [35]. Perry and others report that developers feel confident in producing high-quality, secure code with an LLM [24]. However, the results suggest that developers are *over*confident and put too much trust into the results of the LLM, as they actually produce *less* secure code compared to developers without an LLM. Notably, a group of developers in Perry and others' study invested more time in designing useful prompts and carefully evaluated the responses of an LLM, leading to fewer security vulnerabilities.

This shows that good prompt engineering is necessary, and also further studies provide evidence in this line: Developers who lack prompt engineering skills (in the sense of controlling the output for steering code suggestions for specific functional and non-functional requirements) refrain from using programming assistants [16]. Additionally, there are more reports on negative impacts of LLMs on developers. Mozannar and others report that developers need to spend considerable time in verifying responses of their programming assistants [21], which requires expert knowledge and leads to increased development time. In line with this observation, some developers struggle with editing, debugging, and comprehending the source code generated by programming assistants [32], or are even reluctant to do so, if the generated code does not match the expectation of the developer [1]. This emphasizes that even experts need to familiarize themselves with using programming assistants to accomplish their tasks and need to find suitable ways to integrate them beneficially in their workflows.

For programming beginners, learning programming with a programming assistant might actually add a level of complexity, because they have to learn how to define useful prompts and evaluate the prompts, all while acquiring programming skills; thus, they lack experience to actually understand whether a response is helpful or not, underpinning the necessity and relevance of our study.

2.2 Educational Setting

Closest to our work are studies conducted by Prather and others, who observe how students interact with LLMs in a CS1 course [26], and Kazemitabaar and others, who extract interaction patterns from students using Codex in a self-paced learning environment [12]. Both studies present interaction patterns with LLMs, but from different perspectives. Prather and others focus on students' cognitive difficulties, to which we add insights by identifying *beneficial* patterns and highlighting the need for teaching prompt engineering before using LLMs in classes. Thus, our qualitative analysis extends beyond a teaching context.

Kazemitabaar and others examine interactions with Codex, an LLM that translates natural language into code. While our study aligns with Kazemitabaar's findings on generating and refining code, the vanilla chatbot used by our participants enables a broader range of activities, such as generating explanations for code and concepts. These additional capabilities are reflected in our results, thereby extending the insights presented in these studies.

Looking into benefits of LLM-based tools for students, Choudhuri and others find that students do not benefit significantly from using ChatGPT when solving software engineering tasks. Instead, they find increased levels of frustration, and cases of induced self-doubt [5]. Similarly, Shoufan finds that students without prior knowledge perform equally or worse when using ChatGPT to solve exam-like quiz questions [30]. Our study provides additional insights by taking the conversations of students into account, giving more insights into potential reasons for these observations.

Furthermore, Liu and others [17] explored the use of an LLM-based assistant in an introductory computer science course, similar to our study but with different objectives. They equipped students with a constrained chatbot designed for explaining code, assessing code style, and responding to course-related queries. Unlike in our study, their chatbot was explicitly configured to avoid generating solutions, diverging from the unrestricted, vanilla chatbot we used in our research. Additionally, Liu and others provide their chatbot over the course of a semester for unrestricted use and evaluate it based on student feedback and response accuracy. In contrast, our study was conducted in a controlled setting where the chatbot was only available in in-person tutorials and we collected participants submissions, which allows us to relate different usage patterns to the quality of solutions. Frankford and others have explored how students interact with an AI-tutor that gives feedback, integrated in an automated programming assessment system [10]. Our study, in contrast, provides programming beginners with a chat interface that is not tied to code and can be queried for any kind of questions, meaning students interaction are more varied. This is also reflected in the in-depth interaction patterns yielded by our qualitative analysis.

Notably, Nam and others propose a prompt-less interaction mode within the IDE and find that especially professional developers benefit in task completion. Students, by contrast, did not experience significant benefits from using the LLM-based tool [22]. Our study focuses directly on chat-interface and students interactions, as these interfaces are now freely available for (and are probably used by) students. Other studies also explore CS1 and CS2 problems, but without human participants. Denny and others found that LLMs can solve most of the typical CS1 problems, and a smaller part with some minor modification to the prompts [7]. For typical CS2 problems (similar to our tasks), an LLM can even outperform students in producing correct code [9]. Thus, if correctly used, LLMs can improve productivity. In our study, we focus on qualitatively analyzing how this usage can look in detail. Further studies show that LLMs can explain code in a helpful way depending on code complexity and code length [19], such that even younger programmers (10 to 17 years) can successfully use chatbots with proper guidance [11], and that with using natural-language prompts (in contrast to non-textual prompts or class files), ChatGPT can be useful for programming beginners [23]. While farther away from our focus, these studies still indicate that proper guidance in using a code assistant seems to be key in its successful application. Thus, we cannot just throw a chatbot to programmers and assume they know how to use it properly. While this seems obvious, the reality is that there currently is no real guidance, because we do not know how guidance should look like. With our study, we identify key ingredients for such a guidance.

2.3 Educational Chatbots before LLMs

Prior to the rise of LLMs, several efforts were made to incorporate chatbots in educational environments, particularly in computer science, as evidenced by Kuhail and others meta-study [13]. This literature review reveals that most of these early educational chatbots were primarily chatbot-driven, where the chatbot steers the conversation, often limiting the user to predefined conversation paths or intents. This approach contrasts sharply with LLM-based chatbots, which empower users with complete control over the conversation's direction, depth, and subject.

Closest to our study in this domain is the pilot research conducted by Verleger and others, which examined user interactions with a chatbot based on a question-answer database [33]. Mirroring our methodology, the study analyzes individual prompts (i.e., questions) sent to the chatbot, and finds that students predominantly seek assistance with programming language specifics, including the use of specific methods and basic knowledge on loops, conditions, and algorithms. Unlike LLM-based chatbots such as ours, the chatbot in Verleger and others study is limited to question-answer pairs provided by a human tutor. This restriction narrows the scope of interactions for programming beginners, who state questions beyond the chatbot's pre-defined knowledge base. However, LLM-based chatbots, like ours, respond to all queries, offering users greater flexibility to guide the conversation. Specifically, our study delves deeper into users' interactions at a conversational level, where students make multiple requests. Additionally, we offer initial insights into the disparities between programming beginners who use a chatbot and those who do not, a previously unexplored comparison.

3 Experiment Design

In this section, we explain the details of our experiment and data analysis. All material is available in the project's repository.

3.1 Research Questions

The objective of this study is to understand how programming beginners use a chatbot to expand their knowledge, ask questions, or get assistance with programming tasks. This helps us to understand how the use of chatbots

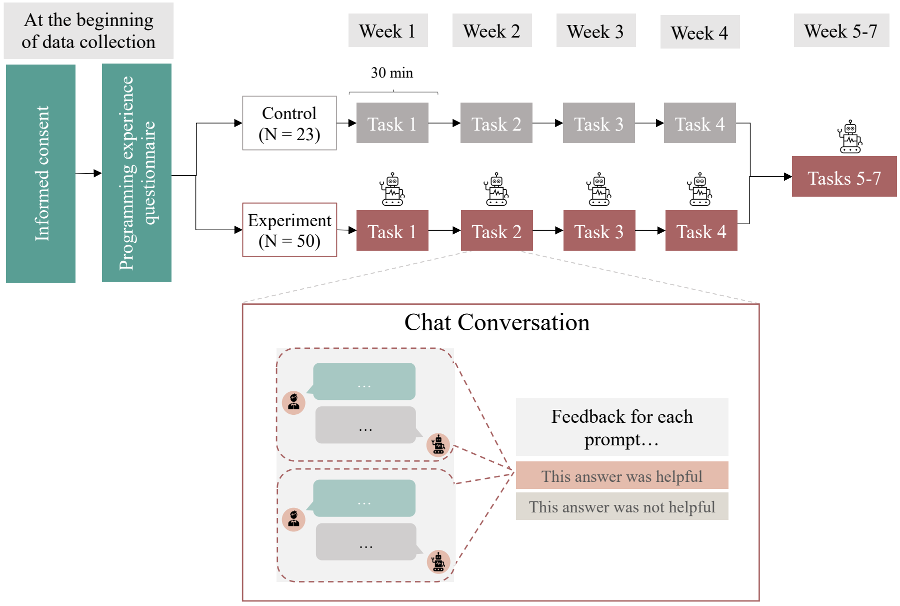


Fig. 2 Study procedure

affects the learning process, and possibly reveals novel interaction styles that are relevant in industry, be it for on-boarding or developers switching to a new programming language or framework. In addition, we aim to extract helpful conversation structures, such that programming beginners can use chatbots more effectively. To this end, we assess the correctness of task solutions, so that we can relate the prompts and conversational structure to programming performance, letting us identify effective patterns of LLM usage. We structure our study around three key research questions in which we examine the overall impact of chatbots on task performance, followed by a systematic analysis of interaction patterns among programming beginners:

RQ$_1$: Can programming beginners better solve programming tasks when using a chatbot?
RQ$_2$: How do programming beginners interact with a chatbot?
RQ$_3$: How do different interaction patterns relate to programming beginners' task performance?

3.2 Procedure

We integrated the experiment in a second-semester programming course at our university, which can be taken by different majors. About 100 students take the course every year, in which basic data structures, such as linked lists and trees, are taught. We use Python and Java in this course. The course consists of a lecture and in-person tutorials, in which students solve tasks with the help of human tutors to deepen their understanding of the course material. We collected data in the first seven weeks of the course. There is no compulsory attendance at our university, which means that students can decide themselves whether they want to visit lectures and tutorials.

We offer six time slots for in-person tutorials. At the beginning of the semester, we assigned each time-slot to either the control or experiment condition, then the students enrolled to one of the time slots. We told students that everyone would use a chatbot during the semester at different times, but did not tell them whether they would enroll in the experiment or control group, meaning that they could not make an educated choice, to which condition they enroll. Our educational system leaves it to students to decide which of the time slots of the tutorials they visit, meaning, after participating in task 1, they could still change the time slot and be in a different condition. However, students only rarely change groups after the first week and no participant changed the group during participation, meaning that a participant was only in one condition. During the first four weeks, students visiting time slots of the experimental condition had access to a chatbot, while the control group had no access. From week five, all participants had access to the chatbot. Figure 2 shows the experimental procedure, including the experimental and control group.

At the beginning of the semester, participants were informed about the goals of the study and how we store and use their data. After we obtained consent for participation, we collected background information and assessed participants' programming experience with an established questionnaire to get an overview of existing programming skills [31]. The prompt and submission data has been collected within the in-person tutorials of the course. Participants in the experimental group received a task description and access to the chatbot, and had 30 minutes to complete the task and submit their solution. Specifically, participants were asked to use the chatbot for any question they encounter during solving a task. We did not provide any further material. For every answer participants received from the chatbot, we collected fine-granular assessments on the usefulness of the chatbot (labeled *chat conversation* in Figure 2). This included feedback regarding completeness, correctness, or precision of the chatbot answer, as well as technical issues. The procedure for the control group was exactly the same, except that the participants were not given access to the chatbot.

In each tutorial, a human tutor was present and explained and clarified all remaining questions of the participants after they submitted their solution. After the the first four weeks, we gave all students of the course access to the chatbot for the rest of the semester, so that also the control group could receive this learning experience.

3.3 Material

3.3.1 Chatbot

To provide participants with access to a state-of-the-art language model and save their interaction data, including their prompts, the response of the chatbot, and the evaluation of the responses, we developed a web application, consisting of a frontend with a chat interface, and a backend that fetches chat completions from Open.AI[5]. We used the LLM *GPT3.5-turbo*, which is recommended for text and code generation because of its lower cost while having similar performance as other models of the GPT3 family. It is also the model used for the free version of ChatGPT, a popular vanilla chatbot. At the time we planned and started our study, this was the most powerful LLM, easily accessible for chat completions.

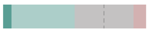
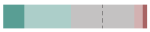
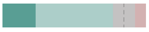
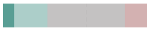
3.3.2 Tasks and Task Performance

We used typical programming tasks to deepen participants' understanding of taught concepts of the course. Each task concerns a different data structure and common algorithms, such as adding an element to a double linked list or determining the height of a tree. We designed the tasks to fit to the knowledge and skill level of students, who all have completed an introductory programming course before joining this CS2 course. The course started with simple data structures, such as arrays and lists, and covered increasingly more complex data structures, including trees and graphs. As the complexity of data structures increased during the semester, so did task complexity. We ensured that a data structure always has been covered in the lecture before students should implement tasks in the tutorial, such that they have a basic understanding of the according data structure.

We measure participants' task performance by the correctness of submitted solutions. In each task, participants should implement a class with specific methods. We assigned one point for each correct method, that is, when the implementation was executable and passed our automated tests. If tests failed, we manually inspected the code. If the failure was due to a naming conflict (such as the wrong naming of a method or attribute), we still rated the implementation as correct. If the failure was due to a small mistake (e.g., returning the entire list instead of the first element), we assigned half a point. In all other cases, the respective method received zero points. For each task, participants could receive between 5 and 6 points. Participants' task performance is then calculated as a score representing the percentage of points achieved. We calculate average scores of task performance across multiple submitted solutions using the arithmetic mean. A complete overview is available at the project's repository.

[5] `https://open.ai/`

Table 1 Participants self-assessed programming experience by study program compared to peers, ranging from much worse to much better, and experience with logic programming, ranging from very inexperienced to very experienced. Within each cell, we provide the color-coded distribution of different answers of participants, alongside the percentage of participants that leaned towards one end of the scale.

Program	*N*	Compared to peers			Logic programming		
Undergraduate	34	50%		9%	47%		9%
Graduate	13	77%		8%	31%		15%

3.4 Participants

73 students from various majors who usually attend the course in their second semester participated. Since participation in the study and in-person tutorial is voluntary due to our local study regulation, not all students of the course participated in the study. At our university, we offer a master's program for students with bachelor's degrees in the humanities or social sciences, so the graduate students have similar levels of experience as the undergraduate students, who are CS majors. This is also reflected in the self-assessed programming experience summarized in Table 1, showing comparable ratings for graduate and undergraduate participants of our study. We assigned participants randomly to the control and experiment group at the beginning of the study. However, due to local laws and regulations, students are allowed to switch tutorial groups, meaning that our initial assignment is not the same as the final control and experiment group. We will provide a discussion on how this threatens the validity of our insights in Section 6.

3.5 Qualitative Analysis Procedure Outline

We analyzed the data at different levels of granularity, starting with examining the *purpose* of individual prompts as the smallest unit of interaction between the chatbot and the participants. Based on the discovered prompt purposes, we categorize whole conversations according to the their *structure* (i.e., sequences of prompt purposes) and *intention* (i.e., the overarching goal of an entire conversation).

As typical for qualitative analysis, a key challenge on all levels of the analysis is to settle on categories with (a) sufficient support and (b) a crisp topic (i.e., not too broad). To this end, we adopted an iterative methodology using a combination of open coding and card sorting. Open coding allowed for the initial identification and labeling of prompt purposes within the data, while card sorting was employed to identify intentions across conversational structures. In the following, we describe this process in detail.

Identifying and assigning prompt-purpose categories To identify different prompt purposes, we first reviewed all prompts manually. Two authors of the paper examined half of the prompts each, using open coding. This resulted in a label for each individual prompt to capture the purpose of the prompt. In some cases, a prompt was too short to clearly understand its purpose. In these cases, it was necessary to review the prompts appearing earlier and later in the conversation for more context. After this initial round of coding, the coders together reviewed the generated codes and clustered similar codes into *prompt-purpose categories*, leading to refined codes.

Next, this prompt-purpose categories were used to review each prompt again. Whenever the refined category did not fully capture the purpose of the prompt, the prompt was marked. These cases were further discussed between the two authors. When they could not reach inter-personal consensus on how to assign a category to the prompt, they gradually refined the categories. Finally, one of the authors reviewed all prompts again to ensure that already categorized prompts still fit into the categories.

Clustering prompts into conversational structures To identify *conversational structures*, we group prompts within each conversation into sequences of the same purpose. For instance, if a programming beginner first used two prompts with purpose A, followed by one prompt with purpose C, and finally three prompts of purpose B, the structure of the conversation would be A-C-B. Finally, we use an open card sorting approach to find a meaningful clustering of these conversational structures, called *conversation intentions*, that is, they represent the underlying goal of the conversation.

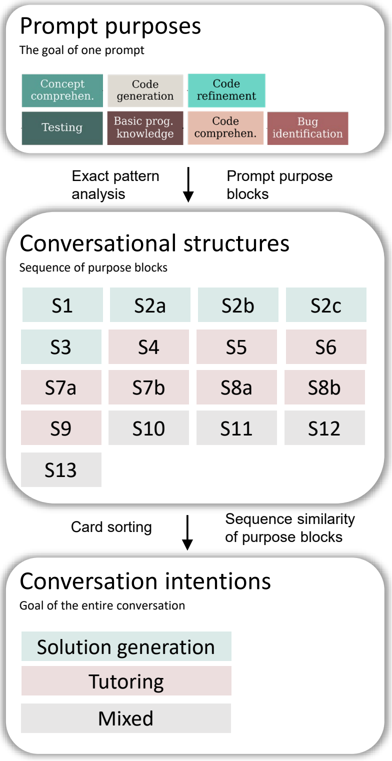


Fig. 3 Structure of results analysis.

4 Results

Having described our analysis plan, we now dive into the results. To systematically explore our large amount of qualitative data, we structure them on three levels of granularity, as shown in Figure 3. On the lowest level, we look at individual prompts, identifying seven *prompt purposes*. Next, on a higher granularity level, we analyze entire *conversational structures* by grouping sequences of prompts with the same purpose in a single block (e.g., multiple prompts of code generation form a single block). A conversational structure is, thus, composed of multiple blocks with distinct purposes. We aim to find typical, possibly repeating structures in conversations and align them for RQ_3 with the achieved scores of corresponding submissions. Finally, on the highest level, we categorize conversational structures according to participants' *conversational intention* via card sorting.

We start with a quantitative overview of our results and our conversation data to put our insights into context. Next, we provide a comparison of the quality of task solutions of programming beginners with and without a chatbot. Afterwards, we describe our findings for the different levels of granularity. For each research question, we first provide the results, followed by an interpretation and discussion for each research question.

4.1 Descriptive Overview

Submission statistics In total, we received 139 submissions, 103 from the experiment and 36 from the control group. Across all submissions, programming beginners score 45.76 points on average, with a standard deviation of 36.02 points, indicating a large variety. The *score* of a submission represents the percentage of correct points for each task (i.e., a score of 100 means that all points have been awarded for a submission).

Prompts and chat conversations statistics In total, we analyzed 756 prompts in 129 conversations from the experiment group. We started with 50 participants using the chatbot for the first study task and ended with five for the last (task 7). This drop-out rate is common for our education system, as there is no obligation to attend tutorials or lectures, yet we still collect interesting insights. However, we do not analyze the prompts in a within-subject style across multiple conversations per participant and rather provide a general analysis of all individual conversations.

We excluded prompts without a clear goal, for example, when participants simply greeted, thanked, or insulted the chatbot, leading to the exclusion of 51 prompts. This way, we ensure that we have only genuine prompts in our data. Furthermore, in 43 cases, participants sent incomplete prompts to the chatbot, for instance, when pasting code but forget to add a question. The subsequent prompt often contains the missing question, so we analyze these successors and exclude the incomplete prompts.

Table 2 gives an overview of the number and length of prompts and conversations per task. On average, a conversation consists of 5.8 prompts, each of which containing 30.6 words.

Helpful and not helpful prompts After each answer of the chatbot, we asked for feedback on the helpfulness of the answer. In total, participants rated 90.6 % of the prompt-answer pairs as helpful. Less than 1 % of the answers were rated as non-helpful due to a technical problem. In all other cases, participants rated the answer of the chatbot as wrong, incomplete, unclear, or incomprehensible.

Table 2 Number of conversations ($\#_{\text{conv}}$) and prompts ($\#_{\text{prompt}}$), mean words ($\overline{W}_{\text{prompt}}$) per prompt and answer ($\overline{W}_{\text{answer}}$) by task, and standard deviation of words per prompt (SD_{prompt}) and answer (SD_{prompt})

Task	$\#_{\text{conv}}$	$\#_{\text{prompt}}$	$\overline{W}_{\text{prompt}}$	SD_{prompt}	$\overline{W}_{\text{answer}}$	SD_{answer}
1	51	325	22.1	42.5	196.8	138.5
2	26	126	45.3	123.2	204.2	161.0
3	18	119	27.1	45.8	335.7	274.0
4	21	113	33.9	67.2	189.4	124.0
5-7	13	73	43.3	109.2	244.2	264.6
all	129	756	30.56	74.2	223.4	189.6

Participants' programming experience To assess the impact of prior programming experience, we provide self-assessed experience scores in all tables (column $\overline{M}_{\text{xp}}$). Experience scores represent participants' averaged programming experience compared to peers, on a scale from 0 (much worse) to 4 (much better). A score of 2 indicates perceived equivalence to peers, while higher values denote higher self-assessed programming experience.

In the experiment group, participants' experience scores were consistently similar across the various interaction patterns assessed for both RQ_2 and RQ_3. Therefore, the experience level has no effect on the interaction patterns of participants, so we omit a detailed discussion on this aspect for RQ_2 and RQ_3.

4.2 RQ_1: Can programming beginners better solve programming tasks when using a chatbot?

To answer RQ_1, we analyze how the task performance of programming beginners using the conversational chatbot differs from those in the control group. Table 3 shows the correctness of programming beginners' submitted solutions split by task and experimental condition, along with averaged experience values and the number of submissions. Across all tasks, programming beginners using the chatbot achieved a mean score of $\overline{M}_{\text{score}} = 47.91$, which indicates an overall better solution quality in the experiment group compared to the control group, who averaged $\overline{M}_{\text{score}} = 39.58$. This is a 21.0 % higher score of the chatbot-assisted beginners compared to those without chatbot assistance. Figure 4 shows the distribution of correctness values in the two groups and provides a more nuanced picture. Notably, programming beginners who used a chatbot exhibit the typical bimodal distribution of skills that is often observed in introductory programming courses [27]. The chatbot might specifically help some students in creating better solutions, while others solve the task similarly or worse than those programming beginners in the control group.

Consistent with the general trend, we find that the experiment group achieves higher scores in the first three tasks. Only within task 4, participants in the control group considerably outperform those using the chatbot. Despite better scores, the experience within the experiment group is lower

Table 3 Task performance between participants in the experiment group ("Chatbot") and the control group ("Control"). We report the mean score $\overline{M}_{\text{score}}$, mean experience $\overline{M}_{\text{xp}}$ (programming ability compared to peers 0: Much worse; 1: Worse, 2: Equal, 3: Better, 4: Much better), and number of participants who submitted a solution #. Further, we report absolute and relative difference $\overline{M}_{\text{diff}}$ between the means of the two groups.

	Chatbot			Control			
Task	$\overline{M}_{\text{score}}$	#	$\overline{M}^{*}_{\text{xp}}$	$\overline{M}_{\text{score}}$	#	$\overline{M}^{*}_{\text{xp}}$	$\overline{M}_{\text{diff}}$
1	51.78	45	1.46	37.37	19	1.81	14.41 (38.6%)
2	26.39	24	1.67	10.00	5	1.60	16.39 (163.9%)
3	46.35	16	1.33	40.62	8	1.60	5.73 (14.1%)
4	68.33	18	1.35	85.00	4	2.00	-16.67 (-19.6%)
all	47.91	103	1.43	39.58	36	1.82	8.33 (21.0%)

*Average over 95 (Chatbot) and 29 (Control) submissions with available self-assessment data.

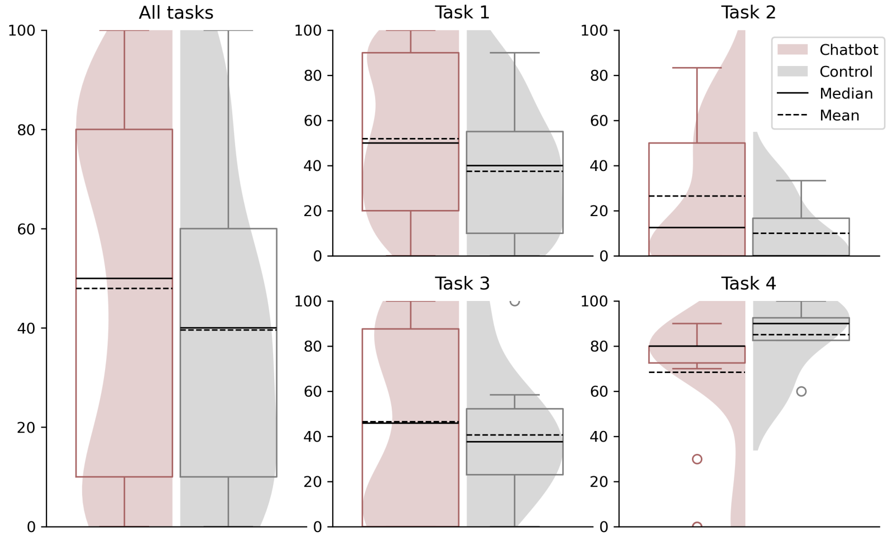


Fig. 4 Distribution of task performance between participants in the experiment group ("Chatbot", light red) and the control group ("Control", gray).

than in the control group for all tasks but task 2. However, the differences in experiences are small, except for task 4, for which only four individuals submit a solution in the control group.

Result I: Some programming beginners using the chatbot outperform those who do not, in three of four tasks. However, for some beginners this is not the case, and despite having access to the chatbot, they do not genereate better solutions than programming beginners of the control group.

4.2.1 Discussion of RQ_1

To answer whether programming beginners can better solve programming tasks when using a chatbot: Yes, our study shows that at least some programming beginners better solve programming tasks with a chatbot. Although we could not fully randomize the assignment of programming beginners to the experimental and control group, and despite a tendency for lower experience values in the experimental group, we find that there is a subgroup of programming beginners who perform better when using a chatbot than those who do not. By contrast, we observe that some students do not benefit in the same way. One possible explanation is, that students who successfully use the chatbot simply copy and paste code. However, if that was the case, we could expect a higher solution quality, especially since LLMs are capable of solving CS2 tasks [9]. Another interpretation is that the chatbot's interactive and personalized learning approach complements existing programming skills of some beginners and potentially compensates for their initial lower experience levels. This might be due to a more engaging and responsive environment, allowing beginners to receive immediate feedback and explanations compared to the use of traditional learning methods (e.g, textbooks, web pages, or human tutors). The bimodal distribution in the group of chatbot users' might also be a sign of negative effects of frustration and self-doubt that might occur in programming beginners [5]. One explanation for our results is that the negative effects of chatbots affect some programming beginners more than others, leading to lower solution quality.

It is also noteworthy that programming beginners of the control group outperform those using the chatbot in the fourth task. However, since the number of participants is rather low for this specific case (only 4), and these participants have a higher average programming experience, we rather assume that only skilled participants of the control group kept on going with the course and submitted solutions, and this might be the major reason for the higher performance (not the usage of a chatbot). This observation leads to an interesting outlook worth further investigations: Chatbots might lower the barrier for low-skilled participants to submit solutions and actively take part in a course. So, chatbots might be a tool to increase engagement. However, future studies are needed to evaluate this interpretation.

As a final note, one must not confuse task performance with learning success. We cannot state that more accurate submissions of the chatbot group also lead to a higher learning success, since the 30-minute session had the focus of solving programming tasks. aMeasuring learning success should entail different measures, such as exams. Local regulations do not allow for a mandated participation in tutorials. To measure learning success, controlled longitudinal studies in which programming beginners are more free to use chatbots outside of an restricted environment as in our study, could be useful. However, in an educational setting (such as a university courses) such studies have severe ethical implications, as long as we cannot be sure whether using the chatbot

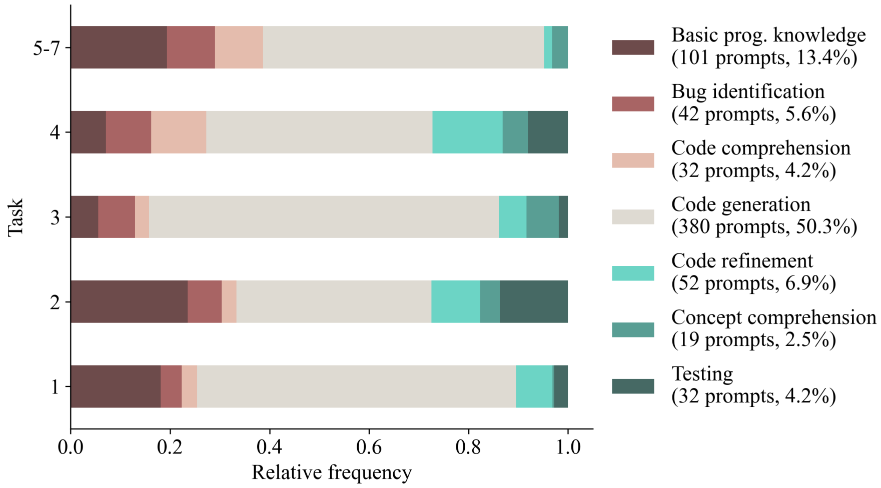


Fig. 5 Distribution of prompt purposes by task

does more harm than good. Our study is an important piece of the puzzle to dispel such concerns and enable such controlled longitudinal studies.

To answer RQ_1, we found that the usage of a vanilla chatbot helps *some* programming beginners to deliver better solutions for given tasks. This underpins the potential chatbots hold for educational and industrial purposes, in effectively aiding programming beginners to producing more correct solutions.

4.3 RQ_2: How Do Programming Beginners Interact with a Chatbot?

To address RQ_2, we present results at various granularity levels, beginning with the purposes of individual prompts and extending to the overarching intentions of entire conversations.

4.3.1 Prompt Purposes

Analyzing the individual prompts used by programming beginners in the experimental group resulted in seven prompt purposes, each of which occur in multiple conversations. We give an overview of all purpose categories of prompts and example prompts in Figure 1. We show the relative distribution of purposes per task in Figure 5. The major purpose category is code generation across all tasks, followed by the request of basic programming knowledge. Next, we explain each purpose in detail and provide example prompts to illustrate our findings. For each prompt, we include a number in brackets that uniquely identifies the prompt within the published dataset.

Code generation For half of all prompts, the primary purpose is to receive generated code. Thus, the focus lies on implementing a particular task rather than understanding the underlying concept, which counts for all tasks, even though for Tasks 2 and 4, this is a bit less pronounced. Task 2 shows the smallest fraction and is centered on object-orientated concepts, such as class structures, where participants are provided with pre-defined classes distributed across different files to integrate their solutions. In the context of our study, one possible explanation is that the necessity to harmonize the generated code with existing source files renders straightforward code generation prompts less effective compared to tasks in which participants work with fewer or no additional source files.

In typical code generation prompts, participants asked the chatbot to "*Create a class for me where accounts with people can be created*" (499), "*Write me a Java code that reverses a list and returns it.*" (9), "*Write a Python code for the following task [pasted given task]*" (200), or "*find duplicate*" (289). Participants frequently copy and paste the given tasks directly into the chatbot, as seen in prompt 200. Alternatively, some participants opt to rephrase the task in their own words, like in prompts 9 and 499. While there are instances of precise task summarizations, more often, participants' prompts lack essential implementation details, with some reducing the task to just keywords or method names, a trend highlighted by prompt 289, in which the task was to implement a method to find a duplicate in a given array and return it. Consequently, the chatbot's responses are often solely based on incomplete or inaccurate information. This indicates that programming beginners are not aware of what information a chatbot requires to effectively solve a task. This is an interesting result, as it points to a mental gap of what chatbots can technically do and what they are expected to do.

Code refinement Beyond simple requests for code generation, 52 prompts asked to refine existing code. With these prompts, the programming beginner requested help in improving efficiency, readability, or overall quality or functionality of their code (either provided or generated in a prior prompt). These inquiries aim at optimizing the code implementation to enhance its maintainability and performance. For instance, a participant asked to change the implementation of an existing method, such that "*The inversion still seems complicated to me, can it be simplified?*" (731). Notably, there is an increase in code refinement prompts in tasks where code generation prompts are less frequent, particularly in tasks 2 and 4. This pattern may reflect a lower satisfaction with the initially generated code. A potential cause for this pattern might be that programming beginners refine their code, because they struggle with integrating their solution within a more complex environment of pre-existing classes, as observed in Task 2.

Result II: In the majority of prompts (50.7 %), programming beginners request the generation of code to solve a given task, but partially lack understanding of what information the LLM needs to solve the task. Refinement of existing code by prompting for specific changes and adjustments is only of limited use (6.9 %).

Code comprehension In 32 cases, prompts contain a question to explain specific lines or sections of a given piece of code. These inquiries aim at gaining a deeper understanding of how certain code segments function or how they contribute to the overall program. For instance, a participant asked for an explanation of the code in simple language: "*Interpret the following code and explain it in simple language*" (710). While this participant asked for a general explanation, some programming beginners had very detailed clarification requests, such as "*what does this part do: if current_node.next: current_node.next.prev= new_node*" (600) or "*what does this @override do with this code from the class double linked list*" (493). We find that the frequency of programming beginners asking for explanations of code increases with more advanced tasks. One interpretation for this pattern is that, as tasks become more sophisticated, programming beginners are increasingly seeking clarity on code implementation.

Concept comprehension In 19 prompts, participants asked for assistance in understanding a certain concept independent of a specific implementation. Hence, this purpose category focuses on gaining a deeper understanding about basic concepts rather than specific code details. For instance, one participant asked broadly "*What is a double linked list[?]*" (519), and another participant wanted to know "*What is a queue in programming[?]*" (640). Participants in our study prompted for explanations of concepts across all tasks with the exception of the first task and with high occurrences in task 2, 3, and 4. This trend could suggest that participants faced greater challenges with advanced data structures, such as Linked Lists and Queues, introduced in later tasks of the course, while requiring less assistance with conceptualizing simpler structures, such as Arrays, which had been the focus of Task 1. However, this is an interpretation and we need further research to evaluate whether it holds.

Basic programming knowledge In 101 prompts, programming beginners requested information on basic programming knowledge, including syntax, rules, or specific features of the programming language. For example, participants asked "*How do I use modulo in Python?*" (126), "*What arguments does .pop() have [in] Python[?]*" (512), and "*How to get [the] length of list in Python[?]*" (75). Most of these prompts relate to questions that belong to a beginner's coding tutorial or looking up functionalities in the API documentation. In some cases, participants' prompts demonstrate a need for assistance with transferring concepts across languages. For instance, in this prompt a participant is inquiring about initializing a fixed-size list in Python, akin to an array declaration in other languages: "*Empty list with 5 entries in Python?*" (150).

Result III: The second largest number of prompts regarding programming knowledge (13.4 %) indicates a considerable use case beyond code generation: LLMs might replace API look-ups and ease transfer learning of new programming languages.

Bug identification In 42 prompts, participants encountered issues in their code and required assistance in resolving them. They either provided a piece of their own code or refer to previously generated code. Often, participants provided error messages with an explanation request, for example: "*Why did I get these error messages: [copy-paste of error messages]*" (307). In addition, participants have passed code to the chatbot and asked "*[pasted code] What is not working here[?]*" (630).

Testing In 32 prompts, the purpose was related to testing, specifically, searching for guidance on how to conduct tests or verify the correctness of code, which included self-written or generated code. These inquiries focus on ensuring the functionality and reliability of code. Prompts usually contain a request for help with testing code: "*Give me another function so that I can test the code*" (193) or "*Write a test case for the complete code*" (133). In three conversations, we find that programming beginners ask the model to test the code for them directly or checking for compliance with the task description: "*Can you test those methods for me?*" (508) or "*This [code] works with this task, doesn't it? [copy paste of the task description]*" (734).

Result IV: Programming beginners use the chatbot to assure code quality by requesting help to find bugs or test code (9.8%). Explaining error messages seems to be a key usage scenario of chatbots.

4.3.2 Conversational Structures and Intentions

Having analyzed the prompt level, we now look at conversation structures, that is, sequences of prompts.

In total, we found 13 distinct conversational structures, that is, sequences of purpose blocks. We give an overview in Table 4, sorted according to their respective intention, structure complexity, and number of conversations in which they appear. Each structure has a unique identifier by which we will refer to it. In some cases (e.g., S2, S7), structures start with the same purpose, but continue with slight variations, such as different combinations of purpose blocks. Notably, 85 conversations (71.4 %) show simple structures, consisting of one or two distinct prompt purposes. Only 34 conversations (28.5 %) show more complex structures, in which prompts had more than two distinct purposes. These complex structures (S3, S9, S13) contain the same purpose, but with variations in their order.

With open-card sorting (cf. Section 3), we categorized the conversational structures further according to their overarching intention (left column of Table 4), that is, generating a solution, tutoring, or a combination thereof. These intentions describe meaningful usage scenarios for programming beginners and represent the highest granularity level of our analysis. Furthermore, Figure 1

Table 4 Conversational structures and corresponding task performance grouped by the two conversational intentions (solution generation and tutoring). We report the mean score $\overline{M}_{\text{score}}$ (colored according to its rank), mean experience $\overline{M}^{*}_{\text{xp}}$ (programming ability compared to classmates 0: Much worse; 1: Worse, 2: Equal, 3: Better), and number of conversations $\#_{\text{conv}}$. Dashed boxes represent sets of conversational structures with varying order of prompt purposes.

			Conversational Structure	$\#_{\text{conv}}$	$\#_{\text{sub}}$	$\overline{M}^{*}_{\text{xp}}$	$\overline{M}_{\text{score}}$
Solution generation				54	46	1.52	53.26
	S1		Code generation	28	22	1.17	44.24
	S2	a	Code generation → Code refinement	15	13	1.85	45.51
		b	Code generation → Testing	3	3	1	80.56
		c	Code generation → Bug identification	1	1	2	80.00
	S3		Code generation → Code refinement → Testing → Code generation → Bug identification	7	7	2	80.48
Tutoring				23	19	1.06	35.79
	S4		Basic prog. knowledge	10	9	1.13	21.67
	S5		Code comprehen.	2	1	1	70.00
	S6		Bug identification	3	2	1	44.17
	S7	a	Basic prog. knowledge → Bug identification	2	2	1	40.00
		b	Basic prog. knowledge → Code comprehen.	2	1	1	30.00
	S8	a	Basic prog. knowledge → Testing	1	1	0	66.67
		b	Testing → Code comprehen.	1	1	2	83.33
	S9		Concept comprehen. → Code comprehen. → Basic prog. knowledge → Bug identification → Basic prog. knowledge	2	2	1	33.00
Mixed				42	37	1.6	48.71
	S10		Basic prog. knowledge → Code generation	10	9	2.13	35.74
	S11		Code generation → Code comprehen. → Code generation	5	4	1.5	65.83
	S12		Concept comprehen. → Code generation → Concept comprehen.	2	2	2.5	56.67
	S13		Basic prog. knowledge → Code generation → Code refinement → Testing → Code comprehen.	25	22	1.33	47.12
				119	102	1.47	46.89

*Average over 95 submissions with available self-assessment data.

shows how individual prompt purposes are associated with different conversation intentions. About two thirds of conversations can be unambiguously assigned to one of two conversational intentions, that is, *solution generation* or *tutoring*. The remaining conversations show signs of both intentions (*mixed*) and we categorize them accordingly as *mixed*. We discuss the conversational structures along these intentions next.

Conversation intention: Solution generation Conversation structures with the intention of solution generation primarily contain prompts for code generation, making up an entire conversational structure in 51.85 % (28/54) of the cases. In other words, there are sequences of prompts that generate code (S1), but do not build on each other (i.e., do not refine existing code) or ask for subsequent development activities, such as testing or debugging. Although the prompts might target different parts of the same task, each prompt generates a new, distinct piece of code. By contrast, in conversations that follow structure S2a, a piece of code is iteratively refined. In many conversations, this pair of purpose blocks (i.e., generation and refinement) is used repeatedly, so code is generated and refined in multiple steps.

Structures S2b and S2c also consist of two purposes, and several conversations contain more than two purposes (S3). They hint at more sophisticated solution-generation structures, such that code generation (and refinement) is followed by prompts concerning the correctness of code. This more advanced use of chatbots potentially leads to higher code quality, and shows that programming beginners actively engaged with both, the generated solutions and the correctness of code.

Result V: With 45.38 %, solution generation is the most prevalent conversation intention. Notably, in most of the conversations, programming beginners are generating code without any other engagement with the chatbot. Only 20.37 % of conversations that target code generation ask for help with debugging or testing the generated code.

Conversation intention: Tutoring Several conversations exhibit the intention to learn, but without requesting code generation. The conversations exhibit an educational and exploratory character, similar to the interactions between a student and a tutor. Some of the prompts used for tutoring can be associated with the generation of a solution, such as the prompt for one-line if-clauses in Figure 1. However, the primary distinction is that tutoring prompts aim to assist students in practicing coding on their own independent of the given task, whereas solution generation prompts indicate a request for the chatbot to directly produce the solution.

Again, we found that conversations containing repetitions of the same prompt purposes make up the majority of conversations (65.23 %; 15/23). Programming beginners ask for basic programming knowledge (S4), code explanations (S5), or help in identifying bugs (S6). By contrast, structure S7 represents a conversation that starts with requesting basic knowledge and is followed by subsequent prompts for understanding (lines of) code or identifying bugs. So for such structures, programming beginners first search for the knowledge they need to solve a task on their own, and then further request help when they encounter errors (S7a) or ask for explanations of code they do not fully understand (S7b). Furthermore, we found that requests for help with testing occur with requests for basic programming knowledge (S8a) and explanations of code (S8b).

Result VI: In 19.33 % of the conversations, programming beginners use the chatbot as a coding tutor, assisting in the creation of a solution, helping with general knowledge, debugging, or testing.

Mixed conversational intentions In about one third of the conversations, participants used the chatbot for both, as tutor for learning and for generating a solution. In simple conversations, programming beginners request basic programming knowledge and the generation of code (S10) within the same conversation (23.81 %; 10/42) or generate code before asking for explanations of code (S11), or ask for explanation of concepts before generating code (S12).

Most of the conversations have an advanced structure, containing different purposes to provide basic knowledge, generate, refine, and test code, indicating an advanced process to understand concepts and generate solutions.

Result VII: 35.29 % of the conversations show mixed intentions, in which programming beginners use the chatbot to generate code for solving a task, but also request explanations and knowledge in a tutoring style.

4.3.3 Discussion of RQ_2

We identified a large variety of usage and communication patterns, despite the homogeneous group of participants and the focused programming tasks.

Our study was designed to submit code solutions to programming tasks, which represents an ideal case to use the chatbot for code generation. As such tasks are common in CS education, other investigations raised concerns about the viability of current typical programming tasks for CS students in the presence of chatbots [7,9]. Our observations might weaken such concerns: We found that almost a fifth of participants used the chatbot solely for tutoring without prompting for a solution. Since the solutions were ungraded, students may feel more comfortable in such a setting for using a chatbot for other, educational use cases. So, educators might provide chatbots especially for voluntary tasks.

While we find some interesting insights in the distribution of specific prompt purposes over time, such as more requests for code explanations in more sophisticated tasks or less code generation in tasks targeting object oriented programming, we must be careful with over-interpreting them. It is possible that these observations are due to the specific challenges of the tasks or effects over time, such as participants becoming more skilled in different areas. Nevertheless, these insights are potential starting points for future research endeavors, in which programming beginners are more closely accompanied in longitudinal within-subject studies.

Furthermore, the results indicate that programming beginners often neglect or are unaware of prompt engineering best practices, such as chain-of-thought [34] or few-shot prompting [4]. Moreover, we identify several interactions that show a lack of understanding of how LLMs work, such as prompting the chatbot to generate code without providing a clear description of the functionality that should be implemented. This gap in knowledge can result in inadequate context provisioning and an increased risk of reliance on incorrect or subpar responses, a previously identified concern when using AI assistants [24]. To effectively integrate chatbots in environments with programming beginners, whether in education or industry, it is, thus, essential to ensure that these beginners clearly comprehend the information required by the chatbot and how they work. Teaching prompt engineering therefore might be a practicable way to help beginners getting started with using chatbots for code generation more effectively. This can be interesting in industrial settings, in which beginners and career changers have to get up to speed quickly, but requires field studies to evaluate its effectiveness. The absence of suitable prompts to generate code might not only be an indicator for missing skills on how to use a chatbot, but could more generally indicate a lack of problem solving skills. While this is an interesting direction for future studies, we believe that teaching problem solving skills and prompt engineering simultaneously might be a promising direction, as generating a good prompt by explaining how a problem was solved before, dissecting it into chunks that are more easy to solve, or eliciting reasoning, can be very similar to understanding the problem itself.

In summary, our results show the clear emergence of specific prompt purposes, which are highly relevant for designing future user studies in an industry and educational context. Our study provides a vital step for researchers to design studies in the future that take the found insights, purposes, and structures into account.

> To answer RQ$_2$, we found diverse prompts and conversation structures, as well as different intentions of conversations. This diversity is surprising, considering the focused tasks. The identified purposes and intentions can guide future research on chatbot design to develop tailored chatbots for programming beginners. In addition, our results underscore the necessity for programming beginners to understand the fundamental workings of LLMs and best practices of prompt engineering for effective chatbot use.

4.4 RQ$_3$: How Do Different Interaction Patterns Relate to Programming Beginners' Task Performance?

Having identified typical interaction patterns on different levels of granularity, we now investigate how they relate to participants' performance in the given

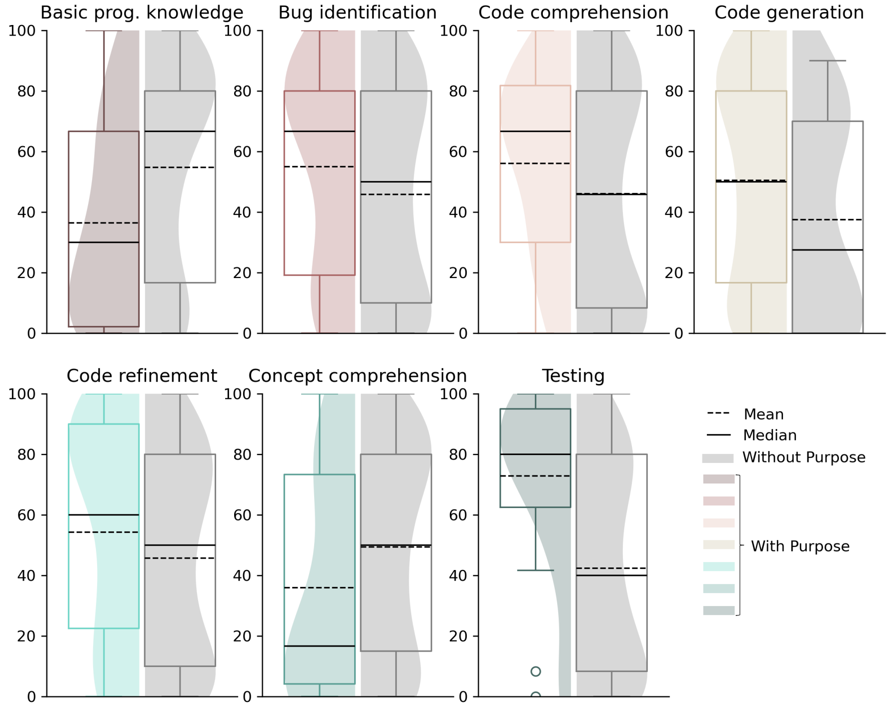


Fig. 6 Distribution of task performance between participants who use a prompt purpose ("With", colored) and those who do not ("Without", gray)

tasks, so we can answer RQ_3. First, we divide participants by prompt purposes and compare their performance to discern trends. Then, we investigate how different conversational structures and intentions are associated with higher or lower task performance of participants.

4.4.1 Task Performance by Prompt Purpose

Table 5 depicts participants' task performance split by prompt purposes and task. To this end, we split participants' submitted solutions into two parts: Solutions of those who used a prompt with the specific purpose in their conversation with the chatbot (column *With purpose*), and solutions of those who did not (column *Without purpose*). In 38 conversations with the chatbot (column #), participants used at least one prompt with the purpose basic programming knowledge. These participants score, on average, 36.36 (column $\overline{M}_{\text{score}}$). Conversely, in 65 conversations, participants did not use prompts querying basic programming knowledge and achieve an average score of 54.67. This represents a difference of -18.31 (-33.5 %) between the two groups, and means that participants who did not ask for basic programming knowledge perform substantially better than participants who did ask for basic programming

Table 5 Prompt purposes and corresponding task performance. We report the mean score $\overline{M}_{\text{score}}$ and absolute and relative difference $\overline{M}_{\text{diff}}$ from the mean score of all submissions. Further, we report mean experience $\overline{M}^{*}_{\text{xp}}$ (programming ability compared to peers 0: Much worse; 1: Worse, 2: Equal, 3: Better, 4: Much better) and number of participants who submitted a solution #.

		With purpose			Without purpose			
Purpose	Task	$\overline{M}_{\text{score}}$	#	$\overline{M}^{*}_{\text{xp}}$	$\overline{M}_{\text{score}}$	#	$\overline{M}^{*}_{\text{xp}}$	$\overline{M}_{\text{diff}}$
Basic prog. knowledge	1	42.11	19	1.41	58.85	26	1.50	-16.74 (-28.4%)
	2	19.44	12	1.40	33.33	12	1.91	-13.89 (-41.7%)
	3	52.78	3	1.33	44.87	13	1.33	7.91 (17.6%)
	4	47.50	4	1.25	74.29	14	1.38	-26.79 (-36.1%)
	all	36.36	38	1.35	54.67	65	1.52	-18.31 (-33.5%)
Bug identification	1	51.25	8	1.43	51.89	37	1.47	-0.64 (-1.2%)
	2	25.00	6	1.80	26.85	18	1.62	-1.85 (-6.9%)
	3	70.00	5	1.40	35.61	11	1.30	34.39 (96.6%)
	4	82.00	5	1.40	63.08	13	1.33	18.92 (30.0%)
	all	55.00	24	1.53	45.76	79	1.41	9.24 (20.2%)
Code comprehension	1	48.57	7	1.14	52.37	38	1.53	-3.8 (-7.3%)
	2	54.17	2	2.00	23.86	22	1.63	30.3 (127.0%)
	3	72.22	3	1.00	40.38	13	1.42	31.84 (78.8%)
	4	57.14	7	1.17	75.45	11	1.45	-18.31 (-24.3%)
	all	56.05	19	1.24	46.07	84	1.49	9.98 (21.7%)
Code generation	1	53.85	39	1.54	38.33	6	1.00	15.51 (40.5%)
	2	28.33	15	2.00	23.15	9	1.00	5.19 (22.4%)
	3	46.43	14	1.31	45.83	2	1.50	0.6 (1.3%)
	4	67.33	15	1.40	73.33	3	1.00	-6.0 (-8.2%)
	all	50.42	83	1.50	37.50	20	1.07	12.92 (34.5%)
Code refinement	1	72.73	11	1.70	45.00	34	1.39	27.73 (61.6%)
	2	25.00	5	2.00	26.75	19	1.56	-1.75 (-6.5%)
	3	25.00	4	1.75	53.47	12	1.18	-28.47 (-53.2%)
	4	62.86	7	1.71	71.82	11	1.10	-8.96 (-12.5%)
	all	54.26	27	1.72	45.66	76	1.41	8.6 (18.8%)
Concept comprehension	1	10.00	1	1.00	52.73	44	1.48	-42.73 (-81.0%)
	2	13.89	3	2.00	28.17	21	1.61	-14.29 (-50.7%)
	3	61.11	3	1.33	42.95	13	1.33	18.16 (42.3%)
	4	40.00	4	1.50	76.43	14	1.31	-36.43 (-47.7%)
	all	35.91	11	1.50	49.35	92	1.44	-13.44 (-27.2%)
Testing	1	88.33	6	1.33	46.15	39	1.49	42.18 (91.4%)
	2	55.56	6	1.67	16.67	18	1.67	38.89 (233.3%)
	3	100.00	2	2.00	38.69	14	1.23	61.31 (158.5%)
	4	64.00	5	1.60	70.00	13	1.25	-6.0 (-8.6%)
	all	72.81	19	1.50	42.28	84	1.46	30.53 (72.2%)

*Average over 95 submissions with available self-assessment data.

knowledge. We provide the distribution of the performance values by prompt purpose in Figure 6. In line with the averaged value, we observe that the distribution of performance values associated with basic programming knowledge is skewed, with most values accumulating at the lower end of the performance scale. This is an interesting and possibly highly impactful result: Essentially, it indicates that students who query for basic knowledge are unfit for solving the task such that these types of prompt purposes might act as early warning signals for individual students and supervisors alike. In the following, we focus on the most interesting observations and refrain from describing each prompt purpose in detail, as this might be repetitive and not much to learn from. Note that the number of participants who use a prompt with a specific purpose and those who do not, often considerably differs, so we must not over-interpret any results.

Overall, we observe for five of the seven prompt purposes higher average performance when used, compared to when not. This difference is most pronounced for participants who use a prompt for testing (~72 %), followed by prompts for code generation (~35 %), code comprehension (~21 %), bug identification (~20 %), and code refinement (~19 %). Notably, these purposes focus on the code to solve a given problem, including a deeper investigation into code artifacts with the goal of either understanding or improving the code.

The two prompt purposes that are associated with low performance are concept comprehension and basic programming knowledge, yielding, on average, a 35.91 %/33.5 % lower score than the overall average of all scores. An immediate conclusion would be that participants simply did not generate code. However, when looking closer at the corresponding conversations that contain also prompts to generate code, this conclusion only partly holds, because participants who additionally generate code, still perform worse than those who do not use such prompts at all. We will discuss this aspect in more detail considering mixed intentions in the next section.

Result VIII: Participants who use the chatbot to generate or work with existing code achieve substantially higher correctness scores compared to requests for basic knowledge and concept comprehension, indicating a lack of knowledge that cannot be filled by the chatbot and, simultaneously, providing a potential signal to supervisors that more education is needed for the corresponding topics.

4.4.2 Task Performance by Conversational Intentions

Next to structures and intentions of conversations, Table 4 depicts participants performance in the corresponding task (column $\overline{M}_{\text{score}}$). Overall, we found that programming beginners intending to generate a solution perform best with an average score of 53.26, followed by mixed intention conversations (48.71, on average, cf. Table 4). Interestingly, programming beginners who used the chatbot solely as a tutor perform worst with an average score of 35.79, indicating that a too large gap in knowledge might not be overcome with a chatbot. We will discuss the most interesting observations per intention next.

Solution generation From all solution-generation structures, conversations can be broadly divided into two groups according to their score. In the low-score group (score below 50), participants' conversations consist only of code generation (S1) or additional refinement requests (S2a). By contrast, in the high-score group (score above 80), programming beginners not only request to generate a solution, but further request testing or debugging it (S2b, S2c, S3). This is a more nuanced perspective on the prior findings on individual prompt purposes: Although all participants' use the chatbot as a tool to generate a solution, those who additionally use the chatbot for code investigation activities (i.e., debugging or testing) perform particularly well. Notably, in the lower scoring group, there is virtually no difference between participants that simply generate code (S1) and those who iteratively refine it (S2a).

Tutoring Submissions related to tutoring as primary intention for conversations yield a wider range of scores. Three structures (S5, S8a, S8b) are associated with higher task performance (above 65, 72.50, on average), whereas five other structures (S4, S6, S7a, S7b, S9) are associated with low task performance (below 45, 28.71, on average). Especially those programming beginners whose conversations contain only code comprehension requests (S5) perform particularly well (70, on average), whereas participants whose conversations contain only requests for basic knowledge (S4) perform particularly poor (21.67, on average). While these scores related to particular structures confirm findings on the prompt purpose level Table 5, they are more pronounced: Especially, we find a stark difference between concept comprehension and code comprehension. While code comprehension seems to align with good performance, concept comprehension does the opposite. So, a chatbot might be beneficial only for certain activities.

Mixed scenario The scores of participants who have mixed intentions shed further light on the performance when prompt purposes are used that are generally associated with lower scores (basic programming knowledge, concept comprehension) and higher scores (code generation, code refinement, code comprehension, and testing). Specifically, the task performance of 9 participants who generate code after requesting basic programming knowledge (S10) is virtually the same as the performance of participants who do not generate code at all (i.e., with tutoring intentions). Again, this means, especially requests for basic programming knowledge indicate struggles of programming beginners, independent of whether they use the chatbot to generate code or not. Conversely, if participants do not request basic programming knowledge (S11, S12), scores are considerably higher and comparable to those of participants whose only intention is to use the chatbot as a tool for solution generation. In summary, programming beginners with mixed intentions can perform similarly well as those whose only intention is to use the chatbot for solution generation. Requests for basic programming knowledge is a strong indicator for struggling beginners and might serve as a warning sign to provide further guidance.

Result IX: Conversations containing validation and testing are associated with substantially better performance of programming beginners. Conversations that include basic programming knowledge not only signal missing knowledge, but the lower performance of these programming beginners might also indicate unsuccessful tutoring of the chatbot.

4.4.3 Discussion of RQ$_3$

Although our study design is mainly focused on how programming beginners interact with chatbots rather than how this interaction influences performance, we can, with over 100 submissions, draw initial insights on promising interactions and potential pitfalls, thus answering RQ$_3$.

In essence, our findings indicate that programming beginners who use the chatbot for in-depth examination of code and identification of errors show improved performance compared to those who do not. This is in line with prior work that has shown that LLM-generated summarization of code are seen as more helpful by students than those created by peers [15]. Furthermore, the better performance of programming beginners utilizing the chatbot to understand long stack traces and error messages aligns with insights from research targeting the comprehensibility of compiler error messages, which showed that enhanced error messages with explanations in natural language, help programming beginners to reduce the overall number of errors they make [2]. Similarly, the effectiveness of testing in conversations confirm prior research that identified the importance of programming beginners validating a chatbot's responses with their own knowledge [23].

Additionally, the conversations and scores suggest that either the tutoring aspect is not helpful (e.g., because understanding the answers might require further knowledge), not personalized (e.g., because the transfer of the concept explanation to the task at hand is not presented in the answer), or, generally, the chat flow is not the appropriate medium for programming beginners. Although this insight needs to be further evaluated, it nevertheless points to a limited success of tutoring capabilities for vanilla chatbots, which, however, are currently the possibly largest LLM-tool in practice. This observation is also relevant for industry, as for on-boarding activities or learning new technologies, solely relying on vanilla chatbots may be counter-productive.

While we find these clear associations between task performance and different prompt patterns, our study is not designed to examine the root cause, which means that different interpretations are valid and have to be considered and evaluated in future research. One possible reason for different performance may stem from the limited time programming beginners were given to solve the task. Clearly, 30 minutes are insufficient to compensate for a lack of understanding in basic programming concepts and data structures, even when using a chatbot.

Another explanation could target different learning styles: Some students might be oriented more toward learning through understanding concepts and focusing less on generating code, which might also lead to less code or code of lower quality within the 30 minutes sessions. While the cause of diminished performance for tutoring requests remains unclear, our results can be used for developing novel detection mechanisms to assess the purpose of prompts and signaling a lack in understanding when basic programming concepts are asked. Such an early identification of struggling programming beginners could potentially decrease drop-out rates and point companies to employees with a need for further training.

To answer RQ_3, code investigation activities are associated with higher task performance, signaling a relevance of such prompts and conversation structures for high code quality. Low performance is associated with prompts with the purpose of learning basic concepts of programming and language specifics. Although we can not clearly identify the root causes for this finding, the identified prompt purposes might serve as markers to identify struggling beginners. Therefore, building early warning systems might be a valuable avenue for education and practice. However, based on our results novel theories for conversational structures and their relationship with code quality, productivity, and early detection of a lack of programming knowledge can be developed.

5 Implications

From our results, we derive five major implications for both, practice and education, which we discuss next.

In general: *Chatbots alone are insufficient* When looking at our study, we make one interesting observation: We have rarely encountered accurate (or even close to accurate) submissions when using the chatbot. This is surprising, because the tasks are basic programming tasks that the LLM has clearly seen during training and successfully solved over and over again in the wild.

So, despite that our programming beginners have been taught the programming concepts directly before working on the tasks and are supposed to solve them on their own, they could not effectively use a chatbot that is capable of given the full, correct solution. Again, why do we not see perfect results across all the tasks? We conjecture that it takes an educated user (at least for now) to get an LLM to provide a full, correct solution for a programming task. This implies that we should not treat chatbots as stand-alone problem solvers for unskilled users, but need to educate in tandem proper CS knowledge to users to make effective use out of this new technology.

In general: *Need for prompt engineering* Crafting a prompt, such that the user receives a satisfactory answer, is not easy. It is not surprising that a number of sources teach prompt engineering[6]. Despite practical tips, such as proper formatting and chain-of-thought, we observed that the required context and capabilities of chatbots seem unclear to our participants. Thus, prior to any chatbot activity, we strongly suggest that practitioners and students alike should receive basic training of LLM mechanics and prompting.

For industry: *Need for on-boarding support* We found that using chatbots can result in better, more accurate solutions. This is interesting for onboarding inexperience developers or career changers. This way, they can become more productive while acquainting the necessary developer skills. So when entering a new programmer position, we suggest using chatbots in addition to existing measures for introduction tasks. However, this should always be accompanied with monitoring metrics to avoid negative effects chatbots might have on programming beginners.

For industry: *Need for conversation guidelines* We found evidence that particular conversation structures are related to higher code quality of submissions. Although we studied only programming beginners in education, it is likely that such a relationship exists in a practical setting and might be true for experienced developers, too. Specifically, developers should always prompt toward the generation of test functions, not just code alone. We suggest the introduction of guidelines or conventions in companies that incorporate chatbot-generated source code by adding a quality insurance step after a code generation prompt.

For education: *Monitoring of student skills* The relationship of distinct prompts and conversations with varying submission scores points to a promising new tool for monitoring student skills. We found possible indicators for when students struggle with given tasks that they are supposed to solve given the current progress of the lecture. This way, targeted, tailor-made teaching activities can be provided to these students and educators may be better informed about the current state even if students are reluctant to communicate their struggles.

6 Threats to Validity

6.1 Internal Validity

Drop-out rate and potential self-selection of participants to the experiment and control group might threaten internal validity. Both, dropout and self-selection, are due to regulatory restrictions of our educational system, such

[6] `https://www.promptingguide.ai/` or `https://www.deeplearning.ai/short-courses/chatgpt-prompt-engineering-for-developers/`

that students are free to choose which on-site tutorial (i.e., experiment vs. control group) they visit and whether they want to visit any tutorial at all. However, when students chose their tutorial group, they were not aware whether they would be in the control or the experiment group, such that they could not make an educated choice.

We observe that students usually chose one time slot at the beginning of the semester that fits their timetable and do not change afterwards. Additionally, to avoid systematic overlaps with other courses of students that could introduce bias, multiple time slots have been offered for both, the experiment and control condition, to further mitigate this threat. Also, our data shows that there was no movement of students between groups after the first session: No participant was part of the experiment group at the beginning and changed to the control group (or vice versa) while the study was running. This indicates that students did not decide intentionally to visit a different tutorial after knowing they were part of the experiment or control group, making self-selection a considerably lower threat than it appears initially. However, as with every study, dropout means that the participants who continue with the study might exhibit different properties than those who abort. In our case, motivated students or students who need assistance during the course might continue visiting the on-site tutorials. However, we have not seen any deviation from previous semesters.

Dropout furthermore threatens conclusions drawn for RQ1, especially Tasks 2 to 4 might be affected, as only four to eight students participated in the control group. However, as mentioned above, the self-selection seems to be a rather minor threat to internal validity. Putting this into perspective: Our data analysis is mainly exploratory to identify avenues for future work, these imbalances between the groups do not threaten our obtained insights. Specifically, the main insights from our study are conversational patterns and prompt purposes for which no control group is required.

Finally, to continue with the next task, participants need to select from the following options after each response from the chatbot: *The answer is helpful*, *The answer is wrong*, *I don't understand the answer*, *The answer is incomplete*, *The answer is imprecise*, *There was a technical problem.* This might introduce bias, as participants want to continue quickly. However, we used the indicators to identify participants who primarily struggle with technological problems. As this did not occur during data collection, we do not derive deeper insights from them.

6.2 External Validity

The self-selection of participants to the experimental and control condition could further threaten external validity, as we did not measure or control participants' familiarity with and interest in chatbots. Finally, the results are limited to similar settings as in our study, especially to inexperienced programmers. Nevertheless, inexperienced programmers constitute a considerable share of developers who use LLMs, and inexperienced might also count for on-boarding programmers who profit from LLMs, making the results still applicable to a sufficiently large context.

6.3 Statistical Conclusion Validity

We do not report a threat here, but explain our rational on why we do not compute statistical tests on our data. The obtained samples are too small (e.g., a control group of size four) and imbalanced to derive meaningful and reliable insights from a significant (or insignificant) result. Instead, we report the difference in percentage between the groups as an interpretable measure of effect size.

6.4 Construct validity

We measure the performance of programming beginners by taking the percentage of points participants achieved in the 30 minutes they were solving a tasks. This is a one-dimensional approach to measuring performance, but since our study is mainly exploratory to provide an initial investigation of programming beginners' performance and chatbot interactions, this is a useful compromise between giving students concrete tasks on which they can use a chatbot (allowing us to observe chatbot usage in a realistic scenario) and observing performance. As with every exploratory study, our results can guide the way to generate hypotheses for future research rather than validating them.

7 Conclusion

Code assistants have made their way into programming and are here to stay. Especially the advent of ChatGPT has boosted the application scenarios, as it is not restricted to code generation, but it can also explain code and related concepts, making it popular as tool to translate concepts into code, thereby helping programming beginners. But how do they use code assistants?

To answer this question, we conducted a study with programming beginners who used a vanilla chatbot based on GPT-3.5 to solve typical tasks in a CS2 course. By comparing the submitted solutions between programming beginners who use a chatbot and those who do not, we validate the general observation that some programming beginners, can solve tasks better when using such a chatbot. Furthermore, in 129 conversations between programming beginners and the chatbot, we observed their behavior and identified different conversation structures, some more successful than others. Especially when participants use the chatbot to refine or test generated or self-implemented code, they were more successful in solving programming tasks, compared to programming beginners who focused on generating code or requesting basic information on concepts or programming-language specific information.

Thus, we cannot just give vanilla chatbots to students as tools to learn programming, but we additionally need to give proper guidance on how to use them—otherwise, students tend to use it mainly for code generation without further reflection on or evaluation of generated code, which can lead to code of lower correctness, even for professional programmers. Thus, it is imperative that in conjunction with adopting programming assistants, also prompt engineering is adopted, so that the drawbacks of using programming assistants do not outweigh the benefits of being more productive and having support for programming whenever necessary.

Acknowledgements We thank the anonymous reviewers for their detailed and insightful comments, which have significantly improved our paper.

Alina Mailach and Norbert Siegmund's work has been supported by the Federal Ministry of Education and Research of Germany and by Sächsische Staatsministerium für Wissenschaft, Kultur und Tourismus in the programme Center of Excellence for AI-research "Center for Scalable Data Analytics and Artificial Intelligence Dresden/Leipzig", project identification number: ScaDS.AI. Norbert Siegmund's work has been funded by the German Research Foundation (SI 2171/2-2).

Data Availability

The material and data from this study are available from GitHub: `https://github.com/mailach/Ok-pal-we-have-to-code-that-now`

References

1. Barke, S., James, M.B., Polikarpova, N.: Grounded Copilot: How Programmers Interact with Code-Generating Models (2022). URL `http://arxiv.org/abs/2206.15000`. ArXiv:2206.15000 [cs]
2. Becker, B.A.: An Effective Approach to Enhancing Compiler Error Messages. In: Proceedings of the 47th ACM Technical Symposium on Computing Science Education, pp. 126–131. ACM, Memphis Tennessee USA (2016). DOI 10.1145/2839509.2844584
3. Becker, B.A., Denny, P., Finnie-Ansley, J., Luxton-Reilly, A., Prather, J., Santos, E.A.: Programming Is Hard - Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation. In: Proceedings of the 54th ACM Technical Symposium on Computer Science Education, vol. 1, pp. 500–506. ACM, Toronto ON Canada (2023). DOI 10.1145/3545945.3569759
4. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., Amodei, D.: Language Models are Few-Shot Learners. In: Advances in Neural Information Processing Systems, vol. 33, pp. 1877–1901. Curran Associates, Inc. (2020)
5. Choudhuri, R., Liu, D., Steinmacher, I., Gerosa, M., Sarma, A.: How far are we? the triumphs and trials of generative ai in learning software engineering. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pp. 1–13 (2024)
6. Denny, P., Becker, B.A., Leinonen, J., Prather, J.: Chat Overflow: Artificially Intelligent Models for Computing Education - renAIssance or apocAIypse? In: Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education, *ITiCSE 2023*, vol. 1, pp. 3–4. ACM, New York, NY, USA (2023). DOI 10.1145/3587102.3588773

7. Denny, P., Kumar, V., Giacaman, N.: Conversing with Copilot: Exploring Prompt Engineering for Solving CS1 Problems using Natural Language. In: Proceedings of the 54th ACM Technical Symposium on Computer Science Education, *SIGCSE 2023*, vol. 1, p. 1136–1142. ACM (2023)
8. Denny, P., Prather, J., Becker, B.A., Finnie-Ansley, J., Hellas, A., Leinonen, J., Luxton-Reilly, A., Reeves, B.N., Santos, E.A., Sarsa, S.: Computing Education in the Era of Generative AI (2023). URL `http://arxiv.org/abs/2306.02608`
9. Finnie-Ansley, J., Denny, P., Luxton-Reilly, A., Santos, E.A., Prather, J., Becker, B.A.: My AI Wants to Know If This Will Be on the Exam: Testing OpenAI's Codex on CS2 Programming Exercises. In: Proceedings of the 25th Australasian Computing Education Conference, ACE '23, p. 97–104. ACM, New York, NY, USA (2023). DOI 10.1145/3576123.3576134
10. Frankford, E., Sauerwein, C., Bassner, P., Krusche, S., Breu, R.: Ai-tutoring in software engineering education. In: Proceedings of the 45th International Conference on Software Engineering: Software Engineering Education and Training, ICSE-SEET '23. ACM (2024)
11. Kazemitabaar, M., Chow, J., Ma, C.K.T., Ericson, B.J., Weintrop, D., Grossman, T.: Studying the Effect of AI Code Generators on Supporting Novice Learners in Introductory Programming. In: Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems, pp. 1–23. ACM, Hamburg Germany (2023). DOI 10.1145/3544548.3580919
12. Kazemitabaar, M., Hou, X., Henley, A., Ericson, B.J., Weintrop, D., Grossman, T.: How novices use llm-based code generators to solve cs1 coding tasks in a self-paced learning environment. In: Proceedings of the 23rd Koli Calling International Conference on Computing Education Research, pp. 1–12 (2023)
13. Kuhail, M.A., Alturki, N., Alramlawi, S., Alhejori, K.: Interacting with Educational Chatbots: A Systematic Review. Education and Information Technologies **28**(1), 973–1018 (2023)
14. Leinonen, J., Denny, P., MacNeil, S., Sarsa, S., Bernstein, S., Kim, J., Tran, A., Hellas, A.: Comparing Code Explanations Created by Students and Large Language Models. In: Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education, *ITiCSE 2023*, vol. 1, p. 124–130. Association for Computing Machinery, New York, NY, USA (2023). DOI 10.1145/3587102.3588785
15. Leinonen, J., Hellas, A., Sarsa, S., Reeves, B., Denny, P., Prather, J., Becker, B.A.: Using Large Language Models to Enhance Programming Error Messages. In: Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1, pp. 563–569. ACM, Toronto ON Canada (2023). DOI 10.1145/3545945.3569770
16. Liang, J.T., Yang, C., Myers, B.A.: A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges. In: 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE), pp. 605–617. IEEE Computer Society, Los Alamitos, CA, USA (2024)
17. Liu, R., Zenke, C., Liu, C., Holmes, A., Thornton, P., Malan, D.J.: 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, SIGCSE 2024, p. 7. ACM, New York, NY, USA (2024). DOI 10.1145/3626252.3630938
18. MacNeil, S., Kim, J., Leinonen, J., Denny, P., Bernstein, S., Becker, B.A., Wermelinger, M., Hellas, A., Tran, A., Sarsa, S., Prather, J., Kumar, V.: The Implications of Large Language Models for CS Teachers and Students. In: Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 2, pp. 1255–1255. ACM, Toronto ON Canada (2022). DOI 10.1145/3545947.3573358
19. MacNeil, S., Tran, A., Hellas, A., Kim, J., Sarsa, S., Denny, P., Bernstein, S., Leinonen, J.: Experiences from Using Code Explanations Generated by Large Language Models in a Web Software Development E-Book. In: Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1, SIGCSE 2023, p. 931–937. ACM (2023). DOI 10.1145/3545945.3569785
20. Meyer, B.: What Do ChatGPT and AI-based Automatic Program Generation Mean for the Future of Software (2022). URL `https://cacm.acm.org/blogs/blog-cacm/268103-what-do-chatgpt-and-ai-based-automatic-program-generation-mean-for_the-future-of-software/fulltext`

21. Mozannar, H., Bansal, G., Fourney, A., Horvitz, E.: Reading Between the Lines: Modeling User Behavior and Costs in AI-assisted Programming (2023)
22. Nam, D., Macvean, A., Hellendoorn, V., Vasilescu, B., Myers, B.: Using an llm to help with code understanding. In: 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE), pp. 881–881. IEEE Computer Society, Los Alamitos, CA, USA (2024)
23. Ouh, E.L., Gan, B.K.S., Jin Shim, K., Wlodkowski, S.: ChatGPT, Can You Generate Solutions for my Coding Exercises? An Evaluation on its Effectiveness in an undergraduate Java Programming Course. In: Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1, ITiCSE 2023, p. 54–60. Association for Computing Machinery, New York, NY, USA (2023). DOI 10.1145/3587102.3588794
24. Perry, N., Srivastava, M., Kumar, D., Boneh, D.: Do Users Write More Insecure Code with AI Assistants? In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, pp. 2785–2799 (2023)
25. Prather, J., Denny, P., Leinonen, J., Becker, B.A., Albluwi, I., Caspersen, M.E., Craig, M., Keuning, H., Kiesler, N., Kohn, T., Luxton-Reilly, A., MacNeil, S., Petersen, A., Pettit, R., Reeves, B.N., Savelka, J.: Transformed by Transformers: Navigating the AI Coding Revolution for Computing Education: An ITiCSE Working Group Conducted by Humans. In: Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 2, pp. 561–562. ACM, Turku Finland (2023). DOI 10.1145/3587103.3594206
26. Prather, J., Reeves, B.N., Denny, P., Becker, B.A., Leinonen, J., Luxton-Reilly, A., Powell, G., Finnie-Ansley, J., Santos, E.A.: "It's Weird That It Knows What I Want": Usability and Interactions with Copilot for Novice Programmers. ACM Transactions on Computer-Human Interaction **31**(1) (2023). DOI 10.1145/3617367
27. Robins, A.V.: 12 novice programmers and introductory programming. The Cambridge handbook of computing education research pp. 327–376 (2019)
28. Ross, S.I., Martinez, F., Houde, S., Muller, M., Weisz, J.D.: The Programmer's Assistant: Conversational Interaction with a Large Language Model for Software Development. In: Proceedings of the 28th International Conference on Intelligent User Interfaces, IUI '23, p. 491–514. ACM (2023). DOI 10.1145/3581641.3584037
29. Sandoval, G., Pearce, H., Nys, T., Karri, R., Garg, S., Dolan-Gavitt, B.: Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants. In: 32nd USENIX Security Symposium (USENIX Security 23), pp. 2205–2222. USENIX Association, Anaheim, CA (2023)
30. Shoufan, A.: Can students without prior knowledge use chatgpt to answer test questions? an empirical study. ACM Transactions on Computing Education **23**(4), 1–29 (2023)
31. Siegmund, J., Kästner, C., Liebig, J., Apel, S., Hanenberg, S.: Measuring and Modeling Programming Experience. Empirical Software Engineering **19**, 1299–1334 (2014)
32. Vaithilingam, P., Zhang, T., Glassman, E.L.: Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In: Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems, CHI EA '22. ACM, New York, NY, USA (2022). DOI 10.1145/3491101.3519665
33. Verleger, M., Pembridge, J.: A Pilot Study Integrating an AI-driven Chatbot in an Introductory Programming Course. In: 2018 IEEE Frontiers in Education Conference (FIE), pp. 1–4 (2018). DOI 10.1109/FIE.2018.8659282
34. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q.V., Zhou, D.: Chain of Thought Prompting Elicits Reasoning in Large Language Models. Advances in Neural Information Processing Systems **35**, 24824–24837 (2022)
35. Weisz, J.D., Muller, M., Ross, S.I., Martinez, F., Houde, S., Agarwal, M., Talamadupula, K., Richards, J.T.: Better Together? An Evaluation of AI-Supported Code Translation. In: 27th International Conference on Intelligent User Interfaces, IUI '22, p. 369–391. ACM, New York, NY, USA (2022). DOI 10.1145/3490099.3511157
36. Wermelinger, M.: Using GitHub Copilot to Solve Simple Programming Problems. In: Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1, pp. 172–178. ACM, Toronto ON Canada (2023). DOI 10.1145/3545945.3569830