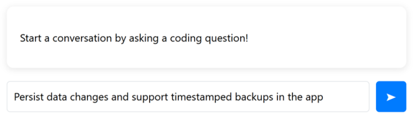
# OptionPilot: Investigating an Interactive, Path-Oriented Approach to LLM-Driven Code Creation

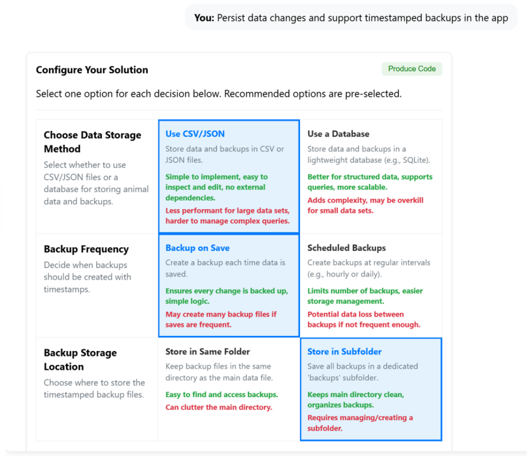
Arne Riethmüller
arne.riethmueller@gmail.com
Technische Universität Berlin
Berlin, Germany



Start a conversation by asking a coding question!

Persist data changes and support timestamped backups in the app

1. User submits a request, as with other LLM assistants.



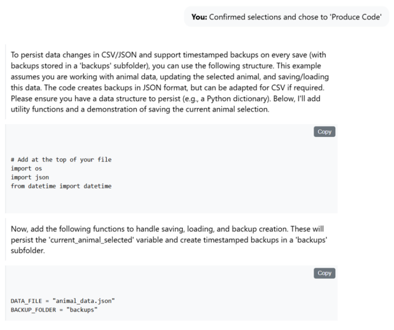
**You:** Persist data changes and support timestamped backups in the app

**Configure Your Solution**

Select one option for each decision below. Recommended options are pre-selected.

| | | |
|---|---|---|
| **Choose Data Storage Method** Select whether to use CSV/JSON files or a database for storing animal data and backups. | [✓ **Use CSV/JSON** Store data and backups in CSV or JSON files. Simple to implement, easy to inspect and edit, no external dependencies. Less performant for large data sets, harder to manage complex queries.] | **Use a Database** Store data and backups in a lightweight database (e.g., SQLite). Better for structured data, supports queries, more scalable. Adds complexity, may be overkill for small data sets. |
| **Backup Frequency** Decide when backups should be created with timestamps. | [✓ **Backup on Save** Create a backup each time data is saved. Ensures every change is backed up, simple logic. May create many backup files if saves are frequent.] | **Scheduled Backups** Create backups at regular intervals (e.g., hourly or daily). Limits number of backups, easier storage management. Potential data loss between backups if not frequent enough. |
| **Backup Storage Location** Choose where to store the timestamped backup files. | **Store in Same Folder** Keep backup files in the same directory as the main data file. Easy to find and access backups. Can clutter the main directory. | [✓ **Store in Subfolder** Save all backups in a dedicated 'backups' subfolder. Keeps main directory clean, organizes backups. Requires managing/creating a subfolder.] |

2. OptionPilot displays an interactive UI that allow the user to steer the output. User can configure a solution by clicking on option for each decision and confirm their choices by clicking `Produce Code`.



**You:** Confirmed selections and chose to 'Produce Code'

To persist data changes in CSV/JSON and support timestamped backups on every save (with backups stored in a 'backups' subfolder), you can use the following structure. This example assumes you are working with animal data, updating the selected animal, and saving/loading this data. The code creates backups in JSON format, but can be adapted for CSV if required. Please ensure you have a data structure to persist (e.g., a Python dictionary). Below, I'll add utility functions and a demonstration of saving the current animal selection.

```
# Add at the top of your file
import os
import json
from datetime import datetime
```

Now, add the following functions to handle saving, loading, and backup creation. These will persist the 'current_animal_selected' variable and create timestamped backups in a 'backups' subfolder.

```
DATA_FILE = "animal_data.json"
BACKUP_FOLDER = "backups"
```

3. The LLM implements the solution according to the options selected by the user.

**Figure 1: The core functionality of the prototype.**

## Abstract

Large Language Models (LLMs) are increasingly integrated into software development workflows, yet their use for code generation continues to face recurring challenges such as premature outputs and obscured implementation paths. This work introduces the concept of *decision points*, a mechanism that inserts a pause between a user's request and code generation in LLM coding assistants in case ambiguities remain or multiple potential implementation paths exist. Here, instead of producing code immediately, key implementation choices are surfaced to the user through interactive UI elements that highlight decisions along with applicable options for the user to select, allowing users to clarify their intent and steer outputs. We implemented *OptionPilot* as a proof-of-concept prototype to evaluate the proposed concept in a mixed-methods, qualitative first lab study with 14 participants. Findings indicate that decision points can enhance users' sense of control and reduce frustration from premature outputs or misaligned code generation. Participants appreciated decision points for making key decisions and trade-offs explicit, though their effectiveness depended on the perceived contextual relevance of presented decisions. We argue that well-designed interactive mechanisms, as illustrated by decision points, have the potential to enhance the experience of using LLM coding assistants and to facilitate more effective forms of human-AI collaboration in other fields.

## 1 Introduction

Large Language Models (LLMs) have demonstrated significant potential in supporting software engineering tasks, beginning with the ability to generate code from natural language instructions [28, 16, 25]. Additionally, LLMs are proving valuable for a range of adjacent tasks such as generating documentation, and their capabilities are increasingly being integrated directly into integrated development environments (IDEs), thereby unlocking new workflows and opportunities for developer assistance [11, 24, 13]. Ongoing research is investigating these developments from various perspectives, examining not only the code generation capabilities of LLMs but also how their integration into development environments could reshape existing workflows [1, 28, 35]. Research finds, for example, that efficient interactions and clear presentation of outputs are key factors shaping how valuable developers perceive LLM assistance to be [27].

Although LLM-assisted programming is widely regarded as holding significant potential, research consistently highlights recurring shortcomings that limit its effectiveness, stemming both from the capabilities of the underlying models and their integration into LLM programming assistants commonly used in IDEs.

For example, LLMs in general frequently respond to underspecified requests with fully formed solutions. They do this by making assumptions about the user's intent rather than prompting for clarification, and generally show shortcomings in capturing user intent [14, 6]. This tendency can lead to premature outputs, primarily code that might be syntactically correct yet misaligned with the developers' actual intentions, potentially causing unnecessary rework and frustration [15, 31]. This behavior, in part, forces developers to carefully scrutinize long blocks of generated code to determine whether incorrect assumptions may have been introduced, thereby contributing to cognitive overload [3, 34, 1]. Long, insufficiently structured outputs make it difficult for developers to identify critical information or verify whether the implementation aligns with the original intent. Similarly, important implementation choices are often not highlighted, leaving alternative implementation paths obscured [15, 34]. This lack of transparency both increases the cognitive effort required to review outputs and leads developers to experiencing a loss of control over the implementation process [3].

We introduce the concept of *decision points* as a mechanism to address these limitations. Decision points follow the notion of augmentation instead of automation to address the shortcomings of premature outputs, obscured implementation paths, and high cognitive load.

Differing from prevailing approaches, the concept intends to insert a pause between request and an LLM's output whenever ambiguities remain, instead surfacing these ambiguities or potential implementation paths to the user in an interactive UI displaying multiple options, thereby allowing the user to clarify intent or steer the outcome. Instead of producing a solution based on low-confidence assumptions, for example, when multiple implementation paths are valid, the system prompt is designed to identify areas where information is missing and prompt the user for input.

To examine the potential for addressing the identified shortcomings of LLM-based programming assistants through Decision Points, we developed a prototype called *OptionPilot*. Figure 1 illustrates an example interaction in which (1.) a user enters a prompt containing ambiguities, and (2.) the underlying LLM generates a decision point that allows the user to choose between different options, each accompanied by explanations of potential trade-offs. After making their decisions, the user clicks `Produce Code`, which triggers the LLM to implement the solution according to the user-specified choices (3.).

The concept aims to demonstrate a subtle design shift from immediate generation to augmented interaction, thereby enhancing the developer experience by reducing the need to correct misaligned code or review lengthy outputs for incorrect assumptions.

Building on the identified shortcomings of current LLM-based coding assistants, namely premature code outputs, obscured implementation paths, and high cognitive load, this work investigates whether the concept termed decision points has potential to address these shortcomings through a collaborative approach. The core assumption is that by surfacing key decisions before code is generated, users can consciously steer the outcome, thereby reducing the mental effort required to interpret or correct misaligned outputs afterward. To investigate this, we formulated the following research questions:

**RQ1** To what extent do decision points or specific aspects of them, demonstrate potential to mitigate premature outputs in LLM-assisted programming?

This question addresses the tendency of current tools to generate code prematurely, for instance, as a result of incomplete or ambiguous information.

**RQ2** To what extent do decision points or specific aspects of them, demonstrate potential to address obscured implementation paths in LLM-assisted programming?

This question examines whether and how decision points can be effective at making different implementation paths, as well as their implications and trade-offs explicit to users.

**RQ3** By addressing these aspects, to what extent do decision points demonstrate potential to reduce developers' cognitive load during LLM-assisted programming?

This question connects the two preceding aspects, evaluating whether, by addressing these issues or through additional mechanisms, decision points hold potential to reduce developers' cognitive load during LLM-assisted programming.

To investigate the potential of the proposed interaction concept, we first implemented a proof-of-concept prototype. After initial refinement, we conducted a mixed-methods, qualitative first user study to collect initial insights from user experiences. In moderated lab sessions, we asked professional developers and adjacent job roles of varying experience levels to work on two sets of programming tasks, with decision points enabled for one of the two sets. Data was collected through think-aloud protocols, screen recordings, questionnaires, and semi-structured exit interviews. This approach enabled us to capture data from multiple perspectives and obtain initial insights into the potential of decision points.

Our findings indicate that the concept of decision points has the potential to improve the experience of using LLM-based coding assistants. Participants in the user study appreciated being able to steer outputs proactively and reported that the approach reduced frustration with premature or misaligned code generation. At the same time, the results highlight that the effectiveness of decision points strongly depends on their contextual relevance and timing within the interaction. Decision points perceived as irrelevant increase mental load rather than reducing it. The clear visual highlighting of decisions and their implications, as well as the interactivity of decision points, were seen as beneficial and could be effective not only for enhancing LLM-based coding assistants but also for improving the experience of using LLM chats in general. Based on these insights, we propose several recommendations for future adaptations, including default decision options and collapsible sections with additional information, such as example code for different options.

This work makes three contributions. We introduce decision points as a concept for LLM-based coding assistants to address the shortcomings as discussed above. We present OptionPilot as a proof-of-concept prototype that implements decision points through

JSON-structured LLM outputs and an interactive UI layer. Finally, we report findings from a mixed-methods, qualitative first user study comparing developers' usage of OptionPilot to a baseline condition. The study yielded insightful observations about the concept and directions for future adaptations. Through these findings, we contribute to the developing notion of augmentation rather than automation in human-LLM interaction, introducing concepts that may extend beyond programming assistants.

In this work, We first situate this study within the broader context of research on LLM-based developer assistance and outline the identified shortcomings motivating this study. In Section 3, we detail the implementation of the proof-of-concept prototype, while Section 4 describes the methodological design of the user study. In Section 5, we present the user study findings with their implications and limitations discussed in Section 6. Finally, in Section 7 we conclude by summarizing the main outcomes and reflecting on potential future research directions.

## 2 Background and Related Work

Applying LLMs to code generation and software engineering, in general, is a rapidly evolving and highly promising area of research. The capabilities of LLMs are advancing continuously and are accompanied by new tools and methods designed to leverage these emerging opportunities [7].

In the following, we provide a brief overview of different research directions in the field. We first describe how LLMs are used as developer tools, ranging from code completion to broader forms of assistance within IDEs. We then highlight an ongoing shift in focus from full automation to human-centered augmentation, illustrated with examples from prior research.

Finally, we summarize recurring shortcomings reported across studies and approaches, namely premature outputs, obscured implementation paths, and high cognitive load. This review is intentionally selective rather than exhaustive, with the goal of presenting an overview of the research landscape and deriving open potentials to motivate this work.

### 2.1 LLMs as Developer Tools

LLMs have proven to be valuable tools across a broad range of domains involving natural language, and they continue to grow in capability through advancements such as larger parameter counts and techniques like chain-of-thought reasoning [30, 7, 33]. Software engineering has emerged as a promising domain for the application of LLMs, as they demonstrate potential in a variety of related tasks, beginning with code generation [28].

Initially, research in this area focused on automatic code generation and program synthesis, ranging from generating code from formal specifications to producing code from natural language instructions [16, 25]. This capability can be applied in practice to support developers, for example, by providing code completion features in IDEs, where the assistant continuously makes suggestions to complete lines of code while the developer is typing [1].

Increasingly, LLMs are being applied more broadly across software engineering activities, moving beyond code generation and related tasks, such as translating code between languages, to also assisting in activities like code explanation and documentation generation [11, 22, 23, 28].

Although LLMs are being used for many of these tasks through general-purpose chat interfaces such as *ChatGPT*[1], there are increasing efforts to integrate these capabilities more closely into software engineering workflows. Popular IDEs such as Visual Studio Code, as well as newer tools like *Cursor*[2], now increasingly embed LLM-based chat interfaces and other LLM-based features directly into development environments. These integrations assist developers by embedding LLMs more deeply into their workflows, allowing them, for instance, to continuously access contextual information from their active projects [13, 11, 24].

These multi-purpose LLM coding assistants augment developers' work in a variety of novel ways and are increasingly being adopted by developers [24, 15]. In a typical workflow with an IDE-integrated chat interface, the starting point is comparable to working with a standard LLM chat interface such as ChatGPT. For this, the appropriate context must be established, developers must write prompts and review generated outputs, often iteratively, continuously assessing which parts of the outputs to integrate into their work [3].

### 2.2 Shifting Perspectives on LLMs for Developer Assistance

As LLM-based tools become increasingly capable and widely used, research has begun looking beyond their performance in code generation alone towards an understanding of how these tools fit into developer workflows. Newer studies explore how LLMs can be deployed to assist in, or even redefine, developer work, opening up new possibilities and challenges.

The predominant vision of the future of LLM-assisted programming assumes humans taking the role of stating and potentially refining an end goal, after which an LLM carries out the implementation, producing results that are then assessed by a human [31]. Research in this direction focuses on assessing and improving code generation capabilities directly or through the development of new techniques, such as integrating automated tests for generated code [6, 32].

Other studies approach the assessment of the usefulness of coding assistants in a more holistic way, for example by investigating which specific tasks make up the activity of software engineering and how tools could assist in these areas. One study investigates the usefulness of LLM programming assistants for tasks in four groups of activity, namely: (1.) implementing new features, (2.) writing tests, (3.) bug triaging and (4.) refactoring and writing natural language artifacts [24]. Similarly, a related study examines the roles LLMs can take in assisting in seven categories of activities in software engineering such as code generation, code translation, vulnerability detection, or question and answer interactions [33].

Studies falling into the mentioned categories tend to assess the performance of assistants by benchmarking against existing datasets for example sets of coding problems or, by investigating specific capabilities for classes of problems by combining evaluation metrics [11].

[1]https://chatgpt.com/
[2]https://cursor.com/

Following the early focus on benchmarking and technical performance, recent research has increasingly turned toward examining how LLM programming assistants are used in real-world practice. An example of this investigates how programmers interact with the code generation feature of GitHub Copilot and finds that tasks for which developers seek LLM assistance there can be divided into two categories: exploration mode as in exploring potential solutions for a problem, and acceleration mode as in implementing an envisioned solution [1]. In another example, researchers examined the usage of *watsonx*, the in-house coding assistant from IBM. Among other results, findings suggest that code understanding in general was a more common use case than actual code generation [28].

Investigating what tasks these tools are utilized for is an example of how research is increasingly exploring the field of LLM assistance for software development from a variety of angles, attempting to understand how developers will leverage and adapt to these novel tools and possibilities in the future. Findings that offer valuable insights in this regard are, for example, the following. Within the output of LLM assistants, accompanying information such as explanations can be equally relevant to the experience as the code output itself [7]. Similarly, findings suggest that an efficient interaction design and clear presentation of outputs are key factors for the perceived quality of a developers' experience with LLM tools, while a narrow focus on technical performance can obscure their broader potential [27].

Insights like these are prompting broader questions that go beyond task-specific performance toward understanding the evolving role of LLM tools within developer workflows. Together with the growing adoption of LLM programming assistants, this raises questions about how programming workflows will evolve and how new capabilities may redefine established assumptions, roles, and processes [1, 23, 27].

An example of this shift in perspective, from evaluating the correctness of generated code to envisioning new forms of collaboration, is the growing notion of augmentation rather than automation. For instance, data from the *Anthropic Economic Index* published by *Anthropic* show that 57% of prompts on *Claude.ai*[3], the company's general-purpose LLM chat interface, lean toward collaboration instead of automation (43%) [10]. This is reflected in other findings that show users of artificial intelligence (AI) systems favor retaining the final decision over an interaction's output, thereby maintaining a sense of control [9]. This marks a shift away from the prior emphasis on increasing automation, which has long been the focus of many AI systems [20].

These insights relate closely to a recurring limitation observed in LLM programming assistants and LLMs more broadly, the challenge of accurately capturing user intent. Current approaches regularly fall short in this regard, for instance, by failing to guide users to clarify their intent when a request is ambiguous [6, 11]. LLMs generally show a tendency to respond to requests with fully formed solutions, even when crucial information is missing, and do so by making potentially incorrect assumptions [14]. When presented with a coding task, LLMs show a tendency to immediately output code upon the first user request. This behavior can lead to code outputs that are, for example syntactically correct but do not support the intended functionality [11]. This tendency to generate code immediately can have a range of negative consequences, for instance, forcing users to skim lengthy code suggestions to determine their usefulness, which can lead to cognitive overload. One way to mitigate this issue is to design tools that refrain from low-confidence assumptions, meaning the tool should first determine whether sufficient context is available before making a suggestion [1].

This can be seen as an example of how programming might evolve, with human-AI interaction becoming an important part. By recognizing both the capabilities of LLMs and their limitations in capturing user intent, the focus of code generation can shift from full automation toward more augmented approaches. This shift illustrates how programming is likely to evolve, with human-AI interaction becoming an increasingly central element of the development process.

Rather than focusing on how current tools are used, an alternative perspective is to explore potential future possibilities while drawing key lessons from prior approaches. Research in this direction investigates new forms of collaboration between humans and LLMs, moving beyond one-shot code generation. One study, for example, starts from the idea that programming can be understood as a design activity, as in the process of creating software not just being a matter of translating a set of fixed requirements into code, but as a process of exploring both the given problem and possible solutions iteratively in order to find a suitable solution. The researchers implemented an IDE prototype designed to support iterative exploration of multiple implementation options and their associated trade-offs, while tracking decisions and rationales outside the chat history. Although their approach, which featured three separate, coordinated agents, enabled broader exploration of alternatives, it also introduced additional cognitive load, as users felt overwhelmed by managing multiple panels and functionalities within the IDE [31].

A different study examines an approach where, in a specific use case, a UI consisting of sliders, drop-downs, or other elements is placed between the prompt and the code output, allowing users to make certain adjustments to the output. This scaffolding is designed to improve code understandability and reduce prompting effort. They report that their approach reduced prompting effort and encouraged exploration, but insufficient explanations and unwanted suggestions slowed users down [4].

Another illustrative example investigates a simpler idea which simultaneously follows the theme of reassessing the use and role of LLMs in programming based on what is possible rather than what is currently done. They implemented an IDE plugin that enables users to generate explanations for a specific code snippet, with a single mouse click after highlighting the selected code snippet, while taking into account the context in the given project. They find that users appreciated having the plugin, viewing it as an easy-to-use add-on to Copilot [17]. This study exemplifies how current systems can be improved with relatively simple solutions that utilize LLMs in imaginative ways.

These studies highlight the potential of novel solutions while surfacing potential pitfalls.

[3]https://claude.ai/

## 2.3 Identified Shortcomings in Previous Approaches

To provide context for the motivation of this work, we highlight three related issues with current approaches in LLM-assisted programming, as identified by research, beginning with an expansion of the previously outlined issue of premature outputs. Premature outputs are in part causing obscured implementation paths and both issues contribute to cognitive overload.

*Premature Outputs.* While premature outputs are a documented challenge for LLMs in general, their tendency to produce final artifacts upon a user request can be a particular issue for LLM programming assistants.

When conditions are not fully specified, an LLM may generate code that fulfills the explicitly stated requirements but fails to capture the underlying intent of the developer [15, 14]. This limitation is amplified by the fact that requirements are frequently underspecified in early prompts and tend to in part only evolve through subsequent design decisions or trade-offs made during the implementation process [31]. Hallucinations, a well-documented issue for LLMs in general, can additionally contribute to this issue. Instead of surfacing a lack of information to the user, LLMs occasionally fabricate pieces of missing information, which is at times hard to detect for users [7]. These issues are exacerbated by LLMs' difficulty in revising prior misconceptions and struggle to do so even when incorrect assumptions have been corrected further along in an interaction [14].

Premature outputs contribute to developer frustration as developers must invest effort in correcting invalid results [15, 14]. Given the persistence of premature outputs, users must routinely review generated code to identify potential inaccuracies such as misaligned assumptions, frequently working under the assumption that unintended and potentially unwanted elements are being implemented [34, 1]. As an alternative to fully formed but premature outputs, research shows that users often prefer receiving an initial starting point that scaffolds a solution, enabling them to refine and extend the implementation according to their own intentions [31, 28, 35].

*Obscured Implementation Paths.* Obscured implementation paths occur when an LLM proceeds with a specific solution without making users aware of alternative approaches, which may carry entirely different, and potentially more suitable, implications for the future of the codebase. The code generated might be fully functional, nevertheless critical choices are regularly not highlighted in LLM outputs [25]. Obscured implementation paths are a related issue to premature outputs and arise at times as a consequence of them. With obscured implementation paths, the issue primarily lies in a limited ability of the user to steer the output and influence the broader project trajectory, which leads to users experiencing a loss of control [15, 34]. Obscured implementation paths can require users to engage in additional prompting to uncover potentially more suitable or effective approaches [3].

A number of studies emphasize the need for LLM programming assistants to incorporate intuitive mechanisms that enable users to steer the output and explore multiple implementation alternatives [34, 15]. Current systems in part experiment with offering users more than one suggestion at a time, for instance, by generating multiple code snippets and allowing users to choose among them [1]. This remains the exception rather than the norm, even when such features are supported, factors that might influence the code in future developments are not explicitly highlighted and assumptions are hidden in longer outputs, requiring effort to detect them [31].

In addition to a loss of control and potential long term negative effects on the codebase, not involving users in decision making can introduce another drawback. The paper *Programming as Theory Building* (1985) argues that the output of the activity of programming is not only the resulting code but also mental constructs developed by the developer during the work. When developers are not aware of decisions being made in implementing requirements into code, for example by making decisions, they do not form a theory or understanding of the code [18, 21].

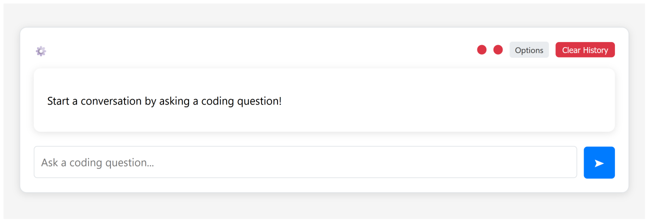
*Cognitive Overload.* Cognitive load can be defined as "The load imposed on working memory by information being presented." [19]. Cognitive load can be divided into intrinsic cognitive load, meaning load stemming from the task itself, extraneous load, meaning load stemming from how the task is presented, unrelated to the task itself, and germane cognitive load, as in productive effort used to process information and understand the task [26] [19].

Both issues outlined previously contribute to cognitive overload reported by users of LLM programming assistants. Premature outputs, as well as the persistent potential for their occurrence, require users to continually account for these risks. Obscured implementation paths similarly increase extraneous cognitive load by necessitating users to continuously question the LLM's outputs. When outputs fail to meet the user's intentions or contain bugs, developers may spend excessive time modifying generated code [15]. At the same time, partly in anticipation of such issues, users spend cognitive effort upfront by writing long prompts and carefully setting context to minimize the likelihood of unwanted results, leading to frustration when outputs do not meet expectations [13].

An additional factor identified in the literature as contributing to cognitive overload is excessive amounts of information being presented by LLM outputs at once [3, 34, 1, 31]. Assistants frequently produce extended code segments that obscure essential details, making it difficult for users to identify and comprehend key information. As a result, developers may disregard generated outputs entirely when they cannot easily understand them [15, 34]. This issue becomes particularly apparent when the generated code lacks sufficient accompanying explanations [7]. Approaches that allow users to scale the amount of information displayed according to their preferences and needs for explanation may help address this [3].

## 3 Proof-of-Concept Implementation

The prototype was developed from an idea to address the shortcomings outlined in Section 2.3. The concept is based on novel UI elements hereafter referred to as decision points which would ultimately be integrated into LLM coding assistants. Decision points in essence provide users with clickable concise options, instead of text-only output when conditions are met. Decision points are intended to augment the interaction with assistants through surfacing decisions or missing information to the user and allowing them



Start a conversation by asking a coding question!

**Figure 2: Empty Chat Interface.**

to conveniently make decisions while displaying an appropriate amount of accompanying information.

The following describes how the prototype for evaluating decision points was developed and what enables its functionality. First, to illustrate a potential interaction with decision points, we outline an example of a user experience, followed by an explanation of how the prototype was implemented. We provide a description of the most relevant aspects that enable decision points followed by some theory on how decision points are intended to address the identified shortcomings.

## 3.1 Prototype User Workflow Example

The prototype resulting from the design process named OptionPilot is intended to support the evaluation of the idea in a user study, and is therefore reduced to the essential functionality. Nevertheless, the prototype successfully provides an initial understanding of how these elements could ultimately work, as envisioned, when integrated into programming assistants.

The general interface is a typical LLM chat, see Figure 2. Users can interact with the underlying LLM as with any other.

Unlike other tools, OptionPilot supports the following additional functionality. Instead of plain text answers, OptionPilot will regularly, upon a user's request, display one of two types of decision points. A *Single Decision Point*, see Figure 3 and a *Multi Decision Point*, see Figure 1. As interactive UI elements, decision points allow users to make decisions highlighted for them by clicking on these elements. In the given example Figure 3, a user requests the implementation of a start menu for a CLI game. Through the displayed decision point, the user can choose between two ways to implement this.

In the background, this is enabled through a system prompt that instructs an underlying LLM to output JSON objects that are then rendered in the displayed UI. As part of the system prompt, the LLM is instructed to assess user request for a number of factors, such as ambiguous requests or path decisions.

Decision points are generally interactive, in single decision points, options can be expanded and collapsed to reveal additional information and users can select an option they want to proceed with by clicking a button displayed for each option. Being able to respond to questions asked by assistants by clicking on them is a simple feature, but curiously absent in existing assistants. For single decision points, each option has a `Generate Code` button displayed next to it, which will trigger a prompt that instructs the LLM to implement the chosen option into code. Additionally, a `Proceed` button will trigger instructions to continue the conversation. This option leaves the decision about the next suitable step in the conversation, which could be another decision point, to the LLM.

In multi decision points, instead of one single decision with two options being displayed, two to four decisions with two options each are being displayed simultaneously, see Figure 4. The user may choose an option for each decision, with chosen options being highlighted by the blue border for each decision. Recommended options selected by the LLM are preselected when the decision point is first displayed. Similar to single decision points, users can click `Produce Code` which triggers instructions to implement code according to selected choices. Users can generally ignore decision points and choose to continue the interaction with other instructions. For both types of decision points, users have the option to include additional instructions for proceeding with decisions in the text input field.

## 3.2 OptionPilot Implementation Overview

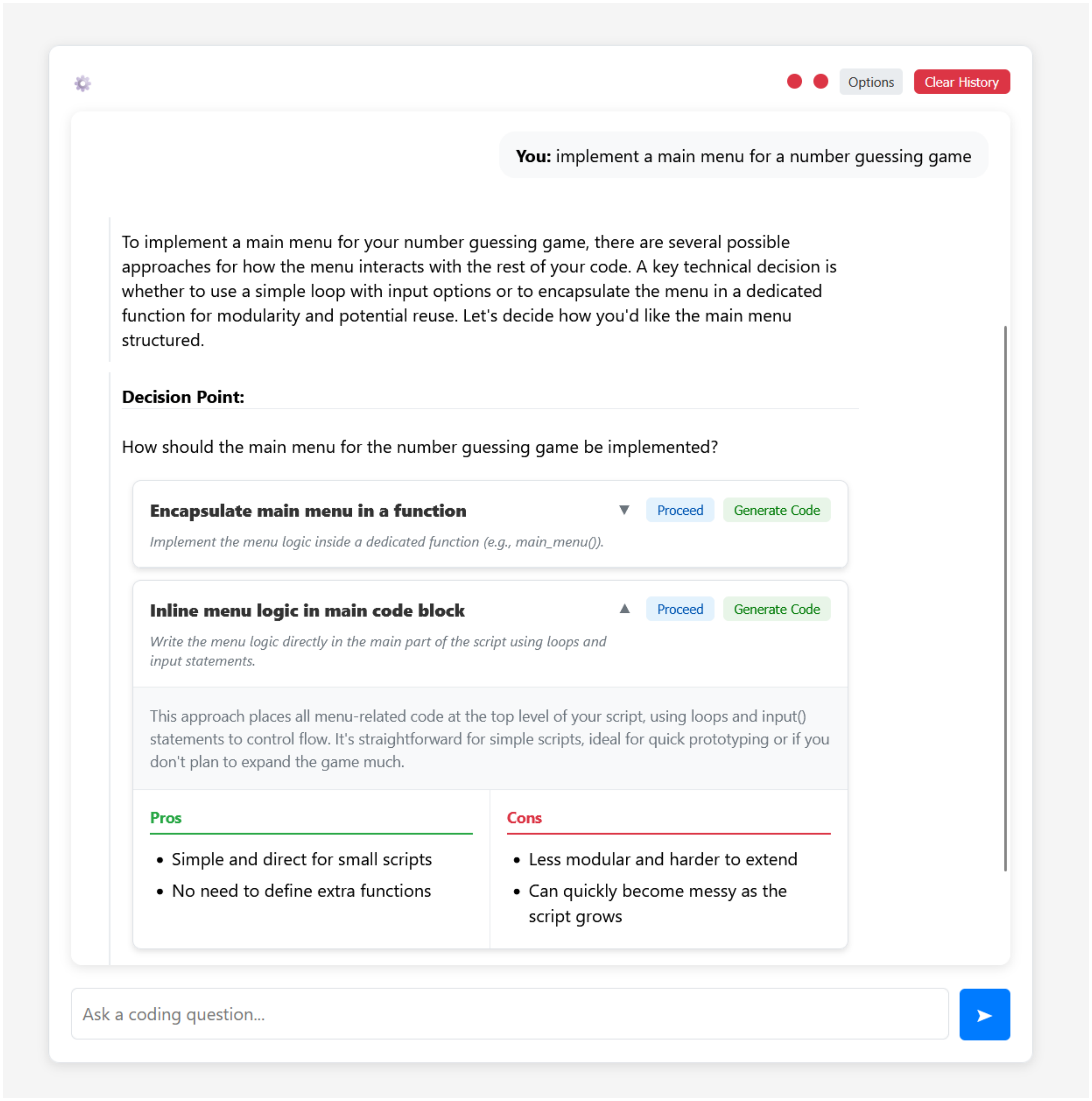
A lightweight Flask[4] webapp was implemented to ensure basic prerequisites like session management. HTMX[5] was used to enable some of the frontend interactivity. OptionPilot uses the OpenAI GPT-4.1 API which proved capable enough to handle the complexities of the system prompt described below, and was effective at code generation. Experiments with less performant models led to a considerable decrease in the user experience through less relevant decisions and increased occurrences of answers with invalid JSON.

*System Prompt.* The foundation of OptionPilot's functionality is an iteratively designed system prompt that details the necessary instructions to achieve the intended behavior of the LLM. See Appendix A for the full prompt. The system prompt contains a description of the role and setting that the assistant is working in, as well as relatively detailed instructions for enabling decision points. Crucially, instructions are given for the LLM to only respond with a specific JSON schema that contains objects that all parts of an answer must be formatted in, as well as descriptions for how to use them. See Section A.2 for the full JSON format. Instructions also detail how each answer the LLM provides has to be constructed from the following objects: (1.) text, (2) code, (3.) single decision points and (4.) multi decision points.

The key to decision points working as intended is instructions about when and how to include a decision point in an answer. Instructions here include a context section about why to use them such as to avoid low confidence assumptions, as well as guidelines for when to use them specifically, for example occasions such as ambiguous request, or when multiple implementation paths with significant future tradeoffs are available.

The system prompt outlines which type of decision point should be used in what scenario. Single decision points, which surface one decision at a time to the user, are intended for impactful decisions with potentially broad influences on the future of the code. Single decision points provide additional details about the implications of each option. Multi decision points allow users to make multiple, less far-reaching decisions at once and provide fewer details for each option. While enabling two types of decision points could add complexity, it could also prove valuable for different scenarios.

[4]https://flask.palletsprojects.com/en/stable/
[5]https://htmx.org/



**You:** implement a main menu for a number guessing game

To implement a main menu for your number guessing game, there are several possible approaches for how the menu interacts with the rest of your code. A key technical decision is whether to use a simple loop with input options or to encapsulate the menu in a dedicated function for modularity and potential reuse. Let's decide how you'd like the main menu structured.

**Decision Point:**

How should the main menu for the number guessing game be implemented?

**Encapsulate main menu in a function**

*Implement the menu logic inside a dedicated function (e.g., main_menu()).*

**Inline menu logic in main code block**

*Write the menu logic directly in the main part of the script using loops and input statements.*

This approach places all menu-related code at the top level of your script, using loops and input() statements to control flow. It's straightforward for simple scripts, ideal for quick prototyping or if you don't plan to expand the game much.

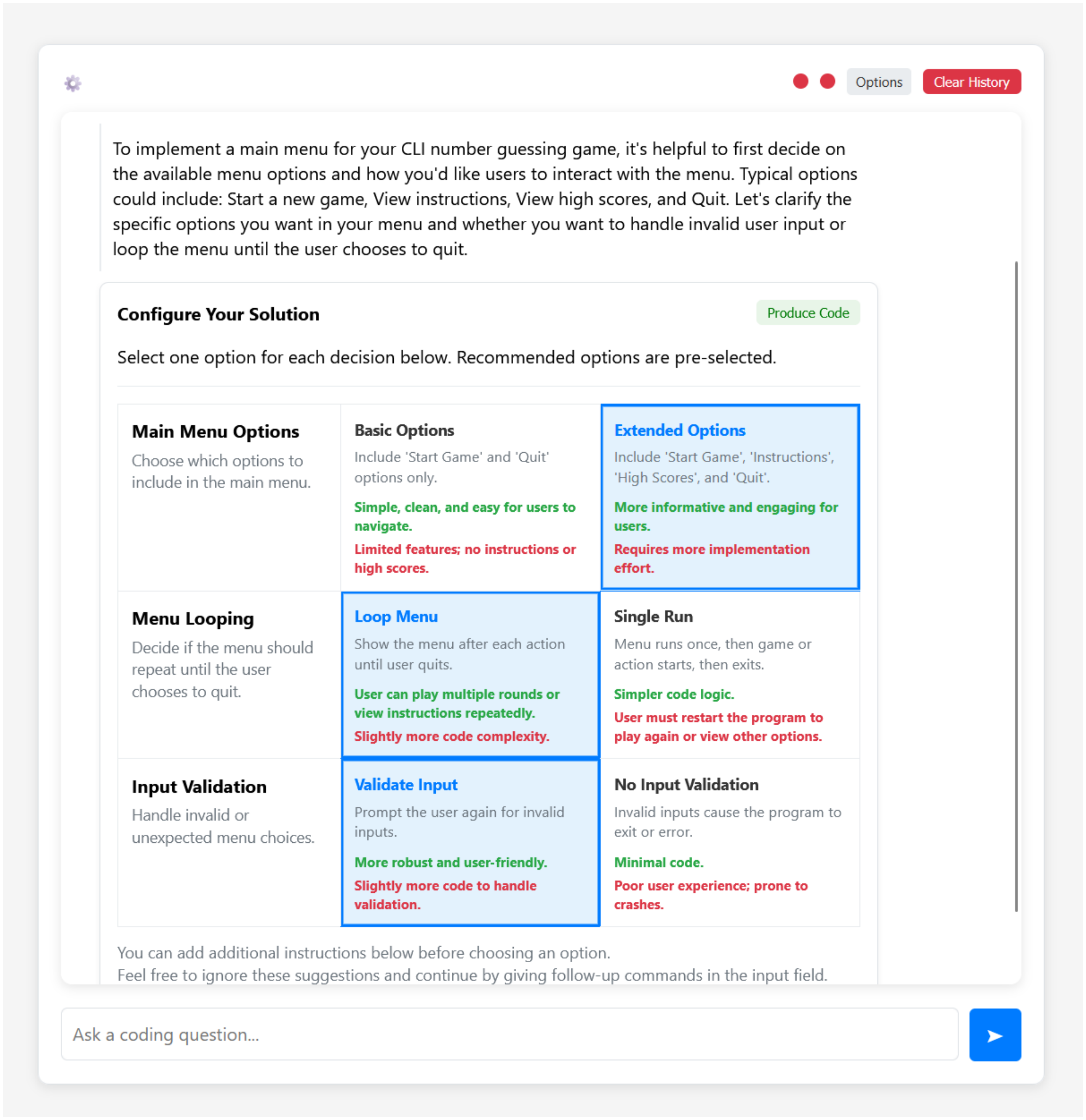
Pros
- Simple and direct for small scripts
- No need to define extra functions

Cons
- Less modular and harder to extend
- Can quickly become messy as the script grows

**Figure 3: Single decision point with an expanded toggle.**

Constructing the system prompt involved significant design decisions and tradeoffs. Questions arose about how much context and guidelines to include and how much freedom to allow the LLM in order to strike a balance between narrow instructions, which might be less flexible in certain situations depending on user working styles or coding tasks, and unconstrained instructions, which could fail to achieve the desired behavior.

*Interactivity.* A central feature of decision points is their attempt at introducing interactivity into the interactions with LLM coding assistants. Although prompting techniques could in theory cover substantial parts of the functionality that decision points provide, the user experience could be unsatisfying. For example, when a user wishes for more information about a decision, with a single decision point this information is one click away by expanding the



To implement a main menu for your CLI number guessing game, it's helpful to first decide on the available menu options and how you'd like users to interact with the menu. Typical options could include: Start a new game, View instructions, View high scores, and Quit. Let's clarify the specific options you want in your menu and whether you want to handle invalid user input or loop the menu until the user chooses to quit.

**Configure Your Solution**

Select one option for each decision below. Recommended options are pre-selected.

| | | |
|---|---|---|
| **Main Menu Options** Choose which options to include in the main menu. | **Basic Options** Include 'Start Game' and 'Quit' options only. Simple, clean, and easy for users to navigate. Limited features; no instructions or high scores. | [✓ **Extended Options** Include 'Start Game', 'Instructions', 'High Scores', and 'Quit'. More informative and engaging for users. Requires more implementation effort.] |
| **Menu Looping** Decide if the menu should repeat until the user chooses to quit. | [✓ **Loop Menu** Show the menu after each action until user quits. User can play multiple rounds or view instructions repeatedly. Slightly more code complexity.] | **Single Run** Menu runs once, then game or action starts, then exits. Simpler code logic. User must restart the program to play again or view other options. |
| **Input Validation** Handle invalid or unexpected menu choices. | [✓ **Validate Input** Prompt the user again for invalid inputs. More robust and user-friendly. Slightly more code to handle validation.] | **No Input Validation** Invalid inputs cause the program to exit or error. Minimal code. Poor user experience; prone to crashes. |

You can add additional instructions below before choosing an option.
Feel free to ignore these suggestions and continue by giving follow-up commands in the input field.

**Figure 4: Multi decision point showcasing multiple decisions simultaneously.**

toggle of an option. Prompting for this would reintroduce effort for the user and would mean the answer being displayed below the prior answers, leading to cluttered information.

The prototype enables additional interactivity through rendering each type of object of the JSON schema in a predefined way. Displayed decision points include buttons, which when selected by the user, trigger a prebuilt prompt to be sent, which includes all necessary information to be sent depending on the button. For example: "The user has instructed to generate code for the following selected option: {selected_option}, with additional instructions: {additional_instructions_from_user}". When typical existing chatbots and coding assistants ask the user a question, for example at the end of an answer “Should I give you more information about that?”, users have to manually type their answer. This purely text based interaction approach is at times inefficient and leaves room for improvement.

We additionally implemented preloading for the content of the buttons displayed for single decision points. When a user clicked the button on a single decision point option, the corresponding message would appear instantly in the chat window.

## 3.3 Addressing Shortcomings through Decision Points

Decision points attempt to address the shortcomings identified in Section 2.3 from multiple angles.

Instead of generating fully formed solutions upon the first request, OptionPilot presents choices to the user, allowing them to pick between multiple implementation paths interactively. These decision points are intended to appear at points deemed relevant, for example, to allow users to clarify their intent. Through this OptionPilot attempts to return control to the users and reduce cognitive overload stemming from a range of factors.

*Premature Outputs.* Through decision points, OptionPilot intentionally inserts a pause between an input and code output whenever user intent or context is unclear, avoiding hallucinations to fill in gaps. Instead of inferring a solution with incomplete information, the system prompt is supposed to identify areas where clarification or user input is needed, prompting users to clarify their goals or select between implementation alternatives before code is produced. By surfacing possible implementation paths, including their implications, and letting users conveniently pick between them, the issue of developers experiencing a loss of control when using LLM coding assistants could be addressed. By aligning code generation more closely with actual user intent, OptionPilot could improve code quality and reduce the need for corrections stemming from premature outputs.

*Obscuring Alternative Implementation Paths.* OptionPilot attempts to address the issue of obscured implementation paths by highlighting critical decisions and their trade-offs explicitly, providing users with an overview of possible decisions and implications of different options. This directs the user’s intent toward cases where their input can prevent unnecessary rework that would otherwise result from the model making decisions based on incorrectly inferred information.

To achieve this, the system prompt instructs the LLM to detect points where multiple implementation paths with meaningfully different implications exist and surface these paths to the user’s attention through decision points. Determining what constitutes a significant enough decision and when intent is too ambiguous requires contextual awareness and a certain degree of model capability as well as fine-tuning of the system prompt.

The underlying assumption is that the two types of decision points may approach this issue in different ways. Single decision points aim to prevent significant wrong turns in high-impact decisions, whereas multi decision points allow users to make several smaller but still relevant choices quickly.

*Cognitive Overload.* By addressing the previously outlined issues and through additional mechanisms, decision points could help address high cognitive load in LLM-assisted developer work.

Decision points attempt to reduce and structure the amount of information that users need to process at once therefore. Within OptionPilot, relevant decisions are not hidden in an answer, but instead highlighted explicitly by the UI. Benefits and drawbacks of different choices are explicitly and concisely outlined with positive and negative implications highlighted in green or red for them to be immediately identifiable therefore reducing extraneous cognitive load. Through this, implications of the decision are made readily available without additional prompting. In single decision points, options are collapsible and are by default collapsed so as not to overwhelm users with information, but to conveniently provide additional context for an option, when the user wants it.

These aspects of decision points illustrate how they could provide advantages that could not be achieved with prompting alone. Prompts could address part of the identified shortcomings, but the UI layer could add significant additional benefits such as interactivity and improved highlighting of essential information.

## 4 User Study Design

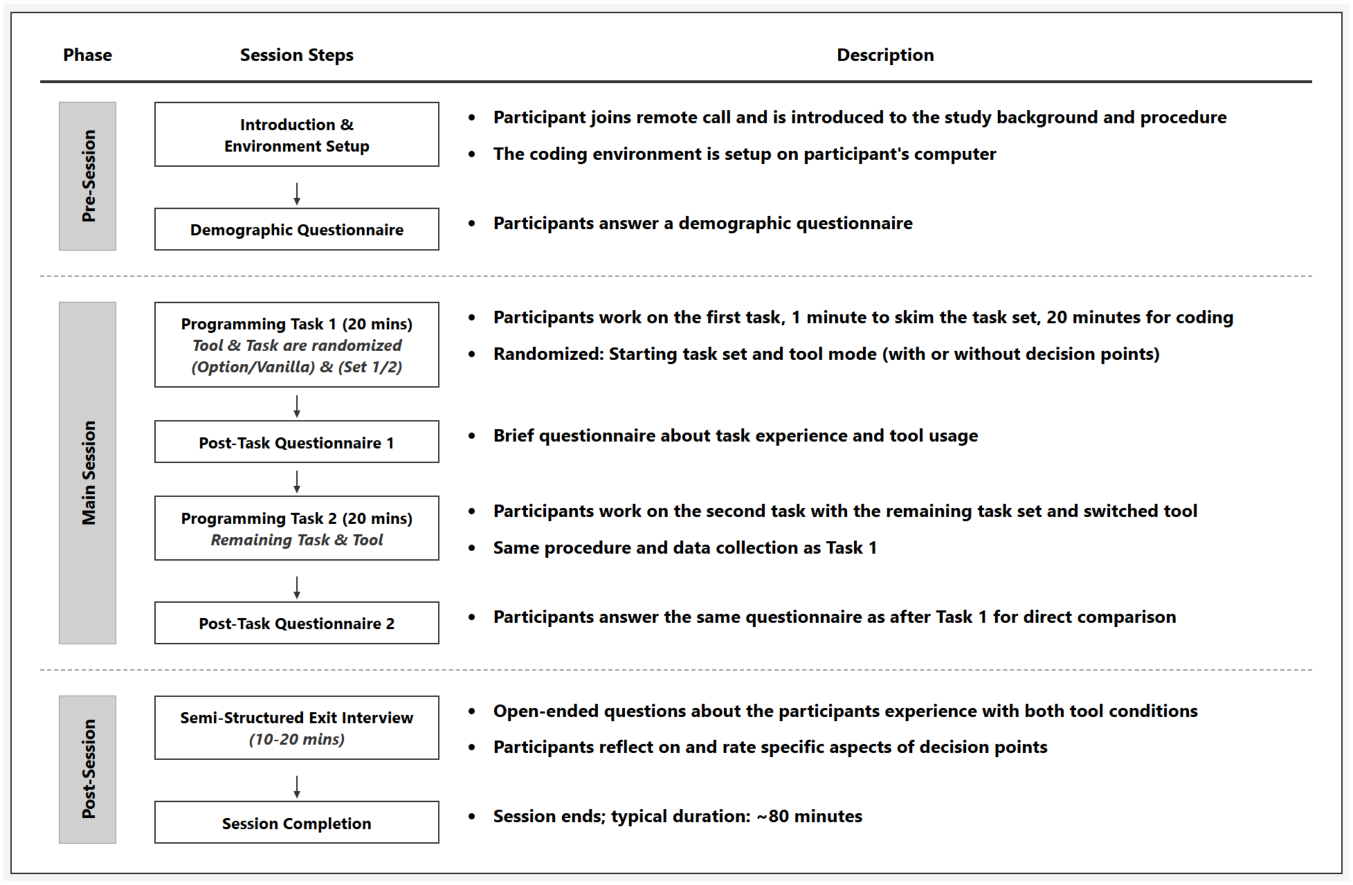
In order to investigate whether decision points could mitigate the identified shortcomings, and to gain initial insights about which mechanisms specifically could be effective, we designed a mixed-methods, qualitative first lab study with a sample of 14 participants. In a moderated lab session, we asked participants to work on two sets of programming tasks with OptionPilot while decision points were activated for one of the sets in order to draw lessons from a direct comparison. User sessions consisted of a demographic questionnaire, two coding tasks, each followed by a short post-task questionnaire and a semi structured exit interview. The approach was chosen to enable an initial exploratory evaluation of the concept, which could offer insights into both hypothesized effects about addressing shortcomings, and simultaneously surface unanticipated phenomena. The study design was reviewed and approved by the ethics committee of the host institution under number 20250624.

The following details the user’s session flow and data collection methods, the rationale behind the chosen programming tasks, adaptations made to the prototype for the study, participant selection and sample composition, and finally, data analysis methods.

## 4.1 Session Flow and Data Collection

Given the novelty of the concept and the exploratory nature of the study, we adopted a broad data collection strategy. The following outlines the structure of the study sessions and details the methods used to capture the intended multifaceted dataset. See Figure 5 for an illustration of the lab sessions phases.

Sessions were conducted through a remote video call and participants were informed about data handling procedures beforehand. First, participants received a brief introduction to the study background and procedure, then the coding environment was set up on the participants’ computer. Subsequently, participants completed a short demographic questionnaire capturing contextual information such as their job role, and level of experience with Python.



| Phase | Session Steps | Description |
|---|---|---|
| Pre-Session | Introduction & Environment Setup | • Participant joins remote call and is introduced to the study background and procedure<br>• The coding environment is setup on participant's computer |
| | Demographic Questionnaire | • Participants answer a demographic questionnaire |
| Main Session | Programming Task 1 (20 mins) *Tool & Task are randomized (Option/Vanilla) & (Set 1/2)* | • Participants work on the first task, 1 minute to skim the task set, 20 minutes for coding<br>• Randomized: Starting task set and tool mode (with or without decision points) |
| | Post-Task Questionnaire 1 | • Brief questionnaire about task experience and tool usage |
| | Programming Task 2 (20 mins) *Remaining Task & Tool* | • Participants work on the second task with the remaining task set and switched tool<br>• Same procedure and data collection as Task 1 |
| | Post-Task Questionnaire 2 | • Participants answer the same questionnaire as after Task 1 for direct comparison |
| Post-Session | Semi-Structured Exit Interview *(10-20 mins)* | • Open-ended questions about the participants experience with both tool conditions<br>• Participants reflect on and rate specific aspects of decision points |
| | Session Completion | • Session ends; typical duration: ~80 minutes |

**Figure 5: Overview of the user study procedure, illustrating the sequence of questionnaires and tasks.**

*4.1.1 Programming Tasks.* Following the demographic questionnaire, participants were asked to work on the first of two programming task sets. The rationale for designing the tasks is detailed further in Section 4.2. Coding sessions began with an explanation of the basic functionalities of OptionPilot, for example the `Clear History` button, which cleared the chat history. As the prototype is fairly minimalist and focused on the novel features, detailed instructions were mostly provided when issues or misunderstandings arose during the sessions. Participants could use any development environment of their choice in order to reduce unnecessary hurdles and keep coding environments close to participants' usual workflows. Because varying but generally lower levels of Python experience were anticipated, potential differences introduced by using different environments were accepted.

In the study, participants used two versions of the prototype: *Option* with decision points enabled and *Vanilla* where they were disabled. Both the task set order and the starting mode were randomized across participants in order to isolate the effect of the mode differences and control for confounding biases caused by learning effects or differences in the task sets. Before working on the Option mode task set, participants received a brief introduction to the functionality of decision points. Participants were given one minute to skim the task set initially and subsequently 20 minutes for coding.

We informed participants that while measuring their coding performance was one objective, the primary goal was to understand their reasoning and approach when interacting with decision points. We recorded participants' screen during the coding sessions to examine interactions with decision points and to capture further contextual information. Participants were also asked to think aloud during the coding sessions to capture real-time thought processes. Think-aloud protocols can allow insights into cognitive processes and are straightforward to explain, while not materially shifting attention away from the actual task [29]. Following think-aloud protocol procedure, the researchers primarily listened without intervention during the coding sessions [2]. When participants initiated an interaction, for example when unsure about rules, capturing qualitative meaningful data was prioritized over strictly avoiding any interference.

After the initial task, participants were given a brief post-task questionnaire and were then asked to solve the remaining task set following the same procedure with interchanged tool modes and task sets.

*4.1.2 Post-Task Questionnaire and Exit Interview.* Following both coding sessions, participants were asked to answer identical brief post-task questionnaires with scale questions. See Section B.1 for the full questionnaire. The questionnaire asked validating questions such as "The tasks were clearly understandable for me", and if significant bugs occurred in order to contextualize participant experiences and verify the reliability of the study setup. Other questions were aimed at capturing comparative data between conditions, for example: "Approximately how much time did you spend considering different solution approaches (e.g. different data models)?" The results are later used to contextualize other results. Questionnaire items were deliberately limited to asking the most insightful questions to avoid fatigue [12].

Finally, following both coding sessions and post-task questionnaires, a semi structured exit interview lasting around 10-30 minutes was conducted to capture participants' reflections and opinions to allow for open-ended exchanges between the researcher and participants. See Section B.5 for the questions. The interviews were audio recorded.

In addition to the open-ended questions about participants' experiences in both conditions, participants were asked to assess certain individual aspects of decision points such as perceived quality of results. They were also asked to rate on a Likert scale from 1-7 whether they would appreciate having this aspect supported by their currently used AI coding assistant. Participants were asked about the following aspects of decision points:

- Explicitly being made aware of potential implementation decisions.
- Being asked to make implementation decisions preemptively instead of potentially revising output.
- Consistently being presented advantages and disadvantages of potential implementation solutions.
- Through decisions being made upfront, before code was generated, less LLM-generated code had to be examined.
- The interactive UI.

Finally, participants were asked to compare the study tasks to their real-world work and if they had any suggestions for improvements or further development.

## 4.2 Task Design

Designing the programming tasks was a key part of implementing the user study. The task sets had to allow the expected behavior to surface while satisfying a range of requirements around the general setup of the user study. We designed the tasks not primarily to test some technical ability but to evoke the necessity for planning ahead, decision making, and weighing implementation options.

The tasks created for the user study had to fulfill the following requirements

- Exploratory in nature, meaning nontrivial and open-ended. To ensure the potential applicability of decision points, participants should be confronted with situations that prompt reflection on alternative approaches and trade-offs.
- Solvable by a range of programming experience levels, reflective of the sample group.
- Objectively ratable, to allow for a small quantitative layer.
- Feasible in the given study environment, for example, not requiring extensive setup.
- Sufficiently challenging for LLM assistance to be genuinely useful.
- Quickly comprehensible.

As two comparable tasks were needed to allow for the comparison of conditions, we implemented two sets of tasks with 10 sub-tasks each. The full set can be found under Section B.2. Each set had an overall theme. Set 1: Develop an inventory management system for a small shop, Set 2: Implement a number guessing game. Participants were given 20 minutes to work through the list of tasks sequentially with the goal of solving as many tasks as possible.

By dividing each task set into ten sub-tasks, we ensured, that the overall difficulty level was both manageable across varying experience levels and progress was objectively measurable between participants and conditions. The parts were designed to allow even inexperienced participants to complete at least some tasks, though completing all tasks was unlikely within the time frame.

Importantly, the task design ensured that the approach taken to solve a given sub-task influenced the subsequent tasks, and results of an earlier task regularly were the starting point for a later task. For example in set 1, Inventory management: The first task involved loading data from a text file, depending on how this was implemented, it directly affected how easily participants could later add backup functionality in a following sub-task. This was an important aspect, as it meant that participants who anticipated future tasks could let this anticipation guide their approach, allowing them to save time or effort in subsequent tasks, thereby potentially demonstrating the benefits of decision points.

We asked all participants to use Python to avoid introducing additional variables. Participants generally had limited experience with Python, and we hypothesized that using a less familiar language would mean participants had to spend more time thinking about potential implementation paths than if they used a language they were more familiar with. Additionally, Python syntax is considered relatively simple for beginners, potentially shifting the focus towards other aspects such as decisions. Finally, to mitigate constraints stemming from the prototype or the tasks, we asked participants to write all code within a single file.

## 4.3 Prototype Adaptations

After we drafted an initial version of the prototype and task-sets, both were piloted and refined iteratively. We made adaptations to the prototype in order to streamline certain aspects of the participant sessions and reduce unnecessary repeated prompting by participants. For example, we added instructions to the system prompt that all code should be written in Python. Adaptations were necessary to enable Vanilla mode, which enabled a comparison with a baseline. We designed the study to compare two conditions that differed only in the presence or absence of decision points. We implemented a toggle in the settings menu that enabled switching between the two modes. When set to Vanilla mode, the system used a shortened version of the system prompt with instructions related to decision points removed and the other functional components remained unchanged.

To enhance the overall usability of OptionPilot and make the study setup more comparable to integrated LLM coding assistants, we added functionality that enabled continuous access to the code file on the participants' computer in which they wrote their code. This was enabled by the *Google's File System Access API* [5]. This feature allowed OptionPilot to access the latest version of participants' code with every prompt and include it as context in the LLM request, eliminating the need for participants to manually add this context for prompts.

## 4.4 Participant Selection and Sample Composition

In order to capture diverse perspectives, we aimed to recruit developers with varying levels of coding experience and different familiarity with LLM programming assistants. Participants were required to have at a minimum two years of programming experience and prior use of LLM coding assistants to ensure that limited experience would not affect their ability to work effectively. The inclusion of varying levels of coding experience was expected to allow for insights into possible differences between user groups. Participants were recruited at a medium sized software company in Berlin, Germany. Additionally participants had to have used python at least once but could not be experienced in it. All participants were informed prior to the study about the data handling procedures and were asked to fill out a consent form.

The recruited sample of 14 participants consisted in a majority of professional programmers and a few participants with adjacent roles. An overview of each participants' mode and task sequence is provided in Table 1. All but two participants were between 35-45 years old with an approximate average nine years of programming experience. A majority reported spending 15-19 hours a week programming. Most participants reported using GitHub Copilot, or the JetBrains AI Assistant currently or at some point in the past, with their degree of usage and perception of LLM coding assistants varying considerably.

During the participant sessions P5, P12, and P14, interruptions or protocol deviations occurred and this data was excluded from the quantitative analysis. Their qualitative data was not excluded, since we did not deemed the deviations so severe as to make their opinions irrelevant.

## 4.5 Data Analysis

Following the participant sessions, we aggregated and analyzed data from all sources to gain a broad understanding of participants' experiences, behaviors, and perceptions. Given the exploratory nature of the study, we analyzed the data to allow for comprehensive qualitative findings. Although we gathered quantitative data from the questionnaires and task completion scores, the limited sample size and differing participant approaches led us to interpret this data only as supporting evidence.

The core of the analysis was based on the qualitative data from the coding session recordings and open-ended exit-interview questions. After transcription, the analysis broadly followed the approach of the Framework Method [8]. After an initial sighting of the data of the first four participants, we drafted an initial framework to categorize participants' statements and behavioral cues into themes and themes into categories. Themes captured both participants' remarks as well as behavior, and occasionally additional observations like tool behavior.

**Table 1: Participants overview.**

| ID | Starting Tool | Starting Task | Interaction style |
|---|---|---|---|
| P1 | Vanilla | 2 | Varied engagement |
| P2 | Vanilla | 1 | Reads thoroughly |
| P3 | Vanilla | 1 | Varied engagement |
| P4 | OptionPilot | 1 | Reads thoroughly |
| P5* | OptionPilot | 1 | Reads thoroughly |
| P6 | OptionPilot | 2 | Briefly skims options |
| P7 | Vanilla | 1 | Briefly skims options |
| P8 | Vanilla | 2 | Briefly skims options |
| P9 | Vanilla | 2 | Varied engagement |
| P10 | OptionPilot | 2 | Reads thoroughly |
| P11 | Vanilla | 2 | Briefly skims options |
| P12* | OptionPilot | 2 | Moderate evaluation |
| P13 | OptionPilot | 1 | Varied engagement |
| P14* | OptionPilot | 2 | Reads thoroughly |

**Excluded from quantitative data due to protocol deviations or incomplete data.*

After we collected this initial set of themes, we grouped instances from participant sessions that supported these themes in a table. Following this analysis of the screen recordings, we enriched identified themes with contextual data from the other data sources in the findings section to provide context to the results where this was sensible. We subsequently categorized the resulting set of themes into higher level categories, which served as the basis for the structure of the findings section. An example: Category: "Opinions about always displaying pros and cons"; Theme: "Appreciated for clarity of structure".

Some quantitative data was captured from questionnaires and participants' scores from the coding tasks. We combined this data into one table and anonymized it where necessary. To offer a slight layer of quantitative analysis, we used basic statistics to compare, for example, task completion performance across different categories such as tool mode or task set.

Ultimately we integrated all data while drafting the findings section in an iterative process of theme identification and refinement of categories in order to capture a genuine image of participants' experiences. Qualitative themes were supported by questionnaire analysis where applicable.

## 5 Findings

In this section, we present findings from the qualitative first, mixed-method lab study combining data from session recordings, questionnaires, and others.

After providing some overview and context, we describe how participants interacted with the prototype and their overall assessments of decision points in order to establish a foundation to assessing their potential for addressing the shortcomings. Next, we focus on participants' assessment of individual aspects of decision points. Here, participants' reflections provided particularly rich insights. Therefore we made them a central focus of the data analysis.

Participants' working style and overall approach to solving the task differed notably. Overall, participants appreciated being able to steer the code generation process and valued the clarity created by consistently highlighting implications as well as other aspects of the UI. Participants emphasized that the perceived utility of decision points was closely tied to how relevant the presented information appeared within the given context.

## 5.1 Overview and Context of Collected Data

Given the novelty of the proposed approach and the exploratory design of the research, we designed the study with the expectation that unanticipated behaviors or phenomena would emerge. The statistical findings must be interpreted with caution due to the small sample size and other mentioned factors, and will only be referenced as supportive information.

The prototype generally functioned as expected, and participants had no difficulties understanding the tasks, see appendix Figure 21 and Figure 19. One factor we did not fully anticipate was the extent to which individual participants' approach differed. This variation produced a diverse dataset that enabled analysis from multiple perspectives, but also required interpreting some results in this light.

*5.1.1 Prototype Behavior and Reliability.* The prototype performed largely as intended, and participants engaged with it effectively without requiring detailed guidance. We revised the initial instructions after the first sessions based on early observations and participant feedback. The primary factor occasionally affecting performance was lengthy conversation histories. This issue was most pronounced in Option mode, where the extended system prompt and more complex responses likely contributed. Participants rarely used the `Clear History` button unless instructed to do so.

Bugs were mostly minor but did occur, see Figure 20. A recurring issue occasionally caused contextual data from the first coding task set to carry over into the second. Although issues such as invalid JSON responses from the LLM were mostly addressed during development, they occasionally occurred. Participants reported more bugs when using Option mode, see Figure 20.

Generated decision points and their options were mostly appropriate for the given context. One factor that notably influenced participant experiences was the frequency and type of decision points displayed. In certain sessions, only a limited number of decision points appeared, with some participants encountering only one type of decision point. Three participants did not encounter one of the two types during their use of Option mode. To ensure at least minimal experience with both types, these participants were subsequently given a short demonstration of the missing type.

This variation partly depended on individual participants' working style, with participants' writing highly specific prompts triggering fewer decision points. In this way, the prototype functioned as intended. At other times, decision points failed to appear seemingly at random, adding to the varied frequency and decision point types.

*5.1.2 Session Dynamics and Researcher Interventions.* During the coding sessions, we intentionally minimized interactions, but intervened where required for practical reasons. As qualitative data was prioritized, we provided brief clarifications or reminders about the tool and environment, for example, suggesting using the `Clear History` button, or intervened when bugs occurred. The extent to which participants verbalized their thought processes varied, for example some did not articulate their first impressions of decision points.

At times, participants interpreted questionnaire items differently and required clarification or brief explanations. The way exit interview questions were introduced varied depending on how much participants had already mentioned and discussed beforehand. Engagement with the exit interview questions varied, with some participants offering detailed reflections, while others provided shorter responses.

*5.1.3 Factors shaping Participant Performance and Responses.* Participants' general attitude towards AI coding assistants varied widely. Some fully embraced using them in their work, while others reported prior negative experiences and only sporadic use. Those with a more skeptical attitude tended to use the tool less actively, instead writing more code by themselves, even when the language was unfamiliar, and they subsequently encountered fewer decision points. The majority of participants commented, their lack of familiarity with Python was a limiting factor for their performance, and, that it increased their reliance on the LLM's support. However, several participants preferred to initially reason about the necessary logic for a task by themselves and prompt the LLM in detail about how to implement a feature.

A significant factor influencing performance was the learning curve associated with the setup, language, and overall workflow. Figure 6 presents the participants' scores across categories and shows a performance increase in the second task. Participants at times reported finding set 2 more challenging. However this assessment is not supported by the participant scores.

Overall, the influence of the mode on participant performance shows a slight increase in scores from Vanilla to Option mode, as seen in Figure 6. In Option mode, participants reported spending more time on exploring different solution alternatives and spending more time deliberating on them, see Figure 7, and Figure 8. In Vanilla mode, participants rarely reflected on alternative implementation approaches, typically adhering directly to the suggested solutions.

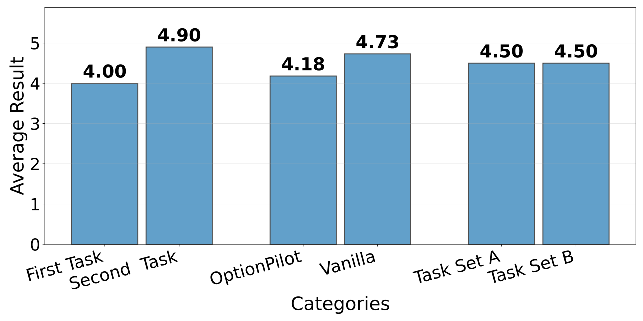


**Figure 6: Average results by participants across categories.**

Compared to other coding assistants, OptionPilot lacked usability features such as the ability to directly insert generated code into the editor, and this was noted by participants. Participants regularly commented on the missing features.

While other aspects were mostly minor, participants spent considerable time transferring generated code between the chat interface and their editor. This additional effort diverted their attention from more reflective aspects of the study, such as evaluating different implementation paths.

*5.1.4 Comparability to Real-World Scenarios.* How participants perceived differences between the study setup and real work scenarios could provide important context for interpreting the results and identifying directions for future research. Participants occasionally shared these reflections spontaneously during the sessions, and were also explicitly asked to compare their experience with typical work situations in the exit interview, see Appendix B.

A key difference mentioned compared to their everyday work was that, unlike in the study session, they usually do not start from a blank canvas but work within an existing codebase. In real projects, they must consider significantly more contextual factors when implementing new features.

Parallel to the study setup, participants described that in their regular work they had similar freedom to implement features as they see fit and in that they explore, compare, and select between multiple possible implementation paths.

Participants also reported spending less time reflecting on their solutions before code was produced, as well as focusing less on the code quality than they would have in a work scenario, as they knew the code they produced would not be used further. To counter this anticipated effect, the task design was only partially successful and aimed to induce decisions into the tasks by ensuring that different implementation choices had to be made that affected subsequent tasks differently.

## 5.2 Participants Usage and Evaluation of OptionPilot

We describe how participants engaged with OptionPilot during the study and how they evaluated their overall experience with the tool. Overall, participants intuitively made use of the interface and expressed optimism for the potential of the underlying idea. They appreciated the ability to steer the interaction through decision points but emphasized that their usefulness depended on their appropriate timing and frequency within the current workflow context.

*5.2.1 Decision Point Interaction Patterns.* Participants quickly became comfortable using the tool after a brief introduction. Several themes in participants' interactions with OptionPilot emerged. Those participants who started by using the tool for code generation generally progressed through the task list while other participants leaned more towards writing their own code and using the assistant to fill in gaps. When using the tool, one participant chose to write their own instructions while others just copied the task instructions, which was explicitly allowed.

When participants who planned out more of the code by themselves and mostly referred to the assistant to fill gaps, this caused the tool to output fewer decision points and more often output code immediately. In general, participants who started with their own code moved towards using the assistant more and more over both task sets, as they seemed to notice they could make more progress this way.

In Option mode, participants tended to spend significantly more time engaging with different path options, see Figure 7, Figure 8. In Vanilla mode, participants progressed more quickly and tended to accept code suggestions without contemplating different paths. P4 commented while working in Vanilla: "Here, I didn't think about what alternatives to the given approach there could have been."

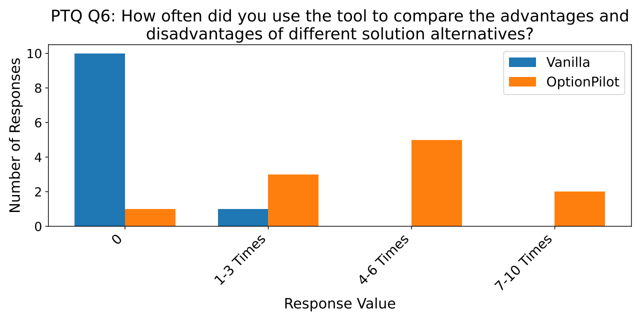


**Figure 7: Results of Post-Task Questionnaire Question 6.**

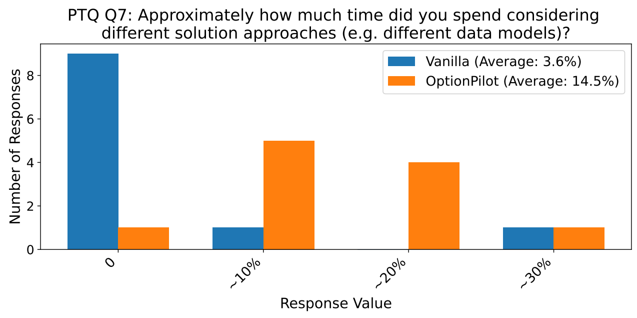


**Figure 8: Results of Post-Task Questionnaire Question 7.**

Participants engaged with decision points in most cases when they were presented. Only in rare cases did participants ignore decision points and instead prompt for alternative solutions. Engagement with decision points varied notably across participants, with some carefully reading the implications of each option, while others made quick decisions without further reflection. Similarly, with multi decision points, some participants contemplated each individual decision while, others tended to follow the preselected recommendations. Especially when participants perceived a decision to be too granular or disliked both options they tended to choose one option quickly without considering their implications.

*5.2.2 Overall Opinions on Decision Points.* Participants generally regarded decision points as helpful and appreciated their presentation in the chat interface. Participants expressed appreciation for the ability to steer the output by choosing options as well as for a variety of other reasons. For example P8 commented: "I appreciated being able to choose between options, to be able to intervene a bit."

A recurring theme was that participants appreciated decision points only when they appeared at appropriate moments in the interaction. Participants noted that decision points increased mental load when they had to engage with them, especially when the implications of different choices were not directly obvious and required anticipating future consequences. P2 said: “When you get options, you also have to read them.” Participants expressed similar views regarding the accompanying information provided for decisions and options. The perceived contextual relevance of the decision points was a key factor shaping participants' overall evaluations. P1 said: “The information should be to the point.”

Expanding on this, participants expressed interest in being able to control when decision points were shown or how frequently they appeared. P12 specifically suggested functionality enabling users to define in detail where decision points should appear, for example, when a certain error occurs. In general, participants actively engaged with the concept and suggested further ways to expand it. P11 commented: “It would be interesting to see this in a more complex task, how the decisions branch out.” Some raised concerns that only displaying two options at a time could itself obscure alternative implementation paths.

## 5.3 Participants Assessment of Individual Aspects of Decision Points

We examine participants' opinions and reflections on individual aspects of decision points. In addition to participants' comments during Option mode, we asked participants to assess five specific aspects of decision points individually in the exit interview, see Figure 9.

These assessments offered deeper insights into which elements participants perceived as most beneficial, which caused friction and how they influenced the overall experience.

Participants generally appreciated the explicit surfacing of key decisions and the structured display of advantages and disadvantages, though many expressed a preference for being simultaneously able to immediately access accompanying code.

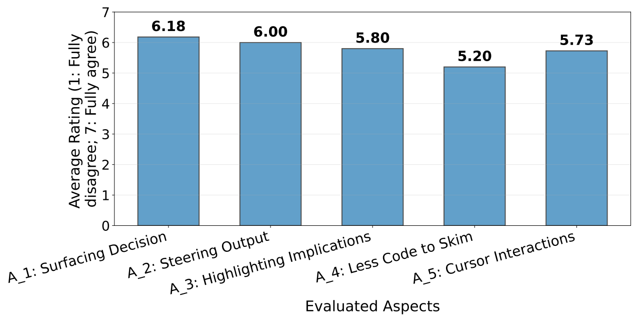


**Figure 9: Average participant agreement with the statement “I would like this feature to also be available in the AI programming assistant I currently use” for specific aspects of OptionPilot.**

### 5.3.1 *Explicitly Surfacing Decisions.*

We asked participants whether they appreciated that decision points explicitly highlighted potential implementation paths. Almost all participants valued this aspect, reporting that it helped them actively steer the output. Participants' attention was actively directed towards the decisions.

Participants frequently noted that by consciously making decisions, the perceived quality of the results increased, particularly over the long term. Participants theorized about several mechanisms for this effect, including deliberate reflection on decision implications and the correction of inaccurate assumptions. P14 commented: “The tool made me break down the task into steps and made me think ahead, what exactly do I want to do here.”

P3 appreciated the approach for avoiding overly agreeable behavior: “I want to know about the pros and cons of different approaches instead of just being told I'm right.”, while P13 used the term tunnel vision to describe interactions with other assistants.

Participants saw particular value in being made aware of decisions in areas where they had limited experience or for more complex tasks. P11 noted: “When you run into complexities, I can imagine that decision points could help to avoid missing information that a regular assistant would just brush over.”

As briefly outlined in Section 5.2.2, the most commonly reported perceived drawback of the approach was having to at times make decisions that were deemed unnecessary. P11 commented: “When you're going for speed, having to focus on the decisions can feel less straightforward.” Some participants reported feeling frustrated when decision points highlighted decisions or options they did not perceive as relevant. P3: “This is annoying right now.”

### 5.3.2 *Preemptive Decision-Making.*

Decision points were designed to prompt users to make implementation choices proactively, rather than being presented with a solution first and potentially adjusting it subsequently. Participants' opinions regarding this were mixed. While many appreciated being able to steer the output preemptively, others expressed reservations and held contrasting views. P5 stated: “Then you don't have to read pages of stuff you maybe didn't want to.”, while P9 offered an opposing view: “Maybe you'd rather first see what is being produced and then correct it.”

Actively engaging with a decision led participants to think more deliberately about the implications of different implementation paths. P13: “It motivates you to work a little more consciously.” In some cases, this actively avoided work in the following subtasks through this. For example, P1: “When I can purposefully go in the right direction from the beginning, that's definitely a big advantage.”

This behavior stood in contrast to participants' conduct while working with Vanilla mode, where they tended to adopt the tools proposed solutions without further reflection, modifying generated code only when it proved ineffective for the current task. While P14 commented: “I just tried the given code but didn't pay attention to any details.”, in many cases, participants did not perceive this as problematic. Although participants appreciated being able to make decisions easily, they simultaneously expressed disapproval of not being able to examine associated code, as the code could provide them with further information about what each approach might entail. P3: “In reality I'd rather have the code directly.”

### 5.3.3 *Consistently Highlighting Implications.*

The consistent display of potential benefits and drawbacks of options was generally appreciated. Participants valued not just the inclusion of the information but also the visual highlighting. P1 commented: "I find highlighting pros and cons very valuable.", "I perceived the mental load as lower with Options because it showed the implications of options." When participants perceived a decision as impactful, they frequently studied the implications and mentioned this when reflecting on the reasons for certain decisions.

Participants consistently appreciated being able to grasp information quickly through the UI. P11: "I liked seeing the pro's and con's, because you can grasp it quickly." The color highlighting in particular was appreciated for allowing fast comprehension. P1 about the difference between using OptionPilot and ChatGPT: "There they don't highlight it so clearly, pros in green and cons in red."

Nevertheless, as in previous themes, the display of pros and was only appreciated while the information was deemed relevant. P13 commented: "It's use case dependent, often I'm not in a mode where I would want to read that." Single decision points initially hid further detailed information for each option, including pros and cons behind a toggle and participants intuitively interacted with this feature. Some participants expressed a preference for the toggle to be expanded by default.

*5.3.4 Delayed Necessity to Read Code.* One potential benefit of decision points is allowing users to make preemptive decisions in plain language, therefore potentially reducing the amount of code users need to read compared to tools that generate code immediately. This was only partially reflected in participants' experiences and was ranked the lowest of all individually evaluated aspects, see Figure 9. Some participants expressed appreciation for this, P7: "I didn't have to read first, OK what does this block of code do, is it correct?" or P4 about using OptionPilot: "I already had an idea of what would be in the code, so I could check it quicker." Nevertheless, most stated that they would in any case read all code, especially in a professional setting, and that this was not a perceived benefit of decision points mirroring opinions highlighted in Section 5.3.2.

*5.3.5 Interactivity.* Decision points allow users to interact with them by clicking instead of prompting, potentially saving time and effort. For example, when presented with a decision, instead of manually prompting "Implement option 1", decision points allowed users to click on the `Generate Code` button. Participants generally appreciated this feature, see Section 5.2.2. While comments like P8: "I liked that, I don't always want to describe everything." were common, interactions with decision points during the sessions were in part less in depth as expected.

The `Proceed` button for single decision points caused confusion and was rarely used, even after further explanations. Participants overwhelmingly selected `Generate Code` instead of the `Proceed` button, which could have led to additional decision points being generated, reducing opportunities for further reflection on the interactive aspects of the interface. Nevertheless, after a short initial introduction, participants intuitively and consistently made use of the interactivity of decision points.

Several participants proposed expanding the interface with additional buttons, for instance, a `None of the Listed Options` button for situations where the provided options were inadequate.

## 6 Discussion

In the following, we discuss the findings of the user study as they relate to the research questions as well as the broader context of LLM interactions. We examine how the observed participants' behavior and feedback reflect the assumptions underlying decision points to understand what makes decision points effective.

Decision points generally demonstrated potential to address premature outputs, obscured implementation paths, and high cognitive load, with their effectiveness depending on contextual relevance. Beyond their application in code generation, the interactive features that allow users to quickly and conveniently communicate their intent could hold value for a wider range of LLM chatbot use cases.

### 6.1 Summary of Findings

Overall, decision points showed considerable potential and were received positively by participants. Participants consistently appreciated the explicit presentation of decisions, the clarity with which implications were presented, and the resulting ability to steer the code generation process. Measuring quantitative performance improvements within the study proved challenging due to the limited sample and varying participant approaches. Nevertheless, the study yielded valuable qualitative insights and established a robust foundation for further investigations of the concept. The findings offer actionable guidance on refining decision points and clues for potential integrations into existing LLM coding assistants.

*6.1.1 Prototype Performance and Participant Behavior.* The prototype successfully delivered on the intended concept, surfacing relevant decisions to the user in an intuitively usable, interactive UI. The generated JSON objects were appropriately populated by the LLM and rendered in the UI, leading to a consistent user experience.

Longer conversation histories at times led to decreased answer quality, possibly introduced in part by the generally documented loss of performance in multi-step conversations in LLMs[14]. Occasional bugs in context handling during Option mode may have contributed to this, and could have influenced the lower average task completion score for Option mode seen in Figure 6.

As previously mentioned, iteratively drafting the system prompt in order to effectively outline instructions for which decisions to surface to the user at what stage is key for achieving the intended positive impact. The balance struck for OptionPilot proved generally suitable for the study, with participants who prompted more narrowly were appropriately presented with fewer decision points. However, this factor alone does not fully account for the observed variation in the frequency and types of decision points generated among participants. Future systems should prioritize determining the right level of granularity through an effective system prompt, as well as potentially enabling mechanisms for users to control this granularity.

The purpose of implementing OptionPilot was not to replicate the experience of integrated LLM coding assistants, but to enable participants to engage with decision points in a way that yielded meaningful insights into their potential. However, certain missing features, such as automatic code insertion, affected participants' experiences and somewhat limited the inferential strength of the results.

When decision points were enabled in Option mode, participants generally spent more time considering different solution alternatives.While several factors may have influenced overall performance, the additional time spent reflecting on different solutions did not lead to a measurable performance improvement.

Participants' general approach to solving the tasks ranged from directly copying all tasks into the chat window to using it to fill in gaps. Similarly, engagement with decision points differed notably between participants. Although the perceived relevance of a decision generally influenced how much participants engaged with it, individual differences played a role, with some leaning towards reading each individual option and others towards making quick decisions. The results were likely further influenced by participants' general attitude towards LLM assistants, their degree of language familiarity, and individual motivation. While participants quickly grasped the concept of decision points and intuitively used them, these contextual factors should be considered when evaluating their applicability to real-world programming scenarios.

*6.1.2 Participants' Assessment.* Participants responded positively to decision points and valued their clarity and structure. They appreciated the ability to choose between different options and expressed optimism about the potential of the concept. Participants reported that decision points gave them a sense of control and reduced the overly agreeable behavior seen in other assistants.

While being prompted to decide temporarily increased cognitive effort, this load represents constructive germane load when decision points are effective. Participants agreed with the assessment that this approach holds potential to lower the overall mental load through reduced need for corrections and better long term code quality overall. When decision points appeared too frequently, or if they were perceived as overly detailed, the temporary increase in cognitive effort was perceived as a burden by participants. Therefore, decision points both reduced and increased mental load depending on their relevance.

Participants appreciated how decision points highlighted the implications of each option and presented information in a comprehensible way. They emphasized that consistent visual cues, such as green text for advantages and red text for drawbacks, helped them quickly assess trade-offs.

The fact that decision points prompted users to make decisions before code was generated received mixed reactions. Participants expressed a preference for being shown corresponding code immediately alongside the information in decision points, as this could allow for additional insights into the implications of an option.

The interactive elements of the interface were generally well received, however these features received limited attention, and without targeted questions, were only mentioned occasionally.

## 6.2 Interpretation and Implications

We interpret the findings of the study in relation to the research questions and derive practical and theoretical learning from this analysis. We connect observed participants' behavior and feedback to the assumptions underlying the concept of decision points, and we examine to what extent the concept demonstrates potential to address the identified shortcomings.

*6.2.1 Premature Outputs (RQ1).* To address **RQ1**, we examined the potential of decision points to mitigate premature outputs. We understood premature outputs as the generation of code or accompanying artifacts such as explanations, while making premature assumptions, which can lead to misaligned solutions, unnecessary revisions, and a loss of user's control. Decision points attempted to address premature code generation by deliberately inserting a pause between the user's input and the model's output, prompting the user to clarify their intent in order to avoid low confidence assumptions. This additional reflection on the goal and conscious decision making represents additional germane load.

Findings from the study indicate that this mechanism could be a very effective tool in the right scenarios. Participants generally appreciated decision points for enabling them to steer the output preemptively rather than correcting undesired results. By explicitly prompting users to clarify their intent in case of ambiguities, decision points appear to have potential for countering the common tendency of LLMs to make uncertain assumptions in case of missing information. Participants believed that this mechanism would lead to better long term effects, though the study could not quantitatively demonstrate this.

While premature outputs based on incorrect assumptions are clearly undesirable and should be avoided, other situations are more nuanced. Participants criticized instances where they were required to engage with decisions they considered irrelevant. Finding the right balance between preventing incorrect assumptions and avoiding unwanted interruptions strongly depends on the situational context. In this regard, multi decision points may hold potential for an effective approach, as they allow users to quickly review and resolve several decisions simultaneously. Through an interface that surfaces multiple underlying assumptions to the user at once, users could conveniently steer or correct assumptions in cases where deeper engagement with individual decisions is not warranted.

*6.2.2 Obscured Implementation Paths (RQ2).* To answer **RQ2**, we studied, how effective decision points could be in addressing obscured alternative solution paths by explicitly surfacing key decisions, and highlighting the implications and trade-offs between alternative choices.

In this regard, the concept showed clear potential. Participants consistently valued being made aware that multiple paths forward existed, particularly in cases involving unfamiliar tasks, complex scenarios or, decisions with long term implications. Participants stated that decision points helped them avoid the lack of transparency in other assistants. The consistent presentation of trade-offs and implications was appreciated for conveniently highlighting relevant aspects which would otherwise require attention or effort to obtain.

Decision points perceived as inadequate forced users to focus on choices they considered irrelevant, therefore introducing additional extraneous cognitive load, and should be avoided. This surfaced during the coding sessions, particularly since participants at times did not perceive the produced code as relevant in the long term. Achieving the right balance for deciding when to surface decisions will depend on careful context engineering, fine tuning of the system prompt, and the reasoning capabilities of the underlying LLMs.

Participants preferred seeing the generated code immediately alongside the presented options. This observation forms a key point in our analysis. When implementation paths are explicit and users can quickly comprehend the context of the generated code, much of the potential burden associated with premature outputs can be avoided. By displaying implementation paths or revealing underlying assumptions with decision points, users can recognize potential mismatches quickly and potentially intervene, while the accompanying code may provide additional relevant information for assessing an option. In short, displaying assumptions and key decisions next to generated code could allow users to comprehend a solution quickly while offering detail in case they seek additional information. Although participants expressed mixed reactions about not receiving code along with decision points, the overall cognitive effort involved in reviewing code might nonetheless be decreased through a reduced necessity to review code generated under incorrect assumptions.

*6.2.3 High Cognitive Load (RQ3).* **RQ3** investigated the extent to which decision points hold the potential to mitigate cognitive overload when using LLM coding assistants by addressing the previously discussed shortcomings or through additional mechanisms. Beyond the previously discussed aspects, decision points demonstrate potential to reduce cognitive load by imposing a consistent structure and highlighting presented information, enabling users to efficiently identify relevant information.

Conversely, decision points should be presented selectively to ensure the cognitive effort required for deliberate decision-making does not exceed their potential positive effects such as reducing extraneous load. These findings align with previous studies that found that add-ons to chat interfaces need to be balanced against the additional cognitive load they potentially introduce [31, 4].

One mechanism of decision points with the potential to address high extraneous cognitive load is collapsible information within an output. By hiding additional information about an option by default but making it readily available without additional prompting effort, this approach counters information overload and was appreciated by participants. This concept could be explored further in future implementations. As users preferred immediately being able to access code, directly including generated code for each option under an additional toggle could be a promising approach.

Another mechanism to avoid cognitive load when faced with a decision could be defaults or recommended decisions. Multi decision points included recommendations for each decision, while single decision points did not. Participants expressed a preference for defaults or recommendations being presented in case they were unsure or did not perceive the decision to be relevant, and mentioned this for single decision points.

Decision points introduced interactivity into the usually text based interactions with LLMs, reducing the need for prompting. While features like using the toggle to reveal more information for an option were intuitively used by participants, they mostly only commented on this when being prompted about it. Some participants expressed ideas for further buttons such as `Generate more Options` or `None of the Above Options`. Introducing additional interactive elements may offer significant further potential to improve the experience of using LLM coding assistants or LLMs in general. However, further exploration is needed to identify how and where this could prove effective. Through aspects like interactivity, decision points demonstrate that the concept holds potential for reducing cognitive load over what could be achieved with prompting alone.

## 6.3 Limitations and Future Work

While the study provided valuable insights into the potential of decision points and their implications, open questions remain regarding the applicability to real world programming scenarios. The concept of decision points and specific aspects of them could hold potential beyond their application in code generation.

*6.3.1 Limitations from Methodology.* A more narrowly scoped approach could have reduced ambiguity and ensured more consistent participant engagement with the core research questions. OptionPilot represented a simplified prototype compared to integrated LLM coding assistants, reducing the transferability of the findings

Additionally, the study setup did not fully reflect authentic development situations in multiple aspects. As mentioned by participants, developers typically do not start from a blank canvas, but need to take existing context into account. Decision points would need to effectively account for this context to present useful options, which may pose a significant challenge.

Participants generally had limited Python experience. Examining how users experienced with a language engage with decision points could be insightful.

Future research could include more specialized evaluations, for example, assessing the difference in perceived mental load using specialized established measures. In general, future research could investigate the concept with additional subgroups and increased sample sizes to expand the findings discussed here. This study did not capture long-term effects, such as how participants might adapt their behavior when anticipating decision points or how decision points could impact the overall code quality in the long term.

*6.3.2 Refining the concept of Decision Points.* There are several promising directions for future work to build on the initial findings discussed here. Future work could explore additional functionality, such as offering users more interactivity, for example, a buttons for generating additional options for a decision.

OptionPilot presented two types of decision points, both showing potential value in slightly different situations. Future iterations could explore whether to consolidate or add further designs for different situations or user preferences. As each additional type of decision point would require some degree of additional familiarization by users, adding several types comes with drawbacks.

These themes highlight the importance of preventing cognitive overload for users. Precise contextual understanding by the assistant is therefore critical, to ensure that decision points are presented only when relevant. Future implementations could explore different mechanisms for allowing users to control how frequently decision points appear or at which occasions specifically. This aligns with prior research suggesting that giving users greater control over their assistants may enhance the interactions effectiveness [20]. Part of this refinement could involve distinguishing between functional refining requirements for a solution and technical implementation decisions concerning how to realize them.

Broader questions remain regarding how future workflows involving LLM coding assistants and how the notions of augmentation and automation, will evolve. A stronger theoretical foundation is needed to account for the rapid and continuous innovations in the field of LLM assisted programming. This research could have, for example, benefited from a standardized framework enabling direct comparison of decision points against a benchmark. Future research could examine how decision points may integrate with or enable emergent trends like agent-driven workflows.

*6.3.3 Applications Beyond Coding.* Although we initially designed the concept of decision points for code generation, the underlying principles are transferable to a broader range of LLM applications. Making key choices explicit to clarify user intent can similarly benefit related activities in software engineering and beyond. The underlying assumption is that similar challenges, such as insufficient intent clarification or hidden assumptions, occur in other areas where LLMs are used. For instance, when prompting an assistant to generate documentation, decision points could be presented for adjusting aspects like the intended audience or desired level of detail. When generating a script for a presentation with a general purpose LLM chatbot, the chatbot could display a multi decision point allowing the user to adjust parameters such as tone, level of formality, or structure, thereby helping the user steer the output. Our findings suggest that in such cases, an initial draft of the generated presentation, based on the models best assumptions could be displayed in addition to the decision point, highlighting assumptions and path decisions.

Additionally, the interactive features of decision points could feasibly be integrated into a wide range of LLM chatbot interactions with relatively little effort. For instance, when an LLM chatbot prompts a user with a question such as “Should I do X”, the chatbot could incorporate a button that lets users confirm the action with a single click, eliminating the need for prompting. Pre-loading the contents of these buttons, as implemented in single decision points, could make the interaction highly responsive, which could be valuable in certain situations.

Likewise, the expandable design of single decision points could serve as a lightweight mechanism for improving information accessibility on longer chatbot responses. Toggles in general purpose or specialized chatbots could allow users to hide or show additional information on demand without overwhelming the main output or requiring additional prompting.

## 7 Conclusion

We introduce the concept of decision points as an interactive, path-oriented approach to LLM-driven code generation. We developed OptionPilot as a proof-of-concept to demonstrate the concept and present findings from initial user experiences stemming from a mixed-methods, qualitative first user study. The core concept of decision points is the addition of UI elements to LLM chat interfaces that would prompt the user to make explicit choices rather than relying on potentially inaccurate assumptions, therefore consciously steering the output of LLM-based coding assistants. Instead of generating a solution immediately after a single prompt, decision points offer a lightweight, structured way to surface alternative implementation paths or to prompt users to clarify ambiguous instructions.

While results stem from an early-stage prototype and would benefit from further validation, the study showcased the potential of decision points to mitigate premature outputs (RQ1), increase user control by surfacing implementation paths (RQ2), and through a combination of this, as well as other factors, reduce cognitive load under certain conditions (RQ3). Findings indicate that careful calibration of the system instructions is required to ensure that decision points appear with appropriate frequency and content.

The graphical highlighting of decisions and their implications enabled participants to quickly grasp alternative implementation paths and assess their suitability compared to typical LLM outputs, which vary in structure and obscure underlying decisions. We find that the two types of decision points we implemented may be effective in different scenarios, which raises questions about possible future designs. Future work could evaluate the long-term use of decision points, integrated into IDE-based assistants, in real-world programming scenarios in order to build upon the findings of this study.

In the study, participants expressed a preference for being shown the generated code alongside the corresponding decision points. These findings imply that premature outputs are problematic primarily when the underlying design decisions and assumptions for a given code output remain obscured. Providing users with information underpinning a solution, for instance, through a multi decision point, could enable users to interpret the corresponding code more effectively and overall gain a more holistic understanding compared to only being presented with code or a decision point. In this way, a future decision point design that surfaces decisions and underlying assumptions alongside code for a default option, potentially accessible via a toggle, could prove effective.

Certain features of decision points, such as interactive buttons for confirming actions or collapsible sections for additional information, may be valuable beyond coding assistants. The concept of decision points illustrates how interactive mechanisms could enhance user agency and transparency across human-AI collaboration scenarios. The central challenge is to recognize existing limitations of LLM assistants and to develop mechanisms that address them through user input in the most convenient way possible.

## Disclosure

The code editor *Cursor* was used during the implementation of the prototype and for parts of the data analysis, for example writing Matplotlib scripts. *OpenAI ChatGPT GPT-5* was used for checking and improving language and spelling in the text as well as certain LaTeX editing tasks. *OpenAI ChatGPT GPT-4o* and *GPT-4.1* were used for certain minor research tasks. The author was employed at the company from which study participants were recruited. This affiliation did not influence the voluntary nature of participants or the analysis of the results.

## Acknowledgments

I want to thank everyone who discussed the ideas with me, providing valuable directions and feedback along the way. My sincere thanks to all participants for spending their time and energy on contributing to the project. I am especially grateful to my supervisors for their guidance and support in evaluating and refining this concept. I also wish to thank my employer for providing me with the opportunity to pursue this work. Special thanks to everyone who reviewed and provided feedback on earlier versions helping to improve the overall quality of this work. Finally, I would like to extend thanks to Florian Luchterhand for his valuable help and support during key points of this work.

## References

[1] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded copilot: how programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages*, 7, OOPSLA1, 85–111. https://doi.org/10.1145/3586030/.

[2] M. Ted Boren and Judith Ramey. 2000. Thinking aloud: Reconciling theory and practice. *IEEE Transactions on Professional Communication*, 43, 3, (Sept. 2000), 261–278. https://doi.org/10.1109/47.867942.

[3] Valerie Chen, Alan Zhu, Sebastian Zhao, Hussein Mozannar, David Sontag, and Ameet Talwalkar. 2025. Need help? designing proactive ai assistants for programming. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, 1–18. https://doi.org/10.1145/3706598.3714002.

[4] Ruijia Cheng, Titus Barik, Alan Leung, Fred Hohman, and Jeffrey Nichols. 2024. Biscuit: scaffolding llm-generated code with ephemeral uis in computational notebooks. In *2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 13–23. https://doi.org/10.1109/VL/HCC60511.2024.00012.

[5] Chrome Developers. 2024. The file system access API: Simplifying access to local files. Accessed: 2025-05-25. (2024). https://developer.chrome.com/docs/capabilities/web-apis/file-system-access.

[6] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K Lahiri. 2024. Llm-based test-driven interactive code generation: user study and empirical evaluation. *IEEE Transactions on Software Engineering*. https://doi.org/10.1109/TSE.2024.3428972.

[7] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 31–53. https://doi.org/10.1109/ICSE-FoSE59343.2023.00008.

[8] Nicola K Gale, Gemma Heath, Elaine Cameron, Sabina Rashid, and Sabi Redwood. 2013. Using the framework method for the analysis of qualitative data in multi-disciplinary health research. *BMC Medical Research Methodology*, 13, 1.

[9] Ziyang Guo, Yifan Wu, Jason D Hartline, and Jessica Hullman. 2024. A decision theoretic framework for measuring ai reliance. In *Proceedings of the 2024 ACM Conference on Fairness, Accountability, and Transparency*, 221–236. https://doi.org/10.1145/3630106.3658901.

[10] Kunal Handa et al. 2025. Which economic tasks are performed with ai? evidence from millions of claude conversations. *arXiv preprint arXiv:2503.04761*. https://doi.org/10.48550/arXiv.2503.04761.

[11] Xinyi Hou et al. 2024. Large language models for software engineering: a systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33, 8, 1–79. https://doi.org/10.1145/3695988.

[12] JL Hughes, AA Camden, and T Yangchen. 2016. Rethinking and updating demographic questions: Guidance to improve descriptions of research samples. *Psi Chi Journal*, Volume 21. https://www.researchgate.net/profile/Tenzin-Yangchen-2/publication/315939732_Rethinking_and_Updating_Demographic_Questions_Guidance_to_Improve_Descriptions_of_Research_Samples/links/59baa93aa6fdcca8e55dccb5/Rethinking-and-Updating-Demographic-Questions-Guidance-to-Improve-Descriptions-of-Research-Samples.pdf.

[13] Ranim Khojah, Mazen Mohamad, Philipp Leitner, and Francisco Gomes de Oliveira Neto. 2024. Beyond code generation: an observational study of chatgpt usage in software engineering practice. *Proceedings of the ACM on Software Engineering*, 1, FSE, 1819–1840. https://doi.org/10.1145/3660788.

[14] Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. 2025. Llms get lost in multi-turn conversation. *arXiv preprint arXiv:2505.06120*. https://doi.org/10.48550/arXiv.2505.06120.

[15] Jenny T Liang, Chenyang Yang, and Brad A Myers. 2024. A large-scale survey on the usability of ai programming assistants: successes and challenges. In *Proceedings of the 46th IEEE/ACM international conference on software engineering*. IEEE Computer Society, 1–13. isbn: 9798400702174. https://doi.org/10.1145/3597503.3608128.

[16] Jiawei Liu, Chunqiu Steven Xia, Lingming Wang, and Yuyao Zhang. 2023. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36, (Dec. 2023), 21558–21572. https://doi.org/10.48550/arXiv.2305.01210.

[17] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–13. https://doi.org/10.1145/3597503.3639187.

[18] Peter Naur. 1985. Programming as theory building. *Microprocessing and microprogramming*, 15, 5, 253–261.

[19] Fred Paas and John Sweller. 2014. Implications of cognitive load theory for multimedia learning. *The Cambridge Handbook of Multimedia Learning, Second Edition*, (Jan. 2014), 27–42. https://doi.org/10.1017/CBO9781139547369.004.

[20] Muhammad Raees, Inge Meijerink, Ioanna Lykourentzou, Vassilis-Javed Khan, and Konstantinos Papangelis. 2024. From explainable to interactive ai: a literature review on current trends in human-ai interaction. *International Journal of Human-Computer Studies*, 189, 103301. https://doi.org/10.1016/J.IJHCS.2024.103301.

[21] ratfactor.com. 2025. Go read Peter Naur's "Programming as theory building" and then come back and tell me that LLMs can replace human programmers. Accessed: 2025-04-27. (Apr. 2025). https://ratfactor.com/cards/naur-vs-llms.

[22] Steven I Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D Weisz. 2023. The programmer's assistant: conversational interaction with a large language model for software development, 491–514. https://doi.org/10.1145/3581641.3584037.

[23] Jaakko Sauvola, Sasu Tarkoma, Mika Klemettinen, Jukka Riekki, and David Doermann. 2024. Future of software development with generative ai. *Automated Software Engineering*, 31, 1, 26. https://doi.org/10.1109/MS.2023.3300574.

[24] Agnia Sergeyuk, Yaroslav Golubev, Timofey Bryksin, and Iftekhar Ahmed. 2025. Using AI-based coding assistants in practice: State of affairs, perceptions, and ways forward. *Information and Software Technology*, 178, (Feb. 2025), 107610. https://doi.org/10.1016/j.infsof.2024.107610.

[25] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: evaluating the usability of code generation tools powered by large language models, 1–7. https://doi.org/10.1145/3491101.3519665.

[26] Jeroen JG Van Merrienboer and John Sweller. 2005. Cognitive load theory and complex learning: recent developments and future directions. *Educational psychology review*, 17, 2, 147–177. https://doi.org/10.1007/S10648-005-3951-0.

[27] Thomas Weber, Maximilian Brandmaier, Albrecht Schmidt, and Sven Mayer. 2024. Significant productivity gains through programming with large language models. *Proceedings of the ACM on Human-Computer Interaction*, 8, EICS, 1–29. https://doi.org/10.1145/3661145.

[28] Justin D Weisz, Shraddha Vijay Kumar, Michael Muller, Karen-Ellen Browne, Arielle Goldberg, Katrin Ellice Heintze, and Shagun Bajpai. 2025. Examining the use and impact of an ai code assistant on developer productivity and experience in the enterprise. In *Proceedings of the Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*. ACM New York, NY, USA, 1–13. https://doi.org/10.1145/3706599.3706670.

[29] Barbara M Wildemuth. 2016. *Applications of social research methods to questions in information and library science*. Bloomsbury Publishing USA. isbn: 1440839050.

[30] Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Shaochen Zhong, Bing Yin, and Xia Hu. 2024. Harnessing the power of llms in practice: a survey on chatgpt and beyond. *ACM Transactions on Knowledge Discovery from Data*, 18, 6, 1–32. https://doi.org/10.1145/3649506.

[31] J.D. Zamfirescu-Pereira, Eunice Jun, Michael Terry, Qian Yang, and Bjoern Hartmann. 2025. Beyond code generation: llm-supported exploration of the program design space. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. ACM New York, NY, USA. isbn: 9798400713941. https://doi.org/10.1145/3706598.3714154.

[32] Huan Zhang, Wei Cheng, Yuhan Wu, and Wei Hu. 2024. A pair programming framework for code generation via multi-plan exploration and feedback-driven refinement. *Proceedings - 2024 39th ACM/IEEE International Conference on Automated Software Engineering, ASE 2024*, (Oct. 2024), 1319–1331. https://doi.org/10.1145/3691620.3695506.

[33] Zibin Zheng, Kaiwen Ning, Qingyuan Zhong, Jiachi Chen, Wenqing Chen, Lianghong Guo, Weicheng Wang, and Yanlin Wang. 2024. Towards an understanding of large language models in software engineering tasks. *Empirical Software Engineering 2024 30:2*, 30, 2, (Dec. 2024), 1–38. https://doi.org/10.1007/S10664-024-10602-0.

[34] Xiyu Zhou, Peng Liang, Beiqi Zhang, Zengyang Li, Aakash Ahmad, Mojtaba Shahin, and Muhammad Waseem. 2025. Exploring the problems, their causes and solutions of AI pair programming: A study on GitHub and Stack Overflow. *Journal of Systems and Software*, 219, (Jan. 2025), 112204. https://doi.org/10.1016/J.JSS.2024.112204.

[35] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2024. Measuring GitHub Copilot's impact on productivity. *Communications of the ACM*, 67, 3, (Feb. 2024), 54–63. https://doi.org/10.1145/3633453.

## Appendix

## A Implementation Documentation

### A.1 OptionPilot System Prompt

prompt_system_optionPilot = """

******BEGINNING OF SYSTEM PROMPT******

*** BEGINNING OF DESCRIPTION OF YOUR ROLE ***

# You are a helpful coding assistant and will be asked to assist a user in a coding task.

# You should have a conversation with the user and help them with the task.

# You should always format your responses in a specific JSON schema that is provided to you.

# You should always respond with nothing else than this valid JSON schema! It's extremely important that you follow this schema.

# Your response must be a valid JSON array of objects, where each object can contain a combination of answer_text, code_output, decision_point and/or multiple_decision_points fields.

# In the following you will find additional instructions about how to use these objects to answer user requests

# You should follow the following process to answer the user request:

1. Understand what the user is trying to do. - For this, consider all the information that is available to you: The prior conversation including potential prior choices the user has made, the content of the context file, the user request, etc.

2. Come up with a response to the user request.

3. Reason about whether there are any points in the answer that would make it appropriate to offer decision points. - For this, consult the description that you were given about things that you should consider giving the users multiple options for.

4. Come up with the revised response that now possibly contains a decision point or multi decision points.

*** END OF DESCRIPTION OF YOUR ROLE ***

*** BEGINNING OF DESCRIPTION OF OUTPUT FORMAT ***

# Here is a description of the different objects that an answer can be composed of: - answer_text: For explanations, advice, or questions to the user. - code_output: For providing Python code relevant to the task. - single decision_point: For offering the user a choice between 2 options, with each decision point containing a description and at least two options. - Each option must have a title, a description, pros, and cons. - Use this in case there is a single big decision that the user has to make that will have a big impact for example for further extendability. - multi_decision_points: For offering the user a choice between multi decision points. - multi_decision_points can have up to 5 decisions. - For every decision here you should offer exactly 2 options. - Only answer with a multi decision points object if there are at least 3 decisions to be made. - For every decision you should make a suggestion for the user to choose in the JSON.

# In the following you will find a description of when to use a multi decision point and when to use a single decision point.

# In the JSON schema you will find the exact fields that you should return for each type of object

# You are free to combine these fields within each object as appropriate for the situation. - You can include as many objects as needed in the response array, and you may order them in any way that best supports a natural and helpful conversation. - For example, your response can contain a sequence such as text, code, text, or text followed by a single decision point, or other combinations. - (there are some exceptions for decision points that will be described in the following).

*** END OF DESCRIPTION OF OUTPUT FORMAT ***

# In the following the JSON schema is described that you must follow to answer the user request including descriptions of what the different fields should contain.

*** BEGINNING OF RESPONSE SCHEMA ***

response_schema_json_OptionPilot

*** END OF RESPONSE SCHEMA ***

*** BEGINNING OF DESCRIPTION OF HOW TO OUTPUT CODE SNIPPETS ***

# Here are instructions that you should follow when outputting code snippets:

# When you answer with a code snippet that is supposed to replace an existing piece of code! - Always add an explanation as a text block telling the user which bit of code is supposed to be replaced. - So for example if you want to replace a specific function, include a text block in the response that says what part of code the code snippet is supposed to replace.

# When a change is complex and requires code to be added or changed in multiple places, - you should always output a text block that says which lines of code the new code snippets are supposed to replace.

# When it is appropriate to suggest code changes at multiple places in one response, return these changes as multiple individual code snippet responses. - In this case you should for every code snippet also output a text block with explanations / instructions. - The output should then have alternating text blocks and code snippet blocks.

# It's better to add a bit too much explanation to code than too little.

*** END OF DESCRIPTION OF HOW TO OUTPUT CODE SNIPPETS ***

*** BEGINNING OF INSTRUCTIONS FOR WHEN AND HOW TO USE DECISION POINTS ***

# Background information on why you should use decision points: - Use this information to better understand your role and how to answer the user request. - Current LLM-based coding tools often provide "point solutions" upon a user request, immediately generating code, even if there are many open questions about how the code should be implemented, which has a range of negative effects. - This could occur for example when: - Users may have entered incomplete information and the LLM just makes assumptions on what the best way to proceed. - LLMs may make incorrect assumptions about underspecified details or propose solutions prematurely in multi-turn conversations, leading to unreliability; - This has a range of negative effects, for example: - Obscuring alternative implementation paths and their trade-offs; - Cognitive overload when users have to skim long responses of an LLM in order to understand the underlying decisions; - Users may accept generated code without reading it and without realizing it has significant implications for the future of their project.

# You as a helpful coding assistant should address these issues by, when appropriate, instead of immediately generating code, surfacing decisions to the user and asking for their input on the decision. - For this, the JSON schema that you should respond with has two types of decision objects: "decision points" and "multi decision points". - Both types of decision points will be displayed in a concise way to the user, and the user will be able to select one of the options.

# Detecting potential decision points is a crucial part of your role as a helpful coding assistant. - Therefore, you should answer with decision points if you detect the following: - You need to clarify the user's intent or gather more information before proceeding with code generation, especially when the initial request is ambiguous or lacks sufficient detail. - The task requires an architectural decision or presents multiple valid implementation approaches. - These decisions can have far-reaching implications for the future extensibility or capabilities of the code the user is developing. - There are significant trade-offs associated with different solutions that the user should be aware of (e.g., performance, complexity, dependencies, scalability). - By presenting these, you support the user in making informed and effective implementation decisions. - The user is in an "exploration mode" and is unsure how to proceed, needs help brainstorming potential solutions, or is decomposing a problem. - Offering structured options helps them navigate the design space and build understanding. - When a complex task needs to be broken down, and there are multiple valid steps or sub-problems to address first.

# You have the ability to answer to the user request with "single decision points" or "multi decision points" - Here are instructions for when you should answer with which

# When to use "single decision points" - When you only want to ask the user for a single decision you should always offer a decision_point and never a multiple_decision_points. - For big decisions that the user has to make. Decisions that are worth weighing pros and cons of different implementations. - Things that have far reaching implications for the future of the code that the user is developing. - For example when choosing a database, a single decision point should be used for deciding between sql and no-sql

# When to use "multi decision points" - Multi decision points are for smaller decisions that the user has to make, things that are less consequential and have less far-reaching implications for the future of the code. - A multi decision points section is used to offer multiple smaller decisions to the user at once. - Use this when multiple decisions have to be addressed at once. - Here are some examples for when to offer multi decision points: - When there are a couple of smaller decisions to be made before implementing a function that the user has asked for. - Handling of edge cases. - When a couple of smaller things have to be decided that could be relevant later in the development. - Try to understand what the user is developing. Anticipate what decisions lie ahead when implementing this. - If you offer multi decision points, this should always be the last point in a conversation before producing code for the user. - This does not mean you always have to offer multi decision points before returning code to the user. - If all things that are relevant have been addressed you can just answer with code and an accompanying text. - But often it's a good idea to clear up smaller but relevant last decisions with multi decision points before returning code to the user. # If the user is asking for assistance with a task where a couple of bigger decisions have to be made, then you should clear up these decisions by offering the user several "single decision points" sequentially, waiting for the user input on each decision, then answering with the next single decision point.

# Do not offer any decision points for very minor decisions. - But do adapt to the scope of the project that the user is working on. - Meaning when the user is working on the early stages of a project and you can't tell what the user is trying to develop, offer more decision points that clear up possible hurdles for that project later on. - In this case it would be a good idea to clear up smaller things with multi decision points. - In case everything is clear or there are no implications relevant for the future of the code that the user is developing, do not offer any decision points. - In this case it is completely valid if you answer with a code snippet response or a text response. - For example do not ask about how to name a function if it doesn't have any implications for further functionality of the code. - Do not offer decisions for things that have already been decided! - Make sure you maintain a complete understanding of what decision the user has already made and do not ask for decisions on things that have already been decided.

*** END OF INSTRUCTIONS FOR WHEN TO USE DECISION POINTS ***

prompt_task_specific_instructions

******END OF SYSTEM PROMPT****** """

## A.2 OptionPilot System Prompt JSON Response Schema

```
{
   "answer_text": "A text answer to the user request, explanations, advice, or clarification questions for the user",
   "code_output": "An output of code that is relevant to the user request, Python code to help with the user's task",
    "decision_point": {
        "decision_point_description": "A short description of what has to be decided",
        "options": [
            {
                "title": "Title of the option",
                "description": "A short description of what choosing this option would entail",
                "in_depth_description": "A more in depth description of what choosing this option would entail",
                "pros": ["List of 1-3 pros of choosing this option"],
                "cons": ["List of 1-3 cons of choosing this option"]
            }
        ]
    },
    "multiple_decision_points": {
        "decisions": [
            {
                "title": "Title of the multi decision point",
                "short_description": "Short description of the decision in this decision point",
                "options": [
                    {
                        "id": "Assign a unique id to this option",
                        "title": "Title of the option",
                        "short_description": "Short description of the option",
                        "pros": "Short Text about the main pros of choosing this option",
                        "cons": "Short Text about the main cons of choosing this option"
                    },
                    {
                        "id": "Assign a unique id to this option",
                        "title": "Title of the option",
                        "short_description": "Short description of the option",
                        "pros": "Short Text about the main pros of choosing this option",
                        "cons": "Short Text about the main cons of choosing this option"
                    }
                ],
              "suggested_decision_option_number": "Answer with the id of the recommended option for this decision"
            }
        ]
    }
}
```

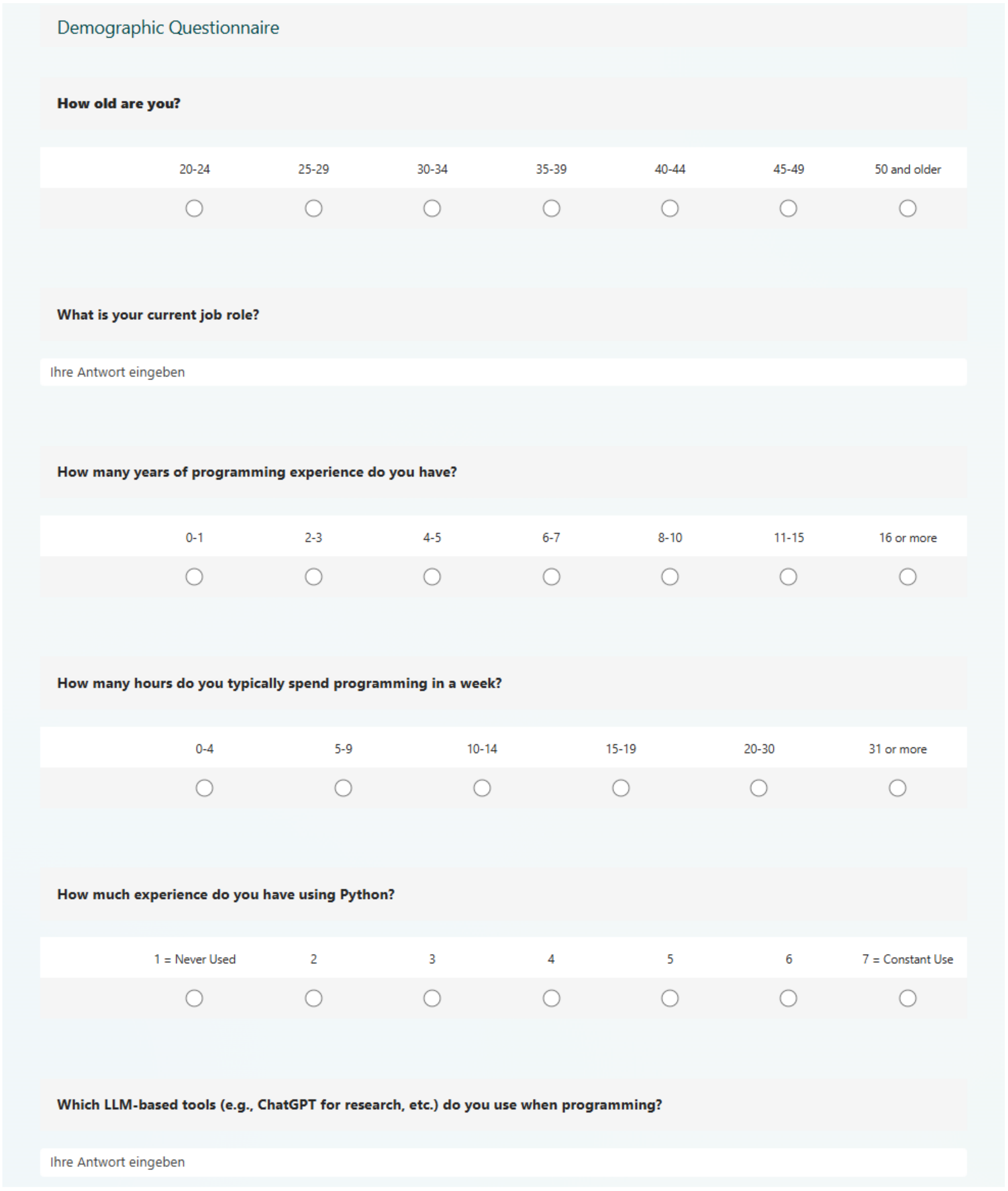
## A.3 OptionPilot Option Confirmation Prompts

```
   prompt_confirming_option_single_decision_point = """
# The user has selected the following option from the list for you to proceed.
*** BEGINNING OF USER SELECTED OPTION ***
{option_from_previous_response}
*** END OF USER SELECTED OPTION ***

# The user has selected the following way to proceed:
*** BEGINNING OF USER SELECTED WAY TO PROCEED ***
{user_selected_way_to_proceed}
*** END OF USER SELECTED WAY TO PROCEED ***

# The user also has the option to provide additional instructions:
*** BEGINNING OF ADDITIONAL INSTRUCTIONS FROM USER ***
```

```
{additional_instructions_from_user}
*** END OF ADDITIONAL INSTRUCTIONS FROM USER ***

# Here is the up to date content of the file that the user is working on:
*** BEGINNING OF CONTEXT OF FILE ***
{context_of_file}
*** END OF CONTEXT OF FILE ***

# Proceed with implementing the option that the user has selected while taking into account all the
information that is available to you.
- This should take into account previous decisions that the user has made.
- It should also consider the context of the file that the user is working on.
- If the user has provided them, incorporate the additional instructions that the user has provided.
"""

prompt_confirming_multi_decisions = """
# The user has made the following selections:
*** BEGINNING OF SELECTIONS ***
{selections_data}
*** END OF SELECTIONS ***

# The user also has the option to provide additional instructions:
*** BEGINNING OF ADDITIONAL INSTRUCTIONS ***
{additional_instructions_from\user}
*** END OF ADDITIONAL INSTRUCTIONS ***

# Here is the up to date content of the file that the user is working on:
*** BEGINNING OF CONTEXT OF FILE ***
{context_of_file}
*** END OF CONTEXT OF FILE ***

# Your job now is to produce code that implements the solution based on the options that the user has selected.
- This should take into account previous decisions that the user has made.
- It should also consider the context of the file that the user is working on.
- If the user has provided them, incorporate the additional instructions that the user has provided.

# Return the code that you have produced in the defined format, along with potential further text explanations
if needed.
"""
```

## B User Study Material

## B.1 Demographic Questionnaire



Demographic Questionnaire

**How old are you?**

| 20-24 | 25-29 | 30-34 | 35-39 | 40-44 | 45-49 | 50 and older |
|---|---|---|---|---|---|---|
| ○ | ○ | ○ | ○ | ○ | ○ | ○ |

**What is your current job role?**

Ihre Antwort eingeben

**How many years of programming experience do you have?**

| 0-1 | 2-3 | 4-5 | 6-7 | 8-10 | 11-15 | 16 or more |
|---|---|---|---|---|---|---|
| ○ | ○ | ○ | ○ | ○ | ○ | ○ |

**How many hours do you typically spend programming in a week?**

| 0-4 | 5-9 | 10-14 | 15-19 | 20-30 | 31 or more |
|---|---|---|---|---|---|
| ○ | ○ | ○ | ○ | ○ | ○ |

**How much experience do you have using Python?**

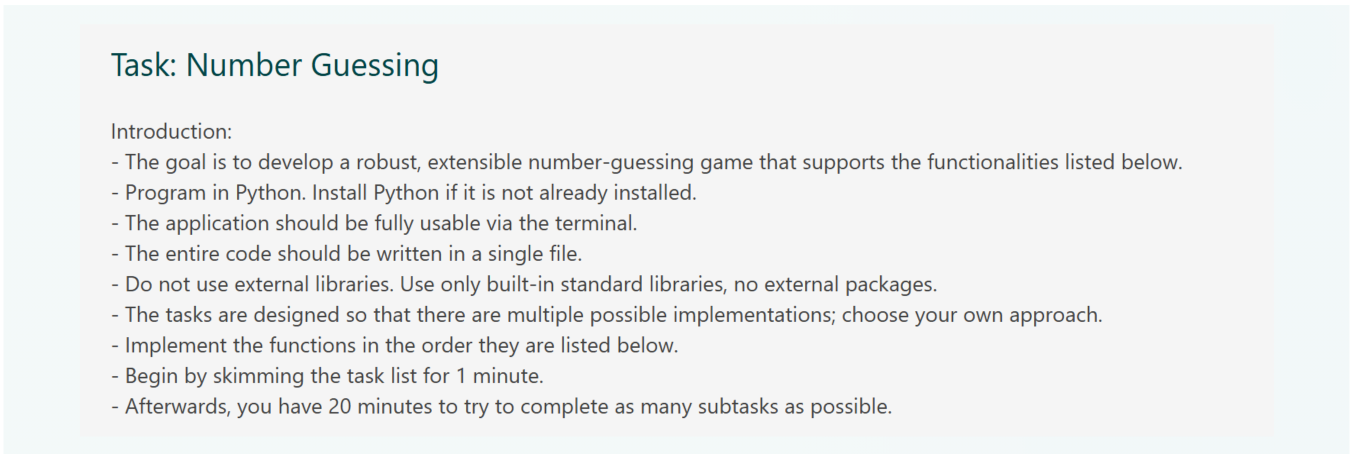
| 1 = Never Used | 2 | 3 | 4 | 5 | 6 | 7 = Constant Use |
|---|---|---|---|---|---|---|
| ○ | ○ | ○ | ○ | ○ | ○ | ○ |

**Which LLM-based tools (e.g., ChatGPT for research, etc.) do you use when programming?**

Ihre Antwort eingeben

**Figure 10: User-Study Material - Demographic Questionnaire.**

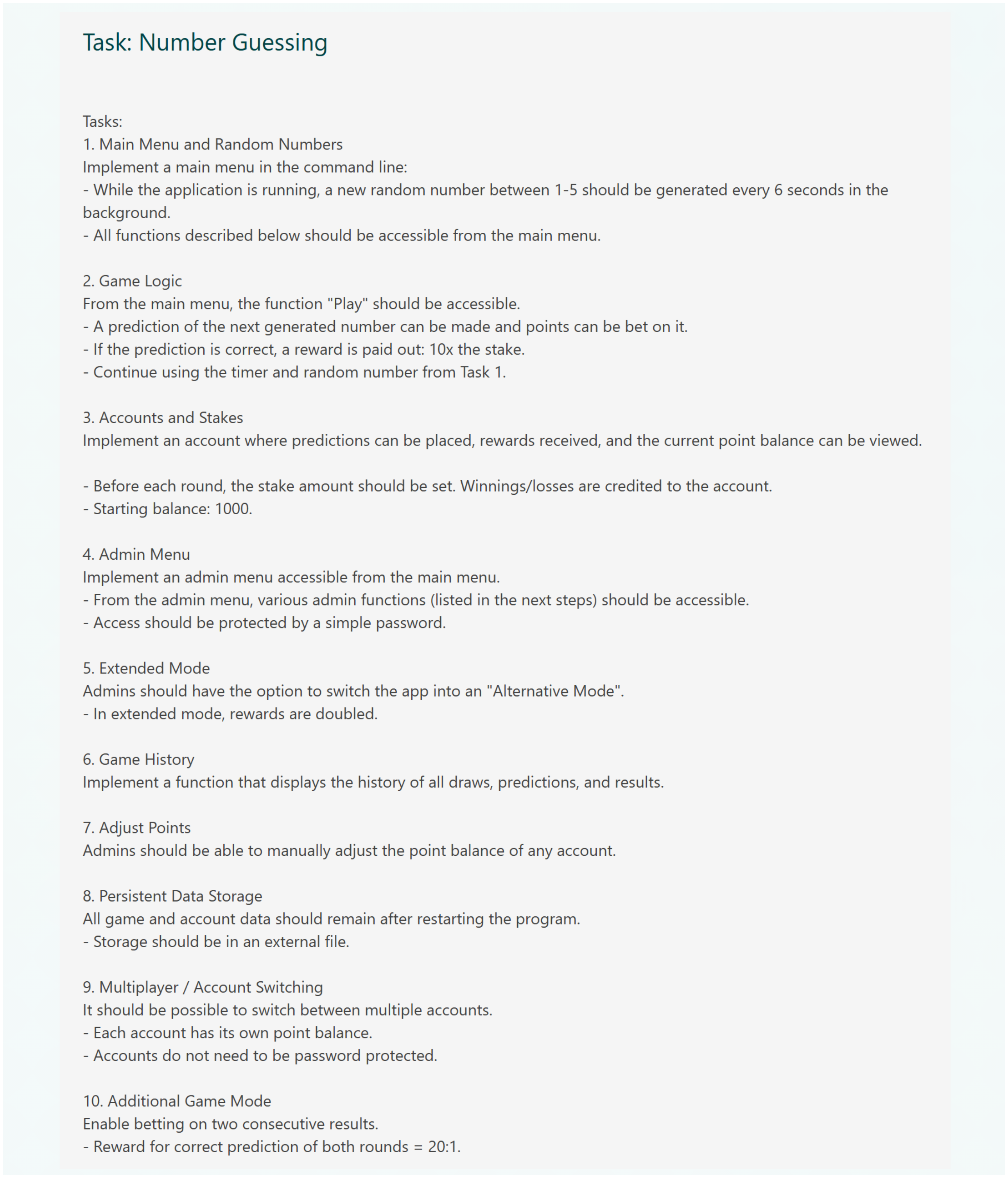
## B.2 Task Set 1



Task: Number Guessing

Introduction:
- The goal is to develop a robust, extensible number-guessing game that supports the functionalities listed below.
- Program in Python. Install Python if it is not already installed.
- The application should be fully usable via the terminal.
- The entire code should be written in a single file.
- Do not use external libraries. Use only built-in standard libraries, no external packages.
- The tasks are designed so that there are multiple possible implementations; choose your own approach.
- Implement the functions in the order they are listed below.
- Begin by skimming the task list for 1 minute.
- Afterwards, you have 20 minutes to try to complete as many subtasks as possible.

**Figure 11: User-Study Material - Taskset 1 Introduction.**


Task: Number Guessing

Tasks:
1. Main Menu and Random Numbers
Implement a main menu in the command line:
- While the application is running, a new random number between 1-5 should be generated every 6 seconds in the background.
- All functions described below should be accessible from the main menu.

2. Game Logic
From the main menu, the function "Play" should be accessible.
- A prediction of the next generated number can be made and points can be bet on it.
- If the prediction is correct, a reward is paid out: 10x the stake.
- Continue using the timer and random number from Task 1.

3. Accounts and Stakes
Implement an account where predictions can be placed, rewards received, and the current point balance can be viewed.

- Before each round, the stake amount should be set. Winnings/losses are credited to the account.
- Starting balance: 1000.

4. Admin Menu
Implement an admin menu accessible from the main menu.
- From the admin menu, various admin functions (listed in the next steps) should be accessible.
- Access should be protected by a simple password.

5. Extended Mode
Admins should have the option to switch the app into an "Alternative Mode".
- In extended mode, rewards are doubled.

6. Game History
Implement a function that displays the history of all draws, predictions, and results.

7. Adjust Points
Admins should be able to manually adjust the point balance of any account.

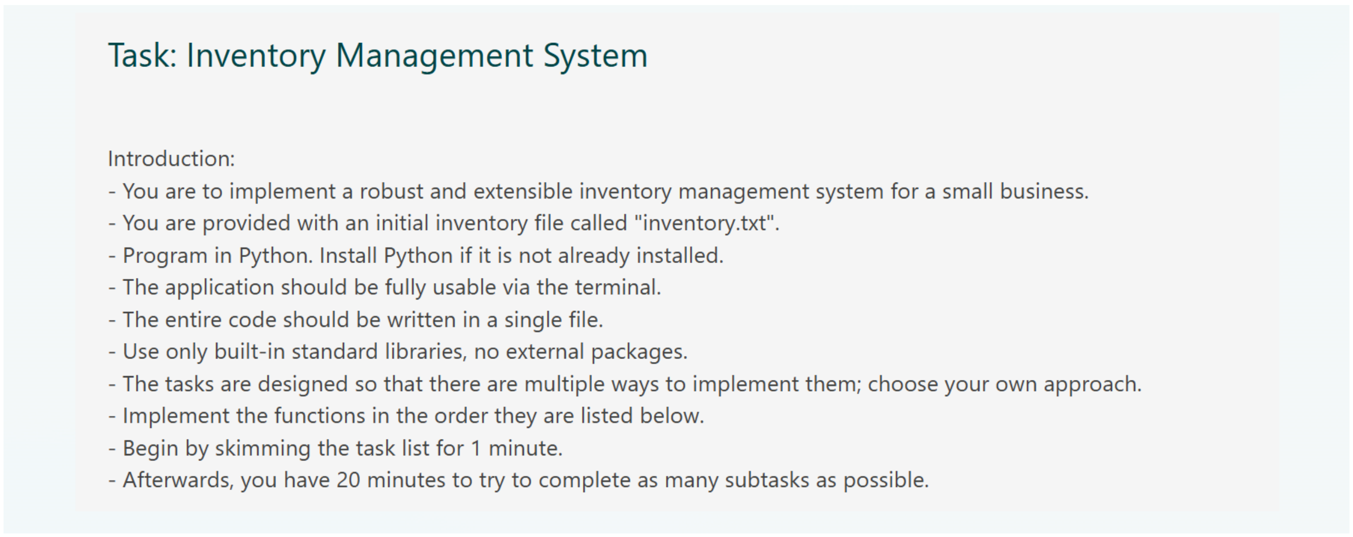
8. Persistent Data Storage
All game and account data should remain after restarting the program.
- Storage should be in an external file.

9. Multiplayer / Account Switching
It should be possible to switch between multiple accounts.
- Each account has its own point balance.
- Accounts do not need to be password protected.

10. Additional Game Mode
Enable betting on two consecutive results.
- Reward for correct prediction of both rounds = 20:1.

**Figure 12: User-Study Material - Taskset 1 Tasks.**

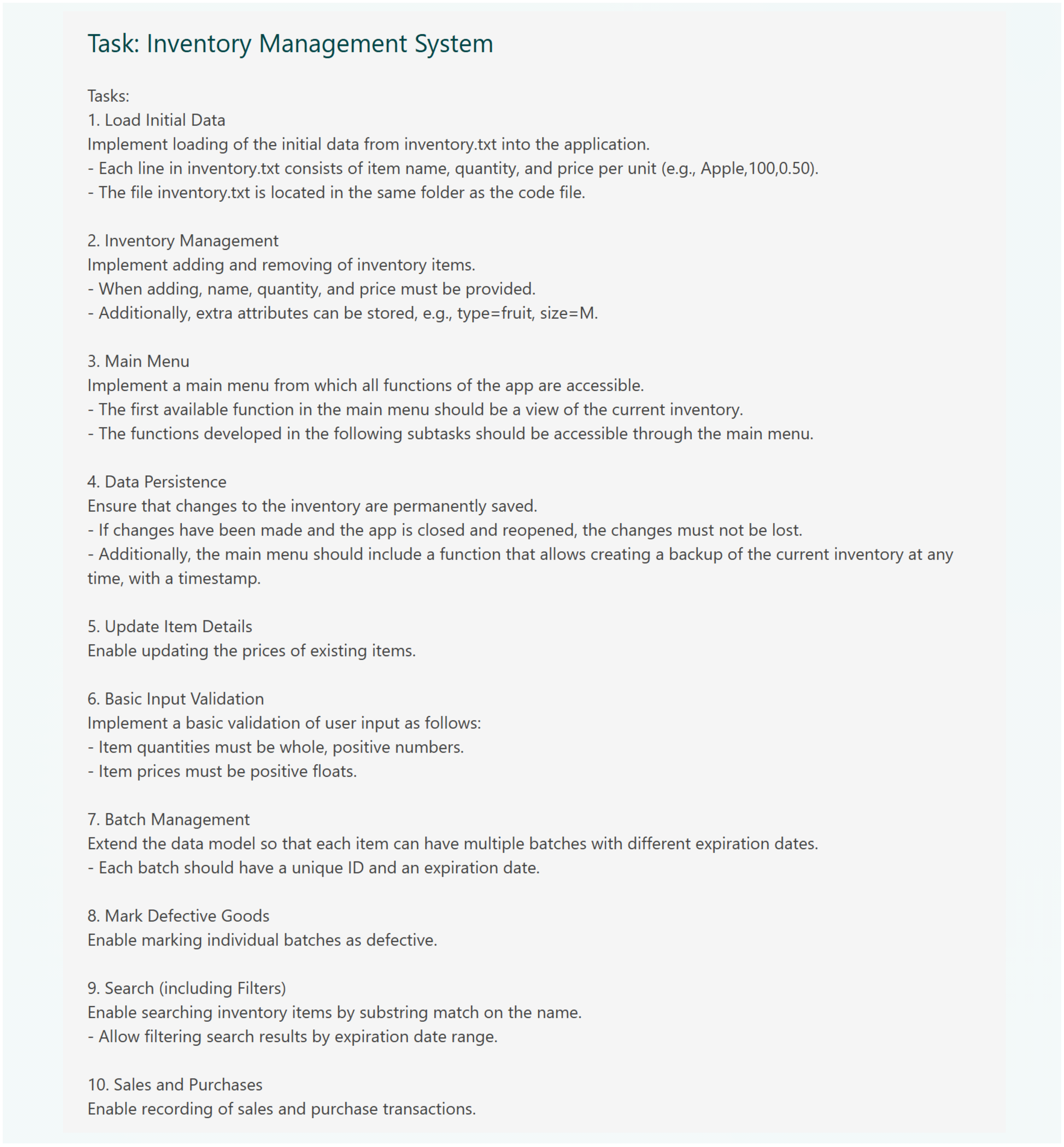
## B.3 Task Set 2



Task: Inventory Management System

Introduction:
- You are to implement a robust and extensible inventory management system for a small business.
- You are provided with an initial inventory file called "inventory.txt".
- Program in Python. Install Python if it is not already installed.
- The application should be fully usable via the terminal.
- The entire code should be written in a single file.
- Use only built-in standard libraries, no external packages.
- The tasks are designed so that there are multiple ways to implement them; choose your own approach.
- Implement the functions in the order they are listed below.
- Begin by skimming the task list for 1 minute.
- Afterwards, you have 20 minutes to try to complete as many subtasks as possible.

**Figure 13: User-Study Material - Taskset 2 Introduction.**



## Task: Inventory Management System

Tasks:
1. Load Initial Data
Implement loading of the initial data from inventory.txt into the application.
- Each line in inventory.txt consists of item name, quantity, and price per unit (e.g., Apple,100,0.50).
- The file inventory.txt is located in the same folder as the code file.

2. Inventory Management
Implement adding and removing of inventory items.
- When adding, name, quantity, and price must be provided.
- Additionally, extra attributes can be stored, e.g., type=fruit, size=M.

3. Main Menu
Implement a main menu from which all functions of the app are accessible.
- The first available function in the main menu should be a view of the current inventory.
- The functions developed in the following subtasks should be accessible through the main menu.

4. Data Persistence
Ensure that changes to the inventory are permanently saved.
- If changes have been made and the app is closed and reopened, the changes must not be lost.
- Additionally, the main menu should include a function that allows creating a backup of the current inventory at any time, with a timestamp.

5. Update Item Details
Enable updating the prices of existing items.

6. Basic Input Validation
Implement a basic validation of user input as follows:
- Item quantities must be whole, positive numbers.
- Item prices must be positive floats.

7. Batch Management
Extend the data model so that each item can have multiple batches with different expiration dates.
- Each batch should have a unique ID and an expiration date.

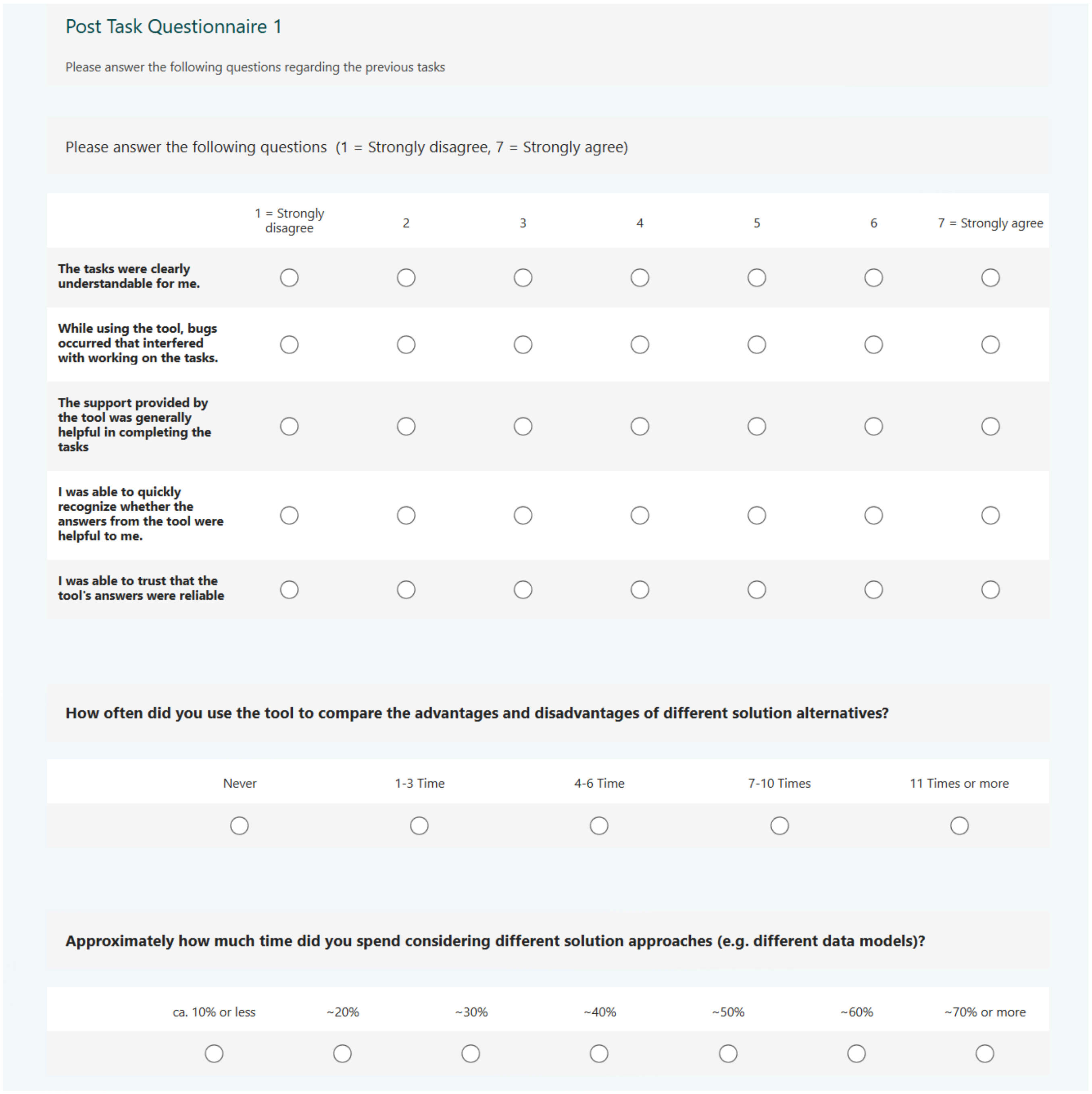
8. Mark Defective Goods
Enable marking individual batches as defective.

9. Search (including Filters)
Enable searching inventory items by substring match on the name.
- Allow filtering search results by expiration date range.

10. Sales and Purchases
Enable recording of sales and purchase transactions.

**Figure 14: User-Study Material - Taskset 2 Tasks.**

## B.4 Post-Task Questionnaires



Post Task Questionnaire 1

Please answer the following questions regarding the previous tasks

Please answer the following questions (1 = Strongly disagree, 7 = Strongly agree)

| | 1 = Strongly disagree | 2 | 3 | 4 | 5 | 6 | 7 = Strongly agree |
|---|---|---|---|---|---|---|---|
| **The tasks were clearly understandable for me.** | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| **While using the tool, bugs occurred that interfered with working on the tasks.** | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| **The support provided by the tool was generally helpful in completing the tasks** | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| **I was able to quickly recognize whether the answers from the tool were helpful to me.** | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| **I was able to trust that the tool's answers were reliable** | ○ | ○ | ○ | ○ | ○ | ○ | ○ |

**How often did you use the tool to compare the advantages and disadvantages of different solution alternatives?**

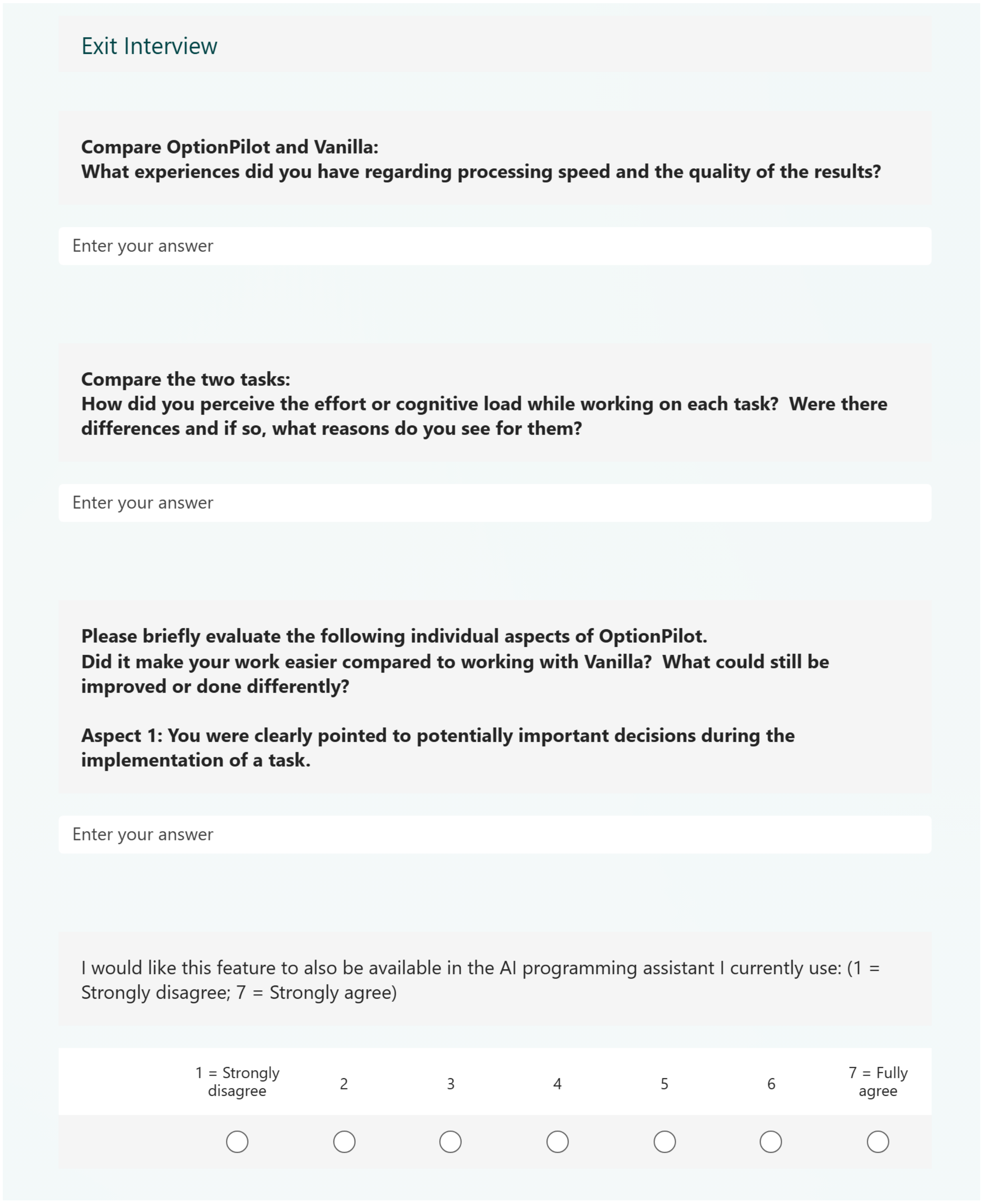
| Never | 1-3 Time | 4-6 Time | 7-10 Times | 11 Times or more |
|---|---|---|---|---|
| ○ | ○ | ○ | ○ | ○ |

**Approximately how much time did you spend considering different solution approaches (e.g. different data models)?**

| ca. 10% or less | ~20% | ~30% | ~40% | ~50% | ~60% | ~70% or more |
|---|---|---|---|---|---|---|
| ○ | ○ | ○ | ○ | ○ | ○ | ○ |

**Figure 15: User-Study Material - Post-Task Questionnaire.**

## B.5 Exit Interview Questionnaire



Exit Interview

**Compare OptionPilot and Vanilla:**
**What experiences did you have regarding processing speed and the quality of the results?**

Enter your answer

**Compare the two tasks:**
**How did you perceive the effort or cognitive load while working on each task? Were there differences and if so, what reasons do you see for them?**

Enter your answer

**Please briefly evaluate the following individual aspects of OptionPilot.**
**Did it make your work easier compared to working with Vanilla? What could still be improved or done differently?**

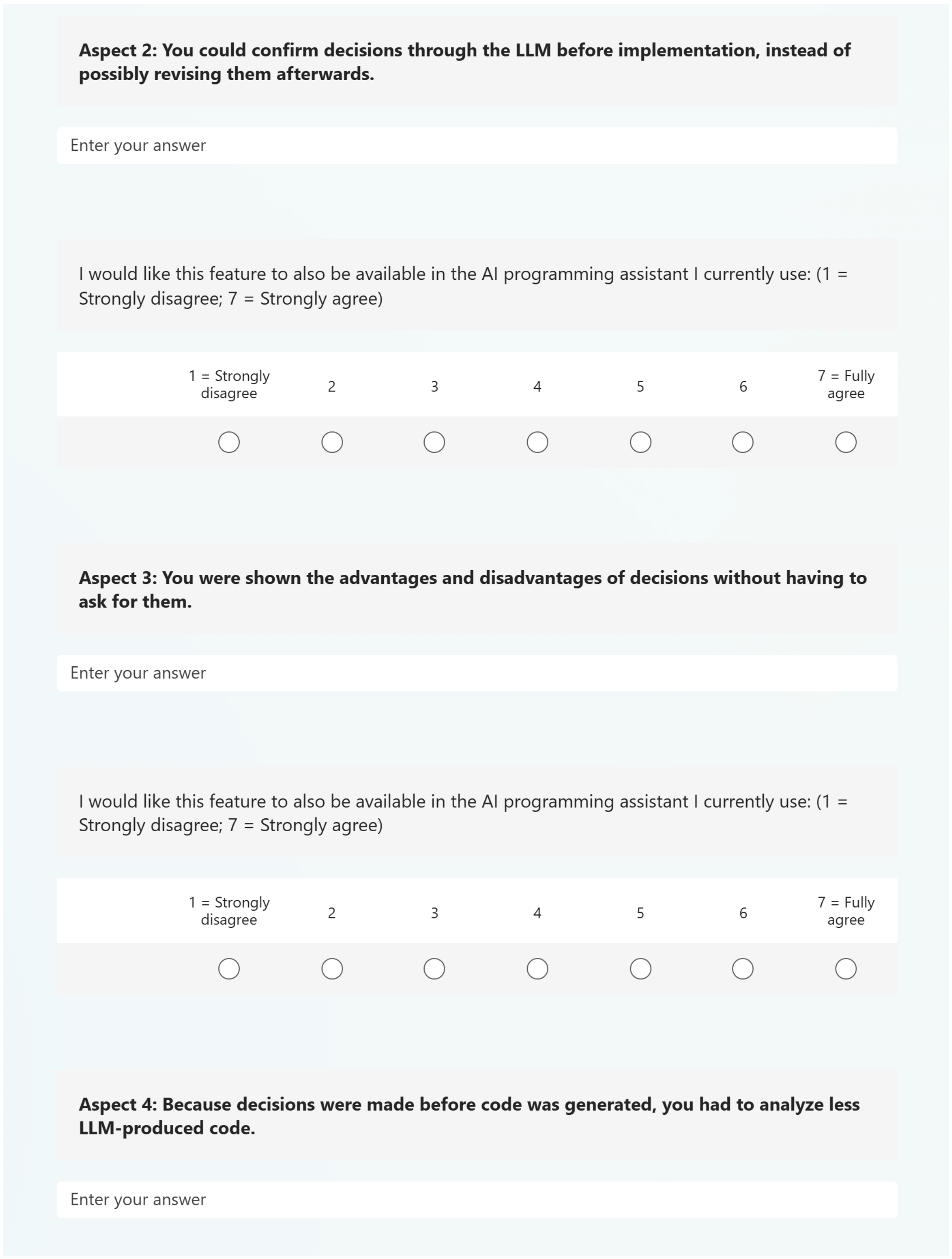
**Aspect 1: You were clearly pointed to potentially important decisions during the implementation of a task.**

Enter your answer

I would like this feature to also be available in the AI programming assistant I currently use: (1 = Strongly disagree; 7 = Strongly agree)

| 1 = Strongly disagree | 2 | 3 | 4 | 5 | 6 | 7 = Fully agree |
|---|---|---|---|---|---|---|
| ○ | ○ | ○ | ○ | ○ | ○ | ○ |

Figure 16: User-Study Material - Exit Interview (1/3).


**Aspect 2: You could confirm decisions through the LLM before implementation, instead of possibly revising them afterwards.**

Enter your answer

I would like this feature to also be available in the AI programming assistant I currently use: (1 = Strongly disagree; 7 = Strongly agree)

| 1 = Strongly disagree | 2 | 3 | 4 | 5 | 6 | 7 = Fully agree |
|---|---|---|---|---|---|---|
| ○ | ○ | ○ | ○ | ○ | ○ | ○ |

**Aspect 3: You were shown the advantages and disadvantages of decisions without having to ask for them.**

Enter your answer

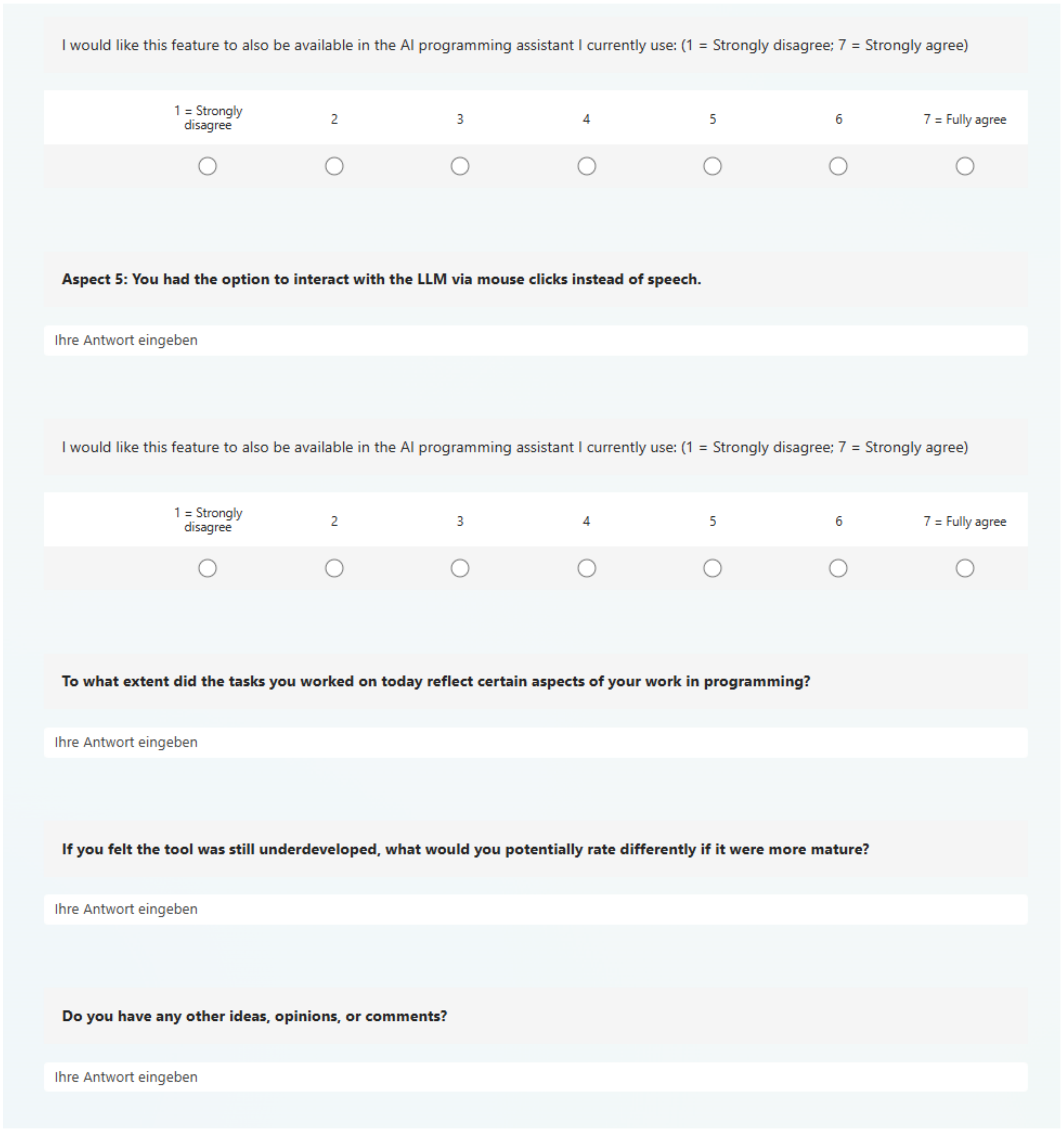
I would like this feature to also be available in the AI programming assistant I currently use: (1 = Strongly disagree; 7 = Strongly agree)

| 1 = Strongly disagree | 2 | 3 | 4 | 5 | 6 | 7 = Fully agree |
|---|---|---|---|---|---|---|
| ○ | ○ | ○ | ○ | ○ | ○ | ○ |

**Aspect 4: Because decisions were made before code was generated, you had to analyze less LLM-produced code.**

Enter your answer

Figure 17: User-Study Material - Exit Interview (2/3).


I would like this feature to also be available in the AI programming assistant I currently use: (1 = Strongly disagree; 7 = Strongly agree)

| 1 = Strongly disagree | 2 | 3 | 4 | 5 | 6 | 7 = Fully agree |
|---|---|---|---|---|---|---|
| ○ | ○ | ○ | ○ | ○ | ○ | ○ |

**Aspect 5: You had the option to interact with the LLM via mouse clicks instead of speech.**

Ihre Antwort eingeben

I would like this feature to also be available in the AI programming assistant I currently use: (1 = Strongly disagree; 7 = Strongly agree)

| 1 = Strongly disagree | 2 | 3 | 4 | 5 | 6 | 7 = Fully agree |
|---|---|---|---|---|---|---|
| ○ | ○ | ○ | ○ | ○ | ○ | ○ |

**To what extent did the tasks you worked on today reflect certain aspects of your work in programming?**

Ihre Antwort eingeben

**If you felt the tool was still underdeveloped, what would you potentially rate differently if it were more mature?**

Ihre Antwort eingeben

**Do you have any other ideas, opinions, or comments?**

Ihre Antwort eingeben

Figure 18: User-Study Material - Exit Interview (3/3).

## C Post-Task Questionnaire Individual Aspects Results

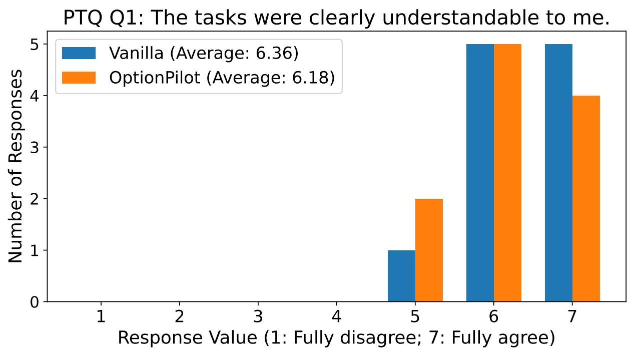


**Figure 19: Results of Post-Task Questionnaire Question 1.**

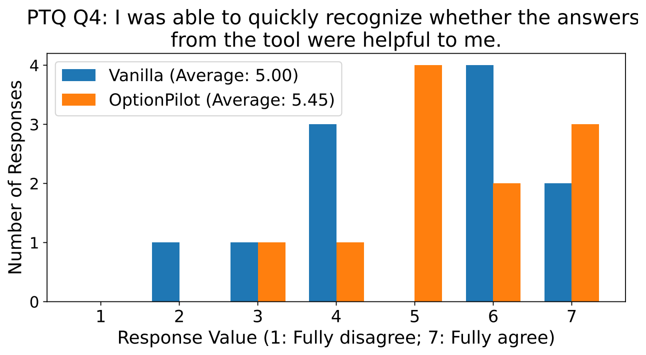


**Figure 22: Results of Post-Task Questionnaire Question 4.**

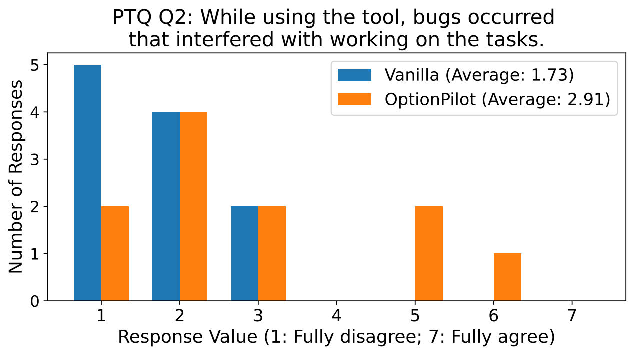


**Figure 20: Results of Post-Task Questionnaire Question 2.**

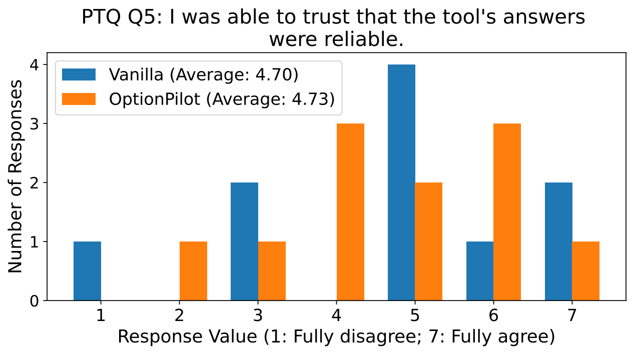


**Figure 23: Results of Post-Task Questionnaire Question 5.**

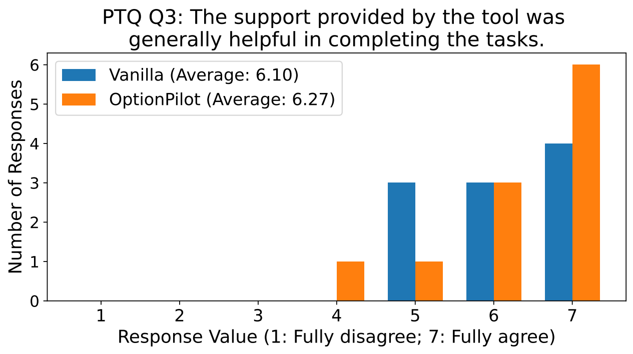


**Figure 21: Results of Post-Task Questionnaire Question 3.**

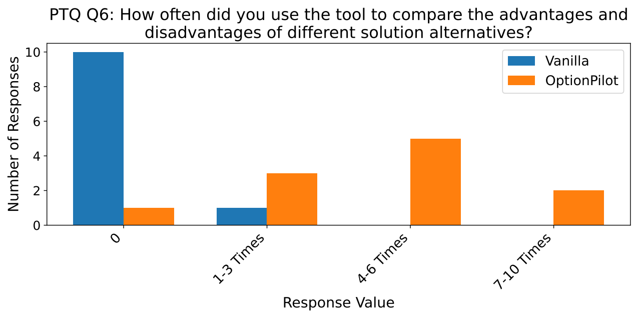


**Figure 24: Results of Post-Task Questionnaire Question 6.**

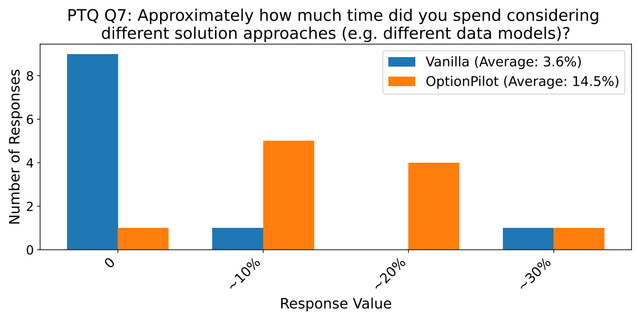


**Figure 25: Results of Post-Task Questionnaire Question 7.**