CoLadder: Supporting Programmers with Hierarchical Code Generation in Multi-Level Abstraction

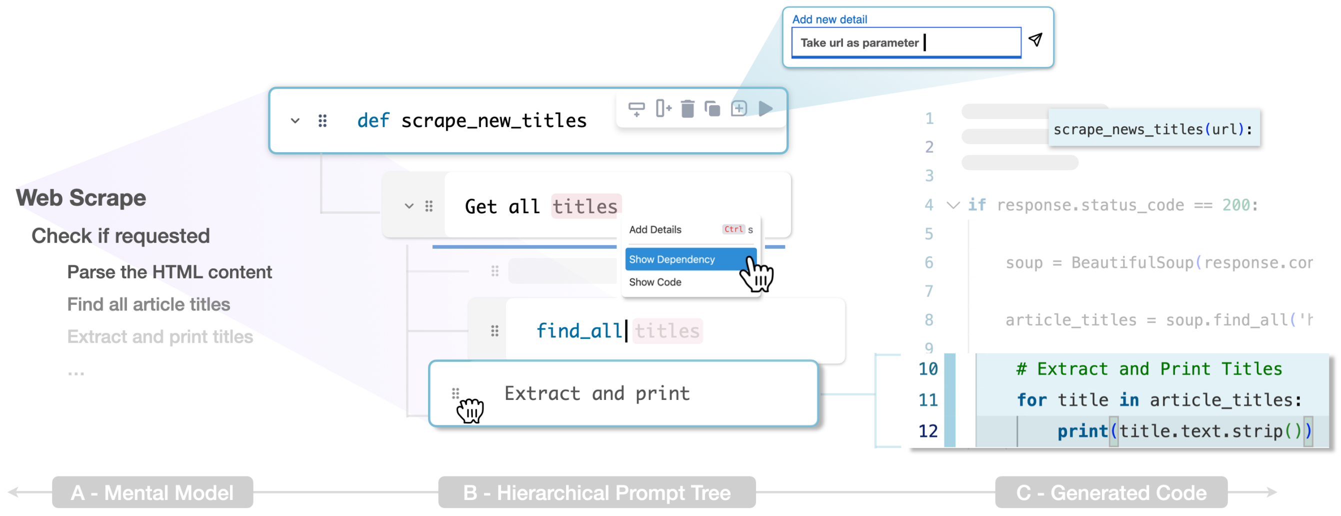
Ryan Yen
University of Waterloo
Waterloo, Ontario, Canada
r4yen@uwaterloo.ca

Jiawen Zhu
University of Waterloo
Waterloo, Ontario, Canada
jiawen.zhu@uwaterloo.ca

Sangho Suh
University of California San Diego
La Jolla, California, USA
sanghosuh@ucsd.edu

Haijun Xia
University of California San Diego
La Jolla, California, USA
haijunxia@ucsd.edu

Jian Zhao
University of Waterloo
Waterloo, Ontario, Canada
jianzhao@uwaterloo.ca



```
scrape_news_titles(url):


if response.status_code == 200:

    soup = BeautifulSoup(response.cor

    article_titles = soup.find_all('h

    # Extract and Print Titles
    for title in article_titles:
        print(title.text.strip())
```

Figure 1: *CoLadder* enables programmers to flexibly decompose tasks, aligning with their mental models for solving programming tasks using LLM-driven code assistants (A). The system provides a tree-based editor that allows programmers to hierarchically express their intent through smaller, modular-based prompt blocks (B). This hierarchical prompt tree structure is then used to generate code, with each prompt block corresponding to a code segment (C). Programmers can directly manipulate the code based on the prompts using a series of block-based operations.

ABSTRACT

Programmers increasingly rely on Large Language Models (LLMs) for code generation. However, they now have to deal with issues like having to constantly switch between generating and verifying code, caused by misalignment between programmers' prompts and the generated code. Unfortunately, current LLM-driven code assistants provide insufficient support during the prompt authoring process to help programmers tackle these challenges emerging from the new workflow. To address these challenges, we employed an iterative design process to understand programmers' strategies when programming with LLMs. Based on our findings, we developed *CoLadder*, a system that assists programmers by enabling hierarchical task decomposition, incremental code generation, and verification of results during prompt authoring. A user study with 12 experienced programmers showed that *CoLadder* is effective in helping programmers externalize their mental models flexibly, improving their ability to navigate and edit code across various abstraction levels, from initial intent to final code implementation.

KEYWORDS

LLM, code generation, system, prompt engineering

1 INTRODUCTION

Recent advances in large language models (LLMs) have led to significant progress in AI-driven code assistants [24, 37, 58] and brought changes to programmers workflows [48, 69, 91]. These LLM-driven code assistants have extended their functionality beyond code completion to generate high-quality code suggestions in response to natural language (NL) prompts [19, 23, 24, 36, 72]. Programmers can now focus on translating high-level ideas into NL prompts without delving into the intricacies of low-level code implementation [13, 18, 34, 79]. While this distinct feature can potentially make the programming process efficient, recent research into programmers' practices and experiences with LLM-driven code assistants has revealed that programmers are encumbered by the constant need to verify the substantial volume of code generated by LLMs [46, 69, 84, 89, 91, 100].

This verification process is essential for programmers to ensure the AI-generated code is correct and integrable with the rest of the code [24]; however, it requires substantial effort and time [46, 84, 89, 91, 100]. This process becomes even more demanding when misalignment between programmers' intended outcomes and generated code occurs. Since LLMs can generate plausible results from almost any NL prompt, programmers often struggle with the misalignment issue when trying to determine how to adapt their prompts effectively in response to unexpected output [34, 63, 86, 97]. This process further imposes an additional cognitive load as programmers continuously switch between modifying NL prompts and verifying the generated code.

Previous research has designed several techniques to streamline this iterative process of code verification and prompt modification [2, 20, 26, 34]. One major approach is to decompose complex problems into sub-problems of a pre-defined abstraction level [20, 46, 80, 96, 97]. Specifically, the programming task is divided into smaller, more manageable sub-tasks, each at a set level of complexity or detail. This division helps programmers effectively identify necessary changes and thereby reduces verification efforts when the result is misaligned with the programmers' intent [14, 25, 51, 97, 104].

While these tools successfully assist programmers in understanding how to use LLMs to translate their intent into code, they provide programmers with limited freedom during task decomposition. Therefore, the effectiveness of these types of systems is limited for two key reasons. Firstly, the pre-defined level of division may not accurately reflect the task structure programmers have in mind, which typically also aligns with the structure of the generated code [1, 43, 81]. Secondly, since each programmer has a unique mental model for solving programming tasks, it becomes challenging for these techniques to adapt and determine the most suitable level of task decomposition for each individual [20, 34, 62, 97].

The goal of this research is thus to explore design opportunities for empowering programmers with the flexibility to decompose tasks based on their individual mental models and effectively translate their intent into generated code. We conducted a formative study with six experienced programmers who regularly use LLM-driven code assistants. Findings suggest that programmers rely on hierarchically structured prompts to externalize their mental models of the programming task structure. However, programmers are hindered due to the lack of informative prompt organization and the inability to directly control and manipulate generated code segments in existing LLM-driven code assistants. The constant context switching between writing prompts and verifying code further prevents them from concentrating on decomposing their tasks.

Based on our findings, we propose *CoLadder*, a system that assists programmers with externalizing hierarchical prompt structures to generate code that aligns with their intent. The development of the system followed an iterative design process, soliciting feedback for feature improvement through a cognitive walkthrough involving six experienced programmers. *CoLadder* introduces the concept of *hierarchical generation* by helping programmers to decompose tasks into smaller subtasks (Fig. 1 A), forming a task structure that reflects programmers' mental models (Fig. 1 B), and generating the corresponding code structure (Fig. 1 C). *CoLadder* considering each prompt as a modular *block* directly linked to corresponding code segments. This allows precise modification of specific code segments by manipulating the corresponding prompt blocks within the context of the hierarchical task structure. Furthermore, *CoLadder* enables verification during the prompt authoring process by providing scaffolding through multiple levels of abstraction, including tasks, sub-tasks, NL prompts, pseudocode, and generated code.

We further conducted a user study with 12 experienced programmers who frequently use LLM-driven code assistants to evaluate the usefulness of *CoLadder*. The results validate that *CoLadder* helps programmers externalize and develop their mental models throughout the prompt authoring and code generation process. The direct manipulation of prompt blocks at multiple levels of abstraction to modify the corresponding code provides programmers with control over the translation of their intent into code. With the supported scaffolding, *CoLadder* prevents programmers from disruptive cognitive switching between prompt authoring and code verification by enabling results verification from prompts. These findings imply the concept of *hierarchical generation* as a design consideration for future developments in interactive LLM-code assistants.

In summary, our contribution is threefold:

- A formative study that identified strategies, and programmers' needs for externalizing mental models and scaffolding intents to generate code.
- An interactive system, *CoLadder*, employs a hierarchical structure and block-based design to provide programmers with code generation capabilities across multiple levels of abstraction, aligning with both the task and code structure.
- A user study demonstrating improved usability by enhancing controllability with *hierarchical generation* and enabling result verification during the prompt authoring process.

2 BACKGROUND AND RELATED WORK

CoLadder aims to improve the collaborative process between programmers and LLMs by addressing key interaction challenges of code generation. Taking into account the verification hurdles, abstraction complexities, and the importance of a programmer's mental model, we delve into innovative solutions ranging from the intricacies of LLM-driven code generation to insights from visual programming tools.

2.1 Programmer-LLM Interaction

While the interaction between programmers and AI has been explored in a variety of ways, our focus is on deep learning model-driven code generation tools. These models possess the ability to produce code based on plain language explanations or incomplete code snippets after being trained on a large amount of existing codebase [7, 42, 49, 56, 88, 102]. Recent advances in LLMs, exemplified by GPT-4, LLaMa, and PaLM 2, mark a significant breakthrough in code generation compared to preceding deep learning models. Previous research has conducted several user-centered studies to understand how programmers interact with LLMs-based code assistants and their perceptions of these tools [10, 29, 59, 69, 71, 77, 84, 89, 91, 100]. Studies have shown that the accuracy of code assistants has improved significantly with the availability of state-of-the-art LLMs [24, 76], thereby increasing perceived productivity [58, 105], especially in tasks that require writing simple code snippets repeatedly [10, 84].

Challenges of Verification. However, programmers now need to dedicate considerable time to verifying AI-generated code suggestions [10, 69]. Liang et al. [59] conducted a survey study highlighting that current code generation tools face several usability challenges, including difficulties in comprehending the relationship between the programmer's prompt and the resulting generated code, as well as struggles in guiding these models to produce desired outputs. Barke et al. [10] describe various behaviours that programmers exhibit when they encounter verification challenges. These behaviours encompass accepting suggestions without verification, postponing the verification process, or using code as the prompt instead of natural language. Vaithilingam et al. [91] report that programmers using LLM-based code assistants often fail to complete their programming tasks. This problem occurs due to an underestimation of the time required to debug the generated code. Excessive verification needs can lead to several issues. [82, 92, 95] For one, programmers are often intimidated by the seemingly overwhelming effort required for code validation, and bypass the verification step. This causes problems like over-reliance on generated suggestions [10, 24, 100], and loss of control over their own programs [91], which then introduces challenges during code modification [3, 20]. For another, programmers are also taxed with the extra cognitive load of switching between programming and debugging tasks [11, 33, 66, 91].

Abstraction Matching Issue. Sarkar et al. [84] observed that programmers often engage in iterative verification and prompt refinement to understand how well LLM-driven code assistants can interpret their prompts and generate the desired code, a process referred to as *abstraction matching*. Programmers are required to grasp the models' capabilities and limitations to understand the necessary naturalistic utterances to generate code that aligns with their intents. This issue is rooted in the notion of the *gulf of execution* [47], i.e. challenges arising when translating user instructions into executable computer tasks. The problem of matching abstractions became more noticeable with LLMs due to their capability to generate code at different *levels of abstraction*, ranging from high-level, conceptual descriptions to low-level, detailed pseudo-code-like statements, which cover innumerable combinations of natural language expressions [62, 97].

Our study centers around the challenges related to verifying generated code, particularly the issue of abstraction matching when translating user intent into the desired code. Additionally, *CoLadder* assists programmers in expressing their intent using a hierarchical structure and composing prompts at multiple levels of abstraction to control the generation of corresponding code structures.

2.2 Improving LLM-based Code Generation

There are several technical strategies, such as fine-tuning and using pre-trained models, that can improve LLM-based code generation by updating part of the model's weights and adapting it to specific coding tasks and domains. Previous research has largely utilized alternative technical strategies, *prompt engineering* [12, 61, 63, 64, 86], to thoughtfully craft prompts for generating desired code, offering a solution to tackle the abstraction matching issue.

In comparison to the technical strategies, several design solutions and systems have been proposed to facilitate interaction and collaboration with LLM-driven code assistants. These strategies encompass various techniques, such as introducing new programming languages [13, 46], automating prompt rewording, providing suggestions for simpler tasks, employing programming by demonstration techniques [28, 60], chaining a series of prompts [96], and supporting task breakdowns [80]. By breaking down complex problems into simpler sub-tasks, the gap in abstraction is reduced, enabling successful guidance of the model to generate code that matches the programmer's intents [14]. However, determining the 'correct' level of abstraction remains a challenge, as overly detailed prompt decomposition can result in the programming process with LLMs resembling the use of a "highly inefficient programming language" [84]. Hence, prior research into natural language interfaces suggests the benefit of managing expectations and gradually revealing the capabilities of the system through user interaction and intervention [62, 65, 82, 92].

Noticing this issue, prior research has provided several alternative approaches in addition to the idea of task breakdowns. Liu et al. introduced an alternative strategy known as *Grounded Abstraction Matching* in response to the observation that existing systems often lack guidance for programmers on how to decompose tasks [62]. Their approach offers an example decomposition of the generated code that users can modify and re-submit to the LLM as a set of instructions. This strategy is particularly helpful for programmers who have not yet formed a clear intent, reducing the occurrence of abstraction matching issues. AI Chains is an approach that enhances programmer control and feedback by breaking down problems into smaller sub-tasks [97]. Each sub-task corresponds to a specific step with an NL prompt, and results from previous steps inform the prompts for subsequent ones. This chaining method increases the chances of success for each sub-task when running the same model on multiple tasks [96, 97].

While the previously mentioned approaches that rely on task breakdowns assist programmers in bridging their intent to code, they do not emphasize scaffolding programmers' mental models for solving programming tasks. Additionally, they primarily focus on local prompt-code correspondence without examining the overall structure matching, especially from task structure to code structure. In contrast, *CoLadder* builds upon the concept of task decomposition and offers increased flexibility and controllability for programmers to not only craft effective prompts but also gain a deeper understanding of how their programming tasks can be logically structured.

2.3 Programmers' Mental Models of Programs

In the programming context, mental models encompass programmers' understanding and interpretation of the code, underlying programming tasks, and the overall structure of the programs they are working on [6, 30, 101]. There are several theories that describe the formation of these mental models [31, 93]. While some theories suggest a bottom-up approach, starting with understanding code syntax to derive semantic meanings, others advocate for a top-down strategy that begins with an initial hypothesis of code functionality and then verifies it through syntax analysis [94]. Programmers must develop mental models of code at different levels of abstraction [8, 101], encompassing specific code statements as well as larger program structures. To support effective interaction and collaboration between programmers and LLMs in tackling programming tasks, it is crucial to provide scaffolding for these mental models [34, 62, 84].

Despite programmers dedicating a significant portion of their time to comprehending code, with this task occupying more than half of their coding time, the integration of tools designed to facilitate this process remains underutilized in their daily routines [15, 43, 50, 67, 98]. In our work, we prioritize assisting programmers in forming mental models to tackle programming tasks effectively and externalizing these models into prompts that generate code aligning with their intents. While LLM-driven code assistants can rapidly generate code, *CoLadder* empowers programmers to concentrate on planning their code and task structure [44].

2.4 Visual Programming Tools

Although *CoLadder* is not designed as a visual programming tool, it leverages design implications from previous literature on visual programming tools to scaffold programmers' intents, endowing them with control over the code-generating processes. Among all the visualizations, we focus on the flow-based design, which is commonly used in most current LLM-related tools, and the block-based design that our design references.

Flow-based designs were adopted in many LLM-driven text generation tools [26, 55, 87, 97], and extract and display high-level sub-tasks to underline the logical flow within a program or text. While this visualization approach simplifies comprehension, it obscures code-level details that programmers need [20, 39, 70]. This approach, although potentially beneficial for novice programmers with no-code experience, may not entirely satisfy the requirements of experienced developers. On the other hand, block-based programming tools stand out for offering granular control over the code, establishing a direct and low-level correspondence between each fragment of the code and its visualization, and facilitating precise representation and controllability of individual code elements [16, 21].

Recognizing the different benefits and drawbacks of these visual programming tools, *CoLadder* leverages a block-based design by representing each prompt as a modular block corresponding to a code segment. This approach allows programmers to flexibly construct a hierarchical prompt structure to generate aligned code structures, facilitating code generation and verification with controllability.

3 COLADDER: DESIGN PROCESS & GOALS

To design an interface that can assist programmers in decomposing tasks based on their individual mental models and translating their intent into generated code, we conducted several iterations of user-centred design to build *CoLadder*. The design process consisted of three stages: 1) *Understanding & Ideation*—including an interview study with experienced programmers to discover the strategies they employ to address the challenges of programming with LLM-driven code assistants; 2) *Prototype & Walkthrough*—the design and development of *CoLadder* informed by established design goals and a cognitive walkthrough for feedback and iterative design (Section 4); 3) *Deploy & Evaluate*—a user study involving 12 experienced programmers to evaluate how programmers interact with *CoLadder* and their perceived usefulness (Section 5). In this section, we describe the first stage of our design process and report the obtained strategies, user needs and design goals that guided the design and development of *CoLadder* (Table. 1).

3.1 Interview Study

We recruited six participants (5 males, 1 female; ages 25 − 27, $M = 25.8, SD = 0.98$) through convenience sampling, all of whom had more than five years of programming experience ($M = 7.88, SD = 4.34$ years) and regularly used the LLM-code generation tools ($M = 8, SD = 2.56$ times/week). Each study session lasted around 45 minutes and we compensated participants 15 CAD for their time. Before the study, we asked each participant to share at least three recent examples of their ChatGPT [74] usage for generating code, in order to nudge them to reflect on how they use LLM-code generation tools. During the study, we interviewed participants about 1) the challenges they encountered in externalizing mental models and translating intents into code, 2) the strategies they employed to tackle these challenges, and 3) their needs throughout the prompt authoring process. The interviews were audio-recorded and subsequently transcribed into text. The first and second authors conducted a thematic analysis [17] and summarized themes related to the strategies participants employed to address challenges they encountered and their corresponding user needs.

3.2 Interview Results: Strategies

Structure Tasks and Prompts Hierarchically. We observed that the development of the mental model for participants to solve programming tasks when prompting LLM-code assistants involves two key steps. First, programmers need to form clear intents for solving programming tasks, which is important to *"verify if the generated code is correct or not" - P3*; Second, they must explore how to effectively construct the prompt to translate those intents to generated code. However, participants encountered difficulties with the linear representation of prompts (e.g., a sentence of comment), hindering their ability to externalize their mental models and understand their own prompts after composing them (**C1**). For instance, P1 expressed, *"It [the prompt] becomes meaningless after a few iterations, as the prompt is not for humans but written for the LLM to understand me."* Every participant adopted a similar strategy to alleviate the cognitive load when forming mental models — by breaking down tasks into smaller subtasks (**S1**). Most (5 out of 6 participants, 5/6 henceforth) participants took things a step further by structuring their tasks hierarchically to externalize their mental models (**S2**). P4 noted, *"I will have a mind-map in my head to structure my prompt step by step"*; and four out of six participants used bullet points to structure the prompt, *"I will use indent when

Table 1: The summary of challenges and strategies reported in Section 3.2 and the resulting design goals (Sections 3.3) and features in *CoLadder* (Sections 4)

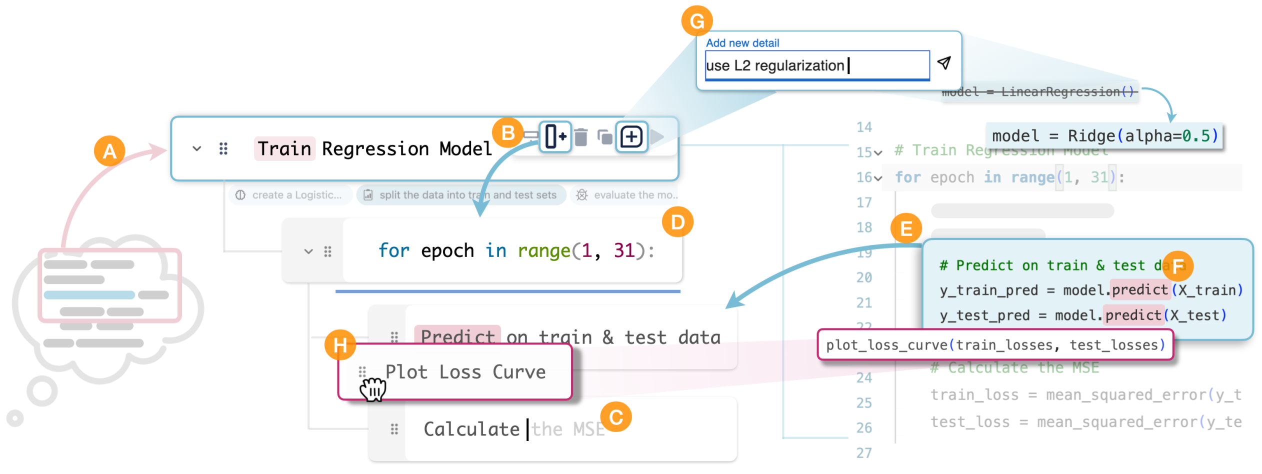
Challenges	**S**trategies	**D**esign **G**oals	Features
C1: Unstructured Prompt	**S1**: Task Decomposition	**DG1**: Hierarchical Prompt Structure	Prompt Tree Structure
	S2: Hierarchical Structure		
C2: Control Loss from Intents to Code	**S3**: Incremental Generation	**DG2**: Direct Manipulation of Prompt Blocks for Code Modification	Prompt Block
	S4: In-Situ Generation		
	S5: Rearrange Code Segments		Block-based Operations
.	**S6**: Replace Code Segments		
C3: Disruptive Context Switching	**S7**: Verify Results during Prompt Authoring	**DG3**: Enabling Code Verification during Prompt Authoring	List Steps
			Auto-Completion
			Recommendation
C4: Unclear Correspondence from Prompt to Code	**S8**: Add Code to Prompt	**DG4**: Assisting Intent-Code Navigation Across Multi-Levels	Corresponding Code Highlight
	S9: Verified by Comments		Semantic Highlight

writing the prompts; this hierarchy structure helps me think about the detailed steps." - P6

Generate and Edit Code Segments by Segments. While participants made efforts to structure their prompts to become more comprehensive, the LLMs exacerbated the difficulties of verification by generating an entire codebase based on the whole prompt. This issue resulted in a sense of control loss and *'fear'* over the code generation process (**C2**), where participants expressed the desire to *"generate the program bit by bit." - P2* Some (3/6) participants mentioned the difficulties in the long-term maintenance of the program, *"I want to be able to return to my code months later and still be able to debug it." - P6* As a result, participants generally preferred to accept the generated code line by line, similar to the use of auto-completion features (**S3**), rather than generating the entire code base at once. Participants frequently used an additional strategy (**S4**) to control the length of the generated code. This strategy involved inserting line breaks between prompts, allowing them to generate code selectively between specific prompts rather than generating lengthy code encompassing all prompts. However, implementing this strategy introduced a misalignment issue between the structured prompts and the code structure, which subsequently made it challenging for participants to determine *"where to insert newly generated code." - P2* Participants (4/6) thus reported an alternative strategy, where they generated self-contained code segments independently, and then merged them into the existing codebase (**S5**). P5 explained the reason, *"I will combine it [generated code] by myself as I didn't know what it would look like beforehand."* Another common strategy is to select a segment of existing code and replace it with the newly generated code (**S6**), enabling manipulation of targeted code segments without impacting other sections. Yet, P3 outlined this tedious process of preserving and combining both existing and newly generated code, *"sometimes if the generated code is partially accurate, I need to initially accept it, make a copy, then undo the changes, and finally paste the copied code below my original code."*

Verify Generated Code through Prompts. All participants expressed frustration with current code generation tools that incessantly suggest results, forcing them to switch between prompt authoring and code verifying continuously (**C3**). They noted that although this context switching may seem trivial, it significantly disrupts their overall programming flow. P6 mentioned, *"Sometimes I know it [the generated code] would not be correct, but I will still look into it,"* and P4 explained that their thought process is *"probably this time the generated code is correct, who knows?"* However, we observed that participants more frequently accepted auto-completed prompts, using them to quickly *"verify if the system had accurately captured my [their] intents." - P3* (**S7**) Some participants (P1, P2, P5), deliberately waited for the code assistants to auto-complete their prompts, which further *"reduced the time needed to verify the generated code." - P2*

Adding Cues to Navigate between Prompt-Code. Participants usually need to go through several iterations of the prompt, which makes it challenging to locate the phrases to modify as the prompt grows increasingly lengthy. All participants thus reported difficulties in navigating back and forth between prompt and generated code segments, finding it *"difficult to find which part [of the prompt] is causing the error that needs to be modified." - P1* (**C4**) Several programmers (4/6) incorporate code syntax into their prompts to facilitate code verification by making it easier to locate keywords (**S8**). For instance, P2 mentioned using pseudo-code-like prompts as a strategy *"to control the generation"* and reduce the cognitive load during the verification process. Some participants (2/6) also mentioned that they would rather rewrite the whole segment of code, rather than attempt to edit and debug it. *"After generating code, I need to spend a lot of time finding the code causing the error and also finding the prompt that resorted to this result, I would just start all over again." - P1* Consistent with previous research analyzing eye-tracking data [89], participants usually valued the generated comments (in natural language) and used them as anchor points to match segments of prompts to code (**S9**). P4 mentioned the reason that *"the generated comments are useful to check if the generated code matches my instructions step-by-step."*



```
Train Regression Model
    for epoch in range(1, 31):
        Predict on train & test data
        Plot Loss Curve
        Calculate the MSE
```

```
~~model = LinearRegression()~~
model = Ridge(alpha=0.5)
# Train Regression Model
for epoch in range(1, 31):
    # Predict on train & test data
    y_train_pred = model.predict(X_train)
    y_test_pred = model.predict(X_test)
    plot_loss_curve(train_losses, test_losses)
    # Calculate the MSE
    train_loss = mean_squared_error(y_t
    test_loss = mean_squared_error(y_te
```

Figure 2: An example workflow of using *CoLadder*. A user creates a high-level prompt (A) based on their mental model of the programming task. Subsequently, they add a sub-task indented underneath (B), incorporating code syntax within the prompt (D). The *List Steps* feature is employed to summarize the generated code into prompts (E). Following verification, the user modifies the prompt, accepting *auto-completed* suggestions (C). To ensure code accuracy, the user employs the *semantic highlight* feature (F). When additional details are needed, they use the *supplement* feature to add detail to the prompt (G). Finally, the user rearranges the prompt structure using the *Drag and Drop* feature (H).

3.3 Design Guidelines

To support the reported strategies (**S1-S9**) programmers leveraged to overcome challenges they encountered (**C1-C4**), we formulated four design guidelines (**DGs**). The overarching goal of the design is to provide *hierarchical generation* for programmers to translate abstract intents into concrete code, aligning the resulting structure with their mental models.

DG1: Offering Hierarchical Prompt Structure. The lack of structured prompts hinders programmers from externalizing the mental models for solving programming tasks (**C1**). The system should support prompt decomposition (**S1**) with a hierarchical representation of the prompt structure that reflects the task structure programmers possess in mind (**S2**). Previous research adopted prompt decomposition by pre-defining the permitted abstraction levels [13, 46, 80, 97], which may not necessarily reflect programmers' own intuition of how they would have decomposed the task [22, 99]. Based on our formative study, participants have a much more free-form prompt structure depending on the nature and complexity of the task. Hence, the system should allow participants to define the levels of abstraction reflecting their own mental models.

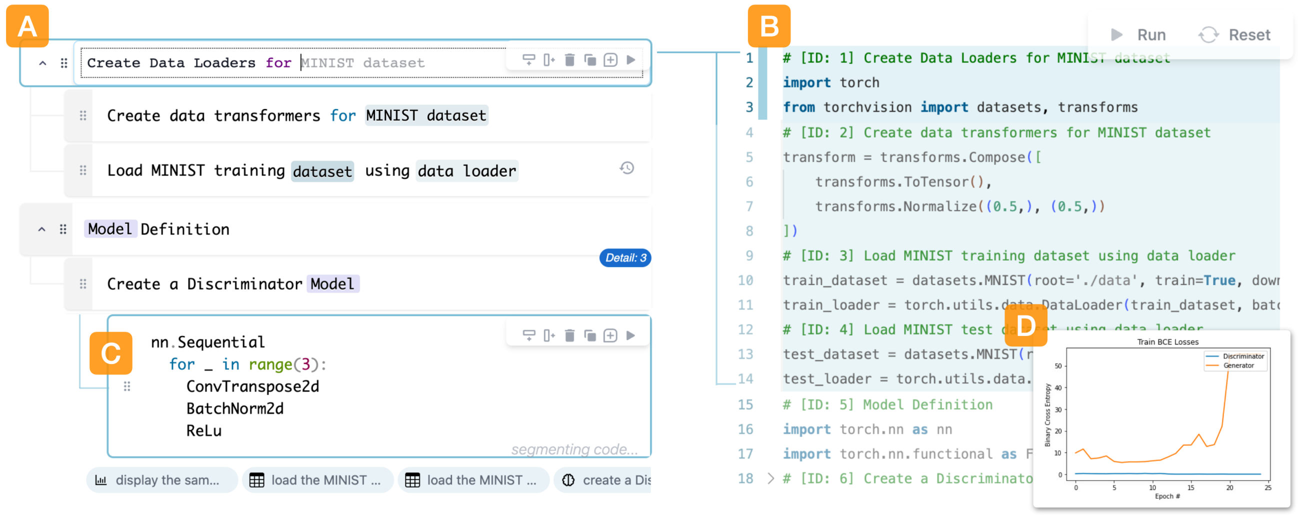
DG2: Direct Manipulation of Prompt Blocks for Code Modification. Programmers often experience control loss when verifying and modifying large amounts of generated code (**C2**). This issue highlights the need to provide programmers with flexible control over the prompt authoring process at different levels of granularity. Such support should include the ability to easily identify, locate, and select the range of modifications (**S6**), insert new prompts (**S4**), and reorganize the structure of prompts (**S5**). The system must facilitate modifications at different levels of abstractions (e.g., high-level tasks or individual prompts) that allow programmers to make the necessary changes without inadvertently modifying unrelated code segments. In addition, the system should generate results incrementally (**S3**) rather than generating entire code simultaneously and overwhelming the programmer.

DG3: Enabling Code Verification during Prompt Authoring. During the iterative refinement of prompts, programmers often experience cognitive overload due to the disruptive context switching between code verification and prompt authoring (**C3**). Findings highlight the possibility of assisting programmers in verifying generated results during the prompt authoring process without necessitating additional context switching. The system should deliver feedback in a non-intrusive manner and provide context to reflect the system's understanding of the task. This approach helps programmers understand whether the system accurately captures their intent (**S7**).

DG4: Assisting Intent-Code Navigation Across Multi-Levels. To facilitate navigation and modification across various levels of abstraction, from user intents to the final generated code (**C4**), the system must ensure correspondence between prompts and code, including a matching between the overall task and code structure. The system should also provide visual cues to highlight the segments of the generated code according to the prompt structure (**S9**), for programmers to navigate and modify the desired code segments. In addition, the system should enable programmers to write prompts containing code syntax (**S8**) to help them efficiently pinpoint the corresponding code.

3.4 Usage Scenario

Casey is a data scientist who wants to build a regression model on a wine-quality dataset. While being experienced in Python,



```
Create Data Loaders for MINIST dataset
    Create data transformers for MINIST dataset
    Load MINIST training dataset using data loader
Model Definition
    Create a Discriminator Model
        nn.Sequential
          for _ in range(3):
            ConvTranspose2d
            BatchNorm2d
            ReLu
```

```
# [ID: 1] Create Data Loaders for MINIST dataset
import torch
from torchvision import datasets, transforms
# [ID: 2] Create data transformers for MINIST dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
# [ID: 3] Load MINIST training dataset using data loader
train_dataset = datasets.MNIST(root='./data', train=True, dowr
train_loader = torch.utils.data.DataLoader(train_dataset, batc
# [ID: 4] Load MINIST test dataset using data loader
test_dataset = datasets.MNIST(r
test_loader = torch.utils.data.
# [ID: 5] Model Definition
import torch.nn as nn
import torch.nn.functional as F
# [ID: 6] Create a Discriminato
```

Figure 3: *CoLadder* comprises four key components: (A) the prompt tree editor, allowing programmers to decompose their intent into smaller prompt blocks; (B) the code editor, facilitating code verification and editing; (C) the prompt block, enabling programmers to compose prompts in *mixed mode*, incorporating both code and natural language; and (D) the execution result panel, which displays the execution result and any associated error messages.

she aims to leverage LLM-driven code assistant to speed up her development process, and thus she launches *CoLadder*. Casey starts by considering the main steps to approach this task by outlining primary objectives, such as partitioning the dataset, building and evaluating the regression model, and plotting the results.

Prompt Authoring with Hierarchical Decomposition. Casey translates her intent into a set of high-level prompts, such as *"Train Regression Model"* (Fig. 2 A), to externalize the code structure she envisioned. Next, she goes deeper and adds some sub-tasks under the high-level prompts with the **[Add Child]** button, adjusting the level of detail as needed (Fig. 2 B). For example, under *"Train Regression Model"*, she adds sub-tasks such as *"Partition the Dataset."* Casey maintains this breadth-first approach, gradually detailing each high-level task with the **[Add Siblings]** button. As Casey added each prompt block, the code was updated in real-time according to the overall task structure. However, the code editor displayed only the code for existing prompt blocks, with the rest of the code segments remaining folded. When Casey crafts these prompts, she leverages the prompt **[Auto-Complete]** feature to quickly formulate more detailed prompts (Fig. 2 C). In some cases, she uses code syntax expressions such as `for epoch in range(1, 31):` or `load_boston()` without the need to translate the code statements to NL (Fig. 2 D). To define finer-grained steps under each sub-task, Casey sometimes adds sub-prompt blocks manually and sometimes utilizes the **[List Steps]** feature, which automatically suggests step-by-step guidance for the code to generate. For example, under *"for epoch in range(1, 31)"*, the listed steps recommend actions like *"Predict on Train and Test data"* that is summarized from the relevant code snippets concerning the model prediction (Fig. 2 E), which assists her in verifying the alignment of the generated code with her intended actions.

Navigating and Verifying through Multi-level Prompts. Having crafted her prompts, Casey navigates through the hierarchical structure with up/down arrow keys, and *CoLadder* highlights the code segments corresponding to the specific prompt block for easy verification. The **[Semantic Highlight]** feature helps her correlate phrases in her prompts with the code segments (Fig. 2 F). For instance, her prompt mentions the " `Predict` *on train & test data*" results in highlighted `.predict()` in the code segments and phrases in the prompt block (e.g., " `Train` *Regression Model*") at the parent level with lower opacity representing lower correlation. Casey now affirms that the LLM accurately interpreted her intent based on the tree structure.

Modification and Block-based Operations. As Casey navigates through prompts, she identifies code segments that require adjustments. In the block *"Train Regression Model"*, she modifies the prompt by the **[Supplement]** feature to specify using L2 regularization and resulted in changing the code from `LinearRegression()` to `Ridge(alpha=0.5)` (Fig. 2 G). Further, she uses the **[DnD]** feature to relocate the block *"Plot Loss Curve"* under the block *"for epoch in range(1, 31)"*, resulting in the generation of a plot for each training epoch (Fig. 2 H).

In the end, Casey compiles and runs her code, observing that the system successfully outputs 30 graphs displaying loss curves that match her intents.

4 COLADDER

Following the design guidelines (Section 3), we iteratively designed the *CoLadder*, a system that helps programmers hierarchically translate their intent to generate code at multi-level abstractions with flexibility (i.e., *Hierarchical Generation*) and direct manipulation

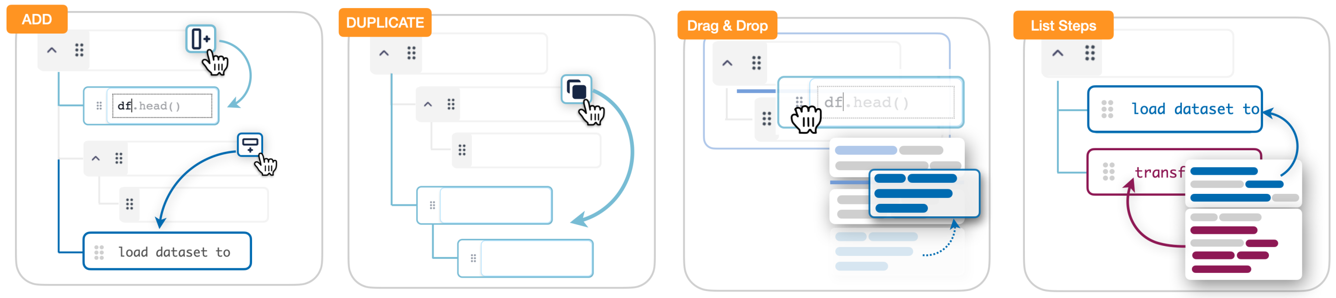


Figure 4: *ADD* Operation enables the addition of sibling blocks or child blocks to the existing prompt structure; *DUPLICATE* operation allows for the cloning of sub-trees; *DnD* operation empowers programmers to reorganize prompt blocks; *List Steps* provides high-level, summarized steps from the generated code.

of the code segments from the prompt through a modular block-based design. *CoLadder* consists of two main UI components: 1) The **prompt tree editor** (Fig. 3 A) allows the programmer to externalize their mental model by decomposing the programming task into smaller prompt blocks (Fig. 3 C); 2) The **code editor** (Fig. 3 B) allows the programmer to verify the generated code and directly edit the program. The programmer can also compile and run the current code to see the results or errors below the code editor (Fig. 3 D).

In the next sections, we will discuss *CoLadder*'s functionalities and design in detail based on **DG1-4**. During the development of *CoLadder*, we incorporated insights from six experienced programmers ($M = 7.88, SD = 4.34$ years) familiar with LLM-based code assistants ($M = 8, SD = 2.56$ times/week). These insights are derived from cognitive walkthrough experiments conducted during the iterative design process.

4.1 From Task Structure to Code Structure

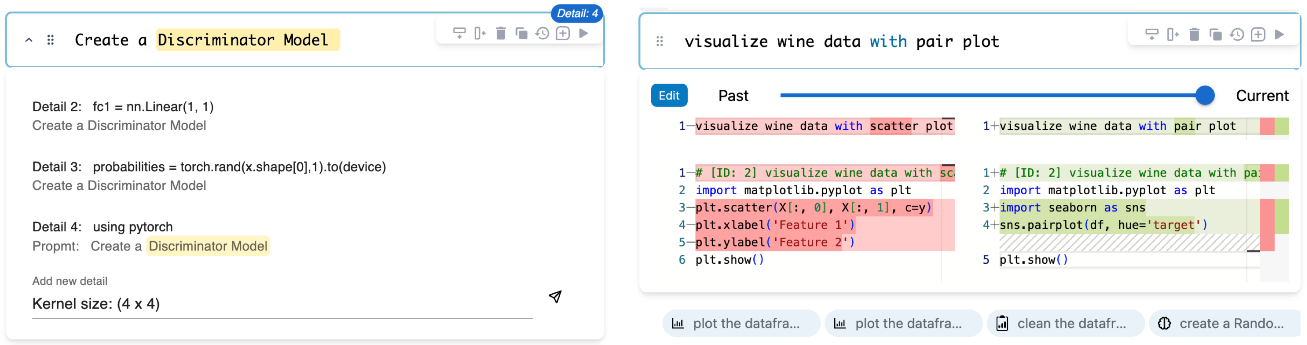
To assist programmers in structuring their prompts hierarchically to externalize their mental models (**DG1**), we offer a tree-based prompt editor that enables programmers to construct prompts that reflect both the task and code structure. Furthermore, we decomposed tasks into smaller sub-tasks at multiple levels of abstraction. This approach enables programmers to directly manipulate their intent to code based on the hierarchical structure (**DG2**).

4.1.1 ***Prompt Tree Editor.*** The tree-based visualization helps programmers organize tasks in line with the top-down mental programming model. This tree editor allows programmers to convey task structure through the horizontal indentation of sub-tasks, while still maintaining the program structure vertically. For instance, if a programmer has added a task, *"Extracting quotes from a web page,"* they can represent the hierarchical task structure and execution order of the code by adding a sub-task, *"Find all class=quote,"* indented underneath. Rather than automatically decomposing tasks, *CoLadder* allows programmers to freely construct prompts that align with their mental models. We further implemented a foldable prompt tree that corresponds to the foldable code editor based on expert suggestions from the walkthrough. This is especially practical for longer programs that require scrolling up and down for verification and modification.

4.1.2 ***Prompt Block.*** Each decomposed task in the tree nodes is referred to as a *prompt block*, where programmers can write the prompt in *mixed mode* (Fig. 3 C). This means programmers can input both NL and code syntax (e.g., control statements or function declarations). Each block functions as a miniature code editor, with semantic highlighting of code syntax and prompts. Several participants (N=4/6) expressed wanting a revision history of prompt blocks to assist them in recalling the rationale behind certain prompts. This form of local versioning benefits rapidly iterative programming tasks such as data science and exploratory programming [52, 53, 83]. Therefore, *CoLadder* documents the version of each iteration of a prompt block, visualizing differences in prompts, and generated code across each iteration. This allows programmers to quickly navigate through prompt iterations and observe the differences between them in order to recover a particular iteration as needed (Fig. 5 Right).

4.1.3 ***Block-based Operations.*** *CoLadder* supports direct manipulation of the prompt structure by providing several prompt block operations that are activated through either buttons or shortcuts. Each block-based operation will update the corresponding code and propagate changes to the rest of the code as necessary.

1. **[Add]** either a block as a sibling (same level) or child (sub-level) based on their intent (Fig. 4 ADD). After adding a block, the programmer can start entering their prompt to guide the system to generate code based on the current task structure in the prompt tree editor;
2. **[Edit]** allows programmers to refine prompts when they want to modify specific code segments. This modification follows the hierarchical structure, meaning that when programmers edit a parent block encompassing multiple child blocks, the resulting changes affect all code segments within those child blocks, ensuring consistent updates across related sections. For instance, when programmers edit a parent block from *"train linear regression model"* to *"train logistic regression model,"* the code segments within the sub-task *"create mode"* will also be updated to align with the new logistic regression model.
3. **[Delete]** unneeded prompt blocks (e.g., parent blocks and all their children). Similar to the **[Edit]**, this operation will only affect a segment of the code and propagate the changes across the rest of the code to prevent errors.



```
Create a Discriminator Model
Detail: 4

Detail 2:  fc1 = nn.Linear(1, 1)
Create a Discriminator Model

Detail 3:  probabilities = torch.rand(x.shape[0],1).to(device)
Create a Discriminator Model

Detail 4:  using pytorch
Propmt:  Create a Discriminator Model

Add new detail
Kernel size: (4 x 4)
```

```
visualize wine data with pair plot

Edit   Past                                  Current
-visualize wine data with scatter plot        +visualize wine data with pair plot

-# [ID: 2] visualize wine data with sc        +# [ID: 2] visualize wine data with pa
 import matplotlib.pyplot as plt               import matplotlib.pyplot as plt
-plt.scatter(X[:, 0], X[:, 1], c=y)           +import seaborn as sns
-plt.xlabel('Feature 1')                      +sns.pairplot(df, hue='target')
-plt.ylabel('Feature 2')
 plt.show()                                    plt.show()
```

Figure 5: Supplement View (Left): Programmers can add additional details to their prompts via a conversational UI. It also shows the history of supplements and expand or hide by clicking the top right icon; History's Diff View (Right): programmers can observe various types of changes (such as edits, additions, supplements), iterations of prompts, and their corresponding generated code in the diff view.

4. **[Duplicate]** copies prompt blocks, and if the code block is a parent block with sub-blocks, it duplicates all its child blocks, creating an identical structure (Fig. 4 DUPLICATE);
5. **Drag and Drop ([DnD])** restructure prompt blocks or elevate certain code segments to a higher-level scope, allowing them to create reusable functions or methods accessible by other parts of the program (Fig. 4 Drag & Drop);
6. **[Supplement]** add extra details either to the entire prompt block or specific phrases within the NL prompt. This feature addresses the need for additional information required by the Language Model (LLM) but not necessarily by the programmer to understand the program. For example, in a web crawling program, specifying the URL to be crawled is critical for the model but trivial for the programmer. Following expert suggestions, we introduced a conversational interface to incorporate supplementary information (Fig. 5 Left). After a programmer submits supplementary information, it will be displayed as a small badge in the top right corner of the prompt block, without being directly added to the prompt text. This design choice ensures that the supplementary details are readily accessible but doesn't clutter the main prompt, maintaining the clarity of the prompt structure.

4.2 Verify Results from Prompt

CoLadder offers diverse informative feedback to assist programmers in assessing whether the system accurately comprehends their intent (**DG3**). This capability enables programmers to concentrate on crafting prompts without experiencing disruptive cognitive shifts between prompt authoring and code verification.

4.2.1 ***List Steps at Lower-Level.*** To help programmers verify the generated code effectively, we support the *List Steps* operation, which allows programmers to understand how the model generated the code for the current prompt (Fig. 4 List Steps). This feature can be used to obtain a high-level summary of the code generated by the LLM. *CoLadder* will semantically segment the generated code and return a step-by-step description of the implementation in the form of *mixed mode*. The returned steps are linked as sub-blocks of the current prompt block, and programmers can further *EDIT* them if needed. This feature serves as scaffolding for programmers, aiding them in comprehending lower-level details, starting from sub-tasks NL prompts, all the way to the pseudocode and actual code.

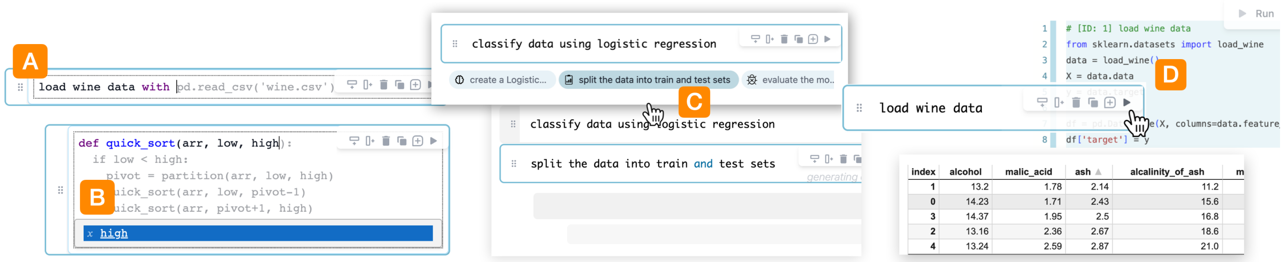
4.2.2 ***Auto-completion.*** According to the findings from the formative study, programmers value the in-line auto-completion and view it as a step to verify the result. *CoLadder* support programmers with two types of auto-complete while editing prompt blocks: 1) Word-level auto-completion based on variables or semantically related naturalistic utterance (Fig. 6 B), and 2) Sentence-level prompt auto-completion from LLM suggests in-line auto-completed prompts based on the context of the current tree structure (Fig. 6 B). Programmers can press the tab button to accept these suggestions.

4.2.3 ***Recommendation.*** After adding a new prompt block, *CoLadder* provides multiple possible next steps to the programmer (Fig. 6 C). These recommended prompts are displayed below the current prompt block in order of relevance scores suggested by the model. Programmers can select one of them to add below. This feature assists programmers in accomplishing their goals in a step-by-step manner and helps them verify if *CoLadder* correctly understands their intent by successfully recommending the appropriate next steps.

4.2.4 ***Interim Results.*** The **[Compile]** operation is provided for each prompt block that allows programmers to independently execute them to display the interim results (Fig. 6 D). *CoLadder* wraps the code according to the prompt tree structure and compiles the code to display the results. Programmers can verify the interim results to determine the next step.

4.3 Navigating Across Multi-Levels

CoLadder offers features for programmers to navigate various abstraction levels to easily locate and modify targeted prompt blocks (**DG4**).



```
load wine data with pd.read_csv('wine.csv')
```

```
def quick_sort(arr, low, high):
  if low < high:
    pivot = partition(arr, low, high)
    uick_sort(arr, low, pivot-1)
    uick_sort(arr, pivot+1, high)
x high
```

```
classify data using logistic regression
```

```
classify data using logistic regression
split the data into train and test sets
```

```
load wine data
```

```
# [ID: 1] load wine data
from sklearn.datasets import load_wine
data = load_wine()
X = data.data
y = data.target

df = pd.Dat   e(X, columns=data.feature
df['target'] = y
```

index	alcohol	malic_acid	ash ▲	alcalinity_of_ash	m
1	13.2	1.78	2.14	11.2	
0	14.23	1.71	2.43	15.6	
3	14.37	1.95	2.5	16.8	
2	13.16	2.36	2.67	18.6	
4	13.24	2.59	2.87	21.0	

Figure 6: (A) Auto-completion that could be completed in the format of natural language and code syntax; (B) Auto-completion based on the variable name used before; (C) Recommendation feature that suggests the next step based on the prompt tree structure; (D) Live execution showing the interim results of the current block.

4.3.1 *Showing Corresponding Code.*

Though all participants found the *CoLadder* helps them verify results from prompts, there were instances when they still needed to confirm the exact generated code. Interestingly, they found that verifying individual code segments was often sufficient, as they trusted the system to seamlessly connect and execute blocks accurately. In response, we offer a code editor view that highlights only relevant code when programmers select the corresponding prompt block, folding other code segments. Different decorations are provided on the code editor's glyph to illustrate the current location of the prompt in the prompt tree. For finer adjustments, programmers can directly modify the code in the code editor, where changes will be propagated back to the corresponding prompt block.

4.3.2 *Semantic Highlight and Dependency.*

CoLadder offers two types of syntax highlighting for prompt blocks: semantic highlighting to differentiate between code and NL in mixed-mode editing. Secondly, highlighting NL phrases to help understand dependencies across prompt blocks at different levels. This is beneficial when the programmer refers to the same variable in the code with different terms (e.g., df, data, table that all refer to the same dataframe). The programmer can select phrases within the prompt, and *CoLadder* will display semantically related phrases throughout the tree, with different entity types shown in distinct colours, and opacity determined by the relevance score (Fig.7). The relevant code segment is also highlighted in the code editor for the programmer to easily identify the phrases from the prompt to the corresponding generated code.

4.3.3 *Keyboard Navigation.*

To allow programmers to focus on prompt authoring, *CoLadder* offers keyboard shortcuts that enable them to swiftly access all the features mentioned above. The arrow keys (↓/↑) are used to move through prompt blocks at various abstraction levels and the code editor will highlight the corresponding code segments. Programmers can also use `Enter` to start editing, `Esc` to record the editing, `Alt` + ↓ /↑ to create siblings/ children/ and `Alt` + `Enter` to activate the *List Steps* feature.

4.4 System Implementation

CoLadder is built using React Typescript, with the code editor employing Monaco Editor [68]. It utilizes the Python web compiler provided by Pyodide [78] and leverages the OpenAI GPT-4 API [75] for performing hierarchical code generation.

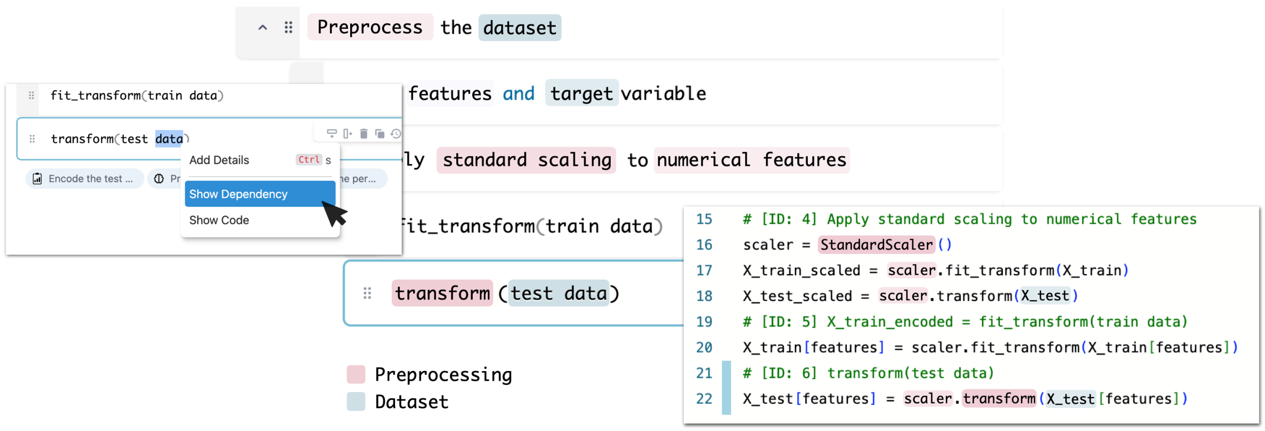
4.4.1 *Tree-based Prompting.*

Instead of simply providing the entire program as a context to the LLM, *CoLadder* utilized a tree structure to provide the programmer with a block-based programming experience. We iteratively developed prompt templates so that the generated program matches the structure of the prompt co-constructed by the programmer and *CoLadder*. The tree structure allows programmers to maintain control while crafting and editing programs block-by-block, making it easy to scaffold mental models without having to verify the entire program. We developed and tested the prompt template using OpenAI's GPT-4, the most advanced and publicly available LLM to date. The following paragraphs describe how different prompt templates are designed to achieve each block operation.

Add a Prompt Block. After the tree structure is parsed based on the tree view, the current prompt block and tree structure will be passed to LLM as the context of the prompt template. We instruct LLM that the generated code should follow a bottom-up approach, starting with the nodes at the lowest indentation levels and combining the children's code with that of the parent nodes. We include a set of few-shot examples for the model to learn the tree structure and an output parser to structure LLM responses into a tree format with nodes containing prompt and code.

Editing. While each prompt block corresponds to a segment of code, programmers can select a prompt block and only apply editing within that range of code without re-rendering all programs. While the intention to edit a prompt might vary, we have stored all the prompt modification history and encoded it into a vector database to avoid LLM re-generating the same result. These buffer memories also serve as a cache that the *CoLadder* could semantically search for the most related results to re-generate by capturing the difference between variants to identify programmers intent. The above mechanism is also applicable to other prompt block operations (e.g., delete, duplicate, DnD, and supplement).

Error Prevention and Correction. As we generate code in a block-by-block manner within the tree structure, errors in the generated code are infrequent. However, code segments occasionally do not align with the rest of the code. To address this, we designed a prompt



```
# [ID: 4] Apply standard scaling to numerical features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
# [ID: 5] X_train_encoded = fit_transform(train data)
X_train[features] = scaler.fit_transform(X_train[features])
# [ID: 6] transform(test data)
X_test[features] = scaler.transform(X_test[features])
```

Figure 7: Programmers can select a phrase within the prompt, and the system will highlight correlated phrases throughout the tree structure and code segments. Colours are used to represent the entity type, while opacity indicates the correlation score.

template called *Propagate Changes* to recursively assess the corresponding code from each prompt block for necessary modifications. After performing prompt block operations, *CoLadder* establishes a *sequential chain* to utilize preceding operation outputs as inputs for propagation. This mechanism proves particularly useful when consecutive interactions are desired, forming a pipeline to execute specific scenarios [96, 97].

5 EVALUATION

We conducted a user study with 12 experienced programmers who regularly use LLM-based code assistants to investigate whether *CoLadder* helps programmers externalize mental models to solve the programming task (**RQ1-DG1**); enables direct manipulation of prompt blocks when translating intent into code (**RQ2-DG2**); assists programmers to verify results during the prompt authoring process (**RQ3-DG3**); and supports programmers navigate and modify across multi-levels of abstractions (**RQ4-DG4**).

5.1 Participants and Apparatus

We recruited 12 participants (7 males, 5 females; ages 23 – 36, $M = 26, SD = 3.54$) through convenience sampling and a mailing list. All participants are experienced programmers ($M = 7.88, SD = 4.34$ years). They also regularly use AI-code generation tools ($M = 8, SD = 2.56$ times/week) and self-reported being familiar with them based on a 5-point Likert scale ($M = 4, SD = 0.74$).

In addition to *CoLadder*, we implemented a *Baseline* with which to compare *CoLadder*. *Baseline* is a code editor that generates code based on inline comments, similar to GitHub Copilot [38]. Both *Baseline* and *CoLadder* are powered by GPT-4, to ensure a fair comparison. Both systems are deployed as web applications, ensuring consistent functionality across all operating systems and web browsers. Participants could either join in-person or remotely through the online video conferencing platform, Zoom. Each study session lasted about 60-75 minutes, and participants were compensated with $30 in local currency. The study was approved by the university's ethics review board.

5.2 Procedure

To compare *CoLadder* and *Baseline*, we used a within-subject design. We began the study by giving the participant an overview of the study procedure. Users completed a consent form and a pre-study questionnaire that collected their demographic information education level and Python proficiency. We randomly assigned a task category (machine learning or data visualization) to each participant. Every participant completed two Python programming tasks based on their assigned category, one using *CoLadder* and the other using *Baseline*. Before each task, participants were given a tutorial on the system and a chance to try it out. Once they felt comfortable, participants had up to 12 minutes to attempt the task. They were encouraged to vocalize their thought process during task completion. After each task, participants completed a post-task questionnaire evaluating the usability and utility of *CoLadder* and *Baseline*. Usability was measured using the UMUX-LITE scale which is directly related to the SUS score [57], and the NASA-TLX scale for perceived cognitive load [41]. Utility was measured using self-defined Likert scale items (Fig. 8). At the end, we conducted a semi-structured interview to gain user insights about the systems and to understand their behaviour during task completion.

6 FINDINGS

Our results indicate the effectiveness and usefulness of *CoLadder* in facilitating programmers to flexibly decompose their tasks, thereby externalizing their mental models. Additionally, the concept of *hierarchical generation* further scaffolds programmers' intent into code across multi-level abstraction, all while providing controllability. We present a detailed qualitative analysis and system log data in themes corresponding to the four **D**esign **G**uidelines and **R**esearch **Q**uestions.

6.1 General Impression

We found that more participants were able to complete the task using *CoLadder* (7 out of 12 participants, 7/12 henceforth) compared

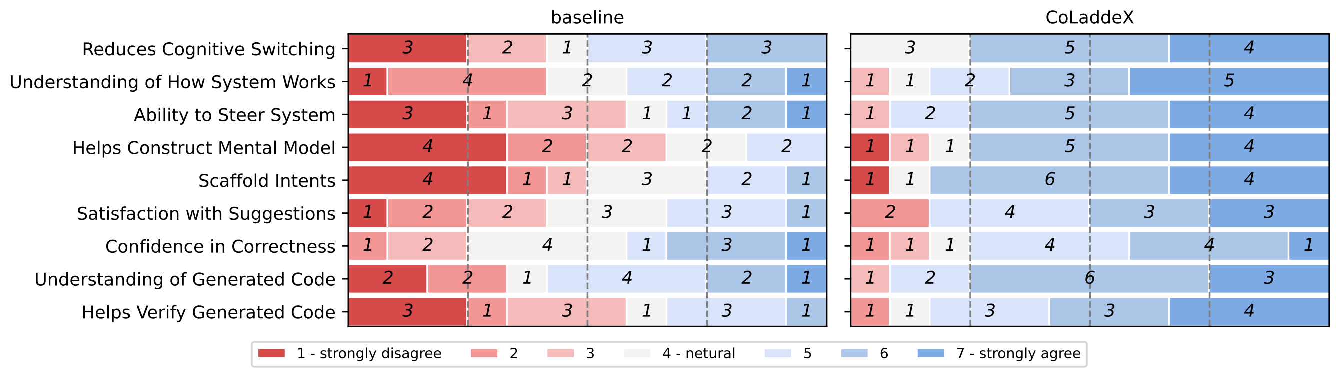


Figure 8: User perception of utility of *Baseline* and *CoLadder*, measured on self-defined 7-point Likert scales.

to *Baseline* (6/12). However, on average, participants took significantly more ($p = .040$) time to complete tasks with *CoLadder* ($M = 11.74, SD = 0.50$ min) compared to *Baseline* ($M = 10.05, SD = 2.79$ min). Compared to *Baseline*, participants found that when using *CoLadder*, they were more satisfied with the suggestions of the system (Mdn = $5.5 > 4.0, p = .030$; i.e. $\text{Median}_{CoLadder} = 5.5 > 4.0 = \text{Median}_{Baseline}$, p-value = 0.030), and had more confidence that the system-generated code was correct (Mdn = $5.0 > 4.0, p = .6$).

To measure the usability of *CoLadder*, we computed the SUS scores based on the UMUX-LITE. The average SUS scores were significantly greater ($p = .02$) for *CoLadder* (Mdn = 90.61), compared to *Baseline* (Mdn = 68.94). Typically, a SUS score above 70 is considered "acceptable" and one above 85 "excellent" [9]. This indicates that *CoLadder* has good usability and is much more usable than *Baseline*.

We also used NASA-TLX to measure the perceived workload associated with each system. Compared to *Baseline*, *CoLadder* had lower mental (Mdn = $3.0 < 4.5, p = .10$), physical (Mdn = $1.0 < 2.0, p = .08$), and temporal (Mdn = $3.0 < 4.5, p = .16$) demand, required less effort (Mdn = $3.0 < 4.5, p = .39$), and led to better performance (Mdn = $6.0 > 5.0, p = .11$) and statistically significantly less frustration (Mdn = $2.0 < 5.0, p = .04$). The overall perceived workload, obtained by averaging all six raw NASA-TLX scores (with the "Performance" measure inverted), was also lower for *CoLadder* than *Baseline* (Mdn = $2.33 < 3.75, p = .11$). Thus participants found *CoLadder* to be less taxing to use compared to *Baseline*, though this difference was not statistically significant.

6.2 Multi-level Prompt Structure to Externalize Mental Models (DG1-RQ1)

Participants felt the prompt tree structure of *CoLadder* helped construct and externalize their mental models for solving the programming task compared to *Baseline* (Mdn = $6.0 > 2.5, p = .008$). Every participant constructed a prompt tree with at least 2 layers, leveraging the prompt tree as an externalization of their mental task structure. As P2 mentioned, *"the tree structure clarifies my approach both in solving the task and in guiding the model to generate the desired code."* Participants mostly (11/12) structured their prompts to horizontally map to the task structure through indentation (e.g., tasks to sub-tasks), while also aligning the order of prompts vertically with the structure of the generated code. P6 elaborated: *"I prioritize defining the overall task structure, using child blocks to differentiate sub-tasks [...] and will adjust the code structure later on based on its execution sequence."* While the flexibility of *CoLadder* allows free-form prompt structuring, our observations revealed two major workflows, as illustrated in Fig. 9:

1. **Breadth-First Search**: Participants (4/12) outlined all primary tasks first and subsequently delved into the finer details by adding sub-tasks (or *Supplements*). Before moving to sub-tasks, they utilized the *List Steps* feature to verify how the code is being implemented.
2. **Depth-First Search**: Participants (7/12) addressed all sub-tasks within a main task before moving on to the next primary task. Participants with a well-defined mental model beforehand were more inclined to adopt this approach, leveraging the *List Steps* feature as a *"cross-validation mechanism" - P8* to ensure the generated code aligned with their specified sub-tasks.

Without the prompt tree structure, participants using *Baseline* typically faced two challenges. First, some participants (4/12) spent a significant amount of time mapping out the task to code, often constructing a lengthy section of comments that included all the steps required to approach the task. This approach required them to *"think about the code in very detail first." - P2* On the other hand, most participants (8/12) developed their mental model while authoring the prompt by adding more context to the prompt to adjust the output. Participants reported that using the linear representation of prompts *"could not fully express their thoughts." - P4*

In summary, participants using *CoLadder* with both approaches found that the tree structure encouraged them to *"contemplate the implementation of the code layer by layer," - P5* alleviating the *"cognitive burden of thinking about the entire code structure beforehand." - P11* This hierarchical organization allowed them to decompose tasks based on their mental model, addressing RQ1 and fulfilling DG1.

6.3 Controlled Scaffolding from Intent to Code (DG2-RQ2)

Participants found *CoLadder* to be more helpful in scaffolding their intents to generate the desired code (Mdn = $6.0 > 3.5, p = .007$) compared to the baseline. They also reported that they could steer *CoLadder* more controllably towards the task goal (Mdn = $6.0 >$

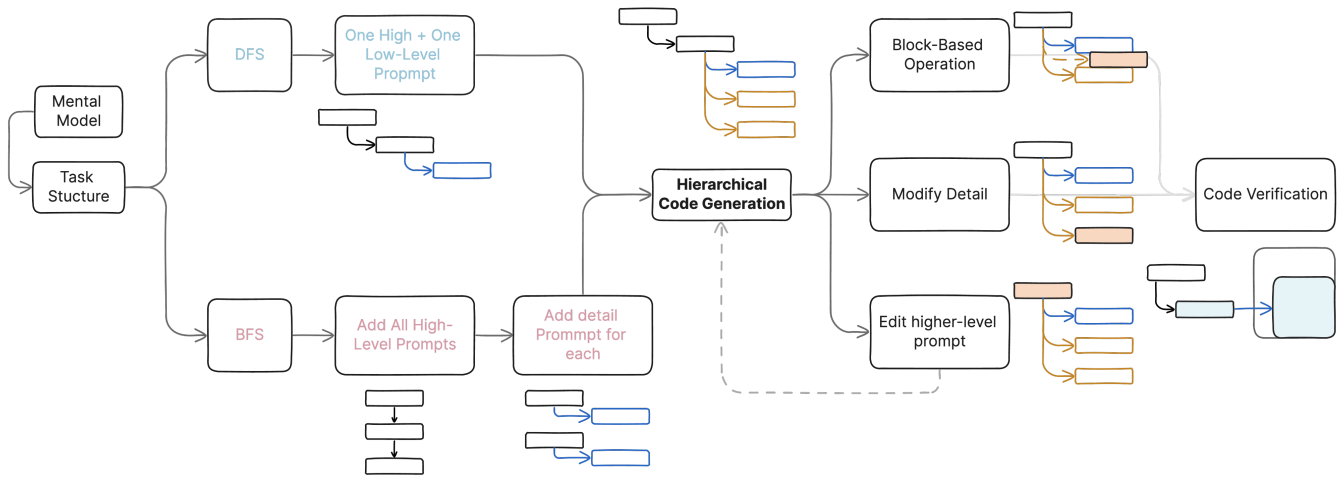

Mental Model
Task Stucture
DFS
One High + One Low-Level Prompmt
BFS
Add All High-Level Prompts
Add detail Prommpt for each
Hierarchical Code Generation
Block-Based Operation
Modify Detail
Edit higher-level prompt
Code Verification

Figure 9: Participants typically initiate their workflow by externalizing their mental models through either a Depth-First Search (DFS) or Breadth-First Search (BFS) approach. Subsequently, *CoLadder* generates code structured in alignment with their carefully crafted task structure. Following code generation, participants proceed with block-based operations, supplement additions, and, in some cases, modify higher-level prompts to regenerate the code. Participants using *CoLadder* typically verify and compile the code at the final stage, whereas those in the *Baseline* condition often need to verify and compile code more frequently throughout the process.

3.0, $p = .007$) with the *"step-by-step approach" - P6*. Participants reported the modular-based design (i.e., prompt blocks) allowed them to focus on one segment of code at a time, which prevented them from *"getting lost while verifying the generated code." - P2* Participants could also easily identify where to add prompt blocks because they *"only need to ensure that the high-level task structure is correctly ordered, without having to search through the code to find the exact position." - P1*

When comparing the authoring processes in both systems, participants (11/12) felt that *CoLadder* provided them with more controllability while modifying code. They found it useful for modifying the targeted code segments, *"without having to worry about affecting other sections." - P4* Participants highlighted the block-based operations (e.g., *DnD*) could help them focus more on structuring prompts, *"I love the drag and drop feature, which allows me to structure the code freely based on my mental model without concerns about the code's structure." - P7* Participants also found that the tree structure assisted them in identifying where to modify the code more easily compared to *Baseline*. *"It's simpler to make changes to the code [with CoLadder] when there's an error. [...] I can modify the parent level and the changes will be reflected in all child blocks as well." - P8*

We observed that most participants (10/12) do not verify the entire program but verify them segments by segments, as they do not worry about the *"possibility of the LLM incorrectly concatenating my [their] code or using different variable names." - P11* Overall, the block-based design enhances participants' ability to directly manipulate components of the prompt to modify specific code segments (DG2). This increased control allows participants to scaffold their intent to generate code segment by segment effectively (RQ2).

6.4 Code Verification during Prompt Authoring (DG3-RQ3)

Compared to *Baseline*, participants found that *CoLadder* significantly reduced the need for cognitive switching between prompt authoring and code verification (Mdn = $6.0 > 4.5, p = .01$). We also observed that participants typically began verification after drafting the initial prompt tree structure. P6 explained, *"The generated code will not be accurate unless I provide details [e.g., by adding child blocks, Supplement operation]"*. All participants experimented with the *List Steps* feature in *CoLadder*, primarily to *"assess generated code alignment with [their] intents." - P9* P2 explained, *"If the steps are correct, I'm confident the code will be too."* This approach was similarly adopted with the *Recommendation* feature and *Auto-Complete* features; participants mostly utilized them *"as a cue to see if the system captured my intent" - P8*. Specifically, participants leverage recommendations to verify the alignment between their intents and the system's comprehension, rather than intending to actually use them. Without these features in *Baseline*, participants have to manually identify specific changes in the code segments corresponding to their prompt modifications by *"comparing the previous and current generated code." - P7* Overall, the purpose of verifying the code for participants using *CoLadder* was to modify their prompts, but not to verify the correctness of the generated code.

Despite the reduced need for frequent code verification, participants using *CoLadder* demonstrated a significantly better understanding of their programs compared to using *Baseline* (Mdn = $6.0 > 5.0, p = .007$). A recall test was conducted to investigate participant code comprehension [32]. Participants could recall code implementation systematically after using *CoLadder*, from higher-level (e.g., the purpose of the task) to lower-level code implementations with details in each step. P11 explained the reason, *"the system [CoLadder] helped me think through the programming task already when I was drafting prompts."* P9 added, *"I do not need to spend time on comprehending code, as I have already verified segments of code corresponded to each prompt before."*

In contrast, in *Baseline*, participants started with detailed codes and gradually summarized and mapped the task steps. We observed that participants compiled the code more in *Baseline* compared to the *CoLadder*, which was used as an alternative approach for *"verifying the generated code." - P12* P1's strategy in using *Baseline* was to *"try to compile the code to see if it works,"* without the need to verify the generated code. However, participants using this approach in *Baseline* faced challenges to modify the code when compiled results were incorrect, where they *"had to verify by cross-referencing task descriptions and generated code." - P1*

In summary, *CoLadder* reduces disruptive cognitive switching between code verification and prompt authoring by enabling features to assist in results verification during the prompt authoring process (RQ3, DG3). Additionally, it offers participants a clearer understanding of their own program compared to *Baseline*.

6.5 Navigation Across Multi-Level (DG4-RQ4)

Participants navigated across various levels of prompt blocks during code verification and noted that *CoLadder* significantly enhanced their ability to verify generated code (Mdn $= 6.0 > 3.0, p = .006$) in comparison to *Baseline*. Participants found it more effective to navigate and make precise modifications with *CoLadder* as they could adopt a *"modular approach" - P11*. They could verify code in segments with the code highlight feature, which *"reduced cognitive load." - P4* Folding and presenting different segments of code depending on the structure of the task allows programmers to *"more easily control the depth of verification required." - P6* Some suggested that having a structural mental model formed beforehand allowed them to *"swiftly locate the segments needing changes." - P8*

As the prompt blocks increased, participants valued the *Semantic Highlighting* feature that simplified the linkage between the task and its corresponding code. The majority (10/12) felt that the *Mixed Methods* writing feature accelerated the verification process by checking *"if the keyword is showing in the right place as [they] thought." - P2* While some participants (4/12) added code syntax to the prompt when using *Baseline*. They did so primarily as an intervention method when the generated code consistently produced errors, rather than as a means to facilitate the verification process. We also observed that participants preferred the use of inline keywords (e.g., *read_csv()*, *sns.pairplot()*) rather than multi-line code when drafting prompts, where they could *"easily identify and track changes" - P11* across multi-level abstractions.

Overall, these findings demonstrate how the *Semantic Highlight* and *Mixed Method* writing supported in *CoLadder* effectively support programmers in navigating and modifying prompts across multiple levels of abstractions (RQ4, DG4).

7 LIMITATIONS

Our primary limitation is associated with the diversity of programming tasks evaluated and the constraints of prompt programming. Some open-ended programming tasks, particularly those focused on rapid iteration (e.g., exploratory programming, exploratory data analysis) [54, 90], tend to prioritize quick idea iteration over code quality. Programmers often need to make swift, small changes such as adjusting parameters and variables [85, 103]. NL-based programming may not be as effective in these cases, as programmers can quickly modify specific parts of the code without waiting for code generation. Some participants (3/12) expressed their desire to have the ability to modify the code directly within *CoLadder*, stating that they wanted to *"adjust some low-level code details directly [in CoLadder]" - P10*. This finding implies that although *CoLadder* successfully enhances control over the code generation process (**RQ2**), we do not significantly improve the control over the program itself. Despite the option to make direct code changes within *CoLadder*, participants mainly focus on prompt authoring and incorrectly perceive that they *"always have to refine the prompt to change the code" - P6*. Future research should explore two facets of controllability that are essential for programmers: controllability in their interactions with AI [5, 45, 73] and controllability over the program itself [39, 70].

Another limitation of *CoLadder* is that the prompt tree structure may not always align with the actual code structure. For instance, while in the programmer's mental model, *"plotting the loss curve"* may be considered a sub-task under *"model evaluation"*, it might exist within the global function scope in the generated code. While we initially designed the separation of the user interface for the prompt editor and code editor with the consideration of this issue, two participants mentioned that our system may be less effective in certain programming languages (e.g., Object-Oriented programming [30]) where the task and code structures can deviate significantly. In our future work, we plan to explore the integration of visualizations such as class diagrams to better express both the program structure and relationships between classes or components [40].

8 DISCUSSION AND FUTURE WORK

We discussed how *CoLadder* assists programmers in constructing mental models for programming tasks, mitigating over-reliance issues, and its potential applicability in both familiar and unfamiliar programming tasks. These insights highlight the valuable role of *CoLadder* and its *hierarchical generation* approach, suggesting design implications for future research in LLM-driven code assistants.

8.1 Mental Models Formation & Development

The findings from our study demonstrate that *CoLadder* effectively supports code generation at multiple levels of abstraction, thereby assisting programmers in scaffolding their intent. This aligns with existing literature on the challenges of understanding the capabilities and limitations of language model-driven code generation systems, as well as the need for clear and naturalistic input to generate specific code that matches the programmer's intent [34, 62, 97]

Additionally, our findings resonate with prior research highlighting the importance of programmers forming mental models of code at different levels of abstraction [84], from specific code statements to larger program structures [8, 101]. This underscores the need for scaffolding useful mental models to facilitate interactions and collaboration between programmers and LLMs in solving programming tasks. In our work, we place a strong emphasis on assisting programmers in forming these mental models to craft effective prompts that generate code aligning with their intentions [22, 99].

Our study highlights the role of *CoLadder* in aiding programmers as they evolve their mental models throughout the problem-solving process [67]. By offering hierarchical prompt structures and block-based operations, *CoLadder* enables programmers to easily adapt and refine their task representations as they gain deeper insights into their programming tasks. This aligns with agile development principles, fostering dynamic adjustments in problem-solving approaches [43].

8.2 Over-Reliance and Program Comprehension

In our findings, we observed the trend where programmers often accepted the suggestions from the baseline code assistant and subsequently modified the generated code. This observation aligns with prior studies that have suggested the possibility of programmers developing an over-reliance on LLM-driven code assistants [10, 24, 100]. However, in the context of *CoLadder*, participants meticulously crafted their programs step by step, demonstrating a deeper understanding of the overall programming structure. This approach appeared to yield benefits in the recall test, as participants showed a greater ability to comprehend the program they had constructed [67].

This finding raises intriguing questions about the potential implications for future maintenance tasks. It prompts further inquiry into whether the careful, step-by-step program crafting facilitated by *CoLadder* might result in more maintainable codebases or offer advantages in scenarios where long-term code comprehension and modification are required [4]. Exploring these aspects in future research could shed light on the effectiveness and sustainability of *CoLadder* in addressing the over-reliance issue commonly encountered in human-AI interaction [5].

8.3 Experiences and Task Familiarity

We designed and studied *CoLadder* with experienced programmers because they can form more well-defined mental models when addressing programming tasks compared to novices [27, 35, 43, 101]. However, we are also interested in how task familiarity might cause differences among experienced programmers. Hence, in the pre-study questionnaire, participants were asked to rate their familiarity with various programming tasks on a 5-point Likert scale. The results showed that all participants reported being familiar with basic machine learning ($M = 4.17, SD = 1.19$) and basic data visualization ($M = 4.25, SD = 0.75$) tasks in Python, as well as overall Python programming ($M = 4.42, SD = 0.51$). We calculated the Spearman correlation between self-perceived familiarity (on a 5-point Likert scale) and the NASA-TLX and self-defined questionnaire responses. The correlation was weak[1] for all items except *"Performance"* ($r = -0.348$) from NASA-TLX, and *"Understanding of How System Works"* ($r = -0.447$) and *"Satisfaction with Suggestions"* ($r = -0.327$) from the self-defined questionnaire, which had moderate correlations.

[1]As a general rule of thumb, a Spearman correlation below 0.3 is considered weak and one between $0.3 - 0.6$ is moderately strong.

Although it is challenging to draw a definitive conclusion regarding whether task familiarity correlates with the usefulness of *CoLadder*, it is worth noting that some participants (2/12) expressed a particular interest in using it for tasks they were less familiar with. Yet, regarding the utility of *CoLadder* in developing a future-maintainable program, participants expressed their willingness to invest more time in crafting the program with *CoLadder*. Potential future work can research the differences between novice and experienced programmers who form distinct mental models when solving programming tasks, to provide insights into how their prior knowledge influences their interactions with code assistants like *CoLadder*.

9 CONCLUSION

In this paper, we present *CoLadder*, an interactive system that aids programmers in code generation and verification. It achieves this by providing hierarchical task decomposition, modular-based code generation, and result verification during prompt authoring. Our iterative design process and interview study uncovered strategies and programmers' needs for externalizing mental models and scaffolding intents for code generation. A user study involving experienced programmers further validated *CoLadder*, demonstrating its capacity to enhance programmers' ability to navigate and edit code across various abstraction levels, from initial intent to final code implementation. In summary, our work provides valuable design insights into the concept of *hierarchical generation* for future LLM-driven code assistants.

REFERENCES

[1] Nahla J Abid, Jonathan I Maletic, and Bonita Sharif. 2019. Using developer eye movements to externalize the mental model used in code summarization tasks. In *Proceedings of the 11th ACM Symposium on Eye Tracking Research & Applications*. 1–9.

[2] Mathieu Acher, José Galindo Duarte, and Jean-Marc Jézéquel. 2023. On Programming Variability with Large Language Model-based Assistant. In *Proceedings of the 27th ACM International Systems and Software Product Line Conference-Volume A*. 8–14.

[3] Naser Al Madi. 2022. How readable is model-generated code? examining readability and visual inspection of GitHub copilot. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.

[4] Nedhal A Al-Saiyd. 2017. Source code comprehension analysis in software maintenance. In *2017 2nd International Conference on Computer and Communication Systems (ICCCS)*. IEEE, 1–5.

[5] Saleema Amershi, Dan Weld, Mihaela Vorvoreanu, Adam Fourney, Besmira Nushi, Penny Collisson, Jina Suh, Shamsi Iqbal, Paul N Bennett, Kori Inkpen, et al. 2019. Guidelines for human-AI interaction. In *Proceedings of the 2019 chi conference on human factors in computing systems*. 1–13.

[6] VenuGopal Balijepally, Sridhar Nerur, and RadhaKanta Mahapatra. 2012. Effect of task mental models on software developer's performance: An experimental investigation. In *2012 45th Hawaii International Conference on System Sciences*. IEEE, 5442–5451.

[7] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989* (2016).

[8] Moritz Balz, Michael Striewe, and Michael Goedicke. 2010. Continuous maintenance of multiple abstraction levels in program code. In *International Workshop on Future Trends of Model-Driven Development*, Vol. 2. SCITEPRESS, 68–79.

[9] Aaron Bangor, Philip T. Kortum, and James T. Miller. 2008. An Empirical Evaluation of the System Usability Scale. *International Journal of Human–Computer Interaction* 24, 6 (2008), 574–594. https://doi.org/10.1080/10447310802205776 arXiv:https://doi.org/10.1080/10447310802205776

[10] 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 (2023), 85–111.

[11] Ashish Bastola, Hao Wang, Judsen Hembree, Pooja Yadav, Nathan McNeese, and Abolfazl Razi. 2023. LLM-based Smart Reply (LSR): Enhancing Collaborative Performance with ChatGPT-mediated Smart Reply System (ACM)(Draft) LLM-based Smart Reply (LSR): Enhancing Collaborative Performance with ChatGPT-mediated Smart Reply System. *arXiv preprint arXiv:2306.11980* (2023).

[12] Gregor Betz, Kyle Richardson, and Christian Voigt. 2021. Thinking aloud: Dynamic context generation improves zero-shot reasoning performance of GPT-2. (March 2021). arXiv:2103.13033 [cs.CL]

[13] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2023. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1946–1969.

[14] Susanne Bødker. 2015. Third-wave HCI, 10 years later—participation and sharing. *interactions* 22, 5 (2015), 24–31.

[15] Nick C Bradley, Thomas Fritz, and Reid Holmes. 2018. Context-aware conversational developer assistants. In *Proceedings of the 40th International Conference on Software Engineering*. 993–1003.

[16] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. 2010. Code Bubbles: A Working Set-Based Interface for Code Understanding and Maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Atlanta, Georgia, USA) *(CHI '10)*. Association for Computing Machinery, New York, NY, USA, 2503–2512. https://doi.org/10.1145/1753326.1753706

[17] Virginia Braun and Victoria Clarke. 2012. *Thematic analysis.* American Psychological Association.

[18] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[19] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712* (2023).

[20] Yuzhe Cai, Shaoguang Mao, Wenshan Wu, Zehua Wang, Yaobo Liang, Tao Ge, Chenfei Wu, Wang You, Ting Song, Yan Xia, et al. 2023. Low-code LLM: Visual Programming over LLMs. *arXiv preprint arXiv:2304.08103* (2023).

[21] Yining Cao, Jane L E, Zhutian Chen, and Haijun Xia. 2023. DataParticles: Block-Based and Language-Oriented Authoring of Animated Unit Visualizations. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 808, 15 pages. https://doi.org/10.1145/3544548.3581472

[22] John M Carroll and Judith Reitman Olson. 1988. Mental models in human-computer interaction. *Handbook of human-computer interaction* (1988), 45–65.

[23] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397* (2022).

[24] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[25] Justin Cheng, Jaime Teevan, Shamsi T Iqbal, and Michael S Bernstein. 2015. Break it down: A comparison of macro-and microtasks. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. 4061–4064.

[26] Yu Cheng, Jieshan Chen, Qing Huang, Zhenchang Xing, Xiwei Xu, and Qinghua Lu. 2023. Prompt Sapper: A LLM-Empowered Production Tool for Building AI Chains. *arXiv preprint arXiv:2306.12028* (2023).

[27] Cynthia L Corritore and Susan Wiedenbeck. 1991. What do novices learn during program comprehension? *International Journal of Human-Computer Interaction* 3, 2 (1991), 199–222.

[28] Allen Cypher and Daniel Conrad Halbert. 1993. *Watch what I do: programming by demonstration.* MIT press.

[29] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Zhen Ming Jack Jiang. 2023. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software* 203 (2023), 111734.

[30] Linda Dawson. 2013. Cognitive processes in object-oriented requirements engineering practice: analogical reasoning and mental modelling. In *Information Systems Development: Reflections, Challenges and New Directions*. Springer, 115–128.

[31] Françoise Détienne. 2001. *Software design–cognitive aspect.* Springer Science & Business Media.

[32] Alastair Dunsmore and Marc Roper. 2000. A comparative evaluation of program comprehension measures. *The Journal of Systems and Software* 52, 3 (2000), 121–129.

[33] Kasra Ferdowsi, Michael B James, Nadia Polikarpova, Sorin Lerner, et al. 2023. Live Exploration of AI-Generated Programs. *arXiv preprint arXiv:2306.09541* (2023).

[34] Alexander J Fiannaca, Chinmay Kulkarni, Carrie J Cai, and Michael Terry. 2023. Programming without a Programming Language: Challenges and Opportunities for Designing Developer Tools for Prompt Programming. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–7.

[35] Vikki Fix, Susan Wiedenbeck, and Jean Scholtz. 1993. Mental representations of programs by novices and experts. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*. 74–79.

[36] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[37] Nat Friedman. 2021. Introducing GitHub Copilot: your AI pair programmer. https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer/

[38] Github. 2023. Github Copilot, Your AI pair programmer. https://github.com/features/copilot

[39] Thomas R. G. Green and Marian Petre. 1996. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing* 7, 2 (1996), 131–174.

[40] Carsten Gutwenger, Michael Jünger, Karsten Klein, Joachim Kupke, Sebastian Leipert, and Petra Mutzel. 2003. A new approach for visualizing UML class diagrams. In *Proceedings of the 2003 ACM symposium on Software visualization*. 179–188.

[41] Sandra G Hart and Lowell E Staveland. 1988. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. In *Advances in psychology*. Vol. 52. Elsevier, 139–183.

[42] Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-based neural code generation. *arXiv preprint arXiv:1808.10025* (2018).

[43] Ava Heinonen, Bettina Lehtelä, Arto Hellas, and Fabian Fagerholm. 2023. Synthesizing research on programmers' mental models of programs, tasks and concepts—A systematic literature review. *Information and Software Technology* (2023), 107300.

[44] A Z Henley and S D Fleming. 2014. The patchworks code editor: Toward faster navigation with less code arranging and fewer navigation mistakes. In *Proceedings of the sigchi conference on human factors in computing systems*. 2511–2520.

[45] Kristina Höök. 2000. Steps to take before intelligent user interfaces become real. *Interacting with computers* 12, 4 (2000), 409–426.

[46] Di Huang, Ziyuan Nan, Xing Hu, Pengwei Jin, Shaohui Peng, Yuanbo Wen, Rui Zhang, Zidong Du, Qi Guo, Yewen Pu, et al. 2023. ANPL: Compiling Natural Programs with Interactive Decomposition. *arXiv preprint arXiv:2305.18498* (2023).

[47] Edwin L Hutchins, James D Hollan, and Donald A Norman. 1985. Direct manipulation interfaces. *Human–computer interaction* 1, 4 (1985), 311–338.

[48] Ellen Jiang, Edwin Toh, Alejandra Molina, Kristen Olson, Claire Kayacik, Aaron Donsbach, Carrie J Cai, and Michael Terry. 2022. Discovering the syntax and strategies of natural language programming with generative language models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–19.

[49] Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. 2021. Treebert: A tree-based pre-trained model for programming language. In *Uncertainty in Artificial Intelligence*. PMLR, 54–63.

[50] Hyeonsu Kang and Philip J Guo. 2017. Omnicode: A novice-oriented live programming environment with always-on run-time value visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. 737–745.

[51] Siddharth Karamcheti, Dorsa Sadigh, and Percy Liang. 2020. Learning adaptive language interfaces through decomposition. *arXiv preprint arXiv:2010.05190* (2020).

[52] Mary Beth Kery. 2018. Towards scaffolding complex exploratory data science programming practices. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 273–274.

[53] Mary Beth Kery, Amber Horvath, and Brad A Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists.. In *CHI*, Vol. 10. 3025453–3025626.

[54] Mary Beth Kery and Brad A Myers. 2017. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 25–29.

[55] Tae Soo Kim, Yoonjoo Lee, Minsuk Chang, and Juho Kim. 2023. Cells, Generators, and Lenses: Design Framework for Object-Oriented Interaction with Large Language Models. (2023).

[56] Kite. 2014. Free AI Coding Assistant and Code Auto-Complete Plugin. https://www.kite.com/blog/product/kite-is-saying-farewell/

[57] James R. Lewis, Brian S. Utesch, and Deborah E. Maher. 2013. UMUX-LITE: When There's No Time for the SUS. In *Proceedings of the SIGCHI Conference*

on Human Factors in Computing Systems (Paris, France) *(CHI '13)*. Association for Computing Machinery, New York, NY, USA, 2099–2102. https://doi.org/10.1145/2470654.2481287

[58] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[59] Jenny T Liang, Chenyang Yang, and Brad A Myers. 2023. Understanding the Usability of AI Programming Assistants. *arXiv preprint arXiv:2303.17125* (2023).

[60] Henry Lieberman. 2001. *Your wish is my command: Programming by example.* Morgan Kaufmann.

[61] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. 2021. What Makes Good In-Context Examples for GPT-3? *arXiv preprint arXiv:2101.06804* (2021).

[62] Michael Xieyang Liu, Advait Sarkar, Carina Negreanu, Benjamin Zorn, Jack Williams, Neil Toronto, and Andrew D Gordon. 2023. "What It Wants Me To Say": Bridging the Abstraction Gap Between End-User Programmers and Code-Generating Large Language Models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems.* 1–31.

[63] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *Comput. Surveys* 55, 9 (2023), 1–35.

[64] Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. 2022. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (Dublin, Ireland). Association for Computational Linguistics, Stroudsburg, PA, USA.

[65] Ewa Luger and Abigail Sellen. 2016. " Like Having a Really Bad PA" The Gulf between User Expectation and Experience of Conversational Agents. In *Proceedings of the 2016 CHI conference on human factors in computing systems.* 5286–5297.

[66] Pingchuan Ma, Rui Ding, Shuai Wang, Shi Han, and Dongmei Zhang. 2023. Demonstration of InsightPilot: An LLM-Empowered Automated Data Exploration System. *arXiv preprint arXiv:2304.00477* (2023).

[67] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. 2014. On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 4 (2014), 1–37.

[68] Microsoft. 2023. The Editor of the Web. https://microsoft.github.io/monaco-editor

[69] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2022. Reading between the lines: Modeling user behaviour and costs in AI-assisted programming. *arXiv preprint arXiv:2210.14306* (2022).

[70] Raquel Navarro-Prieto and Jose J Canas. 2001. Are visual programming languages better? The role of imagery in program comprehension. *International Journal of Human-Computer Studies* 54, 6 (2001), 799–829.

[71] Nhan Nguyen and Sarah Nadi. 2022. An empirical evaluation of GitHub copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories.* 1–5.

[72] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[73] Donald A Norman. 1994. How might people interact with agents. *Commun. ACM* 37, 7 (1994), 68–71.

[74] OpenAI. 2023. ChatGPT (Feb 13 version) [Large language model]. https://chat.openai.com

[75] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[76] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP).* IEEE, 2339–2356.

[77] Rohith Pudari and Neil A Ernst. 2023. From Copilot to Pilot: Towards AI Supported Software Development. *arXiv preprint arXiv:2303.04142* (2023).

[78] Pyodide. 2023. Pyodide is a Python distribution for the browser and Node.js based on WebAssembly. https://github.com/pyodide/pyodide

[79] Laria Reynolds and Kyle McDonell. 2021. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems.* 1–7.

[80] Nico Ritschel, Felipe Fronchetti, Reid Holmes, Ronald Garcia, and David C Shepherd. 2022. Can guided decomposition help end-users write larger block-based programs? a mobile robot experiment. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 233–258.

[81] Scott P Robertson and Chiung-Chen Yu. 1990. Common cognitive representations of program code across tasks and languages. *International Journal of Man-Machine Studies* 33, 3 (1990), 343–360.

[82] 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. In *Proceedings of the 28th International Conference on Intelligent User Interfaces.* 491–514.

[83] Adam Rule, Aurélien Tabard, and James D Hollan. 2018. Exploration and explanation in computational notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems.* 1–12.

[84] Advait Sarkar, Andrew D Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. 2022. What is it like to program with artificial intelligence? *arXiv preprint arXiv:2208.06213* (2022).

[85] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems* 25 (2012).

[86] Hendrik Strobelt, Albert Webson, Victor Sanh, Benjamin Hoover, Johanna Beyer, Hanspeter Pfister, and Alexander M Rush. 2022. Interactive and visual prompt engineering for ad-hoc task adaptation with large language models. *IEEE transactions on visualization and computer graphics* 29, 1 (2022), 1146–1156.

[87] Sangho Suh, Bryan Min, Srishti Palani, and Haijun Xia. 2023. Sensecape: Enabling Multilevel Exploration and Sensemaking with Large Language Models. *arXiv preprint arXiv:2305.11483* (2023).

[88] Tabnine. 2012. Tabnine - AI assistant that speeds up delivery and keeps your code safe. https://www.tabnine.com/

[89] Ningzhi Tang, Meng Chen, Zheng Ning, Aakash Bansal, Yu Huang, Collin McMillan, and Toby Jia-Jun Li. 2023. An Empirical Study of Developer Behaviors for Validating and Repairing AI-Generated Code. Plateau Workshop.

[90] John W Tukey et al. 1977. *Exploratory data analysis.* Vol. 2. Reading, MA.

[91] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts.* 1–7.

[92] Helena Vasconcelos, Gagan Bansal, Adam Fourney, Q Vera Liao, and Jennifer Wortman Vaughan. 2023. Generation probabilities are not enough: Exploring the effectiveness of uncertainty highlighting in AI-powered code completions. *arXiv preprint arXiv:2302.07248* (2023).

[93] Anneliese von Mayrhauser and A Marie Vans. 1995. Industrial experience with an integrated code comprehension model. *Software Engineering Journal* 10, 5 (1995), 171–182.

[94] Anneliese Von Mayrhauser and A Marie Vans. 1995. Program comprehension during software maintenance and evolution. *Computer* 28, 8 (1995), 44–55.

[95] Justin D Weisz, Michael Muller, Stephanie Houde, John Richards, Steven I Ross, Fernando Martinez, Mayank Agarwal, and Kartik Talamadupula. 2021. Perfection not required? Human-AI partnerships in code translation. In *26th International Conference on Intelligent User Interfaces.* 402–412.

[96] Tongshuang Wu, Ellen Jiang, Aaron Donsbach, Jeff Gray, Alejandra Molina, Michael Terry, and Carrie J Cai. 2022. Promptchainer: Chaining large language model prompts through visual programming. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts.* 1–10.

[97] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. 2022. Ai chains: Transparent and controllable human-ai interaction by chaining large language model prompts. In *Proceedings of the 2022 CHI conference on human factors in computing systems.* 1–22.

[98] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. 2017. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering* 44, 10 (2017), 951–976.

[99] Bingjun Xie, Jia Zhou, Huilin Wang, et al. 2017. How influential are mental models on interaction performance? exploring the gap between users' and designers' mental models through a new quantitative method. *Advances in Human-Computer Interaction* 2017 (2017).

[100] Frank F Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-ide code generation from natural language: Promise and challenges. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2022), 1–47.

[101] Nong Ye and Gavriel Salvendy. 1996. Expert-novice knowledge of computer programming at different levels of abstraction. *Ergonomics* 39, 3 (1996), 461–481.

[102] Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696* (2017).

[103] Young Seok Yoon and Brad A Myers. 2014. A longitudinal study of programmers' backtracking. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC).* IEEE, 101–108.

[104] Yichi Zhang and Joyce Chai. 2021. Hierarchical task learning from language instructions with unified transformers and self-monitoring. *arXiv preprint arXiv:2106.03427* (2021).

[105] Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming.* 21–29.

A QUESTIONNAIRE

Below we list the questions we used in the evaluation study questionnaire.

A.1 UMUX-LITE

1. This system's capabilities meet my requirements.
2. This system is easy to use.

A.2 NASA-TLX

1. How mentally demanding was the task?
2. How physically demanding was the task?
3. How hurried or rushed was the pace of the task?
4. How successful were you in accomplishing what you were asked to do?
5. How hard did you have to work to accomplish your level of performance?
6. How insecure, discouraged, irritated, stressed, and annoyed were you?

A.3 Self-Defined Likert Scale Items

1. The system reduces the need for cognitive switching between editing and validation.
2. I had a good understanding of why the system generates such results.
3. I could steer the system toward the task goal.
4. The system helps construct a mental model for solving the task.
5. The system helps scaffold my intents to generate desired code.
6. I'm satisfied with the overall suggestions from the system.
7. I am confident that the system generated the correct code.
8. I understand what my program is about, and how it works.
9. The system helps me verify the generated results.