CoLadder: Manipulating Code Generation via Multi-Level Blocks

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

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

Sangho Suh
University of Toronto
Ontario, Canada
sangho.suh@utoronto.ca

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

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

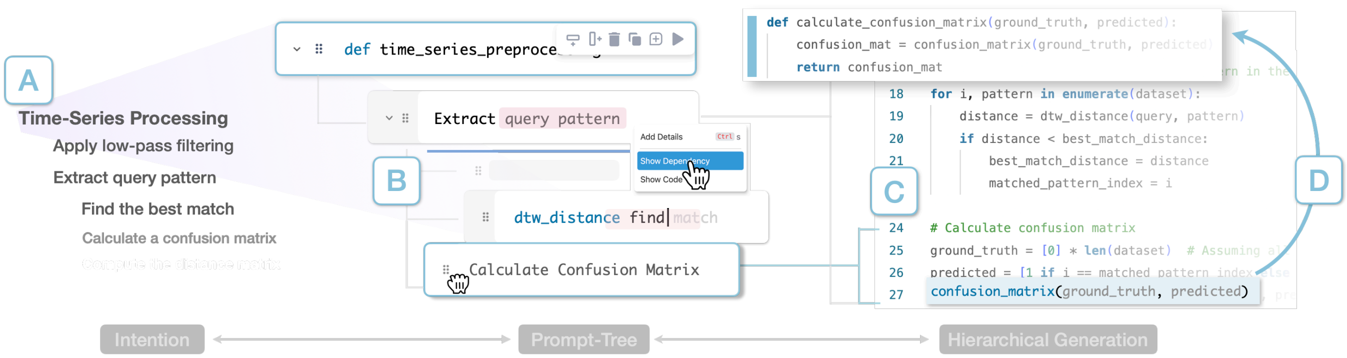

Figure 1: Our system enables programmers to decompose tasks in alignment with their mental models using LLM-driven code assistants. It features a tree-based prompt editor that allows programmers to hierarchically externalize their intent (A) into modular prompt blocks (B). This prompt tree structure generates code, with each block corresponding to a code segment (C). Programmers can manipulate the code through block-based operations; for example, dragging a code block upwards to elevate it to a global function, which refines the code structure accordingly (D).

ABSTRACT

This paper adopted an iterative design process to gain insights into programmers' strategies when using LLMs for programming. We proposed *CoLadder*, a novel system that supports programmers by facilitating hierarchical task decomposition, direct code segment manipulation, and result evaluation during prompt authoring. A user study with 12 experienced programmers showed that *CoLadder* is effective in helping programmers externalize their problem-solving intentions flexibly, improving their ability to evaluate and modify code across various abstraction levels, from their task's goal to final code implementation.

CCS CONCEPTS

• **Human-centered computing → Natural language interfaces**; **User interface programming**.

UIST '24, October 13–16, 2024, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-0628-8/24/10
https://doi.org/10.1145/3654777.3676357

KEYWORDS

Code Generation, Programming Interface, Dynamic Abstraction

ACM Reference Format:
Ryan Yen, Jiawen Stefanie Zhu, Sangho Suh, Haijun Xia, and Jian Zhao. 2024. CoLadder: Manipulating Code Generation via Multi-Level Blocks. In *The 37th Annual ACM Symposium on User Interface Software and Technology (UIST '24), October 13–16, 2024, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3654777.3676357

1 INTRODUCTION

The advent of Large Language Model (LLM)-driven code assistants has changed programmers' approaches to solving programming problems [38, 52, 68]. Instead of detailing their intentions of what to achieve and how to achieve it, programmers can declaratively express their goals to the LLM through natural language prompts. Nonetheless, this approach inadvertently removes the need for programmers to construct step-by-step intentions for solving programming problems [4, 26]. Without constructing these intentions, programmers can become over-reliant on the assistant, as the programmers may not fully understand the problems and solutions generated by LLM [47, 52, 62, 68]. This exacerbates the challenge for programmers when refining the generated code since the initial generation often does not meet their requirements [6, 13, 68, 78].

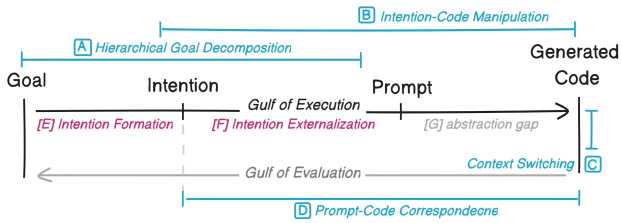

Figure 2: The cognitive processes interact with LLM-driven systems, where A-D refers to our system's design guidelines.

Despite the importance of constructing clear intentions for solving programming problems before articulating prompts for generation, there is a noticeable gap in research addressing this complex multi-stage intention formation process in the context of code generation [47, 66]. Our paper thus aims to explore designs that facilitate this iterative intention formation process when programmers tackle programming tasks with LLMs. We focus on supporting the process of decomposing goals into specific intentions, articulating prompts from those intentions, and refining intentions after evaluating the generated code.

We conducted a formative study with six experienced programmers who regularly use LLM-driven code assistants to understand how to support programmers in constructing, externalizing, and refining their intentions while programming with LLM. Our findings indicate that programmers require structural support to form and externalize their intentions and the ability to directly control and manipulate generated code segments to refine their intentions. Hence, we propose *CoLadder*, a novel system designed to assist programmers in iteratively forming intentions for solving programming problems with LLM-driven code assistants. The system scaffolds programmers in forming and externalizing their intentions hierarchically into a multi-level prompt tree. Each prompt in the tree is a modular *block* linked to code segments, enabling code refinement by direct manipulation (Fig. 1 D). Furthermore, *CoLadder* provides several visual cues for programmers to match the prompt tree with the generated code structure.

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 form, externalize, and iteratively refine their intentions. Refining corresponding generated code by directly manipulating prompt blocks provides programmers with control over the translation of their intentions into code. With such supported scaffolding, our system can better support programmers in evaluating the generated code and externalizing their intentions for refining the generation. In summary, our contribution is threefold:

- Understand programmers' strategies of refining intentions for solving programming problems with LLM-driven code assistants.
- An interface enables programmers to manipulate code generation across multiple levels of abstraction.
- A user study demonstrating improved usability by enhancing controllability and facilitating result evaluation and refinement.

2 GOAL, INTENTION, PROMPT, AND GENERATED CODE

In the pursuit of the *goal* of a programming task, programmers must cultivate clear *intentions* [32, 33, 41]. These intentions are articulated from the comprehension of what the program is intended to achieve (declarative knowledge) and the procedures involved in achieving it (procedural knowledge) [14, 26]. This *intention formation process* involves the dissection of the overarching goal into smaller, manageable subgoals and then the *externalization* of the program's elements with varying depths and extents (Fig. 2 E) [26, 31, 37, 60, 64]. Externalizing layered intentions into actions is crucial (Fig. 2 F), particularly in LLM-driven systems where the *gulf of execution* can become *fuzzy* due to LLMs' ability to generate results from various formats of NL prompts. These prompts then carry the programmer's structured intentions into the LLM generation process. Prior research strived to bridge the *abstraction gap* (Fig. 2 G)—the disparity between the human intention behind the prompt and the code generated by LLMs [47, 62]. In contrast, our research emphasizes supporting intention formation (Fig. 2 A) and its subsequent externalization process, which empowers programmers to precisely guide the code generation with controllability (Fig. 2 B). Moreover, our work incorporates features that reduce the *gulf of evaluation* [54] (Fig. 2 D), enhancing the programmer's ability to perceive, interpret, and evaluate the LLM's output without excessive context switching from prompt authoring and code evaluation (Fig. 2 C).

3 BACKGROUND AND RELATED WORK

We reviewed related research on challenges in programmer-LLM interaction, and existing proposed solutions.

3.1 Challenges in Programmer-LLM Interaction

Previous research on understanding how programmers interact with LLMs-based code assistants pointed out that programmers now need to dedicate considerable time to evaluating AI-generated code suggestions [6, 52]. Excessive evaluation needs can lead to several issues [61, 69, 73]. Programmers are often intimidated by the seemingly overwhelming effort required for code validation and bypass the evaluation step. This causes problems like overreliance on generated suggestions [6, 13, 78], and loss of control over their programs [68], which then introduces challenges during code modification [1, 12, 22]. Programmers are also taxed with the extra cognitive load of switching between programming and debugging tasks [7, 23, 49, 68].

Sarkar et al. [62] observed that programmers often engage in iterative evaluation 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* [36]. 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 [47, 77]. Our study extends the focus from abstraction matching from prompt-code to goal-code, considering the importance of the intention formation process.

3.2 Improving LLM-based Code Generation

Compared to technical approaches like prompt engineering and few-shot learning, several design solutions and systems have been proposed to enhance interaction with LLM-driven code assistants. These strategies encompass various techniques, such as introducing new programming languages [8, 35], automating prompt rewording [24, 74], employing programming by demonstration techniques [16, 46], and supporting task breakdowns [59, 77]. 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"* [62]. 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 [47, 48, 61, 69].

Noticing this issue, prior research has proposed several approaches. Liu et al. introduced *Grounded Abstraction Matching* [47] that provides a decomposed code example that users can modify and submit to the LLM as instructions, assisting programmers with unclear intentions and reducing abstraction matching problems. AI Chains enhance programmer control and feedback by breaking problems into sub-tasks [77]. Each sub-task corresponds to a specific step with an NL prompt, and results from previous steps inform prompts for subsequent tasks. This chaining method increases success rates when using the same model on multiple tasks [76, 77].

While previous approaches that rely on task breakdowns help programmers bridge their intent to code, they do not emphasize the iterative formation and refinement of intentions. Additionally, they focus on local prompt-code correspondence without examining the overall structure, especially from task to code. *CoLadder* builds on task decomposition and offers flexibility and control for programmers to articulate and manipulate prompts and corresponding code blocks at different abstraction levels using a tree-based editor.

4 COLADDER: DESIGN PROCESS & GOALS

We conducted an iterative user-centered design to create *CoLadder*, an interface to help programmers decompose tasks based on their intentions and generate code accordingly. 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 5); 3) *Deploy & Evaluate*—a user study to evaluate how programmers interact with *CoLadder* and their perceived usefulness (Section 6). In this section, we describe the first stage of our design process and report the obtained strategies and design goals that guided the design and development of *CoLadder* (Table. 1).

4.1 Interview Study

We recruited six participants (5 males, 1 female; ages $25-27, M = 25.8, SD = 0.98$) through purposive sampling [21]. In our recruitment process, we sought participants experienced in both programming and the use of LLM-driven code generation tools. Eligibility screening involved a pre-test survey that assessed participants' programming experience on a 5-point scale [1: very inexperienced; 5: very experienced], years of programming experience, and self-reported familiarity with LLM-driven code generation tools. All recruited participants had more than five years of programming experience ($M = 6.67, SD = 1.75$ years) and were familiar with programming (score $M = 4.33, SD = 0.52$), familiar with LLM-code generation tools (score $M = 4.5, SD = 0.55$), and regularly used the LLM-code generation tools ($M = 8, SD = 2.56$ times/week). Detailed can be seen in Appendix B

Participants consented and received $20 for 45-minute study sessions. Before the study, we asked each participant to share at least three recent examples of using ChatGPT [56] for generating code to reflect on their use of LLM tools. During the study, we interviewed participants to assess their challenges in forming and externalizing intentions, translating them into code, and the strategies they used. All interviews were audio-recorded and transcribed. We analyzed the interviews using thematic analysis with both inductive and deductive approaches [9].

4.2 Interview Results

Overall, participants used LLM-code assistants across various programming languages (e.g., Python, JavaScript, and Bash) for a wide range of tasks, such as unfamiliar code generation, algorithm implementation, and code refactoring (Appendix B).

4.2.1 ***Structure Tasks and Prompts Hierarchically***. We observed that the formation of intentions for participants to solve programming tasks when prompting LLM-code assistants involves two key facets. First, programmers need to form clear intentions 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, the linear representation of prompts hinders participants' ability to externalize their intentions and understand their own prompts after composing them (**C1**). Every participant adopted a similar strategy to alleviate the cognitive load when forming intentions — by breaking down tasks into smaller subtasks (**S1**). Most (5 out of 6 participants, 5/6 henceforth) participants structured their tasks hierarchically to externalize their intentions and used indented bullet points to structure the prompt (**S2**).

4.2.2 ***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 evaluation by generating an entire codebase based on the whole prompt. 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

Table 1: The summary of challenges and strategies reported in Section 4.2 and the resulting design goals (Sections 4.3) and features in *CoLadder* (Sections 5)

Challenges	Strategies	Design Goals	Features
C1: Unstructured Prompt to Externalize Intention	**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 Prompts 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**: Evaluate Results during Prompt Authoring	**DG3**: Enabling Code Evaluation during Prompt Authoring	List Steps
			Auto-Completion
			Recommendation
C4: Unclear Correspondence from Prompt to Code	**S8**: Add Code to Prompt	**DG4**: Enhancing Prompt-Code Correspondence	Corresponding Code Highlight
	S9: Evaluated by Comments		Semantic Highlight

specific prompts rather than generating lengthy code encompassing all prompts. Participants (4/6) also reported an alternative strategy, where they generated self-contained code segments independently, and then merged them into the existing codebase (**S5**). 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.

4.2.3 *Evaluate Generated Code through Prompts*.

All participants expressed frustration with current tools that constantly suggest results, leading to continuous switching between prompt authoring and code evaluation (**C3**). They noted that although this context switching may seem trivial, it significantly disrupts their overall programming flow. We also 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*.

4.2.4 *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 evaluation by making it easier to locate keywords (**S8**). Some participants (2/6) preferred rewrite the whole segment of code, rather than attempt to edit and debug it. Participants usually valued the generated comments (in NL) and used them as anchor points to match segments of prompts to code (**S9**).

4.3 Design Guidelines for Supporting Programmers' Strategies

To support the reported strategies (**S1-S9**) programmers leveraged to overcome challenges they encountered (**C1-C4**), we formulated four design guidelines (**DGs**). The main design goal is to offer *hierarchical generation*, enabling programmers to form and externalize abstract intentions into generated code while ensuring alignment.

***DG1: Offering Hierarchical Prompt Structure*.** The lack of structured prompts hinders programmers from forming and externalizing the intentions for solving programming tasks (**C1**). The system should support prompt decomposition (**S1**) with a hierarchical representation of the prompt structure that externalizes the task structure programmers possess in mind (**S2**). Participants adopt a more flexible prompt structure that varies based on the task's nature and complexity. Hence, the system should enable participants to define abstraction levels for externalizing their intentions.

***DG2: Direct Manipulation of Prompt for Code Modification*.** This issue of control loss (**C2**) highlights the need to provide programmers with control over the prompt authoring process. Such support should include the ability to easily identify, and select the range of modifications (**S6**), insert new prompts (**S4**), and reorganize the structure of prompts (**S5**). The system must allow programmers to make the necessary changes without inadvertently modifying unrelated code segments. The system should also generate results incrementally (**S3**) instead of generating entire code simultaneously.

***DG3: Enabling Code Evaluation during Prompt Authoring*.** During the prompt iterations, programmers often experience cognitive overload due to the disruptive context switching between code evaluation and prompt authoring (**C3**). Findings highlight the need to help programmers evaluate 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: Enhancing Prompt-Code Correspondence for Evaluation*.** 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.

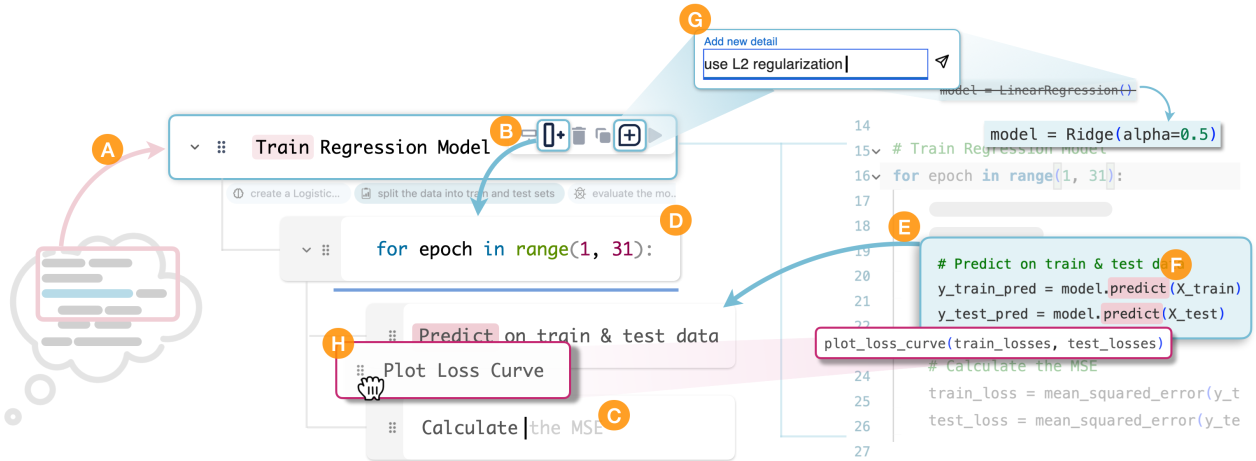

Figure 3: An example workflow of using *CoLadder*. A user creates a high-level prompt (A) based on their intentions for the programming task. Subsequently, they add a sub-task indented underneath (B), incorporating code syntax within the prompt (D). The *List Steps* feature is employed to summarize the generated code into prompts (E). Following evaluation, 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).

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.

4.4 Usage Scenario

Casey is a data scientist who wants to build a regression model on a wine-quality dataset. While being experienced in Python, 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. 3 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. 3 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. 3 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. 3 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. 3 E) assisting her in evaluating the alignment of the intent-code.

Navigating and Evaluating through Multi-level Prompts. Casey then navigates through the hierarchical structure with up/down arrow keys and evaluates highlighted code segments corresponding to the specific prompt block. The **[Semantic Highlight]** feature helps her correlate phrases in her prompts with the code segments (Fig. 3 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. 3 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. 3 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.

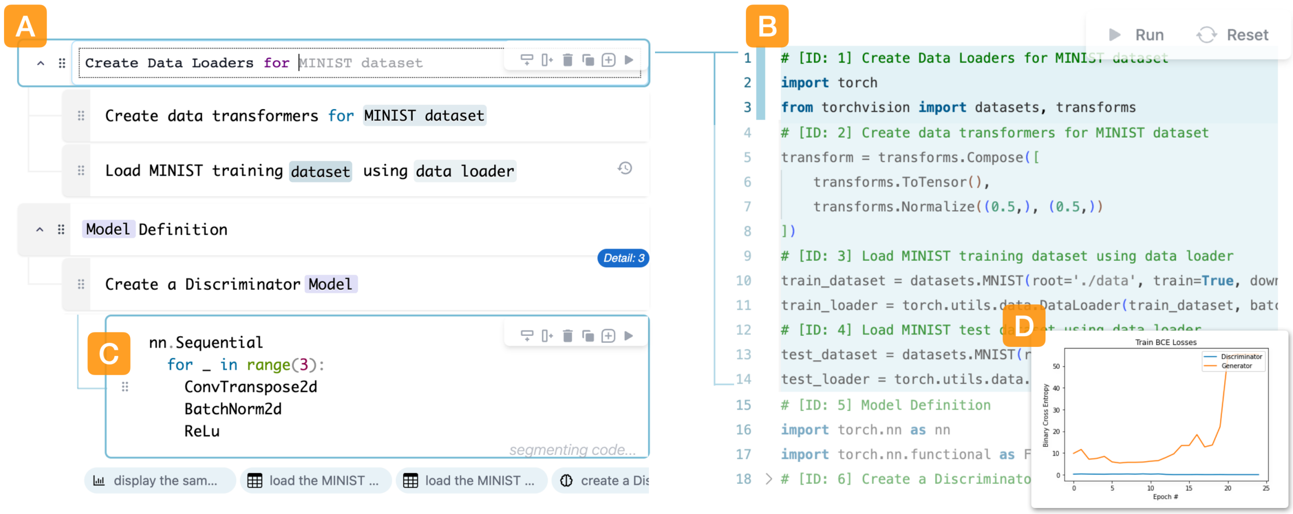

Figure 4: ***CoLadder*** **comprises four key components: (A) the prompt tree editor, allowing programmers to decompose their intent into smaller prompt blocks; (B) the code editor, facilitating code evaluation and editing; (C) the prompt block, enabling programmers to compose prompts in *mixed mode*, incorporating both code and natural language; and (D) the execution result panel, which displays the execution result and any associated error messages.**

5 COLADDER

CoLadder consists of two main UI components: 1) The **prompt tree editor** (Fig. 4 A) allows the programmer to externalize their intentions by decomposing the programming task into smaller prompt blocks (Fig. 4 C); 2) The **code editor** (Fig. 4 B) allows the programmer to evaluate 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. 4 D).

We elaborate on *CoLadder*'s functionalities and design based on **DG1-4**, incorporating insights from six experienced programmers who participated in interview studies and cognitive walkthrough experiments during the iterative design process.

5.1 From Task Structure to Code Structure

To assist programmers in structuring their prompts hierarchically to externalize their intentions (**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**).

5.1.1 ***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 construct prompts flexibly that align with their intentions. Based on expert walkthrough suggestions, we implemented a foldable prompt tree, aligning with the foldable code editor. This is useful for longer programs, eliminating the need for constant scrolling.

5.1.2 ***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. 4 C). Programmers have the flexibility to input both NL and code syntax, aligning with their preference for occasionally using code syntax in the prompt to express their thoughts. For instance, participants often opt for statements like *"for i in..."* rather than NL expressions such as *"iterate through the..."*. Each block functions as a miniature code editor, with semantic highlighting of code syntax and prompts. Several participants (4/6) wanted prompt block revision history to aid in recalling the reasoning behind specific prompts. *CoLadder* documents prompt block iterations, visualizing differences in prompts and generated code, enabling programmers to efficiently navigate and recover specific iterations as required (Fig. 5 B).

5.1.3 ***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. 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 adheres to a

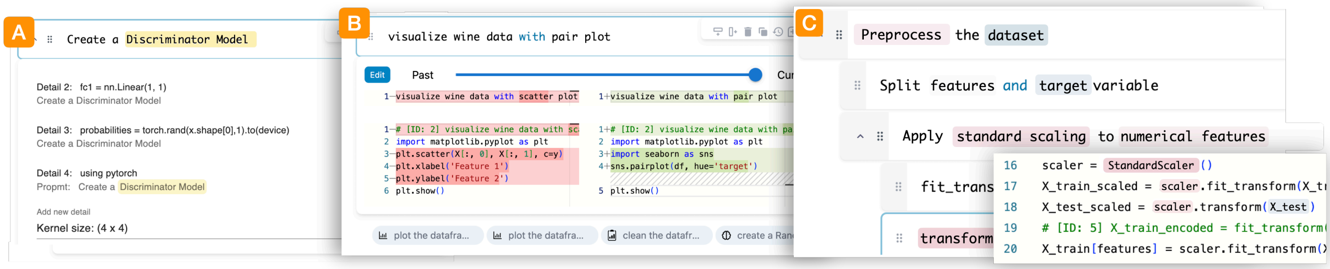

Figure 5: Supplement View (A): Programmers can add additional details to their prompts via a conversational UI; History's Diff View (B): programmers can observe types of changes, iterations of prompts, and their corresponding generated code; Show Dependency (C): Users can select a phrase within the prompt, and the system will highlight correlated phrases throughout the tree structure and code segments. Colors are used to represent the entity type, while opacity indicates the correlation score.

hierarchical structure, so when programmers edit a parent block containing multiple child blocks, the changes apply uniformly to all code segments within those child blocks, ensuring consistency across related sections.

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.
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;
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;
6. **[Supplement]** adds 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 LLM but not necessarily by the programmer to understand the program. Once a programmer submits supplementary information, it will appear as a badge in the top-right corner of the prompt block, accessible by expanding it.

5.2 Evaluate 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 evaluation.

5.2.1 ***List Steps at Lower-Level.*** To help programmers evaluate the generated code effectively, we support the *List Steps* operation The *List Steps* operation semantically segments the current generated code and offers a step-by-step summarization of these code segments. Each step is displayed to users as new sub-prompt blocks, serving as scaffolding to help programmers comprehend the lower-level details of the generated code. For example, if the generated code for current prompt block is about retrieving and processing data from an API, the *List Steps* feature might break down the code into parsing the response and processing and storing the data. These steps are then displayed as individual sub-prompts underneath current prompt block.

5.2.2 ***Auto-completion.*** Participants in the formative study value the in-line auto-completion and view it as a step to evaluate 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.

5.2.3 ***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 step-by-step and helps them evaluate if *CoLadder* correctly understand their intent by recommending the appropriate next steps.

5.3 Facilitating Prompt-Code Correspondence

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

5.3.1 ***Showing Corresponding Code.*** Participants found that evaluating individual code segments was often sufficient. In response, *CoLadder*'s code editor view highlights only relevant code when programmers select the corresponding prompt block, folding other code segments. Different code editor glyph's decorations 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.

5.3.2 ***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, highlight NL phrases to help understand dependencies across prompt blocks at different levels. This is beneficial when

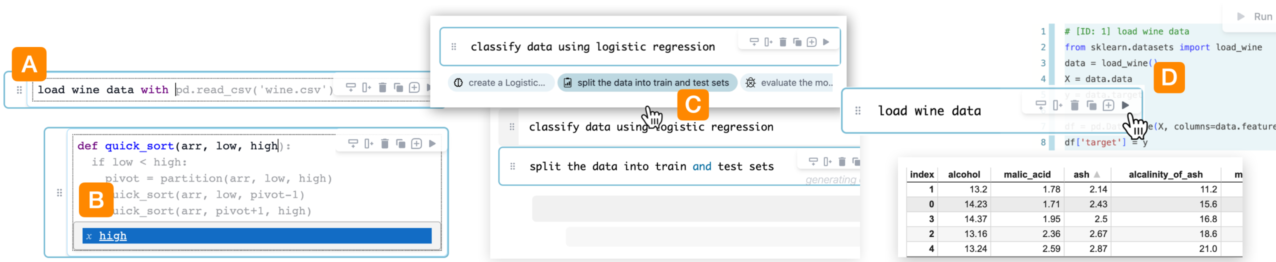

Figure 6: (A) Auto-completion that could be completed in the format of natural language and code syntax; (B) Auto-completion based on the variable name used before; (C) Recommendation feature that suggests the next step based on the prompt tree structure; (D) Live execution showing the interim results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

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 colors, and opacity determined by the relevance score (Fig.5 C). The relevant code segment is highlighted in the code editor for easy identification by the programmer via corresponding phrases from the prompt.

5.3.3 ***Keyboard Navigation.*** *CoLadder* offers keyboard shortcuts that enable programmers to access features and effectively navigate across blocks. The arrow keys (↓/↑) can move through prompt blocks at various levels with the highlighted corresponding code segments in the code editor. 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.

5.4 System Implementation

CoLadder is built on the Next.js framework, enabling server-side rendering for API calls, including the OpenAI GPT-4 API [57] for hierarchical code generation. The user interface incorporates the Monaco Editor [51], providing an intuitive coding experience in both prompt blocks and the main code editor view. To execute and compile results from LLMs, *CoLadder* utilizes Pyodide [58], a potent Python web compiler. Logging is managed through Firebase's Real-Time Database, categorizing interactions and responses by unique user and condition IDs. The entire *CoLadder* platform is built and deployed on Vercel, accessible through a public domain URL.

5.4.1 Prompting Techniques. We structured the prompt tree for code generation into a text-based format with indices and depths for each prompt block [40]. Few-shot examples from the interview study aided the model's in-context learning [11]. An output parser organizes LLM responses into a tree format with unique indexes, prompts, and code. Using the Chain-of-Thought technique [44, 72] via LangChain [42], the LLM generates intermediate reasoning steps in natural language, ensuring logical consistency. The parsed prompt tree, along with specific templates, provides context for block-based operations, enabling targeted code generation (Appendix A). We tested these templates using OpenAI's GPT-4.

5.4.2 Features and Block-based Operations. In the [Add] operation, the LLM adopts a bottom-up approach, integrating child nodes' code with parent nodes. For [Edit] operations, it generates code based on specific prompt blocks, using iteration history as context to discern programmer intent. *CoLadder* automatically records this history during edits. Error prevention mechanisms are applied post-generation to ensure consistent changes (e.g., variable adjustments). A sequential chain uses prior outputs as inputs for subsequent actions, managed with the Myers diff algorithm. When code is edited, the corresponding prompt blocks are updated to reflect these modifications. The Semantic Highlighting feature segments utterances and pairs them with code segments, prompting LLMs to assess similarity. A relevance score is calculated using cosine similarity of text embeddings and categorized with named entity recognition (Appendix A).

6 EVALUATION

We conducted a within-subjects study with 12 participants (7 males, 5 females; ages 23-36, M = 26, SD = 3.54) through purposive sampling [21] via the university mailing list. They were experienced programmers with Python proficiency scores of 4 or higher on a 1 to 5 scale [68, 78], and familiar with LLM-code assistants, particularly with GPT-4. Tasks were selected based on time (12-15 minutes), question type (specific actions), and complexity (prevent complete solution generation). Inspired by previous studies [68, 78], we finalized two tasks each in Machine Learning and Data Visualization categories (Table 3). Tasks featured indirect, picture-based descriptions to encourage independent planning.

We implemented a *Baseline* web-based code editor for comparison with *CoLadder*, both powered by GPT-4. *Baseline* is a web-based code editor that generates code based on inline comments, similar to GitHub Copilot [27]. In *Baseline*, code suggestions are automatically generated and appear as grey text after the cursor position when users pause typing. Users can choose to accept these suggestions by pressing the tab key or ignore them and keep typing.

Each participant completed tasks using both *CoLadder* and *Baseline*. Sessions lasted 60-75 minutes with $30 compensation, including a tutorial, practice session, and task completion. Each participant was allocated to one of two distinct task categories and completed two Python programming tasks based on their assigned category, one using *CoLadder* and the other using *Baseline*. Participants were assigned to tasks and categories using a Latin square design and we counterbalanced all task-category combinations.

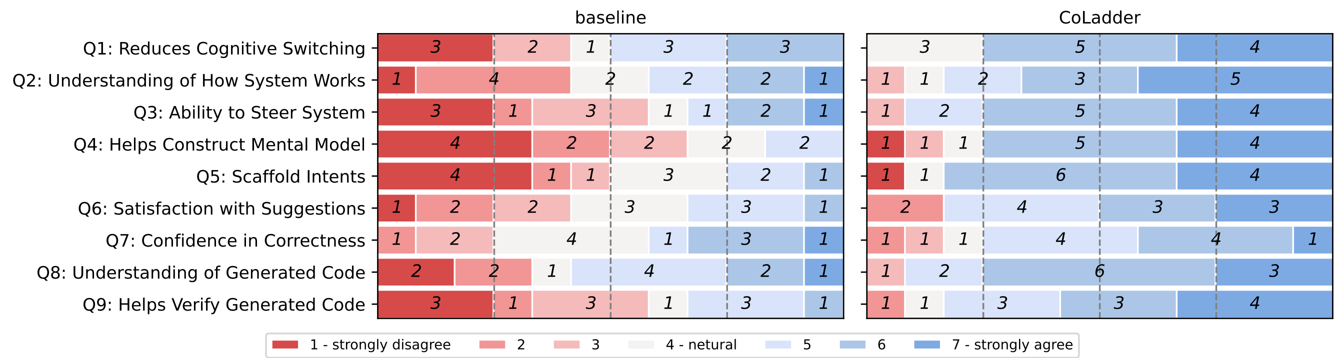

Figure 7: User perception of utility of *Baseline* and *CoLadder*, measured on self-defined 7-point Likert scales (Appendix C.3.3). Dots represent the mean differences of our system compared to the Baseline. Bars indicate the 95% CI calculated using the studentized bootstrap method.

Participants determined the completion of the task based on their satisfaction and judgment. Participants received incentives ($5) if they correctly met all the criteria specified for the tasks.

Participants rated usability and utility after each task using UMUX-LITE [43] and NASA-TLX scales [30] (Appendix C.3.1). Think-aloud data and interviews were transcribed and analyzed through reflexive thematic analysis [10]. Our approach combined inductive and deductive methods to identify codes and themes. We categorized prompts into procedural, declarative, and mixed types across layers and compared survey responses using the Wilcoxon signed-rank test given the ordinal nature of Likert-scale responses.

7 FINDINGS

We present a detailed qualitative analysis and system log data corresponding to the four **D**esign **G**uidelines and **R**esearch **Q**uestions.

7.1 General Impression

7.1.1 Self-Perceived Task Completion and Completion Time. Based on participants' self-evaluation, similar numbers of participants completed the tasks in two conditions (7/12 for *CoLadder* and 6/12 for *Baseline*). 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). The detailed time distribution in Figure 8 reveals that two participants spent less than 7 minutes in *Baseline* condition. This result does not come as a surprise to us, as our goal was to encourage programmers to allocate more time to planning and articulating their intentions and prompts. The qualitative insights from the recall test will discuss the reasons for this outcome and explain why this is not a disadvantage for *Baseline* (Section 7.4.2).

7.1.2 Task Correctness. We executed participants' codes after the study to validate if they correctly met all the specified criteria. With *Baseline*, a higher proportion of tasks (50.0%) remained incomplete, while with *CoLadder*, this percentage was slightly lower (41.67%). The *CoLadder* condition showed a higher rate of tasks that were both completed and correct (50.0%) compared to the *Baseline* condition (25.0%). Notably, more tasks were completed but found to be incorrect using *Baseline* (25.0%) compared to the *CoLadder* condition (8.33%).

7.1.3 Satisfaction and Confidence. Compared to *Baseline*, participants found that when using *CoLadder*, they were more satisfied with the suggestions of the system (Q_6: Mdn_C=6 vs. Mdn_B=4, p=.028, r=.35) . However, while there was an increase in the median confidence level regarding the correctness of the system-generated code, this increase was not statistically significant compared to the baseline condition (Q_7: Mdn_C=5 vs. Mdn_B=4, p=.58, r=.16) . These results suggest that *CoLadder* had no significant impact on participants' perception of the model's accuracy, but it did lead to generated results that were more aligned with their intentions.

7.1.4 Usability and Perceived Cognitive Load. We computed the SUS scores based on the UMUX-LITE. Both our system and the *Baseline* have sufficiently reasonable usability scores, with an average of (Mdn = 90.61) and of (Mdn = 68.94) for the *Baseline*. The overall perceived workload, obtained by averaging all six raw NASA-TLX scores (with the "Performance" measure inverted), did not show a significant difference between *CoLadder* and *Baseline* (Mdn $= 2.33 vs. 3.75, p = .11$).

7.2 Intention Formation and Externalization

Participants felt the prompt tree structure of *CoLadder* significantly helped construct their intentions for solving the programming task compared to *Baseline* (Q_4: Mdn_C=6 vs. Mdn_B=2.5, p=.008, r=.76) .

7.2.1 Various Prompt Types are Used in Different Layers. Every participant constructed a prompt tree with at least 2 layers ($Mdn = 4, SD = 1.17$ 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."* Figure 9 shows that the *declarative* type of prompt blocks (i.e., what tasks to be done) decrease as the number of layers increases. This suggests participants' preference for specifying declarative knowledge in the first three layers. Conversely, the *procedural* type (i.e., the how-to of the task) appears less frequent at first but exhibits an increase in the third

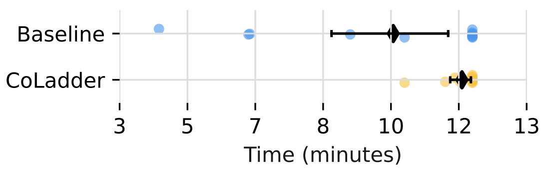

Figure 8: Distribution of time spent on tasks for each participant in both conditions.

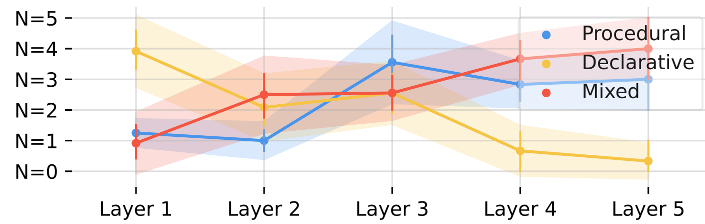

Figure 9: Frequency distribution of distinct block types across five layers in the prompt tree structure, with shaded areas representing standard deviation and error bars indicating 95% confidence intervals.

layer, indicating an increase in procedure-oriented prompts as the layers progress.

7.2.2 Strategies for Structuring Prompt Tree. Most participants (11/12) organized the mapping of their prompts to the structure of the task (in-breadth) through indentation (e.g., task-to-subtask), while also aligning the order of the prompts (in-depth) 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:

1. **Breadth-First Structure**: Participants (4/12) outlined all primary tasks first and subsequently delved into the finer details by adding sub-tasks (or *Supplements*). Before moving to subtasks, they utilized the *List Steps* feature to evaluate how the code is being implemented. This result is similar to the top-down decomposition method leverage for code comprehension [26].
2. **Depth-First Structure**: Participants (7/12) addressed all subtasks within a main task before moving on to the next primary task. Participants with a well-defined intention 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. This approach is similar to the stepwise refinement in program design [26].

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 intention 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*. While two participants constructed a layered prompt as discovered in the formative study, they did not use it as the prompt for code generation. Instead, they utilized it to establish the overall context and then proceeded to refine their prompts in a more detailed manner during the code-writing process.

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" - P11.*

7.3 Controlled Scaffolding from Intention to Code

After externalizing their intentions through the prompt tree, participants proceeded to evaluate the results and made edits to prompts and code as needed. Participants found *CoLadder* to be more helpful in scaffolding their intentions to generate the desired code (Q_5: Mdn_C=6 vs. Mdn_B=3.5, p=.007, r=.77) compared to the *Baseline*.

7.3.1 Editing Prompts before Code. We observed that participants typically began code evaluation after drafting the prompt tree structure. P6 explained, *"The generated code will not be accurate unless I provide details [e.g., by adding child blocks, Supplement operation]."* Participants typically edit prompts first instead of modifying the generated code when using *CoLadder* (Fig. 10). P9 explained that the reason for adjusting prompts in *CoLadder* is *"not because it generates the wrong code, but to adjust my approach for solving the task [intention]."* Analysis of the log data (Fig. 10) revealed that participants made significant code edits in the latter third of their work when using *CoLadder*. Participants mentioned that they felt it was more like *"maintaining a document" - P4* and *"fully noting my thought process" - P1* rather than engaging in prompt engineering, as was the case in the *Baseline* condition.

7.3.2 Block-based Operations Help in Prompt Editing. As in Figure 11 Left, participants in the *Baseline* condition tended to manually edit the code significantly more than in the *CoLadder* condition (Mdn=8.0 vs. Mdn=53.0, p=.002, r=0.9). This finding aligns with the observation that they only edited the code towards the end when using *CoLadder*. However, there is no significant difference in the amount of prompt editing (Mdn=7.0 vs. Mdn=8.0, p=.72, r=0.08), suggesting that block-based operations (as shown in Fig. 11 Right) may reduce the necessity for code editing. Participants found these block-based operations to be more *"intuitive"* and *"direct"* ways to modify the generated code compared to directly editing through text. Participants highlighted the block-based operations 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*.

7.3.3 Modular-based Design Enhance Controllability. Participants reported that they could steer *CoLadder* more controllably towards

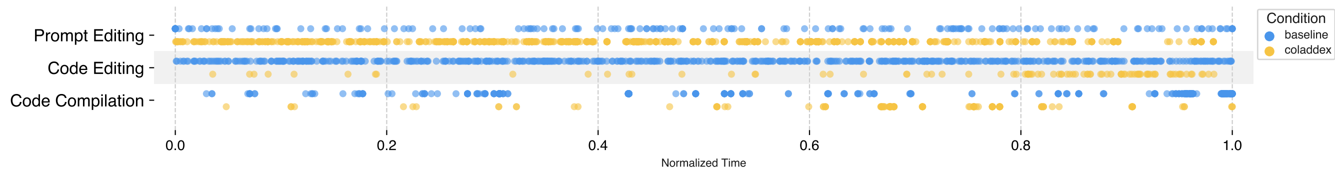

Figure 10: A scatter plot displays events occurring throughout normalized time, synthesized from all participants. In *Baseline*, code editing is spread throughout the workflow, while with *CoLadder*, participants tend to edit code in the later stages.

the task goal (Q_3: Mdn_C=6 vs. Mdn_B=3.0, p=.007, r=.77) with the *"step-by-step approach" - P6*. When comparing the authoring processes in both systems, participants (11/12) felt that *CoLadder* provided them with more controllability while modifying the generated code from prompts. Participants found that 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 are *"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*. They found it useful for modifying the targeted code segments, *"without worrying about affecting other sections" - P4*. 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*.

7.4 Results Evaluation during Prompt Authoring

Compared to *Baseline*, participants found that *CoLadder* significantly reduced the cognitive switching between prompt authoring and code evaluation (Q_1: Mdn_C=6.0 vs. Mdn_B=4.5, p=.01, r=.67) .

7.4.1 List Steps, Auto-Complete, Recommendation Enhancing Intention Alignment Evaluation. All participants experimented with the *List Steps* feature in *CoLadder* (Fig. 11 Right), primarily to *"assess generated code alignment with [their] intents" - P9*. P2 explained, *"If the steps are correct, I am confident the code will be too."* This approach was similarly adopted with the *Recommendation* feature and *Auto-Complete* features (Fig. 11 Right); participants mostly utilized them *"as a cue to see if the system captured my intent" - P8*. Specifically, participants leverage recommendations to evaluate the alignment between their intents and the system's comprehension, rather than accepting the recommendation as the next prompt. 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 evaluating the code for participants using *CoLadder* was to modify their prompts, but not to evaluate the correctness of the generated code.

7.4.2 Improved Recall Performance after Participants Using CoLadder. Despite the reduced need for frequent code evaluation, participants using *CoLadder* demonstrated a significantly better understanding of their programs compared to using *Baseline* (Q_8: Mdn_C=6.0 vs. Mdn_B=5.0, p=.007, r=.77) . A recall test was conducted to investigate participant code comprehension [20]. 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. This result can also be attributed to *CoLadder*'s support in intention formation and externalization compared to the *Baseline*. This trade-off in terms of time efficiency aligns with our previous finding that using *CoLadder* was slower than *Baseline* (Sec. 7.1.1). Participants tended to invest more time in structuring clear intentions in their minds, which aids them in evaluating the code and reduces cognitive load.

7.4.3 Transitioning from Opportunistic Programming to Comprehensive Code Understanding. We observed from Figure 11 that participants ran the code (i.e., *run error* and *run*) significantly more in *Baseline* compared to *CoLadder* (Mdn=24.0 vs. Mdn=18.5, p=.012, r=0.56), which was used as an alternative approach for *"verifying the generated code" - P12*. P1's strategy in using *Baseline* was to *"try to run the code to see if it works,"* without the need to evaluate the generated code. Participants in the *Baseline* condition ran the code throughout the session, whereas participants in the *CoLadder* condition tended to run the code at a later stage (Fig. 10). While this might increase the task completion time, participants using this *opportunistic* approach in *Baseline* faced challenges to modify the code when run results were incorrect, where they *"had to check by cross-referencing task descriptions and generated code" - P1*.

7.5 Enhancing Prompt-Code Correspondence

Participants navigated across various layers of prompt blocks and noted that *CoLadder* significantly enhanced their ability to evaluate generated code (Q_9: Mdn_C=6.0 vs. Mdn_B=3.0, p=.006, r=.80) in comparison to *Baseline*.

7.5.1 Enhancing Navigation and Precise Modification through Corresponding Code Highlight in Modular-Based Approach. Participants found it more effective to navigate and make precise modifications with *CoLadder* as they could adopt a modular approach. They could evaluate 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

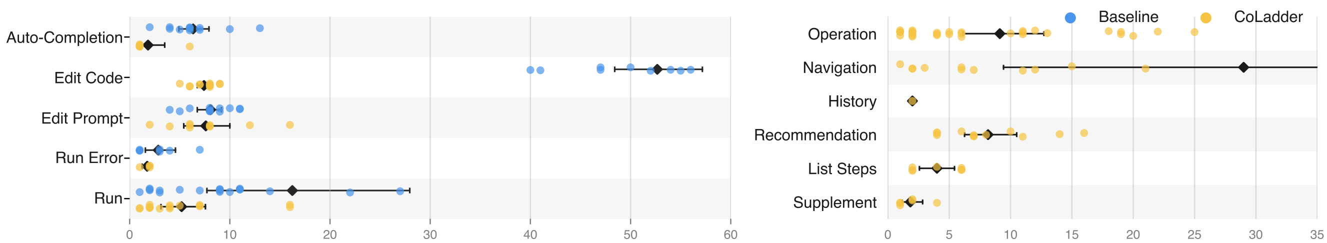

Figure 11: Comparing log event counts between *Baseline* and *CoLadder*. Black dots represent means, and the bars denote 95% CI.

programmers to *"more easily control the depth of verification required"* - *P6*. Some suggested that having a structural intention formed beforehand allowed them to *"swiftly locate the segments needing changes"* - *P8*. We observed that most participants (10/12) do not evaluate the entire program but evaluate 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*. Participants in the *Baseline* condition, on the other hand, encountered difficulties in identifying differences from the integrated prompt. The majority of participants (9/12) expressed frustration with the time-consuming process of *"constantly verifying the same code"* - *P7*, which at times led them to opt for directly modifying the code. P5 further reported challenges of not knowing whether all their intentions were effectively conveyed to *Baseline* and not being able to discern *"which prompts were attributed to incorrect results."*

7.5.2 Facilitating Detailed Code Evaluation with Semantic Highlight and Mixed Methods Writing. 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 evaluation 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 evaluation 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. Figure 9 displays the count of prompt blocks written with mixed methods, ranging from the first to the maximum layer. Interestingly, participants tended to embrace mixed methods in the later layers, considering them to be the form closest to the generated code. While most participants used mixed methods to reflect their intentions, one participant explained the advantage of employing mixed methods in the final layer because *"it is the nearest [in distance] to the generated code [on the right-hand side]"* - *P7*.

8 LIMITATIONS

While our study tested the system with a relatively small program for contextualized results, we designed it to scale and support larger codebases in considering the scalability issue. We implemented lazy loading, where *CoLadder* renders nodes only as they are expanded. The hierarchy and folding mechanisms are also designed to support large codebases. However, a major challenge with scaling is the context window limit of the current model, which could potentially be addressed by adopting retrieval-augmented generation.

8.1 Programming Tasks

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) [39, 67], tend to prioritize quick idea iteration over code quality. Programmers often need to make swift, small changes, such as adjusting parameters and variables [65, 80]. 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. Our study also focuses on the activities of planning and writing code, without fully examining other software engineering tasks such as debugging and testing. However, from observing iterative code editing behavior, we found that participants gained a better understanding of their own code, making them more aware of where changes should be made (Sec. 7.4.3).

8.2 Top-Down Mental Models

Another limitation of *CoLadder* is that the prompt tree structure is specifically useful for the top-down mental model of solving programming tasks. While "mental models of computer programs are founded on the recognition of basic patterns/schemas which are hierarchical and multilayered" [75], it might encompass the bottom-up or incremental approaches commonly used in event-driven programming [17, 70]. In these scenarios, the prompt tree structure may not always align with the actual code structure. While we initially designed the separation of the user interface for the prompt editor and code editor with consideration of this issue, two participants mentioned that our system may be less effective in certain programming languages or scenarios (e.g., Object-Oriented programming [18] or complex projects with multiple files) where the task and code structures can deviate significantly. We argue that while the correspondence between prompts and code might be less obvious, the highlighting of the corresponding (DG4) and hierarchical generation could still be beneficial. This is in contrast to the *Baseline*, where programmers would need to locate the generated code and manually map it to the prompts.

8.3 Programming Languages

We acknowledge the limitation of testing *CoLadder* with scripting languages and recognize that certain programming languages, particularly compiled languages, may face challenges with existing features, such as compilation errors. However, our design of *CoLadder* primarily emphasizes support for the intention formation and externalization stages, arguing that the hierarchical mental representation (DG1) is broadly applicable across various programming scenarios and languages [26, 33, 41]. The need for scaffolding intentions with controllability (DG2) and reducing cognitive switching (DG3) are also common challenges raised by prior works in different programming settings [45]. Overall, while *CoLadder* was primarily designed to tackle challenges in general NL-based programming, we acknowledge the future need for designs more tailored to specific programming tasks and languages. For example, exploring the integration of visualizations such as class diagrams to better convey program structures and relationships between classes or components [29]. We also acknowledge that our system is not designed for visualizing data flow, control flow, or algorithm animations.

9 DISCUSSION AND FUTURE WORK

We discuss how *CoLadder* affects programmers in forming intentions for tasks, balancing control and over-reliance, and the potential effects of task familiarity.

9.1 Intentions Formation & Development

In the programming context, intentions encompass programmers' mental models of understanding and interpretation of the code, underlying programming tasks, and the overall structure of the programs they are working on [4, 18, 79]. Several theories describe the formation of these intentions [19, 71]. Programmers must develop intentions at different levels of abstraction [5, 79], 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 intentions [25, 47, 62]. Our findings resonate with prior research highlighting the importance of programmers forming intentions of code at different levels of abstraction [62], from specific code statements to larger program structures [5, 79]. Participants used *CoLadder* to decompose their prompts hierarchically, which allowed them to create prompts that varied from generalized goals to specific actions. This hierarchical prompt decomposition enabled users to represent their intentions throughout an 'intention formation process', helping them transition from overarching goals to smaller, manageable subgoals. As a result, *CoLadder* supported participants in thinking about individual parts of their code piecemeal, facilitating their intention formation process.

Another line of intention formation describes how experts in a domain adapt their intentions to the context [63]. This literature describes intention formation as *transactional*, involving back-and-forth communication between the user and the situation. This implies it can be achieved through changing user actions, reframing the situation, or altering AI behavior. From the study, it is yet unclear in which aspect the intention is being formed, but future work could investigate how programmers co-adapt their workflow and intentions with these AI-driven code assistants.

9.2 Control and Over-Reliance

We observed subtle differences in *controllability* between directly editing code and translating intentions into prompts. The former involves manual code edits, while the latter focuses on converting decomposed intentions into generated code. Three participants found editing code in *CoLadder* without visual cues less intuitive, indicating that while *CoLadder* enhances control over intention scaffolding, it may not improve direct program control. However, participants required less code editing with *CoLadder* due to alternative methods for modifying code segments through block-based operations. These operations aid in externalizing intentions and intuitive modifications, though they may not enhance overall program control. Future research should explore controllability in AI interactions [3, 34, 55] and program control [28, 53].

We also noticed a trend of over-reliance on baseline code assistants, where programmers often accepted suggestions and modified generated code, aligning with prior studies [6, 13, 78]. In contrast, *CoLadder* users crafted their programs step by step, with a deeper level of detail and decomposition based on the logs, and performed better in recall tests [50]. This raises questions about the implications for future maintenance tasks, suggesting that *CoLadder* might result in more maintainable codebases and offer advantages in long-term comprehension and modification [2]. Future research could further investigate *CoLadder*'s effectiveness in addressing over-reliance issues in human-AI interactions [3].

9.3 Experiences and Task Familiarity

We studied with experienced programmers because they can form more well-defined intentions when addressing programming tasks compared to novices [15, 26, 32, 79]. However, we are also interested in how task familiarity might cause differences among experienced programmers. According to our pre-study questionnaire, 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. 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 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. Potential future research could investigate the impact of varying degrees of task familiarity on the workflow when using *CoLadder*.

10 CONCLUSION

In this paper, we present *CoLadder*, an interactive system that assists programmers in directly manipulating code generation based on their intentions. It achieves this by providing hierarchical task decomposition, modular-based code generation, and result evaluation during prompt authoring. Our iterative design process uncovered strategies and programmers' needs for externalizing intentions and translating them into NL prompts for code generation. A user study further demonstrated *CoLadder*'s capacity to enhance programmers' ability to navigate and edit code across various abstraction levels, from initial goal to final code implementation. Our work provides valuable design implications for future LLM-driven code assistants.

ACKNOWLEDGMENTS

We thank all our participants for their time and valuable input. We would also like to thank our reviewers whose insightful comments have led to a great improvement in this paper. This work is supported in part by the NSERC Discovery Grant, the University of Waterloo faculty start-up fund.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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).

[8] 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.

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

[10] Virginia Braun and Victoria Clarke. 2019. Reflecting on reflexive thematic analysis. *Qualitative research in sport, exercise and health* 11, 4 (2019), 589–597.

[11] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]

[12] 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).

[13] 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).

[14] Richard E Clark, David F Feldon, Jeroen JG Van Merrienboer, Kenneth A Yates, and Sean Early. 2008. Cognitive task analysis. In *Handbook of research on educational communications and technology*. Routledge, 577–593.

[15] 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.

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

[17] Simon P Davies. 1993. Models and theories of programming strategy. *International journal of man-machine studies* 39, 2 (1993), 237–267.

[18] 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.

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

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

[21] Ilker Etikan, Sulaiman Abubakar Musa, Rukayya Sunusi Alkassim, et al. 2016. Comparison of convenience sampling and purposive sampling. *American journal of theoretical and applied statistics* 5, 1 (2016), 1–4.

[22] Li Feng, Ryan Yen, Yuzhe You, Mingming Fan, Jian Zhao, and Zhicong Lu. 2024. CoPrompt: Supporting Prompt Sharing and Referring in Collaborative Natural Language Programming. In *Proceedings of the CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '24)*. Association for Computing Machinery, New York, NY, USA, Article 934, 21 pages. https://doi.org/10.1145/3613904.3642212

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

[24] Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. 2023. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797* (2023).

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

[31] John R Hayes. 2013. A new framework for understanding cognition and affect in writing. In *The science of writing*. Routledge, 1–27.

[32] 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.

[33] J-M Hoc. 1977. Role of mental representation in learning a programming language. *International Journal of Man-Machine Studies* 9, 1 (1977), 87–105.

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

[35] 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).

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

[37] Robin Jeffries. 1982. A comparison of the debugging behavior of expert and novice programmers. In *Proceedings of AERA annual meeting*. 1–17.

[38] 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.

[39] 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.

[40] Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2022. Decomposed prompting: A modular approach for solving complex tasks. *arXiv preprint arXiv:2210.02406* (2022).

[41] Jinwoo Kim, F Javier Lerch, and Herbert A Simon. 1995. Internal representation and rule development in object-oriented design. *ACM Transactions on Computer-Human Interaction (TOCHI)* 2, 4 (1995), 357–390.

[42] LangChain. 2023. https://www.langchain.com/

[43] 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

[44] Chengshu Li, Jacky Liang, Fei Xia, Andy Zeng, Sergey Levine, Dorsa Sadigh, Karol Hausman, Xinyun Chen, Li Fei-Fei, et al. 2023. Chain of Code: Reasoning with a Language Model-Augmented Code Interpreter. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*.

[45] Jenny T Liang, Chenyang Yang, and Brad A Myers. 2023. A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE

Computer Society, 605–617.
[46] Henry Lieberman. 2001. *Your wish is my command: Programming by example.* Morgan Kaufmann.
[47] 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.
[48] 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.
[49] 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).
[50] 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.
[51] Microsoft. 2023. The Editor of the Web. https://microsoft.github.io/monaco-editor
[52] 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).
[53] 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.
[54] Donald A Norman. 1986. Cognitive engineering. *User centered system design* 31, 61 (1986), 2.
[55] Donald A Norman. 1994. How might people interact with agents. *Commun. ACM* 37, 7 (1994), 68–71.
[56] OpenAI. 2023. ChatGPT (Feb 13 version) [Large language model]. https://chat.openai.com
[57] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]
[58] Pyodide. 2023. Pyodide is a Python distribution for the browser and Node.js based on WebAssembly. https://github.com/pyodide/pyodide
[59] 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.
[60] Pablo Romero. 2001. Focal structures and information types in Prolog. *International Journal of Human-Computer Studies* 54, 2 (2001), 211–236.
[61] 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.
[62] 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).
[63] Donald A Schön. 2017. *The reflective practitioner: How professionals think in action.* Routledge.
[64] Amal Shargabi, Syed Ahmad Aljunid, Muthukkaruppanan Annamalai, Shuhaida Mohamed Shuhidan, and Abdullah Mohd Zin. 2015. Program comprehension levels of abstraction for novices. In *2015 International Conference on Computer, Communications, and Control Technology (I4CT)*. IEEE, 211–215.
[65] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems* 25 (2012).
[66] Hari Subramonyam, Roy Pea, Christopher Lawrence Pondoc, Maneesh Agrawala, and Colleen Seifert. 2024. Bridging the Gulf of Envisioning: Cognitive Challenges in Prompt Based Interactions with LLMs. (2024).
[67] John W Tukey et al. 1977. *Exploratory data analysis*. Vol. 2. Reading, MA.
[68] 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.
[69] 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).
[70] Willemien Visser. 1990. More or less following a plan during design: opportunistic deviations in specification. *International journal of man-machine studies* 33, 3 (1990), 247–278.
[71] Anneliese von Mayrhauser and A Marie Vans. 1995. Industrial experience with an integrated code comprehension model. *Software Engineering Journal* 10, 5 (1995), 171–182.
[72] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL]
[73] 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.
[74] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382* (2023).
[75] Susan Wiedenbeck, Vikki Fix, and Jean Scholtz. 1993. Characteristics of the mental representations of novice and expert programmers: an empirical study. *International Journal of Man-Machine Studies* 39, 5 (1993), 793–812.
[76] 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.
[77] 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.
[78] 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.
[79] Nong Ye and Gavriel Salvendy. 1996. Expert-novice knowledge of computer programming at different levels of abstraction. *Ergonomics* 39, 3 (1996), 461–481.
[80] 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.

A PROMPT TEMPLATE

```
You are a professional Python developer and your task is to write code based on user
prompts.
You will be provided with a prompt tree ('$PROMPT_TREE') that each single lines contains
the prompt, id. And it represents parent-child relationships using indentation.
Your objective is to write the whole Python code based on each lines of prompt in the
$PROMPT_TREE.
The written code should contain prompts in the $PROMPT_TREE and it's corresponding ID
(for example: # [ID: 6] set legend) as comments of the code.
The overall written code should be able to perform task described in the prompt tree, and
try to remain the code as the same as the original code $ORIGINAL_CODE (if provided).
Make sure the written code contains all the prompts in the $PROMPT_TREE, but not
necessarily write code for all the lines in the $PROMPT_TREE.

1. Review these examples to understand the format and structure of the expected output.
2. Then, apply similar reasoning and structure to write code for the given prompt tree.

---

Examples:
[...]
[Input Example N]
$PROMPT_TREE:
load iris dataset (ID: 2)
visualize the iris dataset (ID: 1)
  scatter plot (ID: 4)
    set legend (ID: 6)
    display the plot (ID: 7)

Think step by step:
- For 'load iris dataset', import and load the dataset into a DataFrame.
- For 'visualize the iris dataset' with a 'scatter plot', set up the plot with axes.
- 'Set legend' will add a legend to the plot.
- 'Display the plot' will show the visual.

[Output Example N]
# [ID: 2] load iris dataset
from sklearn.datasets import load_iris
data = load_iris()
X = data.data
y = data.target

df = pd.DataFrame(X, columns=data.feature_names)
df['target'] = y
print(df.head())

# [ID: 1] visualize the iris dataset
import matplotlib.pyplot as plt
plt.figure(figsize=(8, 6))
# [ID: 4] scatter plot
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis')

# [ID: 6] set legend
plt.legend(iris.target_names)

# [ID: 7] display the plot
plt.show()

Output:
(Your written code based on the above instructions and examples)
```

Figure 12: Prompt template for the block's operation [Add]

```
You are an experienced developer specialized in iteratively editing code snippets based
on users' requests. Your primary task is to generate a code segment that implements the
modification described in $NEW_PROMPT to replace $PREVIOUS_CODE generated by the
$PREVIOUS_PROMPT. You only have to generate segments of code that can replace the segment
of $PREVIOUS_CODE, instead of generating the entire program again. Make sure the newly
generated code segment can be inserted into the $PROGRAM and perform the task described
in $PROMPT_TREE. Make sure the generated code follows correct Python syntax and does not
include any explanations, context, corrections, or comments. If the prompt given contains
child prompts, you should generate the whole Python code based on each line of the prompt
in the $PROMPT_TREE. The generated code should contain prompts in the $PROMPT_TREE and
its corresponding ID (for example: # [ID: 6] set legend) as comments of the code. The
overall generated code should follow correct Python syntax and be executable to perform
the task described in the prompt tree.

Sample Input:
$NEW_PROMPT: "visualize the iris dataset in pairplot"
$PREVIOUS_CODE: ""
$PREVIOUS_PROMPT: "visualize the iris dataset"

$PROMPT_TREE:
load iris dataset (ID: 1)
visualize the iris dataset (ID: 2)
  plot the dataframe with scatter plot (ID: 4)


$PROGRAM:
# [ID: 1] load iris dataset
from sklearn.datasets import load_iris
data = load_iris()

# [ID: 2] visualize the iris dataset
import matplotlib.pyplot as plt
plt.figure(figsize=(8, 6))

# [ID: 4] plot the dataframe with scatter plot
import seaborn as sns
import pandas as pd
df = pd.DataFrame(data.data, columns=data.feature_names)
df['target'] = data.target
sns.pairplot(df, hue='target')
plt.show()

User Input:
$NEW_PROMPT: new prompt to generate code
{prompt}

$PREVIOUS_CODE: the code segment to change
{previousCode}

$PREVIOUS_PROMPT: the prompt that the user provided before
{previousPrompt}


Background Context:
$PROMPT_TREE: the current task description
{promptTree}

$PROGRAM: the entire program
{allProgram}
```

Figure 13: Prompt template for [Edit]

```
You are an experienced Python developer, specialize in summarizing code snippets. Users
provide a sentence of prompt $PROMPT with the corresponding code snippet $CODE and the
context of the whole program $CODE_CONTEXT.

Your task is to scaffold the $PROMPT into subtasks, semantically segmenting $CODE into >1
subtasks, summarizing each with concise sentences (≤8 words). Include keyword syntax
(e.g., 'for loop', 'if statement') and variable names.

Ensure the sub_prompt differs from the prompt, explaining its purpose.

Return > 1 JSON objects in a list with the structure:
  "sub_prompt": the summary of the subtask,
  "segmented_code": the code segment corresponding to the subtask


User Input:
$PROMPT:
{prompt}
$CODE_CONTEXT:
{codeContext}

Code Segment to Summarize, $CODE:
{code}
```

Figure 14: Prompt template for [List Steps]

```
You are an experienced Python developer specializing in semantic highlighting for code
snippets. Your task is to semantically highlight keywords in a list of prompts that form
a tree structure. You will be provided with a subTree ($SUBTREE) that contains a list of
prompts, a selected text ($selectedText) from the given prompt ($PROMPT), and the code
context ($CODE_CONTEXT) of the entire program.

First, you have to semantically segment all the prompts in the subTree into a list of
constituents. Then, you have to calculate the correlation score between the selected text
and each constituent.

To accomplish this, return a list of JSON objects with the same length as the subTree.
Each object should contain the "original_prompt" and the "semantic_segmentations" of the
corresponding prompt.

The "semantic_segmentations" should be a list of JSON objects with the following
structure:
  "id": the id of the prompt,
  "original_prompt": the original prompt,
  "semantic_segmentations": the semantic segmentation of the prompt,
    "constituent": the constituent of the prompt,
    "corresponding_code": the corresponding code segment of the constituent,


Below is an example illustrating the desired input and output format:
Example input:
$SUBTREE:
"Load the Iris dataset (ID: 1)"
"Visualize the Iris dataset (ID: 2)"
  "Create a dataframe from the iris data (ID: 4)"
  "Create a pairplot of the dataframe (ID: 5)"
  "Display the plot (ID: 6)"

Example Output:
[
  {
    id: 1,
    original_prompt: "Load the Iris dataset",
    semantic_segmentations: [
      {
        constituent: "Load",
        corresponding_code: "from sklearn.datasets import load_iris",
      },
      {
        constituent: "Iris dataset",
        corresponding_code: "load_iris()",
      }
    ]
  },
  ...
]
```

Figure 15: Prompt template for [Semantic Highlighting]

B FORMATIVE STUDY MATERIALS

Sex	Age	Programming Years	AI Familiarity	Programming Familiarity	Usage of LLMs (times/week)	Languages Usage in LLMs	Tasks Usage in LLMs
Male	26	6	5	5	>19	Python, C#/C++, JavaScript	Unfamiliar Code, Algorithm
Female	27	6	4	4	3–5	Python, JavaScript	Unfamiliar Code, Boilerplate, API Usage
Male	25	5	5	4	5–7	Python, Bash	Unfamiliar Code, Debugging
Male	27	10	4	4	7–10	JavaScript, Java, C/C++	Unfamiliar Code, Boilerplate, Code Refactoring
Male	25	7	4	4	3–5	Python, Go, Rust	Debugging, Code Refactoring
Male	25	6	5	5	3–5	Python, C#/C++, JavaScript	Unfamiliar Code, Boilerplate, API Usage

Table 2: Participants in the formative study used various programming languages and accomplished diverse tasks using the LLMs.

C EVALUATION STUDY MATERIALS

C.1 Programming Tasks Categories

Category	Tasks
Basic Python	T1-1 Randomly generate and sort numbers and characters with dictionary
	T1-2 Date & time format parsing and calculation with timezone
File	T2-1 Read, manipulate and output CSV files
	T2-2 Text processing about encoding, newline styles, and whitespaces
OS	T3-1 File and directory copying, name editing
	T3-2 File system information aggregation
Web Scraping	T4-1 Parse URLs and specific text chunks from web page
	T4-2 Extract table data and images from Wikipedia page
Web Server & Client	T5-1 Implement an HTTP server for querying and validating data
	T5-2 Implement an HTTP client interacting with given blog post APIs
Data Analysis & ML	T6-1 Data analysis on automobile data of performance metrics and prices
	T6-2 Train and evaluate a multi-class logistic regression model given dataset
Data Visualization	T7-1 Produce a scatter plot given specification and dataset
	T7-2 Draw a figure with three grouped bar chart subplots aggregated from dataset

Table 3: Overview of 14 programming tasks across 7 categories [78].

C.2 Programming Tasks

Feature Selection and SVM Classification

The task at hand revolves around the concept of logistic regression, a statistical method revered for its effectiveness in classification scenarios. Your journey here is not just about applying this method, but mastering it, understanding its intricacies and how it interacts with the `wine` dataset from scikir-learn at hand.

A key aspect of your exploration involves the notion of regularization, a critical component in the world of machine learning. It's a balancing act, where the regularization parameter 'c' plays a pivotal role. Experimentation is the name of the game here, with values like {0.01, 0.1, 1, 10}, and 100 offering a spectrum of scenarios to explore.

Cross-validation, particularly the 5-fold variety, emerges as an integral part of your strategy. It's not just about applying the model but validating it, testing its mettle against different segments of the data. Also, apply `lbfgs` as the optimization approach.

The culmination of your task is not merely in the application of these techniques but in the synthesis of your findings. The cross-validation mean accuracy stands as a testament to your model's performance, a numerical expression of how well your logistic regression model, fine-tuned with the right balance of regularization, can classify and understand the nuances of the wine dataset.

Expected Output Format:
Cross-validation average accuracies (up to 2 decimal places)

```
Cross-validation Accuracy (c = 0.01): 0.96
```

Figure 16: Machine Learning Task 1

Exploring Wine Quality Through Regression Analysis

In this task, your journey involves the wine dataset from scikit-learn, a resource rich in data yet untapped for its potential in revealing insights into wine quality.

Consider the concept of `quality` in the context of wine. Imagine encapsulating the essence of quality as a singular value, derived from the mean of all 13 feature values in the dataset. This newly crafted `quality` metric becomes your target variable, a beacon guiding your analysis.

The dataset, in its entirety, is a canvas too broad for precise strokes. Hence, partitioning it into a train set and a test set (with a 70-30 split) offers a more focused approach. This step is not just about dividing data; it's about creating two realms – one for training your model and the other for testing its predictions.

Your tool of choice for this expedition is the Decision Tree Regressor. But this is no ordinary application of a regressor. You choose `friedman_mse` as your criterion, a decision that adds a layer of sophistication to how the model assesses quality. Similarly, opting for `random` as the splitter adds an element of unpredictability, mirroring the often unpredictable nature of wine quality itself.

The true measure of your journey's success lies in the evaluation of the trained regressor on the test set. The predictions are held up against reality, and the model's understanding of `quality` is truly tested.

Your final act is one of communication – reporting the train and test accuracies, each a numerical testament to the model's ability to learn and predict. These accuracies, precise up to two decimal places.

Expected Output Format:
train and test accuracies on two separate rows

```
Train accuracy: 0.98
Test accuracy: 0.95
```

Figure 17: Machine Learning Task 2

Scatter Plot Exploration with the Iris Dataset

Conceptual Framework: Your canvas is the well-known `iris` dataset from Scikit Learn, a dataset that offers a fascinating glimpse into the characteristics of various iris flowers.

Visualization Objective: Your primary aim is to create a scatter plot, but not just a simple plot. This scatter plot should vividly represent the relationship between sepal length and sepal width of iris flowers. Each point in the plot is a story, representing an iris flower, with its sepal dimensions providing the narrative.

Details and Nuances:

- The x-axis of your plot will represent the sepal length, while the y-axis will display the sepal width, both crucial measurements in the study of iris flowers.
- The representation of data points is key: solid dots, each marking the presence of an iris flower in this two-dimensional space.
- Diversity in nature is best captured through color. Assign a unique color to each iris species, making the plot not only informative but also visually appealing.
- An interesting twist: arrange the points in ascending order of petal length along the x-axis. This arrangement will reveal patterns and perhaps raise new questions about the relationship between sepal and petal dimensions.
- Finally, clarity in communication is essential. Include a legend in the lower right corner of the plot, correlating each flower species with its designated color.

Outcome: Your scatter plot will be more than a visualization; it will be an insightful exploration of the iris dataset, revealing the intricate relationships between different flower measurements.

Figure 18: Data Visualization Task 1

Visualization Task: Analyzing the Diabetes Dataset

Conceptual Framework: Your next visualization challenge involves the `diabetes` dataset from Scikit Learn, which offers a comprehensive view into the medical profiles of diabetes patients. The dataset contains various health metrics, providing a rich field for analysis and visualization.

Visualization Objective: The task is to create a visualization that effectively communicates the relationships and patterns within the diabetes dataset. This dataset isn't just a collection of numbers; it's a window into the health dynamics of individuals with diabetes.

Details and Nuances:
Your goal is to design a visualization that:

- Highlights the relationship between BMI (body mass index) and the quantitative measure of disease progression.
- Utilizes a heatmap to display the correlation between all variables in the dataset, including the target variable.
- Creates a histogram for one of the serum measurements (of your choice) to analyze its distribution among the patients.

Specific Requirements:

- BMI vs. Disease Progression Plot:
  - X-axis: BMI
  - Y-axis: Quantitative measure of disease progression
  - Points: Represent individual patients
  - Additional Element: Add a line of best fit to understand the general trend
- Heatmap for Correlations:
  - Display correlations between all variables
  - Ensure the heatmap is color-coded for better readability
  - Include values in each cell for precise understanding
- Histogram for a Serum Measurement:
  - Choose any one of the serum measurements
  - X-axis: Serum measurement values
  - Y-axis: Frequency of patients
  - Feature: Include mean and median lines on the histogram

Figure 19: Data Visualization Task 2

C.3 Questionnaire

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

C.3.1 UMUX-LITE.

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

C.3.2 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?

C.3.3 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.