CoPrompt: Supporting Prompt Sharing and Referring in Collaborative Natural Language Programming

Felicia Li Feng*
Computational Media and Arts Thrust
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China

Ryan Yen*
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

Yuzhe You
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

Mingming Fan†
Computational Media and Arts Thrust
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China

Jian Zhao
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

Zhicong Lu
Department of Computer Science
City University of Hong Kong
Hong Kong, China

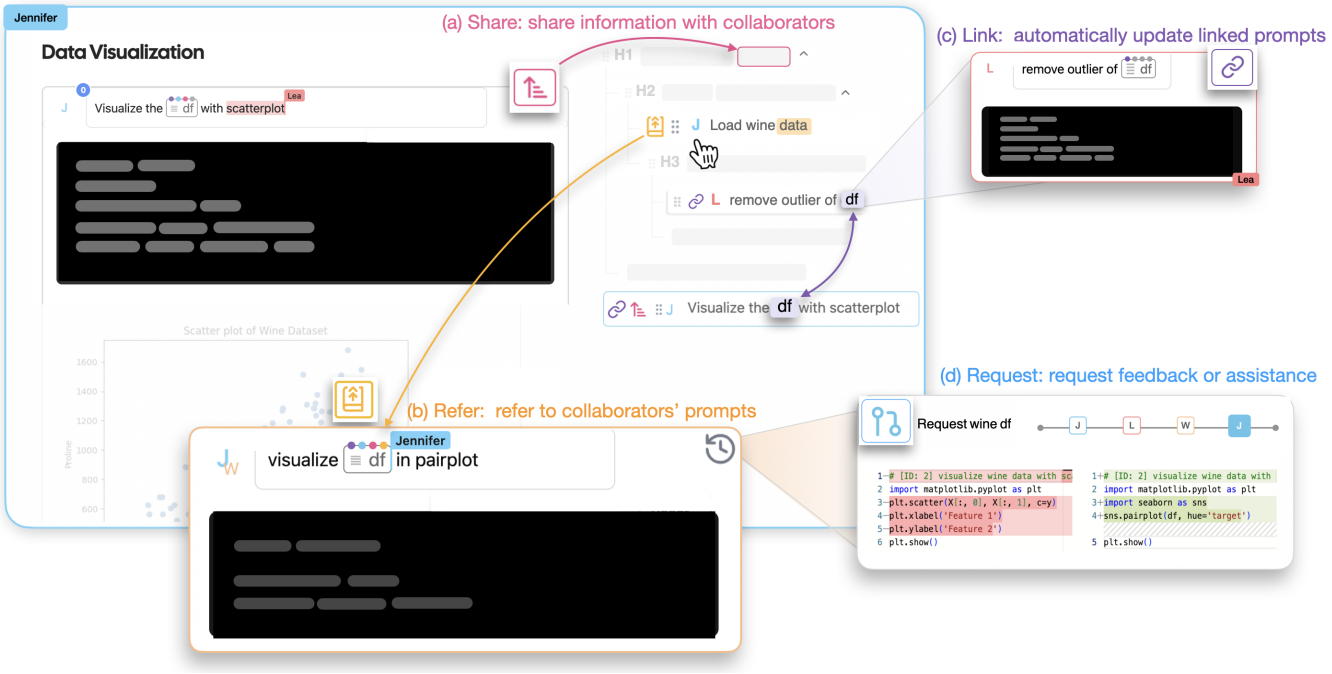

Figure 1: *CoPrompt* enables programmers to conduct collaborative prompt engineering by building upon collaborators' prompts in natural language programming. It provides four mechanisms: (a) *share* mechanism enables programmers to share information with collaborators without much effort or interrupting collaborators' work. (b) *refer* mechanism assists programmers to modify prompts with reference to collaborators' prompts. (c) *link* mechanism automatically updates linked prompts. (d) *request* mechanism enables programmers to request collaborators' assistance or feedback without interrupting collaborators' progress.

ABSTRACT

Natural language (NL) programming has become more approachable due to the powerful code-generation capability of large language models (LLMs). This shift to using NL to program enhances collaborative programming by reducing communication barriers and context-switching among programmers from varying backgrounds. However, programmers may face challenges during prompt engineering in a collaborative setting as they need to actively keep aware of their collaborators' progress and intents. In this paper, we aim to investigate ways to assist programmers' prompt engineering in a collaborative context. We first conducted a formative study to understand the workflows and challenges of programmers when using NL for collaborative programming. Based on our findings, we implemented a prototype, *CoPrompt*, to support collaborative

*Both authors contributed equally.
†Also with Division of Integrative Systems & Department of Computer Science and Engineering, The Hong Kong University of Science and Technology.

prompt engineering by providing referring, requesting, sharing, and linking mechanisms. Our user study indicates that *CoPrompt* assists programmers in comprehending collaborators' prompts and building on their collaborators' work, reducing repetitive updates and communication costs.

1 INTRODUCTION

Collaborative programming has been widely studied and supported through a range of collaborative systems [16, 17, 50, 63, 67, 80–82]. These systems assist programmers in collaboratively writing, discussing, and debugging code across various contexts, such as data science [67, 80–82] and software development [16, 17, 63]. Programming with Natural language (NL) has become more feasible driven by the enhanced code generation capability of large language models (LLMs) [72]. With the context provided by NL prompts, utilizing NL for programming enables programmers to communicate effectively without delving into the intricacies of low-level code [81]. This benefit aligns with programmers' preference for understanding collaborator tasks from a high-level perspective [80].

To get desired code generation results, programmers often need to conduct *prompt engineering*, which involves iteratively refining prompts to guide LLMs in solving programming tasks by verifying the generated results [18, 45, 68]. However, prompt engineering is challenging in collaborative programming. To effectively collaborate with others and ensure that their engineered prompts align with the ongoing work of their collaborators, programmers need to stay informed about their collaborators' progress. This involves regularly reviewing their collaborators' code and engaging in clear communication without causing any disruptions or difficulties for their fellow collaborators. However, programmers encounter difficulties when switching between their code and that of others [16], including issues such as a lack of contextual references [82] and limited support for interactive sharing of intermediate results [7, 47, 61]. Additionally, balancing the inclusion of contextual information in prompts can be challenging for programmers [44, 72]. Deciding on the right amount of detail to incorporate is not always straightforward, resulting in prompts that may either lack essential information (e.g., *"Web Scraping"*) or become overly detailed (e.g., *"Extract all anchor <a> tags from the parsed HTML, and iterate through each"*). This variation can lead to confusion among collaborators and impact the readability and reusability of prompts. In addition, sometimes a prompt is written for better code generation, which only needs to be comprehended by LLMs instead of collaborators (e.g., "example execution results"). These issues increase the cognitive load of making sense of prompts and thus increase the communication cost in collaboration.

The purpose of this research is thus to explore the design of workflows that support programmers in *Prompt Co-Engineering*, which involves collaboratively refining and sensemaking prompts during NL programming. We conducted a formative study to gain insights into potential prompt co-engineering workflows and challenges. Our findings revealed that programmers struggled to maintain a shared common ground, track collaborators' progress, and comprehensively understand the code solely from prompts due to their iterative nature. They also encountered challenges in managing procedural dependencies [48, 73] when dealing with variables represented by different NL prompts, which resulted in repetitive updates. These findings highlight the need to support prompt co-engineering from comprehending collaborators' work to leveraging them to engineer their own prompts.

Informed by these findings, we propose four novel mechanisms for NL programming to reduce the effort needed for building on others' work and sharing information among collaborators: *referring*, *requesting*, *sharing*, and *linking* (Figure 1). *Referring* enables programmers to effortlessly locate and access their collaborators' prompts by presenting user-defined tasks and prompts within a shared multi-level hierarchy view of prompts. *Requesting* and *sharing* enable programmers to share information with their collaborators and solicit feedback to enhance their prompts. Additionally, the *linking* mechanism facilitates automatic updates for elements with procedural dependency, reducing the necessity for repetitive prompt modifications.

Incorporating these mechanisms, we designed *CoPrompt*, a prototype system that assists the workflow of *Prompt Co-Engineering*. *CoPrompt* supports programmers in making sense of collaborators' work with the multi-level hierarchical interactions and contextual prompt information, as well as leveraging collaborators' work and sharing information. To evaluate the usefulness of *CoPrompt* in assisting prompt co-engineering workflow during collaborative NL programming, we conducted a user study involving 12 experienced programmers familiar with LLM-based code assistants and collaborative programming. The results showed that *CoPrompt* effectively supported programmers in understanding their collaborators' prompts and facilitated communication among them to build upon each other's work. In summary, this research makes the following contributions:

- A formative study that uncovered the workflow and challenges of collaborative NL programming.
- *CoPrompt*, a prototype system that supports programmers' prompt co-engineering workflow during collaborative NL programming by comprehending, referring, requesting, sharing, and linking with collaborators' prompts.
- A user study that provided insights into users' perceptions of the usability of *CoPrompt* and design implications for future systems assisting prompt engineering in collaborative NL programming.

2 RELATED WORK

As our research aims to address the challenges of collaborative prompt engineering in collaborative NL programming, we review prior work on natural language programming and LLMs, as well as collaborative programming groupware.

2.1 Natural Language Programming and Prompt Engineering

Natural Language (NL) Programming is the process of using NL to express programming ideas for desired output [54, 55]. Prior work provided insights on how people express computer-like procedures "naturally" and on what features programming languages should include to be more "natural-like" [55]. With the development of

natural language processing (NLP) [10], it has become feasible to use NL to conduct more programming tasks, as it allows more free-form NL utterances to be translated into program code [42]. This advancement increased the accessibility of programming to non-expert users [54] and end-user programmers [35] who lack training in computing.

Recently, the advances in generative AI [56, 84], especially LLMs [8], fostered the capability of generating code from NL prompts, by allowing a wider space of utterances to be transformed into satisfying code snippets. This advance significantly enhanced the performance of AI-driven code assistants [1] and thus improved the satisfaction and accessibility of programming with NL prompts [33, 72]. LLM-powered code assistants allow programmers to write at different levels of abstraction when developing code, which provides a greater degree of freedom [28, 72]. However, the multiple levels of abstractions [23] in the NL prompts also resulted in the abstraction matching problem when using NL for programming, where programmers find it difficult to select an utterance that will translate into the desired system action [49, 64, 72]. A case study investigating the NL prompting process of prototyping also highlighted the difficulty of evaluating whether a prompt is improving [33]. This issue is rooted in the challenges of translating user instructions into executable computer tasks [31].

To mitigate the abstraction matching issue, prior work has investigated ways of prompt engineering, which is the process of engineering an NL prompt to make it more effective in generating desired results [6, 68]. Liu et al. proposed design guidelines for prompt engineering for text-to-image generative models [46]. Common practices in prompt engineering include appending information like explanations [37], demonstrations [13, 41], table schema [77], and relevant examples (few-shot prompts) [32, 45]. Although few-shot prompts have become a popular strategy, they may still behave worse than zero-shot prompts sometimes [68]. To enhance the effectiveness of the prompts, programmers can further specify tasks by constructing the signifier, memetic proxy, and specifying truth-seeking patterns [68].

Other prompt engineering methods include combining specific task information with general intentions (meta-prompts) [68], generating mutations of the prompt [40], elicitating feedback with small data [75], summarizing complicated prompts [36], defining prompt grammar [18], and introducing a new programming language [3, 30]. 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 [71, 78]. There are also practices of breaking down tasks [69] and dividing complex tasks into chained series of sub-tasks [86]. By breaking down complex problems into sub-tasks, the gap in abstraction is reduced, enabling successful guidance of the model to generate code that matches the programmer's intents [4].

However, prompt engineering in the collaborative programming context has not been investigated, which contains specific challenges regarding understanding and leveraging collaborators' work. *CoPrompt* aims to investigate the challenges and the potential benefits the NL prompt with certain levels of abstraction can bring to the collaboration.

2.2 Collaborative Programming

Extensive research in HCI and CSCW has investigated challenges and system designs to assist collaborative programming. Synchronization is a challenging yet significant part of the collaboration, as programmers need to synchronize with their collaborators in various artifacts like data frames, variables, and archives [80]. It is challenging as the artifacts in programming involve procedural dependencies [48, 73]: if one part of the code changes, all related code snippets must be updated to prevent conflict and errors. In addition, programmers often encounter difficulties when switching between their code and that of others [16], including issues such as a lack of contextual references [82] and limited support for interactive sharing of intermediate results [7, 47, 61].

Establishing group awareness can reduce communication costs and thus improve collaboration efficiency [15, 79]. It involves understanding the activities of others, information sharing, and knowledge of group and individual contexts [11, 24, 25, 29, 38, 57]. To facilitate comprehending the complex dependencies and relationships among collaborators' work, Albireo displays the relationships between the cells of a notebook using a dynamic graph structure [85]. Documentation plays an important role in maintaining shared understanding and group awareness [14, 34, 51, 70, 74]. To document the development progress, programmers write comments to make the code easier for both themselves and others to understand [60]. Comments are also essential for sharing intermediate results [62]. However, writing comments is tedious which makes many programmers not bother to write comments in time. The lack of detailed explanations and intention-revealing comments causes trouble for others understanding their work [21]. To make the commenting process easier, Themisto leveraged AI to provide AI-assisted comments based on deep learning, query, and prompt [81]. Their user study suggested that the collaboration between data scientists and Themisto significantly reduced task completion time and resulted in satisfaction.

Reusing collaborators' work is also challenging in collaborative programming [43, 72, 83]. To facilitate referring to collaborators' work, chat.codes enabled programmers to link code with messages in the chatroom [58]. In addition, Codeon provides on-demand remote collaboration assistance by automatically capturing the relevant code context and allows remote helpers to respond with high-level descriptions, code snippets, and NL explanations.[9]. Communication is essential for maintaining shared understanding and group awareness in collaborative work [24]. While it is a time-consuming activity in software developing collaboration [29]. Prior work investigated ways of reducing collaborators' communication costs through documentation, comments, visualizations, and version control systems [9, 21, 27, 60]. Version control is also supported as it is important for maintaining group awareness and dealing with conflicts [17, 27].

However, These collaboration systems have not taken NL programming into consideration, where the challenges of comprehending and leveraging collaborators' work may be different. *CoPrompt* aims to investigate the challenges and potential solutions for the challenges in collaborative NL programming, especially prompt co-engineering.

3 FORMATIVE STUDY

We conducted a formative study to understand the challenges faced by programmers and their needs in the workflow of prompt co-engineering. Specifically, we focused on how programmers comprehend and build upon their collaborators' work to iteratively refine their own prompts for generating code that accurately matches their intents.

3.1 Participants and Procedure

Five pairs of experienced programmers familiar with LLM-based code assistants. Participants reported 6–24 months of experience with an AI code assistant and 3–5 years of experience using computational notebooks. All participants were 20–35 years old and had at least bachelor's degrees in a CS-related field.

We asked participants to work remotely in pairs on an exploratory data programming task [2] using a shared Jupyter notebook in the VSCode editor [53] embedded with the GitHub CoPilot plugin. Participants were asked to join a Zoom meeting first to discuss their task distribution and collaboration workflow. Then, they started working on their own tasks using natural language prompts, and they were allowed to communicate via audio in the meantime. After the 90-minute collaborative programming session, participants were asked to attend follow-up interviews, which lasted approximately 30 minutes. All participants received compensation according to local standards. The studies were logged using VSCode extensions and the process was video recorded and transcribed. The first and second authors conducted a thematic analysis involving cross-referencing timestamped data of prompt modifications from system logs with video recordings, identifying the events that transpired before prompt engineering [5]. We categorized and analyzed a total of 229 instances out of 392 recorded interactions. We excluded 163 actions due to a lack of clear context or relevance to the specific collaborative communication events that preceded prompt modifications.

3.2 General Workflows in Prompt Co-Engineering

In the following paragraphs, we present our findings of the workflows that participants adopted in prompt co-engineering and the challenges **(C)** that they encountered.

In the initial stages, participants convened online to gain an understanding of the data and discuss tasks. Subsequently, high-level task distribution was discussed and noted down with comments at the beginning of a collaborative computational notebook. We noticed that most pairs (N=4/5 pairs) structured the task by hierarchically numbering lists or bullet points in order to externalize the task structure in their minds.

Upon settling on a preliminary task distribution, participants started writing NL prompts independently to accomplish their own distributed tasks. Throughout the process, they maintained communication via Zoom to discuss ongoing and potential code implementations. We observed that all participants started the task with a high-level description of the task (e.g., data preprocessing, data modeling) with the methods involved (e.g., low-pass filtering, linear regression), deliberately omitting details (e.g., variable names, parameters, return values). Participants mentioned the reason is that they have to *"wait until her [collaborator's] work is complete"* -P4 to continue adding more details to the prompt. Therefore, participants do not verify the generated code in detail at first because they know that *"it will eventually be changed later"* -P2. To further refine their prompts, participants primarily go through these three stages:

Stage 1 — Comprehension: Regularly checking in on collaborators' work became a common practice to gain insight into the evolving code generated from prompts. The goal was either to refer to or reuse components of a collaborator's task or to assess their progress and determine the next steps. However, this approach presented several challenges, including the significant time spent *"scrolling up and down to identify changes"* -P9. Most participants (7/10) mentioned that the hierarchical structure (i.e., headings within the markdown cells) in the notebook assisted them in navigating and identifying collaborators' progress, and served as a *"anchor point."* -P4 Participants also encountered difficulties in comprehending their collaborators' code based solely on the prompt, often describing the prompts as *"unorganized"* and *"vague"*. Additionally, P7 highlighted another issue where the generated code *"sometimes not aligned with the prompt,"* further complicating the comprehension process. Lastly, participants faced challenges in tracking their collaborators' current and past progress, including monitoring the revision history of prompt and code, which is essential for understanding *"the reasoning behind code changes"* -P1 (**C1**).

Stage 2 — Pre-Modification Interactions: Programmers often adapt their prompts based on their own experience in the past and the current work, which can be challenging to explicitly document comprehensively. We thus focus on the explicit collaborative strategies employed prior to the start of prompt engineering. Based on the thematic analysis results, these pre-modification interactions consisted of a series of actions (Table 1).

Syncing up with collaborators. Many participants found that their initial task distribution was not detailed enough, which caused redundant effort and inappropriate workflow between collaborators: *"My collaborator and I encoded the data at the same time"* -P3. Participants also reported the tedious process of updating prompts due to procedural dependency [47], in which a downstream prompt only works if a particular upstream prompt works normally. Due to the ever-changing nature of data programming work and procedural dependencies, programmers often need to monitor their collaborators' changes and update their prompts to align the generated results with their collaborators' changes, otherwise, they may *"encounter error messages due to collaborators' modifying the data frame halfway through the process"* -P2 (**C2**).

Requesting for collaborators' feedback and assistance. All participants who were in charge of visualizing correlation (P1, 4, 5, 8, 9) left comments for their collaborators to provide feedback, as it is an essential step for data analysis. We also observed that some participants (P3, 5, 8, 9) requested help from their collaborators when they could not find any reference from existing prompts. For instance, P3 asked his collaborator to handle a sub-task that he failed to complete with CoPilot's assistance. There are also cases where collaborators need to work closely and go through a trial-and-error process together: *"I asked my collaborator to pay attention to change the way of dealing with outliers every time the way of*

Action	Description	Example	*n*
Reactive Communication (Request & Response)	These interactions aim to seek assistance, feedback, or validation regarding reactively handling specific aspects of the prompt or generated code.	"Can someone help me determine the function for encoding?"	61
Clarifying	Participants seek answers to queries about their collaborators' work, including seeking explanations, and verifying the correctness of specific code segments.	"Did you drop the column of xxx?"	57
Sync Up	Participants communicated to align their efforts and ensure consistency in their coding tasks. These interactions helped prevent redundancy and maintain cohesiveness in their prompt engineering process.	"I am doing the encoding for the whole dataset."	49
Reference & Reuse	Participants occasionally referred to and reused (e.g., copy-paste) components from their collaborators' work, utilizing these references to inform their own prompt modifications. These actions fostered a sense of collaboration and knowledge exchange.	"I used the function you wrote in my block. Any concerns?"	33
Proactive Communication	Proactive communication involves participants sharing insights, updates, or relevant information related to their prompts or coding tasks. These exchanges often contributed to a deeper understanding of the prompt's context and its alignment with the programming task.	"Here's an update on the changes I made to the prompt..."	29

Table 1: Five types of actions of the pre-modification interactions.

feature transformation is changed" -P4. However, most participants (N=7) indicated that they desire a non-interruptive method to send their collaborators requests, instead of speaking up, which is too interruptive for them to use frequently (**C3**).

Referring to collaborators' processes and prompts. All participants checked their collaborators' processes and referred to their prompts to improve their own for better generation results across the whole notebook. To leverage others' prompts, participants first locate and read the target prompt in order to make sense of it. Then, they copy portions of the prompt relevant to their task and integrate them into their own prompts to provide contextual information for improved generation results. The redundant process of copy-pasting and modifying prompts can be time-consuming (**C4**), as programmers may trial-and-error to determine the appropriate modifications of the prompts *(P1: "I reused my collaborator's prompt, which did not work as I imagined. After analyzing its context, I realized that I had to copy a prompt several blocks above that").*

Proactive communication for sharing intermediate results and relevant information. Many participants have shared intermediate results with their collaborators that they believe would be useful. They performed three types of strategies of sharing: (1) leaving comments under the block that their collaborators were working on to attract their attention - P3; (2) leaving comments before the block of the shared information and pinning their collaborators using an H2/3 "@" - P5, 6; and (3) ask their collaborators to check their current highlights block for reference - P1, 2, 9. The first strategy requires the comment receiver to locate the shared information, while the second strategy may influence the collaboration efficiency. Though many participants communicated directly through Zoom, it *"disturbed my own progress a bit"* -P10. These strategies are either *"inefficient"* -P9 or *"disruptive"* -P7 (**C4**).

Stage 3 — Prompt Modification & Merge Conflicts. The third stage centers on modifying (i.e., engineering) the prompt. In this stage, participants refine their prompts by copying and pasting utterances or code snippets from collaborators' prompts to clarify details about the variable name, resource, methods, and detailed considerations. During this phase, participants may encounter merge conflicts or issues that need communication for resolution. Participants also expressed a desire to access previous versions of the code, as this helps them *"recall who made specific changes to the prompt"* -P4 and the *"reasons behind those alterations"* -P7.

In summary, the user challenges are as follows:

- **C1**: Effort of maintaining group awareness and shared understanding to enhance collaboration effectiveness.
- **C2**: The repetitive effort of syncing with collaborators' work.
- **C3**: Inconvenient and disruptive ways of requesting collaborators' feedback and assistance.
- **C4**: Repetitive copy-pasting effort for leveraging others' work and disruptive information sharing.

4 DESIGN CONSIDERATIONS

Based on the findings from the formative study, we formulated four Design Considerations (**Ds**). to support prompt co-engineering in collaborative NL programming.

D1: Supporting sense-making of collaborators' progress and prompts. Programmers encountered challenges locating collaborators' work in a shared notebook that lacked a clear outline of NL prompts and code snippets (C1). To support programmers'

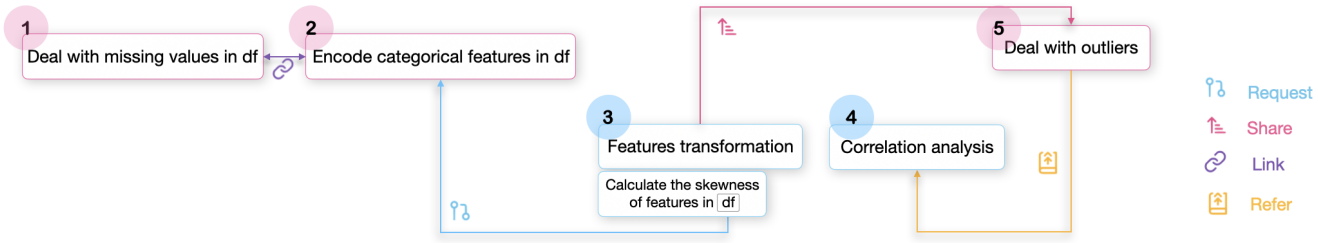

Figure 2: Envisioned Scenario of collaborative NL programming using *CoPrompt*, including tasks from 1 to 5 (pink boxes indicate Alice's tasks, and blue boxes indicate Bob's tasks). Four colors of arrows indicate four types of mechanisms.

locating and sense-making of collaborators' progress, it is important to implement a clearer structure to show the overview of the notebook [18]. This structure should include multiple levels of hierarchy (e.g., tasks, sub-tasks, and prompts) to help programmers identify changes easily. The design should also incorporate sense-making assistive features to help programmers understand the code from the prompt that might be too vague. In addition, the history view should be provided for programmers to track the global activities (e.g., collaborators' works) and local changes (e.g., prompts variations).

D2: Automatic synchronization to reduce repetitive updates. In the exploratory and iterative programming process, the prompts and codes might be updated several times throughout the whole process based on the collaborators' changes (C2). Automatic synchronization of variables and code snippets with procedural dependency should be provided to reduce programmers' cognitive load and enhance their collaboration efficiency. Programmers should be able to easily notice changes made by collaborators and the automatic updates applied to their own work.

D3: Supporting requests for feedback and assistance. The current way of requesting collaborators' feedback and assistance is inconvenient and disruptive (C3). An efficient way of requesting feedback should be provided besides communicating through chat and voice. In addition, considering the situation that the collaborator has not finished the required prompt yet, programmers should be equipped with the function to request knowledge from others and refer to it later.

D4: Reduce effort for sharing knowledge and incorporating others' work. Current ways of referring to collaborators' prompts and reuse are tedious and time-consuming (C4). Programmers need to copy, paste, and modify, which takes a lot of time unnecessarily. A more effortless way of referring to collaborators' prompts should be provided. Additionally, the design should enable proactive sharing of intermediate results with collaborators, promoting sharing without concerns about disrupting their workflow.

5 ENVISIONED SCENARIO

Here, we present a motivating scenario that illustrates the workflow of using *CoPrompt* for prompt co-engineering in NL programming. For simplicity, we describe our scenario using two collaborators, Alice and Bob.

Alice and Bob are remotely collaborating on a data analysis task requiring them to develop a model for predicting house prices. To improve collaboration efficiency, they divide the tasks into smaller segments, allowing each to tackle different tasks separately. They decide that Alice would handle missing values and outliers, as well as encoding categorical features. Meanwhile, Bob is tasked with feature transformation and multi-variable correlation analysis (Figure 2). To track each other's progress and offer/request help for specific tasks, they outline the main tasks and their corresponding sub-tasks in *CoPrompt*'s rich text editor (Figure 3 a), which is synchronously displayed in the multi-hierarchical wiki of *CoPrompt* (Figure 3 b). The wiki's foldable task items provide them a clear overview of task allocation and the collaboration process. After listing all the required tasks, Alice and Bob begin to work on writing their prompts independently.

Although Alice plans to finish her encoding task before Bob begins feature transformation, her progress is delayed due to technical issues. Due to time constraints, Bob cannot wait for Alice to finish her encoding. With *CoPrompt*, Bob decides to create a **request** (Figure 5) from the utterance `df` in the transformation prompt to Alice's encoding prompt, indicating that his feature transformation steps (e.g., calculating the skewness) require Alice's encoded result. Once *CoPrompt* detects that Alice has completed the encoding through semantic analysis and keyword detection, it automatically updates Bob's prompt to leverage the encoded data to generate executable code segments. As a result, Bob no longer needs to manually modify his prompt whenever Alice updates her encoding, saving his time and allowing him to focus more on task completion rather than repetitive code upkeep.

After completing the feature transformation, Bob checks the collaboration progress from the wiki (Figure 3) and notices that Alice needs the transformed data for later outlier handling. In case Alice needs to spend much time determining and verifying the data to be used, Bob decides to proactively share the data with Alice. To share this transformed data, Bob utilizes the **share** mechanism (Figure 6) by highlighting his data frame and clicking on the share icon next to Alice's task item in the wiki. Alice receives a pop-up, allowing her to accept Bob's data without manually locating the data frame for her subsequent workflow. Once accepted, *CoPrompt* automatically regenerates Alice's prompts based on Bob's shared data frame, eliminating the need for Alice to input supplementary information like variable names and sources herself. This reduces the risk of human errors such as typos or incorrect variable references.

After handling both missing values and encoding, Alice anticipates that there may be future adjustments to the missing value

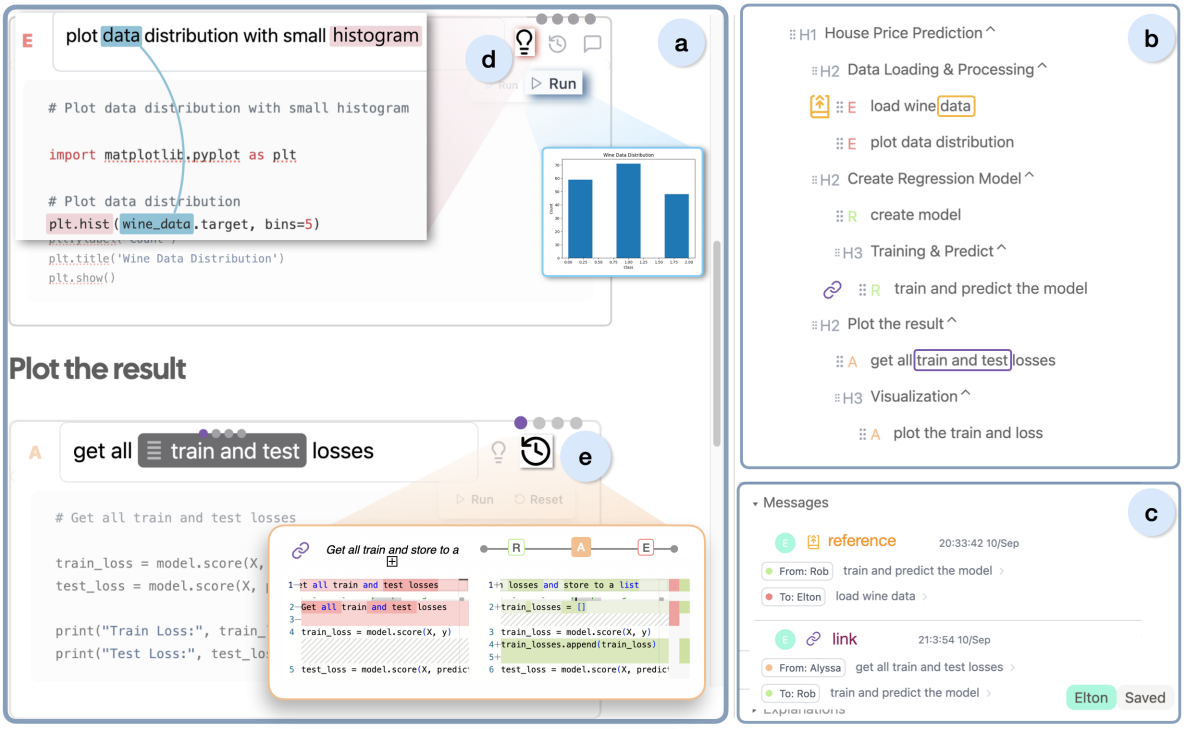

Figure 3: The *CoPrompt* user interface includes (a) a block-based rich text editor for NL inputs, which consists of prompt blocks and execution code blocks. (b) a multi-hierarchy wiki displaying tasks and prompts, (c) a message panel providing a comprehensive log of actions, (d) an explanation view displaying an explanation for prompts and code-prompt relationship, and (e) a history view for version control.

handling based on her past experience, which may also require updates to her encoding methods. To avoid repetitive updating, Alice creates a link (Figure 7) connecting the df in the missing value handling prompt and the df in the encoding prompt. With this link, the encoding prompt gets automatically updated whenever the prompt for missing value handling changes (e.g., when df's variable name or the method for handling missing values changes). This way, Alice avoids the need to repeatedly update the encoding prompt. When handling outliers, Alice needs to refer to the results of Bob's correlation analysis. However, she finds it challenging to navigate through Bob's prompts and results, as many of them contain long execution examples and demonstrative code snippets. To help her better understand Bob's prompts, Alice expands the explanation view (Figure 3d) to see the highlighted prompt utterances and annotated relationships between the NL utterances and code snippets. From the explanation view, Alice understands Bob's considerations for correlation analysis and appropriate criteria for determining outliers. Then, Alice begins writing her prompts to address the outliers. To ensure no information (e.g., criteria for determining outliers) is overlooked, Alice wants to instruct the LLM to determine the outlier handling method based on the results of the correlation analysis. Alice employs the refer mechanism (Figure 4), creating a node that links the outlier handling prompt to the correlation analysis prompt. As a result, *CoPrompt* updates Alice's prompts with appropriate methods, no longer requiring Alice to modify her prompts by copy-pasting and typing manually.

6 DESIGNING COPROMPT

Based on our design considerations, we developed a prototype, *CoPrompt*, to support programmers in their prompt engineering workflow during collaborative NL programming. Specifically, *CoPrompt* consists of five UI components. The **block-based rich text editor** (Figure 3 a) allows programmers to input various block-based components in markdown format, providing them with the flexibility to freely organize their tasks and prompts. These components include different levels of hierarchy (i.e., H1-H6) to represent tasks, sub-tasks, NL prompt blocks, and executable code blocks. The **prompt wiki** (Figure 3 b) displays natural language components in the rich text editor in a table-of-content structure and allows programmers to track other collaborators' workflow. The **message panel** (Figure 3 c) displays detailed information of each interaction, including timestamp, link content, generated code result, link creator, and receiver. The **explanations view** (Figure 3 d) displays explanations of the prompt by showing the matching between prompt utterances and corresponding code segments generated. The **history view** (Figure 3 e) allows programmers to view the change history of a specific prompt.

6.1 Sense-Making and Tracking of the Collaboration Process

CoPrompt is designed with a set of essential real-time collaborative features, such as real-time collaborators' cursor location and text selections. The system features a real-time, block-based rich text editor that supports various markdown elements and hierarchical levels. Additionally, it introduces two custom block types: the **Prompt Block**, for creating NL prompts that generate code segments, and the **Execution Code Block**, which contains generated code that can be compiled and executed to check the interim results.

As changes are made in the text editor, the wiki view automatically updates to display the document structure with a tree-based representation. This structure provides a clear overview of the document, allowing programmers to visualize multiple levels of hierarchy, from headings to NL prompts, down to individual *nodes* (i.e., phrases within the prompts). The Wiki's hierarchy view allows for intuitive navigation of its content by enabling programmers to click and fold each task item, helping them collapse unrelated tasks and concentrate on those of interest. Furthermore, it provides a clear overview of the hierarchical relationships within the project and keeps programmers updated about any new sub-tasks or modifications made by their collaborators (**D1**).

Message Panel. The message panel (Figure 3 c) offers programmers a comprehensive log of actions, allowing them to track and review their collaborative activities. These actions are tied to specific elements within the document, such as prompts, headings, or nodes, which can all be easily accessed by selecting them directly from the messages. Additionally, programmers can quickly identify essential information related to each action, including its type, creator, and timestamp. Messages are displayed in chronological order, with unprocessed actions, such as pending requests, prioritized at the top and highlighted by a small dot. Once addressed, *CoPrompt* automatically updates the panel by removing the highlighting dot to indicate that the action has been resolved. This feature provides a visual representation of the action within the Wiki view.

Explanation View. The explanation view allows programmers to see detailed explanations of the code as well as the relationship between prompts and the generated code. By clicking the *Explain* button (Figure 3 d) for the prompt block, programmers can access semantic highlighting to better understand the code structure. This feature visually highlights key phrases in both the NL prompt and the corresponding code segments, linking them to help programmers understand the relationships between them (i.e. which phrase in the prompts led to the generation of a certain line of code). The view also provides a high-level overview of all the steps within the code, allowing programmers to quickly understand the code's structure, logic, and functionality.

History View. The history view (Figure 3 e) enhances version control and tracking of changes for prompts and their associated code segments. Programmers can review previous versions of their selected prompts, organized chronologically by modification timestamps. The view tracks the evolution of prompts, providing details such as the actions that led to changes, the individuals responsible for modifications, and specific alterations made to the prompts. Additionally, similar to Git version control, *CoPrompt* offers a diff-view that allows programmers to easily identify differences between current and previous versions of both prompts and code segments.

6.2 Supporting Programmers' Prompt Co-engineering

To support the prompt co-engineering workflow, *CoPrompt* first provides some functions commonly used in collaboration interfaces (e.g., annotation and history view) to help programmers make sense of their collaborators' prompts. *CoPrompt* also provides four mechanisms for programmers to (1) refer to collaborators' prompts, (2) request intermediate results from collaborators, (3) share information with collaborators, and (4) link variables for synchronization.

6.2.1 Making sense of collaborators' prompts. In order to leverage collaborators' prompts, programmers first need to make sense of them. To view the prompt's ownership, programmers can check the left of each prompt where the prompt author's icon is displayed, indicating the last person who modified the prompt. To assist programmers in comprehending high-level or more complex prompts from collaborators, we designed the *explanations view*. In this view, the relationship between the utterances in the prompts and the generated code snippets is annotated by different colors of highlights (**D2**). By hovering on the utterance nodes, programmers can see the corresponding code snippets highlighted and a link between the utterance and code. To help programmers track the evolution of prompts and their previous versions, we designed a *history view* where they can find historical versions of both the prompts and the corresponding code.

6.2.2 ***Refer*** *to Collaborators' Prompts for Precise Code Generation.* When programmers need to build upon their collaborators' work, they can use the refer mechanism (Figure 4) to specify which part of collaborators' work should be used to provide the context of their own prompt for *CoPrompt* to generate more precise code (**D3**).

All four mechanisms contain three components: (1) the node in the prompt blocks that are presented in the editor; (2) the node in the prompt that is listed in the *Prompt Wiki* and (3) the additional comments. To use the refer mechanism, the programmer needs to first select a node from the prompt in the editor (*source node*), which could be a word (e.g., a variable name), a phrase (e.g., perform a function with a variable), or the whole prompt. Upon selection, the editor interface will toggle the highlighted mark on the selected node. Next, the programmer chooses a *target node* from the list of prompts displayed in the *Prompt Wiki*. This target node could be a word, a phrase, or the entire prompt. Once selected, the *Prompt Wiki* panel will display the corresponding colored icon based on the current actions associated with the node. Additionally, a colored dot will be added before the prompt, signifying that this particular prompt contains nodes with associated actions. Programmers may also opt to include comments or messages to clarify their intentions, thereby improving the accuracy of code generation and providing context for collaborators to understand the reasoning behind the prompt.

6.2.3 ***Request*** *Collaborators' Assistance for Prompt Engineering.* When programmers require information from their collaborators' incomplete tasks as context for their own prompt, they can follow

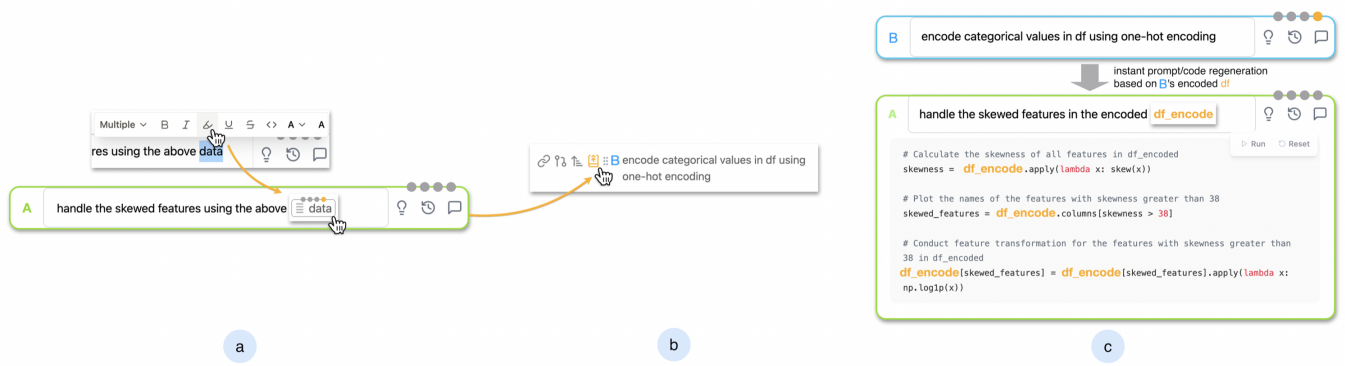

Figure 4: Workflow of refer: (a) select and highlight prompt utterances that should be modified from the editor as a source node; (b) select the target prompt for reference from the wiki as a target node and click the refer icon; (c) A's block will instantly regenerate prompt and code based on the code and execution result of B's prompt.

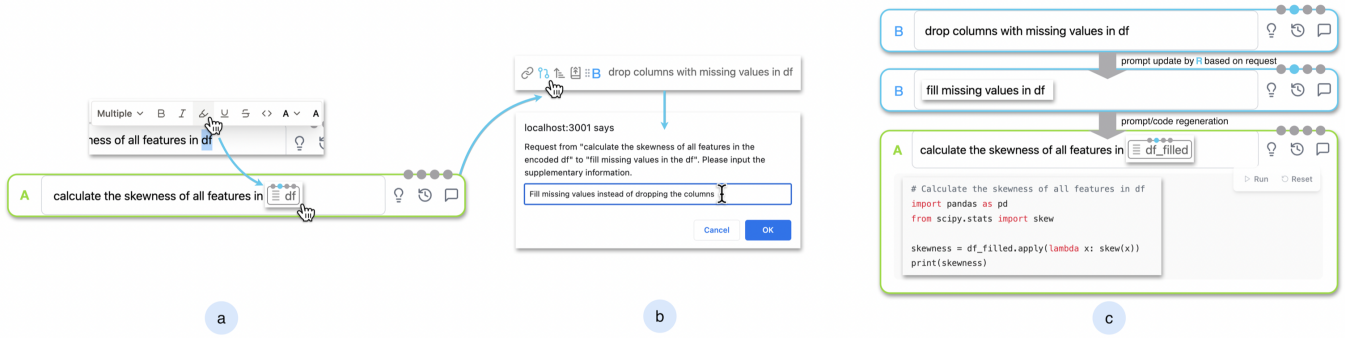

Figure 5: Workflow of request: (a) select and highlight prompt utterances that should be modified from the editor as a source node; (b) select a target prompt of the task that needs the collaborator to finish as a target node and share the result from wiki and click the request icon, fill in a descriptive message for the request; (c) when B updates how they deal with missing values, A's block will regenerate prompt and code accordingly.

a simple process with the *Request* feature supported by *CoPrompt*. First, they create a prompt with a placeholder indicating the expected result from their collaborators' work. Next, they select a *source node* from the placeholder in the editor and choose a *target node* from the collaborator's work listed in the *Prompt Wiki*. This action notifies the creator of the targeted node that a collaborator is awaiting the task to be resolved. Subsequently, the programmer can continue working on other tasks and return to check on this specific task once the collaborator has finished the requested task (**D3**). After creating the request, the system will log the actions in the history and maintain a list of unresolved actions in the cache. When collaborators create prompt blocks that address the actions in the cache, *CoPrompt* will automatically execute the corresponding task (**D2**). For example, if Alice creates a request like *"Return me a new dataframe after removing outliers," CoPrompt* will log the request and place it in the cache. When Bob subsequently creates the prompt *"Remove outliers from the data," CoPrompt* will automatically update Alice's work and mark the task as completed.

6.2.4 ***Share*** *Context to Collaborators.* To share intermediate results or other contexts with collaborators, programmers can employ the share mechanism (Figure 6). This is achieved by choosing a *source node* in the editor and selecting a *target node* in the *Prompt Wiki*. Once this action is taken, the collaborator will receive a pop-up notification indicating that shared context is available. Upon accepting the context, the code associated with the prompt that contains the *target node* will be updated with the context and information provided by the source node. Programmers can also select the *target node* as one of the headings that correspond to a task, especially when there are no existing suitable prompt blocks under that task. In such cases, the recipients can assign this contextual information to the prompt block they create afterwards.

6.2.5 ***Link*** *Elements for Automatic Synchronization.* In a programming context, a variable, prompt, or code segment within a project may undergo multiple changes and could have various names in blocks authored by different programmers. To reduce repetitive updates, programmers can create links to select variables with the same value but different names and link them together. To establish

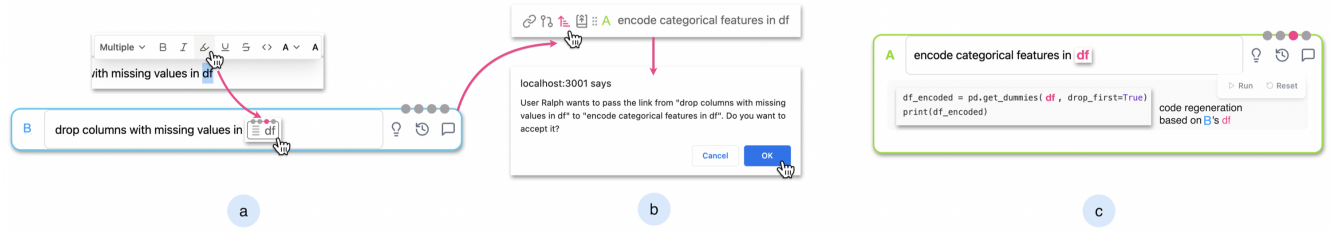

Figure 6: Workflow of share: (a) select and highlight the element to be shared as a source node from the editor; (b) select the prompt to be updated with the shared content from the wiki as a target node and click the share icon to highlight the node, then A will see a pop-up message indicating that the collaborator B would like to share some information with A; (c) when A accepts, A's highlighted block will update its code based on B's df.

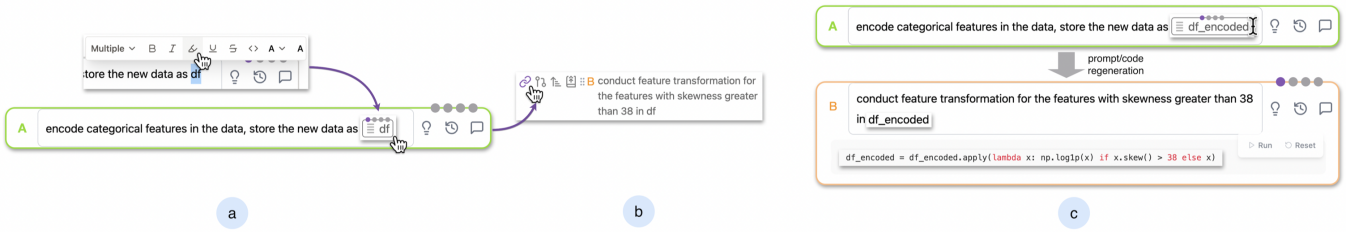

Figure 7: Workflow of link: (a) select and highlight the element to be synced as a source node from the editor; (b) select target prompt that is to be synced with the highlighted element from the wiki as a target node and click the link icon; (c) when A updates the name of the df, B's block will regenerate prompt/code accordingly.

the link, programmers can choose a variable as the *source node* from the prompt in the editor and select another variable from the prompt in the wiki as the *target node*. By doing this, when the prompt linked to one variable is modified, the associated prompt will automatically update based on the changes made to the code. This mechanism also extends to variables with procedural dependencies. It not only streamlines future code generation but also helps prevent potential conflicts (**D4**).

6.3 System Implementation

CoPrompt is a web-based application built using Next.js and React TypeScript. The core of the block-based editor is constructed using TipTap [76], which serves as a headless wrapper for ProseMirror [26], providing the foundation for a rich text WYSIWYG editor. Real-time collaborative capabilities are facilitated through WebSocket techniques, enabling seamless collaboration among users. Actions and Messages are shared in real-time using Firebase Realtime Database [22].

To generate code, the system leverages the OpenAI GPT-4 API [59], while code execution in Python is made possible through Pyodide [66]. Pyodide represents a port of CPython to WebAssembly, enabling the installation and execution of Python packages directly within the browser using micropip. For the execution of Python code in a separate thread, a communication channel is established between the main thread and the Pyodide worker, incorporating a defined communication protocol. It's worth noting that, despite sharing the same context within the collaborative editor, the system ensures that the OpenAI and Python kernels do not overlap, preserving stability and functionality.

7 USER STUDY

We conducted a user study to evaluate the effectiveness of *CoPrompt* in assisting programmers' prompt co-engineering during NL programming by answering the following research questions:

- **RQ1**: How does the system support programmers to understand collaborators' progress and prompts?
- **RQ2**: How does the system support programmers to build on top of collaborators' work?
- **RQ3**: How does the system minimize the redundant updates of prompts or code?

7.1 Participants

We recruited 12 participants (7 female, 5 male, aged 20-31) from a local university. All participants have more than two months of experience in AI-based code assistants, and all have more than three years of experience in using computational notebooks. Participants were compensated $50 for the 120-minute study.

7.2 Procedure

Participants were first informed of the aim of this study and gave their consent. Then, they were asked to participate in a two-part

study including (1) real-time collaboration and (2) following up with the others' work.

7.2.1 Part 1: Real-time collaboration (90 minutes). Participants were asked to perform a data analysis task in pairs using NL prompts collaboratively. Each pair of participants was asked to complete two sessions of collaborative programming: one session to use our *CoPrompt* prototype and another to use the baseline system (VSCode live share with CoPilot plugin). The sequence of the sessions was counter-balanced.

For each session, the experimenter first introduced the features of the system and gave each participant a 10-minute training session on the system, with example tasks to complete. Then, each pair of participants was asked to work on a data science task collaboratively by prompting for around 30 minutes. After finishing the task, participants were asked to rate their experience of *CoPrompt* and the baseline system on a 7-point Likert scale. The experimenters then conducted a semi-structured interview based on the results and observed use patterns to learn participants' perspectives on *CoPrompt*. The data science tasks were modified from a Kaggle competition [12].

7.2.2 Part 2: Following up with the Collaboration Process (30 minutes). In stage 2, we evaluated how a new collaborator followed up with an ongoing collaborative project using *CoPrompt*. We asked participants to review the work of another pair of participants collected from part 1 of the study using *CoPrompt* (e.g., P1 reviewed the work of the second pair - P3 and P4). After reviewing the work for 10 minutes, we asked participants to complete additional tasks to modify the existing work, such as changing the way of dealing with outliers. The experimenters then conducted a semi-structured interview regarding the user experience of reviewing and revising others' work using *CoPrompt*.

7.2.3 Thematic Analysis. All study sessions were recorded and transcribed. We collected data from server-side usage logs, screen recordings, and follow-up interviews. We also took observational notes during the study. Two authors coded the transcripts and generated themes to understand how participants used and assessed *CoPrompt*.

8 RESULTS

Here we report the findings from analyzing participants' survey responses, interview transcripts, think-aloud feedback, and system usage logs to understand 1) how *CoPrompt* support prompt co-engineering and 2) how participants perceive the utility of the four mechanisms for leveraging collaborators' work and sharing information among collaborators.

8.1 Overall Collaboration Behaviors and User Perceptions

All participants successfully completed the programming tasks and utilized all four types of mechanisms available in the study. Notably, participants engaged in more collaborative workflows when using the *CoPrompt* prototype compared to when using the VSCode editor (baseline). In both conditions, all participants initiated their work by crafting prompts at a higher level of abstraction, such as defining the task as *"data visualization,"* and then iteratively refined these prompts to reach a lower level of detail, like specifying *"pair plot df."* This process was facilitated by the Wiki view provided by *CoPrompt*, which allowed participants to easily locate the target prompt block that was available for further iteration. The four mechanisms further supported participants in collaborative efforts without requiring context switching to external communication tools. P4 explained, *"Using CoPrompt, I do not need to wait for my collaborator to finish encoding, as I can write prompts for transformation first and then refer to my collaborator's prompt."* In general, *CoPrompt* facilitated parallel work on programming tasks without being hindered by collaborators' workflows.

To measure the usability of *CoPrompt*, we computed the SUS scores based on the UMUX-LITE [39]. The average SUS scores were significantly greater ($p = 0.02$) for *CoPrompt* (Mdn = 90.61), compared to baseline (Mdn = 68.94). We also used NASA-TLX to measure participants' perceptions of the workload of using the systems. Compared to baseline, *CoPrompt* had lower mental (Mdn = $3.5 < 5.5, p = 0.040$), physical (Mdn = $1.0 < 3.0, p = 0.0179$), and temporal (Mdn = $3.0 < 5.0, p = 0.0082$) demand, required less effort (Mdn = $3.0 < 5.5, p = 0.033$), and led to better performance (Mdn = $4.5 > 3.0, p = 0.0532$) and less frustration (Mdn = $1.5 < 3, p = 0.0187$). The overall perceived workload, obtained by averaging all six raw NASA-TLX scores (with the "Performance" measure inverted), was also lower for *CoPrompt* than baseline (Mdn = $2.5 < 4, p = 0.0532$).

8.2 RQ1: How does the system support programmers to understand collaborators' progress and prompts?

CoPrompt supported programmers' sense-making of collaborators' work by providing the hierarchical overview, generating explanations associated with the prompts, and displaying historical views. In the following sections, we report the detailed usage and perceived utility of these features.

*8.2.1 The holistic overview with multi-levels of details facilitates users tracking and locating collaborators' processes **(D1)**.* All participants found that the Prompt Wiki reduces the need for programmers to keep track of collaborators' progress (Mdn = $6.5 > 2, p = 0.0034$) and helps them understand what the collaborators are doing (Mdn = $5.5 > 3, p = 0.0034$),). The table-of-content-like view allowed participants to view all the sections of the notebook at a higher level by folding the levels below the task level, which facilitated participants in locating their target regions and *"understand the overall structure easily."* -P4 With the navigation function, participants could navigate to their target regions by clicking on the items in the Prompt Wiki, which is straightforward as *"there is no need to scroll back and forth."* -P1 P5 also mentioned that the Prompt Wiki allowed them to see collaborators' new progress directly from the Wiki panel. P2 expressed their preference for the annotation in the Wiki as it indicated the owner of the prompt: *"With the icon before the prompt, the ownership of the prompt and code is clear."* We also noticed that the majority of participants (N=9) collapsed their own sections while leaving the collaborator's sections expanded to *"track on collaborators' work"* -P4 and *"easily identify changes not made by me [participant]."* -P11

*8.2.2 Generating and associating explanations assisted participants' comprehension of prompts **(D2)**.* All participants found that the system helps them understand the prompt written by collaborators (Mdn = 5 > 2.5, $p = 0.0304$). They mentioned that the high-level, step-by-step explanations of the generated code and the semantic connections between prompts and code segments helped them understand their collaborators' prompts. This feature is *"especially useful when the prompt or the code is way too long."* -P1

Participants typically required these explanations before utilizing the **refer** feature, as sometimes struggled to fully grasp their collaborators' intentions through the prompts alone. The prompt explanations played a vital role in scaffolding participants' comprehension by *"providing more details about the code."* -P1 Consequently, participants could reuse or reference their collaborators' work without the need for direct communication, as highlighted by one participant, *"I do not need to keep bothering my partner to ask what the code is about."* -P9 Furthermore, some participants perceived these explanations as a means to reduce conflicts and errors during collaboration. As one participant noted, *"I became less likely to misunderstand my collaborators' intention and modify their code, which always causes conflicts."* -P1

*8.2.3 Providing a history view allows users to have a clear view of the changes **(D1)**.* The history view presents a comprehensive record of all historical versions of a specific prompt, offering users insight into its evolving process, as noted by P5, *"It is good to have a history view showing the changes being made."* This view not only captures the modifications made but also documents the *"interactions that happened"* -P1 and the *"generated result for each iteration"* -P2, thus facilitating a holistic understanding of the prompt's evolution. Participants frequently utilized this feature when changes were enacted after one of the four mechanisms had been activated (e.g., the requested message had been resolved). In such cases, the history view allowed them to identify *"who had altered my code."* -P2 Most participants (N=8) also leveraged the diff view of the code and prompt to resolve the conflict and restore the version. Overall, participants reported that the system supports programmers in reaching common ground with collaborators (Mdn = 5 > 2, $p = 0.0034$).

Interestingly, participants also employed the history view to track their collaborators' progress, using it to check if their partners had *"start work on my requests."* -P12 Moreover, the historical view also helped participants when refining their own prompts. For instance, P7 mentioned, *"the history view helped me recall what or why I had made certain changes to my prompt."*

8.3 RQ2: How does the system support programmers to leverage collaborators' work?

All participants agreed that the four types of mechanisms are (1) easy to use: refer (Mean = 5.75, SD = 1.76), request (Mean = 5, SD = 1.35), share (Mean = 5.5, SD = 1.45) and link (Mean = 5.83, SD = 1.27). (2) intuitive to learn: refer (Mean = 5.67, SD = 1.15), request (Mean = 4.91, SD = 1.68), share (Mean = 5.42, SD = 1.31) and link (Mean = 5.66, SD = 1.30). and (3) could fulfill their requirements: refer (Mean = 5.5, SD = 1.68), request (Mean = 5.17, SD = 1.53), share (Mean = 5.33, SD = 1.61) and link (Mean = 5.75, SD = 1.36). Overall, *CoPrompt* reduces participants' need to communicate with collaborators (Mdn = 5 > 3, $p = 0.0122$) and facilitates participants to modify their prompt with ease (Mdn = 5.5 > 2.5, $p = 0.0065$).

*8.3.1 Request and auto-updates reduce mental load **(D3)**.* *CoPrompt* helps participants to convey their needs clearly to the collaborator (Mdn = 5.5 > 2.5, $p = 0.0122$) and reduce the need to *"remember what tasks have not yet been done."* -P9 In collaboration, since the task distribution may involve procedural dependencies, there are often cases where participants need to wait for their collaborators to complete certain tasks. With the request-detect-update mechanism, participants just need to send the request with brief descriptions. The hierarchical structure also allows them to request tasks without waiting for others to finish. They could request tasks at higher-level headings if a suitable prompt had not yet been created by another collaborator.

Compared to communicating with collaborators using messages or audio in baseline, the requested link takes less cognitive effort since *"it provides contextual information through the node"* -P3 and does not request programmers to *"write too much text in NL to describe the location of changes."* -P2 Some participants (N=4) highlighted that this feature reduces another cognitive load, as they no longer need to carefully craft messages to their collaborators, ensuring a polite tone, *"I do not need to care about the manner."* -P3

In addition, the auto-update of the prompts sending requests after target input is detected from collaborators offloads the tedious review and update process. P9 and P10 expressed their satisfaction with these features: *"Detection and auto-update saved me a lot of effort."* -P10 P9 added, *"I only need to know what the task is about, without thinking about where and when to address it."* However, the automatic update feature makes P12 feel a slight loss of control, and sometimes they would like to decide whether to handle updates manually or automatically.

*8.3.2 Sharing knowledge both proactively and reactively **(D4)**.* Participants passed output or prompt segments using the share mechanism both proactively and reactively. Participants utilized the share mechanism for both proactive and reactive sharing of output or prompt segments. Proactively, some participants shared their results with collaborators, *"I know the encoded result would be used for any following step."* -P1 They also shared insights gained from the result through share, *"I would share some text that may not necessary is the prompt to share insights I got."* -P7 Reactively, participants responded to their collaborators' requests for specific data processing results sent through direct communication by sharing the requested results. Participants also made use of the hierarchical structure to share elements by placing them under relevant headings. This occurred when collaborators had not yet delved into those specific details.

For the receiver of the shared content, most participants indicated that the shared content was useful for clarifying their own prompts and the pop-up message was clear and thorough enough for their quick comprehension of the content, *"very convenient as I do not need to find and refer to my collaborators' work."* -P5 Participants also revealed their trial-and-error strategies for dealing with the pop-up message that is too brief to understand or too long to read, *"Just like using ChatGPT, I just accept the answer and view its execution result to evaluate its effect."* -P3 This strategy was adopted

by many participants (N=9), as the trial-and-error cost is low with the view of the history version and the *"ability to trace back to previous versions."* -P4

*8.3.3 Referencing reduces the effort of carefully reading and copy-pasting **(D4)**.* *CoPrompt* helps participants build on top of collaborators' work easily (Mdn = 6 > 3, $p = 0.0049$). All participants used the refer feature more than once when using *CoPrompt* to perform prompt co-engineering. With the refer mechanism, participants no longer need to read the whole prompt and select utterances for copy-pasting to modify their own prompts. Instead, they handed off the comprehension work to *CoPrompt* by guiding it with the refer mechanism. The multi-level hierarchy display of prompts supported programmers in pinpointing the prompts they wished to refer to and enabled them to select the most appropriate level for reference. All participants agreed that the interaction process of refer *"reduced the cognitive switching between communication and code."* -P5

8.4 RQ3: How does the system reduce the repetitive updates of prompts or code?

The link mechanism effectively reduced the frequency of repetitive updates by offering automatic synchronization **(D2)**. Moreover, *CoPrompt* significantly reduced the necessity for participants to engage in repetitive and iterative prompt modifications (Mdn = 5 > 2, $p = 0.0068$).

The link mechanism was deemed the most intuitive by all participants, and they unanimously agreed that it significantly reduced the workload associated with repetitive updates due to procedural dependencies. P4 expressed that it *"saved a lot of time on keep going back and forth between users"* and *"helped to offload some mental model"* -P5 without the need to keep track of collaborators' changes on a certain task. Additionally, three participants employed links between headings to convey the synchronization of an entire subsection.

All participants mentioned that the refer and share mechanism simplifies the interaction of updating prompts. When there is a need to update existing prompts, many participants (N=9) leveraged refer so that they just needed to indicate the part of the prompt that requires modification and the target prompt reference. The share mechanism assisted programmers' prompt engineering by allowing collaborators to pass knowledge to others and it only requires receivers to accept the shared prompt or code snippets and coarsely navigate to the target prompt, instead of *"carefully locating and manual copy-pasting."* -P7

9 DISCUSSION

9.1 Sense-making and awareness in collaborative NL programming

Our design facilitated the sense-making of the overall collaboration progress and collaborators' work by displaying tasks and prompts in a multi-hierarchy wiki and providing explanations and historical versions of prompts. The multi-hierarchy wiki design is inspired by the user needs for high-level ideas of collaborators' work instead of numerous details [9, 82]. Prior work facilitating sense-making of collaborators' work leverages comments [80] and documentation [81]. Our design provided additional information by annotating relationships between prompt utterances and code snippets, and other explanations in the explanation view. All participants found this explanation view helpful, while they also proposed additional information they would like to know, such as explanations of the changes made to the prompts and code.

9.2 Prompt Co-engineering

9.2.1 More nuanced task distribution. The four mechanisms for leveraging collaborators' work and the multi-hierarchy wiki enabled participants to distribute the tasks into smaller sub-tasks. Prior work mainly investigated the collaboration styles of single authoring, pair authoring, divide and conquer and competitive authoring [65, 80]. Our proposed mechanisms lowered the cost of merging task results and leveraging others' output. This enabled the workflow to be distributed equally so that every collaborator could program at the same time.

9.2.2 Interruptions and conflicts handling. The context of collaborative programming brought specific challenges to prompt engineering, such as difficulty in maintaining group awareness in a fast-changing workspace [15] and interruptions of collaborators' workflow [16]. Our share and request mechanisms involve data flow between two programmers and auto updates of prompts. These mechanisms enabled programmers to leverage collaborators' work without interrupting them, while this may also significantly increase the chance of conflicts and requirements for further modifications. We provided convenient history views and explanation views for programmers to review and discuss solutions for dealing with conflicts.

9.2.3 Level of Automation. The interaction between humans and AI has been investigated in many domains, such as qualitative coding [19, 20] and programming [81]. In our design, the prompt co-engineering process involves not only human collaborators but also the LLMs to interpret collaborators' intentions. The four mechanisms for leveraging collaborators' prompts involve a high level of automation [52] as LLMs are in charge of comprehending. This design significantly reduced the programmers' cognitive load, while it also raised concerns about the generated prompts and code: How to make the generation more satisfying and how to efficiently deal with unsatisfying results. We provided convenient history views for each prompt for programmers to check and roll back. In the future, more mechanisms should be designed to provide a better user experience in dealing with unsatisfying results.

9.3 Limitations and Future Work

Our study of the *CoPrompt* presents important findings in the realm of collaborative NL programming. However, there are certain limitations that should be recognized.

Participant Demography: The *CoPrompt* was exclusively tested with programmers experienced in using LLM-driven code assistants. While NL programming is beneficial to a wide range of expertise levels and promises a no-code experience, our pilot study indicated challenges for programmers with limited experience. Such participants often found it difficult to refine prompts effectively and were less active in the collaborative process. Future studies

should widen the participant demography, including programmers unfamiliar with AI-driven code assistants and end-user programmers. It would be insightful to understand how varying expertise levels influence the approaches to writing NL prompts and their collaborative workflows.

Scope of Experiment Tasks: We limited our experimental tasks to data analysis due to its intrinsically collaborative nature and the rapid iterations it requires. Although this limitation might narrow the insights into collaborative NL programming across different scenarios, our introduced mechanisms and workflow are generalizable in addressing common challenges prevalent in various collaborative programming contexts. Future work can explore either specific support for data scientists or even expand to different programming tasks, such as web development. Another avenue for future research would be long-term deployment studies. Such studies can shed light on evolving usage patterns, potential issues, strategies for prompt co-engineering, and maintenance in collaborative NL programming.

Design Aspects: Our research stands as an initial step in designing systems that foster collaborative NL programming workflows. Our findings detail the usage of our core mechanisms, emphasizing support with an amalgamation of existing designs. Our aim was not to make groundbreaking contributions in visualization or UI design. Nevertheless, future endeavours could delve deeper into design specifics, exploring how visualization tools might further enhance programmers' comprehension and integration of other relevant information.

10 CONCLUSION

In this work, we investigated the potential workflow of using NL prompts to conduct collaborative programming, especially collaborative prompt engineering. Our formative study revealed the workflow of prompt co-engineering and identified four challenges related to comprehension, synchronization, feedback, and reference of collaborators' prompts. To tackle these challenges, we introduced *CoPrompt*, a prototype to support prompt co-engineering through four novel mechanisms: share, refer, request, and link. Our user study indicated that *CoPrompt* effectively supported programmers' workflow of prompt co-engineering from comprehending collaborators' work to leveraging it and sharing context information.

REFERENCES

[1] 2023. Copilot: Your AI pair programmer. https://github.com/features/copilot

[2] DataCanary Anna Montoya. 2016. House Prices - Advanced Regression Techniques. https://kaggle.com/competitions/house-prices-advanced-regression-techniques

[3] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2023. Prompting Is Programming: A Query Language for Large Language Models. *Proc. ACM Program. Lang.* 7, PLDI, Article 186 (jun 2023), 24 pages. https://doi.org/10.1145/3591300

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

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

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

[7] Souti Chattopadhyay, Ishita Prasad, Austin Z. Henley, Anita Sarma, and Titus Barik. 2020. What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3313831.3376729

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[9] Yan Chen, Sang Won Lee, Yin Xie, YiWei Yang, Walter S. Lasecki, and Steve Oney. 2017. Codeon: On-Demand Software Development Assistance. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 6220–6231. https://doi.org/10.1145/3025453.3025972

[10] KR1442 Chowdhary and KR Chowdhary. 2020. Natural language processing. *Fundamentals of artificial intelligence* (2020), 603–649.

[11] Anamaria Crisan, Brittany Fiore-Gartland, and Melanie Tory. 2021. Passing the Data Baton : A Retrospective Analysis on Data Science Work and Workers. *IEEE Transactions on Visualization and Computer Graphics* 27, 2 (2021), 1860–1870. https://doi.org/10.1109/TVCG.2020.3030340

[12] Will Cukierski. 2012. Titanic - Machine Learning from Disaster. https://kaggle.com/competitions/titanic

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

[14] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. 2005. A Study of the Documentation Essential to Software Maintenance. In *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information* (Coventry, United Kingdom) *(SIGDOC '05)*. Association for Computing Machinery, New York, NY, USA, 68–75. https://doi.org/10.1145/1085313.1085331

[15] Paul Dourish and Victoria Bellotti. 1992. Awareness and coordination in shared workspaces. In *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*. 107–114.

[16] Hongfei Fan, Jiayao Gao, Hongming Zhu, Qin Liu, Yang Shi, and Chengzheng Sun. 2017. Balancing Conflict Prevention and Concurrent Work in Real-Time Collaborative Programming. In *Proceedings of the 12th Chinese Conference on Computer Supported Cooperative Work and Social Computing* (Chongqing, China) *(ChineseCSCW '17)*. Association for Computing Machinery, New York, NY, USA, 217–220. https://doi.org/10.1145/3127404.3127447

[17] Hongfei Fan, Chengzheng Sun, and Haifeng Shen. 2012. ATCoPE: Any-Time Collaborative Programming Environment for Seamless Integration of Real-Time and Non-Real-Time Teamwork in Software Development. In *Proceedings of the 2012 ACM International Conference on Supporting Group Work* (Sanibel Island, Florida, USA) *(GROUP '12)*. Association for Computing Machinery, New York, NY, USA, 107–116. https://doi.org/10.1145/2389176.2389194

[18] 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* (Hamburg, Germany) *(CHI EA '23)*. Association for Computing Machinery, New York, NY, USA, Article 235, 7 pages. https://doi.org/10.1145/3544549.3585737

[19] Jie Gao, Yuchen Guo, Gionnieve Lim, Tianqin Zhang, Zheng Zhang, Toby Jia-Jun Li, and Simon Tangi Perrault. 2023. CollabCoder: A GPT-Powered Workflow for Collaborative Qualitative Analysis. arXiv:2304.07366 [cs.HC]

[20] Simret Araya Gebreegziabher, Zheng Zhang, Xiaohang Tang, Yihao Meng, Elena L. Glassman, and Toby Jia-Jun Li. 2023. PaTAT: Human-AI Collaborative Qualitative Coding with Explainable Interactive Rule Synthesis. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 362, 19 pages. https://doi.org/10.1145/3544548.3581352

[21] R Stuart Geiger, Nelle Varoquaux, Charlotte Mazel-Cabasse, and Chris Holdgraf. 2018. The types, roles, and practices of documentation in data analytics open source software libraries: a collaborative ethnography of documentation work. *Computer Supported Cooperative Work (CSCW)* 27, 3-6 (2018), 767–802.

[22] Google. 2023. Firebase is an app development platform that helps you build and grow apps and games users love. Backed by Google and trusted by millions of businesses around the world. https://firebase.google.com/

[23] 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.

[24] Carl Gutwin and Saul Greenberg. 1998. Effects of awareness support on groupware usability. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 511–518.

[25] Carl Gutwin and Saul Greenberg. 2002. A Descriptive Framework of Workspace Awareness for Real-Time Groupware. *Computer Supported Cooperative Work (CSCW)* 11, 3-4 (Sept. 2002), 411–446. https://doi.org/10.1023/a:1021271517844

[26] Marijn Haverbeke. 2023. A toolkit for building rich-text editors on the web. https://prosemirror.net/

[27] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. 2019. Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3290605.3300500

[28] Austin Z. Henley and Scott D. Fleming. 2014. The Patchworks Code Editor: Toward Faster Navigation with Less Code Arranging and Fewer Navigation Mistakes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Toronto, Ontario, Canada) *(CHI '14)*. Association for Computing Machinery, New York, NY, USA, 2511–2520. https://doi.org/10.1145/2556288.2557073

[29] James D Herbsleb, Helen Klein, Gary M Olson, Hans Brunner, Judith S Olson, and Joe Harding. 1995. Object-oriented analysis and design in software project teams. *Human–Computer Interaction* 10, 2-3 (1995), 249–292.

[30] Di Huang, Ziyuan Nan, Xing Hu, Pengwei Jin, Shaohui Peng, Yuanbo Wen, Rui Zhang, Zidong Du, Qi Guo, Yewen Pu, and Yunji Chen. 2023. ANPL: Compiling Natural Programs with Interactive Decomposition. arXiv:2305.18498 [cs.PL]

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

[32] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large Language Models Meet Program Synthesis. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1219–1231. https://doi.org/10.1145/3510003.3510203

[33] Ellen Jiang, Kristen Olson, Edwin Toh, Alejandra Molina, Aaron Donsbach, Michael Terry, and Carrie J Cai. 2022. PromptMaker: Prompt-Based Prototyping with Large Language Models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI EA '22)*. Association for Computing Machinery, New York, NY, USA, Article 35, 8 pages. https://doi.org/10.1145/3491101.3503564

[34] Mira Kajko-Mattsson. 2005. A survey of documentation practice within corrective maintenance. *Empirical Software Engineering* 10 (2005), 31–55.

[35] Amy J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. 2011. The State of the Art in End-User Software Engineering. *ACM Comput. Surv.* 43, 3, Article 21 (apr 2011), 44 pages. https://doi.org/10.1145/1922649.1922658

[36] Kirby Kuznia, Swaroop Mishra, Mihir Parmar, and Chitta Baral. 2022. Less is More: Summary of Long Instructions is Better for Program Synthesis. arXiv:2203.08597 [cs.CL]

[37] Andrew K. Lampinen, Ishita Dasgupta, Stephanie C. Y. Chan, Kory Matthewson, Michael Henry Tessler, Antonia Creswell, James L. McClelland, Jane X. Wang, and Felix Hill. 2022. Can language models learn from explanations in context? arXiv:2204.02329 [cs.CL]

[38] J Chris Lauwers and Keith A Lantz. 1990. Collaboration awareness in support of collaboration transparency: Requirements for the next generation of shared window systems. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 303–311.

[39] 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

[40] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, and Cuiyun Gao. 2023. CCTEST: Testing and Repairing Code Completion Systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1238–1250. https://doi.org/10.1109/ICSE48619.2023.00110

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

[42] Hugo Liu and Henry Lieberman. 2005. Programmatic Semantics for Natural Language Interfaces. In *CHI '05 Extended Abstracts on Human Factors in Computing Systems* (Portland, OR, USA) *(CHI EA '05)*. Association for Computing Machinery, New York, NY, USA, 1597–1600. https://doi.org/10.1145/1056808.1056975

[43] Michael Xieyang Liu, Aniket Kittur, and Brad A. Myers. 2021. To Reuse or Not To Reuse? A Framework and System for Evaluating Summarized Knowledge. *Proc. ACM Hum.-Comput. Interact.* 5, CSCW1, Article 166 (apr 2021), 35 pages. https://doi.org/10.1145/3449240

[44] 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* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 598, 31 pages. https://doi.org/10.1145/3544548.3580817

[45] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-Train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Comput. Surv.* 55, 9, Article 195 (jan 2023), 35 pages. https://doi.org/10.1145/3560815

[46] Vivian Liu and Lydia B Chilton. 2022. Design Guidelines for Prompt Engineering Text-to-Image Generative Models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 384, 23 pages. https://doi.org/10.1145/3491102.3501825

[47] Yang Liu, Tim Althoff, and Jeffrey Heer. 2020. Paths Explored, Paths Omitted, Paths Obscured: Decision Points & Selective Reporting in End-to-End Data Analysis. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3313831.3376533

[48] Yang Liu, Alex Kale, Tim Althoff, and Jeffrey Heer. 2021. Boba: Authoring and Visualizing Multiverse Analyses. *IEEE Transactions on Visualization and Computer Graphics* 27, 2 (2021), 1753–1763. https://doi.org/10.1109/TVCG.2020.3028985

[49] 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* (San Jose, California, USA) *(CHI '16)*. Association for Computing Machinery, New York, NY, USA, 5286–5297. https://doi.org/10.1145/2858036.2858288

[50] Yifan Ma, Batu Qi, Wenhua Xu, Mingjie Wang, Bowen Du, and Hongfei Fan. 2022. Integrating Real-Time and Non-Real-Time Collaborative Programming: Workflow, Techniques, and Prototypes. *Proc. ACM Hum.-Comput. Interact.* 7, GROUP, Article 13 (dec 2022), 19 pages. https://doi.org/10.1145/3567563

[51] Walid Maalej and Martin P. Robillard. 2013. Patterns of Knowledge in API Reference Documentation. *IEEE Transactions on Software Engineering* 39, 9 (2013), 1264–1282. https://doi.org/10.1109/TSE.2013.12

[52] Maximilian Mackeprang, Claudia Müller-Birn, and Maximilian Timo Stauss. 2019. Discovering the Sweet Spot of Human-Computer Configurations: A Case Study in Information Extraction. 3, CSCW, Article 195 (nov 2019), 30 pages. https://doi.org/10.1145/3359297

[53] Microsoft. 2021. Visual studio code for the web. https://code.visualstudio.com/docs/editor/vscode-web

[54] Rada Mihalcea, Hugo Liu, and Henry Lieberman. 2006. NLP (natural language processing) for NLP (natural language programming). In *Computational Linguistics and Intelligent Text Processing: 7th International Conference, CICLing 2006, Mexico City, Mexico, February 19-25, 2006. Proceedings 7*. Springer, 319–330.

[55] L. A. Miller. 1981. Natural language programming: Styles, strategies, and contrasts. *IBM Systems Journal* 20, 2 (1981), 184–215. https://doi.org/10.1147/sj.202.0184

[56] Michael Muller, Lydia B Chilton, Anna Kantosalo, Q. Vera Liao, Mary Lou Maher, Charles Patrick Martin, and Greg Walsh. 2023. GenAICHI 2023: Generative AI and HCI at CHI 2023. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI EA '23)*. Association for Computing Machinery, New York, NY, USA, Article 350, 7 pages. https://doi.org/10.1145/3544549.3573794

[57] Michael Muller, Ingrid Lange, Dakuo Wang, David Piorkowski, Jason Tsay, Q. Vera Liao, Casey Dugan, and Thomas Erickson. 2019. How Data Science Workers Work with Data: Discovery, Capture, Curation, Design, Creation. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3290605.3300356

[58] Steve Oney, Christopher Brooks, and Paul Resnick. 2018. Creating Guided Code Explanations with Chat.Codes. *Proc. ACM Hum.-Comput. Interact.* 2, CSCW, Article 131 (nov 2018), 20 pages. https://doi.org/10.1145/3274400

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

[60] Yoann Padioleau, Lin Tan, and Yuanyuan Zhou. 2009. Listening to programmers — Taxonomies and characteristics of comments in operating system code. In *2009 IEEE 31st International Conference on Software Engineering*. 331–341. https://doi.org/10.1109/ICSE.2009.5070533

[61] Rock Yuren Pang, Ruotong Wang, Joely Nelson, and Leilani Battle. 2022. How Do Data Science Workers Communicate Intermediate Results?. In *2022 IEEE Visualization in Data Science (VDS)*. IEEE, 46–54.

[62] Rock Yuren Pang, Ruotong Wang, Joely Nelson, and Leilani Battle. 2022. How Do Data Science Workers Communicate Intermediate Results?. In *2022 IEEE Visualization in Data Science (VDS)*. 46–54. https://doi.org/10.1109/VDS57266.2022.00010

[63] Soya Park, Amy X. Zhang, and David R. Karger. 2018. Post-Literate Programming: Linking Discussion and Code in Software Development Teams. In *Adjunct Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology* (Berlin, Germany) *(UIST '18 Adjunct)*. Association for Computing

Machinery, New York, NY, USA, 51–53. https://doi.org/10.1145/3266037.3266098
[64] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *2023 IEEE Symposium on Security and Privacy (SP)*. 2339–2356. https://doi.org/10.1109/SP46215.2023.10179324
[65] I.R. Posner and R.M. Baecker. 1992. How people write together (groupware). In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, Vol. iv. 127–138 vol.4. https://doi.org/10.1109/HICSS.1992.183420
[66] Pyodide. 2023. Pyodide is a Python distribution for the browser and Node.js based on WebAssembly. https://github.com/pyodide/pyodide
[67] Luigi Quaranta, Fabio Calefato, and Filippo Lanubile. 2022. Eliciting Best Practices for Collaboration with Computational Notebooks. *Proc. ACM Hum.-Comput. Interact.* 6, CSCW1, Article 87 (apr 2022), 41 pages. https://doi.org/10.1145/3512934
[68] Laria Reynolds and Kyle McDonell. 2021. Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI EA '21)*. Association for Computing Machinery, New York, NY, USA, Article 314, 7 pages. https://doi.org/10.1145/3411763.3451760
[69] 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. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 133 (oct 2022), 26 pages. https://doi.org/10.1145/3563296
[70] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. 2012. How do professional developers comprehend software?. In *2012 34th International Conference on Software Engineering (ICSE)*. 255–265. https://doi.org/10.1109/ICSE.2012.6227188
[71] 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.
[72] 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? https://doi.org/10.48550/arXiv.2208.06213 arXiv:2208.06213 [cs.HC]
[73] Abhraneel Sarma, Alex Kale, Michael Jongho Moon, Nathan Taback, Fanny Chevalier, Jessica Hullman, and Matthew Kay. 2023. Multiverse: Multiplexing Alternative Data Analyses in R Notebooks. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 148, 15 pages. https://doi.org/10.1145/3544548.3580726
[74] Lin Shi, Hao Zhong, Tao Xie, and Mingshu Li. 2011. An empirical study on evolution of API documentation. In *Fundamental Approaches to Software Engineering: 14th International Conference, FASE 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings 14*. Springer, 416–431.
[75] Hendrik Strobelt, Albert Webson, Victor Sanh, Benjamin Hoover, Johanna Beyer, Hanspeter Pfister, and Alexander M. Rush. 2023. Interactive and Visual Prompt Engineering for Ad-hoc Task Adaptation with Large Language Models. *IEEE Transactions on Visualization and Computer Graphics* 29, 1 (2023), 1146–1156. https://doi.org/10.1109/TVCG.2022.3209479
[76] Tiptap. 2023. Tiptap is a suite of open source content editing and real-time collaboration tools for developers building apps like Notion or Google Docs. https://tiptap.dev/
[77] Immanuel Trummer. 2022. CodexDB: Generating Code for Processing SQL Queries using GPT-3 Codex. arXiv:2204.08941 [cs.DB]
[78] 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).
[79] April Yi Wang. 2022. Improving Real-Time Collaborative Data Science Through Context-Aware Mechanisms. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–3. https://doi.org/10.1109/VL/HCC53370.2022.9833140
[80] April Yi Wang, Anant Mittal, Christopher Brooks, and Steve Oney. 2019. How Data Scientists Use Computational Notebooks for Real-Time Collaboration. *Proc. ACM Hum.-Comput. Interact.* 3, CSCW, Article 39 (nov 2019), 30 pages. https://doi.org/10.1145/3359141
[81] April Yi Wang, Dakuo Wang, Jaimie Drozdal, Michael Muller, Soya Park, Justin D. Weisz, Xuye Liu, Lingfei Wu, and Casey Dugan. 2022. Documentation Matters: Human-Centered AI System to Assist Data Science Code Documentation in Computational Notebooks. *ACM Trans. Comput.-Hum. Interact.* 29, 2, Article 17 (jan 2022), 33 pages. https://doi.org/10.1145/3489465
[82] April Yi Wang, Zihan Wu, Christopher Brooks, and Steve Oney. 2020. Callisto: Capturing the "Why" by Connecting Conversations with Computational Narratives. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3313831.3376740
[83] Dakuo Wang, Elizabeth Churchill, Pattie Maes, Xiangmin Fan, Ben Shneiderman, Yuanchun Shi, and Qianying Wang. 2020. From Human-Human Collaboration to Human-AI Collaboration: Designing AI Systems That Can Work Together with People. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI EA '20)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3334480.3381069
[84] Sitong Wang, Samia Menon, Tao Long, Keren Henderson, Dingzeyu Li, Kevin Crowston, Mark Hansen, Jeffrey V Nickerson, and Lydia B Chilton. 2023. ReelFramer: Co-creating News Reels on Social Media with Generative AI. *arXiv preprint arXiv:2304.09653* (2023).
[85] John Wenskovitch, Jian Zhao, Scott Carter, Matthew Cooper, and Chris North. 2019. Albireo: An Interactive Tool for Visually Summarizing Computational Notebook Structure. In *2019 IEEE Visualization in Data Science (VDS)*. 1–10. https://doi.org/10.1109/VDS48975.2019.8973385
[86] 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* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 385, 22 pages. https://doi.org/10.1145/3491102.3517582