NeuroSync: Intent-Aware Code-Based Problem Solving via Direct LLM Understanding Modification

Wenshuo Zhang
The Hong Kong University of Science and Technology
Hong Kong SAR, China
wzhangeb@connect.ust.hk

Leixian Shen
The Hong Kong University of Science and Technology
Hong Kong SAR, China
lshenaj@connect.ust.hk

Shuchang Xu
The Hong Kong University of Science and Technology
Hong Kong SAR, China
sxuby@connect.ust.hk

Jindu Wang
The Hong Kong University of Science and Technology
Hong Kong SAR, China
jwangki@connect.ust.hk

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

Huamin Qu
The Hong Kong University of Science and Technology
Hong Kong SAR, China
huamin@cse.ust.hk

Lin-Ping Yuan*
The Hong Kong University of Science and Technology
Hong Kong SAR, China
yuanlp@cse.ust.hk

Abstract

Conversational LLMs have been widely adopted by domain users with limited programming experience to solve domain problems. However, these users often face misalignment between their intent and generated code, resulting in frustration and rounds of clarification. This work first investigates the cause of this misalignment, which dues to *bidirectional ambiguity*: both user intents and coding tasks are inherently nonlinear, yet must be expressed and interpreted through linear prompts and code sequences. To address this, we propose *direct intent–task matching*, a new human–LLM interaction paradigm that externalizes and enables direct manipulation of the *LLM understanding*, i.e., the coding tasks and their relationships inferred by the LLM prior to code generation. As a proof-of-concept, this paradigm is then implemented in *NeuroSync*, which employs a knowledge distillation pipeline to extract LLM understanding, user intents, and their mappings, and enhances the alignment by allowing users to intuitively inspect and edit them via visualizations. We evaluate the algorithmic components of NeuroSync via technical experiments, and assess its overall usability and effectiveness via a user study (N=12). The results show that it enhances intent–task alignment, lowers cognitive effort, and improves coding efficiency.

*Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
UIST '25, Busan, Republic of Korea
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-2037-6/25/09
https://doi.org/10.1145/3746059.3747668

CCS Concepts

• **Human-centered computing** → *Graphical user interfaces*; HCI theory, concepts and models; • **Computing methodologies** → **Discourse, dialogue and pragmatics**.

Keywords

Human-LLM Alignment, Coding, Bidirectional Ambiguity, Graph Representation, Distillation

ACM Reference Format:
Wenshuo Zhang, Leixian Shen, Shuchang Xu, Jindu Wang, Jian Zhao, Huamin Qu, and Lin-Ping Yuan. 2025. NeuroSync: Intent-Aware Code-Based Problem Solving via Direct LLM Understanding Modification. In *The 38th Annual ACM Symposium on User Interface Software and Technology (UIST '25), September 28–October 01, 2025, Busan, Republic of Korea.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3746059.3747668

1 Introduction

Programming is an essential and practical tool for domain users to solve problems within their areas of expertise. For instance, a marine biologist might need to analyze large amounts of ocean data to study climate change or marine ecosystems. These domain users often lack programming skills and struggle to implement these solutions themselves. Conversational Large Language Models (LLMs), such as ChatGPT, have become popular among these users [65] because they allow users to express their problem-solving intents through natural language prompts and receive automatically generated code. This lowers the barriers to leveraging programming to solve problems. Despite this benefit, users frequently encounter *misalignment* between their intents and the code generated by LLMs [52]. This misalignment typically leads to repetitive cycles of clarification and debugging, causing frustration and task failure.

Current efforts addressing misalignment generally fall into two categories. The first body of work focuses on improving *user-to-LLM communication*, aiming to help users formulate clear and structured prompts, such as logically organized coding tasks or pseudocode [64, 71]. The second targets *LLM-to-user communication*, which seeks to enhance users' understanding of generated code via interactive explanations and visualizations [65, 66]. While effective, they are primarily designed to support professional programmers, who have the expertise to decompose problems into coding tasks and interpret the generated code. In contrast, domain users lack the expertise to identify or articulate misalignment through direct interaction with code. They instead rely heavily on conversational interactions with LLMs, and often result in notable friction to complete their tasks with current approaches.

To better understand why misalignment exists during the conversational process and how it can be effectively addressed, we conducted a two-phase formative study with six domain users to investigate this problem. In the first phase, we analyzed human-LLM conversation histories arising from their daily work. We uncovered *bidirectional ambiguity* as a key reason for misalignment: both user intents and coding tasks are inherently nonlinear and dynamic, yet must be communicated through linear prompts and code representations. Building on this insight, the second phase centered on finding suitable visual representations for alleviating such misalignment. We explored graph visualizations for non-linear coding tasks and employed a tech probe to evaluate their pros and cons.

Upon deeper understanding of the *bidirectional ambiguity*, we propose a novel human-LLM interaction paradigm called *direct intent–task matching* to address this issue (Fig. 1). In traditional paradigms, LLMs generate code directly from user prompts without revealing their "internal understanding". In contrast, our approach introduces a transparent process that externalizes the *LLM understanding*, which refers to the coding tasks and their relationships. These tasks and relationships are inferred by the LLM from a user's prompt and serve as the basis for the code it generates. This paradigm allows users to directly interact with the *LLM understanding*, diagnosing and correcting any inaccuracies or misalignment, before code is generated.

We operationalized this paradigm in a proof-of-concept system named NeuroSync. As shown in Fig. 2, when a user inputs a prompt, NeuroSync first extracts the LLM understanding, user intents, and their mappings. It then visualizes the LLM understanding as a graph and user intents as a tree, allowing users to manipulate and refine the graph according to their intents visualized in the tree. Once the user confirms, NeuroSync generates and displays code guided by the updated LLM understanding, ensuring it accurately aligns with the user intents. Furthermore, NeuroSync incorporates two algorithmic components to address two critical challenges during the process. First, users often lose track of how the LLM understanding graph evolves while the intent changes, especially when the graph grows more complex. We thus designed an intent-aware graph simplication algorithm that can identify the nodes related to the intent changes, allowing for emphasizing them in the visualization. Second, extracting LLM understandings and user intents can be computationally heavy. We then leveraged a novel distillation pipeline to fine-tune small language models to enable faster extraction.

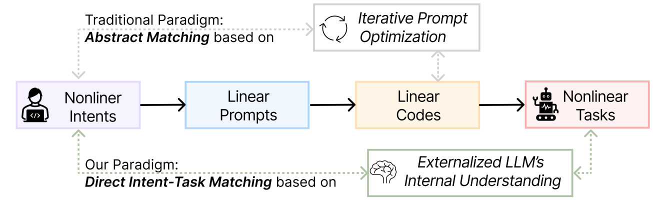

Figure 1: Comparison between the proposed *direct intent–task matching* paradigm and the traditional paradigm for user-LLM interactions in programming.

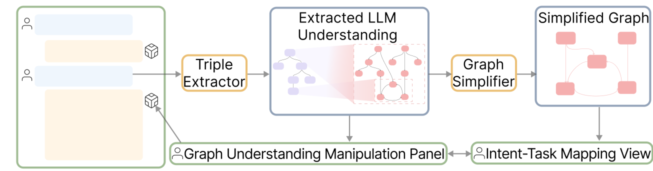

Figure 2: Overview of NeuroSync, a proof-of-concept implementation of the *direct intent–task matching* paradigm. NeuroSync takes user prompts as input, extracts the *LLM understanding,* enables users to refine this understanding through graph-based visualizations, and feeds the refined understanding back to the LLM to generate code that more accurately aligns with user intents.

We carried out technical experiments to assess the graph implications algorithm and the distillation pipeline, revealing the effectiveness of our designed algorithms. We further evaluated NeuroSync through a controlled user study with 12 domain users, compared to a customized baseline. The results demonstrate that NeuroSync substantially improves intent–task alignment, reduces cognitive load, and enhances coding efficiency, confirming the effectiveness of *direct intent–task matching* as a novel conversational coding paradigm for non-professional programmers.

In summary, our contributions are threefold:

- We introduce *direct intent–task matching*, a paradigm that externalizes *LLM understanding*, enabling domain users with limited programming expertise to refine and align coding tasks with their intents before code generation.
- We develop NeuroSync, a proof-of-concept system that implements this paradigm with interactive graph-based visualizations and two algorithmic components: a simplification algorithm to manage graph complexity and a distillation pipeline for faster extraction of LLM understanding.
- We validate NeuroSync through technical experiments and a user study, demonstrating its ability to improve intent–task alignment, reduce cognitive load, and enhance coding efficiency.

2 Related Work

2.1 Human–LLM Alignment in Coding

Human–LLM alignment seeks to ensure that LLM outputs reflect users' intentions, particularly in code-based problem-solving scenarios [59, 60]. Two key issues are commonly identified: how to help

Table 1: Demographics of Participants in the Formative Study. 'Exp.' denotes their user experience within their respective domains. 'Task' indicates the specific tasks performed according to their provided dialog history. 'Rounds' represents the number of conversation rounds conducted.

ID	Domain	Exp.	Task	Interaction Mode/LLM Name	Rounds
P1	Design (Ph.D.)	1.5 Years	Web article crawlers with python	Coding Mode/Qwen	11
P2	Electric Grid (UG)	0.5 Years	Signal analysis with Matlab	Chatting Mode/Doubao	13
P3	Clinical Medicine (Ph.D.)	6 Years	Medical image processing and classification with Python	Chatting Mode/Kimi	21
P4	Theory Math (Ph.D.)	3 Years	Table drawing and adjusting with Latex	Coding Mode/Qwen	8
P5	Policy Making (M.S.)	1 Year	Interview data analysis and visualization with python	Coding Mode/GPT-4o	13
P6	Economics (M.S.)	5 Years	Financial data analysis and visualization with python	Data Analysis Mode/GPT-4o	29

users formulate effective instructions, and how to assess whether the generated outputs match their intents [43].

Prompt engineering is the primary approach for improving alignment in conversational coding systems. Sarkar et al. [41] highlight users' iterative refinement of prompts, describing this iterative process as *abstract matching*, which reflects Norman's gulf of execution [17]. In addition to this, users also encounter a related cognitive barrier known as the "gulf of envisioning" [52], where articulating clear tasks and anticipating model outputs is challenging. To bridge this cognitive gap, prior works have proposed various strategies such as in-context learning [7, 26], structured decomposition approaches (*e.g.,* AI Chains [64], CoLadder [71]), and the *feedforward* mechanism [31, 46, 58]. Among them, CoLadder [71] uses unidirectional, monologue-driven interaction and content in "block" or "chain" UIs [64] are generated statically rather than automatically adapting to users' changing intent during usage, while NeuroSync is used in a bidirectional dialogue scenario and generates editable, simplified graphs on the fly based on dynamic user intents in each interaction round.

Feedforward specifically anticipates outcomes before actions occur by prompting the model to generate interactive previews, such as predicted visual outputs, code suggestions, or descriptive summaries, based on user input. These previews help minimize unnecessary clarification rounds, bridge abstraction gaps between users and AI [28, 50, 56], and support users' metacognitive reasoning [55]. Compared to existing work [28] that directly presents real LLM-generated code directly as feedforward information [28], NeuroSync externalizes the LLM's internal understanding—the tasks and their relationships that are implicitly encoded in the generated code—before code is generated. This reduces the cost of iteration and ensures consistency between preview and final output via knowledge distillation [11].

Alignment and intent specification challenges are also well-studied in program synthesis, where the focus has shifted from complete formal specifications to interpreting ambiguous, high-level user intents. To address this ambiguity, interactive methods refine intent through user feedback, such as selecting distinguishing inputs to disambiguate candidate programs [22, 75], or through multi-turn conversational dialogues [33]. For complex tasks, Gulwani's work in programming-by-example introduced divide-and-conquer strategies [12], recursively partitioning examples, synthesizing distinct programs for subsets, and composing results with conditionals. More recently, reinforcement learning has been used to align programs with functional correctness by rewarding models for passing unit tests [73]. A key distinction lies in the abstraction level: traditional pre-LLM methods emphasize code-level interactions (*e.g.,* input-output examples [13, 37], execution traces [6, 49], or program sketches [51]), whereas NeuroSync operates at the task level, externalizing high-level LLM tasks before code generation.

Current research also highlights the cognitive load associated with comprehending and debugging LLM-generated code [1, 32, 71]. Approaches such as real-time explanations [9, 66] and visualization-based presentations [65] have been proposed to ease understanding. Unlike these post-generation methods, NeuroSync intervenes before code generation, explicitly visualizing tasks and their relations, thus reducing cognitive complexity at an earlier stage.

2.2 Graph-Based Interfaces for LLM Interaction

Conversational LLM interfaces predominantly adopt linear interaction flows, where user prompts and model responses are serialized in turn [65]. While simple, such interfaces struggle with complex tasks due to high cognitive load [65], versioning issues [21], and limited user control [30, 70]. To overcome these limitations, recent systems explore alternative interaction paradigms for complex tasks like creative design [53], exploratory analyses [54], and data analysis programming [65].

Graphs naturally represent relational structures via nodes and edges [39], with diverse visualizations like node-link diagrams [18, 48] and hierarchical trees [74]. Recent studies have leveraged graph-based UIs to enhance LLM-driven management and logical reasoning [67]. For instance, Kim et al. [21] abstract writing processes into graphs to facilitate versioning; WaitGPT [65] visualizes analysis tasks via data-flow graphs; Promptchainer [63] and GoT [2] use graph representations to support multi-step task decomposition. Moreover, tools like Low-code LLM [4] and CoLadder [71] employ graph interfaces to directly organize intents and code snippets, improving alignment and clarity.

Recognizing these advantages, NeuroSync adopts graph-based visualizations to represent coding tasks and their relations. While existing studies focus mostly on static graph visualizations and cognitive load management [72], NeuroSync introduces dynamic graph simplification that automatically adapts graphs in real-time according to evolving user intents, reducing users' cognitive burden during multi-turn interactions.

2.3 LLM Reasoning and Task Structuring

Recent advances in applying LLMs to reasoning-intensive tasks (*e.g.,* programming [23], mathematics [16]) have identified limitations in reasoning capabilities of basic LLM models [10]. To

enhance reasoning, researchers proposed techniques like prompt tuning [29, 40] and structured reasoning prompts such as Chain-of-Thought (CoT) [62], Least-to-Most (L2M) [77], and Tree-of-Thought (ToT) [69]. These methods decompose reasoning into intermediate steps, enhancing model performance on complex problems.

CoT decomposes complex problems into linear sub-steps, which is suitable for problems with sequential logic but has limited ability to support nonlinear reasoning [3]. In contrast, methods such as L2M [77] incrementally introduce complexity, avoiding premature limitations in thought exploration. Variants like Path-of-Thought (PoT) [5], Concept Composition (CoC) [24], and Aggregation-of-Thought (AoT) [42] further optimize efficiency and performance. Meanwhile, ToT explores choices via tree-structured decision points, supporting complex multi-step tasks, with Graph-of-Thoughts (GoT) [2] providing an aggregated state exploration approach.

These approaches aim to improve internal LLM reasoning. In contrast, NeuroSync externalizes and exposes the LLM's inferred task structure—what we term the model's *internal understanding*—before execution. While different in purpose, both lines of work reflect the importance of intermediate structure in aligning model behavior with user intent. In NeuroSync, this structure serves not only as guidance for code generation but also as a manipulable medium for user–LLM alignment.

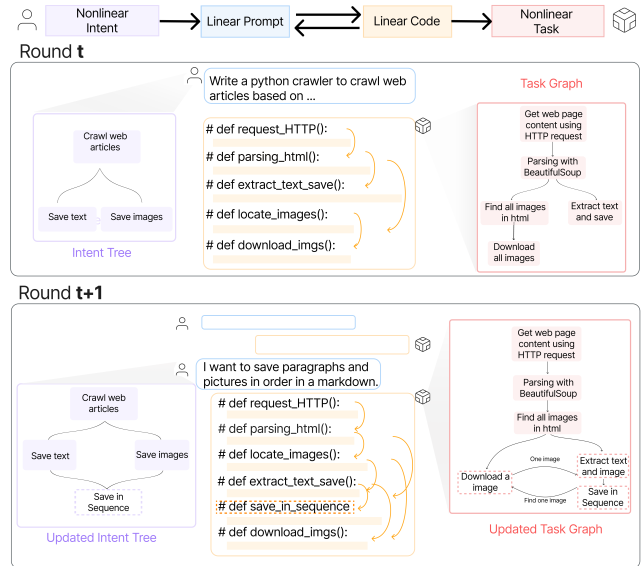

Figure 3: Illustration of bidirectional ambiguity.

3 Formative Study

To examine the causes of human–LLM misalignment in conversational coding and inform system design, we conducted a formative study with domain users who have little coding experiences. The study included (1) interaction history analysis and interviews to uncover misalignment patterns, and (2) semi-structured interviews, informed by a literature review, to explore effective representations of graphs for conveying code tasks and user intent.

3.1 Study 1: Understanding Human-LLM Misalignment

3.1.1 Participants and Data. We invited six participants (P1- P6) to conduct retrospective analysis. They were from diverse domain backgrounds with varying levels of programming experience and education background (Tab. 1). Each provided full records of prior real-world interactions with LLMs while performing coding tasks for problem-solving. On average, each task had 15.83 interaction rounds (SD = 7.76) per session, spanning four LLM platforms and multiple programming languages (*e.g.,* Python, MATLAB, LaTeX) under different LLM modes.

To analyze the reasons behind misalignment during conversational code generation, we conducted an analysis on users' interaction history and observed that the phenomenon of LLMs failing to accurately generate code aligned with user intent is widespread (6/6) and leads to a number of useless interactions.

3.1.2 Analysis Protocol. We analyzed participants' interaction histories with LLMs using an open-coding approach [20]. Two authors independently coded the data and iteratively refined the codes until reaching agreement. The codes were then grouped into themes, which were carefully reviewed and discussed to identify the key findings of the study.

We first annotated each interaction round with: (1) *User intent*, inferred through participant clarification and interaction review; (2) *Prompt quality*, evaluated by independent raters to assess how well the prompt conveyed the intended task; (3) *LLM-executed tasks*, extracted from generated code and mapped to user intent.

Building on these annotations, we further identified potential misalignment points—rounds where user intent remained stable but the generated code deviated from the intended task. This process allowed us to trace breakdowns across turns and identify underlying causes of intent–task misalignment.

3.1.3 Findings. **Bidirectional ambiguity is a major cause of human-LLM misalignment in conversational coding tasks.** During the conversation with LLM for coding tasks, ambiguity is bidirectional: *User-to-LLM*: Users find it challenging to clearly express their needs and the information required by the LLM in their prompts. For example, converting tree-like intent in Fig. 3 into prompt will lose direct structure and cause ambiguity. *LLM-to-User*: Users struggle to understand the specific tasks and execution logic embedded in the code, making it difficult to provide precise modification requests. For example, in Fig. 3, users need to reconstruct codes and code relationships by themselves, which is difficult and low ability of understanding code will lead to ambiguity. This bidirectional ambiguity compounds over turns, causing LLMs to produce code misaligned with user intent. As LLM capabilities grow and inference slows, the cost of these ineffective interactions increases.

User-to-LLM Ambiguity. Users often struggle to express their intent clearly due to three key issues: (1) *Nonlinear intent loss*: User goals are typically hierarchical and evolve over time. However, when mapped into linear prompts, this structure is flattened, leading to semantic ambiguity and loss of global intent. (2) *Contextual omissions*: Prompts often lack critical information due to delayed

Table 2: Comparison of different task representations.

Representation Method	Intuitive	Easy to Modify	Layman-friendly
Task Flow Diagram (Graph) [57]	✓	✓	✓
Pseudocode [34]	×	×	×
UML Activity Diagram [8]	✓	✓	×
Decision Table [38]	×	×	×
Data Flow Diagram [65]	✓	✓	×
Natural Language Description	×	×	✓

articulation and limited user memory. LLMs, with constrained context windows, may miss important prior requirements. (3) *Vague modification guidance*: Domain users, unfamiliar with code internals, frequently describe desired outcomes without specifying what to change. This hampers LLMs to revise code accurately.

LLM-to-User Ambiguity. LLM-generated code embeds multiple interrelated tasks, often structured nonlinearly. Users with limited programming experience face difficulty in unpacking this structure, identifying task boundaries, and understanding the model's reasoning. As a result, they may overlook unintended logic, misinterpret execution flow, or fail to spot partial completions, making it hard to issue precise follow-up instructions. This impairs both comprehension and correction, especially in multi-round interactions where misunderstandings accumulate.

3.1.4 Implications. To reduce LLM-to-user ambiguity, systems must present model-inferred tasks in a more interpretable form, allowing users, especially those with limited coding expertise, to understand and guide code generation without reading raw code.

3.2 Study 2: Exploring Graph-Based Representations for Code Tasks

As discovered in Study 1, prompts and code presented in linear forms do not effectively convey non-linear intents or help users understand non-linear code tasks involved in multiple rounds of interactions. Therefore, in Study 2, we aim to explore whether there are better representations that improve the communication of non-linear coding tasks between users and LLMs.

3.2.1 Representation Survey. We first conducted an expert-driven ideation to enumerate common formats, and then verified each through a focused review of recent representative papers. This approach allowed us to capture a broad spectrum of practical representations without relying on unfocused keyword searches. Results are shown in Tab. 2. Among these, graph-based structures (*e.g.,* task flow graphs) emerged as intuitive, editable, and directly tied to task logic. While promising, we also observed that as programming intent evolves, task graphs can become increasingly complex, highlighting the need for intent-aware abstraction and scalable visual structures.

3.2.2 Graph-Based Two-stage Extractor. To evaluate the feasibility of graph-based representations in real-world interactions, we developed a prototype tool that extracts task graphs from user–LLM interactions. The probe processes both initial prompts and follow-up inputs to incrementally build and update task-level representations (Fig. 4). It supports:

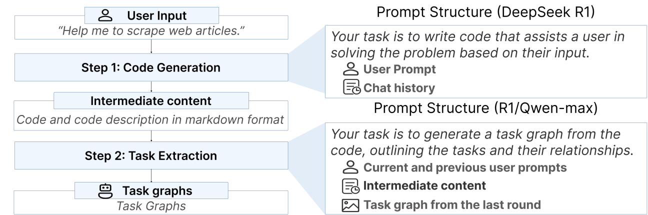

Figure 4: Two-stage extractor. Graph-based task representation will be extracted after the code is generated based on the user prompt.

- *Task Graph Initialization:* At the first interaction, the probe analyzes the user's prompt and the LLM's response (codes) to identify and structure basic tasks, subtask groups, and task dependencies behind the codes. The result is a hierarchical graph, where nodes represent tasks and edges denote logical or sequential relationships.
- *Task Graph Refinement:* In subsequent turns, the graph is incrementally updated based on new prompts, responses, and historical context. Refinement includes task additions, deletions, and updates, especially around variable usage and logic changes, to ensure consistency and traceability across rounds.

Task graphs are rendered as part of the conversation interface and stored for further review. However, during early trials, we noted that API latency became a bottleneck in multi-turn settings, pointing to the need for lighter-weight implementations.

3.2.3 Study Protocol. To assess the effectiveness of task graphs, we conducted semi-structured interviews with the same six interdisciplinary users in Study 1 (P1-P6) and used an open-coding approach [20] for data analysis. We first extracted samples from three key stages of user–LLM interaction: initialization, intent progression, and completion. Each sample included the prompt, LLM-generated code, and the corresponding task graph, presented sequentially to participants. The procedure includes three stages. In Stage 1, participants reviewed only the generated code and described their understanding and obstacles. In Stage 2, they examined both the code and task graph, comparing the ease of task comprehension with and without the graph, and offered suggestions for improvement. In Stage 3, we showed consecutive code–graph pairs from two interaction turns and asked participants to identify task changes, misunderstandings, and strategies for tracking differences. Interviews focused on two core questions: (1) Do graphs improve task understanding compared to directly reading code? (2) What features are needed to support effective graph interpretation and updates? Throughout the process, we observed participants' behaviors in navigating graph changes and concluded with a reflection session to gather feedback on cognitive load, update clarity, and expectations for graph design.

3.2.4 Findings. All participants reported difficulty in understanding raw code due to limited programming knowledge. Key barriers included misalignment between linear code flow and branching task logic (P1, P4, P5), and high cognitive load from memorizing variable usage and logic transitions (P3, P6). In contrast, task graphs were

consistently viewed as more helpful. Participants highlighted two key benefits: (1) Improved task comprehension through clear visualization of task dependencies and subgoals (P1, P3, P4, P5, P6); (2) Enhanced efficiency in locating key logic points and understanding overall code purpose (P1, P3, P5).

However, as interaction rounds increased, graph complexity grew and negatively impacted interpret ability. While some participants were able to reconstruct evolving task intents (P1, P2, P5, P6), others reported confusion or errors (P3, P4). To address this, users suggested: (1) Providing abstracted overviews with zoom-in capabilities (P1-P6); (2) Supporting node-level exploration for localized inspection (P1, P4); (3) Grouping and annotating task clusters to clarify hierarchy and dependencies (P2, P3, P5, P6).

3.2.5 Implications. Use graph-based representations to externalize LLM coding tasks and tree structures to model user intents, enabling fine-grained alignment. To ensure responsiveness, employ lightweight feedforward representations and dynamically abstract graph complexity based on evolving user intent.

3.3 Design Considerations

Based on the findings from the two phases, we derive several design considerations:

DC1: Support Bidirectional Disambiguation through Intent and Task Externalization. To mitigate user–LLM misalignment, systems should support bidirectional disambiguation. On the user-to-LLM side, this requires making user intent explicit and editable, preserving task structure and global context beyond linear prompt input. On the LLM-to-user side, the model's inferred coding tasks should be externalized in interpretable forms, enabling non-programmers to understand, verify, and adjust the system's interpretation without reading raw code.

DC2: Enhance Fluid Modification on Tasks to Align Intent in Multi-round Interactions. Matching intent and tasks in conversational systems often requires multiple rounds, but high latency can disrupt users' focus, making it difficult to think fluidly and modify tasks effectively. To enable effective intent-task alignment, systems should support low-latency interactions. However, extracting LLM understanding and user intent can be computationally expensive, especially when task structures grow in complexity. Therefore, the system should employ lightweight yet accurate feedforward mechanisms that allow for rapid generation and update of intermediate representations.

DC3: Leverage Structured Graph Representations with Intent-Aware Abstraction. Graph-based representations are effective for externalizing the LLM's task structure, while user intent can be modeled as a tree, a specialized directed acyclic graph reflecting hierarchical goal decomposition. To manage complexity and cognitive load, systems should dynamically abstract or simplify task graphs based on evolving user intent, enabling scalable yet focused interaction across varying levels of detail.

4 Direct Intent-Task Matching

Informed by **DC1**, we propose a novel human-LLM interaction paradigm called *direct intent–task matching* (Fig. 1). The paradigm externalizes and enables direct manipulation of the *LLM understanding* to support bidirectional disambiguation.

Specifically, inspired by the concept of *user understanding*, where humans develop their interpretation of LLM outputs, we suggest that LLMs form a kind of understanding of user inputs. We call this **LLM understanding**, which refers to the tasks and their relationships implicitly encoded in the code that an LLM is expected to generate based on user prompts. By exposing this *LLM Understanding* prior to code generation, users can interpret the intended tasks without reading raw code, reducing LLM-to-user ambiguity. Moreover, since the structure is editable, users can directly modify task representations, aligning them with their actual intent. This feedforward representation not only improves transparency but also serves as a lightweight, structured input alongside prompts, helping to resolve user-to-LLM ambiguity.

Direct Intent–Task Matching is a process that allows users to engage directly with the *LLM understanding* before code is generated to address bidirectional ambiguity. Instead of relying on traditional prompt iteration or adjusting mismatched outputs after generation, users can iteratively refine how the LLM interprets their intent into specific coding tasks. This refinement resolves misalignments early in the process, ensuring the LLM's understanding evolves dynamically with each adjustment. By feeding this corrected understanding back into the LLM, users can achieve more efficient, interpretable, and intent-aligned code generation, streamlining the path from intent to output.

5 NeuroSync

We implement the *direct intent–task matching* paradigm into a proof-of-concept system named NeuroSync (Fig. 2).

5.1 Overview

NeuroSync operates in a multi-stage interaction loop. After the user submits a natural language prompt, NeuroSync extracts a structured representation of the LLM understanding (*i.e.,* the predicted code-level tasks and their relationships), alongside the user's intent and their mappings. These representations are then externalized in visual forms: the LLM understanding is shown as a graph based on our formative study 3, while the user intent is presented as a hierarchical tree structure based on previous research [71]. This visual design allows users to directly inspect, correct, and confirm task-level alignment, addressing **DC1** by supporting bidirectional disambiguation, as users no longer need to guess what the LLM will do, nor must they articulate intent solely through prompts.

To maintain low-latency interaction during multi-turn sessions, especially given the cost of extracting structured task representations, we implement a lightweight knowledge distillation pipeline 5.3.2. This pipeline fine-tunes a small language model (SLM) to extract LLM understanding efficiently, based on training data generated by a multi-agent simulation system 5.3.3. This design meets **DC2**, ensuring that feedforward task information can be presented quickly and accurately without introducing delay or performance bottlenecks.

To support effective task visualization and user control, NeuroSync adopts graph-based representations and integrates an *intent-aware graph simplification algorithm* 5.4 that dynamically adjusts the complexity of the understanding graph according to recent

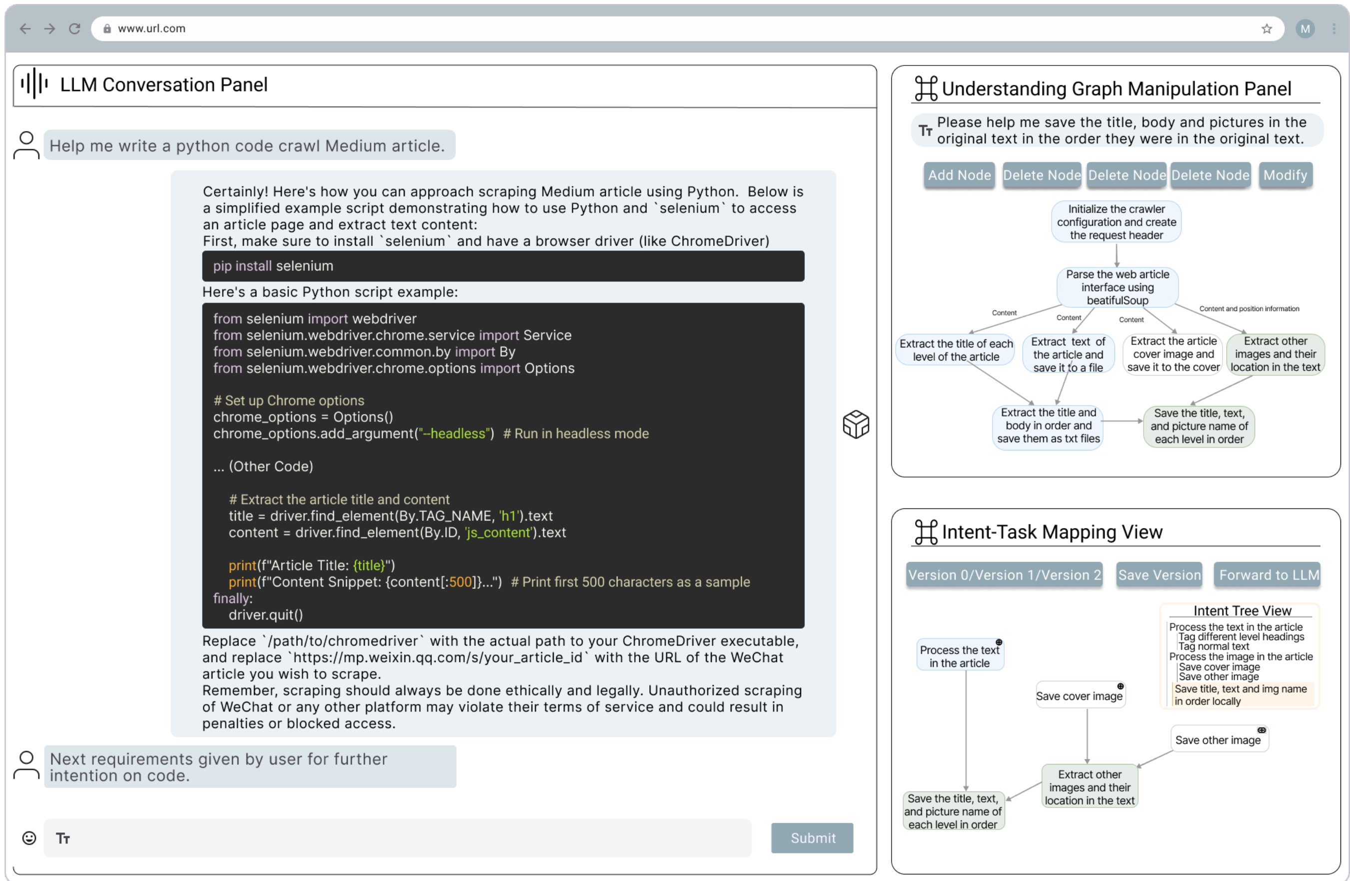

Figure 5: Interface of NeuroSync. Users interact with the LLM through Panel A (LLM Conversation Panel). Before each LLM response, the system generates an LLM understanding graph in Panel B (Understanding Graph Manipulation Panel) and a simplified version in Panel C (Intent–Task Mapping View). Users can edit the task graph in Panel B and explore task structures and intent alignment via Panel C.

intent updates. This design directly responds to **DC3**, enabling scalable interaction through focused abstraction: users can access both global task structure and local task details, and selectively expand or highlight task components as needed.

5.2 Usage Scenario

5.2.1 User Interface. As shown in Fig. 5, NeuroSync's user interface consists of (A) LLM Conversation Panel, (B) Understanding Graph Manipulation Panel, and (C) Intent-Task Mapping View.

(A) LLM Conversation Panel. It functions like a standard LLM interface, where users input prompts and receive responses. However, unlike traditional systems that rely solely on text prompts, NeuroSync also incorporates the current version of the Understanding Graph into the generation process. Users can iteratively refine the graph without submitting a new prompt, enabling multiple rounds of graph-guided responses under the same user intent.

(B) Understanding Graph Panel. Before each LLM response, the system generates a task-level Understanding Graph based on the user's prompt. Users can inspect and directly edit the graph to correct or clarify task understanding, including two levels of modification:

- Graph-Level Modification: Users input natural language instructions into a *modify block*, and NeuroSync applies large-scale structural edits via the LLM API. This is ideal for reorganizing major task flows or introducing new task groups.
- Node-Level Modification: Users can manually adjust nodes and links, *i.e.,* adding, deleting, or editing task descriptions, enabling precise control over subtasks.

(C) Intent–Task Mapping Panel. To reduce cognitive load, this panel presents a simplified view of the Understanding Graph aligned with the user's intent. It includes two components:

- Intent Tree View: A hierarchical tree representing the user's structured goals.
- Simplified Understanding Graph: A filtered version of the full graph, generated using our intent-aware graph simplification algorithm (Sec. 5.4). It highlights nodes directly relevant to the current intent and merges irrelevant ones for clarity.

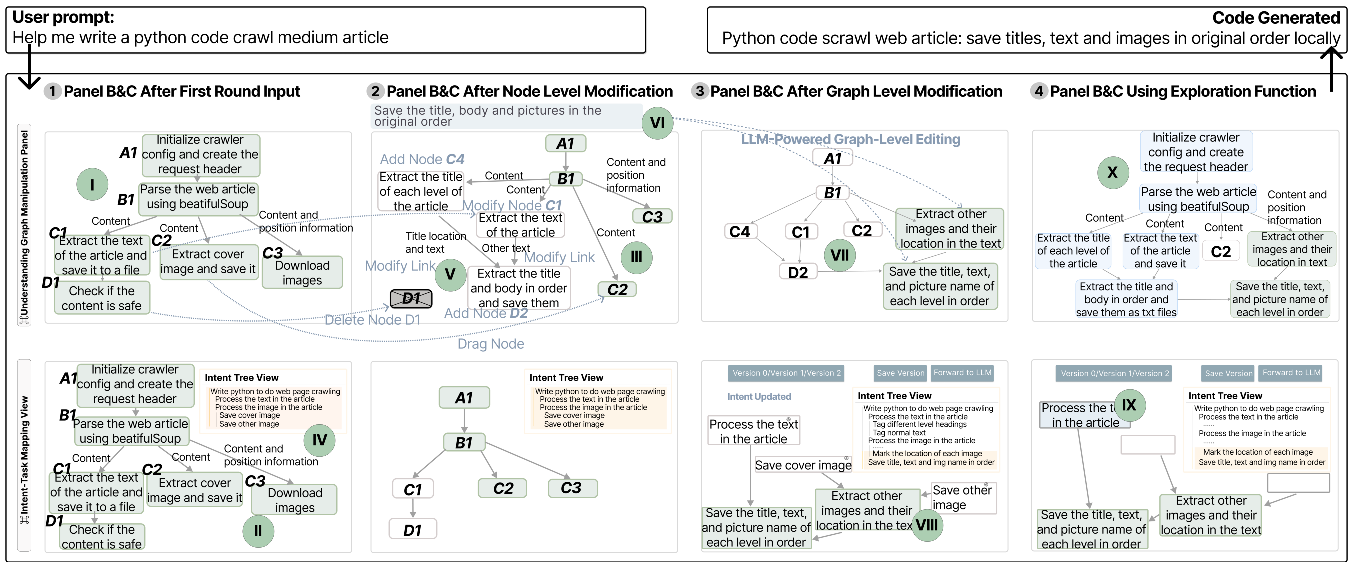

Figure 6: User interactions with NeuroSync. After entering a prompt in Panel A, users engage in a four-stage process of task exploration and modification in Panels B and C prior to code generation. (1) Upon prompt submission, an initial understanding graph is shown in Panel B, along with a simplified version in Panel C, where nodes associated with intent changes are highlighted. (2) Users can interactively explore the graph (*e.g.*, via dragging) and perform fine-grained node-level edits, such as modifying descriptions or adding nodes. (3) Alternatively, users may issue natural language commands to modify the graph. These updates are reflected in both panels, again with intent-relevant nodes highlighted. (4) Users may also click on merged nodes in Panel C to focus on corresponding subgraphs in Panel B. Once confirmed, the updated understanding is passed to the LLM for code generation. Selected nodes are zoomed in for clarity.

Panel C updates when a new prompt is submitted or when major edits occur in Panel B. Clicking a merged node highlights its corresponding region in the full Understanding Graph for focused inspection.

5.2.2 Walkthrough Example. Consider Kelly, a journalism student developing a web-based system for monitoring public opinion. To support her project, she needs a web crawler capable of extracting article content; however, she lacks the programming expertise to implement one herself. As a result, she turns to NeuroSync for assistance.

Upon launching the LLM interface with NeuroSync enabled, Kelly is presented with a standard conversation panel (see Fig. 5A), accompanied by the Understanding Graph panel (B) and the Intent–Task Mapping panel (C). Kelly enters her request in the input field of Panel A, for example: *e.g.*, "*Help me write a Python script to crawl media articles*". Unlike traditional LLM systems, NeuroSync immediately visualizes its inferred understanding in Panel B (see I in Fig. 6) and displays structured user intent in Panel C (II and IV in Fig. 6). The system decomposes the request into a sequence of subtasks—such as "*Initialize the crawler configuration and create the request header*" and "*Download images*"—and organizes them as a task flow diagram (see I in Fig. 6). Panel C shows the same graph structure, highlighting all task nodes (see II in Fig. 6) and presenting the overall intent tree (see IV in Fig. 6).

Kelly interacts with the Understanding Graph by dragging nodes to explore its structure and quickly **identifies a misrepresentation of her original intent** (see III in Fig. 6). Specifically, she notices that the graph includes a task related to content safety checking, which is outside the scope of her current objective. Instead, her goal is to distinguish between different title levels and save them alongside the main article text. To achieve this, she deletes the security-checking node and adds a new node labeled "*Extract the title at each level of the article*" in Panel B. To accommodate this addition, she re-names the existing node "*Extract the text of the article and save it to a file*" as "*Extract the title and body in order and save them as TXT files*". She then creates appropriate links to integrate the new node into the existing task flow. These operations are shown in V in Fig. 6.

After completing the initial modifications and finalizing the graph as shown in Panel B of Region 2, Kelly **encounters a new challenge**: she wants to save the titles, body text, and images from the original webpage in their original order. However, she finds this task complex and is uncertain how to proceed. She enters her revised requirement into the modification input block in Panel B (see VI in Fig. 6) and clicks *Modify*. In response, NeuroSync automatically updates the Understanding Graph by adding two new nodes (highlighted in yellow in VII in Fig. 6) and generates an updated, simplified graph along with a highlighted intent tree in Panel C (VIII in Fig. 6). To reflect the refined goal, NeuroSync merges relevant nodes in the Understanding Graph based on high-level intent

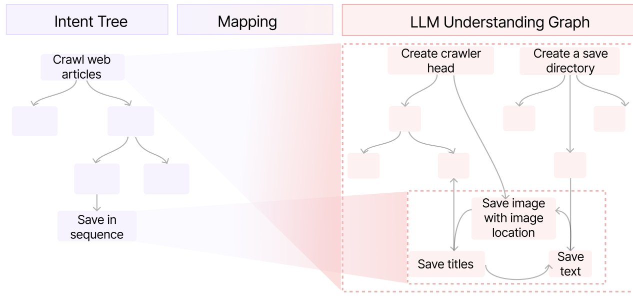

Figure 7: Illustration of a triple, which consists of an intent tree, an LLM understanding graph (*i.e.,* a task graph generated prior to code generation), and their mappings. Each node in the intent tree may correspond to one or more nodes in the LLM understanding graph. The figure highlights the mappings for two example intent nodes.

tree components such as "*Process the text in the article*" and "*Save cover image*", selectively highlighting and retaining only the nodes directly associated with the updated intent.

Finding it challenging to interpret the entire Understanding Graph, Kelly clicks the extension button to focus on a specific sub-intent: "*Process the text in the article*" (IX in Fig. 6). In response, Panel B highlights the subset of nodes related to this intent, allowing her to concentrate more effectively on the text processing components (X in Fig. 6). Once satisfied with the revised structure, Kelly clicks *Confirm Graph*, prompting the LLM to generate code aligned with her refined understanding as represented in the graph.

Compared to her prior experiences—which typically required seven to eight rounds of back-and-forth interaction—Kelly is now able to complete the task in just one or two iterations. Minor errors are easily addressed through a quick update to the graph, followed by code regeneration.

5.3 Triple Extractor

The Triple Extractor updates the LLM understanding, user intent, and their mappings each time a domain user inputs a prompt in the LLM Conversation Panel. The extraction process must be efficient to reduce the delay perceived by users (**DC2**).

An intuitive but slow implementation is the two-stage extractor used in our formative study (Fig. 4). While this approach achieves high accuracy, it requires two rounds of computationally expensive LLM calls and generates many intermediate tokens. To improve efficiency, we construct the Triple Extractor using a fine-tuned Small Language Model (SLM) that can extract triples in a single step while maintaining sufficient similarity to those extracted by the two-stage extractor, *i.e.,* faithful reflection of intent, understanding and their mapping [19]. This is achieved by a novel and cost-effective triple distillation pipeline (Fig. 8) that aligns the SLM with the LLM using data synthesized by a multi-agent system (Fig. 9).

5.3.1 Triples. For clarity, we first define a unified structure, referred to as triples, as follows:

$$triples := \{\texttt{Intent Tree}, \texttt{Understanding Graph}, \texttt{Mapping}\} \quad (1)$$

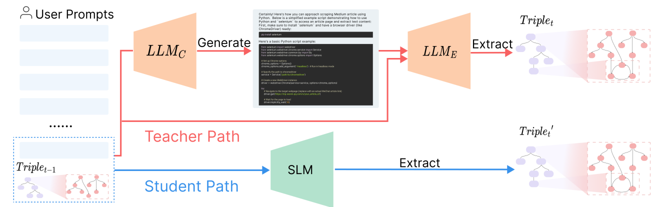

Figure 8: Triple Distillation Pipeline. It aligns the SLM in the student path with the two-stage extractor in the teacher path. The SLM can extract triples directly from prompts, bypassing intermediate code generation to speed up triple extraction.

- `Intent Tree`. A hierarchical structure expressing the user's goal decomposition. The root node represents the high-level objective, while child nodes define sub-intents and operational details. This tree explicitly externalizes user intent in a form suitable for reasoning and alignment (Fig. 7 left).
- `Understanding Graph`. A directed node-link diagram representing the LLM's internal task structure. Each node corresponds to a discrete subtask (*e.g.,* creating directories, validating input), and edges encode dependencies or data flow between them. This graph abstracts execution logic at the task level, without binding to specific code implementations (Fig. 7 right).
- `Mapping`. A cross-structure alignment that links each intent node to a corresponding node or subgraph in the Understanding Graph. This mapping ensures semantic consistency between user goals and generated code tasks, and supports graph-level operations such as task expansion, simplification, and selective editing based on user intent (Fig. 7 middle).

5.3.2 SLM Fine-tuning with a Distillation Pipeline. To address the inefficiency caused by two-round LLM calls, we follow the common practice of fine-tuning existing SLMs for specific coding tasks. We aim to fine-tune an SLM that can (1) generate triples from prompts in a single step, bypassing the need for intermediate code generation, and (2) ensure that the triples extracted directly from prompts align closely with those produced by the two-stage extractor, which depends on intermediate code. The error propagation during multi-round interaction will be mitigated by the self-healing ability of the fine-tuned SLM and users' direct modification.

We draw on knowledge distillation [11], a model compression technique that uses a teacher-student framework to transfer knowledge from a large, complex model (the teacher) to a smaller, more efficient model (the student). Knowledge distillation is well-suited for our goals because it enables the SLM (student) to replicate the outputs of the two-stage extractor (teacher) while being faster. Specifically, our pipeline has a teacher path and a student path, utilizing three models (Fig. 8): a small language model (*SLM*), a conversational language model (LLM_c), and a triple extraction language model (LLM_e). For clarity, we define the following symbols: triples T_t, prompts P_t, and intermediate code C_t, where t represents different rounds.

Teacher Path. We embed the two-stage extractor (Fig. 4) into the teacher path. First, intermediate code is generated by LLM_c, and then triples T_t are extracted using LLM_e, based on

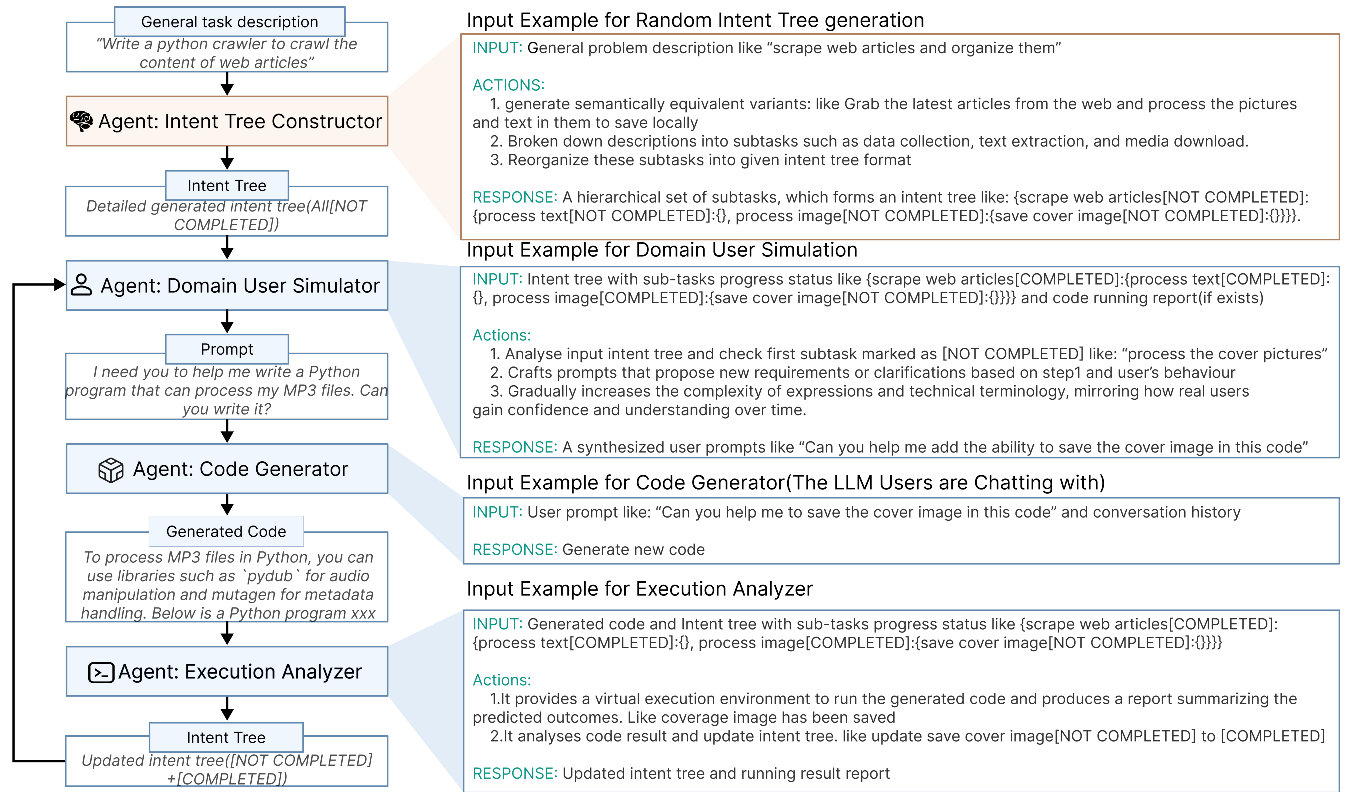

Figure 9: Multi-Agent Module Overview: This module involves four agents designed to interact with each other, simulating a domain user's experience of leveraging an LLM for code generation based on our findings on user behavior patterns.

the previous round's triples T_{t-1}, the current round's prompt P_t, and the intermediate code output C_t. Specifically, we have $T_t = LLM_e(P_t, LLM_c(P_t), T_{t-1})$. Here, LLM_e can be any sufficiently powerful model.

Student Path. To generate more accurate triples without relying on the code produced by LLM_c, it is necessary to construct a knowledge base that connects the beginning and the end of the teacher path and transfer this knowledge to an SLM. To achieve this, we design the student path and train the *SLM*. To preserve the original generalization ability of the SLM, we incorporate a LoRA [15] adapter, keeping the original parameters of the SLM fixed while tuning only the adapter's parameters. The SLM then directly generates the current round's triples $\hat{T}_t$ based on the current prompt P_t and the previous round's triples T_{t-1}, expressed as $T_t = SLM(P_t, T_{t-1})$.

Alignment. We use the mean squared error (MSE), MSE $= \frac{1}{n}\sum_{i=1}^{n}(\hat{T}_{ti} - T_{ti})^2$, to align $\hat{T}_t$ and T_t (the outputs of the student path and teacher path, respectively) across all output tokens. Here, i refers to the token index. In this way, the SLM can bridge the beginning (P_t, T_{t-1}) and end (T_t) of the teacher path without generating intermediate code.

5.3.3 Dataset Generation with a Multi-Agent Module. The distillation pipeline relies on user prompts as input. However, collecting prompts directly from real users is costly and time-consuming. To address this challenge, we introduce a multi-agent module designed to efficiently synthesize prompts. Our formative findings reveal that domain users typically initiate the problem-solving process with an LLM by crafting prompts with a general problem description. They then receive code generated by an LLM, execute it directly without closely reading the code, and refine their prompts for subsequent interactions based on where the results do not match their expectations. To simulate this process, we design the multi-agent system with four specialized agents. Below, we outline the roles of these agents and how they collaborate to generate prompts effectively, with an example shown in Fig. 9.

Agent Design. All four agents are built on LLMs, using well-designed prompts to generate realistic responses:

- *Intent Tree Constructor*. This agent takes a general problem description as input and then decomposes it into a hierarchical set of subtasks, which forms an intent tree. For example, a description of *"scrape web articles and organize them"* might be broken down into subtasks such as data collection, text extraction, and media download. For each description, we ensure the agent can generate semantically equivalent variants in different expressions, accommodating the varying ways users might phrase their tasks.
- *Code Generator*. This agent takes synthetic prompts as input and outputs generated code along with comments. It functions the same as an LLM (*e.g.,* ChatGPT) that a domain user interacts with in real-world scenarios.

- *Execution Analyzer.* This agent simulates a user executing code and comparing the results with their expectations. Specifically, it provides a virtual execution environment to run the generated code and produces a report summarizing the predicted outcomes, file changes, and any errors encountered. Additionally, it analyzes the execution results against the subtasks in the intent tree, updating the state of each subtask as either [COMPLETED] or [NOT COMPLETED].
- *Domain User Simulator*. This agent takes an intent tree with states as input and synthesizes user prompts. Specifically, it crafts prompts that propose new requirements or clarifications based on the first subtask marked as [NOT COMPLETED], simulating how a real user's intent is often shaped by the first unexpected result. The agent is instructed to craft prompts in the style of a domain user without coding expertise, using emotional markers to make user reactions more realistic. As the conversation progresses, the agent gradually increases the complexity of expressions and technical terminology, mirroring how real users gain confidence and understanding over time.

Prompt Generation with Agents. With the four agents, each execution of the following process generates a multi-round prompt history for a single problem:

- *Initialization*: The process begins with a general task description, which the *Intent Tree Constructor* uses to generate an intent tree. This tree serves as the foundation for identifying and tracking subtasks throughout the process.
- *Collaboration Loop*: The system enters an iterative loop of prompt generation, code generation, and state updates. Specifically, the *Domain User Simulator* identifies subtasks marked as [NOT COMPLETED] and crafts refined prompts. These prompts are passed to the *Code Generator*, which produces executable code and detailed explanations. The *Execution Analyzer* then runs the code, evaluates the results, and updates the task status within the intent tree. This iterative loop continues, with each agent collaboratively refining prompts and advancing task completion.
- *Termination*: The process ends when all subtasks are [COMPLETED], or if no progress occurs over five consecutive dialogue rounds. This ensures the workflow is both goal-oriented and time-efficient.

5.4 Graph Simplifier

Since users often lose track of how the complex LLM understanding graph evolves as their intent changes, they require the graph to be simplified in alignment with their updated intent (**DC3**). To address this, we propose an intent-aware graph simplification algorithm that highlights nodes directly related to intent changes and collapses other nodes (Fig. 10). It operates in two stages: intent tracking and graph simplification.

Intent tracking. The algorithm explicitly tracks user intent changes across multiple dialogue rounds. To achieve this, we construct a Nondeterministic Finite Automaton that expresses and automatically updates user intents. Initially, the algorithm analyzes the user's input, extracts the overall goal, and identifies specific subtasks or requirements, organizing this information into an intent tree. As the dialogue progresses, the Intent Tree is updated to reflect the user's latest input. These updates may involve refining existing intents, introducing new intents, merging similar intents, adjusting relationships between parent and child nodes, or confirming that no changes are necessary. After every update, the algorithm synchronizes the Intent Tree with the understanding graph, ensuring that each intent node corresponds to a subgraph within the task graph. This synchronization provides a clear mapping between user intentions and the graph structure, enabling the targeted simplification of the graph in the next stage.

Graph simplification. The algorithm employs recursive topological reduction of the hierarchical intent tree. Given a focus node set $F \subseteq \mathcal{V}_T$ ($\mathcal{V}_T$ being the set of intent tree nodes), the algorithm recursively traverses from the second-layer nodes $\{v_i^{(2)}\}$: for any node v, if its sub-tree $\mathcal{T}(v)$ satisfies $F \cap \mathcal{T}(v) = \emptyset$, the corresponding subgraph $G_v \subseteq G$ is collapsed into a supernode u, establishing a mapping $\phi : V(G_v) \rightarrow u$; otherwise, it recursively checks the direct child nodes $\{c_j\}$ of $\mathcal{T}(v)$, repeating the above judgment for each c_j. Ultimately, all subgraph structures containing members of F are preserved, unrelated branches are merged, and the edge set across subgraphs $\bigcup\{(s', t')|s' = \phi(s), t' = \phi(t), (s, t) \in E\}$ is reconstructed through ϕ, where $\phi(x) = x$ if and only if $x \notin \bigcup V(G_{v_i})$ (v_i being the merged nodes). The corresponding process is shown below (Fig. 10), ensuring graph simplification while balancing global awareness and cognitive load.

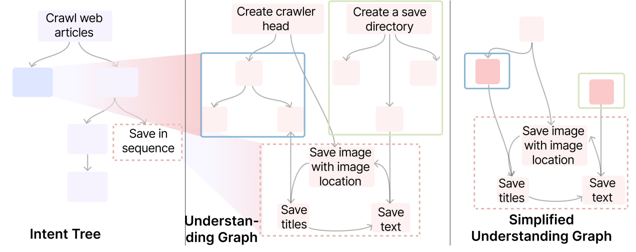

Figure 10: Intent-aware graph simplification algorithm. The left figure illustrates an intent tree, where each node corresponds to a sub-understanding graph. During the simplification process, nodes that are mapped to changes in the intent tree are directly transferred to the simplified graph (*i.e.,* red dashed box). Meanwhile, parts mapped to unchanged nodes are recursively collapsed or zoomed out (*i.e.,* blue and green boxes).

Table 3: Comparison of three fine-tuned SLMs and their zero-shot counterparts. Higher values indicate greater similarity to the ground truth produced by the two-stage extractor.

Metric	**Qwen 1.5B**		**Qwen 7B**		**LLaMa 8B**	
	Zero-shot	Fine-tuned	Zero-shot	Fine-tuned	Zero-shot	Fine-tuned
ROUGE-1	0.8503	**0.8946**	0.8590	**0.9099**	0.8621	**0.9274**
ROUGE-2	0.6941	**0.7884**	0.7259	**0.8209**	0.7277	**0.8545**
ROUGE-L	0.7872	**0.8644**	0.8203	**0.8885**	0.8214	**0.9126**
BLEU	0.8598	**0.9214**	0.8747	**0.9340**	0.8741	**0.9434**

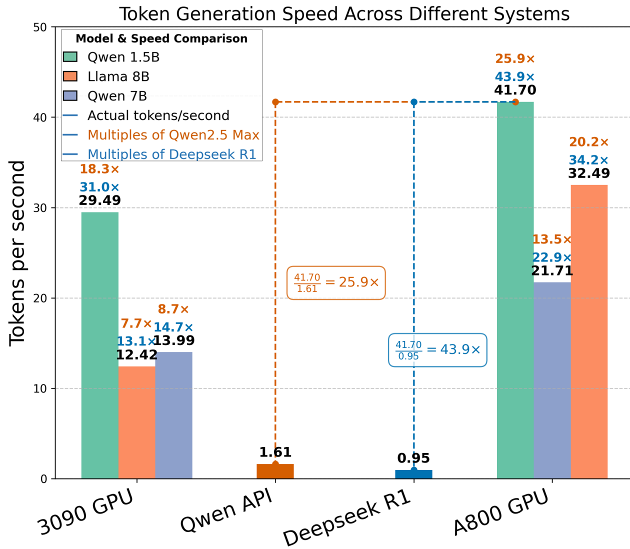

Figure 11: Efficiency gains in SLM inference speed over the two-stage extractor across different hardware settings. Higher values indicate faster extraction speed.

6 Technical Evaluation

This section evaluates how well our distillation pipeline (Sec. 5.3.2) transfers knowledge from the two-stage extractor with two-round LLM calls to an SLM (Fig. 4) for triples extraction. The pipeline is designed to fine-tune the SLM to mimic the behavior of the two-stage extractor while being more lightweight. Therefore, we assess the SLM from two aspects: (1) *similarity*, which measures how closely the triples extracted by the SLM align with those extracted by the two-stage extractor, and (2) *efficiency*, which evaluates the speed of the SLM compared to the two-stage extractor.

6.1 Similarity Between SLM Outputs and the Two-Stage Extractor Outputs

In this evaluation, we compare the outputs of the fine-tuned SLMs with their zero-shot counterparts (acting as baselines), using the outputs of the two-stage extractor as the ground truth.

Metrics. To measure similarity, we employ two widely used metrics in nature language processing: ROUGE [27] and BLEU [36]. ROUGE is a recall-based metric that evaluates how much of the reference text (i.e., the 2-stage method's outputs) is covered in the target text (i.e., the SLM's outputs). Specifically, ROUGE-1 measures overlaps of single words, ROUGE-2 measures overlaps of two consecutive words, and ROUGE-L evaluates the longest common subsequence between the texts. BLEU, on the other hand, is a precision-based metric that evaluates how well the target text matches the reference while penalizing irrelevant content. ROUGE and BLEU provide a balanced evaluation of content coverage and precision, assessing how well the SLMs replicate the LLM.

Baselines. We considered three pre-trained SLMs in our experiments: Qwen-2.5 1.5B, Qwen-2.5 7B, and Llama 8B. They served both as zero-shot baselines and as the foundation for the student models trained during the distillation process.

Experiments. Our experiments consisted of three phases: SLM fine-tuning, inference, and comparison. During SLM fine-tuning, we placed each pre-trained SLM in the student path and fine-tuned each of them on a server with three NVIDIA 3090 GPUs. This process produced three distilled SLMs corresponding to Qwen-2.5 1.5B, Qwen-2.5 7B, and Llama 8B, respectively. Zero-shot baselines used the pre-trained models without fine-tuning, enabling a comparison of improvements from the distillation process. In the inference phase, fine-tuned and zero-shot SLMs generated triples for a testing dataset of 40 samples per epoch, producing one triple per sample. The teacher path's two-stage extractor also generated one triple per sample, serving as ground truth. Lastly, we compared the triples from fine-tuned and zero-shot SLMs to the ground truths using ROUGE and BLEU, averaging the values across the 40 samples in the final epoch.

Result analysis. The results in Tab. 3 demonstrate the effectiveness of our distillation pipeline, with fine-tuned SLMs outperforming their zero-shot counterparts across all metrics. Specifically, these results highlight the pipeline's ability to enhance both content coverage and precision, achieving alignment with the two-stage extractor's outputs at over 90% similarity in most metrics. Larger models, such as LLaMa 8B, show greater improvements. While slight performance gaps remain due to the inherent randomness in language generation, they do not hinder the overall alignment and effectiveness of the fine-tuned models.

6.2 Efficiency Gains in SLM Inference Speed

In this evaluation, we compare the inference efficiency of fine-tuned SLMs to the two-stage extractor for triple extraction.

Metrics. We calculate *valid tokens per second*, which measures the number of tokens generated per second that correspond directly to triples. Unlike the commonly used token per second (*i.e., $Overtime\ Tokens/Overall\ Time$*), this metric excludes irrelevant intermediate tokens, offering a more focused comparison between the fine-tuned SLMs and the two-stage extractor.

Baselines. The two-stage extractor serves as the baseline and we consider two variants of its implementation. Specifically, the two-stage extractor involves two LLM calls: the first LLM, Deepseek R1 [14], is used to generate code, while the second LLM extracts triples based on the generated code. For the second LLM, we evaluate two options: Qwen-Max [68], an SOTA general LLM, and DeepSeek R1 [14], an SOTA reasoning LLM.

Experiments. We evaluated the efficiency of fine-tuned SLMs and the two-stage extractor variants on two hardware platforms: an NVIDIA 3090 server and an NVIDIA A800 server. The 3090 server simulates a setup for individual users, while the A800 server reflects a deployment used in industry settings. Each configuration was tested on 40 user prompts. For each prompt, we calculated valid tokens per second based on the total processing time and the number of valid tokens generated. The results were averaged across all prompts.

Result analysis. The results in Fig. 11 highlight the significant efficiency gains achieved by the fine-tuned SLMs. The distilled SLMs consistently outperformed the two-stage extractor variants

Table 4: Participant demographics in the user study.

ID	Gender	Age	Education	Domain Expertise
P1	M	24	Ph.D	Theoretical Mathematics
P2	M	23	Ph.D	Operations Research
P3	F	23	M.S.	Art
P4	F	23	M.S.	Design
P5	M	26	Ph.D	Civil Engineering
P6	M	24	Ph.D	Economics
P7	F	25	Ph.D	Linguistics
P8	F	24	M.S.	Design
P9	M	23	M.S.	Finance
P10	F	24	Ph.D	Biology
P11	M	28	M.S.	Architecture
P12	F	23	M.S.	Education

at both server settings. For example, using Llama 8B, the A800 server achieved a 22.9× speed improvement compared to the two-stage extractor with Deepseek R1, and the same model on the 3090 server achieved a 13.1× speedup. These findings indicate that our proposed method is advantageous for both individual and industrial applications.

7 Controlled User Study

To evaluate the usability and effectiveness of NeuroSync in supporting conversational LLM-based coding, we conducted a controlled study with 12 domain users with limited programming experience. The study addressed two primary research questions:

RQ1: Can NeuroSync improve domain users' ability to effectively perform conversation-based coding with LLMs?

- *RQ1.1:* Does the graph-based representation facilitate task comprehension and reduce barriers to real-time coding?
- *RQ1.2:* Can NeuroSync support more precise, controlled modifications to users' task-level understanding?
- *RQ1.3:* Does NeuroSync help domain users accomplish intent-aligned code generation more efficiently, with fewer interactions?

RQ2: How does NeuroSync affect user perceptions and behaviors during LLM-based coding interactions?

- *RQ2.1:* How does the system shift users' mental models regarding coding with LLMs?
- *RQ2.2:* How does the interaction paradigm influence user behaviors in multi-turn coding?
- *RQ2.3:* What additional or unforeseen impacts does using NeuroSync introduce?

7.1 Methods

7.1.1 ***Participants.*** We recruited 12 postgraduate students (six males, six females) aged 23 to 28 years (M = 24.17, SD = 1.53), from diverse domains such as art and design, linguistics, education, economic, finance, and architecture. Participants self-rated themselves as having general experience within their own domain (M = 3.58, SD = 0.51), but limited familiarity with coding (M = 2, SD = 0.85). They reported prior experience with LLM-powered chatbots (M = 4.16, SD = 0.73), based on a 5-point Likert scale (1 = lowest, 5 = highest). Detailed demographic information is provided in Tab. 4.

7.1.2 ***Tasks.*** Participants completed two programming tasks: a web crawler task and an audio processing task. In the *web crawler task*, they implemented a crawler that retrieved content from a specified WeChat article URL. In the *audio processing task*, participants developed a Python script that converted MP3 audio into text and extracted keywords relevant to sentiment analysis.

7.1.3 ***Baseline.*** We used a simplified version of NeuroSync , referred to as the Baseline, which excluded graphical representations while preserving conversational features (*e.g.,* ChatGPT-style prompting). The baseline included standard Python syntax highlighting and detailed inline code comments. Both experimental conditions maintained identical user interface styles to control for differences in visual aesthetics. The read-only task graph was not considered a baseline because feedforward and editing were designed as an integrated mechanism to address misalignment; evaluating them separately would not reflect their intended use.

7.1.4 ***Apparatus.*** All participants completed the tasks on desktop environments, each connected via SSH to a GPU server equipped with an NVIDIA A800.

7.1.5 ***Procedure.*** We employed a counterbalanced within-subjects design. Participants were divided into two clusters, each performing both tasks, one task using NeuroSync, the other using the baseline. Task order and condition assignments were balanced within clusters. Each session comprised an introduction (5 mins), two task sessions (10–45 mins each), post-task questionnaires (10 mins each), and a final semi-structured interview (15–25 mins). Interviews were audio-recorded, and each participant was compensated $12/hour (approximately 1.5 hours per session). To ensure the tasks remained exploratory and truly started from scratch, we designed the task session by distributing task information and completion criteria across two complementary communication channels, each serving a different purpose. Participants first received a written task description outlining high-level goals, followed by oral delivery of more detailed background context. This separation was intended to promote diverse problem-solving approaches and avoid uniform strategies, such as copying all requirements directly into the LLM. After task presentation, participants were asked to articulate their understanding to confirm comprehension before proceeding independently. Once they believed they had completed the task, the facilitator reviewed their outcome against general completion criteria and provided feedback indicating whether the task was complete or needed further refinement.

7.1.6 ***Measurements.*** We evaluated both systems across three aspects: System Usability, Cognitive Load, and Coding Efficiency. *System Usability* was measured using a custom 7-point Likert questionnaire that evaluated learnability, code comprehension, task modification accuracy, and alignment. *Cognitive Load* was assessed using the NASA-TLX, which included a measure of perceived cooperation to capture subjective mental workload differences between conditions. *Coding Efficiency* was evaluated through task completion times, durations of task-focused thinking and manipulation

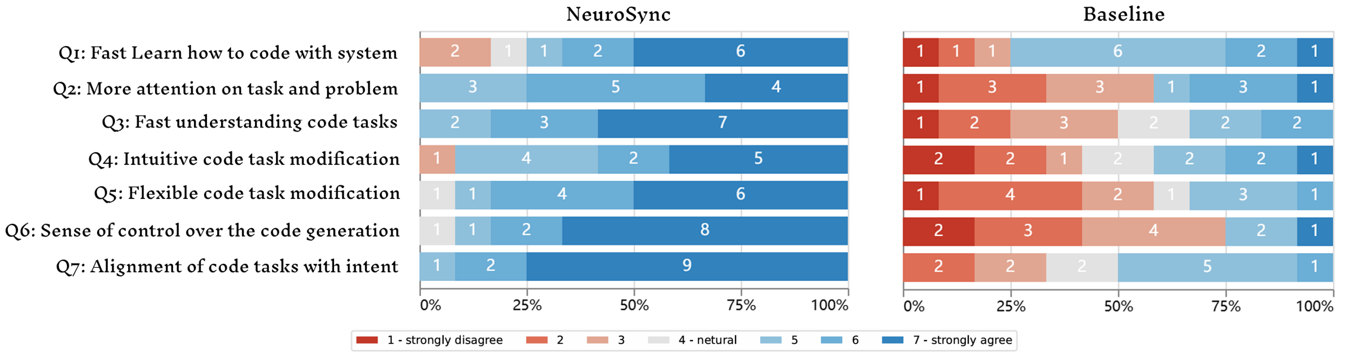

Figure 12: User ratings on the baseline and NeuroSync with a 7-point Likert scale.

Table 5: Comparison of mean scores in 7 questions (Fig. 12) between the baseline and NeuroSync with statistical analysis. *M.* denotes the mean score for each system, *Diff.* indicates the difference, and *t* and *p* report the paired t-test results.

Dim.	*M.*(*Baseline*)	*M.*(*Ours*)	Diff.	t	p
Q1	4.58	5.75	1.17	2.18	.05150
Q2	3.83	6.08	2.25	3.28	.00738
Q3	3.67	6.42	2.75	5.25	.00027
Q4	3.83	5.83	2.00	3.32	.00687
Q5	3.33	6.25	2.92	5.24	.00028
Q6	3.08	6.42	3.33	5.61	.00016
Q7	4.08	6.67	2.58	6.49	.00004

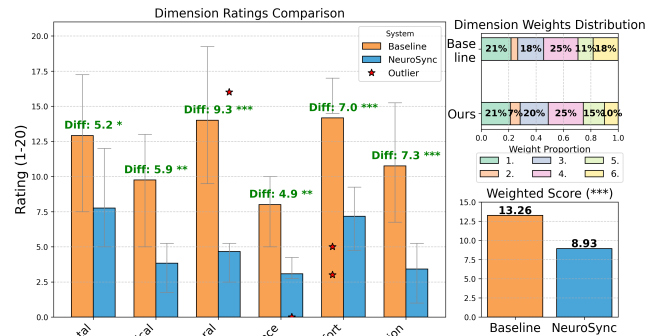

Figure 13: User ratings on the baseline and NeuroSync using NASA-TLX. Dimension weight shows each dimension's relative importance to the overall workload.

(excluding waiting time for LLM inference), and the number of LLM queries made during the tasks.

7.1.7 ***Analysis.*** Throughout the study, we collected audio recordings, interaction logs, and questionnaire data. We used an open-coding approach [20] for data analysis. For quantitative measures that met assumptions of normality and homogeneity of variance, paired t-tests were employed to assess statistical significance. Subjective ratings were evaluated using the Wilcoxon signed-rank test. Audio recordings were transcribed and categorized based on the research questions. The results are presented according to this analytical approach.

7.2 Quantitative Results

To evaluate NeuroSync against the Baseline, we assessed performance across three dimensions: *system usability for coding*, *cognitive load*, and *coding efficiency*.

7.2.1 ***System Usability for Coding (RQ1).*** To assess usability, we administered a questionnaire (Q1–Q7) using a 7-point Likert scale, covering learnability (RQ1.1), code comprehension (RQ1.1), task modification (RQ1.2), and misalignment reduction (RQ1.3). As shown in Fig. 12 and Tab. 5, NeuroSync consistently outperformed the baseline across all dimensions:

Lower Learning Threshold. NeuroSync showed an improvement in learnability (Q1; Baseline: 4.58 vs. NeuroSync: 5.75; $p =$ .051). While the addition of graph-based interactions introduced new functionality, users found the system easier to learn.

Improved Task Comprehension. Participants rated the graph representation as highly intuitive for understanding code logic (Q3; mean diff = 2.75, $p < .001$) and focusing on task structure (Q2; mean diff = 2.25, $p < .001$), enabling direct mapping between intent and generated functionality.

Enhanced Control and Modification. NeuroSync significantly enhanced users' ability to identify, modify, and refine code tasks (Q4–Q6; all $p < .01$). The explicit task-level editing mechanism allowed users to bypass abstract prompt tuning, reducing cognitive overload and frustration.

Reduced Intent–Code Misalignment. Participants reported fewer instances of code diverging from their original intentions when using NeuroSync (Q7; mean diff = 2.58, $p < .001$), validating the effectiveness of task-level alignment.

7.2.2 ***Cognitive Load (RQ2.1).*** We used NASA-TLX to evaluate perceived mental workload after completing tasks with each system. As shown in Fig. 13, NeuroSync led to significantly lower cognitive load across all six dimensions:

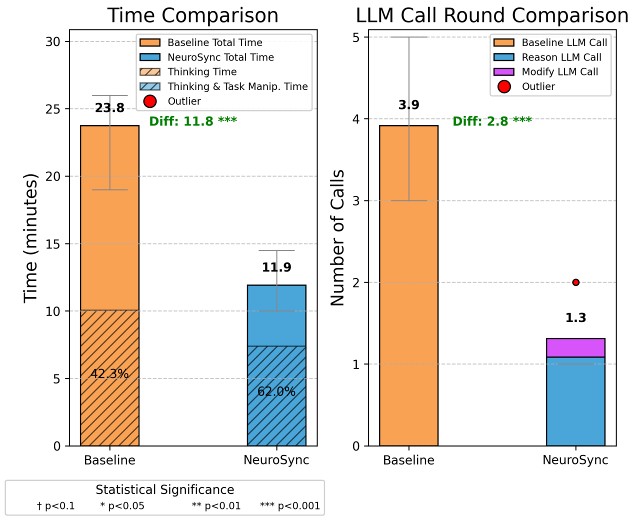

Figure 14: Quantitative results of baseline and NeuroSync on time consumption and LLM call rounds.

Overall Load Reduction. Participants experienced lower total workload with NeuroSync (Baseline: 13.26 vs. NeuroSync: 8.93; $p < .001$), especially in task time demand (D3; diff = 9.3, $p < .001$) and frustration (D6; diff = 7.3, $p < .001$), with frustration levels dropping from 18% to 10%.

Performance Not a Bottleneck. Although performance scores (D4) improved with NeuroSync (diff = 4.9, $p < .01$), this dimension showed the smallest margin. This suggests that LLMs already met baseline task requirements, and the benefit of NeuroSync was in making their output more controllable and understandable.

Shifted Cognitive Effort. Mental Demand (D1) and Effort (D5) were slightly higher than other NeuroSync dimensions (D1 = 7.75; D5 = 7.16), though still significantly lower than Baseline. This reflects the tradeoff: while NeuroSync reduces effort in code understanding, it introduces new cognitive demands in interacting with the graph.

7.2.3 ***Coding Efficiency (RQ2.2).*** We recorded task duration, user thinking/manipulation time (excluding LLM wait time), and total LLM calls per task. As shown in Fig. 14, key findings included:

Faster Task Completion. Participants using NeuroSync completed tasks significantly faster (23.8 mins vs. 13.9 mins; $p < .001$), with fewer LLM calls (3.9 vs. 1.3; $p < .001$). This demonstrates that task-level editing reduced the need for iterative prompt correction.

Increased Task Focus. With NeuroSync, users spent more time on task reasoning (62% vs. 42.3%) rather than interpreting or rewriting prompts. The externalization of task structure helped them focus on problem-solving rather than system communication.

7.3 Qualitative Insights

Through post-task interviews, participants expressed generally positive feedback on the usability and effectiveness of NeuroSync.

Improved Task Understanding. One of the most recognized advantages was its ability to assist users in understanding the structure and logic of code through the pre-generated task graph (P1–P7, P9–P10, P12). Compared with directly reading raw code, the graph provided a clearer and more intuitive overview of task flow, especially for users with limited programming experience. As P3 mentioned, *"I didn't need to understand every line of code—I just looked at the flow and knew what was going on."* This visual representation enabled participants to enter the problem-solving process more efficiently and reduced their reliance on reading and interpreting code syntax line by line.

Reduced Programming Barrier. The graph-based interaction was considered effective in lowering the entry barrier for programming tasks. Several participants (P1, P3–P5, P8–P9, P12) commented that they could express their ideas more clearly through structured tasks in the graph, rather than struggling to describe them precisely using natural language prompts. For example, P4 stated, *"The graph helped me break down what I wanted into manageable parts—without thinking about how to write it in code."* This form of externalized task structure encouraged users to reflect on and refine their intentions more systematically, especially when handling multi-step tasks.

More Accurate Modifications. Participants (P3–P9, P12) also emphasized the benefits of the graph-editing mechanism in improving task modification. Compared with traditional prompt-based interaction, direct manipulation of task nodes allowed for more accurate and targeted adjustments. As P6 said, *"Instead of rewriting everything, I just fixed the node that was wrong and got what I wanted."* Moreover, when users were uncertain about how to modify the graph manually, the natural language-based modification interface provided a convenient alternative (P2–P3, P7–P8, P10). P2 noted, *"Sometimes I didn't know how to change the graph directly, so I just typed what I wanted, and it worked."* These complementary interaction modalities enhanced users' control over task editing.

Fewer Dialog Turns. Furthermore, many participants (P2–P3, P5, P7–P10) reported that the system effectively reduced the number of interaction rounds required to align code with their intentions. The holistic representation of tasks, along with the ability to directly revise sub-tasks, allowed users to express and adjust their goals more clearly. As P8 observed, *"Normally it takes me five tries to get it right. With this, I got most of it on the first go."* This improvement in efficiency was particularly appreciated by users who had prior experience with LLM-based tools.

Effective Graph Simplification. The graph simplification mechanism was also highly valued by all participants. The integration of the intent tree with the simplified task graph enabled users to efficiently identify key task components and understand overall logic at both macro and micro levels. P9 commented, *"It's like zooming out and zooming in at the same time. I could see the big picture and the small details without getting lost."* The highlight feature, which linked simplified and detailed views, further facilitated focused editing and task tracing during multi-round interactions.

Different Interaction Patterns. The externalized LLM understanding shifted the participants' interaction patterns in two ways. The first was misalignment resolution. With the baseline, participants often addressed only partial misalignments and introduced new ones due to incomplete reviews of code and prompts. In contrast, NeuroSync's task graph allowed them to detect and resolve all

misalignments in a single pass by directly modifying relevant nodes, improving their understanding of the problem-solving process. The second was the timing of instruction. With the baseline, participants issued instructions reactively after each code generation round, resulting in more iterations. NeuroSync, however, enabled users to proactively align intent before generation—typically after reviewing the entire intent tree in the first round—thereby reducing the number of iterations. For example, P6 commented, *"I can solve many problems at once before generating code, which is really convenient compared with using ChatGPT directly."* In later rounds, the simplified graph further accelerated instruction delivery.

Changed Debugging and Testing Behaviors. We observed how NeuroSync changed developer behaviors in testing and debugging. First, users could perform code-free, task-directed testing and debugging, directly interacting with subtasks without needing to understand or modify code, as required in traditional program development. Participants (*e.g.,* , P7, P11) reported that understanding bugs and testing outcomes became less difficult, allowing them to focus more on the task itself. Second, by leveraging the understanding graph, they shifted from sequential, step-by-step debugging and testing to parallel, one-shot processes. Participants noted that compressing multi-round debugging and testing improved efficiency.

Limitations and Suggestions. Nevertheless, a few limitations were also reported. Some participants (*e.g.,* P1, P10) found that the initial learning curve of the graph interface was relatively steep, especially for users without prior exposure to task-structured programming. In addition, participants held differing opinions on the appropriate level of detail in graph nodes. While some preferred concise and abstract task descriptions, others expressed the need for more technical detail, such as variable names and code-level semantics (P6, P7). Thus, they suggested providing customizable levels of granularity and tutorial support to accommodate their diverse backgrounds and preferences.

Design Takeaways. Based on the above qualitative insights, we distill two design takeaways for future systems that wish to externalize LLM Understanding. First, since externalization allows users to shift from sequential to parallel misalignment resolution, future systems need to thoughtfully design representations (*e.g.,* , task graphs) that consolidate sequential steps in domain-specific workflows like programming. Second, direct task-intent matching involves multi-round interactions where user intents are constantly changing. The system should provide targeted information aligned with updated user intents to reduce users' cognitive load and improve system usability.

8 Discussion

We reflect on the broader implications of our approach, potential for generalization, and directions for future improvement.

8.1 Towards Personalized LLM Task Representations

While the graph-based representation in NeuroSync has been shown to support effective task viewing, tracking, and editing, our user study revealed considerable variability in user preferences regarding how such representations should be presented. Specifically, participants expressed differing needs around the level of granularity (*e.g.,* whether nodes should encapsulate high-level concepts or detailed operations), the inclusion of domain-specific metadata (*e.g.,* Python library names), and the spatial layout of task graphs. The current system adopts a uniform design grounded in formative study findings, but does not yet support individual customization. Enabling personalization could further lower the interaction threshold and increase system transparency across diverse skill levels.

To achieve this, adaptive strategies such as active learning offer promising potential [25]. By continuously collecting feedback from user interactions, such as graph edits, task confirmations, and exploration behaviors, the system could learn users' preferred presentation styles and task framing patterns. Over time, this would allow the graph interface to evolve toward more user-aligned views, enhancing usability and interpretability. Future work could explore fine-grained user modeling and incremental interface adaptation to realize personalized task understanding at scale.

8.2 Beyond Code: Generalizing the Paradigm

Although this work focuses on code generation, the core concept of externalizing LLM understanding and aligning it with user intent can be extended to a wide range of complex reasoning tasks. In domains such as writing assistance [76], data analysis [65], data visualization [47], and creative design [44, 45], users often face similar challenges of intent drift, semantic ambiguity, and nonlinear task structures. The graph-based intermediate layer proposed in NeuroSync offers a general mechanism for making LLM reasoning more accessible and editable, supporting iterative refinement across diverse contexts.

In addition, for the NLP community, our findings suggest new directions for aligning LLMs with human goals, particularly through intent-structured feedforward representations and task-aware input conditioning. Rather than optimizing whole output sequences, users can adjust high-level logic directly through graph manipulation, potentially enabling more efficient feedback collection and targeted preference learning, such as via RLHF or DPO [61]. In HCI, this work contributes to the broader conversation on feedforward mechanisms [31], demonstrating how intermediate representations can reduce interaction overhead and cognitive burden during co-creative workflows.

Moreover, the proposed paradigm of *direct intent–task matching* holds promise for real-world deployment. Its lightweight, plug-in-style implementation makes it suitable for integration with major LLM platforms, providing non-technical users with a more structured and controllable way to complete tasks. In educational settings such as programming literacy or STEM learning [35], this paradigm may also help cultivate procedural thinking by shifting attention from code syntax to task logic, an avenue that merits further exploration.

8.3 Limitations and Future Work

Limitations. Despite its advantages, NeuroSync also has several limitations. First, efficiency remains a practical concern. The current pipeline introduces latency (10–15 seconds) when generating and updating the understanding graph, which may hinder the fluidity of interaction. Improving backend processing efficiency through

caching, incremental updates, or lightweight modeling may help reduce user wait time and improve responsiveness. Second, NeuroSync is effective for single-task alignment, it is less suited to evolving multi-task scenarios where user goals shift or expand over time. Thus, complex scenarios, such as large-scale multi-file projects, were not covered in our user study and require further investigation. Lastly, we evaluated the quality of the triples generated by fine-tuned SLMs based on user feedback, which involved users directly examining the understanding graphs in Panel B and the user intent and mappings through the simplified graph in Panel C. While users provided positive feedback on the triples, future work could include a direct technical assessment of triple quality, requiring benchmarks and graph evaluation methods.

Future work. First, NeuroSync currently injects the adjusted LLM understanding as textual descriptions. Future work could investigate more advanced integration mechanisms, such as incorporating task representations into model embeddings or prompts via structured schema or feature vectors, to better preserve semantic intent. Second, though NeuroSync offers benefits for debugging and testing, it also introduces some challenges when developing software without writing code. For example, NeuroSync mainly focuses on solving task-level bugs while leaving code-level bugs for users to solve manually; combining different levels of debugging requires further exploration. Additionally, "code without code" may limit users' long-term coding skills. NeuroSync could be improved by adding features to help users learn to code and prepare for debugging in large codebases.

9 Conclusion

We address the problem of *bidirectional ambiguity* in conversational LLM programming for non-professional users. To resolve this, we propose *direct intent–task matching*, a new paradigm that externalizes LLM understanding for direct user inspection and editing before code generation. We realize this approach in *NeuroSync*, a system that combines visual representations, graph simplification, and distillation-based efficient extraction to support alignment. Through technical evaluations and a user study, we show that NeuroSync improves alignment, reduces cognitive load, and enhances coding efficiency, offering a promising direction for more transparent and accessible human–LLM collaboration.

Acknowledgments

The authors would like to thank the reviewers for their constructive feedback. The authors also wish to thank Liwenhan Xie, Bopei Nie, Jason Wong, Rui Sheng and Yanna Lin for their advice and support. This work was supported by the RGC GRF Grant 16218724.

References

[1] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111.

[2] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. 2024. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 17682–17690.

[3] Zhenni Bi, Kai Han, Chuanjian Liu, Yehui Tang, and Yunhe Wang. 2024. Forest-of-Thought: Scaling Test-Time Compute for Enhancing LLM Reasoning. *arXiv preprint arXiv:2412.09078* (2024).

[4] Yuzhe Cai, Shaoguang Mao, Wenshan Wu, Zehua Wang, Yaobo Liang, Tao Ge, Chenfei Wu, Wang You, Ting Song, Yan Xia, et al. 2023. Low-code llm: Visual programming over llms. *arXiv preprint arXiv:2304.08103* 2 (2023).

[5] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588* (2022).

[6] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Execution-guided neural program synthesis. In *International Conference on Learning Representations*.

[7] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu, et al. 2022. A survey on in-context learning. *arXiv preprint arXiv:2301.00234* (2022).

[8] Marlon Dumas and Arthur HM Ter Hofstede. 2001. UML activity diagrams as a workflow specification language. In *International conference on the unified modeling language*. Springer, 76–90.

[9] Kasra Ferdowsi, Ruanqianqian Huang, Michael B James, Nadia Polikarpova, and Sorin Lerner. 2024. Validating AI-Generated Code with Live Programming. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–8.

[10] Gaël Gendron, Qiming Bao, Michael Witbrock, and Gillian Dobbie. 2023. Large language models are not strong abstract reasoners. *arXiv preprint arXiv:2305.19555* (2023).

[11] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. 2021. Knowledge distillation: A survey. *International Journal of Computer Vision* 129, 6 (2021), 1789–1819.

[12] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.

[13] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.

[14] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[15] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. *ICLR* 1, 2 (2022), 3.

[16] Xijie Huang, Li Lyna Zhang, Kwang-Ting Cheng, Fan Yang, and Mao Yang. 2024. Fewer is More: Boosting Math Reasoning with Reinforced Context Pruning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). ACL, Miami, Florida, USA, 13674–13695.

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

[18] Pourang Irani, Maureen Tingley, and Colin Ware. 2001. Using perceptual syntax to enhance semantic content in diagrams. *IEEE Computer Graphics and Applications* 21, 5 (2001), 76–84.

[19] Alon Jacovi and Yoav Goldberg. 2020. Towards faithfully interpretable NLP systems: How should we define and evaluate faithfulness? *arXiv preprint arXiv:2004.03685* (2020).

[20] M Juliet and Strauss Corbin. 2015. *Basics of qualitative research: Techniques and procedures for developing grounded theory*. SAGE Publications, Incorporated.

[21] Tae Soo Kim, Yoonjoo Lee, Minsuk Chang, and Juho Kim. 2023. Cells, generators, and lenses: Design framework for object-oriented interaction with large language models. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. 1–18.

[22] Vu Le, Daniel Perelman, Oleksandr Polozov, Mohammad Raza, Abhishek Udupa, and Sumit Gulwani. 2017. Interactive program synthesis. *arXiv preprint arXiv:1703.03539* (2017).

[23] Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia, and Brian Ichter. 2023. Chain of code: Reasoning with a language model-augmented code emulator. *arXiv preprint arXiv:2312.04474* (2023).

[24] Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia, and Brian Ichter. 2023. Chain of code: Reasoning with a language model-augmented code emulator. *arXiv preprint arXiv:2312.04474* (2023).

[25] Dongyuan Li, Zhen Wang, Yankai Chen, Renhe Jiang, Weiping Ding, and Manabu Okumura. 2024. A survey on deep active learning: Recent advances and new frontiers. *IEEE Transactions on Neural Networks and Learning Systems* (2024).

[26] Jia Li, Ge Li, Chongyang Tao, Huangzhao Zhang, Fang Liu, and Zhi Jin. 2023. Large language model-aware in-context learning for code generation. *arXiv preprint arXiv:2310.09748* (2023).

[27] Chin-Yew Lin. 2004. ROUGE: A Package for Automatic Evaluation of Summaries. In *Text Summarization Branches Out*. ACL, Barcelona, Spain, 74–81.

[28] Michael Xieyang Liu, Advait Sarkar, Carina Negreanu, Benjamin Zorn, Jack Williams, Neil Toronto, and Andrew D Gordon. 2023. "What it wants me to say": Bridging the abstraction gap between end-user programmers and code-generating large language models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–31.

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

[30] Damien Masson, Sylvain Malacria, Géry Casiez, and Daniel Vogel. 2024. Directgpt: A direct manipulation interface to interact with large language models. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–16.

[31] Bryan Min and Haijun Xia. 2025. Feedforward in Generative AI: Opportunities for a Design Space. *arXiv preprint arXiv:2502.14229* (2025).

[32] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2024. Reading between the lines: Modeling user behavior and costs in AI-assisted programming. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. 1–16.

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

[34] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 574–584.

[35] Yang Ouyang, Leixian Shen, Yun Wang, and Quan Li. 2024. NotePlayer: Engaging Computational Notebooks for Dynamic Presentation of Analytical Processes. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. ACM, 1–20.

[36] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, Pierre Isabelle, Eugene Charniak, and Dekang Lin (Eds.). Association for Computational Linguistics, Philadelphia, Pennsylvania, USA, 311–318.

[37] Oleksandr Polozov and Sumit Gulwani. 2015. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 107–126.

[38] Udo W Pooch. 1974. Translation of decision tables. *ACM Computing Surveys (CSUR)* 6, 2 (1974), 125–151.

[39] Marko A Rodriguez and Peter Neubauer. 2010. Constructions from dots and lines. *arXiv preprint arXiv:1006.2361* (2010).

[40] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. 2024. A systematic survey of prompt engineering in large language models: Techniques and applications. *arXiv preprint arXiv:2402.07927* (2024).

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

[42] Bilgehan Sel, Ahmad Al-Tawaha, Vanshaj Khattar, Ruoxi Jia, and Ming Jin. 2023. Algorithm of thoughts: Enhancing exploration of ideas in large language models. *arXiv preprint arXiv:2308.10379* (2023).

[43] Hua Shen, Tiffany Knearem, Reshmi Ghosh, Kenan Alkiek, Kundan Krishna, Yachuan Liu, Ziqiao Ma, Savvas Petridis, Yi-Hao Peng, Li Qiwei, et al. 2024. Towards Bidirectional Human-AI Alignment: A Systematic Review for Clarifications, Framework, and Future Directions. *arXiv preprint arXiv:2406.09264* (2024).

[44] Leixian Shen, Haotian Li, Yun Wang, Tianqi Luo, Yuyu Luo, and Huamin Qu. 2024. Data Playwright: Authoring Data Videos With Annotated Narration. *IEEE Transactions on Visualization and Computer Graphics* (2024), 1–14.

[45] Leixian Shen, Haotian Li, Yun Wang, and Huamin Qu. 2025. Reflecting on Design Paradigms of Animated Data Video Tools. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. ACM, 1–21.

[46] Leixian Shen, Haotian Li, Yifang Wang, Xing Xie, and Huamin Qu. 2025. Prompting Generative AI with Interaction-Augmented Instructions. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems, CHI EA '25*. ACM, 1–9.

[47] Leixian Shen, Enya Shen, Yuyu Luo, Xiaocong Yang, Xuming Hu, Xiongshuai Zhang, Zhiwei Tai, and Jianmin Wang. 2023. Towards Natural Language Interfaces for Data Visualization: A Survey. *IEEE Transactions on Visualization and Computer Graphics* 29, 6 (2023), 3121–3144.

[48] Leixian Shen, Zhiwei Tai, Enya Shen, and Jianmin Wang. 2024. Graph Exploration With Embedding-Guided Layouts. *IEEE Transactions on Visualization and Computer Graphics* 30, 7 (2024), 3693–3708.

[49] Eui Chul Shin, Illia Polosukhin, and Dawn Song. 2018. Improving neural program synthesis with inferred execution traces. *Advances in Neural Information Processing Systems* 31 (2018).

[50] M Soegaard and RF Dam. 2007. Gulf of Evaluation and Gulf of Execution. *Retrieved June* 21 (2007), 2012.

[51] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. 404–415.

[52] Hari Subramonyam, Roy Pea, Christopher Pondoc, Maneesh Agrawala, and Colleen Seifert. 2024. Bridging the Gulf of Envisioning: Cognitive Challenges in Prompt Based Interactions with LLMs. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–19.

[53] Sangho Suh, Meng Chen, Bryan Min, Toby Jia-Jun Li, and Haijun Xia. 2024. Luminate: Structured Generation and Exploration of Design Space with Large Language Models for Human-AI Co-Creation. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–26.

[54] Sangho Suh, Bryan Min, Srishti Palani, and Haijun Xia. 2023. Sensecape: Enabling multilevel exploration and sensemaking with large language models. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. 1–18.

[55] Lev Tankelevitch, Viktor Kewenig, Auste Simkute, Ava Elizabeth Scott, Advait Sarkar, Abigail Sellen, and Sean Rintel. 2024. The metacognitive demands and opportunities of generative AI. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. 1–24.

[56] Yuan Tian, Jonathan K Kummerfeld, Toby Jia-Jun Li, and Tianyi Zhang. 2024. SQLucid: Grounding Natural Language Database Queries with Interactive Explanations. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. 1–20.

[57] Wil MP Van der Aalst. 1998. The application of Petri nets to workflow management. *Journal of circuits, systems, and computers* 8, 01 (1998), 21–66.

[58] Jo Vermeulen, Kris Luyten, Elise van den Hoven, and Karin Coninx. 2013. Crossing the bridge over Norman's Gulf of Execution: revealing feedforward's true identity. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1931–1940.

[59] Yufei Wang, Wanjun Zhong, Liangyou Li, Fei Mi, Xingshan Zeng, Wenyong Huang, Lifeng Shang, Xin Jiang, and Qun Liu. 2023. Aligning large language models with human: A survey. *arXiv preprint arXiv:2307.12966* (2023).

[60] Zhichao Wang, Bin Bi, Shiva Kumar Pentyala, Kiran Ramnath, Sougata Chaudhuri, Shubham Mehrotra, Xiang-Bo Mao, Sitaram Asur, et al. 2024. A comprehensive survey of LLM alignment techniques: RLHF, RLAIF, PPO, DPO and more. *arXiv preprint arXiv:2407.16216* (2024).

[61] Zhichao Wang, Bin Bi, Shiva Kumar Pentyala, Kiran Ramnath, Sougata Chaudhuri, Shubham Mehrotra, Xiang-Bo Mao, Sitaram Asur, et al. 2024. A comprehensive survey of LLM alignment techniques: RLHF, RLAIF, PPO, DPO and more. *arXiv preprint arXiv:2407.16216* (2024).

[62] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

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

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

[65] Liwenhan Xie, Chengbo Zheng, Haijun Xia, Huamin Qu, and Chen Zhu-Tian. 2024. Waitgpt: Monitoring and steering conversational llm agent in data analysis with on-the-fly code visualization. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. 1–14.

[66] Litao Yan, Alyssa Hwang, Zhiyuan Wu, and Andrew Head. 2024. Ivie: Lightweight anchored explanations of just-generated code. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–15.

[67] Youfu Yan, Yu Hou, Yongkang Xiao, Rui Zhang, and Qianwen Wang. 2024. Knownet: Guided health information seeking from llms via knowledge graph integration. *IEEE Transactions on Visualization and Computer Graphics* (2024).

[68] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115* (2024).

[69] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems* 36 (2024).

[70] Ryan Yen and Jian Zhao. 2024. Memolet: Reifying the Reuse of User-AI Conversational Memories. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. 1–22.

[71] Ryan Yen, Jiawen Stefanie Zhu, Sangho Suh, Haijun Xia, and Jian Zhao. 2024. CoLadder: Manipulating Code Generation via Multi-Level Blocks. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. 1–20.

[72] Vahan Yoghourdjian, Yalong Yang, Tim Dwyer, Lee Lawrence, Michael Wybrow, and Kim Marriott. 2020. Scalability of network visualisation from a cognitive load perspective. *IEEE transactions on visualization and computer graphics* 27, 2 (2020), 1677–1687.

[73] Zishun Yu, Yunzhe Tao, Liyu Chen, Tao Sun, and Hongxia Yang. 2023. B-Coder: Value-Based Deep Reinforcement Learning for Program Synthesis. *ArXiv* abs/2310.03173 (2023). https://api.semanticscholar.org/CorpusID:263671681

[74] Rui Zhang, Ziyao Zhang, Fengliang Zhu, Jiajie Zhou, and Anyi Rao. 2024. Mindalogue: LLM-Powered Nonlinear Interaction for Effective Learning and Task Exploration. *arXiv preprint arXiv:2410.10570* (2024).

[75] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L Glassman. 2020. Interactive program synthesis by augmented examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 627–648.

[76] Zheng Zhang, Jie Gao, Ranjodh Singh Dhaliwal, and Toby Jia-Jun Li. 2023. VISAR: A Human-AI Argumentative Writing Assistant with Visual Programming and Rapid Draft Prototyping. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. ACM, New York, NY, USA, 1–30.

[77] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. 2022. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625* (2022).