SPROUT: an Interactive Authoring Tool for Generating Programming Tutorials with the Visualization of Large Language Models

Yihan Liu, Zhen Wen, Luoxuan Weng, Ollie Woodman, Yi Yang, Wei Chen.

Abstract—The rapid development of large language models (LLMs), such as ChatGPT, has revolutionized the efficiency of creating programming tutorials. LLMs can be instructed with text prompts to generate comprehensive text descriptions for code snippets provided by users. However, the lack of transparency in the end-to-end generation process has hindered the understanding of model behavior and limited user control over the generated results. To tackle this challenge, we introduce a novel approach that breaks down the programming tutorial creation task into actionable steps. By employing the tree-of-thought method, LLMs engage in an exploratory process to generate diverse and faithful programming tutorials. We then present *SPROUT*, an authoring tool equipped with a series of interactive visualizations that empower users to have greater control and understanding of the programming tutorial creation process. A formal user study demonstrated the effectiveness of *SPROUT*, showing that our tool assists users to actively participate in the programming tutorial creation process, leading to more reliable and customizable results. By providing users with greater control and understanding, *SPROUT* enhances the user experience and improves the overall quality of programming tutorial.A free copy of this paper and all supplemental materials are available at https://osf.io/uez2t/?view_only=5102e958802341daa414707646428f86.

Index Terms—Large language model, programming tutorial, authoring tool, interactive visualizations.

1 Introduction

Programming tutorials are used to teach learners coding skills and how to solve programming problems. Numerous programmers disseminate their programming expertise through authoring various tutorials [1], such as blog articles, online videos, programming games, etc. As the demand to learn programming has expanded, there has been a rise in the number of tutorial authors (*e.g.*, developers [2], computer science educators [3], data scientists [4], etc.), and the research on improving authoring experiences and the workflow of programming tutorials.

The recent prevalence of **L**arge **L**anguage **M**odels (LLMs) has revolutionized the conventional process of authoring articles [5]–[7]. Similarly, programming tutorial authors can provide LLMs with code snippets and text instructions to generate tutorials. The prominent ability of LLMs diminishes the effort for authors to author programming tutorials.

However, the text generated by LLMs is not always perfect or error-free [8]. Sometimes the exhaustive LLM-generated content is also not aligned with users' expectations. These problems call for further refinement on the auto-generated content. Owing to the inherent non-transparency of the generation process of LLMs, users lose access to the intermediate steps of the creation, leading to limited control and customizability when authoring articles [6]. Additionally, to achieve ideal modifications often necessitates users' continuous interactions with LLMs via text-based conversation interfaces, which can not adequately support the complex requirements of users seeking to fine-tune tutorials. It is even more time-consuming when dealing with programming tutorials, as it requires precise verification for the correctness and consistency.

The aforementioned limitations pose challenges in the process of authoring programming tutorials using LLMs. These challenges have an impact on the overall quality and reliability of LLM-generated programming tutorials. Some previous works delve in training LLMs to generate code-related content [9], [10] or proposing advanced prompting methods to promote the generation quality of LLMs [11]–[13]. However, to tackle with programming tutorial task, these approaches are not out-of-box solutions for generating faithful content accurately under human's interventions. Several interactive systems have also been developed to enhance the efficiency of interactions with LLMs. While proving useful, these systems primarily concentrate on specific needs, such as creative writing or data-driven article authoring [6], [14]. Unfortunately, these approaches are not suitable for the context of programming tutorial authoring, which necessitate accurate information organized in a more coherent way aligning with the source code. Therefore, novel approaches are needed that account for the need to generate customizable programming tutorial and guarantee its faithfulness and consistency with the source code.

Recent studies propose instructing LLM to *"think step by step"* to enhance the faithfulness and consistency of the generated results [15], [16]. Inspired by this, we decompose the tutorial authoring tasks into a step-by-step exploratory process towards final document accomplishment. This approach has potential to enables users to attain a lucid com-

Yihan Liu, Zhen Wen, Luoxuan Weng, Ollie Woodman, Yi Yang, and Wei Chen are with the State Key Lab of CAD&CG, Zhejiang University. E-mail: {lyh1024 | wenzhen | lukeweng | orw | yang-yi | chenvis}@zju.edu.cn.
Wei Chen is the corresponding author.
Manuscript received April 19, 2021; revised August 16, 2021.

prehension and control over the intermediate steps throughout the entire creation process. Thus, this study investigates methods to enhance user comprehension and control over the text generation process with LLMs, with the ultimate goal of improving the quality and experience of authors. For the sake of better understanding the underlying challenges in the authoring process, we conducted a formative study with 6 experienced tutorial authors to have a grasp of the writing process and the challenges they encounter when transitioning from their conventional workflow to the authoring workflow with LLMs. Participants reported that they can not effectively engage in the creation process using typical conversation interfaces due to the difficulty of controlling the generated content solely with text. The lack of visual cues between source code and generated text also led to the overhead in comprehending and verifying the tutorial.

Consequent upon the the findings of formative study, we designed *SPROUT*[1] (***S**tep-by-step **pro**gramming **tut**orial authoring tool*), an interactive system that breaks down the tutorial creation task into actionable steps. These steps are accompanied by interactive visualizations that empower users to have a greater control and understanding over the exploratory process, ending with more accurate and reliable results. *SPROUT* adopts tree-like prompting strategies to allow LLMs to generate diverse and faithful tutorial content more reasonably, which serves as a entry point for subsequent user interventions to make accurate adjustment on tutorial in different aspects. A series of purpose-specific interactions are also provided on the basis of the real-time visualizations of the text generation process with LLMs, aiding users in crafting the tutorial from multiple levels. To facilitate an intuitive understanding of the generated content, *SPROUT* leverages the inherent connections between the source code and tutorial, which are distilled by LLMs and demonstrated with visual representations to users, enabling the explicit comprehension of the creation process and model's behavior. To evaluate the effectiveness of *SPROUT*, we conducted a technical evaluation and a user study. The technical evaluation demonstrated that *SPROUT* achieved accurate text-code connections. The user study revealed that *SPROUT* effectively assisted users in generating, modifying, and understanding programming tutorials with LLMs, ultimately leading to more reliable and customized results. In summary, this paper makes the following contributions:

- Understanding of the challenges inherent in the process of authoring programming tutorials with LLMs.
- *SPROUT*, an interactive system that utilizes novel prompting strategies and interactive visualizations to facilitate the step-by-step generation of programming tutorials with LLMs.
- A technical evaluation demonstrating the effectiveness of our prompting methods and a user evaluation confirming the usability of *SPROUT* and its support for the programming tutorial authoring workflow.

[1] *SPROUT* is derived from*"**S**tep-by-step **pro**gramming **tut**orial authoring tool"*, with the metaphorical representation of the thoughts generating process of Large Language Models, symbolizing the system's goal to enable users a more controllable experience by utilizing the interactive visualizations of this process.

2 Related Work

2.1 Programming Tutorial Authoring Tools

Tutorials are a prevalent medium through which developers disseminate coding and programming knowledge. Typically, a tutorial consists of source code, textual explanations, code examples, and multimedia components such as images and videos [17]. Crafting a high-quality tutorial necessitates consideration of various perspectives [1], [18], [19]. Even experienced authors require significant time to structure and present the tutorial content. Therefore, numerous tools have been designed to improve the tutorial authoring process. In-editor plugins like JTutor [20] and JTourBus [21] facilitate the construction of Java tutorials within code editors. Some studies leverage interaction provenance collected from operating systems [22] or web-based editors [23] to capture the programming process and produce tutorials. For example, VT-Revolution [24] offers interactive video tutorial creation based on user actions, while Chat.codes [25] promotes collaborative code annotations and explanations. Other works support users in drafting code examples by propagating changes to multiple code versions [26] or extracting concise code using a mixed-initiative strategy [27].

Among the many forms of tutorials, the step-by-step structure is particularly accessible for novice authors due to its straightforward nature. Also, describing code as *"a tour through the source code"* [21] proves beneficial for beginners in programming. Therefore, our primary emphasis is on the authoring of textual content in step-by-step programming tutorials, where LLMs exhibit optimal performance. We investigate the pain points in the authoring process and leverage LLMs to assist users in efficiently generating, modifying and understanding tutorial content, which furthers the line of research on programming tutorial authoring.

2.2 LLMs for Document Generation

The remarkable generative capabilities of LLMs have spurred the proliferation of LLM-based applications across various domains in writing scenarios. In the domain of code-related content generation, several studies investigate the potential of LLMs in crafting code explanations by comparing LLM-generated content with that sourced from students [28] or previous approaches [29], and validate its accuracy and comprehensibility. Some endeavors have integrated these auto-generated code explanations into educational scenarios. For instance, MacNeil et al. incorporate LLM-generated code explanations within a web software development e-book [30]. Other efforts utilize LLMs to generate a myriad of CS learning materials including programming assignments and multifaceted code explanations [31], [32]. Additionally, Sarsa et al. explore LLMs' abilities in creating programming exercises and code explanations, with evaluations indicating that the generated content is novel, sensible and ready to use in certain cases [33]. There are also developer tools for flexible code comment generation, for instance, GitHub Copilot [34] provides comprehensive text content pertaining to the code context. Some researchers have developed LLM-based agents [35], [36] responsible for documenting in software engineering, showcasing substantial potential in authoring code-related content.

Nonetheless, previous works mostly focus on generating high-quality content, assessing the thoroughness and accuracy of code explanations. For effective presentation to readers, such content often necessitates additional refinements, either through manual modifications or carefully-designed prompts. NLP researchers are actively examining diverse prompt strategies and frameworks to optimize LLMs' performance [15], [37] or mitigate *hallucinations* [38], [39], seeking to harness their full potential. Our work builds on top of these endeavors by employing advanced prompt strategies to more effectively incorporate LLMs into the programming tutorial authoring workflow, ending with reliable and high-quality tutorials that meet users' expectations.

2.3 Visual Interfaces for LLMs

Despite the impressive capabilities of LLMs, traditional text-based conversational interfaces present challenges when it comes to complex tasks that require the integration and manipulation of diverse data types [6], [16], [40]–[42], as they lack the capacity for direct multimodal data manipulation and the visualization necessary for complex, dynamic content creation and information synthesis [43], [44]. Therefore, a variety of visual interfaces have been developed to enhance user interactions with LLMs. For instance, researchers have designed interactive interfaces for visual programming prompt chains to lower the barrier for non-AI-experts [45]–[47]. Some works aim to assist users generate better prompt with fow-shot examples [48], coordinated visualizations [49] or direct manipulation actions [50] in graphical interfaces. In terms of diverse output representations, Graphologue [43] translates LLM responses into graphical diagrams, while Sensecape [51] constructs a multi-level abstraction for information exploration and sensemaking. Both of them facilitate the information-seeking process. Other works focus on designing interfaces for specific writing tasks, such as email writing [52], story creation [14], argumentative writing [16], scientific writing [5] and journalistic angle ideation [53]. These interfaces ensure users achieve ideal results through iterative interactions. Notably, DataTales [6] introduces more flexible and intuitive interactions for authoring data-driven narratives, while TaleBrush [54] utilizes line sketching interactions with GPT-based models to control and understand a protagonist's fortune in co-created stories. More recently, Kim et al. develop a framework for object-oriented interaction with LLMs, which can generalize to a wider range of writing tasks [55].

Motivated by these prior studies, our work aims to facilitate the process of authoring programming tutorials with LLMs. Despite the similarities to various writing tasks, the unique aspects such as writing style, the process of creation, the nature of data involved, and the necessity for consistency and faithfulness make their methodology unsuitable for the task of authoring programming tutorials. Through a set of interactive visualizations, our approach empowers users to actively participate in the generation and refinement of tutorials while maintaining flexibility and precision in their control on the process. Additionally, we extract and present visual representations of the connections between the source code and the corresponding LLM-generated results, facilitating users' explicit understanding of the tutorial content.

3 Formative Study

We conducted a formative interview study to analyze the challenges encountered in tutorial creation and workflow with LLMs, from which we distilled four essential design requirements to improve the authoring process.

3.1 Participants and Procedure

In order to comprehend the challenges and difficulties encountered across various tutorial authoring scenarios, six experienced tutorial authors were interviewed (age from 23 to 28). Their wide-ranging expertise in authoring diverse tutorials for various users spans from novice to expert levels. Thus, the insights derived from their collective experiences provide a more comprehensive and universally applicable understanding, beneficial to programming tutorial authors of various proficiency levels. Four of them are experienced programmers (E1-4) and two are educators. One of the two educators teaches computer science (E5), while the other instructs competitive programming (E6). All of them have more than three years of experience authoring programming tutorials. In addition, they all have used ChatGPT in recent months. Before the interview, we collected twenty text-based programming tutorials from various sources, including Stack Overflow, Wikipedia, GeeksforGeeks, etc., as well as those auto-generated by LLMs. In our formal interviews, we first inquired about the authors' writing process and commonly-used tools for creating tutorials, then presented them with all of the collected tutorials and solicited their opinions on the advantages and downsides of each. Afterwards, participants were asked to create programming tutorials utilizing ChatGPT and document editors. Finally, we collected feedback on their authoring experience, focusing on how they utilized LLMs in their workflow and what made it challenging to create high-quality tutorials in this process. The interviews were conducted through online meetings and lasted 45 to 60 minutes.

3.2 Findings

All participants tried to author programming tutorials with LLMs. They provided the source code, retrieved the resulting tutorial, and iteratively modified it by seeking ChatGPT's assistance. We found that they usually managed to generate different versions with multiple categories of content to enrich the tutorial. The five most common categories among them are title, background, code explanation, notification, and summary. The behavior analysis indicated a frequent tendency among participants to leverage LLMs for generating content, structuring the tutorial, and polishing details. However, during the tutorial authoring process with LLMs, participants commonly encountered challenges that impacted production efficiency.

Difficult to generate tutorials that accurately reflect diverse coding scenarios. This challenge stems from a need to encompass a wide array of programming paradigms, languages, and problem-solving techniques within instructional content. It potentially creates gaps in understanding for learners who come from different experience levels or who are looking to apply their skills to a variety of contexts. As such, during the authoring process with ChatGPT, participants often complained about the difficulty to obtain tutorials aligning with their mental model. For example, E5 hoped that ChatGPT could only explain the crucial parts of the code since the tutorial is for experienced programmers, but only found the output elaborated on some unimportant details, calling for additional effort in modifying the prompt to generate tutorials again. Some participants proposed the desire to obtain tutorials with different frameworks, which can *"provide more inspiration when having no ideas about what to do next"* (E4). Apart from that, during the iterative tutorial authoring process, some participants noted that the content generated by ChatGPT occasionally deviated from being faithful to the original source code. It is significant to keep the tutorial consistent with the original source code. Such experiences underscore the need for reasonable prompting strategies for generating rich and reliable content in diverse writing scenarios with different requirements.

Lack of flexible interactions for tutorial generation and modification. As auto-generated content seldom fully met participants' expectations, they had to iteratively modify the tutorial content during the exploration. Some of them sought to simplify this process using ChatGPT, but only to find it *"inconvenient and confusing"* (E6). E6 reported that combining two paragraphs to shorten the document necessitated a meticulously crafted prompt, otherwise ChatGPT's response was merely *"a non-sensible mess"* (E6). Additionally, participants mentioned concerns over the tediousness of constantly *"switching back and forth between the editor and ChatGPT"* (E4), crafting prompts, and ensuring that the new content seamlessly integrated into the existing context.

Significant overhead for understanding and verifying the tutorial. The linear conversation interface of ChatGPT, where user inputs and LLM outputs appear sequentially, becomes problematic due to the lack of visual linkages between code and text, particularly in multi-step interactions with iterative refinement. They usually needed to *"browse the whole document"* (E5) to discern which code segment each paragraph is describing, as ChatGPT omitted code references. The situation was exacerbated during modifications, as responses often excluded the original content and code, forcing participants to *"painstakingly match code snippets to their text because of their separate presentations"* (E3). Moreover, the independently showcased nested code often lost its proper indentation, pushing participants to trace back to its context. This caused *"additional mental overload to recognize the correct execution scope"* (E2). Collectively, participants felt that these unpredictabilities *"somewhat negated the intended efficiency gains from Large Language Models"* (E6).

3.3 Design Requirements

To tackle the problems concluded above, we aim to implement an interactive system to improve the authoring process with LLMs and obtain desirable programming tutorials. The **D**esign **R**equirements can be summarized as follows:

DR1. Generating diverse and faithful content. In programming tutorial authoring, users tend to utilize LLMs to generate a wide range of content, including variations in structure, level of detail, and other aspects, in order to inspire their writing ideas. Simultaneously, they require the generated content aligns precisely with the intended functionality and logic. Therefore, the system should generate diverse content while maintaining its faithfulness to the original source code provided by users.

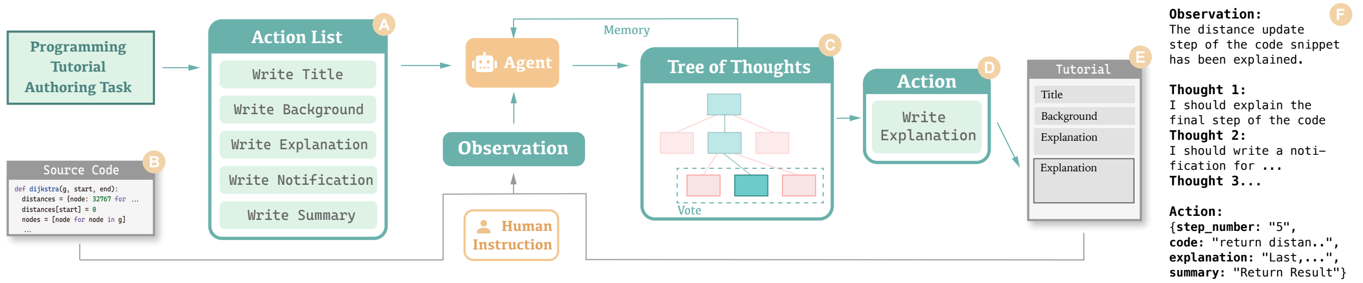
DR2. Supporting precise and in-context modification. Users often engage in iterative interactions with LLMs through text-based conversations to refine the generated tutorial content according to their preferences, mainly focusing on the aspects of tutorial structure and content details. However, even slight modifications in the input prompts can result in significant and unforeseen changes in the entire generated output. To maintain the coherence and accuracy of the final output, the system should support users in making precise modifications while preserving the contextual integrity of the existing generated tutorial content.

DR3. Maintaining explicit connection between inputs and outputs. Users usually need to carefully read the source code and outputs with the purpose of understanding and verifying the LLM-generated content. However, in linear conversation interfaces, this task becomes time-consuming and challenging. Additionally, the lack of transparency and traceability between user inputs and model outputs can result in inadequate trust and confidence towards the final tutorial. Therefore, in this system, the tutorial content should be augmented with explicit visual representations of connections between the generated content and its related code snippets, which should be extracted and maintained by the system. This allows users to easily comprehend and verify the tutorial and the reasoning behind it.

DR4. Providing intuitive and flexible interactions. Utilizing natural language as the channel to communicate with LLMs poses challenges for users to clearly convey their requirements and ideal output. In order to acquire qualified and satisfactory tutorials, users have to craft them continually, by some cumbersome editing operations and iterations on prompts for generation and modifications on the draft from ChatGPT. Therefore, to circumvent these issues, the system should be synthesized with intuitive and flexible interactions to eliminate users' efforts for manual adjustment on content or revision on prompts, for instance, enable direct manipulation of the source code or generated textual content. This allows authors to focus more intently on the programming tutorial itself.

4 Prompting for Programming Tutorial Authoring

A primary goal of our research is to make the LLM generation process more transparent and controllable. To achieve this, we adopt the tree-of-thought (ToT) methodology [15] to prompt models. It enables models to take intermediate steps towards completing the tutorial. By adopting the ToT methodology, we allow models to explore multiple choices during generation process. This promotes a more interactive and iterative approach to tutorial authoring, empowering users to have greater control over the generated content. In this section, we present our proposed prompt design that combines the ToT methodology with the specific requirements of programming tutorial authoring. Figure 1 illustrates the framework of our prompt strategies. Our prompt strategies are developed and tested on `gpt-3.5-turbo` version of ChatGPT [56], one of the most popular LLMs. Please refer to supplemental materials for original prompts.



```
def dijkstra(g, start, end):
  distances = {node: 32767 for ...
  distances[start] = 0
  nodes = [node for node in g]
  ...
```

```
Observation:
The distance update
step of the code snippet
has been explained.

Thought 1:
I should explain the
final step of the code
Thought 2:
I should write a noti-
fication for ...
Thought 3...

Action:
{step_number: "5",
code: "return distan..",
explanation: "Last,...",
summary: "Return Result"}
```

Fig. 1. The framework of *SPROUT*'s prompt strategies. We break down the programming tutorial creation task into actionable steps (A) which are provided as initial system prompts to the agent. During the generation process, the agent receives the source code (B) and user instructions, generating multiple potential thoughts (C) based on its memory and the observation derived from the source code and current tutorial content (E). Then it takes the proper action (D) and plans for subsequent steps. The iterative process continues until the tutorial is complete. (F) shows an example iteration.

4.1 Generation of Thoughts

We envision the LLM as an agent that takes on the role of making decisions on how to plan and execute steps to accomplish programming tutorials. The agent is instructed to think in a step-by-step manner to generate faithful content, and expand thoughts in a tree-like structure to produce diverse tutorial content (DR1).

Dividing task into actions. We break down the programming tutorial creation task into discrete and actionable steps. This allows the agent to consider these steps as fundamental units of a plan. Through the formative study (section 3), we have identified 5 key components of tutorial content, including the title, background, code explanation, notification, and summary. Consequently, we define a list of actions that correspond to writing different content within the tutorial, e.g., 'write title' or 'write background'. The action list and the descriptions of actions are provided to the agent before solving tasks.

Thinking step by step. To guide the decision-making process leading to faithful results, we instruct the agent to think step-by-step following the ReAct (Reason and Act) paradigm [37], which requires the agent to solve the task in a specific order: *observation*, *thought*, *action*. At each step, the agent first presents the observation derived from the source code and the current content of tutorial. Subsequently, it generates the thought of the appropriate next action to be taken with a reasonable justification. The agent then takes the action and continues to plan for the next step. This process continues iteratively until the tutorial creation process is completed. This systematic approach allows the agent to make informed decisions at each stage of the tutorial creation process, resulting in faithful and reliable outcomes.

Expanding tree of thoughts. We integrate the tree-of-thought methodology in the decision-making process to enable the agent to generate multiple potential thoughts for consideration, leading to diverse content in the generated tutorial. For example, after explaining a specific code snippets, the agent may generate thoughts such as writing a notification highlighting common mistakes related to the code or directly providing a summary to conclude the tutorial. Consequently, the agent votes on the generated thoughts and make a decision on the next action to take. To provide an exploratory space for authors, we maintain the tree of thoughts as a memory space for the agent. Users are allowed to navigate through the tree of thoughts to explore multiple potential results of generation.

4.2 Manipulation of Model Actions

The above prompt strategies enables a completely automatic process to create tutorial without human intervention. We propose to perform in-context interventions by inserting instructions in the gap of each step to manipulate the model actions. This capability enables authors to make contextual adjustments to the tutorial materials, resulting in more tailored and customized programming tutorials (DR2).

In particular, we design prompt strategies to support authors in the process of generating content (generation), adjusting the structure of tutorial (organization), and refining the detail of content (refinement), which are derived from the formative study (section 3). These strategies serve as underlying methods within features of our system (see section 5):

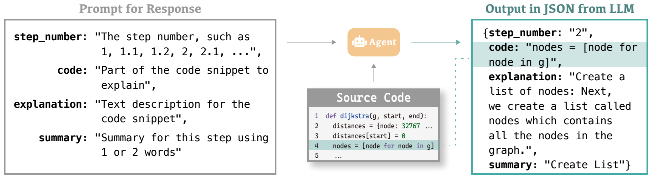
- **Generation:** "*The next step should be to write for* ⟨ *code* ⟩". This prompt is used to enable users to define a specific range of code to explain. It instructs the agent to thought closely associated with the specific code fragments when generating new content for tutorial.
- **Organization:** "*Explain* ⟨ *code* ⟩ *in the next multiple steps*" or "*Explain* ⟨ *code* ⟩ *in one paragraph*". These prompts are used to adjust the structure of the tutorial. They assist users in organizing the tutorial by separating the explanation of important code into multiple steps or summarizing less important code in a concise manner.
- **Refinement:** "*Explain* ⟨ *code* ⟩ *in the style of* ⟨ *style* ⟩", "*Refine the last step* ⟨ *'shorter' — 'longer'* ⟩". These prompts are used to refine the writing style or level of detail in a paragraph, which allows users to modify the tutorial content in a more granular manner.

These prompt strategies empower authors to iteratively refine and customize the programming tutorial content according to their individual preferences and needs.

4.3 Extraction of Text-Code Connections

The relationship between the LLM-generated tutorial and its underlying code are not always clarified. The extraction of text-code connections strategy focuses on identifying and establishing links between relevant textual explanations and corresponding code snippets (DR3).

We instruct the agent to present each code-related text accompanied with its connection to the code. The agent is required to provide extra information when writing textual descriptions for a specific piece of code, e.g., *step number*, *code*, *explanation*, and *summary*. We then link the reference code with the source code using string matching. Figure 2 showcases an example of our method. This strategy is seamlessly integrated into the generation process, and also ensures that textual explanations are closely associated with the relevant code, enhancing the overall comprehensibility and effectiveness of the tutorial materials. The tree of thoughts annotated with extracted connections helps authors structure their thoughts and logically sequence the tutorial materials. It assists authors in creating and organizing tutorial content effectively, aiding authors in maintaining coherence and clarity throughout the tutorial.



```
step_number: "The step number, such as
              1, 1.1, 1.2, 2, 2.1, ...",
       code: "Part of the code snippet to
              explain",
explanation: "Text description for the
              code snippet",
    summary: "Summary for this step using
              1 or 2 words"
```

```
def dijkstra(g, start, end):
  distances = {node: 32767 ...
  distances[start] = 0
  nodes = [node for node in g]
  ...
```

```
{step_number: "2",
 code: "nodes = [node for
 node in g]",
 explanation: "Create a
 list of nodes: Next,
 we create a list called
 nodes which contains
 all the nodes in the
 graph.",
 summary: "Create List"}
```

Fig. 2. An example output with text-code connection. The agent is instructed to provide text-code connection information within its response.

5 SPROUT

We develop *SPROUT*, a prototype system that supports efficient and flexible step-by-step programming tutorial authoring with LLMs according to interactive visualizations of the intermediate generation process. In this section, we first present an overview of the system, introducing the interface design and visual representations in section 5.1. Next, we introduce a series of features that are designed for enhancing user experience in the process of tutorial authoring (section 5.2-section 5.6).

5.1 Overview

Our system consists of 6 views as demonstrated in fig. 3. The **code view** (fig. 3 A) allows users to enter the source code they intend to describe about. The **tutorial view** (fig. 3 B) displays the text content generated by LLM with a block-based structure, where a block presents a paragraph. Between the code view and tutorial view lies the **chain view** (fig. 3 C), which serves as a connector, assisting users to navigate between code and tutorial. Within this view, each node is a simplified visual representation of the discrete output units generated during the step-by-step creation process. Each node displays information about a specific paragraph and the corresponding code it describes. The chain is concatenated by these nodes, which collectively represent the entire tutorial.

The **outline view** (fig. 3 D) visualizes the intermediate generation steps of LLM in the form of a tree graph. It consists of nodes and branches, illustrating the hierarchical structure of the LLM's thought process. The **branch view** (fig. 3 E) enables users to switch between different branches to explore alternative thought paths and make adjustments to the structure of generated tutorial. The **node space view** (fig. 3 F) offers alternative content options for a selected node. Users have the flexibility to choose from these alternatives and customize them in multiple dimensions.

5.2 Content Generation from Code Snippets

SPROUT supports two interactions that enable the generation of diverse and user-customized content from the source code provided by users (DR1, DR4).

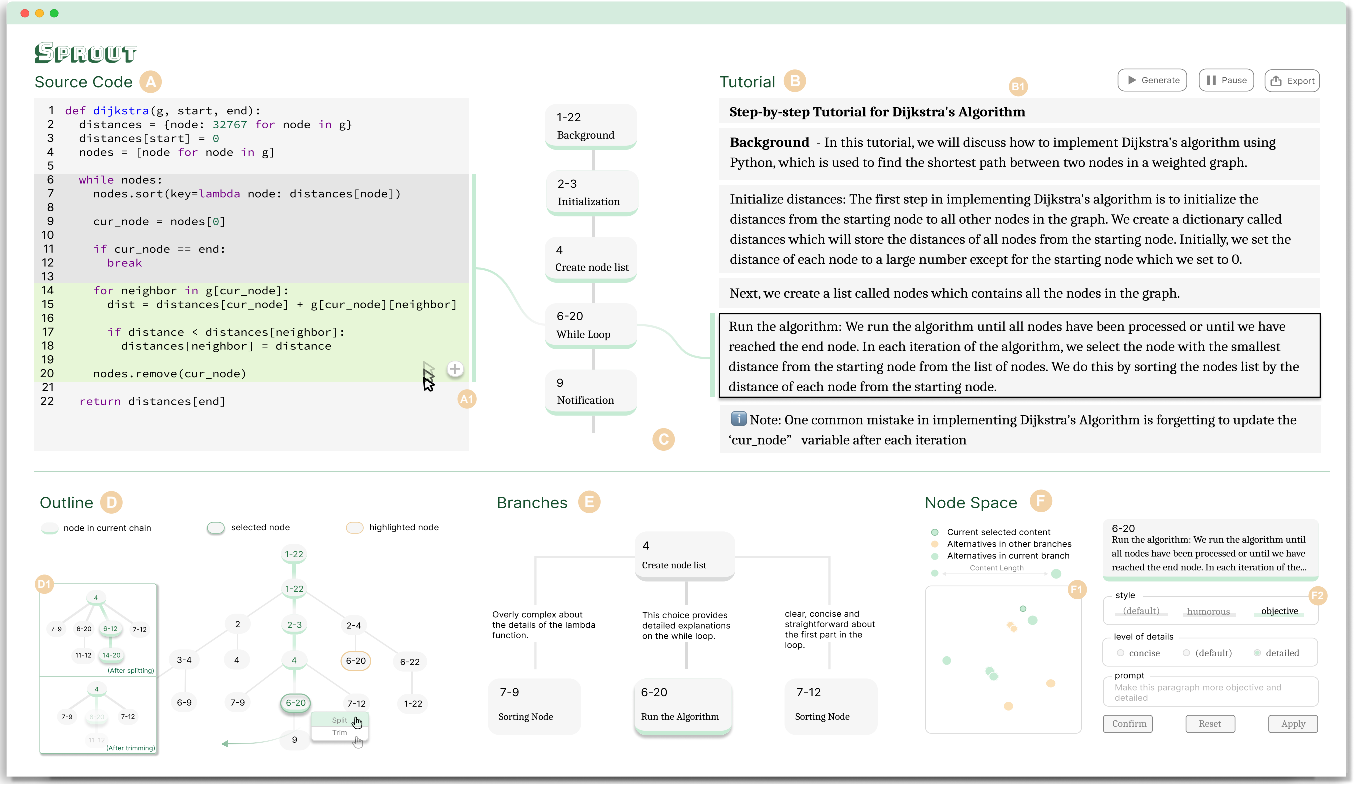
Agent generation. Sometimes users have limited knowledge about the code they want to describe or feel hesitated about the next steps to take. In this situation, they can make the LLM-based agent shoulder the responsibility to sketch the framework. In the tutorial view (fig. 3 B), users can click on the *Generate* option to start the generation. *SPROUT* then automatically generates the tutorial content step by step, following the tree-of-thought methodology. To ensure users' awareness and control over the tutorial generation process, *SPROUT* updates the tutorial content and renders the tree graph in real-time. Meanwhile, users can use the *Pause* option to halt the generation process and review the current tutorial content, ensuring it aligns with their expectations, especially if the auto-generated content significantly deviates from what users anticipated.

User-defined generation. In some cases, users have a clear intention regarding the code snippet they want to write about next. However, they may find it challenging to effectively communicate their ideas in written form to inform the LLMs. *SPROUT* saves users' arduous work to write and test various prompts to get a satisfactory response. Instead, they can simply brush their target code snippet and click the floating button to effortlessly add new content to the current tutorial (fig. 3 A1). Supported by prompting strategies we proposed (section 4), *SPROUT* generates paragraphs related to the code with consideration of the coherence to the surrounding contexts. And the node corresponding to the newly generated content is appended to the end of the current chain. This ensures that the generated content is seamlessly integrated and follows a logical progression within the existing tutorial.

5.3 Tutorial Modification through Tree Graph

SPROUT enables users to utilize the tree graph as the primary controller for flexible and intuitive modification of the tutorial content through manipulating the nodes within the tree (DR2, DR4). This feature aims to alleviate the need for users to directly prompt the LLMs and reduces their burden. For instance, users can consolidate multiple paragraphs to reduce their quantity, emphasize important code snippets by elaborating on them, and conveniently remove unnecessary parts from the tree graph for easier content management.

Condensing through node grouping. Sometimes LLMs provide overly exhaustive content for users, resulting in unclear key points in the tutorial and overwhelming readers. Users have to manually identify paragraphs that are less important or too familiar to the majority of readers and condense the content together in the typical workflow. *SPROUT* thus supports node grouping in the outline view to simplify this process. For example, when a user finds out that the content of two paragraphs can be integrated into



```
def dijkstra(g, start, end):
  distances = {node: 32767 for node in g}
  distances[start] = 0
  nodes = [node for node in g]

  while nodes:
    nodes.sort(key=lambda node: distances[node])

    cur_node = nodes[0]

    if cur_node == end:
      break

    for neighbor in g[cur_node]:
      dist = distances[cur_node] + g[cur_node][neighbor]

      if distance < distances[neighbor]:
        distances[neighbor] = distance

    nodes.remove(cur_node)

  return distances[end]
```

Step-by-step Tutorial for Dijkstra's Algorithm

Background - In this tutorial, we will discuss how to implement Dijkstra's algorithm using Python, which is used to find the shortest path between two nodes in a weighted graph.

Initialize distances: The first step in implementing Dijkstra's algorithm is to initialize the distances from the starting node to all other nodes in the graph. We create a dictionary called distances which will store the distances of all nodes from the starting node. Initially, we set the distance of each node to a large number except for the starting node which we set to 0.

Next, we create a list called nodes which contains all the nodes in the graph.

Run the algorithm: We run the algorithm until all nodes have been processed or until we have reached the end node. In each iteration of the algorithm, we select the node with the smallest distance from the starting node from the list of nodes. We do this by sorting the nodes list by the distance of each node from the starting node.

ℹ Note: One common mistake in implementing Dijkstra's Algorithm is forgetting to update the 'cur_node" variable after each iteration

4
Create node list

Overly complex about the details of the lambda function.

This choice provides detailed explanations on the while loop.

clear, concise and straightforward about the first part in the loop.

7-9
Sorting Node

6-20
Run the Algorithm

7-12
Sorting Node

6-20
Run the algorithm: We run the algorithm until all nodes have been processed or until we have reached the end node. In each iteration of the...

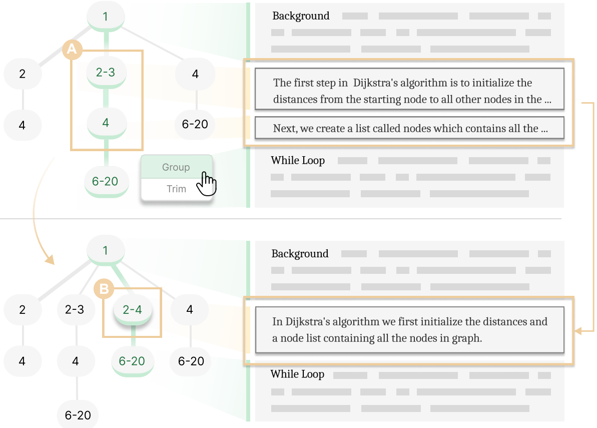
style: (default) humorous [✓ objective]

level of details: ○ concise ○ (default) ● detailed

prompt: Make this paragraph more objective and detailed

Fig. 3. The system interface of *SPROUT*. The Code View (A) displays the source code. The Tutorial View (B) presents the LLM-generated tutorial content. Between them is the Chain View (C), showing the node chain of current paragraphs selected by users. The Outline (D), Branches (E), and Node Space (F) Views provide interactive visualizations for multi-level modifications such as elaborating, adjusting structure, and polishing content.

one, they can select them in the tree graph and navigate to the *Group* option to make the LLM achieve a seamless content combination (fig. 4). The selected nodes will all be replaced by the newly-integrated node in a new branch, ensuring access to their old version.



The first step in Dijkstra's algorithm is to initialize the distances from the starting node to all other nodes in the ...

Next, we create a list called nodes which contains all the ...

In Dijkstra's algorithm we first initialize the distances and a node list containing all the nodes in graph.

Fig. 4. Select two nodes (A) and group them into one (B).

Elaborating through node splitting. In addition to condensing, elaboration is another essential task for maintaining a balanced quantity throughout the entire tutorial. When users explore the tree graph and discover that a key concept is mentioned but lacks sufficient explanation, they can select the corresponding node and navigate to the *Split* option. *SPROUT* prompts the LLM to generate additional content, which is then presented in new paragraphs. Simultaneously, *SPROUT* creates a copy of the branch where the selected nodes are located, and the selected nodes are split into newly-generated nodes (fig. 3 D1).

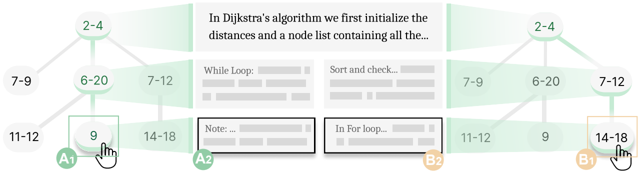
Deleting through node trimming. Since *SPROUT* showcases complete thinking paths of the agent when authoring the tutorial, the tree graph may accumulate redundant nodes as users iterate on the tutorial, making it difficult to navigate and manage effectively. Therefore, *SPROUT* offers a node trimming feature that allows users to delete specific nodes in the tree graph that contain the undesired content. When users pinpoint some redundant or unnecessary paragraphs, they can use the *Trim* option to delete the corresponding nodes, and their children nodes will be removed from the tree at the same time (fig. 3 D1).

5.4 Context Switch across Branches

To facilitate a more diverse (DR1) and flexible (DR4) organization of the tutorial's content, *SPROUT* provides two interactions to assemble paragraphs and switch between various contexts generated by LLMs flexibly.

5.4.1 Quick Assembling

SPROUT supports swift switching between different branches. When users are not satisfied with the current version of the tutorial and desire to explore additional possibilities, they can select an arbitrary node in the graph as the tail of the chain. As a result, the selected node and its ancestor nodes will be automatically assembled together and replace the original chain (fig. 5). By selecting an arbitrary node as the starting point for exploration, users can delve into new directions and uncover untapped potential in the tutorial's structure and content.


In Dijkstra's algorithm we first initialize the distances and a node list containing all the...

While Loop:

Sort and check...

Note: ...

In For loop...

Fig. 5. Users can achieve quick assembling nodes by selecting any node in the tree graph as the tail of the chain. For example, after user clicks on node (A1) in outline view, the content in tutorial view (A2) updates correspondingly. (B1) and (B2) is another example.

5.4.2 Step-by-Step Assembling

Instead of quickly assembling a chain of nodes, *SPROUT* also provides a deliberate method to manually decide on each step from any selected node in the graph as the entry point. The branch view (fig. 3 E) is designed to demonstrate the details of multiple reasoning paths from the selected parent node, which encompasses two aspects:

- **Choices:** When the agent generates several candidate thoughts for the next step, we recommend the best choice through a voting process. At each step, the agent casts its vote for the most promising thoughts. *SPROUT* then presents the top 3 choices, visually representing the voting results through the line width that connects the nodes. This visual encoding serves as a reference for evaluating the reliability of the diverse thinking paths.
- **Reasons:** *SPROUT* instructs the agent to provide explicit reasons for its choices. Leveraging our prompt strategies, the agent mostly bases its reasoning on the logical connection between code snippets or the coherence of textual paragraphs related to nodes. *SPROUT* presents the textual reason of each choice on the links, enabling users to make more informed and discerning decisions.

When a candidate node is chosen, the branch view transitions to a deeper level, allowing users to delve further into exploring the possibilities for the next paragraph. This process can be repeated iteratively until users finalize the new framework of the tutorial. Moreover, users retain the capability to navigate back to previous nodes, allowing them to backtrack and revise their decisions whenever necessary. This iterative and flexible approach empowers users to progressively explore the content of the tutorial.

5.5 Detail Refinement in Node Space

After confirming the structure of the tutorial, users often need to further refine the textual content to align it with their expectations (DR2). They may not satisfied with the text content pertaining to a particular intent, such as an explanation for specific code fragments, and try to refine it. To streamline this process, *SPROUT* implements two-part interactions, providing users with a systematic approach to refine the content and achieve the desired outcome (DR4).

5.5.1 Exploring Alternatives

SPROUT offers users the ability to chose alternative representations for a specific intent. LLM-generated paragraphs are visualized as scattered nodes in the node space view (fig. 3 F1). We utilize OpenAI's text embedding model [57] to measure the semantic similarity of generated paragraphs and projected the embeddings onto a 2D plane using the UMAP algorithm [58]. As a result, paragraphs with similar meanings are positioned closer to each other in this 2D space, which facilitates the quick retrieval of semantically relevant alternatives.

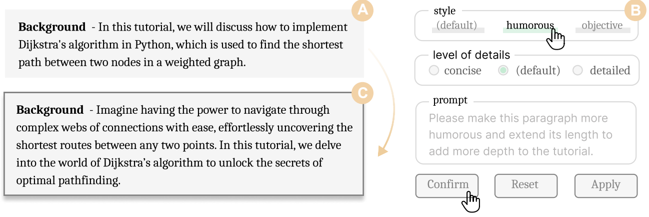
The node alternatives that share the same intent as the user-selected node are highlighted, with color distinguishing their origin, aligning with the color encoding in the outline view (fig. 3 D). Nodes are considered to have the same intent if they are generated through the same action type and are associated with the same code fragments. For example, two nodes may both describe the first line of code but with distinct representations. This view enables users to filter and select content that aligns with their specific intent, serving as a foundation for subsequent steps.

5.5.2 Rewriting Details

According to the feedback from the formative study, we conclude several aspects that users primarily focus on when they intend to refine the generated tutorial content. *SPROUT* then allows users to customize the content from the following aspects:

- **Writing styles.** Users may have their preferences for the style of the content. *SPROUT* provides several options to make LLMs rephrase the content in an designated style.
- **Level of details.** Users might be troubled by the unpredictable initial quantity of LLM-generated content, while an excellent tutorial should be reasonably detailed. Therefore, *SPROUT* allows users to adjust the level of details while preserving the originally conveyed information.
- **Prompt Refinement.** In order to meet a wider range of user needs, in addition to the two aspects mentioned above, *SPROUT* also supports authors to freely modify the prompt in order to improve the content.

As shown in fig. 6, when users find the background is too brief and not attracting enough, they can adjust the level of details and choose a polishing style in the configuration panel, then navigate to the *Confirm* option to prompt the LLM to rephrase the content accordingly.


Background - In this tutorial, we will discuss how to implement Dijkstra's algorithm in Python, which is used to find the shortest path between two nodes in a weighted graph.

Background - Imagine having the power to navigate through complex webs of connections with ease, effortlessly uncovering the shortest routes between any two points. In this tutorial, we delve into the world of Dijkstra's algorithm to unlock the secrets of optimal pathfinding.

style: (default) [✓ humorous] objective

level of details: ○ concise ● (default) ○ detailed

prompt: Please make this paragraph more humorous and extend its length to add more depth to the tutorial.

Fig. 6. In *SPROUT*, users can polish the content and adjust the levels of detail of any paragraph (A) by setting options in configuration panel (B), then retrieve polished paragraph (C) from LLM.

5.6 Visualization for Tutorial Understanding

To strike a balance between clearly separating two kinds of media resources (i.e., code and text) and offering visual elements of their connections (DR3), *SPROUT* designs visual representations for users to understand what the text block is about and where it comes from.

Brief Block. Although the chain view provides the structure of a tutorial explicitly, it is still arduous work for users to grasp the main idea of each paragraph. Therefore, *SPROUT* prompts LLMs to brief each paragraph in few words to enable users to quickly comprehend the key concept it intends to convey.

Elements Matching. To reduce the switch of users' attention between the code view and the tutorial content, *SPROUT* presents a set of visual cues for element matching between the source code and the tutorial document. We extract the connection between the code snippet and paragraph, then present the information in the node. Furthermore, *SPROUT* provides visual links of the connection between code snippet and text blocks when users focus on certain node or text block; thus, they can locate the text and code quickly. During the tutorial generation stage, the focused node will be automatically updated as the latest generated node, enabling users to follow the LLM's generation process more easily and absorb the newly-added information in a more moderate way.

6 Technical Evaluation

To validate the effectiveness of our prompting technique, we conducted a technical evaluation focusing on the accuracy of the extracted text-code connections.

6.1 Experiment Settings

Dataset. We collected 10 code snippets with diverse programming languages, lengths, and structures, covering different categories. These snippets were fed into our system to automatically generate programming tutorials. Then, we manually labeled the code segments that each paragraph references, establishing a ground truth for evaluating text-code connection extraction accuracy.

Procedure. The labeling work and accuracy calculation were conducted by 3 experienced programmers. They separately manually labeled the connection between each paragraph and code segments. They then compared the three labeled results and performed a secondary verification if there were differences among these results. This procedure was designed to guarantee the accuracy of the result, preventing potential errors that may occur during an individual's labeling process. After that, they examined the text-code connections extracted by *SPROUT* to calculate their accuracy. Subsequent analyses were conducted on failure cases and contributing factors to these errors.

6.2 Results

The dataset comprised 10 programming tutorials, encompassing 138 paragraphs and the corresponding text-code connections. For the details of the procedure, generated content, and statistical analysis, please refer to the supplemental materials.

Metrics. The extracted text-code connections exhibited an overall accuracy of **86.2%**, comprising 119 correct connections and 19 erroneous ones.

Error Analysis. Of the 19 erroneous connections identified, we categorized them into three types: (1) *No Code* (5/19), (2) *Incorrect Code Range* (12/19), and (3) *Incorrect Code Content* (2/19). For the first error type, the LLM failed to provide any code for the associated paragraph. For the second error type, the LLM provided an inappropriate code range mismatched with the corresponding paragraph. Notably, the majority (10/12) presented the entire code range excessively, whereas only two instances demonstrated incomplete or unrelated connections. For the third error type, the LLM provided fictitious code content misaligned with the original code snippet. We discern these errors as manifestations of LLMs' *hallucination*, particularly evident when employing complex prompting strategies like Tree-of-Thoughts prompting strategy, coupled with a high temperature which might lead to more unexpected results.

Summary. Despite the few failure cases, the results indicate the reliability and accuracy of our prompting technique in extracting text-code connections. This efficiently facilitates users in obtaining faithful content, laying a robust foundation for further refinements of the content.

7 User Study

To evaluate the effectiveness of our system in facilitating the programming tutorial authoring process, we conducted a user evaluation to collect feedback on (1) the features of *SPROUT*, and (2) its support for tutorial authoring compared to a baseline interface. Subsequently, we analyzed user interaction logs to gain an understanding of the emergent workflow patterns and user behaviors using *SPROUT*.

7.1 Methodology

7.1.1 Participants

We recruited 12 participants (P1-P12) for our experiment, consisting of 5 females and 7 males, aged 22-28, from a local university. P1-6 were experienced tutorial authors who had also participated in the formative study (E1-6). The remaining participants were students with experience in code documentation, but none had experience in authoring programming tutorials. All participants had more than 4 years of programming experience and had read programming tutorials in the past. Ten of them were familiar with ChatGPT and regularly utilized it for writing tasks (*e.g.*, ideation, drafting, polishing), while two had heard of it but seldom incorporated it into daily work. Participants attended our experiment through online meetings, used our system deployed online, and the process was recorded.

7.1.2 Experiment Condition

The experiment was conducted under two conditions:

- ChatGPT (*baseline*). The baseline interface offered an environment comprising a document editor, a code viewer, and ChatGPT. Users could freely arrange these elements within the interface.
- *SPROUT* (*ours*). The system integrated a code viewer and document editor, enriched with a series of interactive visualizations of the creation process with LLMs.

Each participant was tasked with creating tutorials using both of the aforementioned tools. The order of tool usage was counterbalanced among participants: half began with ChatGPT, while the other half started with *SPROUT*.

7.1.3 Procedure

The experiment consists of the following phases:

Introduction and Training. First, the purpose of the experiment was presented. Participants then signed a consent form and filled out a pre-study questionnaire regarding their background and experience with tutorial authoring. After that, they were introduced to *SPROUT*'s features, followed by a demonstration of crafting a complete algorithm tutorial using these features. We gave the participants adequate time to familiarize themselves with the system and encouraged them to reproduce the tutorial independently. All participants were free to ask any clarifying questions.

Targeted Task. For this task, participants were provided with a piece of code for reference. Their objective was to author a tutorial based on a given framework, which involved two sub-tasks: modifying the content of at least one paragraph and describing the main idea of each paragraph. Participants underwent two trials in this phase, creating programming tutorials in each trial using two different tools – *baseline* and *ours* – respectively.

Open-ended Task. For this task, only the problem's background and the source code were provided. Participants were encouraged to create a step-by-step programming tutorial freely using *SPROUT*. They were expected to produce at least three tutorial versions and select one as the final result. We ensured participants had enough time to validate the generated content. There were no time constraints on the writing process during trials, allowing them to create tutorials to the best of their abilities.

Post-study Interview. Upon completion of all trials, we conducted a semi-structured interview with each participant. They were first invited to fill out a questionnaire with 5-point Likert-scale questions, regarding five features in *SPROUT*, the general usability, and the support for programming tutorial authoring with our system and the baseline. Then, we inquired their thoughts on how *SPROUT* could help their actual workflow, the advantages and limitations of the current system design, as well as their suggestions for potential improvements.

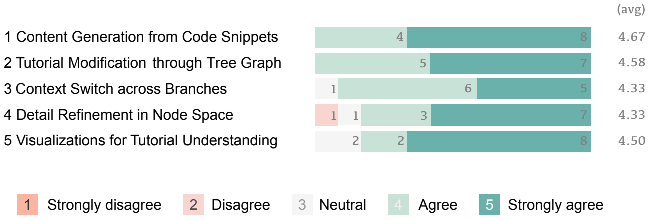


Fig. 7. User feedback regarding the features in *SPROUT*, measured on a 5-point Likert-scale.

7.2 Results Analysis

7.2.1 Features

We evaluated five features of *SPROUT* through 5-point Likert-scale questionnaires in the post-study interview. Figure 7 illustrates the detailed ratings.

Participants chose different generation methods based on their needs. They responded favorably to the feature of generating content based on selected code, finding it useful when they *"already have an initial framework in mind"* (P6) for authoring content related to familiar code. The free generation was also praised since it could *"provide diverse perspectives as a starting point when at lost"* (P3) and allowed them to *"pick up a best one among different versions"* (P1).

Participants utilized visual connections to quickly grasp the tutorial content. The visual representations of connections were favored by all participants, as it alleviated the effort to *"maintain the connections in mind"* (P5). In traditional chat-based interfaces, even if referred code is provided, participants sometimes still needed to *"scroll back to check the position of the code snippet within the complete code"* (P11) and *"the indentation of the code is missing"* (P2).

Participants switched between thoughts smoothly and sensibly. According to some participants, utilizing the branch view with thoughts is like *"building text blocks under the guidance of LLMs"* (P3). Moreover, the quick assembling in Outline by selecting leaf nodes enables *"swift comparison"* (P9) between documents based on different thoughts.

Participants employed the tree graph to modify the tutorial. Most participants praised the graph-level modifications such as splitting node, describing them as *"novel and convenient interactions within a document"* (P7) that would be *"frequently used"* (P10). For example, P6 split the node which described the main loop in Dijkstra algorithm, because he found the content was too brief. Similarly, P8 utilized "Trim" to delete the content describing the engineering process, which *"can be omitted in a tutorial"*.

Participants achieved easy detail refinement in node space. When focusing on specific content from LLMs, the node space serves as *"a good container which collects all the content for me"* (P4). *SPROUT* also enables purpose-specific content refinement. For instance, P9 easily got a concise paragraph in *SPROUT*, whereas with ChatGPT, he had to test prompts iteratively. However, P7 expressed dissatisfaction about this feature as his demands were not covered currently, such as introducing metaphors in the tutorial.

7.2.2 Tutorial Authoring Support Analysis

At the end of the study, we asked participants to rate the two authoring tools across seven dimensions (shown in fig. 8). Their feedback demonstrated the system usability and its effectiveness in supporting tutorial creation process.

***SPROUT* enables flexible tutorial generation for enhanced engagement.** Based on participants' feedback, the generation methods in *SPROUT* allow them to conveniently tailor LLM-generated content to match their expectations ($\mu = 4.5 > 3.83$, $p = .021$), offering different degrees of autonomy, while it is challenging with ChatGPT to maintain such control. For example, P4 started the authoring process with the free generation as the *"initial execution for inspirations"*, paused the process, and then searched for *"a reasonably detailed content from all branches"* in the branch view to serve as a foundation for the tutorial. After spotting a favorable structure, he proceeded to manually select the code pieces for content generation, thereby ensuring the resulting output met his expectations.

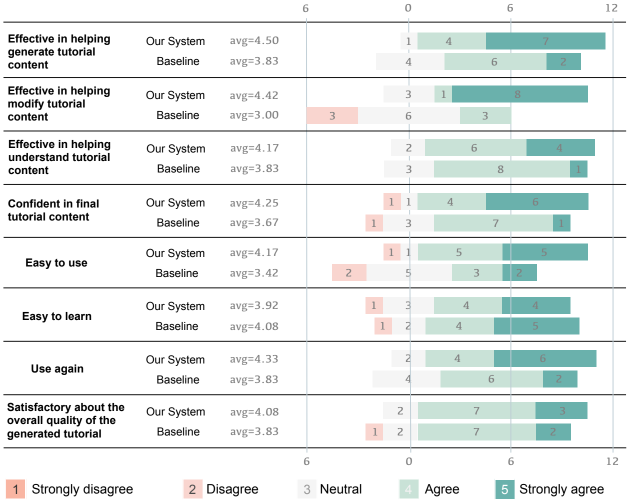


Fig. 8. The results of the questionnaire regarding the authoring support of our system and the baseline.

***SPROUT* provides easier modifications on tutorials hierarchically.** Participants reported that the interactions provided in *SPROUT* enabled easy tutorial modifications across various aspects ($\mu = 4.42 > 3$, $p = .007$), eliminating the need for *"repeatedly writing prompts"* (P11). Some participants concentrated on the narrative order of the tutorial, while others tended to manipulate the style or detail levels of certain paragraphs. Often, participants employed a combination of these methods. For example, P3 first decided the tutorial structure according to LLM recommendations, then grouped some nodes to shorten the tutorial, and finally refined the text of an unsatisfactory paragraph. In contrast, when working with ChatGPT, participants had to interact with it in multiple rounds, as the prompts were *"ambiguous to convey their demands in mind"* (P7).

***SPROUT* enhances tutorial comprehension and reliability during the authoring process.** Participants reflected that they could produce tutorials with more comprehension ($\mu = 4.17 > 3.83$, $p = .157$) and confidence ($\mu = 4.25 > 3.67$, $p = .07$) on the outcomes throughout the workflow of *SPROUT*. During the generation process, the connections are updated synchronously, which *"offers clear visual cues"* (P1) about the content of each paragraph. P12 also pointed out that when utilizing ChatGPT to modify content, the revised content often appeared separately from the original tutorial and code, leaving him uncertain about its correctness. Moreover, we found that most participants would browse the entire tutorial to guarantee the key points were covered. In *SPROUT*, the visual connections can support users to *"easily verify the coverage of the provided code"* (P7), thus speeding up the examination process.

***SPROUT* elevates the level of engagement and ease for authors.** Participants appreciated that *SPROUT* provided a more user-friendly experience in the process of authoring tutorials comparing to ChatGPT ($\mu = 4.17 > 3.42$, $p = .038$), without significantly increasing the learning effort ($\mu = 3.92 < 4.08$, $p = .564$). P2 said the novel visual interface made him *"more engaged"* with the authoring work. Participants also conveyed a positive inclination towards integrating SPROUT into their real work practices ($\mu = 4.33 > 3.83$, $p = .083$).

***SPROUT* produces user-satisfying tutorials of superior quality.** Participants expressed their satisfaction with the quality of tutorials generated by *SPROUT* ($\mu = 4.08 > 3.83$, $p = 0.317$), appreciating aspects such as structure, clarity, and educational value. P5 suggested that our system has the potential to serve as an *"interesting learning tool"*, in addition to its capabilities as an authoring tool. Beyond the quality of the textual content, the visual design of the system also contributes substantial educational value. As a teacher, P6 valued the ease with which he could customize tutorials to cater to learners of varying levels with minimal effort.

7.2.3 Authoring Workflow Analysis

We analyzed the interaction logs from participants to investigate the impact of *SPROUT* on the authoring workflow. Our observations revealed diverse workflow patterns adopted by participants in completing their tutorials, along with notable behaviors when interacting with *SPROUT*.

Workflow Patterns. Two predominant workflow patterns were observed among participants. The most prevalent one was to utilize the automatic generation feature first to produce a complete tutorial, with a primary focus on the tutorial view and chain view. Following the completion of the generation process, participants proceeded to modify the tutorial using the Outline, Branches, and Node Space to suit their specific requirements. The second pattern emerged when participants desired to take the initiative. In this situation they used user-defined generation, modified the content then moved to the next step, where they switched attention between different views. Our system offered robust support for both of the aforementioned workflows.

Observed Behaviors. Participants sometimes tended to utilize the features of *SPROUT* with novel target according to their original authoring habits, which were not considered before. For instance, the demonstration of the reasons in branch view also assisted participants to verify the reasoning of tutorials upon completion and during the final verification process. In terms of tree graph, some experienced participants utilized it to compare different branches for optimal structuring apart from understanding the thinking steps of LLM. The node space view also served as an organizational tool for participants to efficiently access their desired content among multiple versions of paragraphs resulting from iterative content refinement.

8 Discussion and Future Work

In this section, we reflect on our research, mainly discussing the implications, limitations, and future work.

8.1 Implications

Incorporating LLMs into programming tutorial authoring workflow. The development of large language models opens vast possibilities to facilitate the programming tutorial authoring process through reducing manual effort on writing and editing. While our work provides a step-stone towards programming tutorial authoring with the assistance of LLMs, future work could investigate on further enriching

This article has been accepted for publication in IEEE Transactions on Visualization and Computer Graphics. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2024.3410523

tutorials with more multifaceted content, such as execution examples, flow charts, or figures illustrating code execution processes, which calls for utilizing multi-modal approaches and specific interactions to combine versatile media into current workflow. Recent state-of-the-art research leveraging in-context learning technologies has been successful in producing code-related content of remarkable quality [13], [59]. Our work, however, concentrates on enhancing the interaction between humans and AI. Integrating these two approaches with advanced techniques, such as active learning, has the potential to yield more user-centric and efficient tools [60]. We believe that our work can be a solid foundation for future researches on authoring programming tutorials. By incorporating a wider array of materials, we can ultimately create content with higher quality, granting readers a richer understanding of the programming concepts presented.

Step-by-step exploratory process offers a more controlled authoring experience. Numerous studies have delved into advancing strategies that prompt LLMs to think step by step [12], [15], [37] for higher quality content. Beyond this, we see potential in using this decomposed process as an entry point to grant users broader access to the intermediate steps of the authoring workflow. This empowers them to achieve anticipated outcomes, transforming a typically passive experience into an active one. The evaluation results indicated that the interactive, segmented representations of the creation process and generated tutorial content in *SPROUT* offer more precise generation and allow for multi-level adjustments to modules. Such an approach addresses the often-encountered challenge of accurately conveying intentions in chat-based interfaces.

Maintaining connections between provided inputs and LLM-generated outputs ensures transparency and traceability. We identify and visualize latent connections between the source code and the generated tutorial, as we believe it can assist users to comprehend and verify the reasoning behind LLMs' creation process. Informed by our evaluation results, users utilized these presented connections to examine the outcomes after generating or modifying procedures, easing the burden of spending extra cognitive efforts to recognize and memorize them. Moreover, consistently maintaining these connections significantly improves user confidence in the final results crafted collaboratively by users and LLMs. This is essential in materials-backed authoring scenarios, where consistency and correctness are vital. Besides, our LLM-based methodology and visual design strategies for establishing connection between cross-modal data promise to be beneficial across a wide range of multi-modal creation and data analysis scenarios [61]–[63].

Striking a balance on the degree of interaction integrity in interface design. A key finding is that user preferences for interacting with LLMs vary by individuals. For example, depending on the specific operations they intend to perform, some individuals preferred utilizing integrated interactions as a means to effortlessly obtain accurate outcomes, while others expressed reservations regarding the comprehensiveness of these approaches in covering all possible scenarios. There exists a conflict between the extensiveness of features coverage and the precision of the results after execution when designing interactions. Thus, our research could inspire future studies to more systematically explore the interface design paradigm for LLM-based applications, which could range from completely-free form to structured module-based interactions.

Applying methodologies to diverse LLM-based creation scenarios. User feedback from our evaluation underscores the importance of both fidelity and innovative capabilities of LLMs in authoring tasks. Designing and developing systems or tools for creative scenarios requires a harmonious mix of adaptability and innovation to ensure their use is seamless and user-friendly [64]. Although our study focuses on utilizing novel prompt strategies and interactive visualizations for programming tutorial authoring, the methodologies employed herein can potentially be extended to other creation scenarios that require consistency with reference materials, such as fact-based analysis for legal documents [65], medical applications [66], [67]. The underlying idea involves designing purpose-specific and scenario-oriented prompting methods to enable appropriate exploratory workflows. The visualization of the generation process of LLMs can also be generalized to the majority of authoring tasks [6], [14], [16] to make the process more tracable. Furthermore, the value of a step-by-step rationale extends beyond authoring application, with potential benefits to other domains such as commonsense Q&A [68] or text learning [69]. By breaking down complex or lengthy content into manageable segments, the learning process becomes more palatable through this incremental methodology. This suggests the possibility of integrating the step-by-step methodology into applications aimed at enhancing the learning workflow (e.g., mathematical concepts and language learning), which can offer a more structured and digestible approach to knowledge acquisition [70].

8.2 Limitations and Future Work

Although our work has shown promising results, there are several limitations and opportunities for future research.

Scalability. Although our system effectively supports tutorial authoring process in most cases, it may face limitations in adequately covering the context when dealing with lengthy source code and tutorials, primarily due to token limitations. Nevertheless, these limitations can be addressed with advancements in LLM capabilities or by utilizing memory summarization techniques.

Potentiality. While our study focuses on modifying content within a single chain of thought, the current design framework can support more varied tutorial revision operations. For example, future work could explore facilitating users in merging thoughts from two distinct paths, ensuring context preservation. This enhancement, rooted in our current prompting strategies, would also necessitate broadening the scope of graph manipulations to enable more flexible interactions for users to customize the content.

Generalizability. Further exploration into multi-modal approaches can benefit tasks involving non-textual elements, like data visualization. Based on the nature of source materials and expected outcomes, visual representations beyond tree graphs should be further explored to enhance the comprehension and manipulation during the creation process. For example, future researches can investigate how to properly reify the creation process with LLMs when handling non-linear structured data.

Limitations. While the current prompting methods are sufficiently reliable and accurate for the authoring workflow, the extraction accuracy can be further enhanced. Future work could focus on utilizing correction prompts to examine the extracted connections or integrating add-on modules to identify connections to guarantee a higher accuracy. Additionally, while insightful, our study is limited by its laboratory setting as it does not encompass assessments in real educational scenarios. Introducing student agents for tutorial assessment and iterative modification could potentially enhance its education value. Conducting a evaluation set in real education scenarios is also suggested for a more holistic understanding of the realistic learning impact.

9 Conclusion

This work presents *SPROUT*, a programming tutorial authoring tool with LLMs which breaks down the programming tutorial authoring task into actionable steps and adopts novel prompting strategies to generate high-quality and diverse tutorial content. With a series of interactive visualizations of the LLM generation process, users are able to effectively engage in an exploratory process to generate, modify, and understand the generated tutorial. A user study with 12 participants demonstrates that users can effectively utilize *SPROUT* to author programming tutorials with LLMs in a transparent and traceable process, leading to more controllable and reliable results.

Acknowledgments

We would like to thank Jiehui Zhou and Minfeng Zhu for their heartwarming support. We thank anonymous reviewers for their insightful reviews. This paper is supported by the National Natural Science Foundation of China (62132017, 62302435), Zhejiang Provincial Natural Science Foundation of China (LD24F020011) and "Pioneer" and "Leading Goose" R&D Program of Zhejiang (2024C01167).

References

[1] A. S. Kim and A. J. Ko, "A pedagogical analysis of online coding tutorials," in *Proc. ACM SIGCSE*, 2017, pp. 321–326.

[2] H. Jiang, J. Zhang, Z. Ren, and T. Zhang, "An unsupervised approach for discovering relevant tutorial fragments for apis," in *Proc. IEEE/ACM ICSE*, 2017, pp. 38–48.

[3] S. Hamouda, S. H. Edwards, H. G. Elmongui, J. V. Ernst, and C. A. Shaffer, "Recurtutor: An interactive tutorial for learning recursion," *ACM Trans. Comput. Educ.*, vol. 19, no. 1, pp. 1:1–1:25, 2019.

[4] A. Y. Wang, D. Wang, J. Drozdal, M. J. Muller, S. Park, J. D. Weisz, X. Liu, L. Wu, and C. Dugan, "Documentation matters: Human-centered AI system to assist data science code documentation in computational notebooks," *ACM Trans. Comput. Hum. Interact.*, vol. 29, no. 2, pp. 17:1–17:33, 2022.

[5] K. I. Gero, V. Liu, and L. Chilton, "Sparks: Inspiration for science writing using language models," in *Proc. ACM DIS*, 2022, pp. 1002–1019.

[6] N. Sultanum and A. Srinivasan, "Datatales: Investigating the use of large language models for authoring data-driven articles," in *Proc. IEEE VIS*, 2023, pp. 231–235.

[7] S. Biswas, "Chatgpt and the future of medical writing," p. e223312, 2023.

[8] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. Bang, A. Madotto, and P. Fung, "Survey of hallucination in natural language generation," *ACM Comput. Surv.*, vol. 55, no. 12, pp. 248:1–248:38, 2023.

[9] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021.

[10] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *MAPS@PLDI*, 2022, pp. 1–10.

[11] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Proc. NeurIPS*, 2020.

[12] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Proc. NeurIPS*, 2022.

[13] T. Ahmed, K. S. Pai, P. Devanbu, and E. T. Barr, "Improving few-shot prompts with relevant static analysis products," *arXiv preprint arXiv:2304.06815*, 2023.

[14] A. Yuan, A. Coenen, E. Reif, and D. Ippolito, "Wordcraft: story writing with large language models," in *Proc. ACM IUI*, New York, NY, USA, 2022, pp. 841–852.

[15] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," *CoRR*, vol. abs/2305.10601, 2023.

[16] Z. Zhang, J. Gao, R. S. Dhaliwal, and T. J.-J. Li, "VISAR: A human-ai argumentative writing assistant with visual programming and rapid draft prototyping," in *Proc. ACM UIST*, 2023, pp. 5:1–5:30.

[17] A. Head, J. Jiang, J. Smith, M. A. Hearst, and B. Hartmann, "Composing flexibly-organized step-by-step tutorials from linked source code, snippets, and outputs," in *Proc. ACM CHI*, 2020, pp. 1–12.

[18] R. Tiarks and W. Maalej, "How does a typical tutorial for mobile development look like?" in *Proc. ACM MSR*, 2014, pp. 272–281.

[19] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns, "What makes a good code example?: A study of programming q&a in stackoverflow," in *Proc. IEEE ICSM*, 2012, pp. 25–34.

[20] C. Kojouharov, A. Solodovnik, and G. Naumovich, "Jtutor: an eclipse plug-in suite for creation and replay of code-based tutorials," in *Proc. ETX*, 2004, pp. 27–31.

[21] C. Oezbek and L. Prechelt, "Jtourbus: Simplifying program understanding by documentation that provides tours through the source code," in *Proc. IEEE ICSM*, 2007, pp. 64–73.

[22] A. Mysore and P. J. Guo, "Torta: Generating mixed-media GUI and command-line app tutorials using operating-system-wide activity tracing," in *Proc. ACM UIST*, 2017, pp. 703–714.

[23] E. L. Ouh, B. K. S. Gan, and D. Lo, "ITSS: interactive web-based authoring and playback integrated environment for programming tutorials," in *Proc. ICSE (SEET)*, 2022, pp. 158–164.

[24] L. Bao, Z. Xing, X. Xia, and D. Lo, "Vt-revolution: Interactive programming video tutorial authoring and watching system," *IEEE Trans. Software Eng.*, vol. 45, no. 8, pp. 823–838, 2019.

[25] S. Oney, C. Brooks, and P. Resnick, "Creating guided code explanations with chat.codes," *Proc. ACM Hum. Comput. Interact.*, vol. 2, no. CSCW, pp. 131:1–131:20, 2018.

[26] S. Ginosar, L. F. D. Pombo, M. Agrawala, and B. Hartmann, "Authoring multi-stage code examples with editable code histories," in *Proc. ACM UIST*, 2013, pp. 485–494.

[27] A. Head, E. L. Glassman, B. Hartmann, and M. A. Hearst, "Interactive extraction of examples from existing code," in *Proc. ACM CHI*, 2018, p. 85.

[28] J. Leinonen, P. Denny, S. MacNeil, S. Sarsa, S. Bernstein, J. Kim, A. Tran, and A. Hellas, "Comparing code explanations created by

students and large language models," in *Proc. ACM ITiCSE*, New York, NY, USA, 2023, pp. 124–130.

[29] J. Y. Khan and G. Uddin, "Automatic code documentation generation using gpt-3," in *Proc. ACM ASE*, 2022, pp. 1–6.

[30] S. MacNeil, A. Tran, A. Hellas, J. Kim, S. Sarsa, P. Denny, S. Bernstein, and J. Leinonen, "Experiences from using code explanations generated by large language models in a web software development e-book," in *Proc. ACM SIGCSE*, 2023, pp. 931–937.

[31] S. MacNeil, A. Tran, J. Leinonen, P. Denny, J. Kim, A. Hellas, S. Bernstein, and S. Sarsa, "Automatically generating CS learning materials with large language models," *CoRR*, 2022.

[32] S. MacNeil, A. Tran, D. Mogil, S. Bernstein, E. Ross, and Z. Huang, "Generating diverse code explanations using the gpt-3 large language model," in *Proc. ACM ICER*, New York, NY, USA, 2022, pp. 37–39.

[33] S. Sarsa, P. Denny, A. Hellas, and J. Leinonen, "Automatic generation of programming exercises and code explanations using large language models," in *Proc. ACM ICER*, New York, NY, USA, 2022, pp. 27–43.

[34] Github, "Github copilot," https://docs.github.com/en/copilot, 2023.

[35] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, C. Zhang, J. Wang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, and J. Schmidhuber, "Metagpt: Meta programming for a multi-agent collaborative framework," *arXiv preprint arXiv:2308.00352*, 2023.

[36] C. Qian, X. Cong, W. Liu, C. Yang, W. Chen, Y. Su, Y. Dang, J. Li, J. Xu, D. Li, Z. Liu, and M. Sun, "Communicative agents for software development," *arXiv preprint arXiv:2307.07924*, 2023.

[37] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," in *Proc. ICLR*. OpenReview.net, 2023.

[38] S. Dhuliawala, M. Komeili, J. Xu, R. Raileanu, X. Li, A. Celikyilmaz, and J. Weston, "Chain-of-verification reduces hallucination in large language models," 2023.

[39] R. Sennrich, J. Vamvas, and A. Mohammadshahi, "Mitigating hallucinations and off-target machine translation with source-contrastive and language-contrastive decoding," 2023.

[40] R. Yen, J. Zhu, S. Suh, H. Xia, and J. Zhao, "Coladder: Supporting programmers with hierarchical code generation in multi-level abstraction," *arXiv preprint arXiv:2310.08699*, 2023.

[41] Y. Feng, X. Wang, K. K. Wong, S. Wang, Y. Lu, M. Zhu, B. Wang, and W. Chen, "Promptmagician: Interactive prompt engineering for text-to-image creation," *IEEE Trans. Vis. Comput. Graph.*, 2023.

[42] B. Wang, Y. Li, Z. Lv, H. Xia, Y. Xu, and R. Sodhi, "Lave: Llm-powered agent assistance and language augmentation for video editing," in *Proceedings of the 29th International Conference on Intelligent User Interfaces*, 2024, pp. 699–714.

[43] P. Jiang, J. Rayan, S. P. Dow, and H. Xia, "Graphologue: Exploring large language model responses with interactive diagrams," in *Proc. ACM UIST*, 2023, pp. 3:1–3:20.

[44] S. Suh, M. Chen, B. Min, T. J.-J. Li, and H. Xia, "Structured generation and exploration of design space with large language models for human-ai co-creation," *arXiv preprint arXiv:2310.12953*, 2023.

[45] T. Wu, E. Jiang, A. Donsbach, J. Gray, A. Molina, M. Terry, and C. J. Cai, "Promptchainer: Chaining large language model prompts through visual programming," in *Proc. ACM CHI*, 2022, pp. 1–10.

[46] T. Wu, M. Terry, and C. J. Cai, "Ai chains: Transparent and controllable human-ai interaction by chaining large language model prompts," in *Proc. ACM CHI*, 2022, pp. 1–22.

[47] I. Arawjo, C. Swoopes, P. Vaithilingam, M. Wattenberg, and E. L. Glassman, "Chainforge: A visual toolkit for prompt engineering and LLM hypothesis testing," in *Proc. ACM CHI*, 2024, pp. 304:1–304:18.

[48] E. Jiang, K. Olson, E. Toh, A. Molina, A. Donsbach, M. Terry, and C. J. Cai, "Promptmaker: Prompt-based prototyping with large language models," in *Proc. ACM CHI*, 2022, pp. 35:1–35:8.

[49] A. Mishra, U. Soni, A. Arunkumar, J. Huang, B. C. Kwon, and C. Bryan, "Promptaid: Prompt exploration, perturbation, testing and iteration using visual analytics for large language models," *arXiv preprint arXiv:2304.01964*, 2023.

[50] D. Masson, S. Malacria, G. Casiez, and D. Vogel, "Directgpt: A direct manipulation interface to interact with large language models," in *Proc. ACM CHI*, 2024, pp. 1–16.

[51] S. Suh, B. Min, S. Palani, and H. Xia, "Sensecape: Enabling multilevel exploration and sensemaking with large language models," in *Proc. ACM UIST*, 2023, pp. 1:1–1:18.

[52] S. M. Goodman, E. Buehler, P. Clary, A. Coenen, A. Donsbach, T. N. Horne, M. Lahav, R. MacDonald, R. B. Michaels, A. Narayanan *et al.*, "Lampost: Design and evaluation of an ai-assisted email writing prototype for adults with dyslexia," in *Proc. ACM SIGACCESS*, 2022, pp. 1–18.

[53] S. Petridis, N. Diakopoulos, K. Crowston, M. Hansen, K. Henderson, S. Jastrzebski, J. V. Nickerson, and L. B. Chilton, "Anglekindling: Supporting journalistic angle ideation with large language models," in *Proc. ACM CHI*, 2023, pp. 1–16.

[54] J. J. Y. Chung, W. Kim, K. M. Yoo, H. Lee, E. Adar, and M. Chang, "Talebrush: Sketching stories with generative pretrained language models," in *Proc. ACM CHI*, 2022, pp. 1–19.

[55] T. S. Kim, Y. Lee, M. Chang, and J. Kim, "Cells, generators, and lenses: Design framework for object-oriented interaction with large language models," in *Proc. ACM UIST*, New York, NY, USA, 2023.

[56] OpenAI, "Introducing ChatGPT," https://openai.com/blog/chatgpt, [Online; accessed 2023-04-30].

[57] OpenAI, "Introducing text and code embeddings," https://openai.com/blog/introducing-text-and-code-embeddings, [Online; accessed 2023-10-01].

[58] L. McInnes, J. Healy, and J. Melville, "Umap: Uniform manifold approximation and projection for dimension reduction," *arXiv preprint arXiv:1802.03426*, 2018.

[59] M. Geng, S. Wang, D. Dong, H. Wang, G. Li, Z. Jin, X. Mao, and X. Liao, "Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning," in *Proc. ICSE*, 2024, pp. 39:1–39:13.

[60] Q. Li, Z. Yu, H. Xu, and B. Guo, "Human-machine interactive streaming anomaly detection by online self-adaptive forest," *Frontiers Comput. Sci.*, vol. 17, no. 2, p. 172317, 2023.

[61] Y. Feng, J. Chen, K. Huang, J. K. Wong, H. Ye, W. Zhang, R. Zhu, X. Luo, and W. Chen, "iPoet: interactive painting poetry creation with visual multimodal analysis," *J. Vis.*, vol. 25, no. 3, pp. 671–685, Jun 2022.

[62] Y. Feng, X. Wang, B. Pan, K. K. Wong, Y. Ren, S. Liu, Z. Yan, Y. Ma, H. Qu, and W. Chen, "XNLI: Explaining and diagnosing nli-based visual data analysis," *IEEE Trans. Vis. Comput. Graph.*, pp. 1–14, 2023.

[63] X. Wang, Z. Wu, W. Huang, Y. Wei, Z. Huang, M. Xu, and W. Chen, "VIS+AI: integrating visualization with artificial intelligence for efficient data analysis," *Frontiers Comput. Sci.*, vol. 17, no. 6, p. 176709, 2023.

[64] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," 2023.

[65] J. Cui, Z. Li, Y. Yan, B. Chen, and L. Yuan, "Chatlaw: Open-source legal large language model with integrated external knowledge bases," *arXiv preprint arXiv:2306.16092*, 2023.

[66] A. J. Thirunavukarasu, D. S. J. Ting, K. Elangovan, L. Gutierrez, T. F. Tan, and D. S. W. Ting, "Large language models in medicine," *Nature medicine*, vol. 29, no. 8, pp. 1930–1940, 2023.

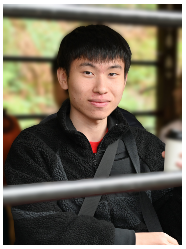
[67] Y. Jin, F. Zhu, J. Li, and L. Ma, "Tcmfvis: A visual analytics system toward bridging together traditional chinese medicine and modern medicine," *Vis. Informatics*, vol. 7, no. 1, pp. 41–55, 2023.

[68] F. Ocker, J. Deigmöller, and J. Eggert, "Exploring large language models as a source of common-sense knowledge for robots," *arXiv preprint arXiv:2311.08412*, 2023.

[69] E. Kasneci, K. Seßler, S. Küchemann, M. Bannert, D. Dementieva, F. Fischer, U. Gasser, G. Groh, S. Günnemann, E. Hüllermeier *et al.*, "Chatgpt for good? on opportunities and challenges of large language models for education," *Learning and individual differences*, vol. 103, p. 102274, 2023.

[70] M. Yu, Y. Wang, X. Yu, G. Shan, and Z. Jin, "Pubexplorer: An interactive analytical system for visualizing publication data," *Vis. Informatics*, vol. 7, no. 3, pp. 65–74, 2023.

Yihan Liu received the B.E. degree in computer science and technology from the Zhejiang University, China, in 2022. She is currently working towards the master's degree at the State Key Lab of CAD&CG, Zhejiang University. Her research interests include visual analytics, human–computer interaction, and software engineering.



Zhen Wen is a Ph.D. candidate at the State Key Lab of CAD & CG, College of Computer Science and Technology, Zhejiang University. He received the B.E. degree in Software Engineering from Zhejiang University of Technology in 2021. His research interests include visual analytics, software engineering, and quantum computing.

Luoxuan Weng is currently a Ph.D. candidate in the College of Computer Science and Technology, Zhejiang University, where he also received his B.E. degree in Software Engineering in 2021. His research interests include visual analytics, human-centered computing, and natural language processing.

Ollie Woodman is a Masters Student at the State Key Lab of CAD & CG, College of Computer Science and Technology, Zhejiang University. He obtained a Bachelors of IT in Software Development and a Bachelors of Arts in Chinese Studies at Monash University in 2022. His research interests include large language models, software engineering and machine learning.

Yi Yang is currently a Ph.D. candidate in the College of Computer Science and Technology at the Zhejiang University. She received the B.E. degree in Computer Science and Technology from the China University of Mining and Technology in 2023. Her research interests include computer vision and deep learning.

Wei Chen is a professor in the State Key Lab of CAD&CG at Zhejiang University. His current research interests include visualization and visual analytics. He has published more than 80 IEEE/ACM Transactions and IEEE VIS papers. He actively served in many leading conferences and journals, like IEEE PacificVIS steering committee, ChinaVIS steering committee, paper cochairs of IEEE VIS, IEEE PacificVIS, IEEE LDAV and ACM SIGGRAPH Asia VisSym. He is an associate editor of IEEE TVCG, IEEE TBG, ACM TIST, IEEE T-SMC-S, IEEE TIV, IEEE CG&A, FCS, and JOV. More information can be found at: http://www.cad.zju.edu.cn/home/chenwei.