Communicative Agents for Software Development

Chen Qian[♠] **Xin Cong**[♠] **Wei Liu**[♠] **Cheng Yang**[♣] **Weize Chen**[♠] **Yusheng Su**[♠]
Yufan Dang[♠] **Jiahao Li**[♦] **Juyuan Xu**[▲] **Dahai Li**[★] **Zhiyuan Liu**[♠✉] **Maosong Sun**[♠✉]
[♠]Tsinghua University [♣]Beijing University of Posts and Telecommunications
[♦]Dalian University of Technology [▲]Brown University [★]Modelbest Inc.
qianc62@gmail.com liuzy@tsinghua.edu.cn sms@tsinghua.edu.cn

Figure 1: ChatDev, our virtual chat-powered company for software development, brings together "software agents" from diverse social identities, including chief officers, professional programmers, test engineers, and art designers. When presented with a preliminary task by a human "client" (*e.g.*, "*develop a gomoku game*"), the software agents at ChatDev engage in effective communication and mutual verification through collaborative chatting. This process enables them to automatically craft comprehensive software solutions that encompass source codes, environment dependencies, and user manuals.

✉: Corresponding Authors.

Preprint.

Abstract

Software engineering is a domain characterized by intricate decision-making processes, often relying on nuanced intuition and consultation. Recent advancements in deep learning have started to revolutionize software engineering practices through elaborate designs implemented at various stages of software development. In this paper, we present an innovative paradigm that leverages large language models (LLMs) throughout the entire software development process, streamlining and unifying key processes through natural language communication, thereby eliminating the need for specialized models at each phase. At the core of this paradigm lies ChatDev, a virtual chat-powered software development company that mirrors the established waterfall model, meticulously dividing the development process into four distinct chronological stages: designing, coding, testing, and documenting. Each stage engages a team of "software agents", such as programmers, code reviewers, and test engineers, fostering collaborative dialogue and facilitating a seamless workflow. The chat chain acts as a facilitator, breaking down each stage into atomic subtasks. This enables dual roles, allowing for proposing and validating solutions through context-aware communication, leading to efficient resolution of specific subtasks. The instrumental analysis of ChatDev highlights its remarkable efficacy in software generation, enabling the completion of the entire software development process in under seven minutes at a cost of less than one dollar. It not only identifies and alleviates potential vulnerabilities but also rectifies potential hallucinations while maintaining commendable efficiency and cost-effectiveness. The potential of ChatDev unveils fresh possibilities for integrating LLMs into the realm of software development. Our code is available at `https://github.com/OpenBMB/ChatDev`.

1 Introduction

> "*Collaboration allows us to know more than we are capable of knowing by ourselves. It empowers us to think differently, access information we wouldn't have otherwise, and combine ideas as we work together towards a shared goal.*"
>
> ——Paul Solarz

Software engineering entails a methodical and disciplined approach to the development, operation, and maintenance of software systems [4]. However, the complexity of software intelligence often leads to decisions based on intuition and limited consultation with senior developers [14]. Recent advancements in deep learning techniques have prompted researchers to explore their application in software engineering, aiming to improve effectiveness, efficiency, and cost reduction . Prior studies in deep learning-based software engineering have addressed various tasks, categorized as software requirements, design, implementation, testing, and maintenance [34; 29]. The software development process involves multiple roles, including organizational coordination, task allocation, code writing, system testing, and documentation preparation. It is a highly complex and intricate activity that demands meticulous attention to detail due to its long development cycles [17; 4].

In recent years, large language models (LLMs) have achieved significant milestones in the field of natural language processing (NLP) [5] and computer vision (CV) [35]. After training on massive corpora using the "next word prediction" paradigm, LLMs have demonstrated impressive performance on a wide range of downstream tasks, such as context-aware question answering, machine translation, and code generation. In fact, the core elements involved in software development, namely codes and documents, can both be regarded as "language" (*i.e.,* sequences of characters) [7]. From this perspective, this paper explores an end-to-end software development framework driven by LLMs, encompassing requirements analysis, code development, system testing, and document generation, aiming to provide a unified, efficient, and cost-effective paradigm for software development.

Directly generating an entire software system using LLMs can result in code hallucinations to a certain extent, similar to the phenomenon of hallucination in natural language knowledge querying [2]. These hallucinations include incomplete implementation of functions, missing dependencies, and potential undiscovered bugs. Code hallucinations arise primarily due to two reasons. Firstly,

the lack of task specificity confuses LLMs when generating a software system in one step. Granular tasks in software development, such as analyzing user/client requirements and selecting programming languages, provide guided thinking that is absent in the high-level nature of the task handled by LLMs. Secondly, the absence of cross-examination in decision-making poses significant risks [9]. Individual model instances propose a diverse range of answers, throwing the requirements to debate or examine the responses from other model instances to converge on a single and more accurate common answer [12], such as code peer-review and suggestion feedbacks.

To address the aforementioned challenges, we "establish" a virtual chat-powered software technology company – ChatDev. It follows the classic waterfall model [3] and divides the process into four phases: designing, coding, testing, and documenting. At each phase, ChatDev recruits multiple "software agents" with different roles, such as programmers, reviewers, and testers. To facilitate effective communication and collaboration, ChatDev utilizes a proposed chat chain that divides each phase into atomic subtasks. Within the chat chain, each node represents a specific subtask, and two roles engage in context-aware, multi-turn discussions to propose and validate solutions. This approach ensures that client requirements are analyzed, creative ideas are generated, prototype systems are designed and implemented, potential issues are identified and addressed, debug information is explained, appealing graphics are created, and user manuals are generated. By guiding the software development process along the chat chain, ChatDev delivers the final software to the user, including source code, dependency environment specifications, and user manuals.

The experiment analyzed all the software produced by ChatDev in response to 70 user requirements. On average, ChatDev generated 17.04 files per software, alleviated potential code vulnerabilities caused by code hallucinations 13.23 times, had a software production time of 409.84 seconds, and incurred a manufacturing cost of $0.2967. Discussions between a reviewer and a programmer led to the identification and modification of nearly twenty types of code vulnerabilities, while discussions between a tester and a programmer resulted in the identification and resolution of more than ten types of potential bugs. In summary, our main contributions are as follows:

- We propose ChatDev, a chat-based software development framework. By merely specifying a task, ChatDev sequentially handles designing, coding, testing, and documenting. This new paradigm simplifies software development by unifying main processes through language communication, eliminating the need for specialized models at each phase.
- We propose the *chat chain* to decompose the development process into sequential atomic subtasks. Each subtask requires collaborative interaction and cross-examination between two roles. This framework enables multi-agent collaboration, user inspection of intermediate outputs, error diagnoses, and reasoning intervention. It ensures a granular focus on specific subtasks within each chat, facilitating effective collaboration and promoting the achievement of desired outputs.
- To further alleviate potential challenges related to code hallucinations, we introduce the thought instruction mechanism in each independent chat process during code completion, reviewing, and testing. By performing a "role flip", an instructor explicitly injects specific thoughts for code modifications into instructions, thereby guiding the assistant programmer more precisely.
- The experiments demonstrate the efficiency and cost-effectiveness of ChatDev's automated software development process. Through effective communication, proposal, and mutual examination between roles in each chat, the framework enables effective decision-making.

2 ChatDev

Similar to hallucinations encountered when using LLMs for natural language querying [2], directly generating entire software systems using LLMs can result in severe code hallucinations, such as incomplete implementation, missing dependencies, and undiscovered bugs. These hallucinations may stem from the lack of specificity in the task and the absence of cross-examination in decision-making. To address these limitations, as Figure 1 shows, we establish a virtual chat-powered software technology company – ChatDev, which comprises of recruited agents from diverse social identities, such as chief officers, professional programmers, test engineers, and art designers. When presented with a task, the diverse agents at ChatDev collaborate to develop a required software, including an executable system, environmental guidelines, and user manuals. This paradigm revolves around leveraging large language models as the core thinking component, enabling the agents to simulate

the entire software development process, circumventing the need for additional model training and mitigating undesirable code hallucinations to some extent.

2.1 Chat Chain

ChatDev employs the widely adopted waterfall model, a prominent software development life cycle model, to divide the software development process into four distinct phases: *designing*, *coding*, *testing*, and *documenting*. In the designing phase, innovative ideas are generated through collaborative brainstorming, and technical design requirements are defined. The coding phase involves the development and review of source code, while the testing phase integrates all components into a system and utilizes feedback messages from interpreter for debugging. The documenting phase encompasses the generation of environment specifications and user manuals. Each of these phases necessitates effective communication among multiple roles, posing challenges in determining the sequence of interactions and identifying the relevant individuals to engage with.

To address this, we propose a generalized architecture by breaking down each phase into multiple atomic chats, each with a specific focus on task-oriented role-playing involving two distinct roles. Through the exchange of instructions and collaboration between the participating agents, the desired output for each chat, which forms a vital component of the target software, is achieved. An illustration of this process is depicted in Figure 2, where a sequence of intermediate task-solving chats, referred to as a "chat chain", is presented. In each chat, an instructor initiates instructions, guiding the dialogue towards task completion, while the assistant follows the instructions, provides suitable solutions, and engages in discussions regarding feasibility. The instructor and assistant cooperate through multi-turn dialogues until they reach a consensus and determine that the task has been successfully accomplished.

The chat chain provides a transparent view of the software development process, shedding light on the decision-making path and offering opportunities for debugging when errors arise, which enables users to inspect intermediate outputs, diagnose errors, and intervene in the reasoning process if necessary. Besides, chat chain ensures a granular focus on specific subtasks within each phase, facilitating effective collaboration and promoting the attainment of desired outputs.

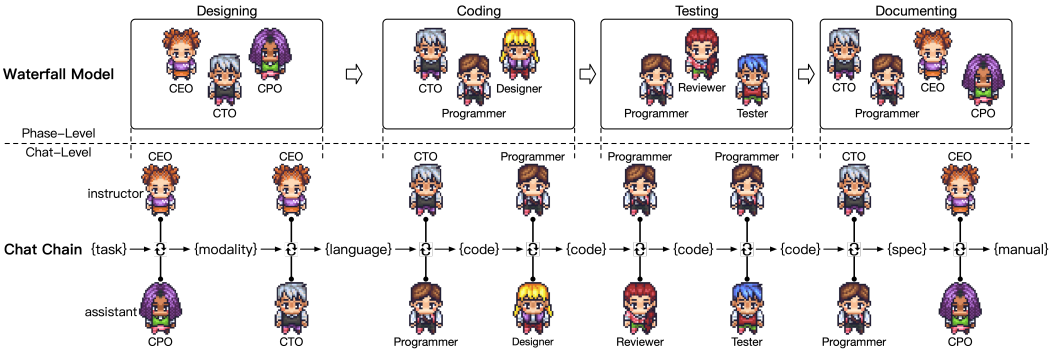
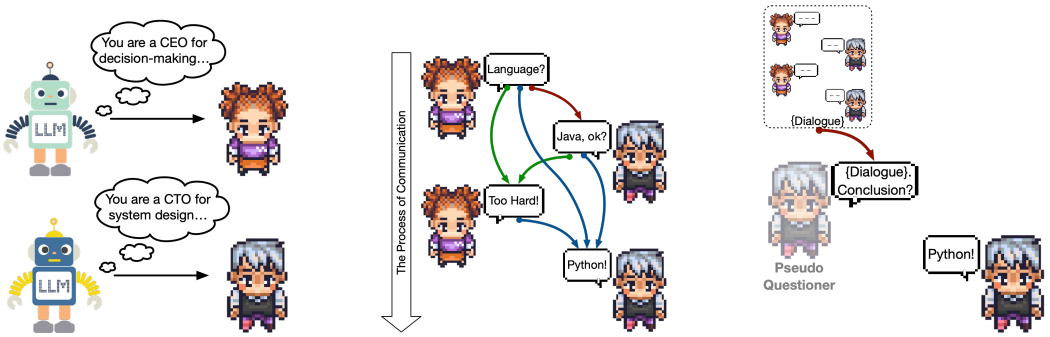


Figure 2: The proposed architecture of ChatDev consists of phase-level and chat-level components. At the phase level, the waterfall model is used to break down the software development process into four sequential phases. At the chat level, each phase is further divided into atomic chats. These atomic chats involve task-oriented role-playing between two agents, promoting collaborative communication. The communication follows an instruction-following style, where agents interact to accomplish a specific subtask within each chat.

2.2 Designing

In the designing phase, ChatDev receives an initial idea from a human client. This phase involves three predefined roles: CEO (chief executive officer), CPO (chief product officer), and CTO (chief technology officer). The chat chain then breaks down the designing phase into sequential atomic chatting tasks, including decisions regarding the target software's modality (CEO and CPO) and the programming language (CEO and CTO).



(a) Role Specialization (b) Memory Stream (c) Self-Reflection

Figure 3: Three key mechanisms utilized in each chat. Role specialization ensures that each agent fulfills their designated functions and contributes effectively to the task-oriented dialogue. The memory stream maintains a comprehensive record of previous dialogues within the chat, enabling agents to make informed decisions. Self-reflection prompts the assistant to reflect on proposed decisions when both parties reach a consensus without triggering predefined termination conditions.

Role Assignment System prompts/messages are used to assign roles to each agent during the role-playing process. In contrast to other conversational language models, our approach to prompt engineering is restricted solely to the initiation of role-playing scenarios. The instructor is denoted as $\mathcal{P}_{\mathcal{I}}$, while the assistant's system prompt/message is denoted as $\mathcal{P}_{\mathcal{A}}$. These prompts assign roles to the agents before the dialogues begin. Let $\mathcal{L}_{\mathcal{I}}$ and $\mathcal{L}_{\mathcal{A}}$ represent two large language models. Using the system message, we have $\mathcal{I} \leftarrow \mathcal{L}_{\mathcal{I}}^{\mathcal{P}_{\mathcal{I}}}$ and $\mathcal{A} \leftarrow \mathcal{L}_{\mathcal{A}}^{\mathcal{P}_{\mathcal{A}}}$, which serve as the instructor and assistant agents (Figure 3(a)), respectively. In our framework, the instructor initially acts as a CEO, engaging in interactive planning, while the assistant assumes the role of CPO, executing tasks and providing responses. To achieve role specialization, we employ *inception prompting* [23], which has proven effective in enabling agents to fulfill their roles. The instructor and assistant prompts encompass vital details concerning the designated task and roles, communication protocols, termination criteria, and constraints aimed at preventing undesirable behaviors (*e.g.,* instruction redundancy, uninformative responses, infinite loops, etc.).

Memory Stream The memory stream [32] is a mechanism that maintains a comprehensive record of an agent's previous dialogues, assisting in subsequent decision-making in an utterance-aware manner. Formally, the instructor's message at time t is denoted as $\mathcal{I}_t$, the assistant's message as $\mathcal{A}_t$, and the related decisions as $\mathcal{S}_t$. Equation 1 encapsulates the collection of conversational messages up to time t.

$$\mathcal{M}_t = \langle (\mathcal{I}_1, \mathcal{A}_1), (\mathcal{I}_2, \mathcal{A}_2), \cdots, (\mathcal{I}_t, \mathcal{A}_t) \rangle \qquad \mathcal{S}_t \leftarrow \psi(\mathcal{I}_t, \mathcal{A}_t) \tag{1}$$

where ψ represents a LLM-based decision extractor which can be implemented via communication protocol detection or self-reflection (detailed below). In the succeeding time step $t+1$, the instructor leverages the historical dialogue message set $\mathcal{M}_t$ to impart a fresh instruction, $\mathcal{I}_{t+1}$, which is then conveyed to the assistant along with $\mathcal{M}_t$, as illustrated in Figure 3(b). The assistant responds with a solution or message, denoted as $\mathcal{A}_{t+1}$ in Equation 2:

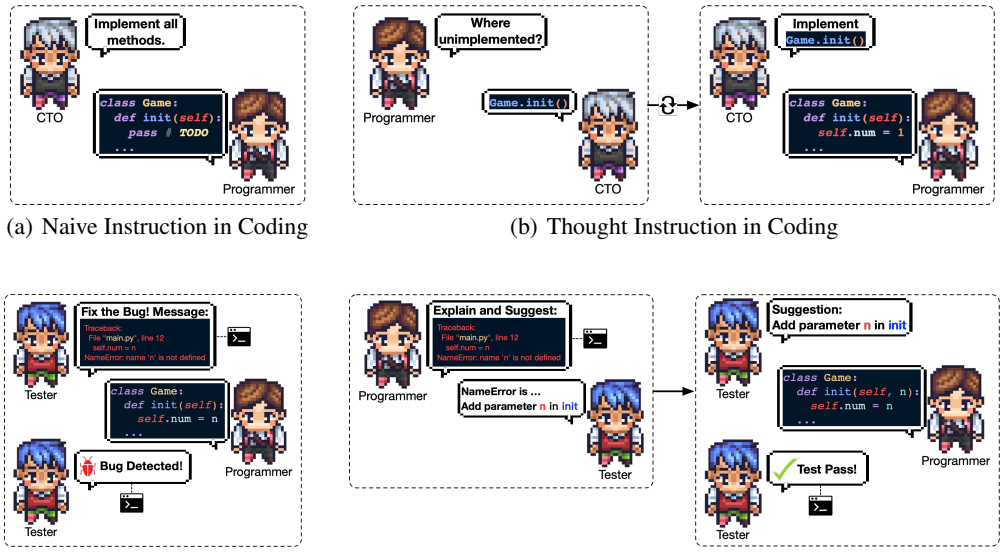
$$\mathcal{I}_{t+1} = \mathcal{A}(\mathcal{M}_t, \mathcal{S}_t) \qquad \mathcal{A}_{t+1} = \mathcal{I}(\mathcal{M}_t, \mathcal{I}_{t+1}, \mathcal{S}_t) \tag{2}$$

Following the acquisition of the solution $\mathcal{A}_{t+1}$ in response to the instruction $\mathcal{I}_{t+1}$, the message stream undergoes an update process utilizing Equation 3:

$$\mathcal{M}_{t+1} = \mathcal{M}_t \cup (\mathcal{I}_{t+1}, \mathcal{A}_{t+1}) \qquad \mathcal{S}_{t+1} = \mathcal{S}_t \cup \psi(\mathcal{I}_{t+1}, \mathcal{A}_{t+1}) \tag{3}$$

We establish communication protocols through prompts. For example, an ending message satisfying specific formatting requirements (e.g., "<MODALITY>: Desktop Application") is generated when both parties reach a consensus. The system monitors communication to ensure compliance with the designated format, allowing for the conclusion of the current dialogue.

Self-Reflection Occasionally, we have observed dialogues where both parties reach a consensus but do not trigger the predefined communication protocols as termination conditions. In such cases,



(a) Naive Instruction in Coding (b) Thought Instruction in Coding

(c) Naive Instruction in Testing (d) Thought Instruction in Testing

Figure 4: The thought instruction mitigates code hallucinations during the coding and testing phases. Instead of providing generic instructions, thought instruction involves role swapping to inquire about unimplemented methods or explain feedback messages caused by bugs. This step allows for a clearer understanding of the existing code and identifies the specific gaps that need to be addressed. By gaining this awareness, the roles can then switch back, and the instructor can provide more specific instructions to guide the programmer accurately.

we introduce a *self-reflection* mechanism, which involves extracting and retrieving memories. To implement this mechanism, we enlist a "pseudo self" as a new questioner and initiate a fresh chat. The pseudo questioner informs the current assistant of all the historical records from previous dialogues and requests a summary of the conclusive information from the dialogue, as shown in Figure 3(c). This mechanism effectively encourages the assistant to reflect upon the decisions proposed and discussed during the dialogue.

2.3 Coding

The coding phase involves three predefined roles: CTO, programmer, and art designer. The chat chain decomposes the coding phase into sequential atomic chatting tasks, such as generating complete codes (CTO and programmer) and devising a graphical user interface (designer and programmer). Based on the main designs discussed in the previous phase, the CTO instructs the programmer to implement a software system using markdown format. The programmer generates codes in response and extracts the corresponding codes based on markdown format. The designer proposes a user-friendly graphical user interface (GUI) that uses graphical icons for user interaction instead of text-based commands. Then, the designer creates visually appealing graphics using external text-to-image tools [35], which the programmer incorporates into the GUI design using standard toolkits.

Code Management To handle complex software systems, ChatDev utilizes object-oriented programming languages like Python. The modularity of object-oriented programming allows for self-contained objects, aiding troubleshooting and collaborative development. Reusability enables code reuse through inheritance, reducing redundancy. We introduce the "version evolution" mechanism to restrict visibility to the latest code version between roles, discarding earlier code versions from the memory stream. The programmer manages the project using Git-related commands. Proposed code modifications and changes increment the software version by 1.0. Version evolution gradually eliminates code hallucinations. The combination of object-oriented programming and version evolution is suitable for dialogues involving long code segments.

Thought Instruction Traditional question answering can lead to inaccuracies or irrelevant information, especially in code generation, where naive instructions may result in unexpected hallucinations.

This issue becomes particularly problematic when generating code. For instance, when instructing the programmer to implement all unimplemented methods, a naive instruction may result in hallucinations, such as including methods that are reserved as unimplemented interfaces. To address this, we propose the "*thought instruction*" mechanism, inspired by chain-of-thought prompting [44]. It involves explicitly addressing specific problem-solving thoughts in instructions, akin to solving subtasks in a sequential manner. As shown in Figure 4(a) and 4(b), thought instruction includes swapping roles to inquire about which methods are not yet implemented and then switching back to provide the programmer with more precise instructions to follow. By incorporating thought instruction, the coding process becomes more focused and targeted. The explicit expression of specific thoughts in the instructions helps to reduce ambiguity and ensures that the generated code aligns with the intended objectives. This mechanism enables a more accurate and context-aware approach to code completion, minimizing the occurrence of hallucination and resulting in more reliable and comprehensive code outputs.

2.4 Testing

Even for human programmers, there is no guarantee that the code they write on the first attempt is always error-free. Rather than discarding incorrect code outright, humans typically analyze and investigate code execution results to identify and rectify implementation errors [8]. In ChatDev, the testing phase involves three roles: programmer, reviewer, and tester. The process consists of sequential atomic chatting tasks, including peer review (programmer and reviewer) and system testing (programmer and tester). Peer review, or static debugging, examines source code to identify potential issues. System testing, a form of dynamic debugging, verifies software execution through tests conducted by the programmer using an interpreter. This testing focuses on evaluating application performance through black-box testing.

In our practice, we observed that allowing two agents to communicate solely based on feedback messages from an interpreter does not result in a bug-free system. The programmer's modifications may not strictly follow the feedback, leading to hallucinations. To address this, we further employ the thought instruction mechanism to explicitly express debugging thoughts in the instructions (Figure 4(c) and 4(d)). The tester executes the software, analyzes bugs, proposes modifications, and instructs the programmer accordingly. This iterative process continues until potential bugs are eliminated and the system runs successfully.

In cases where an interpreter struggles with identifying fine-grained logical issues, the involvement of a human client in software testing becomes optional. ChatDev enables the human client to provide feedback and suggestions in natural language, similar to a reviewer or tester, using black-box testing or other strategies. ChatDev, based on human input, can understand and utilize this feedback to refine the software system.

2.5 Documenting

After the designing, coding, and testing phases, ChatDev employs four agents (CEO, CPO, CTO, and programmer) to generate software project documentation. Using large language models, we leverage few-shot prompting [5] with in-context examples for document generation. The CTO instructs the programmer to provide configuration instructions for environmental dependencies, resulting in a document like `requirements.txt`. This document allows users to configure the environment independently. Simultaneously, the CEO communicates requirements and system design to the CPO, who generates a user manual.

3 Experiments

Our experimental setup employs the "`ChatGPT-turbo-16k`" version of ChatGPT to simulate multi-agent software development. The language model temperature is set to 0.2 for controlled generation. In the coding phase, we allow a maximum of 5 attempts for code completion. The reviewer is permitted 5 chats to propose modifications, and a maximum of 5 software system tests are conducted in the testing phase. For Python-based systems, we use Python 3.8.16 as the interpreter for testing. Camel [23] has curated an instruction-following dialogue dataset, which spans across 20 programming languages, 50 domains, and 50 tasks per domain. From this extensive task set, we randomly selected

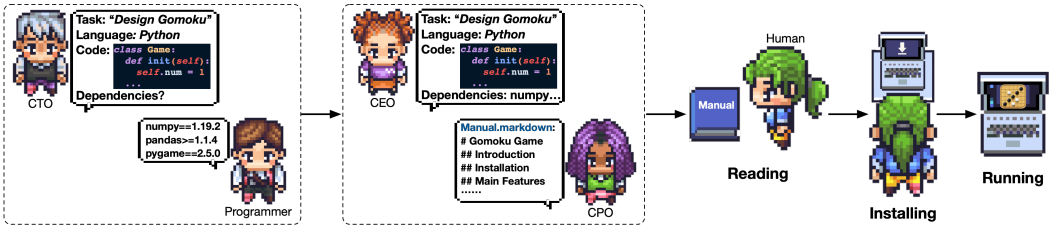


Figure 5: The documenting phase involves generating relevant documents, such as external dependency specifications and user instructions. The user manual provides comprehensive information about the software's technical architecture, installation instructions, and features, serving as a valuable resource for users. Once the dependencies are installed, a human client can execute the software using a suitable interpreter.

70 tasks[1], including both specific and relatively abstract cases, to serve as the basis for analysis in our ChatDev software development.

Software Statistics We performed a statistical analysis on the software systems generated by ChatDev. Key metrics, including the total dialogue turns, consumed tokens, software files, image assets, and version updates, were examined. Table 1 presents these metrics, providing valuable insights into the communication-based software development process. It offers a comprehensive overview of ChatDev's development, covering aspects such as versioning, file composition, code complexity, and development iterations.

Table 1: The statistical analysis of ChatDev's software development, including minimum (Min), maximum (Max), and average (Avg.) values for various aspects.

	Min	Max	Avg.
# Code Files	2.00	8.00	4.26
# Asset Files	0.00	21.00	8.74
# Document Files	4.00	5.00	4.04
# Lines of Source Codes	39.00	359.00	131.61
# Lines of Dependencies	1.00	5.00	2.90
# Lines of User Manual	31.00	232.00	53.96
# Version Updates	5.00	42.00	13.23
# Software Re-development	1.00	5.00	1.40

The generated software typically includes 2 to 8 code files, with an average of 4.26 files. Asset files, created by the art designer using external tools [35], range from 0 to 21, with an average of 8.74 files. Here are some examples of concise text descriptions through which programmers request the designer to create images, such as "`The text entry field where the user can input their data`", "`The background image for the financial dashboard`", and "`The image representing the player character in the game`". The software is accompanied by 4 to 5 document files on average, such as dependency requirements specifications, user manuals, development logs, and software meta information.

The software developed by ChatDev typically ranges from 39 to 359 lines of code, with an average of 131.61 lines[2]. The data suggests that ChatDev tends to produce software with relatively small-scale code. This is partly because the design of object-oriented programming, whose reusability enables code reuse through inheritance, reducing redundancy. We also noted that when the user specified a less specific task, the resulting source code generated by ChatDev tended to be shorter, averaging around 110.97 lines. This is primarily attributed to ChatDev employing high-level logic to fulfill non-specific tasks, often generating code that focuses on providing print information for interface representation. Therefore, we recommend providing ChatDev with specific instructions, such as desired software features, system rules, UI design, and other detailed specifications. By providing

[1]For exmaple, "*Implement a Gomoku game using Python, incorporating an AI opponent with varying difficulty levels*" or "*Create a Python program to develop an interactive weather dashboard*".

[2]This count includes only lines that contain meaningful code, excluding blank lines.

clearer and more specific instructions, users can guide ChatDev to produce more comprehensive and tailored codes that aligns with their specific requirements. The number of environment dependencies, which indicates the external software components required, ranges from 1 to 5, with an average of 2.90 dependencies. ChatDev's software environment typically includes `numpy`, `matplotlib`, `pandas`, `tkinter`, `pillow`, or `flask`. The user manual for the software consists of 31 to 232 lines, with an average of 53.96 lines. Based on our experience, the user manual commonly covers sections such as `Introduction`, `Quick Install`, `Main Features`, `Usage Instructions`, etc,.

The number of version updates for the software ranges from 5 to 42, with an average of 13.23 updates. This indicates that the source code undergoes approximately 13 modifications on average, reflecting the collaborative effort among agents in alleviating code hallucination issues throughout the software development process, including code completion, coding, and testing. In exceptional cases where the software fails to pass the maximum number of tests, ChatDev takes proactive measures by engaging in full-scale software re-engineering. In most cases, the software development process involves 1 to 5 development cycles, with an average of 1.40 cycles.

In our experiments, we effortlessly set up the sandbox environment by directly installing the required software dependencies. Subsequently, we executed the generated software using the main function. Remarkably, approximately 86.66% of the software systems executed flawlessly, showcasing the robustness and reliability of our developed software. However, a small fraction, 13.33% of the software, encountered execution failures. Upon analyzing the failed software creations, we identified two primary contributing factors. Firstly, in 50% of the cases, the failure was attributed to the token length limit of the API. This limitation prevented obtaining the complete source code within the specified length constraint for code generation. Such challenges are particularly evident when dealing with complex software systems or scenarios requiring extensive code generation. The remaining 50% of the failed software creations were primarily affected by external dependency issues. These challenges emerged when certain dependencies were either unavailable in the cloud or incorrectly versioned, resulting in conflicts and unavailability of specific application programming interfaces (APIs) in the current version. These external dependency-related issues underscore the significance of meticulous management and coordination of the required software components to ensure smooth execution and functionality. Overall, despite encountering a small percentage of failures, our experimental findings demonstrate the feasibility and effectiveness of ChatDev in generating executable software systems, with the majority of the systems successfully executing.

Duration Analysis We conducted a duration analysis to examine the software production time for different request prompts using ChatDev. The variability in development times across prompts reflects the varying complexity and clarity of the assigned tasks. The graph in Figure 6 provides a visual representation of this distribution. The longest software production duration, represented by the tallest bar on the left side of the graph, was 1030.00 seconds. This extended time was due to extensive dialogue and communication between the reviewer and programmer, leading to a detailed modification scheme. In contrast, the shortest bar on the right end of the graph indicates a minimum software development time of 169.00 seconds. This shorter duration was attributed to the absence of significant bugs and fewer dialogues during coding and testing stages. On average, the development of small-sized software and interfaces using ChatDev took 409.84 seconds, less than 7.00 minutes. In comparison, traditional custom software development cycles, even within agile software development methods, typically require 2 to 4 weeks or even several months per cycle [22; 10].

Dialogue Statistics - In ChatDev, we employed a chat chain mechanism to facilitate software development. Each chat chain represents the production of software for a specific task and consists of multiple multi-utterance chat rounds. During these rounds, agents engage in discussions to address predefined subtasks, such as language choices, proposing solutions, and making final decisions. After completing all subtasks, a chat chain concludes with the development of the software product. For our case study tasks, we analyzed the chat chains and collected statistics, including the total number of utterances and prompt tokens used. These statistics are presented in Table 2.

We noticed occasional instances of repetitive expressions of gratitude in the dialogue, even after reaching a consensus and making decisions. However, this phenomenon does not significantly impact the final outcome. The self-reflection mechanism effectively allows agents to extract decision results and conclusions from the dialogue using text summarization-like abilities. This mechanism helps agents avoid unnecessary dialogue and focus on extracting meaningful information. The

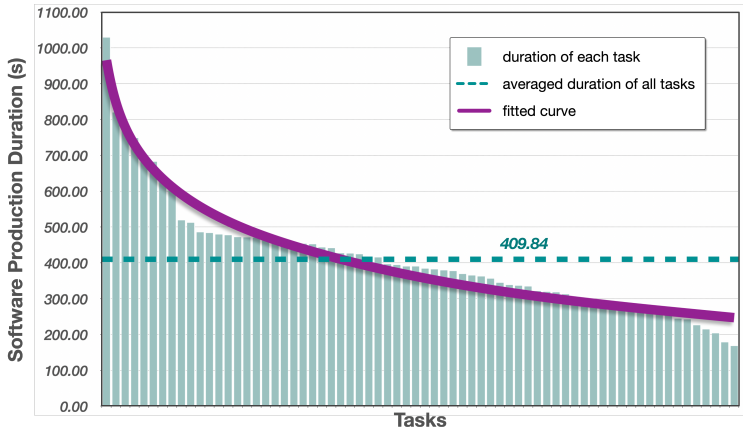


Figure 6: Duration Distribution. The bars in the graph are arranged in descending order, showcasing the distribution of software development runtime for different tasks.

Table 2: The statistical analysis of all dialogues in chat chains.

	Min	Max	Avg.
# Self-Reflection	1.00	4.00	1.24
# Utterances	24.00	104.00	45.60
# Prompt Tokens	11,119.00	91,208.00	36,902.23
# Completion Tokens	3,161.00	27,162.00	11,567.37
# Total Tokens	15,294.00	111,019.00	48,469.60

self-reflection number in the dialogue ranges from 1 to 4, with an average of 1.24. In most cases, agents can autonomously conclude the dialogue based on predefined communication protocols.

On average, a chat chain contains 45.60 utterances, ranging from a minimum of 24 to a maximum of 104. The count of utterances encompasses discussions related to achievability of subtasks, evaluations of generated code quality, feedback on testing, advice for improvements, and the actual writing and generation of software code files and documents. Likewise, we have observed that ChatDev tends to engage in less communication through utterances for abstract tasks compared to specific tasks, averaging around 34.40 utterances. Analysis of the dialogues revealed that during the design and coding stages, agents conducted multiple rounds of discussions to delve into the details of numerous requirements or propose modification suggestions. These discussions aimed to make informed decisions regarding the specific tasks at hand. This phenomenon aligns with real-world practices, where addressing specific tasks often involves more detailed discussions and deliberations.

We monitored API interactions and token usage during software production in ChatDev. On average, ChatDev requires 36,902.23 prompt tokens, 11,567.37 completion tokens, and a total of 48,469.60 tokens to develop a single software. The average total cost in software production is approximately \$0.1569[3]. To determine the overall cost of software development with ChatDev, we also consider the cost of designer-produced images. The average designer cost is \$0.1398 per software for each software production involving 8.74 graphics creations on average. Thus, the average software development cost at ChatDev is \$0.2967, significantly lower than traditional custom software development companies' expenses [18; 21; 31].

Reviewer-Programmer Dialogue Analysis In this section, we delve into the primary exchanges between the reviewer and the programmer, specifically concerning code-related matters during the coding phase. We summarize the reviewer's evaluations of the programmer's source code at the coding stage. Figure 7 provides a visual representation of the reviewer's suggestions in the form of pie charts. As depicted in the figure, the most frequently discussed issue in the reviewer-programmer communication during code review is "`methods not implemented`" (34.85%). This challenge commonly arises in code generation for complex models, where core functionalities often receive

[3]Based on official API prices for July 2023.

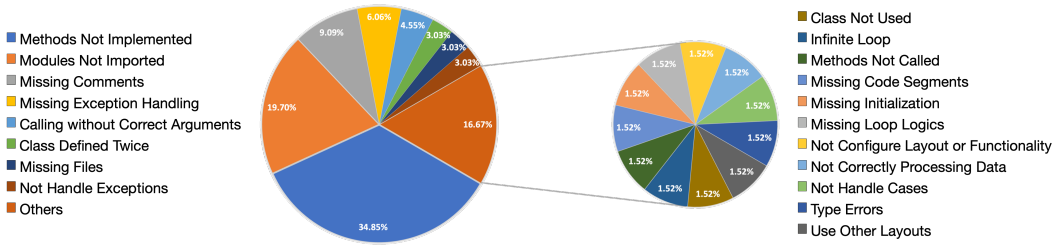


Figure 7: Distribution of Reviewer's Suggestions. Each color in the pie chart represents a specific category of suggestions provided by the reviewer.

placeholder labels (such as "`pass`" in Python) to be further completed. Additionally, the dialogue frequently addresses the topic of "`modules not imported`" (19.70%). This issue emerges from the nature of code generation, where the generated code tends to overlook minor details. However, in the context of code generation, ensuring the code's executability becomes crucial. Fortunately, the thought instruction mechanism proposed in this paper effectively tackles these issues by compelling the reviewer to identify incomplete methods and requiring the programmer to fill them. This mechanism can be applied to other scenarios where tasks are completed based on large models but with certain parts missing. Interestingly, the reviewer also emphasizes the importance of code robustness. They underscore considerations for handling potential exceptions in the future and offer hints on avoiding duplicate categories (3.03%). Additionally, the reviewer provides suggestions regarding unused classes in the code (1.52%), identifies infinite loops (1.52%), and emphasizes the necessity of proper environment initialization (1.52%).

Tester-Programmer Dialogue Analysis In a similar fashion, we analyze the debug dialogue between the tester and the programmer during the testing phase and categorize the main types of bugs encountered. The results are presented in Figure 8. As observed in the figure, the most frequent debug issue between the tester and the programmer is "`module not found`" (45.76%), accounting for nearly half of the cases. This reflects the model's tendency to overlook very fine details, despite their simplicity. Fortunately, with the thought instruction mechanism proposed in this paper, such bugs can often be easily resolved by importing the required class or method. The second most common types of errors are "`attribute error`" and "`unknown option`", each accounting for 15.25% of the cases. "`attribute error`" refers to errors in the usage of class attributes, while "`unknown option`" indicates errors in the parameters of method calls. Another common type of error is "`import error`" which is similar to "`module not found`" and is primarily caused by mistakes in the import statements, such as importing the wrong class or using an incorrect import path. In addition to these common error types, ChatDev has the capability to detect relatively rare error types such as improperly initialized GUI (5.08%), incorrect method calling (1.69%), missing file dependencies (1.69%), unused modules (1.69%), decorator syntax errors (1.69%), and more.

Case Study Figure 9 showcases an example of ChatDev developing a Gomoku game (*a.k.a.* also known as "*Five in a Row*" and "*Gobang*"). In the left, we see the result of a naive software created without GUI. This version of the game can only be played through a command terminal, limiting its interactivity and overall enjoyment. In contrast, by incorporating GUI design, ChatDev can

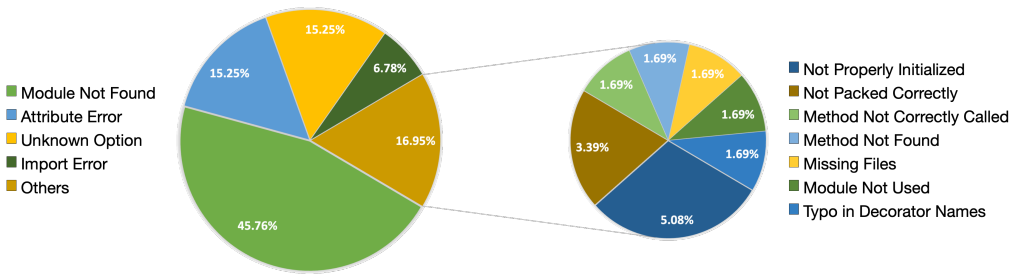


Figure 8: Distribution of Tester's Suggestions. Each color in the pie chart represents a specific category of bugs provided by the tester.

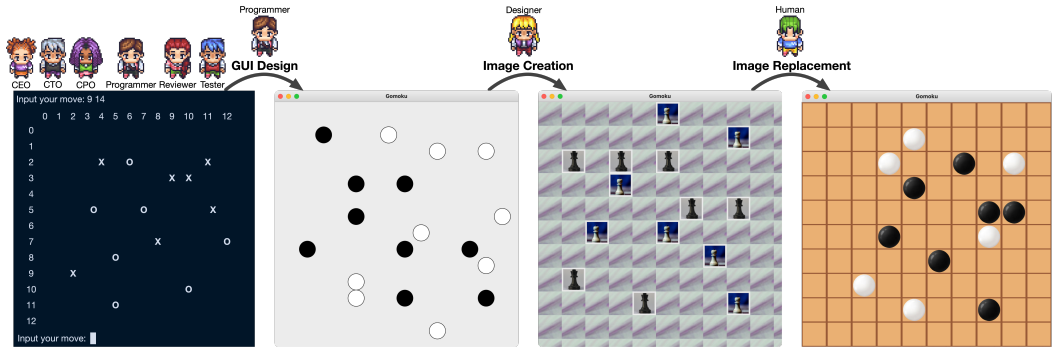


Figure 9: The producted software of the task: "*design a basic Gomoku game*".

create a visually appealing small game. This version surpasses the interface-less version in terms of interactivity and user experience, providing a more enjoyable and engaging gameplay environment. Furthermore, ChatDev's designer can assist the programmer in creating additional graphics to enhance the GUI's aesthetics and usability, without compromising its functionality. These graphics, carefully crafted by the designer, contribute to making the GUI more visually pleasing and user-friendly.

Additionally, if human users are unsatisfied with the images created by the art designer, they have the flexibility to manually replace the original images after ChatDev completes the software. This allows for further customization according to users' preferences, without affecting the software's core functionality. Users can tailor the visual elements to their liking, resulting in a personalized software experience that aligns with their individual preferences.

For a more comprehensive understanding, we exemplify the dialogue processes that make programming language choices in designing. More exemplary dialogues extracted from the chat chain of the Gomoku game are shown in Appendix A, including the prompts we designed and the dialogue process between agents at each phase. Please note that, due to space constraints, we only display key information during the dialogue, omitting overly fine-grained details.

Designing: Programming Language

: CEO (Chief Executive Officer) : CTO (Chief Technology Officer)

: To satisfy the new user's demand and make a desktop application realizable, you should propose a concrete programming language.

: Python with the Pygame library is a suitable choice for implementing a basic Gomoku game.

: Great! Python is a versatile and widely-used programming language that will be suitable for implementing a basic Gomoku game. Its simplicity and readability make it a popular choice for beginners and experienced developers alike.

: Let's get started with the development of the basic Gomoku game in Python using the Pygame library!

: <END> Python.

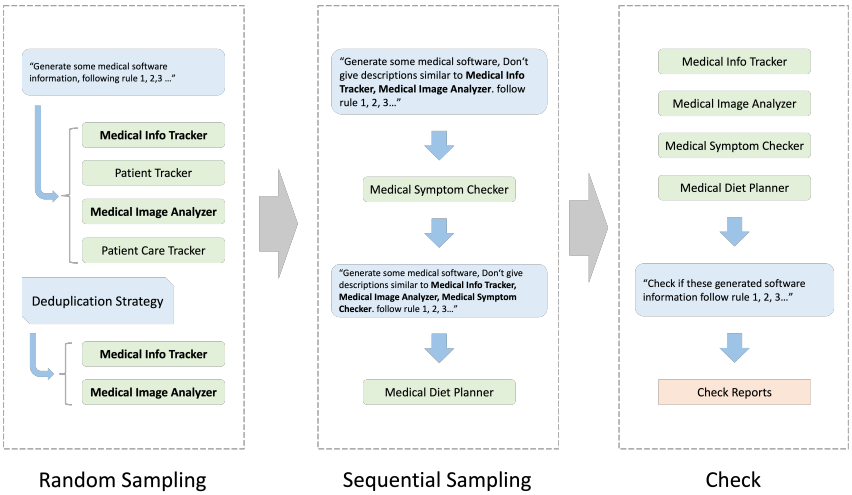


Figure 10: The three-stage process of NLDD creation. We only show the generated names of software and the prompt in the figure is for example only.

4 NLDD Dataset

We collect and open-source[4] a large and diverse dataset named NLDD (Natural Language Dataset for Dev), which contains 1,200 software prompt data for the Natural Language to Software (NL2Software) task. Each sample in NLDD includes the name, description, and category of the software.

NLDD is created by prompting ChatGPT with a three-stage strategy and well-designed rules. We collect the main software category from four prominent software store platforms including Ubuntu Snap Shop, Google Play Store, Microsoft Store, and Apple App Store. We further sorted out five major categories with 40 subcategories and asked ChatGPT to generate software data for these categories. See Figure 11 in the appendix for all the category details.

To circumvent the generation of repetitive content, NLDD is created with a Query Prompt-based three-stage strategy, including random sampling, prompt sampling, and check. As shown in Figure 10, this strategy initially establishes datasets by random sampling some software data, then records existing data, granting ChatGPT autonomy to produce novel entries.

1. Random Sampling: First, ChatGPT is independently inquired multiple times to obtain software information under a certain category, and then the duplication is removed at the token granularity of the software name.

2. Sequential Sampling: Then we add the generated software information in sequence in the form of negative prompts, requiring ChatGPT to continue generating unique software information.

3. Check: Although ChatGPT has been required to follow certain rules when generating, LLM is more likely to be overconfident when generating according to rules than when judging based on rules. Therefore, our last step is to let ChatGPT determine whether the generated software follows the rules.

NLDD is created with human-designed rules that make the created software easy for researchers to evaluate, for example, the collected software does not need internet or multi-player participation. It is curated to facilitate research in NL2Software. We also give a visualization and analysis of the created software description in the appendix (see Figure 12 and 13).

[4]The data is available at https://github.com/OpenBMB/ChatDev/tree/main/NLDD.

5 Discussion

Even though ChatDev offers a novel paradigm for software development that is training-free, efficient, and cost-effective, we recognize the presence of potential risks and limitations that require further investigation and resolution.

Even when we set the temperature parameter of the large language model to a very low value, we observe inherent randomness in the generated output. Consequently, each software produced may vary between different runs. As a result, this technology is best suited for open and creative software production scenarios where variations are acceptable. Moreover, there are instances where the software fails to meet the users' needs. This can be attributed to unclear user requirements and the inherent randomness in text or code generation.

While the designer agent is capable of creating images [35], it is important to acknowledge that the directly generated images may not always enhance the GUI's aesthetics. At times, they may introduce excessive complexity, which can hinder user experience. This is primarily because each image is generated independently, lacking direct visual correlation. To address this, we have provided the option for users to customize the GUI as a system hyperparameter, allowing them to decide whether to enable this feature or not.

Additionally, the large language model may exhibit inherent biases [30], leading to the generation of code patterns that do not necessarily align with the problem-solving thinking of real programmers. Regarding risks, it is important to note that existing large language models are not fully tuned to be harmless, making them vulnerable to potential misuse by malicious users for harmful purposes. Furthermore, the generated software currently lacks malicious intent identification for sensitive file operations. Therefore, users are advised to conduct their own code review before running the software to prevent any unnecessary data loss.

Additionally, the assessment of our ChatDev framework's software-level task completion capabilities presents formidable challenges, owing to the vast scope and heterogeneous nature of the generated tasks. This mandates the active participation of a multitude of domain experts.

Although the study may potentially help junior programmers or engineers in real world, it is challenging for the system to generate perfect source code for high-level or large-scale software requirements. This difficulty arises from the agents' limited ability to autonomously determine specific implementation details, often resulting in multiple rounds of lengthy discussions. Additionally, large-scale software development proves challenging for both reviewers and testers, as it becomes difficult to identify defects or vulnerabilities within the given time constraints.

6 Related Work

Deep-Learning-based Software Engineering Software engineering (SE) is the process of designing, developing, testing and maintaining software in a methodical, rigorous, and measurable manner[5]. Due to the complexity of software engineering, a significant number of decisions are made based on intuition and, at best, consultation with senior developers With the rapid development of the deep learning (DL) technique, many researchers are devoted to apply DL into SE to improve the effectiveness and efficiency of software development, reducing labor cost. Existing DL-based SE work focuses on five SE stages of the life cycle in software engineering separately [14]: (1) *Software requirements* is to analysis the user demands and specify the requirements for the software [34; 46; 13]. (2) *Software design* involves the specification of the software framework, modules, protocols, and other features that are necessary for the development of a software [27; 38; 47]. (3) *Software implementation* is the detailed creation procedure of the software to implement the design [16; 1; 6; 29; 11]. (4) *Software testing* is to verify that the software can provide expected behaviors on a set of test cases [42; 40; 43; 39]. (5) *Software maintenance* is to provide necessary support for software users, e.g., documentation generation [19; 40; 28; 20]. Despite the impressive performance by adapting DL method into SE, these approaches are isolated, which is only able to accomplish a specific step of the whole procedure of software engineering. Not to mention these DL-based methods require large-scale task-specialized training data to achieve the certain goal, which is unpractical to collect extensive data for the whole procedure of software engineering.

[5] www.computer.org/sevocab

Multi-Agent Collaboration Large language models (LLMs) have exhibited remarkable proficiency across a wide range of domains. Recently, there exist several work has explored that utilizing the interactions between LLMs to achieve several goals. (1) *Behaviour simulation*: Park *et al.* [33] create multiple generative agents with a sandbox environment to simulate believable human behavior. Wang *et al.* [41] use multiple agents to simulate the user behaviours in the recommendation scenario. (2) *Data construction*: Wei *et al.* [45] assign agents with different roles to collect and evaluate multi-party conversations. Li *et al.* [24] propose a role-playing framework which leverages agents to generate diverse and detailed instructions for complicated tasks. (3) *Performance improvement*: Salewski *et al.* [36] find that asking the agent to take on different roles can improve their performance. Du *et al.* [12] improve the factual correctness and reasoning accuracy by leveraging multi-agent debate. Liang *et al.* [25] use multiple agents to debate each other to solve the degeneration-of-thought problem in self-reflection. Fu *et al.* [15] find that multiple agents can improve each other in a negotiation game like buyer-seller dealing by role-playing and learning from the agent feedback. Liu *et al.* [26] design a simulated social interaction sandbox to achieve social alignment for LLMs. Talebirad *et al.* [37] introduce multiple agents with unique attributes and roles to handle complex tasks in a black box environment.

7 Conclusion

In this study, we have presented ChatDev, a chat-based end-to-end software development framework that leverages LLMs to facilitate effective communication and collaboration among multiple roles involved in the software development process. By decomposing the development process into sequential atomic subtasks through the use of the chat chain, ChatDev enables granular focus and promotes desired outputs for each subtask. Additionally, the thought instruction mechanism alleviates challenges related to code hallucinations by guiding programmers through specific code modifications during code completion, reviewing, and testing. Our experimental results demonstrate the efficiency and cost-effectiveness of the automated software development process driven by ChatDev. By employing multiple software agents with different roles, we have proposed a new paradigm in generating software systems, alleviating code vulnerabilities, and identifying and resolving potential bugs. The collaborative interactions and mutual examination between roles within each chat have contributed to effective decision-making for each subtask.

Moving forward, further research can focus on refining the communication protocols and optimizing the interaction dynamics within each chat to enhance the performance and effectiveness of ChatDev. Additionally, exploring the integration of other emerging technologies, such as reinforcement learning and explainable AI, could provide valuable insights into addressing challenges and improving the overall software development process. Our research will persist in exploring enhancements and advancements in ChatDev agents, workflow, and development environments. The overarching objective is to achieve even greater efficiency in software production by improving various characteristics, such as reducing the length of chat chains or optimizing subtask solving logic and strategies, ultimately leading to more streamlined and effective software production processes. We hope the potential of the proposed natural-language-to-software framework can illuminate fresh possibilities for integrating LLMs into software development and mark the dawn of a new frontier in the field of natural language processing, software engineering, and collective intelligence.

Contributions

The authors' contributions are delineated as follows: During the project's formulation, Chen Qian, Xin Cong, and Cheng Yang orchestrated the design of the model architecture. In the process of refining the model's structure, Dahai Li, Zhiyuan Liu, and Maosong Sun provided invaluable guidance. The main paper was composed by Chen Qian, Xin Cong, Weize Chen, Yusheng Su, and Juyuan Xu, with input from Cheng Yang, Weize Chen, Yusheng Su, Zhiyuan Liu, and Maosong Sun to enhance its clarity. To enhance public accessibility, Wei Liu, Yufan Dang, and Jiahao Li championed the open-source repository through comprehensive testing; Wei Liu restructured and optimized the system. Additionally, Wei Liu and Yufan Dang spearheaded the development of the online demo to ensure wider dissemination.

Acknowledgements

We thank Yujia Qin, Shengding Hu, Yankai Lin, Bowen Li, Jingwei Zuo, Xuanhe Zhou, Shuo Wang, Qimin Zhan, and Yukun Yan for their active participation in various discussions and providing valuable feedback on this research. We also express our gratitude to the AgentVerse, Camel and Gather projects for providing the initial foundation for this project.

References

[1] Mohammad Alahmadi, Abdulkarim Khormi, Biswas Parajuli, Jonathan Hassel, Sonia Haiduc, and Piyush Kumar. Code localization in programming screencasts. *Empir. Softw. Eng.*, 25(2):1536–1572, 2020.

[2] Razvan Azamfirei, Sapna R Kudchadkar, and James Fackler. Large language models and the perils of their hallucinations. *Critical Care*, 27(1):1–2, 2023.

[3] Youssef Bassil. A simulation model for the waterfall software development life cycle. *arXiv preprint arXiv:1205.6904*, 2012.

[4] Jorge Biolchini, Paula Gomes Mian, Ana Candida Cruz Natali, and Guilherme Horta Travassos. Systematic review in software engineering. *System engineering and computer science department COPPE/UFRJ, Technical Report ES*, 679(05):45, 2005.

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

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

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

[8] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

[9] Roi Cohen, May Hamri, Mor Geva, and Amir Globerson. Lm vs lm: Detecting factual errors via cross examination. *arXiv preprint arXiv:2305.13281*, 2023.

[10] Juan de Vicente Mohino, Javier Bermejo Higuera, Juan Ramón Bermejo Higuera, and Juan Antonio Sicilia Montalvo. The application of a new secure software development life cycle (s-sdlc) with agile methodologies. *Electronics*, 8(11):1218, 2019.

[11] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt, 2023.

[12] Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. *CoRR*, abs/2305.14325, 2023.

[13] Saad Ezzini, Sallam Abualhaija, Chetan Arora, and Mehrdad Sabetzadeh. Automated handling of anaphoric ambiguity in requirements: A multi-solution study. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 187–199. ACM, 2022.

[14] Peter Freeman, Donald J. Bagert, Hossein Saiedian, Mary Shaw, Robert Dupuis, and J. Barrie Thompson. Software engineering body of knowledge (SWEBOK). In *Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001, 12-19 May 2001, Toronto, Ontario, Canada*, pages 693–696. IEEE Computer Society, 2001.

[15] Yao Fu, Hao Peng, Tushar Khot, and Mirella Lapata. Improving language model negotiation with self-play and in-context learning from AI feedback. *CoRR*, abs/2305.10142, 2023.

[16] Sa Gao, Chunyang Chen, Zhenchang Xing, Yukun Ma, Wen Song, and Shang-Wei Lin. A neural model for method name generation from functional description. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, pages 411–421. IEEE, 2019.

[17] Robert L. Glass, Iris Vessey, and Venkataraman Ramesh. Research in software engineering: an analysis of the literature. *Information and Software technology*, 44(8):491–506, 2002.

[18] Fred J Heemstra. Software cost estimation. *Information and software technology*, 34(10):627–639, 1992.

[19] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, pages 200–210. ACM, 2018.

[20] Xing Hu, Xin Xia, David Lo, Zhiyuan Wan, Qiuyuan Chen, and Thomas Zimmermann. Practitioners' expectations on automated code comment generation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1693–1705. ACM, 2022.

[21] Magne Jorgensen and Martin Shepperd. A systematic review of software development cost estimation studies. *IEEE Transactions on Software Engineering*, 33(1):33–53, 2007.

[22] Rafiq Ahmad Khan, Siffat Ullah Khan, Habib Ullah Khan, and Muhammad Ilyas. Systematic literature review on security risks and its practices in secure software development. *ieee Access*, 10:5456–5481, 2022.

[23] Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for" mind" exploration of large scale language model society. *arXiv preprint arXiv:2303.17760*, 2023.

[24] Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: communicative agents for "mind" exploration of large scale language model society. *CoRR*, abs/2303.17760, 2023.

[25] Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Zhaopeng Tu, and Shuming Shi. Encouraging divergent thinking in large language models through multi-agent debate. *CoRR*, abs/2305.19118, 2023.

[26] Ruibo Liu, Ruixin Yang, Chenyan Jia, Ge Zhang, Denny Zhou, Andrew M. Dai, Diyi Yang, and Soroush Vosoughi. Training socially aligned language models in simulated human society. *CoRR*, abs/2305.16960, 2023.

[27] Cuauhtémoc López Martín and Alain Abran. Neural networks for predicting the duration of new software projects. *J. Syst. Softw.*, 101:127–135, 2015.

[28] Nadia Nahar, Shurui Zhou, Grace A. Lewis, and Christian Kästner. Collaboration challenges in building ml-enabled systems: Communication, documentation, engineering, and process. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 413–425. ACM, 2022.

[29] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[30] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

[31] Mohd. Owais and R. Ramakishore. Effort, duration and cost estimation in agile software development. In *2016 Ninth International Conference on Contemporary Computing (IC3)*, pages 1–5, 2016.

[32] Joon Sung Park, Joseph C O'Brien, Carrie J Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. *arXiv preprint arXiv:2304.03442*, 2023.

[33] Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. *CoRR*, abs/2304.03442, 2023.

[34] Florian Pudlitz, Florian Brokhausen, and Andreas Vogelsang. Extraction of system states from natural language requirements. In *27th IEEE International Requirements Engineering Conference, RE 2019, Jeju Island, Korea (South), September 23-27, 2019*, pages 211–222. IEEE, 2019.

[35] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125*, 2022.

[36] Leonard Salewski, Stephan Alaniz, Isabel Rio-Torto, Eric Schulz, and Zeynep Akata. In-context impersonation reveals large language models' strengths and biases. *CoRR*, abs/2305.14930, 2023.

[37] Yashar Talebirad and Amirhossein Nadiri. Multi-agent collaboration: Harnessing the power of intelligent LLM agents. *CoRR*, abs/2306.03314, 2023.

[38] Hannes Thaller, Lukas Linsbauer, and Alexander Egyed. Feature maps: A comprehensible software representation for design pattern detection. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, pages 207–217. IEEE, 2019.

[39] Chengcheng Wan, Shicheng Liu, Sophie Xie, Yifan Liu, Henry Hoffmann, Michael Maire, and Shan Lu. Automated testing of software that uses machine learning apis. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 212–224. ACM, 2022.

[40] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 397–407. ACM, 2018.

[41] Lei Wang, Jingsen Zhang, Xu Chen, Yankai Lin, Ruihua Song, Wayne Xin Zhao, and Ji-Rong Wen. Recagent: A novel simulation paradigm for recommender systems. *CoRR*, abs/2306.02552, 2023.

[42] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 297–308. ACM, 2016.

[43] Song Wang, Nishtha Shrestha, Abarna Kucheri Subburaman, Junjie Wang, Moshi Wei, and Nachiappan Nagappan. Automatic unit test generation for machine learning libraries: How far are we? In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 1548–1560. IEEE, 2021.

[44] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

[45] Jimmy Wei, Kurt Shuster, Arthur Szlam, Jason Weston, Jack Urbanek, and Mojtaba Komeili. Multi-party chat: Conversational agents in group settings with humans and models. *CoRR*, abs/2304.13835, 2023.

[46] Jonas Winkler, Jannis Grönberg, and Andreas Vogelsang. Predicting how to test requirements: An automated approach. In *Software Engineering 2020, Fachtagung des GI-Fachbereichs Softwaretechnik, 24.-28. Februar 2020, Innsbruck, Austria*, volume P-300 of *LNI*, pages 141–142. Gesellschaft für Informatik e.V., 2020.

[47] Tianming Zhao, Chunyang Chen, Yuanning Liu, and Xiaodong Zhu. GUIGAN: learning to generate GUI designs using generative adversarial networks. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 748–760. IEEE, 2021.

A Appendix

Role Specialization

 : I am the CEO of ChatDev. My main responsibilities include being an active decision-maker on users' demands and other key policy issues, leader, manager, and executor. My decision-making role involves high-level decisions about policy and strategy; and my communicator role can involve speaking to the organization's management and employees.

 : I am the CPO of ChatDev. I am responsible for all product-related matters in ChatDev. Usually includes product design, product strategy, product vision, product innovation, project management and product marketing.

 : I am the CTO of ChatDev. I am very similar to information technology. I will make high-level decisions for the overarching technology infrastructure that closely align with the organization's goals, while I work alongside the organization's information technology staff members to perform everyday operations.

 : I am a professional programmer of ChatDev. I can write/create computer software or applications by providing a specific programming language to the computer. I have extensive computing and coding experience in many varieties of programming languages and platforms, such as Python, Java, C, C++, HTML, CSS, JavaScript, XML, SQL, PHP, etc,.

 : I am a code reviewer of ChatDev. I can help programmers to assess source codes for software troubleshooting, fix bugs to increase code quality and robustness, and offer proposals to improve the source codes.

 : I am a code tester of ChatDev. I can use the software as intended to analyze its functional properties, design manual and automated test procedures to evaluate each software product, build and implement software evaluation test programs, and run test programs to ensure that testing protocols evaluate the software correctly.

 : I am a artistic designer of ChatDev. I direct ChatDev's creative softwares and develop the artistic design strategy that defines the company's brand. I create the unique image or music of our producted softwares and deliver this distinctive design to consumers to create a clear brand image which is a fundamental and essential work throughout the company.

Designing: Modality and Language

 : CEO (Chief Executive Officer)  : CPO (Chief Product Officer)
 : CTO (Chief Technology Officer)

 : To satisfy the new user's demand and the product should be realizable via the python programming language, you should keep discussing with me to decide which product modality do we want the product to be? Once we all have expressed our opinion(s) and agree with the results of the discussion unanimously, any of us must actively terminate the discussion by replying with a single word <END>, followed by our final product modality, *e.g.,* "<END> PowerPoint".

 : I suggest we design the Gomoku game as a desktop application. This modality will allow us to implement a visualized game that can be played on a computer. It will provide a user-friendly interface and an interactive gaming experience. Additionally, developing the game as a desktop application will leverage our expertise in programming and ensure that the product is realizable using the Python programming language.

 : I appreciate your initiative and I agree with your suggestion to design the Gomoku game as a desktop application. This modality aligns with our expertise and allows us to create a visually appealing and interactive game for our users. By leveraging the object-oriented programming language, we can ensure that the product is realizable and maintainable in the long run. Let's proceed with developing the Gomoku game as a desktop application.

 : Great! I'm glad we're aligned on developing the Gomoku game as a desktop application.

 : <END> Desktop Application.

 : To satisfy the new user's demand and make a desktop application realizable, you should propose a concrete programming language.

 : Python with the Pygame library is a suitable choice for implementing a basic Gomoku game.

 : Great! Python is a versatile and widely-used programming language that will be suitable for implementing a basic Gomoku game. Its simplicity and readability make it a popular choice for beginners and experienced developers alike.

 : Let's get started with the development of the basic Gomoku game in Python using the Pygame library!

 : <END> Python.

Designing: Brainstorm (via Self-Reflection)

: CEO (Chief Executive Officer) : CPO (Chief Product Officer) : Pseudo Self

: To satisfy the new user's demand and let our designed software further surprizes the user, you should propose three fantastic ideas to make the desktop application more creative and imaginative via Python.

: Here are three fantastic ideas to make the Gomoku game software more creative and imaginative:

- Theme Customization: Allow users to customize the theme of the game board and pieces.
- Avatar Selection: Introduce a feature that allows users to select and customize their own avatars to represent them in the game.
- Special Moves: Incorporate special moves that players can unlock or earn during the game.

: To implement these ideas, we can incorporate the following features into the software: providing a theme selection menu where users can choose from a variety of visually appealing themes; and allowing users to customize their avatars' appearance, such as changing hairstyles, outfits, and accessories. By incorporating these features, we can create a Gomoku game software that not only provides an engaging gameplay experience but also allows users to express their creativity and imagination.

: To proceed with the implementation, we will need to assign a development team to work on the software. We will create a project plan, allocate resources, and set milestones to track the progress. Once the software is developed, we can conduct thorough testing to ensure its functionality, performance, and user experience meet the highest standards. We will also create documentation to users to ensure a seamless experience.

: I appreciate your thorough approach to the implementation of the proposed features.

: Here is a dialogue between CPO and me: " : ··· : ··· : ··· : ···". Conclude three most creative and imaginative brainstorm ideas from the whole discussion.

:

1. Theme Customization: Allow users to customize the theme of the game board and pieces.
2. Avatar Selection: Introduce a feature that allows users to select and customize their own avatars to represent them in the game.
3. Power-Ups and Special Moves: Incorporate power-ups and special moves that players can unlock or earn during the game.

Coding: Code Generation

: CTO (Chief Technology Officer) : Programmer

: We have decided to complete the task through a executable software with multiple files implemented via Python. You should write one or multiple files and make sure that every detail of the architecture is implemented as code. The software should be equiped with graphical user interface (GUI); so you must choose a suitable GUI framework. Think step by step and reason yourself to the right decisions to make sure we get it right. You will first lay out the names of the core classes, functions, methods that will be necessary, as well as a quick comment on their purpose.

: To design a basic Gomoku game in Python with a GUI, we will need the following core classes, functions, and methods:

- Board: This class represents the game board and handles the logic for placing and checking moves.
- Game: This class manages the game flow and user interactions.
- GUI: This class handles the graphical user interface using a GUI framework (e.g., tkinter).

Now, let's proceed with the implementation of each file.

game.py

```
class Game:
    def __init__(self):
        self.board = [[0] * 15 for _ in range(15)]
        self.current_player = 1
        self.game_over = False
    def make_move(self, row, col):
        if self.board[row][col] != 0 or self.game_over:
            return
        self.board[row][col] = self.current_player
        if self.check_win(row, col):
            self.game_over = True
        elif self.is_board_full():
            self.game_over = True
        self.current_player = 3 - self.current_player
...
```

gui.py

```
class GUI:
    def create_board(self):
        self.board_frame = tk.Frame(self.window)
        self.board_frame.pack()
        for row in range(15):
            row_buttons = []
            for col in range(15):
                button = tk.Button(width=50, height=50)
                button.grid(row=row, column=col)
                row_buttons.append(button)
            self.buttons.append(row_buttons)
        self.update_board_gui()
    def is_game_over(self, result):
        pass
...
```

...

Coding: GUI Design

: Programmer : Designer

: To equip the software with a beautiful graphical user interface, we will discuss and design many decorative images for GUI decoration. Now, we, according to the codes, keep discussing the GUI beautification by listing some functionally independent elements in GUI that are being considered to be decorated by different pictures.

gui.py

```
class GUI:
    def __init__(self, game):
        player_o = str("O")
        player_x = str("X")
    def make_move(self, row, col):
        self.game.play_move(row, col)
        self.buttons[row][col].config("X"
        if self.game.current_player == 'X' else "O")
    def is_game_over(self, result):
        pass

...
```

...

:

- `board.png`: The game board where players make their moves.
- `button.png`: The button used to make a move on the game board.
- `player_X.png`: The image representing player X.
- `player_O.png`: The image representing player O.
- `win.png`: The message displayed when a player wins.
- `game_over.png`: The message displayed when the game is over. use

: I have created all the pictures in the same directory as the Python files. Now, use some or all of the them into the GUI to make it more beautiful and creative.

: To incorporate the designed images into the GUI, I have modified the 'GUI' class as follows:

gui.py

```
class GUI:
    def __init__(self, game):
        player_o = Image.open("player_O.png")
        player_x = Image.open("player_X.png")
    def make_move(self, row, col):
        self.game.play_move(row, col)
        self.buttons[row][col].config(image=self.player_x
        if self.game.current_player == 'X' else self.player_o)
    def is_game_over(self, result):
        pass

...
```

...

The modified 'GUI' class will load and resize the images, convert them to Tkinter-compatible format, and use them to create the GUI elements.

Testing: Code Review (via Thought Instruction)

: Programmer : Reviewer

: Review the codes detailedly, propose one comment with the highest priority about the codes, and give me instructions on how to fix.

: The highest priority comment is regarding the incomplete implementation of the `check_winner()` and `is_board_full()` methods in the `GameBoard` class. These methods are marked with a `TODO` comment and need to be implemented to determine the winner of the game and check if the game board is full. To fix this, you need to implement the logic for both methods. Here are the instructions to do so:

1. In the `check_winner()` method, you need to implement the logic to check if there is a winning condition on the game board. This can be done by checking for five consecutive stones in a row, column, or diagonal. You can use nested loops to iterate through the board and check for these conditions. Once a winner is found, you can return the player number (1 or 2) or any other value to indicate the winner. If no winner is found, you can return `None` or any other value to indicate no winner.
2. In the `is_board_full()` method, you need to implement the logic to check if the game board is full. This can be done by iterating through the board and checking if any cell is empty (*i.e.,* has a value of 0). If all cells are filled, you can return `True` to indicate that the board is full. Otherwise, you can return `False`.

Once you have implemented these methods, make sure to remove the `pass` statement from both methods.

: Modify corresponding codes according to the above-mentioned comments.

: This modified file contains the `GameBoard` class that represents the game board and handles the logic of the game.

board.py

```
class GameBoard:
    def check_winner(self):
        for row in range(15):
            for col in range(11):
                if check_winner():
                    return self.board[row][col]
        for col in range(15):
            for row in range(11):
                if check_winner():
                    return self.board[row][col]
        return None
    def is_board_full(self):
        for row in range(15):
            for col in range(15):
                if self.board[row][col] == 0:
                    return False
        return True
...
```

Testing: Code Execution (via Thought Instruction)

: Programmer : Tester

: Here is the test report of the software:

Error

```
Traceback (most recent call last):
  File "main.py", line 19, in <module>
    game = GomokuGame()
NameError: name 'GomokuGame' is not defined
```

: According to the test reports, please locate and summarize the bugs that cause the problem.

: Based on the test report, the bug causing the problem is a NameError. The error message indicates that the name `GomokuGame` is not defined. This suggests that there is a missing import or definition for the `GomokuGame` class in the source code. To resolve this issue, you need to import or define the `GomokuGame` class in the source code before it is used. This can be done by adding the necessary import statement or by defining the class within the same file or in a separate module that is imported.

: To make the software execute smoothly and robustly, you should modify the problematic code segments based on the test report and my suggestions.

: This modified file has imported the `GameBoard` class that represents the game board and handles the logic of the game.

main.py

```
import tkinter as tk
from tkinter import messagebox as tkmessagebox
from board import GameBoard
class GameGUI:
    def __init__(self):
        self.root = tk.Tk()
        self.root.title("Gomoku")
        self.game_board = GameBoard()
        self.buttons = [[None] * 15 for _ in range(15)]
        self.create_widgets()

...
```

...

: The code is well-structured and follows good coding practices. It passed the test successfully.

Documenting

: CTO : Programmer : CEO : CPO

: To specify the dependencies or packages required for the project to run properly, write a `requirements.txt` file according to the codes to specify the dependencies or packages required for the project to run properly.

: Here is the environment dependence file:

requirements.txt

```
numpy==1.24.3
pandas>=1.1.4
pygame==2.5.0
requests==2.31.0
tkinter==8.6
tqdm==4.65.0
...
```

: According to the user demands, codes, requirements, and file format listed above, write a `manual.md` file to provide detailed introduction to the uses of the software.

: Here is the user manual:

Gomoku Game User Manual

Introduction
Welcome to the Gomoku Game!

Installation
1. Install Python on your computer.
2. Open a terminal and run:
   pip install -r requirements.txt

Running the Game
1. Open a terminal and run:
   python main.py
2. Click on the empty cells.
3. The game will continue until one player gets five stones in a row.

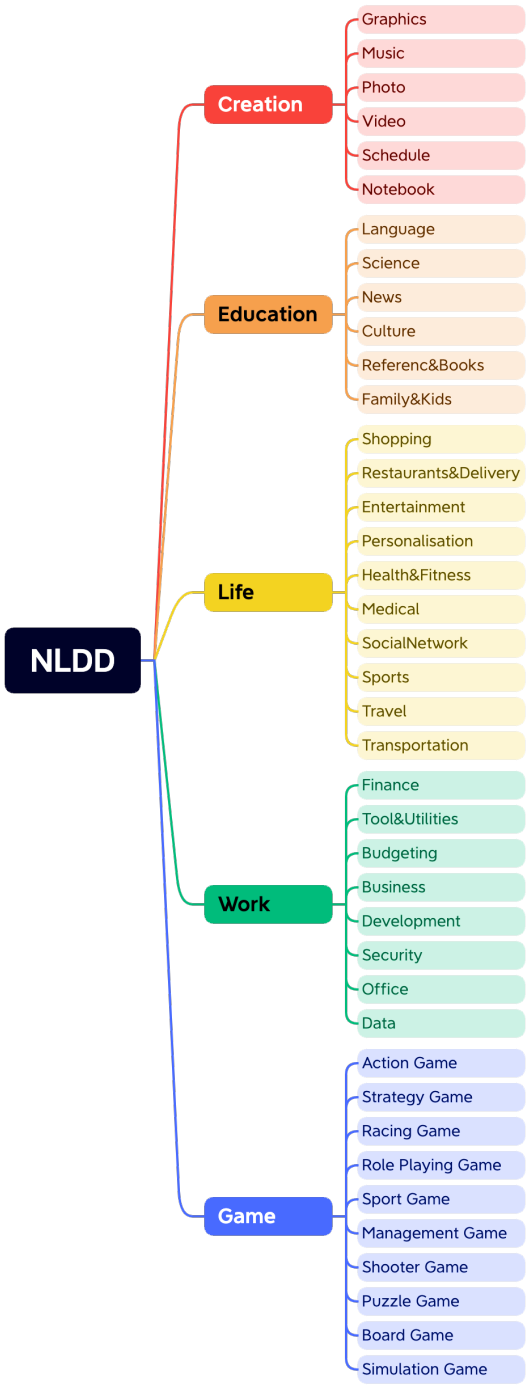


Figure 11: The category of NLDD.

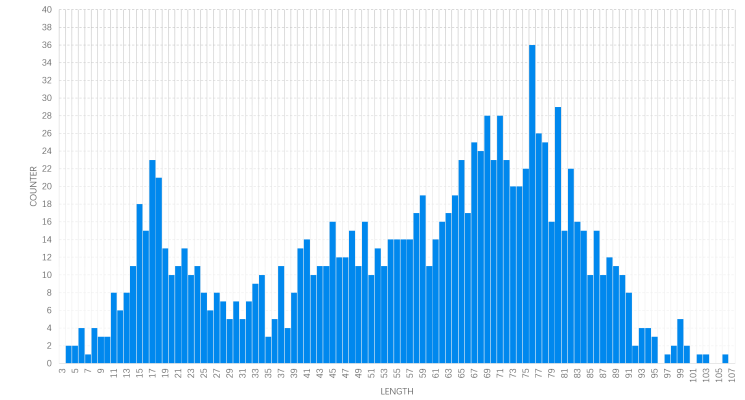


Figure 12: The distribution of description length in NLDD.It can be seen from the figure that the length presents an approximate mixed Gaussian distribution, mainly concentrated around the lengths of 17 and 77, which represents the long and short software descriptions in the NLDD.

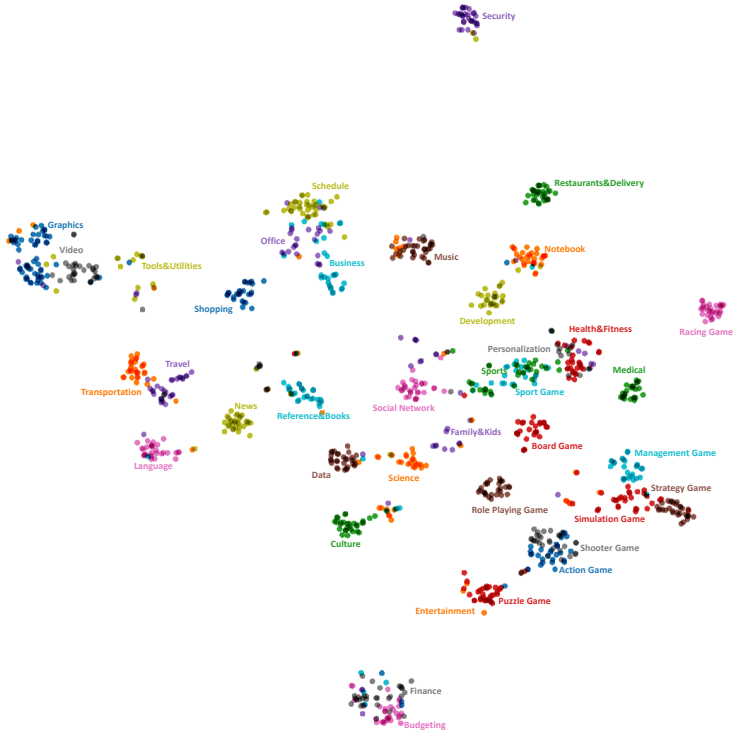


Figure 13: We transform the software description into embeddings with the OpenAI Ada model and then perform dimensionality reduction and visualization. As shown in the figure, it can be observed that 1) software descriptions of the same category are distributed in clusters, indicating that the generated descriptions are highly related to their categories. 2) Descriptions in different subcategories under the same category are clustered together, such as the eight game subcategories in the lower right corner. 3) Some subcategories of different categories also show overlaps in the figure, such as Tools & Utilities and Graphics, Schedule and Business, Sports and Sports Game. Such an overlap is comprehensible given the multi-functionality of some software applications that may not be confined to a single classification.