ChatDev: Communicative Agents for Software Development

Chen Qian[★] **Wei Liu**[★] **Hongzhang Liu**[♠] **Nuo Chen**[★] **Yufan Dang**[★]
Jiahao Li[★] **Cheng Yang**[♣] **Weize Chen**[★] **Yusheng Su**[★] **Xin Cong**[★]
Juyuan Xu[★] **Dahai Li**[♦] **Zhiyuan Liu**[★✉] **Maosong Sun**[★✉]
[★]Tsinghua University [♠]The University of Sydney [♣]BUPT [♦]Modelbest Inc.
qianc62@gmail.com liuzy@tsinghua.edu.cn sms@tsinghua.edu.cn

Abstract

Software development is a complex task that necessitates cooperation among multiple members with diverse skills. Numerous studies used deep learning to improve specific phases in a waterfall model, such as design, coding, and testing. However, the deep learning model in each phase requires unique designs, leading to technical inconsistencies across various phases, which results in a fragmented and ineffective development process. In this paper, we introduce ChatDev, a chat-powered software development framework in which specialized agents driven by large language models (LLMs) are guided in what to communicate (via *chat chain*) and how to communicate (via *communicative dehallucination*). These agents actively contribute to the design, coding, and testing phases through unified language-based communication, with solutions derived from their multi-turn dialogues. We found their utilization of natural language is advantageous for system design, and communicating in programming language proves helpful in debugging. This paradigm demonstrates how linguistic communication facilitates multi-agent collaboration, establishing language as a unifying bridge for autonomous task-solving among LLM agents. The code and data are available at https://github.com/OpenBMB/ChatDev.

1 Introduction

Large language models (LLMs) have led to substantial transformations due to their ability to effortlessly integrate extensive knowledge expressed in language (Brown et al., 2020; Bubeck et al., 2023), combined with their strong capacity for role-playing within designated roles (Park et al., 2023; Hua et al., 2023; Chen et al., 2023b). This advancement eliminates the need for model-specific designs and delivers impressive performance in diverse downstream applications. Furthermore, autonomous agents (Richards, 2023; Zhou et al., 2023a) have gained attention for enhancing the capabilities of LLMs with advanced features such as context-aware memory (Sumers et al., 2023), multi-step planning (Liu et al., 2023), and strategic tool using (Schick et al., 2023).

Figure 1: ChatDev, a chat-powered software development framework, integrates LLM agents with various social roles, working autonomously to develop comprehensive solutions via multi-agent collaboration.

Software development is a complex task that necessitates cooperation among multiple members with diverse skills (*e.g.,* architects, programmers, and testers) (Basili, 1989; Sawyer and Guinan, 1998). This entails extensive communication among different roles to understand and analyze requirements through natural language, while also encompassing development and debugging using programming languages (Ernst, 2017; Banker et al., 1998). Numerous studies use deep learning to improve specific phases of the waterfall model in software development, such as design, coding, and testing (Pudlitz et al., 2019; Martín and Abran, 2015;

✉: Corresponding Author.

Gao et al., 2019; Wang et al., 2016). Due to these technical inconsistencies, methods employed in different phases remain isolated until now. Every phase, from data collection and labeling to model training and inference, requires its unique designs, leading to a fragmented and less efficient development process in the field (Freeman et al., 2001; Ernst, 2017; Winkler et al., 2020).

Motivated by the expert-like potential of autonomous agents, we aim to establish language as a unifying bridge—utilizing multiple LLM-powered agents with specialized roles for cooperative software development through language-based communication across different phases; solutions in different phases are derived from their multi-turn dialogues, whether dealing with text or code. Nevertheless, due to the tendency of LLM hallucinations (Dhuliawala et al., 2023; Zhang et al., 2023b), the strategy of generating software through communicative agents could lead to the non-trivial challenge of *coding hallucinations*, which involves the generation of source code that is incomplete, unexecutable, or inaccurate, ultimately failing to fulfill the intended requirements (Agnihotri and Chug, 2020). The frequent occurrence of coding hallucination in turn reflects the constrained autonomy of agents in task completion, inevitably demanding additional manual intervention and thereby hindering the immediate usability and reliability of the generated software (Ji et al., 2023).

In this paper, we propose ChatDev (see Figure 1), a chat-powered software-development framework integrating multiple "software agents" for active involvement in three core phases of the software lifecycle: design, coding, and testing. Technically, ChatDev uses a *chat chain* to divide each phase into smaller subtasks further, enabling agents' multi-turn communications to cooperatively propose and develop solutions (*e.g.,* creative ideas or source code). The chain-structured workflow guides agents on what to communicate, fostering cooperation and smoothly linking natural- and programming-language subtasks to propel problem-solving. Additionally, to minimize coding hallucinations, ChatDev includes an *communicative dehallucination* mechanism, enabling agents to actively request more specific details before giving direct responses. The communication pattern instructs agents on how to communicate, enabling precise information exchange for effective solution optimization while reducing coding hallucinations. We built a comprehensive dataset containing software requirement descriptions and conducted comprehensive analyses. The results indicate that ChatDev notably improves the quality of software, leading to improved completeness, executability, and better consistency with requirements. Further investigations reveal that natural-language communications contribute to comprehensive system design, while programming-language communications drive software optimization. In summary, the proposed paradigm demonstrates how linguistic communication facilitates multi-agent collaboration, establishing language as a unifying bridge for autonomous task-solving among LLM agents.

2 Related Work

Trained on vast datasets to comprehend and manipulate billions of parameters, LLMs have become pivotal in natural language processing due to their seamless integration of extensive knowledge (Brown et al., 2020; Bubeck et al., 2023; Vaswani et al., 2017; Radford et al.; Touvron et al., 2023; Wei et al., 2022a; Shanahan et al., 2023; Chen et al., 2021; Brants et al., 2007; Chen et al., 2021; Ouyang et al., 2022; Yang et al., 2023a; Qin et al., 2023b; Kaplan et al., 2020). Furthermore, LLMs have demonstrated strong role-playing abilities (Li et al., 2023a; Park et al., 2023; Hua et al., 2023; Chan et al., 2023; Zhou et al., 2023b; Chen et al., 2023b,a; Cohen et al., 2023; Li et al., 2023b). Recent progress, particularly in the field of autonomous agents (Zhou et al., 2023a; Wang et al., 2023a; Park et al., 2023; Wang et al., 2023e; Richards, 2023; Osika, 2023; Wang et al., 2023d), is largely attributed to the foundational advances in LLMs. These agents utilize the robust capabilities of LLMs, displaying remarkable skills in memory (Park et al., 2023; Sumers et al., 2023), planning (Chen et al., 2023b; Liu et al., 2023) and tool use (Schick et al., 2023; Cai et al., 2023; Qin et al., 2023a; Ruan et al., 2023; Yang et al., 2023b), enabling them to reason in complex scenarios (Wei et al., 2022b; Zhao et al., 2023; Zhou et al., 2023a; Ma et al., 2023; Zhang et al., 2023a; Wang et al., 2023b; Ding et al., 2023; Weng, 2023).

Software development is a multifaceted and intricate process that requires the cooperation of multiple experts from various fields (Yilmaz et al., 2012; Acuna et al., 2006; Basili, 1989; Sawyer and Guinan, 1998; Banker et al., 1998; France and Rumpe, 2007), encompassing the requirement analysis and system design in natural lan-

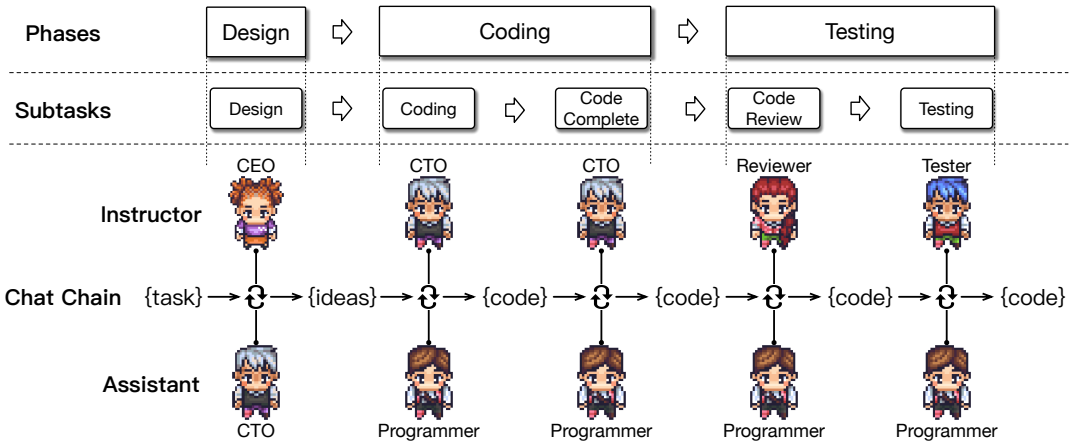

Figure 2: Upon receiving a preliminary task requirement (*e.g.,* "*develop a Gomoku game*"), these software agents engage in multi-turn communication and perform instruction-following along a chain-structured workflow, collaborating to execute a series of subtasks autonomously to craft a comprehensive solution.

guages (Pudlitz et al., 2019; Martín and Abran, 2015; Nahar et al., 2022), along with system development and debugging in programming languages (Gao et al., 2019; Wang et al., 2016; Wan et al., 2022). Numerous studies employ the waterfall model, a particular software development life cycle, to segment the process into discrete phases (*e.g.,* design, coding, testing) and apply deep learning to improve the effectiveness of certain phases (Winkler et al., 2020; Ezzini et al., 2022; Thaller et al., 2019; Zhao et al., 2021; Nijkamp et al., 2023; Wan et al., 2018; Wang et al., 2021).

3 ChatDev

We introduce ChatDev, a chat-powered software-development framework that integrates multiple "software agents" with various social roles (*e.g.,* requirements analysts, professional programmers and test engineers) collaborating in the core phases of the software life cycle, see Figure 1. Technically, to facilitate cooperative communication, ChatDev introduces *chat chain* to further break down each phase into smaller and manageable subtasks, which guides multi-turn communications between different roles to propose and validate solutions for each subtask. In addition, to alleviate unexpected hallucinations, a communicative pattern named *communicative dehallucination* is devised, wherein agents request more detailed information before responding directly and then continue the next round of communication based on these details.

3.1 Chat Chain

Although LLMs show a good understanding of natural and programming languages, efficiently transforming textual requirements into functional software in a single step remains a significant challenge. ChatDev thus adopts the core principles of the waterfall model, using a chat chain ($\mathcal{C}$) with sequential phases ($\mathcal{P}$), each comprising sequential subtasks ($\mathcal{T}$). Specifically, ChatDev segments the software development process into three sequential phases: design, coding, and testing. The coding phase is further subdivided into subtasks of code writing and completion, and the testing phase is segmented into code review (static testing) and system testing (dynamic testing), as illustrated in Figure 2. In every subtask, two agents, each with their own specialized roles (*e.g.,* a reviewer skilled at identifying endless loops and a programmer adept in GUI design), perform the functions of an instructor ($\mathcal{I}$) and an assistant ($\mathcal{A}$). The instructor agent initiates instructions, instructing ($\rightarrow$) the discourse toward the completion of the subtask, while the assistant agent adheres to these instructions and responds with ($\rightsquigarrow$) appropriate solutions. They engage in a multi-turn dialogue (C), working cooperatively until they achieve consensus, extracting (τ) solutions that can range from the text (*e.g.,* defining a software function point) to code (*e.g.,* creating the initial version of source code), ultimately leading to the completion of the subtask. The entire task-solving process along the agentic workflow can be formulated as:

$$\begin{aligned}
\mathcal{C} &= \langle \mathcal{P}^1, \mathcal{P}^2, \ldots, \mathcal{P}^{|\mathcal{C}|} \rangle \\
\mathcal{P}^i &= \langle \mathcal{T}^1, \mathcal{T}^2, \ldots, \mathcal{T}^{|\mathcal{P}^i|} \rangle \\
\mathcal{T}^j &= \tau\big(\mathsf{C}(\mathcal{I}, \mathcal{A})\big) \\
\mathsf{C}(\mathcal{I}, \mathcal{A}) &= \langle \mathcal{I} \rightarrow \mathcal{A},\ \mathcal{A} \rightsquigarrow \mathcal{I} \rangle_{\circlearrowleft}
\end{aligned} \tag{1}$$

The dual-agent communication design simplifies communications by avoiding complex multi-agent topologies, effectively streamlining the consensus-reaching process (Yin et al., 2023; Chen et al., 2023b). Subsequently, the solutions from previous tasks serve as bridges to the next phase, allowing a smooth transition between subtasks. This approach continues until all subtasks are completed. It's worth noting that the conceptually simple but empirically powerful chain-style structure guides agents on what to communicate, fostering cooperation and smoothly linking natural- and programming-language subtasks. It also offers a transparent view of the entire software development process, allowing for the examination of intermediate solutions and assisting in identifying possible problems.

Agentization To enhance the quality and reduce human intervention, ChatDev implements prompt engineering that only takes place at the start of each subtask round. As soon as the communication phase begins, the instructor and the assistant will communicate with each other in an automated loop, continuing this exchange until the task concludes. However, simply exchanging responses cannot achieve effective multi-round task-oriented communication, since it inevitably faces significant challenges including role flipping, instruction repeating, and fake replies. As a result, there is a failure to advance the progression of productive communications and hinders the achievement of meaningful solutions. ChatDev thus employs inception prompting mechanism (Li et al., 2023a) for initiating, sustaining, and concluding agents' communication to guarantee a robust and efficient workflow. This mechanism is composed of the instructor system prompt P_I and the assistant system prompt P_A. The system prompts for both roles are mostly symmetrical, covering the overview and objectives of the current subtask, specialized roles, accessible external tools, communication protocols, termination conditions, and constraints or requirements to avoid undesirable behaviors. Then, an instructor $\mathcal{I}$ and an assistant $\mathcal{A}$ are instantiated by hypnotizing LLMs via P_I and P_A:

$$\mathcal{I} = \rho(LLM, \mathsf{P}_I), \;\; \mathcal{A} = \rho(LLM, \mathsf{P}_A) \quad (2)$$

where ρ is the role customization operation, implemented via system message assignment.

Memory Note that the limited context length of common LLMs typically restricts the ability to maintain a complete communication history among all agents and phases. To tackle this issue, based on the nature of the chat chain, we accordingly segment the agents' context memories based on their sequential phases, resulting in two functionally distinct types of memory: *short-term memory* and *long-term memory*. Short-term memory is utilized to sustain the continuity of the dialogue within a single phase, while long-term memory is leveraged to preserve contextual awareness across different phases.

Formally, short-term memory records an agent's current phase utterances, aiding context-aware decision-making. At the time t during phase $\mathcal{P}^i$, we use $\mathcal{I}_t^i$ to represent the instructor's instruction and $\mathcal{A}_t^i$ for the assistant's response. The short-term memory $\mathcal{M}$ collects utterances up to time t as:

$$\mathcal{M}_t^i = \langle (\mathcal{I}_1^i, \mathcal{A}_1^i), (\mathcal{I}_2^i, \mathcal{A}_2^i), \ldots, (\mathcal{I}_t^i, \mathcal{A}_t^i) \rangle \quad (3)$$

In the next time step $t+1$, the instructor utilizes the current memory to generate a new instruction $\mathcal{I}_{t+1}^i$, which is then conveyed to the assistant to produce a new response $\mathcal{A}_{t+1}^i$. The short-term memory iteratively updates until the number of communications reaches the upper limit $|\mathcal{M}^i|$:

$$\begin{aligned} \mathcal{I}_{t+1}^i = & \mathcal{I}(\mathcal{M}_t^i), \;\; \mathcal{A}_{t+1}^i = \mathcal{A}(\mathcal{M}_t^i, \mathcal{I}_{t+1}^i) \\ & \mathcal{M}_{t+1}^i = \mathcal{M}_t^i \cup (\mathcal{I}_{t+1}^i, \mathcal{A}_{t+1}^i) \end{aligned} \quad (4)$$

To perceive dialogues through previous phases, the chat chain only transmits the solutions from previous phases as long-term memories $\tilde{\mathcal{M}}$, integrating them at the start of the next phase and enabling the cross-phase transmission of long dialogues:

$$\mathcal{I}_1^{i+1} = \tilde{\mathcal{M}}^i \cup \mathsf{P}_{\mathcal{I}}^{i+1}, \;\; \tilde{\mathcal{M}}^i = \bigcup_{j=1}^{i} \tau(\mathcal{M}_{|\mathcal{M}^j|}^j) \quad (5)$$

where P symbolizes a predetermined prompt that appears exclusively at the start of each phase.

By sharing only the solutions of each subtask rather than the entire communication history, ChatDev minimizes the risk of being overwhelmed by too much information, enhancing concentration on each task and encouraging more targeted cooperation, while simultaneously facilitating cross-phase context continuity.

3.2 Communicative Dehallucination

LLM hallucinations manifest when models generate outputs that are nonsensical, factually incorrect, or inaccurate (Dhuliawala et al., 2023; Zhang

et al., 2023b). This issue is particularly concerning in software development, where programming languages demand precise syntax—the absence of even a single line can lead to system failure. We have observed that LLMs often produce *coding hallucinations*, which encompass potential issues like incomplete implementations, unexecutable code, and inconsistencies that don't meet requirements. Coding hallucinations frequently appear when the assistant struggles to precisely follow instructions, often due to the vagueness and generality of certain instructions that require multiple adjustments, making it challenging for agents to achieve full compliance. Inspired by this, we introduce *communicative dehallucination*, which encourages the assistant to actively seek more detailed suggestions from the instructor before delivering a formal response.

Specifically, a vanilla communication pattern between the assistant and the instructor follows a straightforward instruction-response format:

$$\langle \mathcal{I} \rightarrow \mathcal{A},\ \mathcal{A} \rightsquigarrow \mathcal{I} \rangle_{\circlearrowleft} \tag{6}$$

In contrast, our communicative dehallucination mechanism features a deliberate "role reversal", where the assistant takes on an instructor-like role, proactively seeking more specific information (*e.g.,* the precise name of an external dependency and its related class) before delivering a conclusive response. After the instructor provides a specific modification suggestion, the assistant proceeds to perform precise optimization:

$$\langle \mathcal{I} \rightarrow \mathcal{A},\ \langle \mathcal{A} \rightarrow \mathcal{I},\ \mathcal{I} \rightsquigarrow \mathcal{A} \rangle_{\circlearrowleft},\ \mathcal{A} \rightsquigarrow \mathcal{I} \rangle_{\circlearrowleft} \tag{7}$$

Since this mechanism tackles one concrete issue at a time, it requires multiple rounds of communication to optimize various potential problems. The communication pattern instructs agents on how to communicate, enabling finer-grained information exchange for effective solution optimization, which practically aids in reducing coding hallucinations.

4 Evaluation

Baselines We chose some representative LLM-based software development methods as our baselines. GPT-Engineer (Osika, 2023) is a fundamental single-agent approach in LLM-driven software agents with a precise understanding of task requirements and the application of one-step reasoning, which highlights its efficiency in generating detailed software solutions at the repository level. MetaGPT (Hong et al., 2023) is an advanced framework that allocates specific roles to various LLM-driven software agents and incorporates standardized operating procedures to enable multi-agent participation. In each step agents with specific roles generate solutions by adhering to static instructions predefined by human experts.

Datasets Note that, as of now, there isn't a publicly accessible dataset containing textual descriptions of software requirements in the context of agent-driven software development. To this end, we are actively working towards developing a comprehensive dataset for software requirement descriptions, which we refer to as SRDD (Software Requirement Description Dataset). Drawing on previous work (Li et al., 2023a), we utilize existing software descriptions as initial examples, which are then further developed through a process that combines LLM-based automatic generation with post-processing refinement guided by humans. As a result, this dataset includes important software categories from popular platforms such as Ubuntu, Google Play, Microsoft Store, and Apple Store. It comprises 1,200 software task prompts that have been carefully categorized into 5 main areas: Education, Work, Life, Game, and Creation. All these areas are further divided into 40 subcategories, and each subcategory contains 30 unique task prompts.

Metrics Evaluating software is also a challenging task, especially when trying to assess it on a holistic level. Under the current limitation of scarce benchmark resources, traditional function-oriented code generation metrics (*e.g.,* `pass@k`), cannot seamlessly transfer to a comprehensive evaluation of entire software systems. The main reason for this is that it is often impractical to develop manual or automated test cases for various types of software, especially those involving complex interfaces, frequent user interactions, or non-deterministic feedback. As an initial strategy, we apply three fundamental and objective dimensions that reflect different aspects of coding hallucinations to evaluate the agent-generated software, and then integrate them to facilitate a more holistic evaluation:

- *Completeness* measures the software's ability to fulfill code completion in software development, quantified as the percentage of software without any "placeholder" code snippets. A higher score indicates a higher probability of automated completion.

Method	Paradigm	Completeness	Executability	Consistency	Quality
GPT-Engineer	🤖	0.5022†	0.3583†	0.7887†	0.1419†
MetaGPT	🤖🤖	0.4834†	0.4145†	0.7601†	0.1523†
ChatDev	🤖🤖	**0.5600**	**0.8800**	**0.8021**	**0.3953**

Table 1: Overall performance of the LLM-powered software development methods, encompassing both single-agent (🤖) and multi-agent (🤖🤖) paradigms. Performance metrics are averaged for all tasks. The top scores are in bold, with second-highest underlined. † indicates significant statistical differences ($p \leq 0.05$) between a baseline and ours.

Method	Evaluator	Baseline Wins	ChatDev Wins	Draw
GPT-Engineer	GPT-4	22.50%	77.08%	00.42%
	Human	09.18%	90.16%	00.66%
MetaGPT	GPT-4	37.50%	57.08%	05.42%
	Human	07.92%	88.00%	04.08%

Table 2: Pairwise evaluation results.

Method	Duration (s)	#Tokens	#Files	#Lines
GPT-Engineer	15.6000	7,182.5333	3.9475	70.2041
MetaGPT	154.0000	29,278.6510	4.4233	153.3000
ChatDev	148.2148	22,949.4450	4.3900	144.3450

Table 3: Software statistics include Duration (time consumed), #Tokens (number of tokens used), #Files (number of code files generated), and #Lines (total lines of code across all files) in the software generation process.

- *Executability* assesses the software's ability to run correctly within a compilation environment, quantified as the percentage of software that compiles successfully and can run directly. A higher score indicates a higher probability of successful execution.
- *Consistency* measures how closely the generated software code aligns with the original requirement description, quantified as the cosine distance between the semantic embeddings of the textual requirements and the generated software code[1]. A higher score indicates a greater degree of consistency with the requirements.
- *Quality* is a comprehensive metric that integrates various factors to assess the overall quality of software, quantified by multiplying[2] completeness, executability, and consistency. A higher quality score suggests a higher overall satisfaction with the software generated, implying a lower need for further manual intervention.

Implementation Details We divided software development into 5 subtasks within 3 phases, assigning specific roles like CEO, CTO, programmer, reviewer, and tester. A subtask would terminate and get a conclusion either after two unchanged code modifications or after 10 rounds of communication. During the code completion, review, and testing, a communicative dehallucination is activated. For ease of identifying solutions, the assistant begins responses with "<SOLUTION>" when a consensus is reached. We used ChatGPT-3.5 with a temperature of 0.2 and integrated Python-3.11.4 for feedback. All baselines in the evaluation share the same hyperparameters and settings for fairness.

[1]Comments should be excluded from the code to avoid potential information leakage during evaluations.

[2]One can also choose to average the sub-metrics, which yields similar trends.

4.1 Overall Performance

As illustrated in Table 1, ChatDev outperforms all baseline methods across all metrics, showing a considerable margin of improvement. Firstly, the improvement of ChatDev and MetaGPT over GPT-Engineer demonstrates that complex tasks are difficult to solve in a single-step solution. Therefore, explicitly decomposing the difficult problem into several smaller, more manageable subtasks enhances the effectiveness of task completion. Additionally, in comparison to MetaGPT, ChatDev significantly raises the *Quality* from 0.1523 to 0.3953. This advancement is largely attributed to the agents employing a cooperative communication method, which involves autonomously proposing and continuously refining source code through a blend of natural and programming languages, as opposed to merely delivering responses based on human-predefined instructions. The communicative agents guide each subtask towards integrated and automated solutions, efficiently overcoming the restrictions typically linked to manually established optimization rules, and offering a more versatile and adaptable framework for problem-solving.

To further understand user preferences in practical settings, we use the setting adopted by Li et al. (2023a), where agent-generated solutions are com-

Variant	Completeness	Executability	Consistency	Quality
ChatDev	0.5600	**0.8800**	**0.8021**	**0.3953**
≤Coding	0.4100	0.7700	0.7958	0.2512
≤Complete	**0.6250**	0.7400	0.7978	0.3690
≤Review	0.5750	0.8100	0.7980	0.3717
≤Testing	0.5600	**0.8800**	**0.8021**	**0.3953**
╲CDH	0.4700	0.8400	0.7983	0.3094
╲Roles	0.5400	0.5800	0.7385	0.2212

Table 4: Ablation study on main components or mechanisms. $\leq x$ denotes halting the chat chain after the completion of the x phrase, and ╲ denotes the removing operation. CDH denotes the communicative dehallucination mechanism.

pared in pairs by both human participants and the prevalent GPT-4 model to identify the preferred one.[3] Table 2 shows ChatDev consistently outperforming other baselines, with higher average win rates in both GPT-4 and human evaluations.

Furthermore, the software statistics presented in Table 3 indicates that the multi-agent paradigm, despite being slower and consuming more tokens than the single-agent method, yields a greater number of code files and a larger codebase, which may enhance the software's functionality and integrity. Analyzing the dialogues of agents suggests that the multi-agent communication method often leads agents to autonomously offer functional enhancements (*e.g.,* GUI creation or increasing game difficulty), thereby potentially resulting in the incorporation of beneficial features that were not explicitly specified in requirements. Taking all these factors together, we posit that the fundamental characteristics of multi-agent software development take on greater significance, surpassing short-term concerns like time and economic costs in the current landscape.

4.2 Ablation Study

This section examines key components or mechanisms within our multi-agent cooperation framework by removing particular phases in the chat chain, communicative dehallucination, or the roles assigned to all agents in their system prompts. Figure 4 shows that the code complete phase enhances *Completeness*, with testing critical for *Executability*. *Quality* steadily rises with each step, suggesting that software development optimization is progressively attained through multi-phase communications among intelligent agents. Meanwhile, eliminating communicative dehallucination results in a decrease across all metrics, indicating its effectiveness in addressing coding hallucinations. Most interestingly, the most substantial impact on performance occurs when the roles of all agents are removed from their system prompts. Detailed dialogue analysis shows that assigning a "prefer GUI design" role to a programmer results in generated source code with relevant GUI implementations; in the absence of such role indications, it defaults to implement unfriend command-line-only programs only. Likewise, assigning roles such as a "careful reviewer for bug detection" enhances the chances of discovering code vulnerabilities; without such roles, feedback tends to be high-level, leading to limited adjustments by the programmer. This finding underscores the importance of assigning roles in eliciting responses from LLMs, underscoring the significant influence of multi-agent cooperation on software quality.

[3]For fairness, GPT-4's evaluation mitigated possible positional bias (Wang et al., 2023c), and human experts independently assessed the task solutions, randomized to prevent order bias.

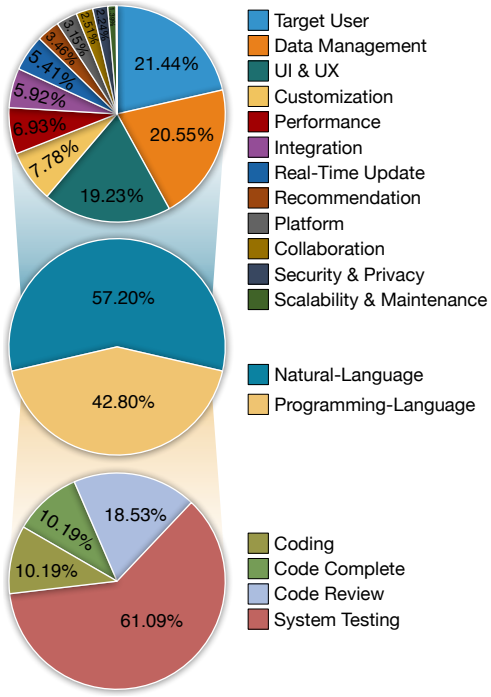

Figure 3: The utterance distribution of agent communications throughout the entire development process.

4.3 Communication Analysis

Our agent-driven software development paradigm promotes cooperative agents through effective communication for automated solution optimization. Phases in the chat chain have varying levels of engagement in natural and programming languages.

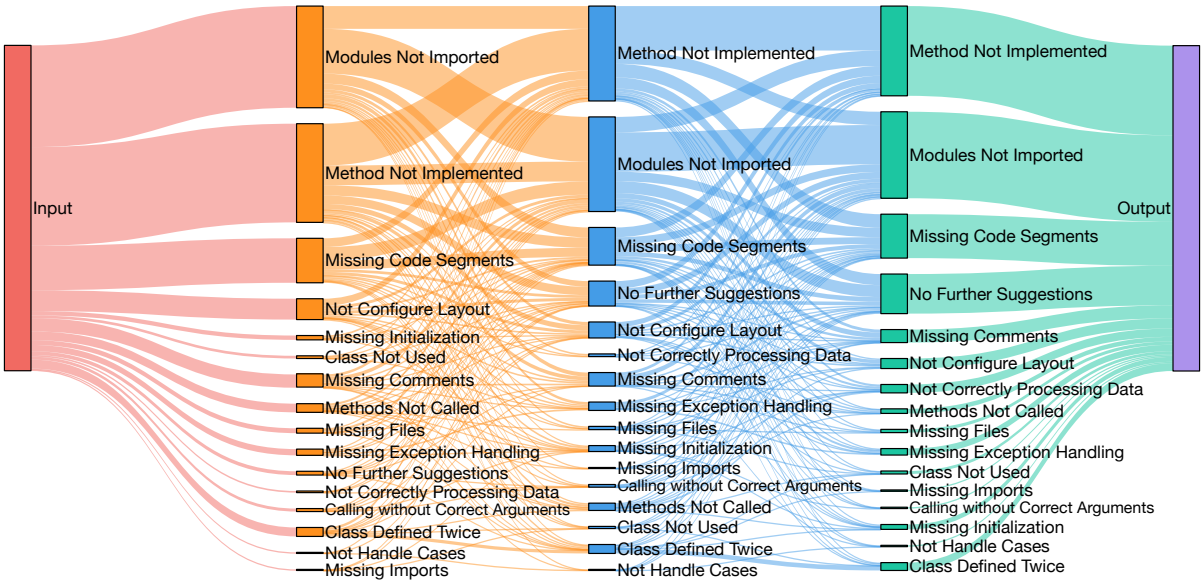

Figure 4: The chart demonstrates the distribution of suggestions made by a reviewer agent during a multi-round reviewing process, where each sector in the chart represents a different category of suggestion.

We now analyze the content of their communications to understand linguistic effects.

Figure 3 depicts a communication breakdown, with natural language at 57.20%. In the natural-language phase (*i.e.,* design), natural language communication plays a crucial role in the thorough design of the system, with agents autonomously discussing and designing aspects like target user, data management, and user interface. Post-design phases show a balanced mix of coding, code completion, and testing activities, with most communication occurring during code reviews. This trend is due to agents' self-reviews and code fixes consistently propelling software development; otherwise, progress halts when successive updates don't show significant changes, leading to a natural decrease in code review communications.

We explore the properties of static debugging dynamics in code reviews resulting from communication between reviewers and programmers, as depicted in Figure 4. The data uncovers that during the review phase, reviewers may spot different issues through language interactions. The programmer's intervention can transform certain issues into different ones or a state where no further suggestions are needed; the increasing proportion of the latter indicates successful software optimization. Particularly, the "`Method Not Implemented`" issue is most common in communication between reviewers and programmers during code reviews, accounting for 34.85% of discussions. This problem usually arises from unclear text requirements and the use of "placeholder" tags in Python code, necessitating additional manual adjustments. Furthermore, the "`Module Not Imported`" issue often arises due to code generation omitting crucial details. Apart from common problems, reviewers often focus on enhancing code robustness by identifying rare exceptions, unused classes, or potential infinite loops.

Likewise, we analyze the tester-programmer communication during the testing phase, illustrating the dynamic debugging dynamics in their multi-turn interactions with compiler feedback, as depicted in Figure 5. The likelihood of successful compilation at each step is generally higher than encountering errors, with most errors persisting and a lower probability of transforming into different errors. The most frequent error is "`ModuleNotFound`" (45.76%), followed by "`NameError`" and "`ImportError`" (each at 15.25%). The observation highlights the model's tendency to overlook basic elements like an "import" statement, underscoring its difficulty in managing intricate details during code generation. Besides, the tester also detects rarer errors like improperly initialized GUIs, incorrect method calls, missing file dependencies, and unused modules. The communicative dehallucination mechanism effectively resolves certain errors, frequently resulting in "compilation success" after code changes. There's a significantly low chance of returning to an error state from a successful compilation. Over time, the multi-turn communication process statisti-

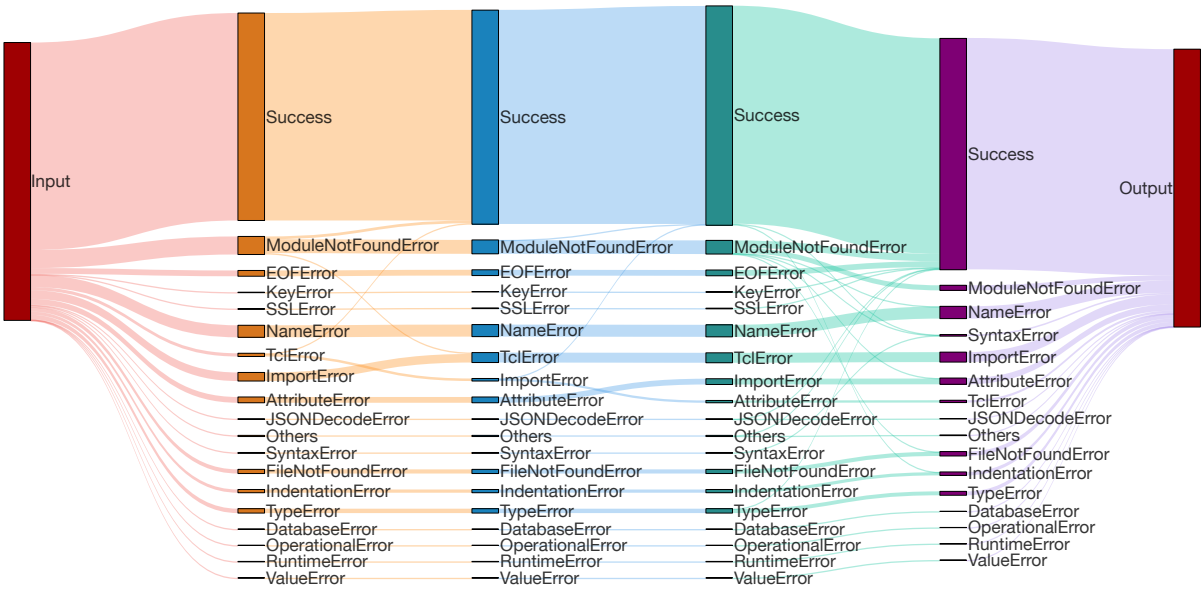

Figure 5: The diagram illustrates the progression of iterations in a multi-round testing process, where each colored column represents a dialogue round, showcasing the evolution of the solution through successive stages of testing.

cally shows a consistent decrease in errors, steadily moving towards successful software execution.

5 Conclusion

We have introduced ChatDev, an innovative multi-agent collaboration framework for software development that utilizes multiple LLM-powered agents to integrate fragmented phases of the waterfall model into a cohesive communication system. It features *chat chain* organizing communication targets and *dehallucination* for resolving coding hallucinations. The results demonstrate its superiority and highlight the benefits of multi-turn communications in software optimization. We aim for the insights to advance LLM agents towards increased autonomy and illuminate the profound effects of "language" and its empowering role across an even broader spectrum of applications.

6 Limitations

Our study explores the potential of cooperative autonomous agents in software development, but certain limitations and risks must be considered by researchers and practitioners. Firstly, the capabilities of autonomous agents in software production might be overestimated. While they enhance development quality, agents often implement simple logic, resulting in low information density. Without clear, detailed requirements, agents struggle to grasp task ideas. For instance, vague guidelines in developing a Snake game lead to basic representations; in information management systems, agents might retrieve static key-value placeholders instead of external databases. Therefore, it is crucial to clearly define detailed software requirements. Currently, these technologies are more suitable for prototype systems rather than complex real-world applications. Secondly, unlike traditional function-level code generation, automating the evaluation of general-purpose software is highly complex. While some efforts have focused on *Human Revision Cost* (Hong et al., 2023), manual verification for large datasets is impractical. Our paper emphasizes completeness, executability, consistency, and overall quality, but future research should consider additional factors such as functionalities, robustness, safety, and user-friendliness. Thirdly, compared to single-agent approaches, multiple agents require more tokens and time, increasing computational demands and environmental impact. Future research should aim to enhance agent capabilities with fewer interactions. Despite these limitations, we believe that engaging a broader, technically proficient audience can unlock additional potential directions in LLM-powered multi-agent collaboration.

Acknowledgments

The work was supported by the National Key R&D Program of China (No.2022ZD0116312), the Postdoctoral Fellowship Program of CPSF under Grant Number GZB20230348, and Tencent Rhino-Bird Focused Research Program.