Think-on-Process: Dynamic Process Generation for Collaborative Development of Multi-Agent System

Leilei Lin*
leilei_lin@126.com
Capital Normal University
Beijing, China

Yingming Zhou*
aizhouym@gmail.com
Capital Normal University
Beijing, China

Wenlong Chen
wenlongchen@sina.com
Capital Normal Univeristy
Beijing, China

Chen Qian†
qianc62@gmail.com
Tsinghua University
Beijing, China

ABSTRACT

Software development is a collaborative endeavor that requires individuals from different departments to work together in order to collectively develop a high-quality software system. In this context, people have begun to explore a method that leverages multi-agent systems based on LLMs to carry out software development. However, existing research tends to rigidly fix the software development process in a framework in code form, thus failing to dynamically adjust the software development process in real-time to meet the more flexible and variable software environment. In this paper, we propose a dynamic process generation framework, named **ToP (Think-on-Process)**. The core idea of ToP is to leverage experiential knowledge (i.e., process models) to guide LLMs in generating software development processes (i.e., instances). These instances will guide multi-agent in software development and employ a compiler to provide feedback on the development outcomes. Subsequently, we utilize heuristic algorithms to filter the instances and apply process mining algorithms to derive process model. Finally, the process model will be converted into text, formatted as prompts, to enhance the ability of LLMs to generate other instances. Experiments demonstrate that our framework ToP significantly enhances the dynamic process generation capability of the GPT-3.5 and GPT-4 for five categories of software development tasks.

KEYWORDS

software development process, large language models, multi-agent, hallucination phenomena

1 INTRODUCTION

In the software development lifecycle, errors that occur during the development phases can result in substantial resource wastage and a decline in workforce efficiency. This can be particularly harmful to startups and small businesses. To maintain the integrity and tractability of the software development phases, numerous companies tackle these challenges by adopting structured software development processes [14, 15]. These processes outline the necessary steps that a development team should take during each production phase, and they strictly manage various facets, including planning, quality assessment, and scheduling. By strictly adhering to a

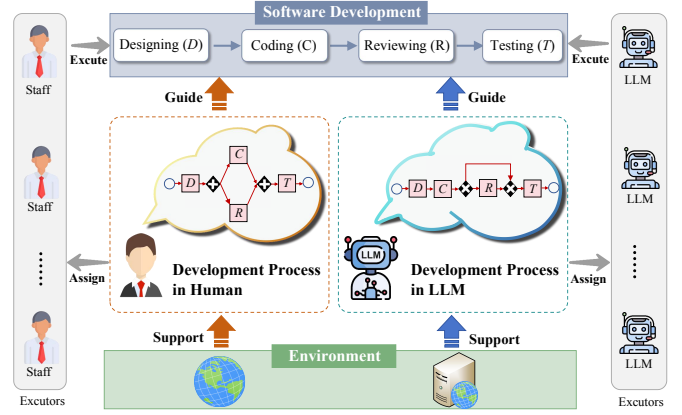*Equal Contribution.
†Corresponding Author.

Figure 1: Traditional software development and agent-based software development.

structured development process, project managers can effectively oversee the project's progress, enabling the early identification and rectification of potential issues. This proactive approach helps to avert future project setbacks and avoidable financial losses [36, 49].

The central concept of a multi-agent system is to decompose complex problems into several distinct subtasks, which are then allocated to autonomous agents within the system for processing [18]. By facilitating communication and collaboration among these agents, the system can effectively solve the original complex problems. Multi-agent systems continue to attract scholarly interest across various disciplines due to their adaptability in addressing complex issues and their potential for intelligence [62]. In recent years, Large Language Models (LLMs) have made substantial advancements in the field of Natural Language Processing (NLP) [9, 10, 35, 52], leveraging massive corpora for training with the objective of "next word prediction." LLMs have exhibited remarkable performance across a spectrum of downstream tasks. For instance, Ni et al. [31] have employed LLMs as automated analysis tools for continuous report analysis, Nijikam et al. [32] have used them to generate executable code through iterative dialogue, and Cheng et al. [13] have harnessed LLMs to create software engineering infrastructure that empowers individuals to innovate with AI. The question arises: How can we harness the full potential of LLMs in
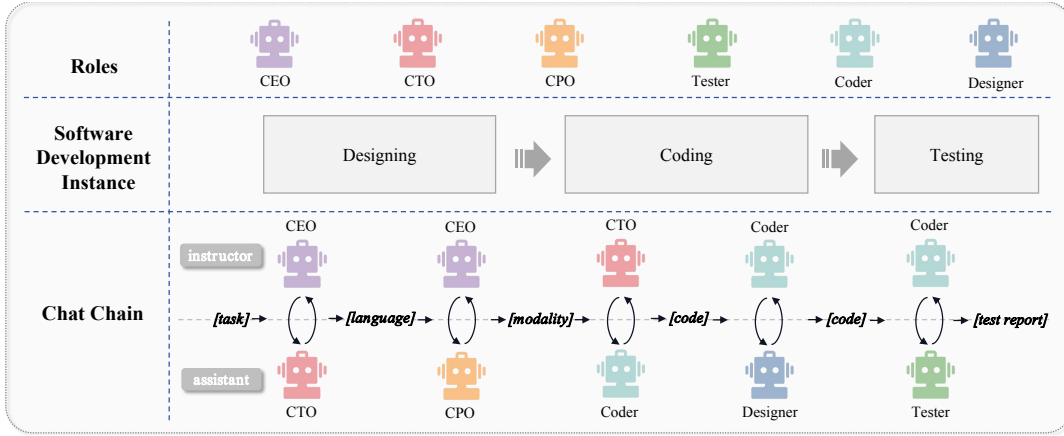
Figure 2: The Chat Chain in Multi-Agent Software Development.

software development? In essence, it involves creating a network of agents that collaborate to execute tasks, which primes us to anticipate the performance of LLM-powered multi-agent system in software development with great interest. As depicted in Figure 1, two fundamentally different approaches to software development are contrasted. Conventional software development relies on human expertise, defining the development process in advance, and tasking different departmental staff with specific phase tasks under the leadership's guidance. In contrast, an LLM-powered multi-agent system leverages the collaboration of multiple autonomous agents to navigate the entire software development lifecycle without human intervention. This method achieves a level of automation in software development that far surpasses traditional models, significantly reducing labor requirements.

Researchers have begun to integrate multi-agent system into software development processes. Qian et al. [39] introduced Chat-Dev, a virtual chat-based software technology company capable of automating software development according to the waterfall model. Qian et al. [40] enhanced the capability of autonomous agents by enabling them to gather shortcut experiences from historical trajectories. These past experiences are then employed for mutual inference, which decreases the recurrence of repetitive errors in software development. Despite these advancements, the current framework for software development utilizing multi-agent system faces limitations, notably its static nature. This implies that the phases in the software development process are immutable, which hampers its ability to accommodate the flexible and evolving demands of modern development. A dynamic software development process, however, can adapt to various development scenarios and optimize resource utilization while satisfying diverse user requirements. Consequently, there is a need to investigate a dynamic process generation framework that can facilitate real-time collaboration within a multi-agent system during software development. Our contributions are as follows:

- We have developed an architecture capable of dynamically generating various process instances to facilitate software development across diverse scenarios. This architecture

assigns autonomous agents to collaborate on development tasks based on these instances.
- To address potential hallucination issues that LLMs may encounter during instance generation, we use temperature settings and external environment (e.g. compiler) to filter out erroneous instances (referring to those with incorrect development phases, unreasonable order of development phases, or both, as well as software compilation errors guided by instances), and use this mechanism to retain high-quality and executable instances.
- We utilize process mining algorithms to extract the software development process from successful instances. These process is then transformed into textual descriptions, which used to augment the dynamic instance generation capabilities of LLMs. The efficacy of our approach has been validated through rigorous experimental analysis.

2 BACKGROUND

Research into guiding multi-agent software development with static processes is extensive and well-documented. In the following section, we will present ChatDev [39], a virtual chat-based software technology company. ChatDev comprises agents with distinct roles that collaborate to develop the necessary software. Additionally, we will delve into the Chat Chain and the three mechanisms employed within ChatDev to gain insight into how the company addresses the challenges of missing dependencies and index out-of-bounds errors that LLMs might face when generating comprehensive software systems.

Chat Chain: Common software development life cycles encompass models such as the waterfall model, V-model, spiral model, and agile development [3, 4, 6]. These models represent distinct approaches to managing the software development process. ChatDev employs the waterfall model, which structures software development into four phases: Designing, Coding, Testing, and Documenting. The strategic allocation of roles for collaborative development within each phase is crucial to the overall success of the software. The Chat Chain mechanism employed by ChatDev breaks down each phase into multiple atomic chats. In each atomic chat, two
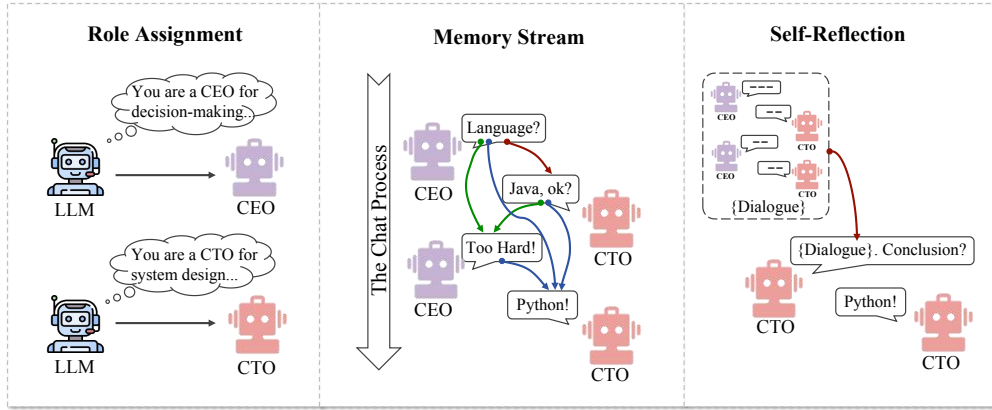
Figure 3: Three mechanisms used in every atomic chat.

agents with predefined roles take on the roles of instructor and assistant, respectively, to engage in focused communication and task execution, ultimately achieving the desired output. Figure 2 illustrates a software development instance comprising three phases: Designing, Coding, and Testing. Each phase is executed through one or two atomic chats. By sequentially completing these atomic chats, the collaborative development of the software is brought to fruition. The Chat Chain's structure provides users with a clear view of every detailed task involved in software development, facilitating subsequent inspections and enhancements. Moreover, the division of software development into smaller subtasks through atomic chats significantly mitigates the potential for code hallucinations, enhancing the overall reliability of the development process.

Having grasped how ChatDev disaggregates software development phases into various atomic chats via the Chat Chain mechanism, we now turn our attention to the three mechanisms employed within each chat, as depicted in Figure 3:

Role Assignment: Prompts are recognized as a critical tool to steer LLMs in the desired direction, with their efficacy substantiated by numerous studies [42, 60, 65]. In role-playing scenarios, prompts containing details such as assigned tasks, roles, output specifications, and communication protocols are provided to different agents. This ensures that each agent comprehends and fulfills its designated responsibilities.

Memory Stream: The memory stream mechanism [35] involves the thorough recording of past communication between agents. This historical dialogue data aids agents in decision-making during new dialogues, preventing the erosion of past experiences [63]. Additionally, by establishing a communication protocol, agents are assured of delivering terminal messages in a consistent format upon reaching a consensus.

Self-Reflection: Occasionally, the assistant's decisions may not align with the communication protocol even when a consensus is reached. This can lead to outputs in subsequent development phases that fall short of expectations. Self-reflection addresses this issue by prompting the assistant to revisit past dialogues (effectively initiating a new dialogue round based on historical context). This process facilitates the derivation of conclusions that adhere to

the established communication protocol. These mechanisms collectively enhance the effectiveness and reliability of the collaborative development process managed by ChatDev.

3 METHODOLOGY

By dynamically generating the development process, our framework can effectively meet the adaptable and evolving needs of software development, while also providing guidance for ongoing multi-agent collaboration. As depicted in Figure 4, we will outline our framework through four stages: Instance generating, Software developing, Instance filtering, and LLMs enhancing.

3.1 Instance generating

Dynamic process generation involves the creation of various instances based on LLMs' existing knowledge of software development, tailored to the specific and varied needs of users. These instances serve as blueprints to direct the activities of multiple agents involved in software development. It is essential to recognize that software development processes can differ significantly depending on the project requirements. Consequently, adhering to a single, static development process is not always feasible or efficient. Our proposal for dynamic process generation aims to accommodate this variability, enabling a wider array of diverse instances to emerge and guide the development process accordingly. The concept of instances and diverse instances is explained as follows:

- Instance: An instance represents a software development process that includes different phases of software development. These phases are arranged in chronological order. As shown in Figure 4, "Designing (D) → Coding (C) → Testing (T)" is an instance.
- Diverse instances: Diverse instances refer to multiple instances that are pairwise different from each other, including different phases, orders, or both. For example, "D → C → T", "D → R → C → T", and "D → C → R →T" are diverse instances.

As shown in 5, when LLMs receives a development task similar to "Develop a tetris game," we use the prompt engineering to prompt it to generate diverse instances. The core components of prompt
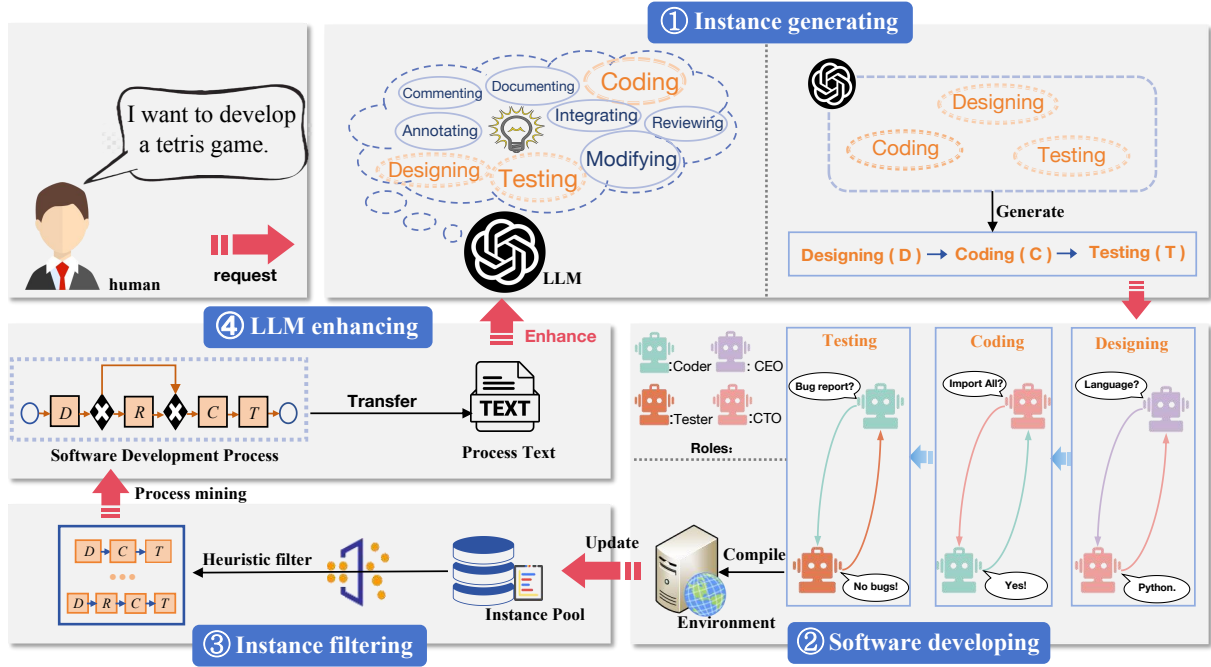
Figure 4: An overview of our method named ToP.

includes four parts: 1) *User Task*: Provided by the user, which is then passed as a prompt to LLMs; 2) *Phase Explanations*: The phase explanations provide a detailed description of each phase and the development tasks that need to be completed, ensuring that LLMs can generate the correct phases in subsequent instance generation; 3) *Experiential Instances*: To better guide LLMs in accurately generating instances, provide some existing instances each time; 4) *Output Format*: Using the special symbol '→' to connect different phases into a complete instance. LLMs assigns probabilities to all possible tokens based on its understanding of the entire context. Subsequently, it selects the next token in the output based on these probabilities. Temperature is a hyperparameter used to regulate the

creativity level of LLMs in text generation. By adjusting the temperature value, the probability distribution of LLMs can be affected, resulting in more concentrated or diversified text output.

3.2 Software developing

Once an instance is generated, ChatDev uses it as a blueprint to guide the software development process. ChatDev systematically progresses through each phase detailed in the instance, orchestrating the collaboration of multiple agents to fulfill the tasks. Prior to executing a task, each agent is assigned a specific role, which may change depending on the phase. As illustrated in Figure 4, during the Designing phase, for example, ChatDev assigns two agents the roles of CEO (chief executive officer) and CTO (chief technology officer). These agents then work together through the Chat Chain to complete the design tasks. Throughout their dialogue, ChatDev incorporates three key mechanisms: Role Assignment, Memory Stream, and Self-Reflection, to ensure that the software development process is both efficient and accurate. ChatDev supports a total of seven roles: CEO (chief executive officer), CPO (chief product officer), CTO (chief technology officer), Programmer (also known as Coder), Reviewer, Designer, and Tester. Additionally, ChatDev offers a diverse set of software runtime environments, including numpy, matplotlib, pandas, tkinter, pillow, and flask, among others. Once the software development is complete, testers compile the software within the provided runtime environment and assess whether the instance used to guide the development was successful. These records serve as environmental feedback for the LLMs, facilitating continuous improvement and learning.
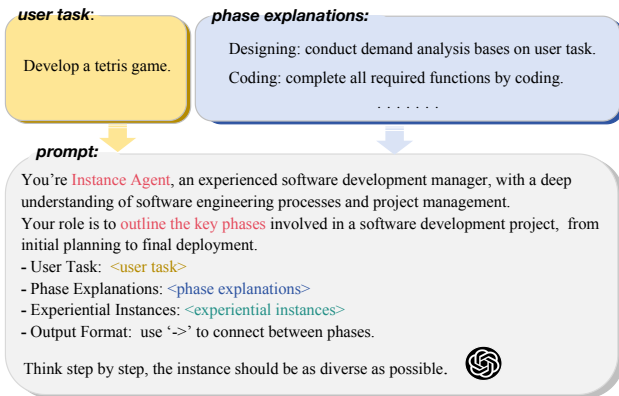

Figure 5: The prompts for LLMs to generate instances.

3.3 Instance filtering

Once the software development stage is completed, the instances are logged in an instance pool. As detailed in Table 1, each entry in the pool primarily consists of: the specific content of the instance, the frequency of occurrence of the instance, and the success count of instances that successfully passed compilation.

During the instance generation process guided by prompts, LLMs are susceptible to what we term "instance hallucinations." Instance hallucinations refers to the occurrence of one or more of the following anomalies: 1) Incorrect phases. For example, the phase "UsernameSet" appears in the instance "Designing → Coding → UsernameSet → Testing," but this phase is not defined in the phase explanations and does not have any specific meaning in software development; 2) Incorrect order. For example, the instance "Designing → Testing → Coding" indicates that testing is performed before coding, which is contrary to the standard sequence of software development. 3) A combination of both incorrect phases and incorrect order. Clearly, instances containing such irregularities have the potential to lead to failures in subsequent software compilation. To address this issue, we employ two mechanisms: the *Replay mechanism* and the *Heuristic filter*, to reduce the impact of instance hallucinations.

Table 1: Examples for illustrating records within the instance pool.

ID	Instance	Frequency	Success Counts
1	C → T	14	6
2	D → C → T	21	13
...	...	...	...
n	D → R → C → T	30	21

Replay mechanism: To leverage the experience gained from prior development efforts and enhance the efficiency of software development, we maintain a comprehensive record of every instance used to guide development in the instance pool. However, given the occurrence of hallucination phenomena, some successfully compiled instances may be instance hallucinations, and some failed instances may also be correct. Therefore, it is necessary to randomly select instances from the instance pool for re-evaluation. This process is also called "replay".

$$UCT(i) = 1 - \frac{Q(i)}{N(i)} + C * \sqrt{\frac{\ln N}{N(i)}} \quad (1)$$

Where, $Q(i)$ represents the successful compilation count of the $i - th$ instance. $N(i)$ represents the occurrence frequency of the $i - th$ instance. C is a constant used to balance success rates and occurrence frequency, defaulting to 1. N represents the sum of occurrence frequency of all instances.

Heuristic filter: After replaying, the frequency and success count of each instance in the instance pool are updated. At this point, we can use heuristic rules to extract instances with high *success rate (SR)* as the experience for LLMs:

$$SR(i) = \frac{Q(i)}{N(i)} \quad (2)$$

It should be noted that *Heuristic filter* is to filter those instances with high success rate for experience gathering, while the focus of the *Replay mechanism* is to select those instances with lower success rate and lower frequency of occurrence in the instance pool for testing, thereby reducing the impact of instance hallucinations on the entire software.

3.4 LLM enhancing

The final stage of our framework is LLM enhancing, where we mine process models from instances with a high success rate. The objective is to improve the dynamic instance generation capabilities of LLM by translating these process models into a textual description that is more accessible and comprehensible for LLM to utilize as prompt input.

Process mining has garnered widespread attention in the field of business process management in recent years [54]. This is due to its ability to accurately construct business process models from vast amounts of event logs and guide the step-by-step execution of business processes. In this paper, we opt to employ the Inductive Miner (IM) algorithm [21] to mine process model from instances, representing it in the form of BPMN (Business Process Model and Notation). The main reason for this choice is that IM excels in all performance metrics, as highlighted in the preceding Related Work section. BPMN is widely embraced in both industry and academia for its rich set of elements that enable the creation of simple and understandable process models, accurately expressing the user's intentions [33]. In the Table 3, we list the main elements covered in this paper, including start and end events, sequence flow, activities and gateways, where activities representing steps or phases in task execution, and gateways representing relationships between these steps.

Table 2: The BPMN elements and descriptions.

BPMN Elments	Diagrams	Descriptions
Start Event	🟢	Represents the beginning of a process.
End Event	🟠	Marks the conclusion or completion of a process.
Activities	▭	Tasks or work that needs to be performed within the process.
Exclusive Gateway	⨂	Denotes a decision point where only one of multiple paths can be taken.
Parallel Gateway	⊕	Allows the process to split into multiple parallel paths or merge from multiple parallel paths.
Sequence flow	⟶	Represents the order in which activities, gateways, and events are performed or executed.

Definition 1 (directly follows relation →→). Let $\sum$ be an alphabet such that $\top \notin \sum$ and $\bot \notin \sum$ and let L be a language over $\sum$. We define the following relations:

$$a \twoheadrightarrow b \Leftrightarrow \exists_{t \in L} \; t = < \dots \; a, b, \dots >$$

$$\top \twoheadrightarrow a \Leftrightarrow \exists_{t \in L} \; t = < a, \dots >$$

$$a \twoheadrightarrow \bot \Leftrightarrow \exists_{t \in L} \; t = < \dots, a >$$

Definition 1 states a relationship between two phases, where t represents an instance, a or b represent phase, $\top$ and $\bot$ represent the start event and end event, respectively. The directly follows graph (DFG) is a directed graph: its nodes are phases, and its edges

denote which phases can directly follow one another. In Figure 6, we provide an example illustrating how to mine the software development process model from two instances based on the IM algorithm. First, a directly follows graph is constructed from filter instances based on directly follows relation. Second, the IM algorithm transforms DFG into a process tree based on four *cut operations*: sequence ($\rightarrow$), exclusive ($\times$), parallel ($+$), and loop ($\circlearrowleft$). The process tree is a directed connected graph without cycles. A node in the graph is either a *branch node* or a *leaf node*. Each leaf node represents phase from the collection of instances.

Each branch node, or operator node, has one or more children. These children can be other operator nodes or leaf nodes. It is worth noting that an invisible activity τ is often added in process tree to maintain the soundness. Finally, the process tree is equivalently transformed into BPMN to graphically display the software development process model from the instances. And the BPMN shows that the IM algorithm accurately mines selection relationships within dynamic instances by constructing the directly follows graph and conducting cut operations.
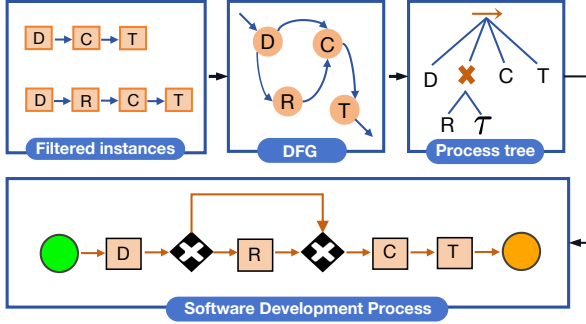

Figure 6: An example for illustrating process model mining.

Understanding BPMN can be challenging, especially since there are pairs of exclusive and parallel gateways in the model. Due to the emergence of different gateways, the relationship between activities in the model becomes more complicated, including sequence, concurrency and selection. Therefore, if a BPMN model is directly input into LLM, there's no guarantee that LLM will accurately interpret the relationships between different activities within the model. Well, the impressive performance of LLM in various benchmark tests in recent years demonstrates their ability to comprehend the semantic information within text [11, 64]. To bridge this gap, we first translate the BPMN model into a process textual description, as illustrated in Figure 7. This conversion involves three key steps: 1) *Text Planning*: This initial step involves deciding what information needs to be communicated and establishing the sequence in which it should be presented. 2) *Sentence Planning*: In this step, we select the appropriate vocabulary to convey the information that was identified in the previous step. 3) *Realization*: The last step is to convert the acquired information into grammatically correct sentences.

Once the conversion is complete, we obtain a process textual description that retains the original semantics of the BPMN model. This description is then integrated into the prompt to enhance the
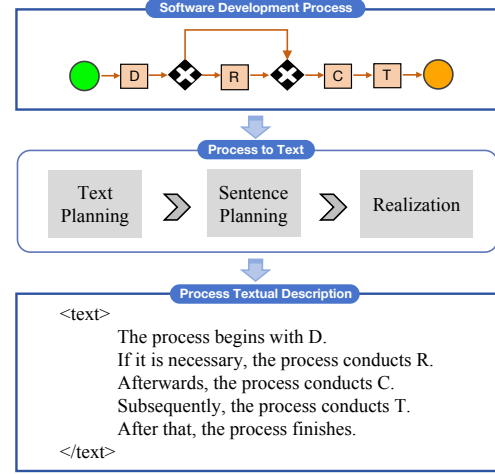

Figure 7: Conversion steps from software development process to process textual description.

dynamic instance generation capabilities of LLMs. The updated prompt, which includes the process textual description, is depicted in Figure 8.
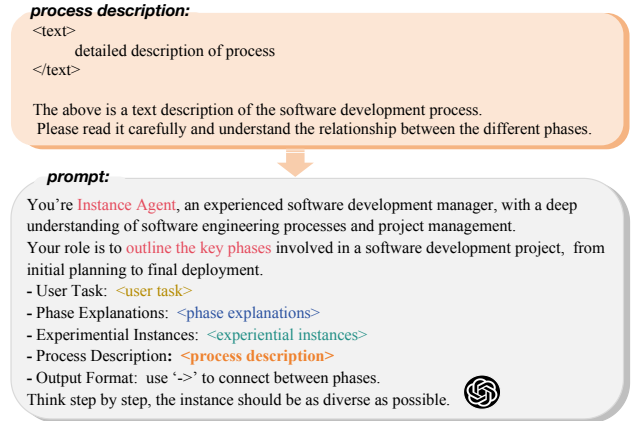
Figure 8: Prompt contents after adding process description.

4 EXPERIMENTS

4.1 Setup

Our experiment utilized OpenAI's GPT-3.5-turbo model[1] as the backbone for simulating multi-agent software development. For the Python-based system, we employed Python 3.9.18 as the external compiler interpreter, and all the code is publicly available[2].

4.1.1 Datasets: To validate the feasibility of our framework, we chose to use the NLDD dataset [39], an open source dataset dedicated to the field of "Natural Language to Software Generation", and the dataset contains more than 1200 software task prompts,

---

[1]https://openai.com/
[2]https://github.com/Aizhouym/Think-on-Process

these prompts cover the five major areas of Creation, Game, Education, Work, and Life. By selecting software tasks from these five categories with equal probability as user demand input, our experiment accounts for the diverse software development needs across various fields.

4.1.2 Configurations: During the software development of ToP, we use ChatDev[3] as the backbone. After generating an instance based on different user requirements, this instance is divided into multiple phases according to the software development lifecycle for execution. Each phase requires the participation of agents with different roles. [47] refined the software development lifecycle into five different categories based on the knowledge areas mentioned in SWEBOK [1]: 1) software requirements, 2) software design, 3) software construction, 4) software quality, and 5) software maintenance. To achieve more efficient software development, we further subdivided these five categories, resulting in a default instance with 14 phases, as shown in Table 3.

4.2 Hyperparameter Analysis

Large language models can typically control the randomness of their output by adjusting the temperature parameter. Generally, the temperature value falls within the range [0.0, 1.5]. A higher value indicates greater randomness and creativity, but it may also lead to more errors [66]. Conversely, a lower value results in reduced randomness, generating content that is overly conservative and repetitive.

As the temperature value decreases, the diversity instances decrease. However, when the temperature value increases, a large number of irrelevant instances will be generated, resulting in a waste of learning resources. Therefore, we need to explore a balance that allows LLMs to generate diverse instances while ensuring a reasonable success rate. Before running the experiment, we would like to introduce the diversity metrics:

- *Diversity*: The diversity of each instance is quantified by comparing its phases and order variations against a default instance, which is composed of 14 predefined phases based on expert knowledge. A higher diversity value signifies a greater differences between the instances and a richer variety in the generated instances. Conversely, a lower diversity value suggests a lesser degree of variation, with instances tending to be more similar or invariant [30].

$$Diversity(i) = \frac{Change(Phases) + Change(Orders)}{2} \quad (3)$$

$$Change(Phases) = \frac{|P - (P \cap N)| + |N - (P \cap N)|}{|P \cup N|} \quad (4)$$

$$Change(Orders) = \frac{|O - (O \cap E)| + |E - (O \cap E)|}{|O \cup E|} \quad (5)$$

In equation 3, diversity of instance i is calculated by computing the average of the changes in phases and orders compared to the default instance. In equation 4, P represents the set of phases constituting the default instance, and N represents the set of phases constituting instance i. In equation 5, O represents the set of orders constituting the default instance, and E represents the set of orders constituting instance i.

Figure 9 illustrates the relationship between temperature and two metrics: success rate and diversity. When the temperature increases from 0.0 to 0.2, the success rate experiences a substantial decline of 25.7%. Concurrently, diversity shows a marked increase, rising by 9.6%. As the temperature fluctuates between 0.2 and 0.6, there is a minimal change in both success rate and diversity, suggesting a relatively stable condition. Beyond a temperature of 0.6, however, there is a notable decrease in success rate accompanied by a significant rise in diversity. For instance, at a temperature of 1.4, over half of the instance contents are randomly generated, and the success rate falls below 30%, indicating that these instances are almost impossible to successfully guide software development.

In summary, as the temperature rises, the diversity of instances produced by LLMs increases. Conversely, the success rate of these instances in subsequent software development diminishes. Therefore, to ensure that instances generated by LLMs exhibit increased diversity while still achieving a reasonable level of success in test compilation, the temperature should not be set too high (resulting in low success rate) or too low (leading to low diversity). An intermediate temperature value is often the most effective choice. In this study, we default to a temperature of 0.6.

4.3 UCT Analysis

The occurrence of instance hallucinations can lead to instances that contain errors when compiled within the external environment (for example, the wrong instance may pass the compilation, but the correct one cannot pass). This, in turn, may result in the incorrect mining of the software development process model. To address this issue, we implement a replay mechanism that selects instances from the instance pool for re-evaluation. The choice of strategy significantly impacts the selection results. Common strategies include:

Table 3: Software Development Lifecycle (SDLC) and Specific Phases

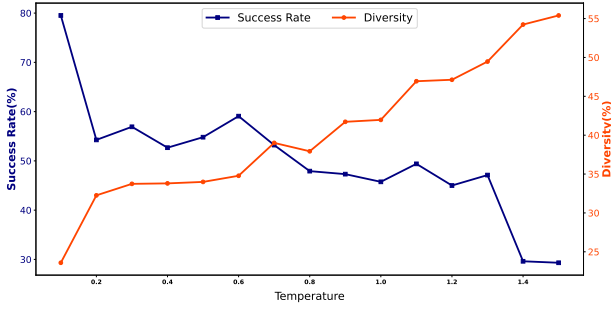| SDLC	Specific Phases
Software Requirements	DemandAnalysis,LanguageChoose
Software Design	DesignReview
Software Construction	Coding, CodeComplete, Annotation, CodeConclusion, CodeReviewComment
Software Quality	CommentJudgement, CodeReviewModification, TestErrorSummary, TestModification
Software Maintenance	EnvironmentDoc, Manual

Figure 9: The impact of different temperature on *SR* and *Diversity*.

Figure 10: Comparison of different selection strategies.

1) *Failure rate-based*: This strategy gives priority to those instances with higher failure rates, hoping to reduce the number of instances that are semantically correct but fail to compile. 2) *Frequency-based*: This strategy preferentially selects instances with low access times, because those instances with low frequency cannot determine whether they are instance hallucinations. 3) *UCT (Upper Confidence Bound for Trees)*: This strategy is an overall consideration of instance failure rate and access frequency, seeking to achieve a balance between these two factors.

To assess which instance selection strategy is most effective for the replay mechanism, we manually selected 100 diverse instances (including hallucination instances) from the instance pool which containing 1000 instances. We then employed three different selection strategies for replay and evaluated required attempt times for each strategy to complete all instance traversals. It's important to note that each instance retained its frequency and success rate attributes when it was initially selected from the instance pool. As depicted in Figure 10, the Frequency-based strategy initially quickly accesses instances with smaller times. However, as the number of times increases, the available instances for selection decrease, making it challenging to access instances with high initial frequency. In contrast, the UCT strategy may not perform as well at the outset but demonstrates strong access stability, ensuring continuous access to new instances. After 197 times, the UCT strategy successfully traversed all instances. The least efficient strategy is Failure rate-based, because even if the Times = 500, it is still impossible to complete the traversal of all instances. This is because if there are instances with success rate = 100% among 100 instances, these instances will never be accessed.

To sum up, UCT as an instance selection strategy for the replay mechanism is relatively reasonable and reliable. It effectively combines elements of both the Failure rate-based and Frequency-based strategies, providing a balanced approach to refining the instance pool.

4.4 Filtering Mechanism Effectiveness

From LLMs generating instances, to multi-agent collaboration for software development, to subsequent completion of external environment compilation, the entire process takes about 8 minutes. In this article, we generated 1,000 instances and stored 350 different variants (i.e., diverse instances) in the instance pool. We counted the number of instances obtained under different success rate filters,
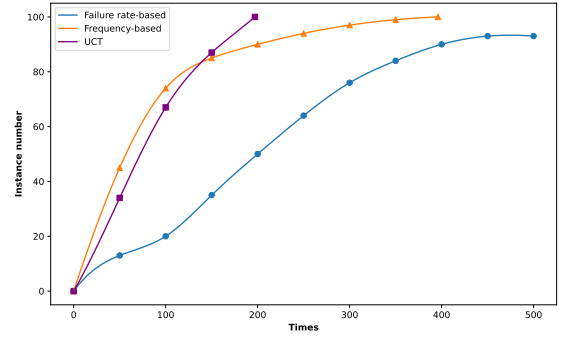
and the distribution of results is shown in Figure 11. The data reveals that the majority of instances generated by LLMs are not compatible with the external environment. This outcome is understandable, given that software development under LLMs guidance involves many independent adjustments, which undoubtedly increase the likelihood of errors. Humans, on the other hand, can adjust the development process in real-time based on past experience and external feedback.
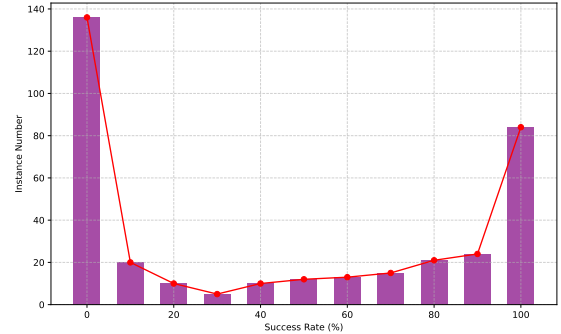

Figure 11: The number of process instances under different success rates.

At the same time, it can be observed that the line chart presents a bimodal distribution, with the inflection point located at 30%. Therefore, we choose a success rate of 30% as the threshold for instance filtering to reduce the impact of instance hallucinations.

In order to clearly compare the differences in software development process models before and after filtering, we use the IM algorithm to mine the models of the instance pools of the two situations respectively. The results are presented in Figure 12.

Figure12(a), the software development process obtained directly without filtering is chaotic. The model includes 20 phases (here we regard each activity as a phase in software development) and 14 parallel gateways and 58 exclusive gateways, and there are obvious errors in the phases (e.g., "UsernameSet", "ExternalReview", "Comments", "EmailSet", "EmploymentDoc") and orders (e.g., "DesignReview → DemandAnalysis", "Coding → CommentJudgement", "LanguageChoose → Manual", "Annotation → TestErrorModification", "CodeReviewComment → DesignReview") in the process

(a) The process model without filtering.

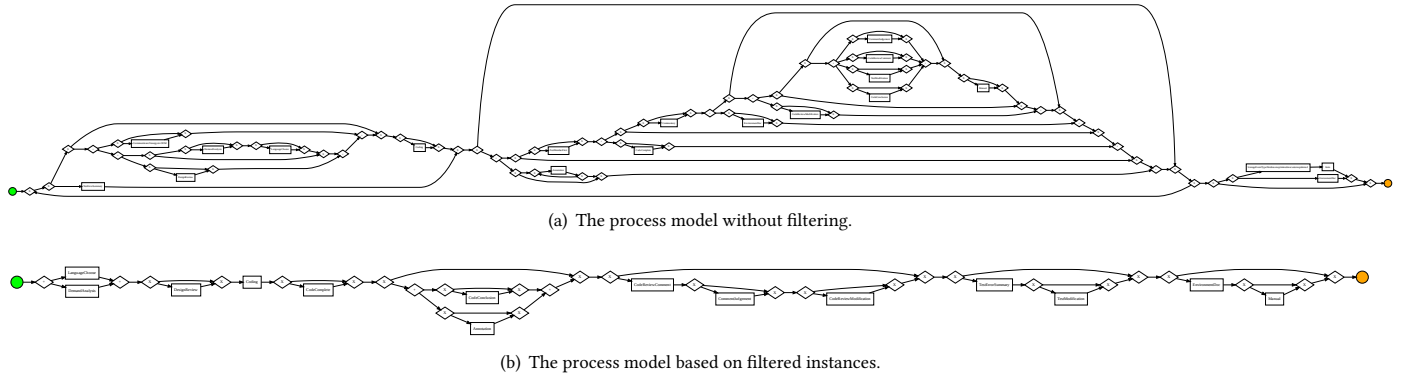(b) The process model based on filtered instances.

Figure 12: The process models of software development for multi-agent system.

model. In contrast, Figure 12(b) presents a clear software development process. The number of phases is reduced to 14, aligning with the phases outlined in our experimental phase explanations. Additionally, the number of gateways is reduced by 65%, with only 4 parallel gateways and 24 exclusive gateways, indicating a more efficient and coherent process model.

4.5 Enhancement Experiment

The software development process model can serve as the accumulated development experience for LLM, enhancing their dynamic process generation capabilities. By converting the process model into a process textual description with equivalent semantics, we can input this description to LLM as a prompt. To assess the effectiveness of this approach, 200 different software development tasks were selected from each of the five major categories of the NLDD dataset as user input. The success rates of the software development process under the two methods (with and without process textual description input) were then compared.

From Figure 13(a), it is evident that the incorporation of process textual description directly impacts the software development success rate of LLM. The development success rate of LLM is elevated by an average of 32% across the five distinct software development categories when compared to the baseline without enhancement.

Notably, the category with the highest success rate is Game, with a leap from 30% to 67% after LLM enhancement. Conversely, the category with the lowest success rate is Education, where the enhancement brings the success rate up from 18% to 46%. This phenomenon is logical, as LLM possess a deeper understanding of simple codes and can readily write and call functions. Education-related software development, however, relies heavily on specific professional domain knowledge, posing a challenge for general language models. Figure 13(b) illustrates the results obtained with GPT-4. Due to the promotion of the model's own performance, the success rate across all categories has increased no matter existing the enhancement or not. In coparision to different categories, it is noteworthy that the highest success rate before enhancement is only 56%. Post-enhancement, the lowest success rate across all categories reaches 78%, while the highest reaches 85%, highlighting a substantial improvement.

In conclusion, integrating process textual descriptions with equivalent semantics into LLM can substantially bolster their generation capabilities. This approach enables the acquisition of more robust and comprehensive development software.
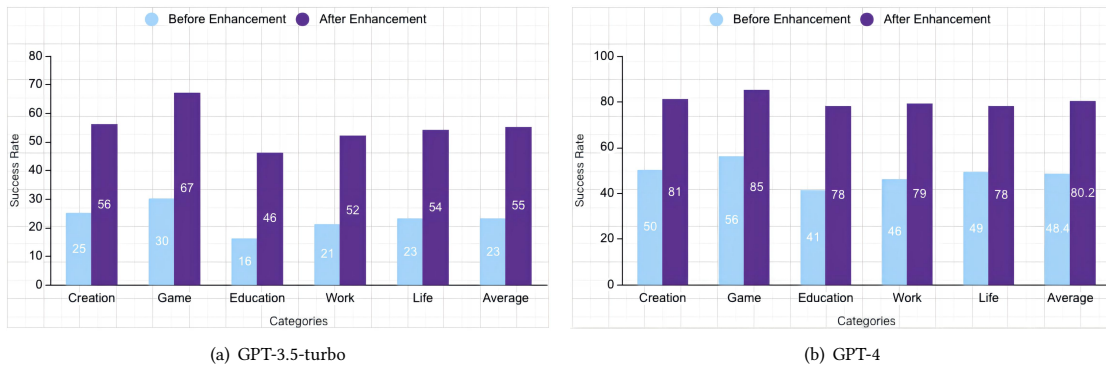

(a) GPT-3.5-turbo

(b) GPT-4

Figure 13: Effectiveness of ToP's enhancements to LLMs on 5 different types of development tasks.

5 RELATED WORK

Software Development Software development is crucial for human utilization of automated data processing. It should be carried out gradually and in phases, involving continuous user participation in the design and programming processes [17, 37, 51]. An efficient and rational software development process can guide developers in swiftly accomplishing intricate development tasks. Bassil et al. [7] proposes a novel waterfall model that effectively mitigates potential issues in the software development process, such as budget overruns, delivery delays, and customer dissatisfaction. Additionally, in various stages of software development, people are increasingly adopting AI technologies to assist and expedite the software development process [19, 58, 61]. For instance, during the requirements phase, Pudlitz et al. [38] suggests the use of a self-trained named-entity recognition model employing bidirectional LSTM and CNN to extract the status of natural language requirements. And during the test phase, unlike LLMs, which is used for the entire unit test [26], Lemieux et al. [22] combined the traditional search-based software testing technology with LLMs technology. When the test coverage of the traditional technology reaches the maximum, LLMs generates corresponding test cases for the functions that are not covered. However, the software processes employed in the aforementioned studies are static, neglecting the consideration of dynamic software development processes.

Large Language Models Large Language models such as ChatGPT [34] and LLaMA [52] have demonstrated their remarkable capabilities in the field of natural language processing, after extensive training on massive datasets [12, 20, 46, 59]. Qin et al. [41] proposed a new sorting prompt technique (Pairwise Ranking Prompt) that has achieved significant improvements on existing datasets. Li et al. [24] leveraged the outstanding ability of large language models in semantic understanding to propose LLaRA, which cleverly transforms large models into dense retrieval encoders, greatly improving the performance of the model on various dense retrieval benchmarks. Song et al. [50] proposes using LLMs for few-shot planning and enhancing LLMs with physical principles to generate and update grounded plans in the current environment. Sharan et al. [48] introduces a novel hybrid planner that combines traditional rules with LLMs. It utilizes the reasoning capabilities of LLMs for task planning, achieving state-of-the-art performance on the nuPlan benchmark. Planing technology enables LLMs to efficiently choose different processes to accomplish tasks, providing a foundational guarantee for the generation of dynamic processes.

Recently, there has been a growing interest in developing software with LLMs [27, 28, 45]. CodeGen [32] can generate code for user-defined tasks step by step through multi-turn dialogues with LLMs. ChatDev [39] divides software development into four steps and guides the collaboration of multi-agent through these steps to accomplish the development of applications and the generation of documentation. However, LLMs in the process of software development are prone to experiencing hallucination phenomena. Lightman et al. and Mackenna et al. [25, 29] believed that the generation of hallucinations in LLMs is caused by overfitting or noisy data during the training process. In summary, LLMs can effectively generate dynamic processes, but they are prone to hallucination

phenomena, which may hinder their effectiveness in guiding software development.

Process Mining Process mining is a data-driven technique that can be used to discovery, analyze, and improve the ongoing process model based on historical records (e.g., massive instances) [54]. Process discovery is one of the three major scenarios in process mining, which is to automatically mine process models from historical records without any apriori information [53]. Until now, many process discovery algorithms have been proposed like Alpha [55], Genetic Miner [56], Inductive Miner [21], etc. It is crucial to emphasize that process discovery algorithms not only showcase historical behaviors but also reveal additional behaviors beyond the historical scope. In other words, the process model's behavior may surpass historical behavior, constituting what is commonly referred to as the generalization [57] ability in the field of process mining. Essentially, the extra behaviors derived from historical records can be leveraged to enhance the capabilities of LLMs. Furthermore, fitness [8] and precision [2] serve as assurances to prevent the process model from deviating from reality. In this setting, Inductive Miner (IM) is selected as the mining algorithm due to its commendable performance in generalization, fitness, and precision [5]. Moreover, the resulting process models from IM are structured, facilitating readability and comprehension. Though BPMN models represent complex relationships between different activities by exclusive gateways, parallel gateways, swimlanes, etc. It is difficult for most people to understand BPMN models because it often requires corresponding expert knowledge. The emergence of BPM2Text technology solves this problem well [16, 23, 43, 44], by converting BPMN models into natural language texts with the same semantics, so that non-professionals can also understand and verify these models well.

6 CONCLUSIONS

The automation of software development is a valuable research topic that can effectively enhance team efficiency. In this paper, we propose a framework called ToP for dynamically generating software development processes, which can produce various instances to guide multi-agent in software development tasks. To start with, LLMs would generate an instance based on user requirement, then the instance be used to guide multi-agent for software developing. Furthermore, we utilize compilers as external tools for feedback on these instances, employing heuristic algorithm to mitigate the phenomenon of hallucination in the generated instances. Finally, we apply process mining techniques to extract process models from successful instances as experiential knowledge. ToP leverages experiential knowledge to enhance the ability of LLMs to generate other instances. Extensive experiments have been conducted on two LLMs, GPT-3.5 and GPT-4, demonstrating that our approach effectively improves the success rates of software development across five categories of tasks.

In the future, we will continue to explore the following aspects: on one hand, identifying more comprehensive feedback mechanisms to replace compilers; on the other hand, fully leveraging failed instances to further enhance LLMs, rather than limiting our focus to successful instances only.

REFERENCES

[1] Alain Abran, Pierre Bourque, Robert Dupuis, and James W Moore. 2001. *Guide to the software engineering body of knowledge-SWEBOK*. IEEE Press.

[2] Arya Adriansyah, Jorge Munoz-Gama, Josep Carmona, Boudewijn F van Dongen, and Wil MP van der Aalst. 2013. Alignment based precision checking. In *Business Process Management Workshops: BPM 2012 International Workshops, Tallinn, Estonia, September 3, 2012. Revised Papers 10*. Springer, 137–149.

[3] Jide ET Akinsola, Afolakemi S Ogunbanwo, Olatunji J Okesola, Isaac J Odun-Ayo, Florence D Ayegbusi, and Ayodele A Adebiyi. 2020. Comparative analysis of software development life cycle models (SDLC). In *Intelligent Algorithms in Software Engineering: Proceedings of the 9th Computer Science On-line Conference 2020, Volume 1 9*. Springer, 310–322.

[4] Adel Alshamrani and Abdullah Bahattab. 2015. A comparison between three SDLC models waterfall model, spiral model, and Incremental/Iterative model. *International Journal of Computer Science Issues (IJCSI)* 12, 1 (2015), 106.

[5] Adriano Augusto, Raffaele Conforti, Marlon Dumas, Marcello La Rosa, Fabrizio Maria Maggi, Andrea Marrella, Massimo Mecella, and Allar Soo. 2018. Automated discovery of process models from event logs: Review and benchmark. *IEEE transactions on knowledge and data engineering* 31, 4 (2018), 686–705.

[6] Sundramoorthy Balaji and M Sundararajan Murugaiyan. 2012. Waterfall vs. V-Model vs. Agile: A comparative study on SDLC. *International Journal of Information Technology and Business Management* 2, 1 (2012), 26–30.

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

[8] Alessandro Berti and Wil MP van der Aalst. 2019. Reviving Token-based Replay: Increasing Speed While Improving Diagnostics. *ATAED@ Petri Nets/ACSD* 2371 (2019), 87–103.

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

[10] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712* (2023).

[11] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2024. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology* 15, 3 (2024), 1–45.

[12] Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chen Qian, Chi-Min Chan, Yujia Qin, Yaxi Lu, Ruobing Xie, et al. 2023. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors in agents. *arXiv preprint arXiv:2308.10848* (2023).

[13] Yu Cheng, Jieshan Chen, Qing Huang, Zhenchang Xing, Xiwei Xu, and Qinghua Lu. 2023. Prompt sapper: a LLM-empowered production tool for building AI chains. *ACM Transactions on Software Engineering and Methodology* (2023).

[14] Paul Clarke and Rory V O'connor. 2012. The situational factors that affect the software development process: Towards a comprehensive reference framework. *Information and software technology* 54, 5 (2012), 433–447.

[15] Gerry Coleman and Rory V O'Connor. 2008. An investigation into software development process formation in software start-ups. *Journal of Enterprise Information Management* 21, 6 (2008), 633–648.

[16] Luis Delicado Alcántara, Josep Sanchez-Ferreres, Josep Carmona Vargas, and Lluís Padró. 2017. Nlp4bpm: Natural language processing tools for business process management. In *BPM Demo and Industrial Track 2017 Proceedings*. 1–5.

[17] Mihai Liviu Despa. 2014. Comparative study on software development methodologies. *Database Systems Journal* 5, 3 (2014).

[18] Ali Dorri, Salil S Kanhere, and Raja Jurdak. 2018. Multi-agent systems: A survey. *Ieee Access* 6 (2018), 28573–28593.

[19] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. *arXiv preprint arXiv:2310.03533* (2023).

[20] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. arXiv:2001.08361 [cs.LG]

[21] Sander JJ Leemans, Dirk Fahland, and Wil MP Van der Aalst. 2014. Discovering block-structured process models from event logs containing infrequent behaviour. In *Business Process Management Workshops: BPM 2013 International Workshops, Beijing, China, August 26, 2013, Revised Papers 11*. Springer, 66–78.

[22] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 919–931.

[23] Henrik Leopold, Jan Mendling, and Artem Polyvyanyy. 2012. Generating natural language texts from business process models. In *Advanced Information Systems Engineering: 24th International Conference, CAiSE 2012, Gdansk, Poland, June 25-29, 2012. Proceedings 24*. Springer, 64–79.

[24] Chaofan Li, Zheng Liu, Shitao Xiao, and Yingxia Shao. 2023. Making Large Language Models A Better Foundation For Dense Retrieval. *arXiv preprint arXiv:2312.15503* (2023).

[25] Hunter Lightman, Vineet Kosaraju, Yura Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's Verify Step by Step. *ArXiv* abs/2305.20050 (2023). https://api.semanticscholar.org/CorpusID:258987659

[26] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 168–172.

[27] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568* (2023).

[28] Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. 2023. Experiences from using code explanations generated by large language models in a web software development e-book. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 931–937.

[29] Nick McKenna, Tianyi Li, Liang Cheng, Mohammad Javad Hosseini, Mark Johnson, and Mark Steedman. 2023. Sources of Hallucination by Large Language Models on Inference Tasks. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. https://api.semanticscholar.org/CorpusID:258865517

[30] Álvaro Menezes and Rafael Prikladnicki. 2018. Diversity in software engineering. In *Proceedings of the 11th International workshop on cooperative and human aspects of software engineering*. 45–48.

[31] Jingwei Ni, Julia Bingler, Chiara Colesanti-Senni, Mathias Kraus, Glen Gostlow, Tobias Schimanski, Dominik Stammbach, Saeid Ashraf Vaghefi, Qian Wang, Nicolas Webersinke, et al. 2023. CHATREPORT: Democratizing Sustainability Disclosure Analysis through LLM-based Tools. *arXiv preprint arXiv:2307.15770* (2023).

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

[33] Object Management Group. 2011. BPMN Specification. https://www.bpmn.org/.

[34] OpenAI. 2022. ChatGPT. https://openai.com/blog/chatgpt/.

[35] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. 1–22.

[36] Freddy Paz and José Antonio Pow-Sang. 2016. A systematic mapping review of usability evaluation methods for software development process. *International Journal of Software Engineering and Its Applications* 10, 1 (2016), 165–178.

[37] Mary Poppendieck and Michael A Cusumano. 2012. Lean software development: A tutorial. *IEEE software* 29, 5 (2012), 26–32.

[38] Florian Pudlitz, Florian Brokhausen, and Andreas Vogelsang. 2019. Extraction of system states from natural language requirements. In *2019 IEEE 27th International Requirements Engineering Conference (RE)*. IEEE, 211–222.

[39] Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. 2023. Communicative agents for software development. *arXiv preprint arXiv:2307.07924* (2023).

[40] Chen Qian, Yufan Dang, Jiahao Li, Wei Liu, Weize Chen, Cheng Yang, Zhiyuan Liu, and Maosong Sun. 2023. Experiential co-learning of software-developing agents. *arXiv preprint arXiv:2312.17025* (2023).

[41] Zhen Qin, Rolf Jagerman, Kai Hui, Honglei Zhuang, Junru Wu, Jiaming Shen, Tianqi Liu, Jialu Liu, Donald Metzler, Xuanhui Wang, et al. 2023. Large Laguage models are zero effective text rankers with pairwise ranking prompting. *arXiv preprint arXiv:2306.17563* (2023).

[42] Laria Reynolds and Kyle McDonell. 2021. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–7.

[43] Raphael Rodrigues, Leonardo Azevedo, Kate Revoredo, and Henrik Leopold. 2014. A tool to generate natural language text from business process models. *CBSoft 2014-Tool Session* (2014).

[44] Raphael De Almeida Rodrigues, Leonardo Guerreiro Azevedo, and Kate Cerqueira Revoredo. 2016. BPM2Text: A language independent framework for Business Process Models to Natural Language Text. *iSys-Brazilian Journal of Information Systems* 9, 4 (2016), 38–56.

[45] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[46] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2024. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems* 36 (2024).

[47] Saad Shafiq, Atif Mashkoor, Christoph Mayr-Dorn, and Alexander Egyed. 2021. A literature review of using machine learning in software development life cycle stages. *IEEE Access* 9 (2021), 140896–140920.

[48] S P Sharan, Francesco Pittaluga, Vijay Kumar B G, and Manmohan Chandraker. 2023. LLM-Assist: Enhancing Closed-Loop Planning with Language-Based Reasoning. arXiv:2401.00125 [cs.AI]

[49] S Shylesh. 2017. A study of software development life cycle process models. In *National Conference on Reinventing Opportunities in Management, IT, and Social Sciences*. 534–541.

[50] Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M Sadler, Wei-Lun Chao, and Yu Su. 2023. Llm-planner: Few-shot grounded planning for embodied agents with large language models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2998–3009.

[51] Thomas Stober and Uwe Hansmann. 2010. *Best practices for large software development projects*. Springer.

[52] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[53] Wil Van der Aalst. 2011. Process Mining: Discovery. *Conformance and Enhancement of Business Processes* (2011).

[54] Wil Van Der Aalst. 2012. Process mining: Overview and opportunities. *ACM Transactions on Management Information Systems (TMIS)* 3, 2 (2012), 1–17.

[55] Wil Van der Aalst, Ton Weijters, and Laura Maruster. 2004. Workflow mining: Discovering process models from event logs. *IEEE transactions on knowledge and data engineering* 16, 9 (2004), 1128–1142.

[56] Wil MP Van der Aalst, AK Alves De Medeiros, and Anton JMM Weijters. 2005. Genetic process mining. In *Applications and Theory of Petri Nets 2005: 26th International Conference, ICATPN 2005, Miami, USA, June 20-25, 2005. Proceedings 26*. Springer, 48–69.

[57] Boudewijn F van Dongen, Josep Carmona, and Thomas Chatain. 2016. A unified approach for measuring precision and generalization based on anti-alignments. In *International conference on business process management*. Springer, 39–56.

[58] Chengcheng Wan, Shicheng Liu, Sophie Xie, Yifan Liu, Henry Hoffmann, Michael Maire, and Shan Lu. 2022. Automated testing of software that uses machine learning APIs. In *Proceedings of the 44th International Conference on Software Engineering*. 212–224.

[59] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291* (2023).

[60] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382* (2023).

[61] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. *arXiv preprint arXiv:2303.07839* (2023).

[62] Jing Xie and Chen-Ching Liu. 2017. Multi-agent systems and their applications. *Journal of International Council on Electrical Engineering* 7, 1 (2017), 188–197.

[63] Zeyu Zhang, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Quanyu Dai, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. 2024. A Survey on the Memory Mechanism of Large Language Model based Agents. *arXiv preprint arXiv:2404.13501* (2024).

[64] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223* (2023).

[65] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910* (2022).

[66] Yuqi Zhu, Jia Li, Ge Li, YunFei Zhao, Jia Li, Zhi Jin, and Hong Mei. 2023. Hot or Cold? Adaptive Temperature Sampling for Code Generation with Large Language Models. arXiv:2309.02772 [cs.SE]