

RTADev: Intention Aligned Multi-Agent Framework for Software Development

Jie Liu^{1,2}, Guohua Wang^{1,2}, Ronghui Yang^{1,2},
Jiajie Zeng^{1,2}, Mengchen Zhao^{1,2*}, Yi Cai^{1,2*}

¹Key Laboratory of Big Data and Intelligent Robot (SCUT), MOE of China

²School of Software Engineering, South China University of Technology

Correspondence: {zzmc, ycai}@scut.edu.cn

Abstract

LLM-based Multi-agent frameworks have shown a great potential in solving real-world software development tasks, where the agents of different roles can communicate much more efficiently than humans. Despite their efficiency, LLM-based agents can hardly fully understand each other, which frequently causes errors during the development process. Moreover, the accumulation of errors could easily lead to the failure of the whole project. In order to reduce such errors, we introduce an intention aligned multi-agent framework RTADev, which utilizes a self-correction mechanism to ensure that all agents work based on a consensus. RTADev mimics human teams where individuals are free to start meetings anytime for reaching agreement. Specifically, RTADev integrates an alignment checking phase and a conditional ad hoc group review phase, so that the errors can be effectively reduced with minimum agent communications. Our experiments on various software development tasks show that RTADev significantly improves the quality of generated software code in terms of executability, structural and functional completeness. The code of our project is available at <https://github.com/codeagent-rl/RTADev>.

1 Introduction

Large Language Models (LLMs) have been demonstrated to be effective in various code generation tasks and become efficient assistants to human developers (Austin et al., 2021; Chen et al., 2021b). However, single LLM performs poorly when facing complex software development projects, where the logical relationships of different modules could be super complicated. A natural extension is to mimic human software development by introducing multiple role-playing LLM-based agents, so that a complex software development task can be decomposed into a sequence of tasks, including

requirement analysis, architecture design and code generation. Benefiting from the fast speed of content generation and communication, LLM-based multi-agent systems have become promising solutions to automated software development.

Despite their efficiency, LLM-based agents often misunderstand the goal of the task as well as the intentions of other agents, causing considerable errors during the development process. For example, the architect agent might ignore some requirements generated by the product manager and draw an incomplete architecture diagram. Also, the programmer agent might misunderstand some functional dependencies in the architecture diagram and write erroneous codes. Even worse, the errors at the early stages could accumulate and significantly influence the final outputs of the project. We summarize these errors as *misalignment-oriented errors*, since the agents' understandings of the task are not aligned with each other. Figure 1 gives a concrete example to illustrate how the errors accumulate and result in the failure of a project.

One straightforward way to reduce the misalignment-oriented errors is to improve the reasoning ability of the agents, so that they understand the context more accurately. However, it is still hard to ensure alignment between agents given that even humans misunderstand each other frequently. Another possible solution is to design effective multi-agent frameworks that encourage communication between agents. For example, MetaGPT encodes Standardized Operating Procedures into prompt sequences for a streamlined workflow, where a test engineer is employed at the end of development process (Hong et al., 2024). ChatDev introduces a chat-powered software development framework, in which several assistant agents are employed for code reviews (Qian et al., 2023). Unfortunately, these frameworks adopt fixed local communication protocols, which do not guarantee the alignment of all agents.

*These authors are co-corresponding authors.

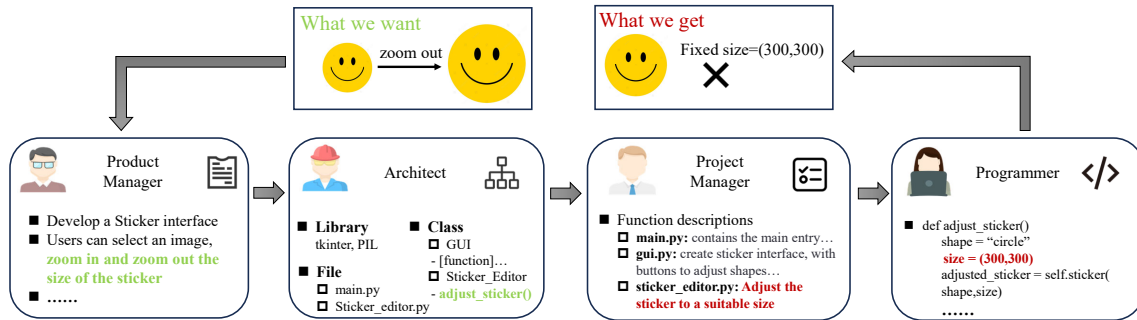


Figure 1: An example illustrating the misalignment during the software development process. The original requirement is a sticker that can be zoom in and zoom out by users. Unfortunately, the project manager misunderstands the requirement and writes a misleading function description (shown in red). Consequently, the programmer assigns a fixed size to the sticker in her codes.

In this paper, we introduce a flexible multi-agent framework called RTADev, which aims to align the intentions of all agents so that they can work based on a consensus. Following existing works (Adetokunbo and Adenowo, 2013; Hong et al., 2024), RTADev employs five role-playing LLM-based agents, who work collaboratively to develop specific software. Specifically, given a description of requirements, the *product manager* outputs a detailed and numerated requirement document. Then, the *architect* translates the requirement document into an architecture diagram. Hence, a *project manager* sets out a task plan for the *programmer* to execute. Finally, a *test engineer* will verify whether the program meets the original requirements.

To address the misalignment during the development process, RTADev introduces an effective real-time alignment (RTA) mechanism, inspired by human development teams where people can start meetings anytime to reach an agreement. Specifically, RTADev maintains a Shared Certified Repository (SCR) to store the certified deliverables, which represent agents’ consensus on the target software. Whenever a new deliverable is generated, it will go through an alignment checking phase before being added to the SCR. The alignment checking fails when at least one agent believes that the newly generated deliverable violates the certified consensus. In such cases, the agents who vote for violation will form an ad hoc team and solve the issues together with the agent responsible for the current deliverable. The above processes will repeat until the newly generated deliverable passes the alignment checking. Note that RTADev can effectively reduce errors in the development process while requires minimum agent communi-

cations, thus it successfully balances the quality and the efficiency. Our contributions can be summarized as follows:

- We propose a novel and flexible LLM-based multi-agent framework RTADev, which ensures that all agents work based on common understandings of the target software. Compared with existing LLM-based software development frameworks, RTADev is more robust to errors caused by LLMs and requires minimum agent communications.
- We propose a checkpoint-based prompting method for alignment checking, which efficiently guides different agents to check if the current deliverable aligns with the certified consensus. We standardize all the prompt sequences and provide them in Appendix A.7 for reproduction of RTADev.
- We construct a functionality-driven benchmark FSD-Bench to evaluate RTADev, which contains 120 realistic software development tasks from various domains. Experimental results show that RTADev largely improves the quality of generated codes in terms of different metrics.

2 Related Works

2.1 Human Software Development frameworks

Through decades of practice, people have developed many mature software development frameworks regarding to various scenarios. For example, the Waterfall model is a sequential and linear software development process that breaks down the software development lifecycle into distinct, discrete phases (Adetokunbo and Adenowo, 2013).

The V-model is an evolution of the traditional Waterfall model, which introduces parallelism and feedback loops between development and testing activities (Sundramoorthy and Murugaiyan, 2012). The Agile model is a flexible and iterative approach that emphasizes rapid delivery of high-quality software through close collaboration and continuous improvement (Samar et al., 2020). In practice, the implementation of these models heavily depends on human intelligence, with the assumption that humans could understand each other through various ways of communication. Therefore, these models do not directly apply to LLM-based agents. Our framework RTADev fills this gap by designing an effective real-time alignment mechanism, under which the LLM-based agents could cooperate based on common understandings.

2.2 LLMs for Code Generation

Large Language Models (LLMs) have shown surprising performance across a wide range of natural language processing tasks (Radford et al., 2019; Ouyang et al., 2022; Achiam et al., 2023; Touvron et al., 2023; Yuan et al., 2023, 2025). LLMs also significantly boost the performance of code generation tasks and demonstrate promising potential to be applied at scale. For example, Codex achieves a success rate of 28.8% in solving a set of 164 hand-written programming problems (Chen et al., 2021a). Copilot, a code generation tool powered by Codex, has captured the interest of over 1 million professional developers (Microsoft, 2023). Other models, including InCoder (Fried et al., 2023), CodeLlama (Lea et al., 2022), CodeLlama (Roziere et al., 2023) and ChatGPT (OpenAI, 2022), also achieve human-level performance in various code generation tasks. However, these works focus only on code generation ability of single LLMs, ignoring the collaboration and communication between multiple agents, which is crucial to complex software development tasks.

2.3 Multi-Agent Frameworks for Software Development

The key challenge in multi-agent software development is to efficiently coordinate with diverse roles of agents while ensuring consistent understanding of the outcomes (Unterkalmsteiner et al., 2015; Bjarnason et al., 2019). Several multi-LLM software development frameworks have been proposed to enhance collaboration between agents. For example, the Self-Collaboration framework assigns

different roles to LLMs and encourages them to finish sub-tasks collaboratively (Dong et al., 2024). MetaGPT proposes the first concrete and standardized software development framework, where multiple LLM-based agents complete different tasks following a linear workflow (Hong et al., 2024). However, the linear nature of Self-Collaboration and MetaGPT prohibits the agents to communicate with previous agents, thus the errors can easily accumulate. ChatDev introduces a set of assistant agents to verify if the generated codes are compliant with original requirements (Qian et al., 2023). EvoMAC introduces a self-evolving multi-agent framework where the feedback can back propagate to each working agent (Hu et al., 2024). However, these frameworks follow fixed feedback procedures, thus are usually inefficient dealing with frequently occurring errors.

3 Methodology

In this section, we present the details of our framework RTADev. Generally, RTADev comprises a general workflow and a real-time alignment mechanism, which includes an alignment checking phase and a conditional ad hoc group review phase.

3.1 General Workflow

For simplicity, our general workflow follows a typical software development process, where several important roles of agents work in a linear manner. Such a Waterfall-like model has also been employed in many LLM-based software development frameworks (Hong et al., 2024; Qian et al., 2023). Note that although our general workflow is linear, the real-time alignment (RTA) mechanism requires participation of previous agents, which makes RTADev actually a non-linear model. Also, the RTA mechanism is compatible with non-linear workflows such as Agile (Samar et al., 2020), because the alignment checking phase can be flexibly placed in any workflows. As shown in Figure 2, RTADev employs the following five role-playing LLM-based agents.

Product Manager. The Product Manager agent performs requirement analysis based on the raw functional description of some target software (Arora et al., 2024; Jin et al., 2024). In our framework, we ask the Product Manager agent to output the Product Requirement Document (PRD) as a list of numerated requirements that describe the

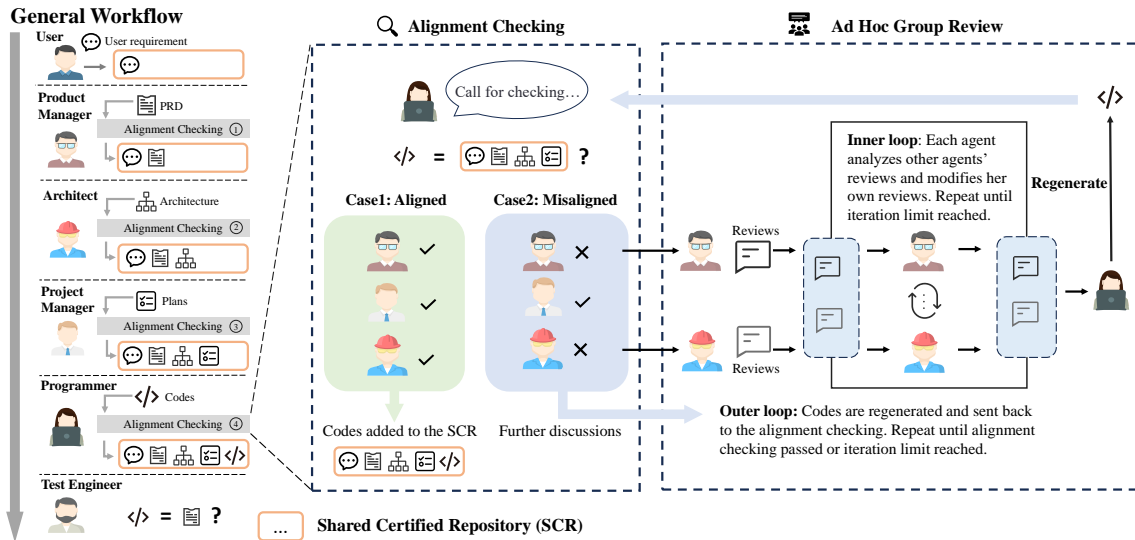


Figure 2: The overall framework of RTADev. The left part shows the general workflow, where five LLM-based agents work in a sequential manner to accomplish requirements inputted by the user. The middle part shows the alignment checking phase, in which all previous agents need to check if the newly generated deliverable is consistent with previous ones stored in the SCR. The right part shows a conditional ad hoc group review phase, depending on whether the newly generated deliverable passes the alignment checking. After the group review, the current agent regenerates deliverable for another round of alignment checking. The outer loop continues until the regenerated deliverable passes the alignment checking or the limit of iterations is reached. We only showcase the alignment checking and the ad hoc group review initiated by the Programmer agent due to space limitation.

functionalities of the target software.

Architect. The Architect agent analyzes the PRD and determines the general architecture of the software, including the technology stack, the relationships between classes and the Graphical User Interface (GUI) if necessary. In our framework, the Architect agent writes the architecture in JSON format, which can be converted to Unified Modeling Language (UML) diagrams using the Mermaid tool (Sveidqvist and Contributors to Mermaid, 2014).

Project Manager. This Project Manager agent schedules a code plan based on the PRD and the architecture diagram. For simplicity, the code plan is represented by a list of files to be created. We also ask the Project Manager agent to explicitly match all the requirements in the PRD with the files in the code plan, in order to ensure that all the requirements are considered.

Programmer. Unlike GPT-Engineer (Osika., 2023) and ChatDev (Qian et al., 2023) where the agent generates code solely based on the raw description of requirements, our Programmer agent generates code based on the PRD, the architecture diagram and the code plan outputted by previous agents. Benefiting from task decomposition and scheduling, the Programmer agent can generate code for much more complex software.

Test Engineer. Existing works have explored using LLMs for software testing from various perspectives, including unit test case generation (Li and Yuan, 2024) and GUI testing (Liu et al., 2024). In our framework, the Test Engineer agent focuses on general metrics such as structural completeness, executability and functional completeness. Detailed testing procedures are elaborated in Section 5.

Shared certified repository. Our framework maintains a shared certified repository (SCR) to store intermediate deliverables, including PRD, architecture diagrams, task plans and codes. Each newly generated deliverable will go through an alignment checking procedure before being added to the SCR. In this way, SCR actually represents the agents’ common understandings of the software and provides basis for subsequent activities. At each phase in the general workflow, the working agent retrieves all items in the SCR and generate new deliverable based on them. At the alignment checking and the ad hoc group review phases, each agent only retrieves the item generated by herself, so that each agent could concentrate on a specific perspective.

3.2 Real-Time Alignment Mechanism

It is very common in real-world software development that team members have different under-

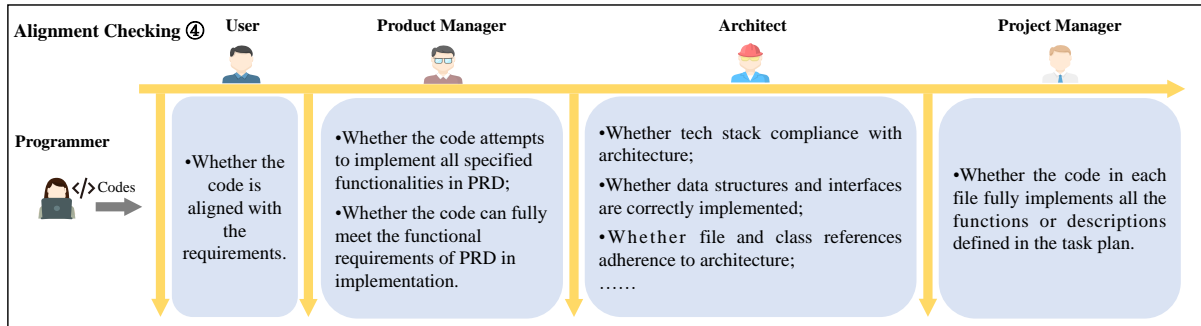


Figure 3: Illustration on the alignment checking phase initiated by the Programmer agent. The horizontal arrow indicates the order of agents to be queried and the vertical arrows indicate the order of checkpoints in the querying prompts. Illustrations of other alignment checking phases are shown in Appendix A.7. Note that the user involvement is optional, depending on whether human instructions are allowed in practice.

standings of the task, which might significantly impede the development process. Through decades of practice, people have developed many solutions to address the misalignment of understanding. For example, a quick group meeting can help team members to align their opinions when some problems arise. Such an alignment mechanism in teamwork is key to software development since the misalignment-oriented errors frequently occur during the development process.

The design of our real-time alignment (RTA) mechanism follows two principles. First, each newly generated deliverable should be checked before being used, thus any occurring errors can be reduced in real time. Second, for the sake of efficiency, we should keep the minimum number of participants in reviews and discussions. This is extremely important when more agents are included to solve more complex tasks.

3.2.1 Alignment Checking

The goal of alignment checking is to ensure that the newly generated deliverable correctly realizes the requirements and is consistent with the intermediate deliverables in the SCR. As shown in Figure 2, our framework includes four alignment checking phases. In each alignment checking phase, we call the agent responsible for the current generation phase *initiator*, and all the previous agents *supervisors*. For example, in the third alignment checking phase, the Project Manager serves as the initiator, while the Product Manager and the Architect serve as supervisors. Note that the user can optionally participate in the alignment checking, depending on the practical constraints.

During each alignment checking phase, the initiator sends the newly generated deliverable to all

supervisors. Then, each supervisor retrieves their outcomes from the SCR and checks whether the newly generated deliverable is consistent with their previous outcomes. Note that the alignment checking requires sophisticated logical reasoning, which is still a big challenge for even most advanced LLMs. Therefore, if we directly ask an LLM if multiple files are logically consistent, we probably cannot get a reliable result.

To this end, we propose a checkpoint-based prompting method to guide the alignment checking process, inspired by the Chain-of-Thought (CoT) method (Wei et al., 2022). The key idea of our checkpoint-based method is to decompose each checking task into several important checkpoints. In each alignment checking phase, the supervisors are queried sequentially following the horizontal arrow. When a supervisor is being queried, she will retrieve the deliverable generated by herself from the SCR and check if it is consistent with the querying deliverable following the checkpoints along the vertical arrow. The supervisor will approve the querying deliverable only when all the checkpoints along the vertical arrow are met. Figure 3 gives an example of alignment checking initiated by the Programmer agent. If all supervisors approve the querying deliverable, it will be added to the SCR. Otherwise, an ad hoc group review phase will be initiated to resolve the misalignment.

A concrete prompt includes the querying deliverable, the deliverable retrieved by the supervisor, the necessary checking points and a counter-example explaining failures on meeting the checkpoints. More details on the structure of the prompts can be found in Appendix A.7.

3.2.2 Ad Hoc Group Review

In human development teams, a quick discussion or meeting is perhaps the most efficient way to align with each other. Existing works have shown that discussions in LLM-based multi-agent communities also help resolve conflicts and accelerate teamwork (Park et al., 2023; An et al., 2024). Note that in the above alignment checking process, each supervisor checks the initiator’s deliverable only from their own perspectives. Since there might be conflicts in the supervisors’ reviews, the initiator would be confused when regenerating their deliverable. Therefore, a group review phase can help to form clearer feedback to the initiator.

We call it an ad hoc group review because only supervisors who vote for misalignment will participate in the group review, thus the group is formed in an ad hoc manner. There are two motivations behind this design. First, we try to keep the minimum number of participants in the discussion, which also accords with efficiency principles of human meetings. Second, if the group incorporates too many agents, it is hard to reach a consensus. Note that our framework can be extended by employing more agents to play each role, in which cases the ad hoc design could largely improve the efficiency.

Specifically, there are two loops in the ad hoc review phase, as shown in Figure 2. In the inner loop, each agent analyzes other agents’ reviews and modifies their own reviews. In the outer loop, the initiator regenerates their deliverable based on the final reviews outputted by the supervisors after discussion, until the regenerated deliverable passes the alignment checking or the maximum number of iterations is reached. We will show how to determine the maximum number of iterations of both inner and outer loops in Appendix A.4.

3.3 Adaptation to Agile Frameworks

Although we adopt a Waterfall-like development process as the general workflow, the RTA mechanism can also be adapted to Agile frameworks such as Scrum (Samar et al., 2020). The extension can be done by performing alignment checking and group review on milestone deliverables. In fact, the RTA itself reflects the spirits of Agile.

4 FSD-Bench: Functionality-Driven Software Development Benchmark

Existing work has proposed some software development datasets based on simple instructions, such

as SRDD (Qian et al., 2023), 50days50projects (Zhang et al., 2024), and SoftwareDev (Hong et al., 2024). These datasets provide a comprehensive corpus of textual software requirements. However, we thoroughly reviewed the description of each software requirement and noticed several issues. Firstly, many descriptions are incomplete, ending with ellipses. Secondly, some tasks are almost impossible for current LLM-based software development, such as the creation of a video game. Third, these datasets lack effective test cases for evaluation. To this end, we develop a new Functionality-driven Software Development Benchmark (**FSD-Bench**), which covers three types of tasks: Website, Desktop Application and Game development.

4.1 Benchmark Construction

FSD-Bench contains 120 realistic tasks selected from SRDD and 882 pieces of functional descriptions of the tasks. In addition, we design 1195 test cases to comprehensively verify if the developed software accomplishes the functionalities.

User Requirements. The requirements for each software consist of four key components, including Software Description, Core Features, Programming Language, and Data Storage. Software Description describes an overview of the task, outlining the objectives and expected software. The Core Feature lists the feature that the software must contain in the form of sub-points. The Programming Language describes the user’s high-level technical requirements, indicating whether the software will be a website, desktop application, or game. Finally, the Data Storage briefly describes the manner in which the software’s data will be stored, with the use of local files specified as a compromise to facilitate batch automated testing.

Test Cases. Each user requirements document is paired with a test cases document. We design several test cases for each feature using Step-Expected format, where Step denotes the sequence of operations and Expected denotes the expected result of each sequence of operations. We combined manual design and LLM generation to create each test case document, where we first manually design test cases for the most critical core features and then prompt the LLM (GPT-4o) with the software requirements to generate additional test cases for other core features. More details about the FSD-Bench can be found in Appendix A.3.

Paradigm	Method	SC (%)	Exec. (%)	FSD-Bench				Tokens	Time (s)	HumanEval (%)
				FC (%)			Pass@1			
				Website	Desktop	Game	Average			
Single-Agent	GPT-4o-Mini	89.17	88.33	39.14	38.57	33.98	37.23	1953.23	14.806	87.20
	GPT-Engineer	75.83	80.00	29.14	38.57	29.13	32.28	5919.57	23.044	88.41
Multi-Agent	AutoGen	84.17	85.83	42.57	37.67	27.83	36.02	6345.60	24.559	85.36
	MetaGPT	77.50	85.83	19.71	48.43	35.92	34.69	44122.05	67.582	87.20
	ChatDev	81.67	95.83	30.57	49.78	42.71	41.02	39318.98	389.118	86.59
	RTADev	90.00	97.60	55.14	62.78	73.54	63.83	70652.60	143.770	91.46
		+0.93	+1.85	+29.53	+26.11	+72.18	+55.61	-	-	+3.45

Table 1: Overall performance of RTADev and baselines. Performance metrics are averaged for all tasks. The best results are shown in **bold**. **Red** values represent the relative improvement in percentage brought by RTADev, compared with the second-best results.

5 Experiments

5.1 Experimental Settings

Datasets. Our experiments cover both the proposed **FSD-Bench** and the widely used **HumanEval** dataset, which comprises 164 Python function completion tasks based on given requirements (Chen et al., 2021b).

Baselines. We select **GPT-4o-Mini** and four open-sourced agent-based methods as baselines: **GPT-Engineer** (Osika., 2023), **AutoGen** (Wu et al., 2023), **MetaGPT** (Hong et al., 2024), and **ChatDev** (Qian et al., 2023). These methods represent a diverse set of single-agent and multi-agent frameworks for LLM-based software development. To ensure a fair comparison, all baselines and our RTADev framework are powered by the GPT-4o-Mini. More implementation details can be found in Appendix A.1.

Metrics. We use various metrics to evaluate the code generated by different methods. Specifically, experiments on HumanEval adopt the **pass@1** metric following existing works (Chen et al., 2021b). For experiments on FSD-Bench, the **executability** metric calculates the percentage of tasks whose code run successfully in the compiling environment. **Structural completeness** (SC) is a statistical metric that measures the completeness of code structures. SC is measured by a script that matches unexpected placeholders (e.g., PASS, TODO) using a regular expression. To evaluate how the functionalities in the PRD are implemented, we introduce a new metric **functional completeness** (FC), which is calculated as the ratio of implemented requirements to the total number of requirements. In fact, executability and SC focus on evaluating code quality from a lower

level, while FC focuses on a higher functionality level. We believe that FC is more suitable for complex software development tasks. In addition, we report the total number of **tokens** and **time** consumed during the entire generation process. More details about metrics can be found in Appendix A.2

5.2 Main Results

As shown in Table 1, RTADev performs better than all baseline methods in all metrics, which demonstrates the superiority of our framework. In terms of SC, RTADev achieves the best results because our real-time alignment mechanism can effectively detect codes that are not fully implemented or have placeholders. Also the multi-agent discussion greatly improves structural completeness of regenerated codes. In terms of executability, RTADev also achieves the best results, potentially due to the feedback from the Test Engineer so that the low-level bug can be fixed in real time. FC is the most important metric as it comprehensively measures whether the code meets the functional requirements. ChatDev achieves the second-best result in terms of average FC among the multi-agent methods, benefiting from its code completion phase which involves multiple rounds of code improvement. RTADev outperforms ChatDev by 55.17% in terms of average FC, indicating that the real-time alignment mechanism indeed ensures that all agents work towards the same goal. Moreover, RTADev has significantly improved the FC metric of three types of software development tasks compared with all baselines, which shows that the RTA mechanism is effective and can be used for a variety of software development tasks. We provide some case studies to show the quality of software developed by RTADev in Appendix A.5.

	Architect	Project Manager	Programmer	SC (%)	Exec. (%)	FC (%)			
						Website	Desktop	Game	Average
a)	-	-	-	71.67	81.67	32.62	48.21	41.67	40.83
b)	-	✓	✓	83.33	91.67	57.22	67.16	57.74	60.71
c)	✓	-	✓	85.00	85.00	48.12	63.43	55.36	55.64
d)	✓	✓	-	66.67	83.33	39.57	59.21	49.40	49.06
e)	✓	✓	✓	88.33	100	65.24	70.83	79.10	71.72

Table 2: Ablation study on three important agents with/without RTA. Best performances are in **bold**.

5.3 Computational Cost Analysis

We can see from Table 1 that RTADev consumes more time and tokens than single-agent approaches and MetaGPT, but less than ChatDev. This result is under expectation because RTADev incorporates more agent communications than MetaGPT. Meanwhile, RTADev significantly outperforms all baselines in all metrics. We believe that spending more time to achieve better results is worthwhile, especially considering that the time consumption is still much lower than that of the human developers. Furthermore, in the real-time alignment phase, there are two hyper-parameters that control the maximum number of rounds for alignment checking and ad hoc group review. These parameters allow for balancing between performance and time consumption in practice. We show how to determine these parameters in Appendix A.4.

5.4 Ablation Study

To further evaluate the improvements brought by the RTA mechanism, we implement four variants of RTADev (indexed by a-d in Table 2) by skipping some alignment checking phases. And 60 tasks are selected from FSD-Bench for ablation experiments. We can find that these variants perform worse than RTADev (indexed by e) in all metrics, demonstrating the effectiveness of the RTA mechanism for each agent. Additionally, variant d performs worse than all other variants except variant a, which shows that the RTA mechanism initiated by the programmer is very critical and may have the greatest impact on the quality of the final generated software.

5.5 Misalignment Analysis

As the motivation of RTADev is to reduce the misalignment between agents, we report the number of misaligned deliverables, which is counted by running alignment checking procedures. For example, the blue bar in Figure 4 represents the number

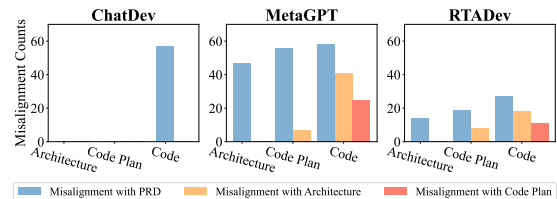


Figure 4: Comparison of misalignment counts.

of cases where the codes are not aligned with the PRD. To make a fair comparison, for MetaGPT and RTADev, we count the number of misalignments across three phases: architecture design, code planning and code writing. Considering that ChatDev only generates code, we only show the number of misalignments between the PRD and the code.

Compared to the baselines, RTADev effectively reduces the misalignment at most phases, indicating that by incorporating the RTA mechanism can efficiently resolve misalignment during the whole process. The only exception is a slight increase (5.6%) in misalignment between architecture and code plan comparing with MetaGPT. This is because that MetaGPT hard encodes a strict correspondence between architecture and code plan, which reduces misalignment manually. Note that the misalignment cannot be completely reduced, possibly due to the hallucination of LLMs.

6 Conclusion

LLM-based multi-agent frameworks have a great potential for automated software development, yet they suffer from misalignment between agents. In this paper, we introduce a novel and flexible multi-agent framework RTADev, which introduces an effective real-time alignment (RTA) mechanism to ensure that all the agents work based on a consensus. We construct a realistic FSD-Bench with extensive unit test cases to evaluate the functional completeness of software. Benefiting from the RTA mechanism, RTADev significantly outperforms strong baselines, demonstrating its superiority and potential in complex software development tasks.

Limitations

Our work faces following limitations. First, although RTADev reduces the misalignment-oriented errors from a mechanism design perspective, such errors may still exist. We hypothesis that it is due to the limitation of the agents' reasoning ability. Perhaps a stronger base model than the GPT-4o-Mini we used would mitigate this issue. Second, there is still a gap between the FSD-Bench we constructed and industrial development tasks. However, we believe that FSD-Bench is suitable for evaluating current LLM-based code generation methods, since even the best method achieves less than 75% in terms of functional completeness, as is shown in Table 1. Third, in our experiments, each role in RTADev is played by a single agent. A natural extension is to employ more agents to perform each development tasks. However, employing more agents might increase the computational cost.

Acknowledgments

This research is supported by the Science and Technology Planning Project of Guangdong Province (2020B0101100002), the Fundamental Research Funds for the Central Universities, South China University of Technology (x2rjD2240100), Guangdong Provincial Fund for Basic and Applied Basic Research—Regional Joint Fund Project (Key Project) (2023B1515120078), Guangdong Provincial Natural Science Foundation for Outstanding Youth Team Project (2024B1515040010), Guangdong Basic and Applied Basic Research Foundation (2025A1515010247), and the Fundamental Research Funds for the Central Universities (2024ZYGXZR069).

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Denowo Adetokunbo and Basirat Adenowo. 2013. Software engineering methodologies: a review of the waterfall model and object-oriented approach. *International Journal of Scientific Engineering Research*, pages 427–434.

Zhang An, Yuxin Chen, Leheng Sheng, Xiang Wang, and Tat-Seng Chua. 2024. On generative agents in recommendation. In *Proceedings of the 47th international ACM SIGIR conference on research and*

development in Information Retrieval, pages 1807–1817.

- Chetan Arora, John Grundy, and Mohamed Abdelrazek. 2024. Advancing requirements engineering through generative ai: Assessing the role of llms. In *Generative AI for Effective Software Development*, pages 129–148. Springer.
- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.
- Elizabeth Bjarnason, Helen Sharp, and Björn Regnell. 2019. Improving requirements-test alignment by prescribing practices that mitigate communication gaps. *Empirical Software Engineering*, 24:2364–2409.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, and Greg Brockman. 2021a. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021b. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2024. Self-collaboration code generation via chatgpt. *ACM Transactions on Software Engineering and Methodology*.
- Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. Incoder: A generative model for code infilling and synthesis. In *International Conference on Learning Representations*.
- Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. *MetaGPT: Meta programming for a multi-agent collaborative framework*. In *The Twelfth International Conference on Learning Representations*.
- Yue Hu, Yuzhu Cai, Yaxin Du, Xinyu Zhu, Xiangrui Liu, Zijie Yu, Yuchen Hou, Shuo Tang, and Siheng Chen. 2024. Self-evolving multi-agent collaboration networks for software development. In *arXiv preprint arXiv:2410.16946*.
- Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2024. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*.

- Hung Lea, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 21314–21328.
- Kefan Li and Yuan Yuan. 2024. Large language models as test case generators: Performance evaluation and enhancement. *arXiv preprint arXiv:2404.13340*.
- Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2024. Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.
- Microsoft. 2023. Microsoft attracting users to its code-writing generative ai software. <https://www.euronews.com/next/2023/01/25/microsoft-results-ai>.
- OpenAI. 2022. Chatgpt: Optimizing language models for dialogue. <https://openai.com/index/chatgpt/>.
- Anton Osika. 2023. [Gpt-engineer](#).
- Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744.
- 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*, pages 1–22.
- 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*.
- Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.
- Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, and Jer´emy Rapin. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.
- Al-Saqqa Samar, Samer Sawalha, and Hiba AbdelNabi. 2020. Agile software development: Methodologies and trends. *International Journal of Interactive Mobile Technologies*.
- Balaji Sundramoorthy and Sundararajan Murugaiyan. 2012. Waterfall vs. v-model vs. agile: A comparative study on sdlc. *International Journal of Information Technology and Business Management*, pages 26–30.
- Knut Sveidqvist and Contributors to Mermaid. 2014. [Mermaid: Generate diagrams from markdown-like text](#).
- 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*.
- Michael Unterkalmsteiner, Tony Gorschek, Robert Feldt, and Eriks Klotins. 2015. [Assessing requirements engineering and software test alignment—five case studies](#). *Journal of Systems and Software*, 109:62–77.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.
- Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. Auto-gen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*.
- Li Yuan, Yi Cai, Jin Wang, and Qing Li. 2023. Joint multimodal entity-relation extraction based on edge-enhanced graph alignment network and word-pair relation tagging. In *Proceedings of the AAAI conference on artificial intelligence*, volume 37, pages 11051–11059.
- Li Yuan, Yi Cai, Jingyu Xu, Qing Li, and Tao Wang. 2025. [A fine-grained network for joint multimodal entity-relation extraction](#). *IEEE Transactions on Knowledge and Data Engineering*, 37(1):1–14.
- Sai Zhang, Zhenchang Xing, Ronghui Guo, Fangzhou Xu, Lei Chen, Zhaoyuan Zhang, Xiaowang Zhang, Zhiyong Feng, and Zhiqiang Zhuang. 2024. Empowering agile-based generative software development through human-ai teamwork. *arXiv preprint arXiv:2407.15568*.

A Appendix

A.1 Implementation Details

In order to reduce human factors in the experimental results, we skip the steps in alignment checking where human inputs are required. Although effective human feedback may help to improve the performance of our methods, we believe that removing human factors leads to a fairer comparison.

In addition, we set the maximum number of iterations as 1 for the inner loop, and 3 for the outer loop in the real-time alignment mechanism. We use GPT-4o-Mini with a temperature of 0.2. All baselines in the experiments share the same hyperparameters and settings.

A.2 Metrics Details

Following existing works (Qian et al., 2023; Hong et al., 2024), we use structural completeness (SC) and executability to evaluate RTADev. In addition, we propose a new metric called functional completeness (FC) to evaluate how the developed software meets the functional requirements. The metrics are elaborated as follows.

- *Structural Completeness* (SC) is a statistical metric that measures the completeness of code structures. In practice, codes generated by LLMs are usually incomplete, where some important classes and functions are replaced by placeholders (e.g., PASS, TODO). For each software development task, we flag it as structural complete if the generated codes do not contain any placeholders, and incomplete otherwise. The SC of a set of tasks is calculated as the percentage of structurally complete tasks. A higher SC score of a framework indicates a better ability to generate complete code. SC is measured by a script that matches placeholders using a regular expression.
- *Executability* measures whether the generated codes could run successfully within the compiling environment, regardless of whether the functional requirements are met. In other words, the executability focuses on checking if there are low-level bugs in the code. We calculate this metric as the percentage of tasks that compile and run successfully. A higher score indicates better ability of the framework to generate bug-free code.
- *Functional Completeness* (FC) measures how the software meets the functional requirements. The FC of a task is calculated as the percentage of the implemented requirements to total requirements. Compared with SC and executability metrics, FC is a more advanced metric because it measures the quality of generated code from a higher level. We perform unit testing for generated software according to the test cases in the FSD-Bench to verify whether functionalities are implemented.

A.3 Benchmark Details

The Software Requirement Description Dataset (SRDD) introduced by ChatDev represents a comprehensive corpus of textual software requirements, specifically curated to facilitate agent-driven software development. The dataset comprises 1,200 discrete software tasks, which are classified into five principal domains: Education, Work, Life, Games, and Creation. The dataset incorporates software descriptions from major platforms, including Ubuntu, Google Play, Microsoft Store, and Apple Store, thereby providing a diverse and representative sample of software requirements across various domains. However, we thoroughly reviewed the description of each software requirement and noticed several issues. Firstly, many descriptions are incomplete, ending with ellipses. Secondly, some tasks are almost impossible for current LLM-based software development, such as the creation of a video game. Third, SRDD lacks effective test cases for evaluation. To this end, we develop a new Functionality-driven Software Development Benchmark (**FSD-Bench**), which covers three main types of software development: Website, Desktop Application and Game development. FSD-Bench contains 120 realistic tasks selected from SRDD and 882 pieces of functional descriptions of the tasks. We also provide an overview of website, desktop application, and game in Table 3, 4, and 5 respectively. In addition, we design 1195 test cases to comprehensively verify if the software accomplish the functionalities.

Part 1: User Requirements. As shown in Figure 5, the requirements for each software consist of four key components, including Software Description, Core Features, Programming Language (Technology Stack), and Data Storage. The first component is the Software Description, which describes an overview of the task, outlining the objectives and expected software. The Core Feature, on the other hand, lists the feature that the software must contain in the form of sub-points. The Programming Language (Technology Stack) describes the user’s high-level technical requirements, indicating whether the software will be a website, desktop application, or game. Finally, the Data Storage section briefly describes the manner in which the software’s data will be stored, with the use of local files specified as a compromise to facilitate batch

automated testing.

Part 2: Test Cases. As shown in Figure 6, each user requirement document is paired with a test case document. We design several test cases for each feature using Step-Expected format, where 'Step' denotes the sequence of operations and 'Expected' denotes the expected result of each sequence of operations. We combined manual design and Large Language Model (LLM) generation to create each test case document, where we first manually design test cases for the most critical core features and then prompt the LLM(GPT-4o) with the software requirements to generate additional test cases for other core features.

Part 3: Unit Testing. As shown in Figure 7, upon completion of the code generation task, the test cases and codebase are provided to the LLM (GPT-4o) to generate unit test code. We then manually review and correct the generated unit test code to address any potential errors. The LLM is prompted to generate unit test functions on a feature-by-feature basis, with features containing multiple test cases written into the same unit test function. If a feature is not implemented by the software, a unit test function that simply returns a failure will be generated. Finally, we execute the unit test code in batches to evaluate the completeness of the software's core features.

Website		
DailyJournalApp	EcoFriendlyLivingTips	FreelancerMarketplace
GreenLivingGuide	OnlineShoppingCenter	OnlineVintageMarket
ParentingAdviceForum	RemoteJobBoard	TaskManager

Table 3: 9 Website Software in FSD-Bench.

Desktop Application		
BookshelfManager	DataSummarizer	ExpensePlanner
InvestmentTracker	OfficeStockManager	PhotoStickerMaker
ScienceLibrary	ShapeMaster	ShoppingPlanner

Table 4: 9 Desktop Applications Software in FSD-Bench.

Game				
Balls	BlockConnect	Gomoku	Brick	ColorLinkPuzzle
2048	JigsawMania	TilePlacer	Tank	NumberMystery

Table 5: 10 Game Software in FSD-Bench.

User Input

Software Description

Objective

Image Enhancer is a photo software application for enhancing the quality and appearance of images. It provides a range of editing tools, including brightness, contrast, and saturation adjustment, along with filters and effects to enhance colors and tones. Users can also perform basic cropping and resizing. The software aims to provide a simple yet powerful tool to enhance and improve photos.

Core Features

1. Import and select a image.
2. Adjust brightness of images.
3. Adjust contrast of images.
4. Adjust saturation of images.
5. Apply filters to images.
6. Apply effects to enhance colors and tones.
7. Crop images.
8. Resize images.

Language

Use python to develop a application. You can use tkinter library to build graphical user interfaces.

Data Storage

Data will be stored in local files.

Figure 5: User requirement of Image Enhancer.

<p>### Functionality 1. Import and Select an Image</p> <hr/> <p>Step Import a valid image file (e.g., JPEG, PNG).</p> <p>Expectation The image loads successfully and is displayed in the editor.</p>
<p>### Functionality 2. Adjust Brightness of Images</p> <hr/> <p>Step Increase the brightness of an image to maximum.</p> <p>Expectation The image becomes brighter, with visible changes in luminance.</p> <hr/> <p>Step Increase the brightness of an image to maximum.</p> <p>Expectation The image becomes brighter, with visible changes in luminance.</p>
<p>### Functionality 4. Adjust Saturation of Images</p> <hr/> <p>Step Increase saturation to maximum.</p> <p>Expectation Colors in the image appear more vivid and intense.</p> <hr/> <p>Step Decrease saturation to minimum.</p> <p>Expectation The image becomes grayscale.</p>
<p>### Functionality 3. Adjust Contrast of Images</p> <hr/> <p>Step Increase contrast to maximum.</p> <p>Expectation The difference between light and dark areas becomes more pronounced.</p> <hr/> <p>Step Decrease contrast to minimum.</p> <p>Expectation The image appears flat with reduced distinction between light and dark areas.</p>
<p>### Functionality 5. Apply Filters to Images</p> <hr/> <p>Step Select and apply a predefined filter (e.g., "Sepia").</p> <p>Expectation The filter is applied, and the image reflects the changes.</p> <hr/> <p>Step Apply multiple filters sequentially.</p> <p>Expectation Filters are layered, and the image reflects combined effects.</p>
<p>### Functionality 6. Apply Effects to Enhance Colors and Tones</p> <hr/> <p>.....</p>
<p>### Functionality 7. Crop Images</p> <hr/> <p>.....</p>
<p>### Functionality 8. Resize Images</p> <hr/> <p>.....</p>

Test Image File
Relative path:
./lena.jpeg

Figure 6: Test cases for each feature of Image Enhancer in step-expected format.

```

import unittest
from unittest.mock import patch
from PIL import Image
import os
from main import Main
from image_enhancer import ImageEnhancer

class TestImageEnhancer(unittest.TestCase):

    def setUp(self):
        """Set up the initial conditions for testing."""
        self.app = Main()
        self.enhancer = self.app.enhancer
        self.test_image_path = "./lena.jpeg"

        # Ensure test image exists
        if not os.path.exists(self.test_image_path):
            raise FileNotFoundError(f"Test image {self.test_image_path} not found.")

        # Load the test image
        self.enhancer.load_image(self.test_image_path)

    def test_adjust_brightness(self):
        """Test brightness adjustment."""
        # Test increasing brightness
        initial_image = self.enhancer.image.copy()
        self.enhancer.adjust_brightness(2.0) # Max brightness
        self.assertNotEqual(
            initial_image, self.enhancer.image, "Brightness adjustment failed."
        )

        # Test decreasing brightness
        initial_image = self.enhancer.image.copy()
        self.enhancer.adjust_brightness(0.0) # Min brightness
        self.assertNotEqual(
            initial_image, self.enhancer.image, "Brightness adjustment failed."
        )

    def test_adjust_contrast(self):
        """Test contrast adjustment."""
        # Test increasing contrast
        initial_image = self.enhancer.image.copy()
        self.enhancer.adjust_contrast(2.0) # Max contrast
        self.assertNotEqual(
            initial_image, self.enhancer.image, "Contrast adjustment failed."
        )

    def test_adjust_saturation(self):
        .....

    def test_apply_filter(self):
        .....

    def test_apply_grayscale_effect(self):
        .....

    def test_apply_sepia_effect(self):
        .....

    def test_resize_image(self):
        .....

    @patch("builtins.input", return_value="100")
    def test_crop_interactive_input(self, mock_input):
        .....

    @patch("builtins.input", return_value="400")
    def test_resize_interactive_input(self, mock_input):
        .....

    def test_save_image(self):
        .....

if __name__ == "__main__":
    unittest.main()

```

Figure 7: Unit testing code generated by GPT-4o.

A.4 Grid Search for Optimal Hyper-parameters

The maximum numbers of inner and outer loop iterations are two key hyper-parameters in RTADev. Increasing the numbers of iterations might lead to better performance, but it also brings more costs. Therefore, we need to find a balance between performance and efficiency. In practice, we randomly sample 20 tasks from the FSD-Bench and run a grid search procedure to find the optimal hyper-parameters. In order to reduce the number of hyper-parameters, we skip the first three alignment checking phases and only run the alignment checking and multi-agent discussion phases initiated by the Programmer agent, as the last alignment checking phase involve all the agents. We set the maximum number of outer loop iterations (A in Figure 8) as 1,2,3,5,7 and the maximum number of inner loop iterations (M in Figure 8) as 1,2,3, resulting in 15 sets of experiments in total.

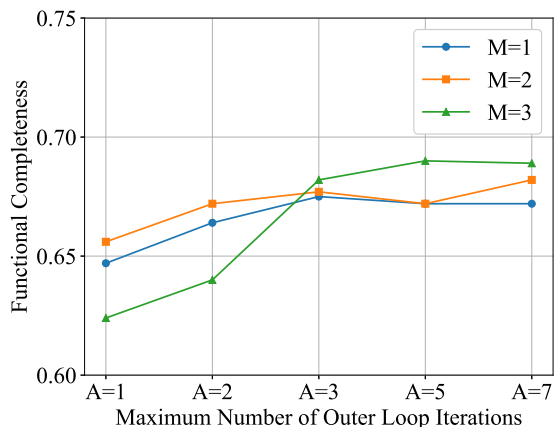


Figure 8: The Functional Completeness performance of RTADev under different hyper-parameter settings in 20 tasks.

As shown in Figure 8, overall, with the increase of the maximum number of outer loop iterations, the FC performance also improves clearly. By averaging over M , we find that the performance on FC increases gradually from $A = 1$ to $A = 7$. This also aligns with our intuition that more rounds of regeneration and alignment checking help to improve the quality of code. Note that the number of alignment checking in our work refers to the maximum allowable number of checking, so the outer loop is possibly finished before reaching the limit. This termination often happens for simpler projects, therefore increasing the number of alignment checking is more suitable for addressing com-

plex projects. In our experiments, we choose $A=3$ since it provides a balance.

Regarding to the number of inner loop iterations, the intuition is that more discussions between supervisors could lead to better results. However, as shown in Figure 8, we do not find a clear trend of improvement by increasing M , especially when A is small. As the number of alignment checking increases, the benefits of discussion start to manifest. Overall, the increase on the number of inner loop iterations shows a slight improvement, which is less important than that of the outer loop.

A.5 Case Study

We selected a set of software in the FSD-Bench as examples and compared the effects of the software generated by MetaGPT, ChatDev and RTADev, which intuitively demonstrated the superiority of RTADev in software development.

As illustrated in Figure 9, the requirement for this software is described as: A software that allows users to create customized stickers using their own photos. Users can select an image, choose the desired shape and size of the sticker, and add text or decorative elements. The software provides easy-to-use tools for cropping, resizing, and adding effects to the photos. Once the sticker is created, users can save it as a transparent PNG file to use in messaging apps or social media platforms. The software generated by RTADev implements all the functionalities mentioned in the software description, while the software generated by ChatDev does not implement functionalities such as cropping, resizing, and adding special effects. In addition, when implementing the functionality where users can choose the desired shape and size of the sticker, the software generated by ChatDev could only select stickers of a fixed size. Figure 10,11,12,13 show four more examples, these examples show that RTADev can reduce the misalignment in the software development process after the real-time alignment mechanism, and generate more comprehensive and accurate software that meets the requirements.

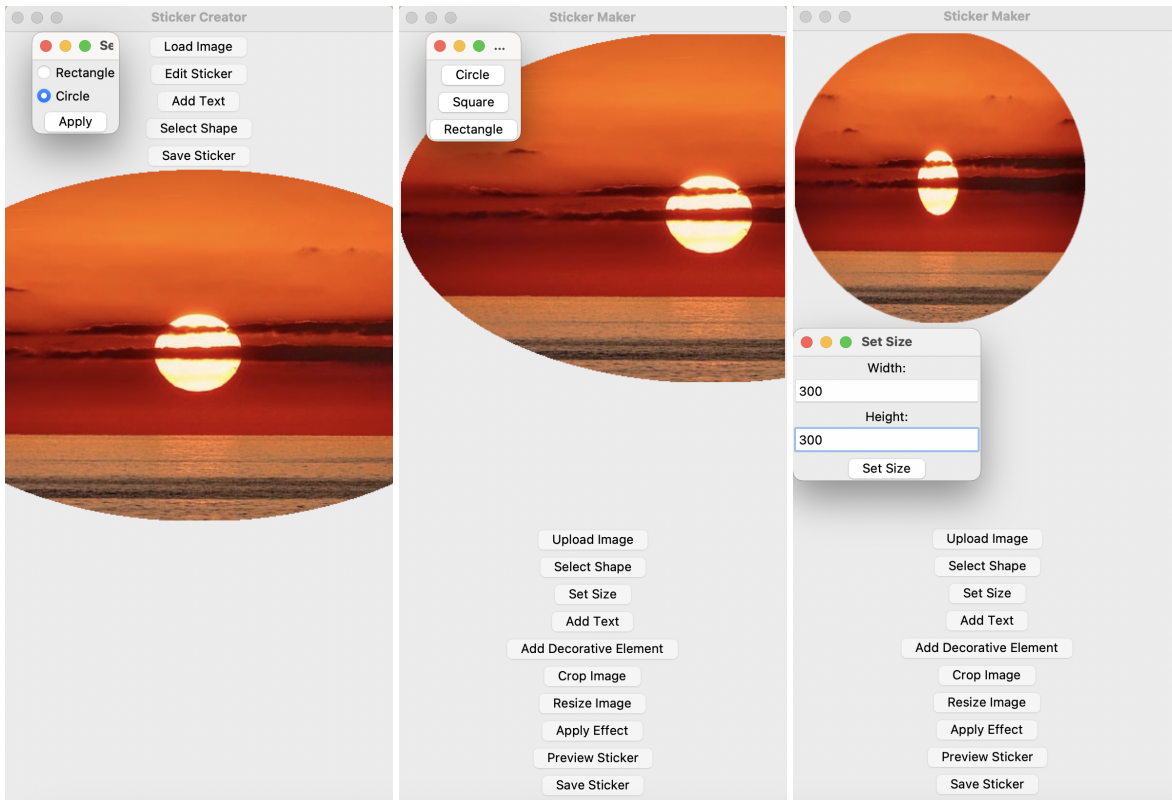


Figure 9: An example of software generated by ChatDev(left) and RTADev(middle and right). In the left picture, we uploaded an image and selected the sticker shape as a circle, and then we cannot change the sticker size. In the middle picture, we uploaded an image and selected the sticker shape as a circle. In the right picture, we set the desired sticker size.

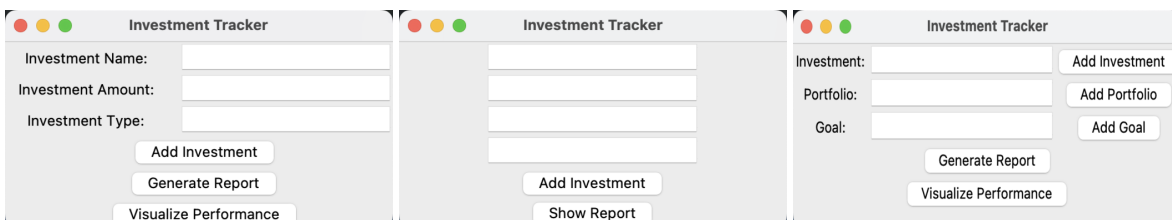


Figure 10: Investment Tracker. From left to right, generated by MetaGPT, ChatDev, and RTADev. The software requires five core functionalities: 1. Input investment details. 2. Categorize investments into different portfolios. 3. Provide visualizations showing investment performance over time. 4. Generate reports on investment performance. 5. Set investment goals. RTADev implements all functionalities, while MetaGPT lacks the function of setting investment goals, and ChatDev only implements the functionalities of input investment and generating reports.

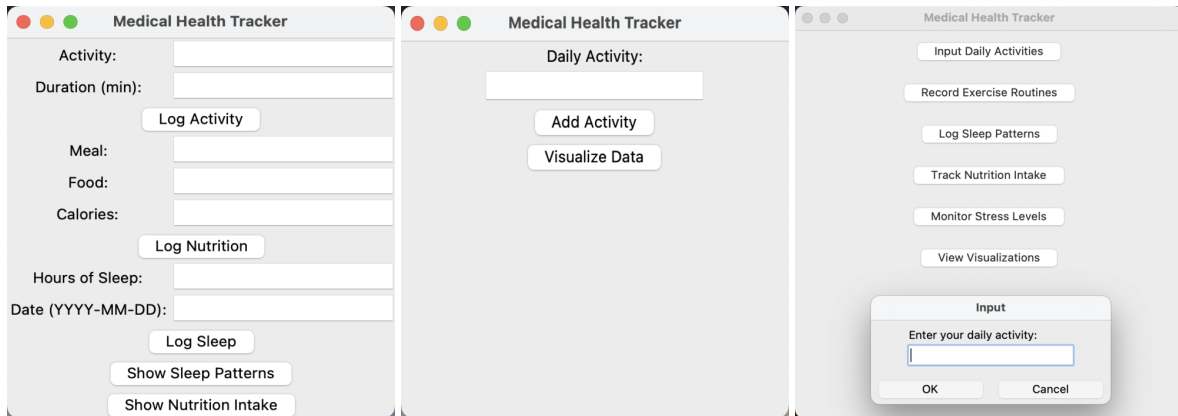


Figure 11: Medical Health Tracker. From left to right, generated by MetaGPT, ChatDev, and RTADev. The software requires six core functionalities: 1. Input daily activities. 2. Record exercise routines. 3. Log sleep patterns. 4. Track nutrition intake. 5. Monitor stress levels. 6. Provide visualizations for health trends analysis. RTADev implements all functionalities, while MetaGPT lacks the function of monitoring stress levels and viewing visualizations, and ChatDev only implements the functionalities of input daily activity and viewing visualizations.

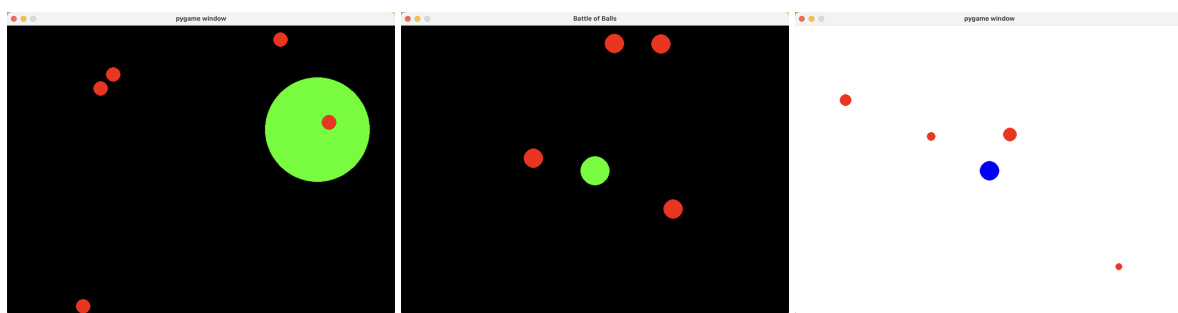


Figure 12: Balls. From left to right, generated by MetaGPT, ChatDev, and RTADev. A requirement for this game is described as: when the player's ball collides with a smaller enemy ball, the player's ball grows in size, and the smaller ball is consumed. In the left picture, the green ball can continuously capture the red ball, which does not meet the user's requirement that each ball can only be captured once. In the middle and right pictures, the game is played according to the user's requirements.

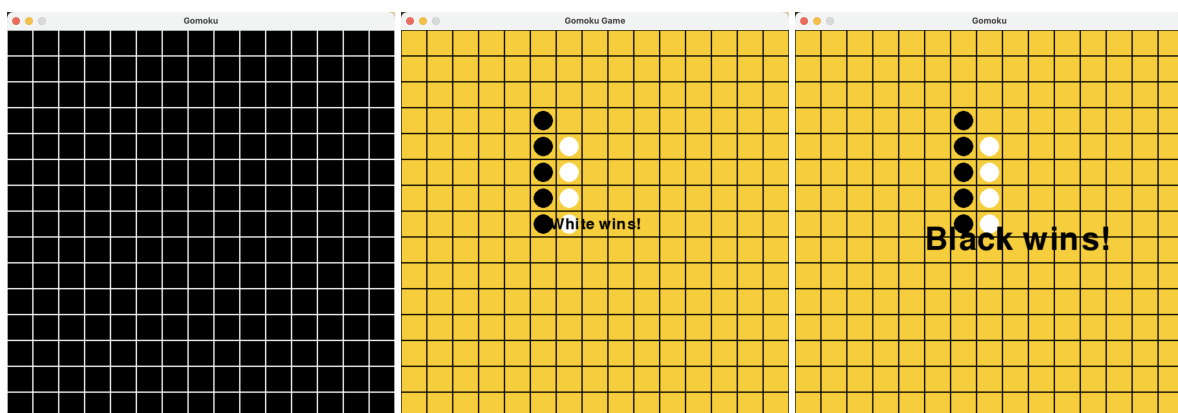
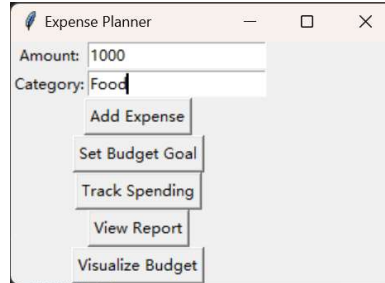


Figure 13: Gomoku. From left to right, generated by MetaGPT, ChatDev, and RTADev. In the left picture, only the chessboard is shown, and neither the black nor the white pieces can be placed. In the middle picture, even though the five black pieces are connected in a line, the software determines that the white pieces win. In the right picture, both the black and white pieces can be placed normally, and the black pieces are correctly determined to win.

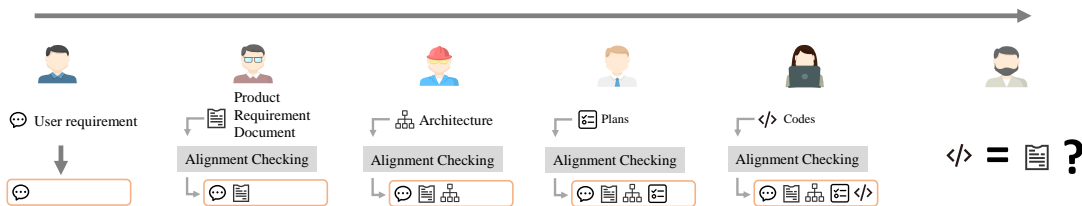
A.6 A Concrete Example

We provide an example of gradually generating ExpensePlanner software project using RTADev. We first show a framework diagram, then demonstrate each phase with corresponding agent's prompt and output.



The generated Expense Planner software

General Workflow



User

User needs to input the initial user requirement, typically in natural language.

User requirement



Software Description

Objective

ExpensePlanner is a user-friendly software application that allows individuals to plan and track their expenses efficiently. It provides a simple and intuitive interface for users to input and categorize their expenses. Users can set budget goals, track their spending, and receive notifications when they exceed predefined limits.

ExpensePlanner also offers visual representations of budget breakdowns and generates reports to help users analyze their financial habits. With ExpensePlanner, individuals can effectively manage their expenses and make informed financial decisions.

CoreFeatures

1. Input expenses.
2. Categorize expenses into predefined categories.
3. Set budget goals.
4. Track spending against budget goals.
5. Provide visual representations of budget breakdowns.
6. Generate reports to analyze financial habits.

Language

Use python to develop an application. You can use tkinter library to build graphical user interfaces.

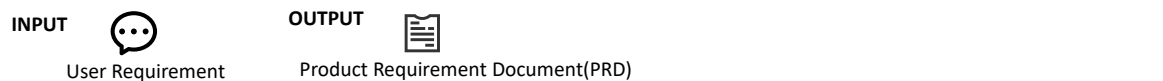
Data Storage

Data will be stored in local text files.

Figure 14: Middle:framework diagram. Bottom:user requirement given by user.

Product Manager

Product manager performs requirement analysis based on the user requirement and generate Product Requirement Document(PRD).



Prompt for generating PRD



role definition

You are an excellent Software Requirements Analyst, and your task is to analyze an initial project description, generate the functional requirements section of the Software Requirements Specification (SRS) document (non-functional requirements are not required). Aim to achieve functional requirements, only require to implement demo.

```
# Context
## system description
{user_requirement}
-----
# Format Example
[OUTPUT]
## general overview of the project
## software functional requirements here...
[/OUTPUT]

## Instruction
- general overview of the project: copy the system description here.
- software function requirements: based on the system description, output system's
function requirements

# Constraint
Format: output wrapped inside [OUTPUT][OUTPUT] like format example, nothing else.

# Action
follow Instruction, Carefully ouput only functional requirement based on the
original system description.
Aim to achieve functional requirements, only require to implement demo.
do not output ```plaintext or other ``` in the start and the end, output directly.
User Authentication, register, login is not needed.
```

Product Requirement Document



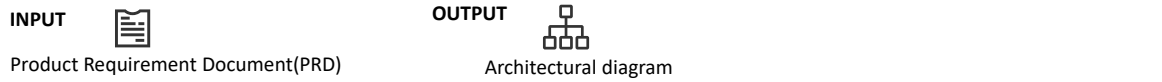
```
## general overview of the project
ExpensePlanner is a user-friendly software application that allows individuals to
plan and track their expenses efficiently. It provides a simple and intuitive
interface for users to input and categorize their expenses. Users can set budget
goals, track their spending, and receive notifications when they exceed predefined
limits. ExpensePlanner also offers visual representations of budget breakdowns and
generates reports to help users analyze their financial habits. With ExpensePlanner,
individuals can effectively manage their expenses and make informed financial
decisions.

## software functional requirements
1. The system shall allow users to input expenses.
2. The system shall allow users to categorize expenses into predefined categories.
3. The system shall allow users to set budget goals.
4. The system shall allow users to track spending against budget goals.
5. The system shall provide visual representations of budget breakdowns.
6. The system shall generate reports to analyze financial habits.
```

Figure 15: Top: prompt for product manager to generate PRD. Bottom: Generated PRD.

Architect

Architect analyzes the PRD and determines the general architecture of the software.



Prompt for generating Architectural diagram



role definition

You are a Architect, your goal is design a concise, usable, complete software system. the constraint is make sure the architecture is simple enough and use appropriate open source libraries. Aim to achieve functional requirements, only require to implement demo.

```
## Context
functional requirements
{prd}
-----
## format example
[CONTENT]
"Implementation approach": "We will ...",
"File list": ["main.py","game.py"],
"Data structures and interfaces": "
classDiagram
    class Main {
        -SearchEngine search_engine
        +main() str
    }...
    Main --> SearchEngine
    SearchEngine --> Index",
"UI design": "- A canvas for... with ..."
[/CONTENT]

## Nodes: "<node>: <type> # <instruction>"
- Implementation approach: <class 'str'> # Analyze the difficult points of the
requirements, select the appropriate open-source framework. If require GUI, you
must also choose a GUI framework (e.g., in Python, you can implement GUI via
tkinter, Pygame, Flexx, PyGUI, etc,)
- UI design:<class 'str'> # ...
- Data Storage: .....
- File list: typing.List[str] # Only need relative paths. ALWAYS write a main.py
here
- Data structures and interfaces: <class 'str'> # Use mermaid classDiagram code
syntax...

## Constraint
Language: Please use the same language as Human INPUT.
Format: output wrapped inside [CONTENT][/]CONTENT] like format example, nothing else.

# Attention
1. If a feature of software requires a GUI, you also need to carefully consider...
## Action
Follow instructions of nodes and Attention, generate output and make sure it
follows the format example.
```

Figure 16: prompt for architect to generate architectural diagram.

INPUT

Product Requirement Document(PRD)

OUTPUTArchitectural diagram

Architectural Diagram

"Implementation approach": "We will use Python with the Tkinter library for the graphical user interface. The application will allow users to input and categorize expenses, set budget goals, and track spending. Data will be stored in local text files, ensuring simplicity and ease of access."

"UI design": "- The main window will contain input fields for expenses, categories, and budget goals. There will be buttons for adding expenses, viewing reports, and visualizing budget breakdowns. A separate window will display charts for visual representation of expenses."

"Data Storage": "Data will be stored in local text files. We will have separate files for expenses, budget goals, and categories. The files will be named 'expenses.txt', 'budget_goals.txt', and 'categories.txt'. Each file will store data in a simple, human-readable format."

"File list": ["main.py", "dietary_planner.py", "database.py", "ui.py"],

"Data structures and interfaces":

```
class ExpensePlanner {
    -List<Expense> expenses
    -List<BudgetGoal> budget_goals
    -List<Category> categories
    +add_expense(amount: float,
category: str) void
    +set_budget_goal(amount: float)
void
    +track_spending() float
    +generate_report() str
    +visualize_budget() void
}
```

```
class BudgetGoal {
    -float amount
    +__init__(amount: float)
}
```

```
class Expense {
    float amount
    -str category
    +__init__(amount: float, category: str)
}
```

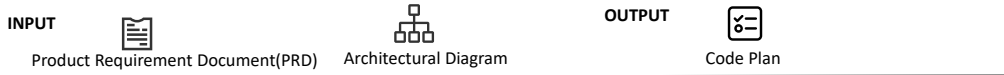
```
class Category {
    -str name
    +__init__(name: str)
}
```

```
ExpensePlanner --> Expense
ExpensePlanner --> BudgetGoal
ExpensePlanner --> Category
```

Figure 17: Generated architectural diagram.

Project Manager

Project Manager schedules a code plan based on the PRD and the architectural diagram.



Prompt for generating Code Plan



role definition

You are a Project Manager, named Eve, your goal is break down tasks according to functional requirement/architecture, generate a task plan, and analyze task dependencies to start with the prerequisite modules. the constraint is use same language as user requirement.

```
## Context
functional requirement:
{prd}
architecture:
{architecture}

## Nodes: "<node>: <type> # <instruction>"
- Required packages: typing.List[str] # Provide required packages in requirements.txt
format.
- Required Other language third-party packages: typing.List[str] # List down the
required packages...
- Logic Analysis: typing.List[typing.List[str]] # Provide a list of files with the
classes/methods/functions to be implemented, including dependency analysis and imports.
- Task list: typing.List[str] # Break down the tasks into a list of filenames...
- Shared Knowledge: <class 'str'> # Detail any shared knowledge, like common utility
functions...

## Constraint
Language: Please use the same language as Human INPUT.
Format: output wrapped inside [CONTENT][CONTENT] like format example, nothing else.

## Action
Follow instructions of nodes, generate output and make sure it follows the format
example.
```

Code Plan

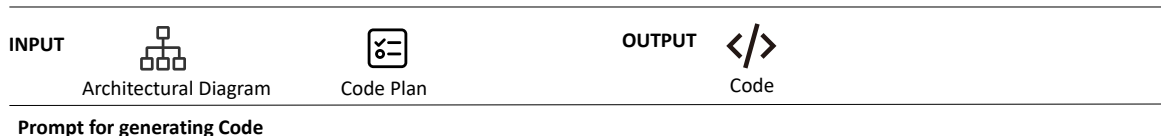


```
"Required packages": ["tkinter", "matplotlib"],
"Required Other language third-party packages": ["No third-party dependencies
required"],
"Logic Analysis": [
    ["main.py",
     "Contains the main application logic and GUI setup."],
    ["expense.py",
     "Contains the Expense class for managing individual expenses and includes methods
to track spending against budget goals."],
    ["budget_goal.py",
     "Contains the BudgetGoal class for managing budget goals and includes methods to
interact with the Expense class."],
    ["category.py",
     "Contains the Category class for managing expense categories."],
    ["report.py",
     "Contains functions for generating reports and visualizations."],
    "Task list": [
        "category.py",
        "expense.py",
        "budget_goal.py",
        "report.py",
        "main.py"
    ],
    "Shared Knowledge": "`category.py`, `expense.py`, and `budget_goal.py` contain classes
that are used in `main.py` for managing expenses, categories, and budget goals, with
explicit interactions defined between Expense and BudgetGoal classes."
```

Figure 18: Top: prompt for project manager to generate code plan. Bottom: Generated code plan.

Programmer

Programmer generates code file based on the architectural diagram and code plan.



```
role definition
You are a professional engineer; the main goal is to write google-style, elegant,
modular, easy to read and maintain code.
Output format carefully referenced "Format example".

# Context

## Design
{architecture}
## Task Plan
{task_plan}
-----
# Format Example
### main.py
```python
...

ui.py
```python
...
-----
# Instruction: Based on the context, follow "Format example", write code. .

# ATTENTION
1. Use '###' to SPLIT CODE SECTIONS, neither '#' and not '##'. do not forget ``` in
each file, refer the example. Output format carefully referenced "Format example".
2. Follow design: YOU MUST FOLLOW "Data structures and interfaces". DONT CHANGE ANY
DESIGN. Do not use public member functions that do not exist in your design.
3. Follow task: YOU MUST write Comprehensive codes to complete task of each file in
task list.
4. CAREFULLY CHECK THAT YOU DONT MISS ANY NECESSARY CLASS/FUNCTION IN THIS FILE.
5. You must import the third-party libraries used in your code
6. Determine the order of writing the files based on your understanding of the project.
7. Write out EVERY CODE DETAIL, DON'T LEAVE TODO,PASS,PLACEHOLDER.
8. Only write code result, do not output any other content in the start or in the end.
```

Figure 19: Prompt for programmer to generate code.

INPUT  Architectural Diagram	 Code Plan	OUTPUT  Code
--	--	--

Code

</>

```

### expense.py
```python
from category import Category

class Expense:
 def __init__(self, amount: float, category: str):
 self.amount = amount
 self.category = category

 @staticmethod
 def track_spending(expenses: list, budget_goals: list) -> dict:
 spending_summary = {}
 total_spent = sum(exp.amount for exp in expenses)
 for goal in budget_goals:
 spending_summary[goal.amount] = total_spent - goal.amount
 return spending_summary
...

main.py
```python
import tkinter as tk
from tkinter import messagebox
from expense import Expense
from budget_goal import BudgetGoal
from category import Category
from report import generate_report, visualize_budget

class ExpensePlanner:
    def __init__(self):
        self.expenses = []
        self.budget_goals = []
        self.categories = []

        self.load_data()

        self.root = tk.Tk()
        self.root.title("Expense Planner")

        self.setup_ui()

    def setup_ui(self):
        tk.Label(self.root, text="Amount:").grid(row=0, column=0)
        self.amount_entry = tk.Entry(self.root)
        self.amount_entry.grid(row=0, column=1)

        tk.Label(self.root, text="Category:").grid(row=1, column=0)
        self.category_entry = tk.Entry(self.root)
        self.category_entry.grid(row=1, column=1)

        tk.Button(self.root, text="Add Expense", command=self.add_expense).grid(row=2,
        column=0, columnspan=2)
        tk.Button(self.root, text="Set Budget Goal",
        command=self.set_budget_goal).grid(row=3, column=0, columnspan=2)
        tk.Button(self.root, text="Track Spending",
        command=self.track_spending).grid(row=4, column=0, columnspan=2)
        ...
...

```

Figure 20: Generated code.

Tester

Tester test the project and provide feedback to programmer. If error occurs, the programmer needs to fix the error and revise codes

INPUT	 Code	OUTPUT	Test Report
-------	---	--------	--------------------

Test Report



Error: Traceback (most recent call last):

```
Traceback (most recent call last):
  File "C:\Users\25929\.conda\envs\mfg\Lib\Tkinter\__init__.py", line 1967, in __call__
    return self.func(*args)
    ^^^^^^^^^^^^^^^^^^^^^
  File "d:\02-Project\ExpensePlanner\review_code\main.py", line 37, in add_expense
    amount = float(self.entry.get())
    ^^^^^^^^^
AttributeError: 'ExpensePlanner' object has no attribute 'entry'
```

INPUT	 Code	Test Report	OUTPUT	 Code
-------	---	--------------------	--------	---



```
### main.py
```python
import tkinter as tk
from tkinter import messagebox
from expense import Expense
...
def setup_ui(self):
 tk.Label(self.root, text="Amount:").grid(row=0, column=0)
 self.amount_entry = tk.Entry(self.root)
 self.amount_entry.grid(row=0, column=1)

 tk.Label(self.root, text="Category:").grid(row=1, column=0)
 self.category_entry = tk.Entry(self.root)
 self.category_entry.grid(row=1, column=1)
def add_expense(self):
 amount = float(self.entry.get())
 category = self.category_entry.get()
 expense = Expense(amount, category)
 self.expenses.append(expense)
 messagebox.showinfo("Success", "Expense added successfully!")

def set_budget_goal(self):
 goal_amount = float(self.amount_entry.get())
 goal = BudgetGoal(goal_amount)
 self.budget_goals.append(goal)
...
```
```

Figure 21: Top: Test report. Bottom: The code revised according to the test report.

A.7 Standardized Prompt Sequences

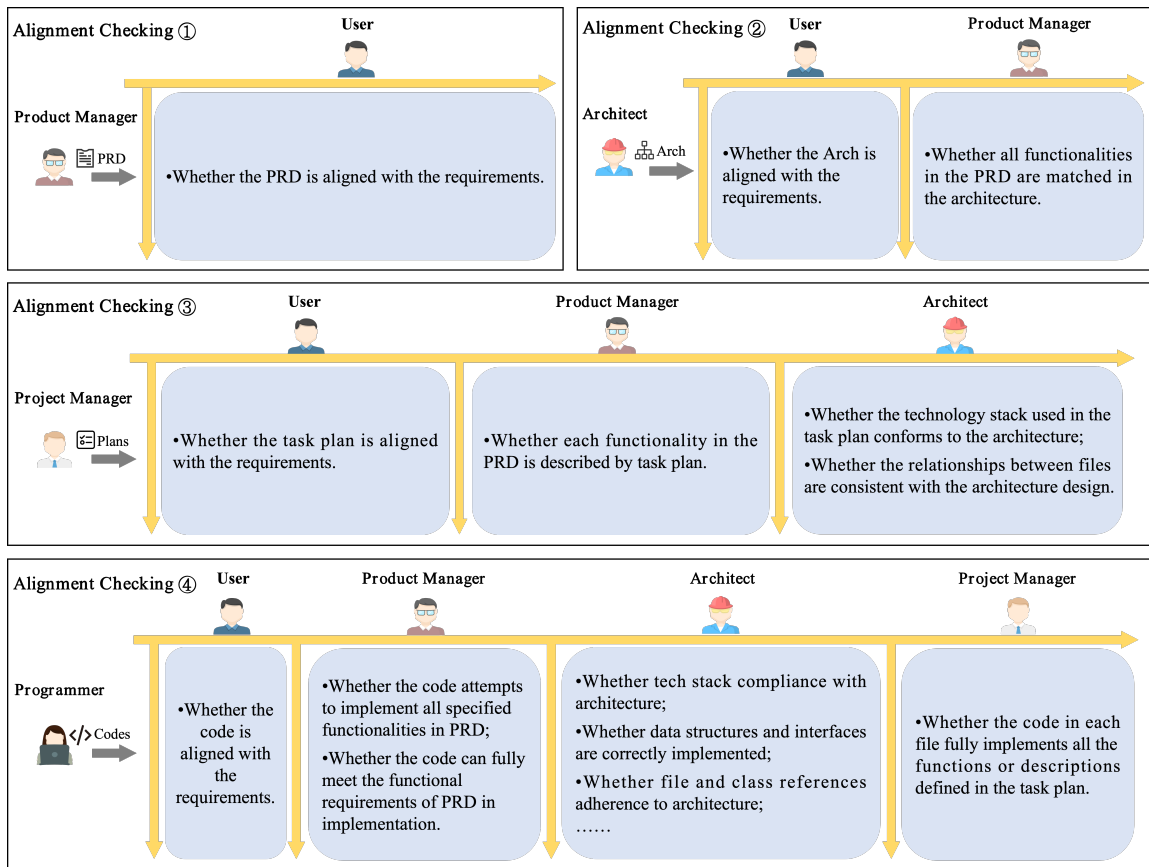
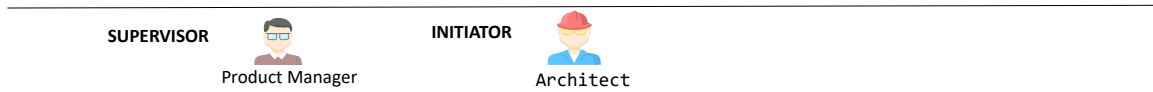






Figure 22: Illustration on the alignment checking phase. The horizontal arrow indicates the order of agents to be queried and the vertical arrows indicate the order of checkpoints in the querying prompts.



 Architecture

Alignment Checking

You are a Product Manager.
This is a Requirement Document:
{prd}

This is a Architecture:
{architecture}

Example:
example for not match
Requirement Document:

2.3. The system shall allow users to choose the shape of the sticker (e.g., circle, square, custom shape).

Architecture:
class ImageEditor {
 +upload_image(file_path: str) Image
 +add_decorative_elements(image: Image, element: str, position: tuple)
Image
 +save_image(image: Image, file_path: str) void
}

Not match. The architecture does not explicitly mention the function of selecting shapes. need to add relevant methods in the ImageEditor class and add a shape selection menu in the GUI class.

.....

final Summary: [NOTMATCH]

example for match
Requirement Document:

2.5. The system shall allow users to add text to the sticker.

Architecture:
+add_text(image: Image, text: str, position: tuple, font: str, size: int, color: str) Image
match. add_text() mention requirement of add text to the sticker.

....

final summary: [MATCH]

Action
Analyze whether all the functions in the requirements are match in the architecture(such as Class and Function).
Add a summary for each analysis, whether it is match or not. use --- to separate each requirement check.
In the final summary, output whether it is MATCH or NOTMATCH(warpped in [], [MATCH] for MATCH summary and [NOTMATCH] for NOTMATCH summary).
Only output [MATCH] or [NOTMATCH] in final summary based on your analysis.
Follow Example and output your result.

Figure 23: Prompt of alignment checking between PRD and architectural diagram.

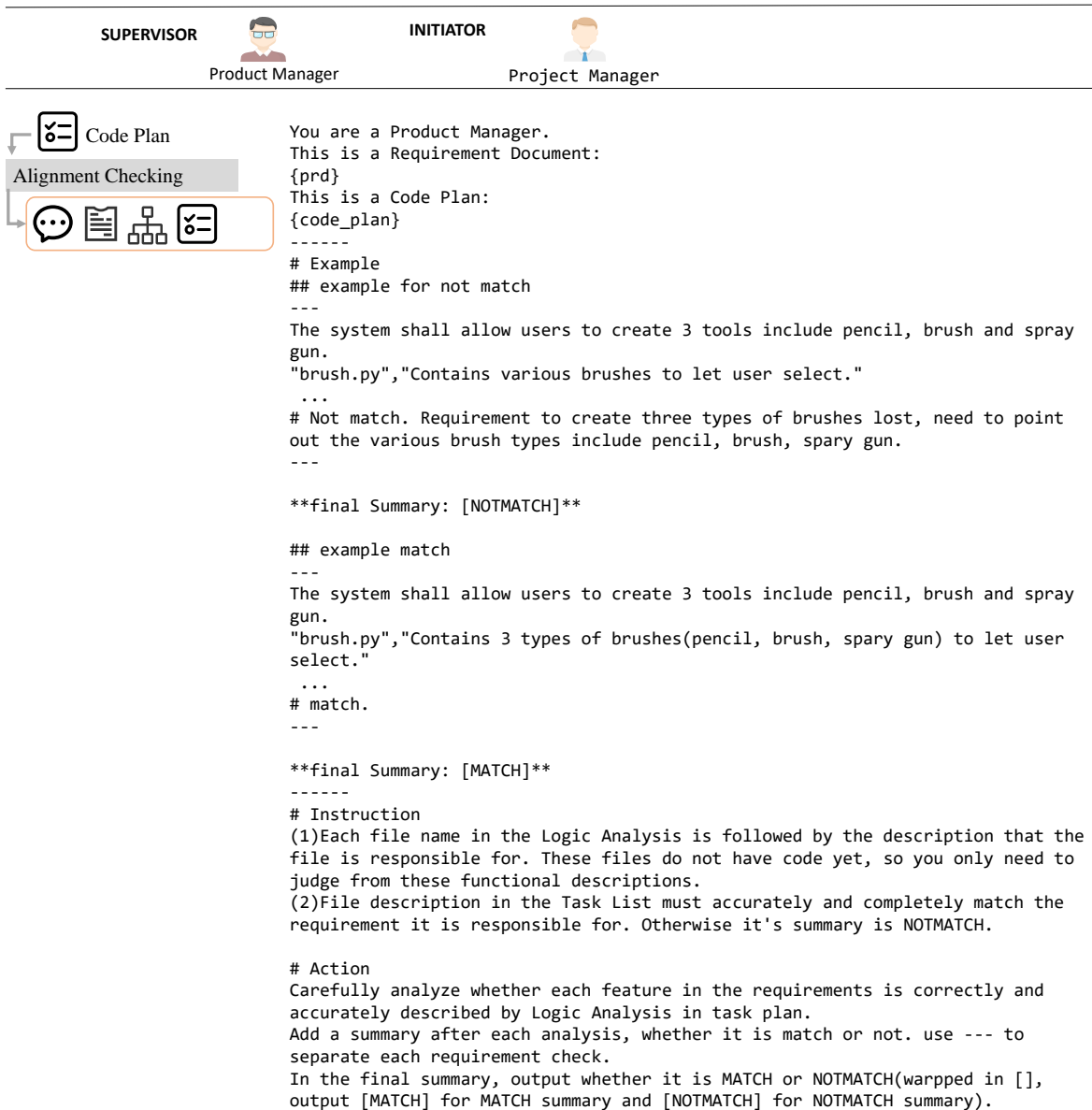


Figure 24: Prompt of alignment checking between PRD and code plan.



Figure 25: Prompt of alignment checking between architectural diagram and code plan.

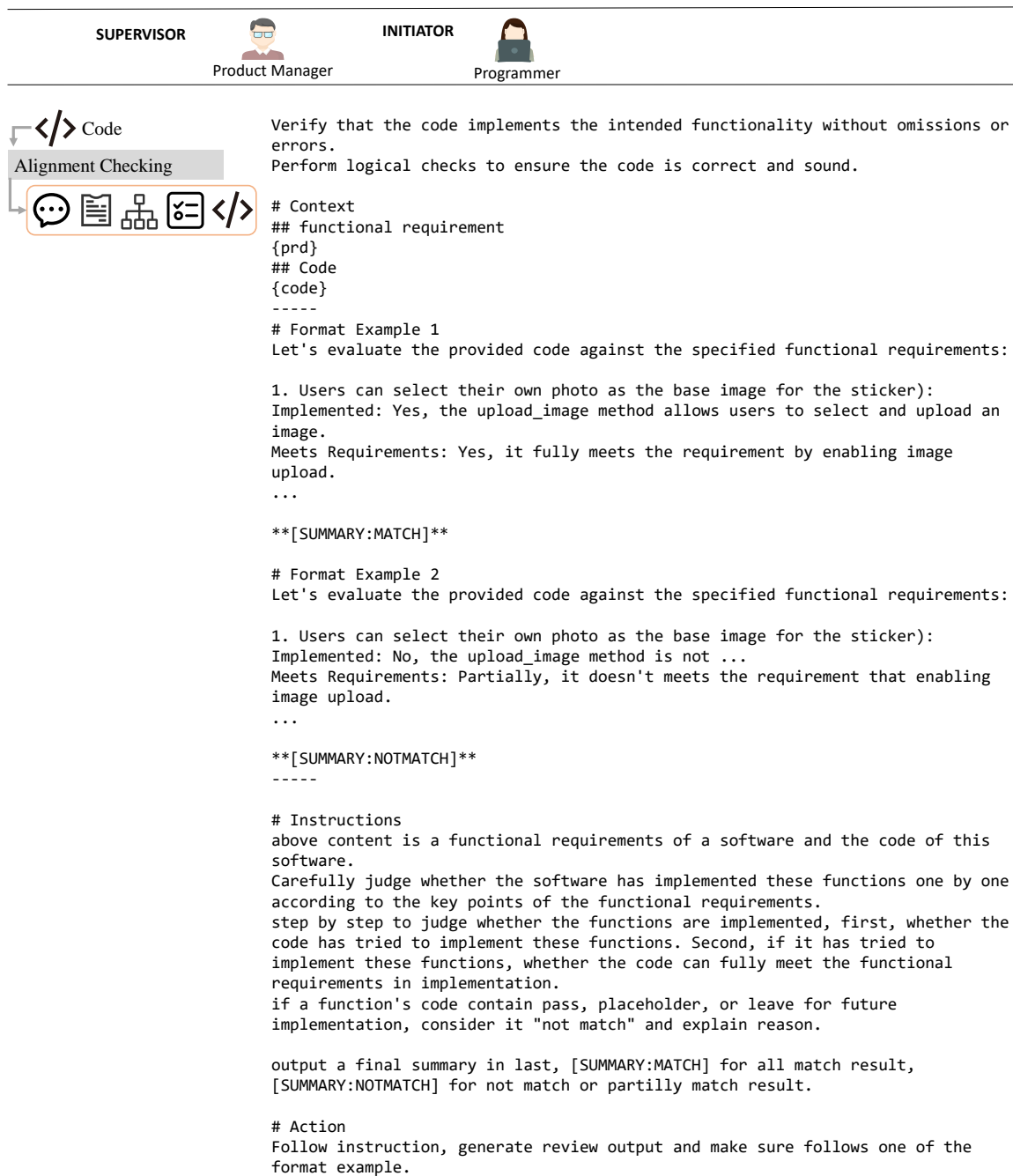


Figure 26: Prompt of alignment checking between PRD and code.

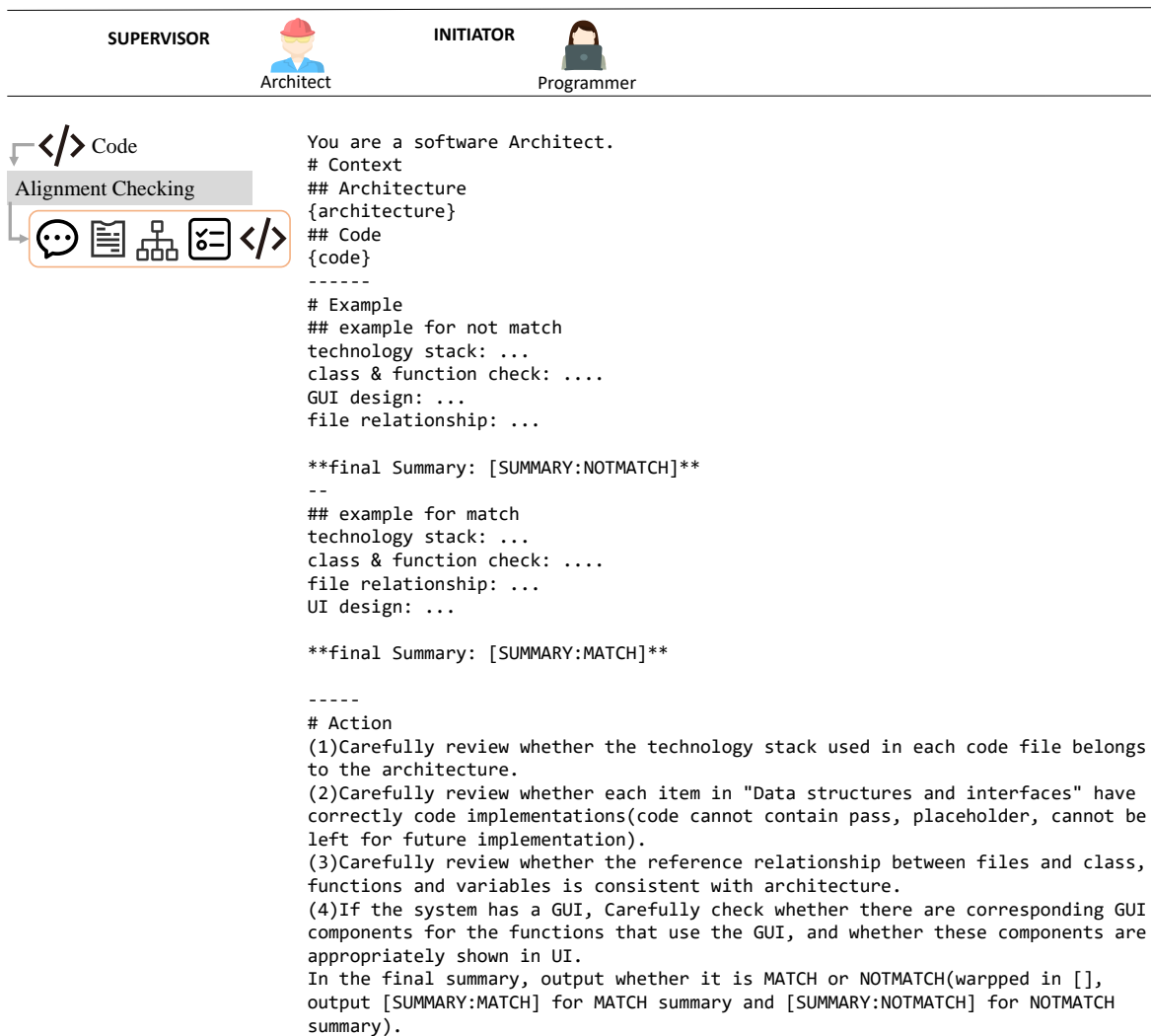


Figure 27: Prompt of alignment checking between architectural diagram and code.

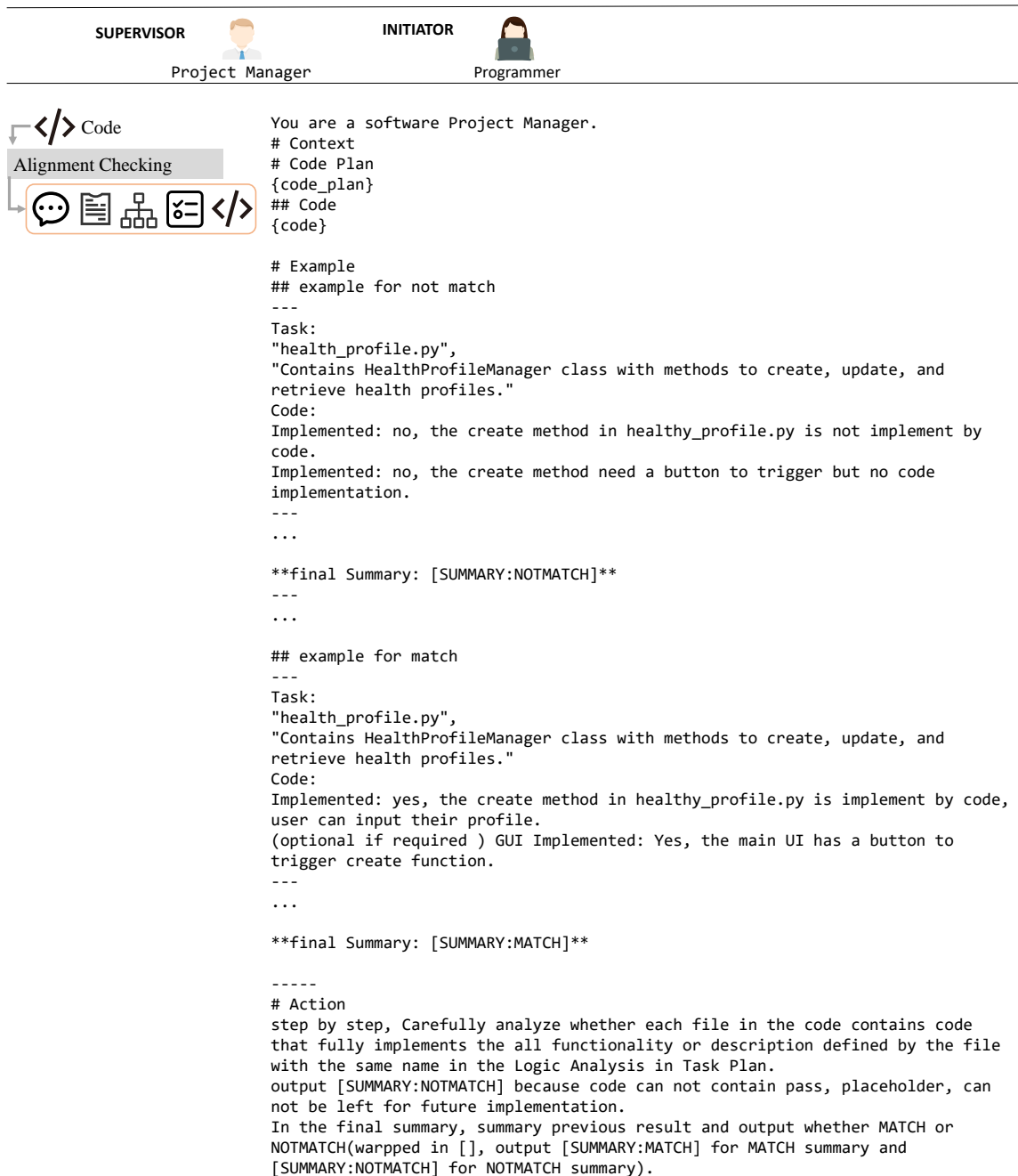


Figure 28: Prompt of alignment checking between code plan and code.

A.8 Multi Agent Discussion

We provide a multi-agent discussion prompt during the code alignment process as an example.

Multi Agent Discussion

Prompt for Multi Agent Discussion

```
You are {role}, You are reviewing a Code based on your content.
You have generated your review result, and others have also generated
review result, All of you are in a team.

---
# Context
## {role_own_content}
{roles_own_content} # such as PRD, architecture, plan, for
corresponding roles.

## Code
{code}

# Review Result
## Your result
{role_alignment_checking_result}
## other's review result
{others_alignment_checking_result}

# Action
First, you need to carefully analyze other's review result of Code and
your review result.
Second, Using other's review result as reference, based on your
functional requirement document, you need to regenerate a new review
result of the Code.

# Constraint
(1)format of Regenerated result must carefully follow your original
review result' format.
(2)no need to copy others' opinions directly.
(2)Do not need to explain in the start and end.
```

Figure 29: Prompt for Multi-Agent Discussion during the code alignment process.