METAGPT: META PROGRAMMING FOR MULTI-AGENT COLLABORATIVE FRAMEWORK

Sirui Hong[1] **Xiawu Zheng**[2] **Jonathan Chen**[1] **Yuheng Cheng**[3] **Jinlin Wang**[1] **Ceyao Zhang**[3]
Zili Wang **Steven Ka Shing Yau**[4] **Zijuan Lin**[2] **Liyang Zhou**[5]
Chenyu Ran[1] **Lingfeng Xiao**[1,6] **Chenglin Wu**[1] *
[1]DeepWisdom [2]Xiamen University [3]The Chinese University of Hong Kong, Shenzhen
[4]Nanjing University [5]University of Pennsylvania
[6]University of California, Berkeley

ABSTRACT

Recently, remarkable progress has been made in automated task-solving through the use of multi-agent driven by large language models (LLMs). However, existing LLM-based multi-agent works primarily focus on solving simple dialogue tasks, and complex tasks are rarely studied, mainly due to the LLM hallucination problem. This type of hallucination becomes cascading when naively chaining multiple intelligent agents, resulting in a failure to effectively address complex problems. Therefore, we introduce MetaGPT, an innovative framework that incorporates efficient human workflows as a meta programming approach into LLM-based multi-agent collaboration. Specifically, MetaGPT encodes Standardized Operating Procedures (SOPs) into prompts to enhance structured coordination. Subsequently, it mandates modular outputs, empowering agents with domain expertise comparable to human professionals, to validate outputs and minimize compounded errors. In this way, MetaGPT leverages the assembly line paradigm to assign diverse roles to various agents, thereby establishing a framework that can effectively and cohesively deconstruct complex multi-agent collaborative problems. Our experiments on collaborative software engineering benchmarks demonstrate that MetaGPT generates more coherent and correct solutions compared to existing chat-based multi-agent systems. This highlights the potential of integrating human domain knowledge into multi-agent systems, thereby creating new opportunities to tackle complex real-world challenges. The GitHub repository of this project is publicly available on: `https://github.com/geekan/MetaGPT`.

1 Introduction

Multi-agent systems that utilize Large Language Models (LLMs) offer remarkable prospects for replicating and enhancing human workflows. However, existing systems [1, 2, 3, 4, 5, 6, 7] tend to oversimplify the complexities inherent to real-world applications. These systems are primarily struggling to promote effective collaboration through conversational and tool-based interactions. This leads to challenges such as achieving coherent interactions, mitigating unproductive feedback loops, and guiding meaningful collaborative engagements [2, 3, 8, 9, 10]. In parallel, humans have developed widely accepted Standardized Operating Procedures (SOPs) across many domains through prolonged collaborative practice [11, 12, 13]. These SOPs play a critical role in supporting task decomposition and efficient coordination. A comprehensive understanding and integration of real-world practices is essential. By addressing these prevalent limitations and integrating these insights as meta programming, we can facilitate the emergence of an innovative paradigm in the design and organization of LLM-based multi-agent systems, thereby enhancing their effectiveness and applicability.

Furthermore, SOPs outline responsibilities among team members and set standards for component outputs, ensuring that tasks are executed consistently and accurately, in alignment with the defined roles and quality standards [11, 12, 13, 14]. In software engineering, the waterfall methodology delineates sequential phases from requirements analysis to

*Chenglin Wu (E-mail: alexanderwu@fuzhi.ai) is the corresponding author with DeepWisdom, 361000, China

deliverables, fostering effective collaboration among multiple roles. [11, 12]. Moreover, human roles possess specialized expertise tailored to their assigned responsibilities. Software engineers leverage their programming proficiency to implement code, while product managers conduct market analysis to formulate business needs. Nevertheless, without standardized outputs, collaboration becomes disorderly [15, 14, 16]. For example, product managers analyze competition and user needs to create Product Requirements Documents (PRDs) with a standardized structure, which guides the development process. Such normative artifacts are critical outputs that crystallize collective understanding to advance complex, multifaceted projects that require interconnected contributions from various roles [17, 18, 19]. Therefore, in MetaGPT, we equip the agent with the ability to generate standard actions output, such as high-quality requirements documents, design artifacts, flowcharts, and interface specifications. These intermediate standardized outputs significantly increase the success rate of the final code execution.

To well organize SOPs in multi-agent collaboration, we further introduce a new meta programming mechanism [20], which allows programs to manipulate other programs as data. Specifically, the techniques enable programs to automatically generate, analyze, transform, and modify code at runtime. This enables adaptive and flexible programming capabilities. In this way, SOPs act as a meta-function, taking the team and requirements as inputs to synthesize the target code. When SOP is implemented with a dedicated team, which includes software agents, it effectively creates a programming function that can generate code based on the given inputs. In essence, by combining standardized procedures with a coordinated multi-agent system, meta programming can be used to leverage agent cooperation in dynamically constructing functions for programming tasks, such as code generation. This demonstrates how meta programming can utilize agent-based techniques to enable flexible and automated software synthesis.

In this work, we present MetaGPT, a meta programming technology that utilizes SOPs to coordinate LLM-based multi-agent systems. SOPs serve as our strategy for organizing the collaboration of multi-agents which enhances the efficiency of their cooperative efforts. Firstly, each agent is identified by descriptive information, which allows the system to initialize with an appropriate role-specific prompt prefix. This approach integrates domain knowledge into agent definitions. Secondly, we analyze efficient human workflows to encapsulate procedural knowledge required for collaborative tasks. These SOPs are encoded into the agent architecture using role-based action specifications. Finally, a shared environment connects agents, providing visibility into their actions and enabling them to collaborate, access tools, and share resources. Agents produce standardized action outputs to facilitate knowledge sharing. Additionally, agents in MetaGPT can actively observe and retrieve relevant information, which is a more efficient approach compared to passively receiving data through dialogue. The environment mirrors the infrastructure of human workplaces that facilitate team collaboration.

To demonstrate the effectiveness of our design, we showcase the collaborative software development workflow and conduct experiments to implement the associated code. In code generation benchmarks, MetaGPT achieves a new state-of-the-art with 81.7% and 82.3% in Pass@1 over direct methods. When compared to other LLM-based programming frameworks, such as AutoGPT [21], LangChain [22], and AgentVerse [23], MetaGPT can handle significantly higher levels of software complexity and distinguishes itself through its extensive functionality. Remarkably, in our experimental evaluations, we have achieved a 100% task completion rate, further underscoring the robustness and efficiency of the MetaGPT framework.

In conclusion, our research outlines a transformative paradigm for LLM-based multi-agent systems by integrating SOPs. MetaGPT exemplifies the potential of meta programming, particularly within the LLM-based multi-agent environment. This convergence of structured processes, agent collaboration, and advanced model-driven techniques is transforming the landscape of software development. The significant contributions of our endeavor include:

- **Introduction of a meta programming framework.** We propose MetaGPT, an advanced framework for LLMs-based multi-agent collaboration. The meta programming paradigm makes MetaGPT highly convenient and flexible for building multi-agent systems. This provides teams with an accessible platform to easily develop and validate collaborative multi-agent systems.

- **Integration of human SOP process design.** In a groundbreaking effort, we have successfully integrated SOPs into the system's design. This integration empowers our LLM-based agents to generate diverse, high-quality, structured intermediary designs and documentation. It significantly enhances robustness by reducing errors in LLM-based multi-agent collaboration. This design equips the system with the ability to systematically engineer solutions for complex tasks, demonstrating decoupling and problem-solving capabilities similar to those of human teams.

- **Achieving state-of-the-art performance.** We have achieved new state-of-the-art performance on HumanEval [24] and MBPP [25] benchmarks through rigorous experiments. Extensive quantitative and qualitative analyses compellingly validate the effectiveness of MetaGPT for multi-agent programming and complex task resolution. The results indicate that MetaGPT has the potential to address hallucination issues in LLMs, thereby guiding collaborative LLM systems toward more effective designs.

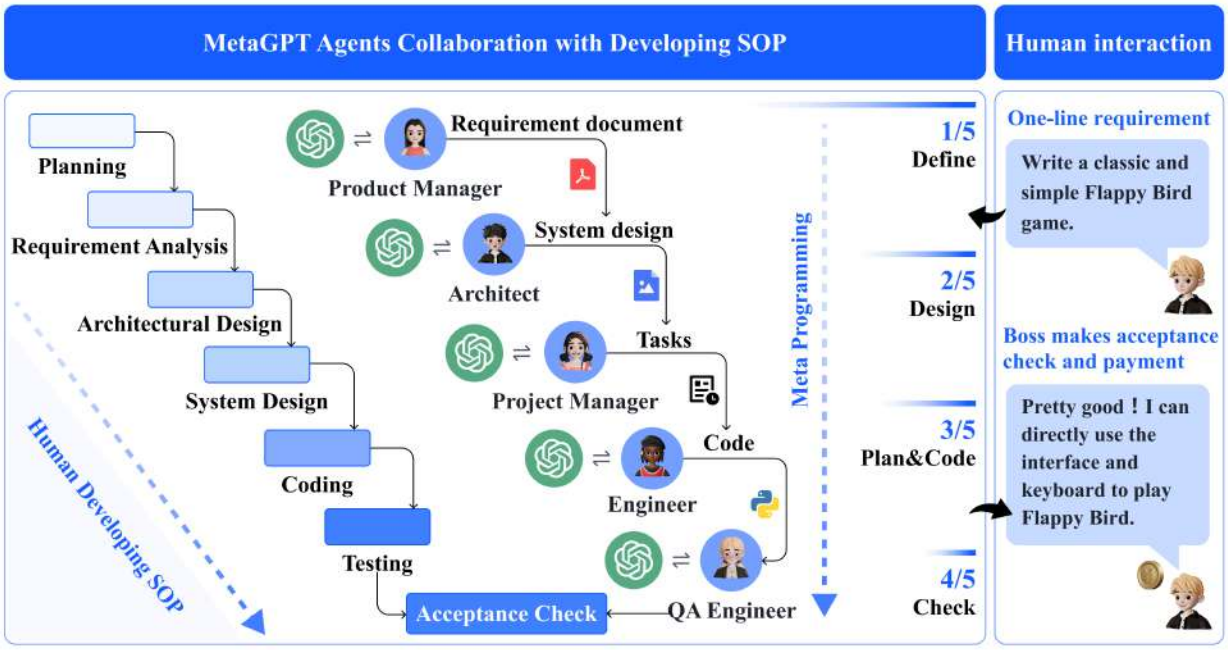

Figure 1: **A comparative depiction of the software development SOP between MetaGPT and real-world human team.** SOP defines job roles and workflows in society. In software engineering, the waterfall method outlines steps from analysis to delivery, promoting teamwork across roles. The MetaGPT approach showcases its ability to decompose high-level tasks into detailed actionable components handled by distinct roles (ProductManager, Architect, ProjectManager, Engineer, QA Engineer), thereby facilitating role-specific expertise and coordination. This methodology mirrors human software development teams. The diagram illustrates how MetaGPT is designed to handle complex tasks and promote clear role delineations, making it a valuable tool for complex software development scenarios

2 Related Work

LLM-based Automatic Programming Automatic programming is a hot research topic in NLP. Researchers trained classifiers to identify and reject erroneous programs [26] and developed the mechanism of iterative feedback to generate embedded control programs [27]. There are also state-of-the-art methods that utilize majority voting to select candidate programs [28] and use execution results to improve program synthesis [29, 30]. More recently, agents based on LLMs [31, 32, 33] have facilitated the development of automatic programming. Li *et al.* [2] proposed a simple role-play agent framework that realizes automatic programming based on one-sentence user requirements through the interaction of two roles. Moreover, Qian *et al.* [10] utilized multiple agents for software development, but they did not incorporate advanced human teamwork experience. Although existing multi-agent cooperation [2, 10] has currently improved productivity, they did not fully draw on efficient workflows in human production practices. Thus, they are hard to solve more complex software engineering problems.

Besides, A series of the fundamental works on which this paper is based need to be mentioned. ReAct [31] utilizes Chain of Thought prompts [34] to generate reasoning trajectories and action plans with LLMs. Reflexion [32] infers more intuitive future actions through self-reflection. Both papers illustrate how the Re-Act style loop of reasoning is a good design paradigm for empowering LLM-based agents. ToolFormer [35] can teach themselves to use external tools via simple APIs. SWIFTSAGE [33] assigns difficult situations to slow thinking to deal with, while normal situations are dealt with directly by fast thinking. Based on the above design, we emphasize that role division of labor is helpful for complex task processing.

Multi-agent Collaboration Prior works have explored using multiple LLMs in a collaborative setting to solve complex tasks [36, 10]. The motivation is that by cross-agent interaction, LLMs can collectively exhibit enhanced performance by aggregating their strengths. There have been many previous explorations of multi-agent, including collective thinking [4, 5, 6, 7], conversation dataset collection [2, 37], sociological phenomenon research [1, 38], collaboration for efficiency [3, 8, 2, 10]. *(i) Collective thinking*: many advanced works [4, 5, 6, 7] enhanced the task solving capabilities of LLM by integrating the multiple-agent discussion. *(ii) Conversation dataset collection*: Wei *et al.* [37] and Li *et al.* [2] built a conversation datasets through Role Playing. *(iii) Sociological phenomenon research*: Park *et al.* [1] constructed a town of 25 agents, realizing simple language interaction, social cognition, and memory of the agents, while there is a lack of cooperation and production. Akata *et al.* [38] studied LLM cooperation and

coordination behavior by having multiple agents play a repeated game. *(iv) Collaboration for efficiency*: Cai *et al.* [3] modeled cost reduction by having a combination of large models as tool makers and small models as Tool users. Zhang *et al.* [8] built a framework for collaboration between agents that enables verbal communication, collaboration, and improved efficiency. Li *et al.* [2] and Qian *et al.* [10] proposed a multi-agent framework for software development. However, their cross-agent communication is natural language conversations, not a standardized software engineering document, and does not incorporate advanced human process management experience. The key issues persist in multi-agent cooperation around maintaining coherence, avoiding unproductive loops, and directing beneficial interactions. Therefore, this paper emphasizes the practice of advanced human processes (*e.g.*, SOPs in software development) in multi-agent systems.

Autonomous System Design Existing autonomous systems like AutoGPT [21] automate tasks by breaking down high-level goals into multiple sub-goals and executing them in ReAct-style loops, while facing challenges with coherence and validation. LangChain [22] helps develop LLM applications in combination with other computational tools or knowledge bases. Recently, the multi-agent architecture has been proven to be an effective design. GPTeam [39] creates LLM-based multiple agents that work together to achieve a predetermined goal. AgentVerse [23] is a LLM powered multi-agent scenario simulation framework. Langroid [40] builds LLM-based multi-agent programming. SocraticAI [41] improves problem-solving by leveraging the power of multiple agents in socratic dialogue. Since most of them are not embedded in advanced human management experience, they are unable to solve larger and more complex real-world projects. Our MetaGPT incorporates human workflow insights for more controlled and validated performance.

3 Meta Programming for Collaborative Agents via Standardized Operating Procedure

In this section, we first provide an overview of MetaGPT, our proposed meta programming multi-agent collaborative framework, for addressing complicated real-world problems, as demonstrated in Figure 2. Then, in Section 3.2, we will elaborate on the design of the core components in our framework. We choose software development as an example to illustrate how our MetaGPT dispatches multiple agents to implement the SOPs in a software development team. This allows for the completion of the end-to-end development process with only one human input for task requirements.

Furthermore, in Section 3.3, we present a practical example (as illustrated in Figure 3) to illustrate how MetaGPT coordinates multiple agents with corresponding roles to fulfill a one-line requirement: *Make the 2048 sliding tile number puzzle*.

3.1 Framework Overview

We examine the design and workflow of MetaGPT by investigating its core architecture, its knowledge-sharing component, and the rationale behind its intricate workflows. The design of MetaGPT is organized into two distinct layers.

Foundational Components Layer. This layer establishes core building blocks necessary for individual agent operations and system-wide information exchange, including **Environment**, **Memory**, **Roles**, **Actions**, and **Tools**. As depicted in Figure 2, **Environment** enables shared workspace and communications. **Memory** stores and retrieves historical messages. **Roles** encapsulate domain-specific skills and workflows. **Actions** execute modular sub-tasks. **Tools** provide common services and utilities. This layer offers an underlying infrastructure for agents to function in assigned roles, interacting with each other and the system.

Collaboration Layer. Built upon the foundational components layer, the collaboration layer orchestrates individual agents to collaboratively resolve complex problems. It institutes essential mechanisms for cooperation: Knowledge Sharing and Encapsulating Workflows.

- **Knowledge Sharing.** This mechanism allows agents to exchange information effectively, contributing to a shared knowledge base. Agents can store, retrieve, and share data at different levels of granularity. It not only enhances coordination but also reduces redundant communication, increasing overall operational efficiency.
- **Encapsulating Workflows.** This mechanism leverages SOPs to break down complex tasks into smaller, manageable components. It assigns these sub-tasks to suitable agents and supervises their performance by standardized outputs, ensuring that their actions align with the overarching objectives.

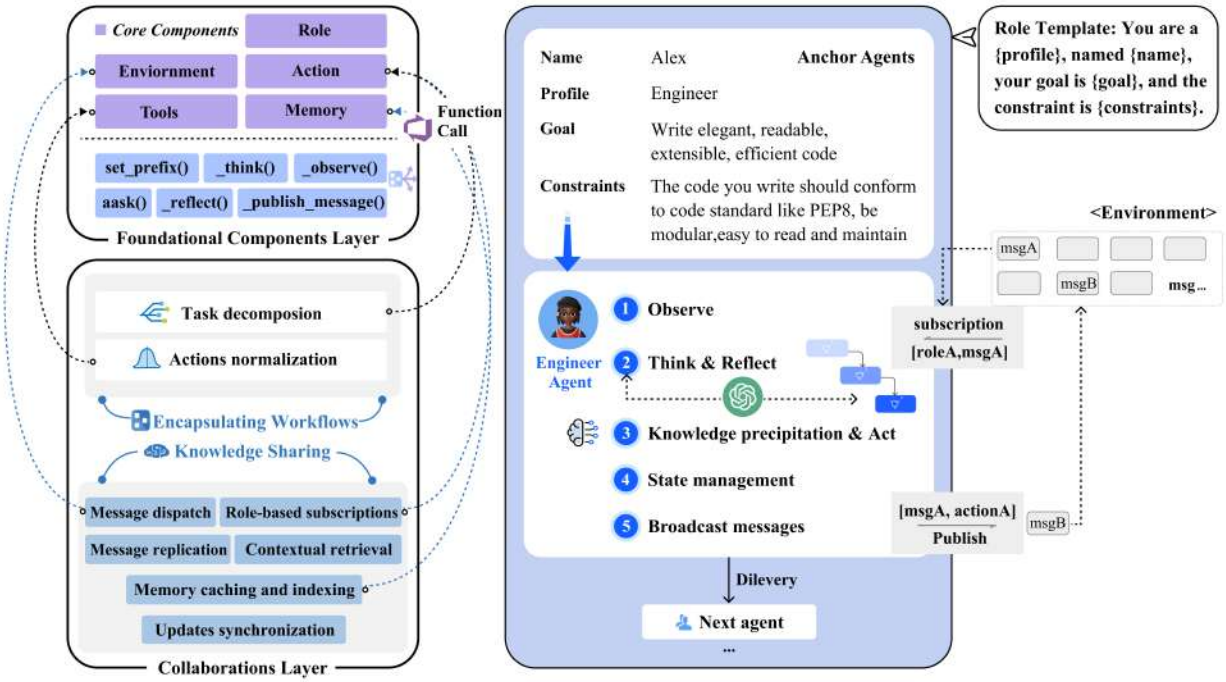

Figure 2: **Overview of the MetaGPT framework.** Presented is a two-layer architectural design: i) the Foundational Components Layer, which is essential for agent operations and system-wide communication, and ii) the Collaboration Layer, which facilitates agent coordination through key mechanisms such as knowledge sharing and workflow encapsulation. Within this framework, the agent capabilities in MetaGPT have been significantly enhanced. The instantiation of agents, guided by specialized role prompts known as "anchor agents," equips roles with capabilities for observation, thinking, reflection, and knowledge accumulation. These roles interact with the environment through established subscription and publication methods.

The division into foundational and collaborative layers promotes modularity while ensuring both individual and collective agent capabilities. The components offer reusable building blocks and utilities while the collaboration modules integrate purposeful coordination.

3.2 Core Mechanism Design

3.2.1 Role Definitions

The MetaGPT framework facilitates the creation of various specialized role classes, such as ProductManager, Architect, and others, which inherit from the base Role class. A base role class is characterized by a set of key attributes: name, profile, goal, constraints, and description. Specifically, **Profile** embodies the domain expertise of the role or job title. For instance, an Architect's profile might encompass software design, while a ProductManager's profile could concentrate on product development and management. **Goal** represents the primary responsibility or objective that the role seeks to accomplish. A ProductManager's goal could be presented in natural language as efficiently creating a successful product. **Constraints** denote limitations or principles the role must be followed when performing actions. For example, an Engineer could have constraints to write standardized, modular, and maintainable code. The constraints might dictate as *The code you write should conform to code standards like PEP8, be modular, easy to read, and maintain*, as shown in Figure 2. **Description** provides additional concrete identity to help establish a more comprehensive role. Role initialization in the MetaGPT framework uses natural language to comprehensively describe the responsibilities and constraints of each role. Roles not only aid human understanding but also guide the LLMs to generate actions that align with the role's profile, enabling each agent to be proficient in its respective task. We define this process as anchor agents, which help humans to encode domain-specific responsibilities and capabilities into an LLM-based agent, and also add behavior guidance on expected functions. We will discuss this further in Section 3.2.2

For instance, an Engineer in a software company can be initialized using role-specific settings in MetaGPT, as illustrated in Figure 2: ***Role: You are a [profile], named [name], your goal is [goal], and the constraint is [constraints].***

The comprehensive role definitions provided by the MetaGPT framework enable the creation of highly specialized LLM-based agents, each tailored to specific domains and objectives. The role definitions not only introduce a layer of behavior guidance based on expected functionality but also facilitates the creation of diverse and specialized agents, each expert in its domain. This leads to the development of more effective and efficient LLM-based agents capable of handling a wide range of tasks.

In MetaGPT, intelligent agents not only receive and respond to information, but they also observe the environment to extract critical details. These observations guide their thinking and subsequent actions. Finally, significant information is extracted from the environment and stored in memory for future reference, effectively making every agent an active learner. These agents take on specialized roles and follow certain key behaviors and workflows as shown in Figure 2:

Think & Reflect. Roles can retrieve role descriptions to frame thinking, and then reflect on what needs to be done and decide the next actions, via _think() function. "Think first, then act" - carefully deliberate before replying

Observe. Roles can observe the environment and think/act based on observations using the _observe() function. They watch for important information and incorporate it into their memory to enrich their contextual understanding and inform future decisions.

Broadcast messages. Roles can broadcast messages into the environment using the _publish_message() function. These messages contain details about current execution results and related action records, for publishing and sharing information.

Knowledge precipitation & Act. Roles are not only broadcasters but also recipients of information from their environment. Roles can assess the incoming messages for relevancy and timeliness, extract relevant knowledge from shared environment and maintain an internal knowledge repository to inform decisions. They execute actions via consulting the LLM with enriched contextual information and self-knowledge. Execution results are encapsulated as Message while norm artifacts are shared by the environment.

State management. Roles can track their actions by updating their working status and monitoring a to-do list. This enables a role to process multiple actions sequentially without interruption. When executing each action, the role first locks its status. After completing the action, it marks the status as unlocked. This prevents other actions from interrupting the workflow. This is a crucial capability in role design, making roles more human-like. It empowers roles with more natural execution dynamics based on real-world human collaboration phenomena.

In summary, the MetaGPT framework is a versatile and powerful approach to designing and implementing intelligent agents with specialized capabilities. These agents, with the help of role features, can effectively collaborate, learn, adapt, and perform various tasks, making them valuable assets in a wide range of applications and domains.

3.2.2 Prompts Instantiating SOPs

As previously discussed, MetaGPT uses prompts to instantiate real-world SOPs into well-defined agent workflows. We demonstrate how MetaGPT transforms SOPs into executable action instances via natural language prompts. This process involves using prompts to instantiate SOPs, providing step-by-step guidance based on established practices, and ensuring consistent, structured execution of complex sequencing tasks.

First, we introduce the Action class in detail, then demonstrate the design of standardizing action-level granular prompts. Within the MetaGPT framework, Action serves as the atomic unit for agents to execute specific tasks, specified through natural language. The key attributes include:

Prefix. A role-specific prefix is injected into the prompt to establish a role context. The set_prefix() method configures identifiers for role-specific prompts.

LLM proxy. Each Action contains an LLM proxy, which can be invoked via the aask() method to enrich action details using input context expressed in natural language prompts. Additionally, various role-specific context parsing functions can be implemented within the Action class. These functions are designed to extract and provide sufficient contextual information from inputs to the LLMs. These parsers can selectively extract the most relevant information from inputs to create clear, focused prompts for the LLM proxy. This helps ensure the LLM is provided with sufficient context to generate useful output, rather than being overwhelmed by excessive irrelevant data.

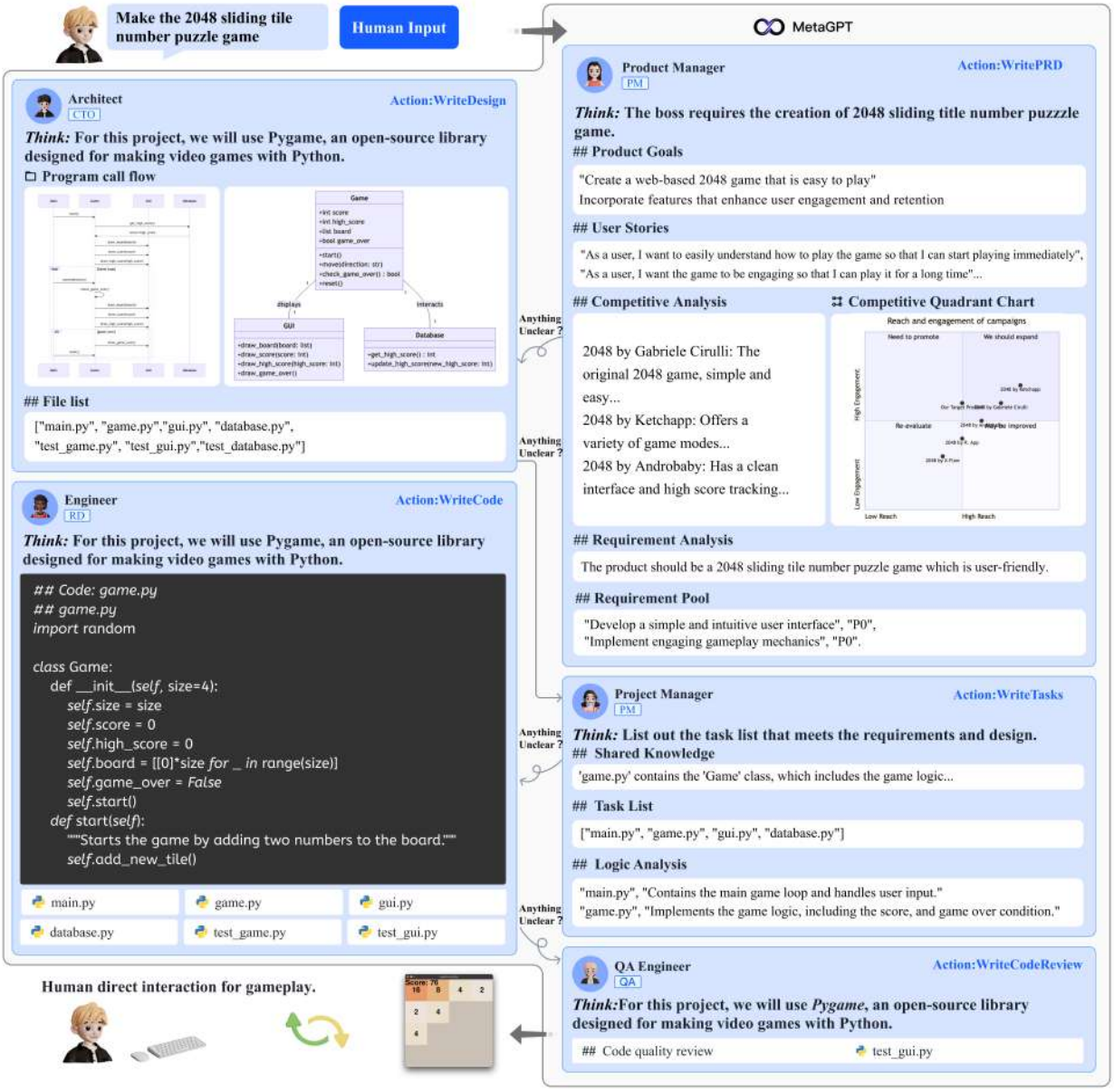

Figure 3: **A schematic diagram of the software development process within the MetaGPT framework.** This diagram illustrates the sequential software development process within the MetaGPT framework. Upon receiving a requirement from humans, the product manager commences the process by conducting requirement and feasibility analyses. The architect then formulates a specific technical design for the project. Next, the project manager performs sequence flow illustrations to address each requirement. The engineer takes responsibility for the actual code development, followed by a quality assurance (QA) engineer who carries out comprehensive testing. This schematic showcases MetaGPT's emulation of the real world.

This approach benefits the LLMs by reducing irrelevant noise and concentrating inputs on key context points. As a result, the prompts instantiate not only workflows but also the context awareness required to adapt execution appropriately based on inputs.

Standardized outputs schema. A structural representation defining expected output schema, is used to extract structured data. We provide basic methods to parse LLM results into structured outputs.

Instruct content. Structured data can be extracted from action output using the standardized output schema. This information is encapsulated as a message and published to the environment ultimately.

Retry mechanism. Defined by the number of attempts and waiting time to enable retrying Actions for robustness.

In summary, each action in MetaGPT requires the definition of standardized output by encoding high-quality expert-level structural key points. The LLMs then refine the action based on the standardized output schema for the specific task. Essentially, we provide each action with a prompt template that conforms to the standards for the role. This template helps steer the LLM's behavior to generate normalized outputs.

We define a WritePRD action for a ProductManager agent in MetaGPT to showcase the process. As shown in Figure 3, we incorporate domain expertise for this action by specifying the required outputs, including product goals, user stories, competitive analysis, requirements analysis, and a prioritized requirement pool. These outputs encapsulate key artifacts and practices in product management, following industry conventions. Additionally, we implement supporting skills for the ProductManager agent, such as web search APIs to enhance analysis, and diagramming tools like mermaid [42] to visualize competitive quadrant charts. The ProductManager efficiently structures output sections. By equipping the agent with these complementary capabilities that align with real-world product management responsibilities, the WritePRD action can execute the subtask while adhering to standardized workflows.

In this manner, the WritePRD action exemplifies how MetaGPT action definitions combine domain knowledge, output schemas, and assistive skills to transform high-level SOPs into executable and customizable procedures for agents. By incorporating real-world best practices into action specifications, MetaGPT connects theoretical expertise with structured execution that is customized for collaborative workflows. Moreover, each action becomes more than just an isolated function. It forms part of a comprehensive set of guidelines that govern the behavior of the LLM within its role, ensuring the production of high-quality, well-organized, and task-specific outputs.

3.2.3 Actions for Standardized Outputs

The effectiveness of MetaGPT's instantiated workflows relies heavily on the standardized outputs for each action. These outputs leverage expert domain knowledge and industry best practices to adapt workflows to specific roles and contexts. Structured output designs serve the following purposes:

- Standardized outputs promote consistent LLM results that are predictable, repeatable, and in line with agent responsibilities. They guide high-quality, structured, and task-specific LLM generation by setting output expectations.
- Additionally, standardized schemas act as blueprints that constrain LLM behavior within appropriate boundaries for the role. This helps maintain focus on the target task and prevents digressions. As actions form part of comprehensive role-based guidelines, this role-conscious steering ensures that outputs align with real-world quality standards.

Our method's capabilities and effectiveness are demonstrated on more complex system designs beyond simple game examples, such as content recommendation engines, search algorithm frameworks, and LLM-based operating systems. More detailed results can be found in Appendix A.

As depicted in Figure 4, the architect agent generates a detailed system-level diagram illustrating the software architecture. The diagram includes clear definitions of crucial modules such as User, CollaborativeFilteringModel, and Recommender. The diagram is supplemented with information about the key fields and methods in each module, helping engineers better understand the fundamental workflows and functional components.

Furthermore, the design incorporates the calling relationships between modules, following the principles of separation of concerns and loose coupling at the system level. The translation from human natural language to a structured technical design provides specific and practical information that can aid in the implementation of engineering projects, going beyond general summaries.

Although the system design provides an overall framework and modular design, it is still insufficient for engineers to implement complex system coding. Engineers still require additional details on how to perform operations within

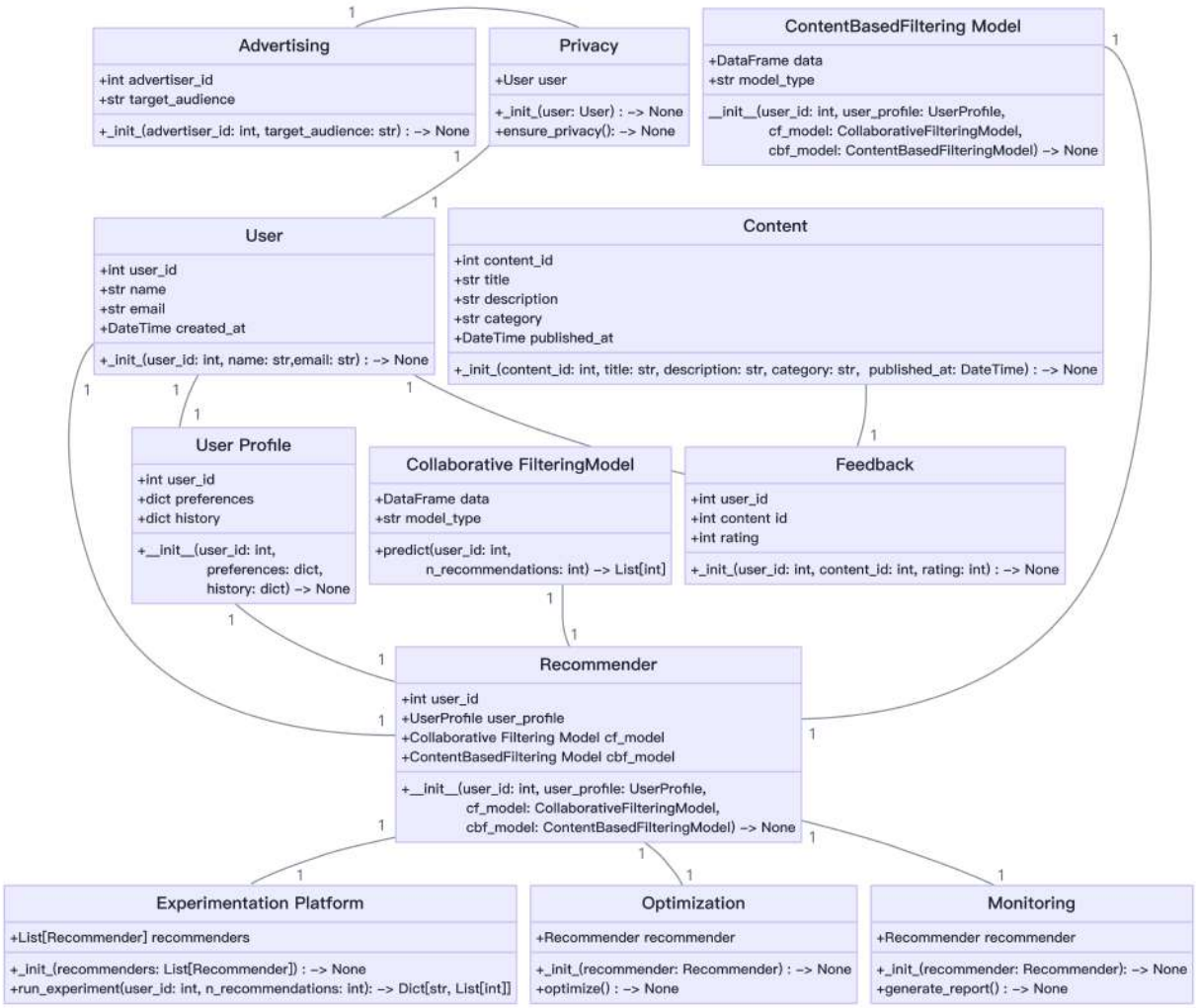

Figure 4: **The system interface design is automatically generated by the architect agent in MetaGPT.** Taking content recommendation engine development as an example.

and between modules to convert the design into functional code. As illustrated in Figure 5, the architect also creates a sequence flow diagram based on the system interface design, depicting the processes, objects involved, and the sequence of messages exchanged between them required to carry out the functionality. As mentioned, these supplementary details make collaborations easier, especially for engineers and project managers who are responsible for detailing the code design.

Hence, the consistent, synergistic outputs of the architect roles are crucial for improving code quality by simplifying the engineer's task of translating specifications into functional code. They reduce ambiguity, misinterpretations, and confusion that can arise from freeform natural language.

In conclusion, the design and implementation of standardized outputs in MetaGPT offer a powerful tool for handling complex tasks. The conversion of complex tasks, as defined in natural language, into standardized outputs promotes collaborative consistency, thereby reducing the need for excessive dialogue turns that could result in incoherence. Furthermore, it can clearly and consistently represent structural information, which can be challenging to convey unambiguously through natural language alone, especially for LLM-based agents. By providing these structured and standardized outputs, different agents gain a clear and aligned understanding of their tasks and responsibilities. This approach not only streamlines communication but also enhances the ability of LLM-based multi-agent systems to administer and execute complex tasks more effectively.

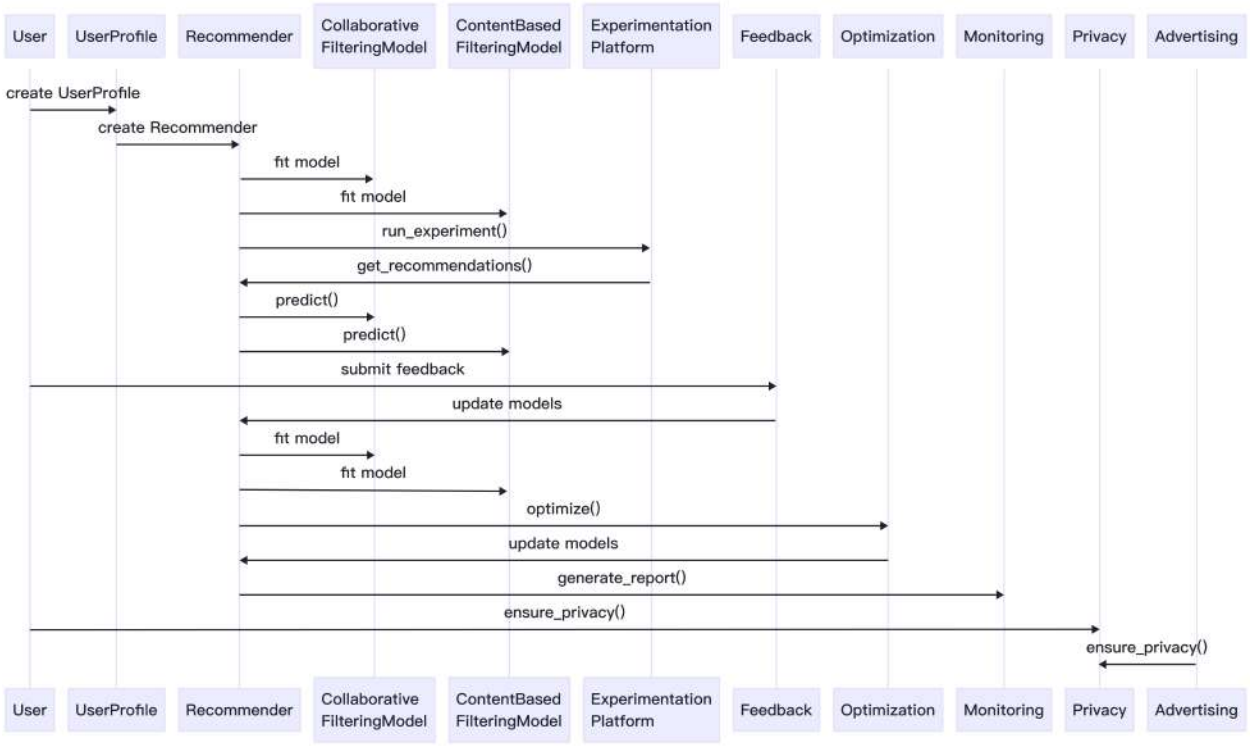

Figure 5: **The sequence flow diagram is automatically generated by the architect agent in MetaGPT.** Taking content recommendation engine development as an example

3.2.4 Knowledge Sharing Mechanism & Customized Knowledge Management

In MetaGPT, each agent proactively curates personalized knowledge by retrieving relevant historical messages from shared environment logs. Instead of passively relying on dialogues, agents utilize role-based interests to extract relevant information. Specifically, the environment replicates messages to provide a unified data repository. Agents register subscriptions based on message types that are meaningful for their roles. Matching messages are automatically dispatched to notify the appropriate agents. Internally, agents maintain a memory cache that indexes subscribed messages by their content, publishing agent, and other attributes. The retrieval mechanism allows agents to acquire additional contextual details by querying this storage. Updates synchronize across the memories of linked agents to maintain a consistent view. This decentralized yet unified access pattern mirrors how human organizations function. Team members have shared records but customize their views based on their responsibilities. By structuring information flows based on agent roles, MetaGPT enables autonomous agents to effectively access relevant knowledge.

As previously discussed, each agent in MetaGPT maintains a memory cache that indexes subscribed messages relevant to its role, allowing for personalized knowledge curation. Specifically, the centralized replication of messages creates a unified data source. Agents can register subscriptions to automatically receive role-relevant messages from this source. Internally, the agent indexes memory caches by content, source, and attributes to facilitate quick retrieval in the relevant context. Rather than relying on one-size-fits-all communications, this decentralized yet federated knowledge ecosystem mirrors the way human teams customize information views based on individual responsibilities while still relying on shared records.

Message sharing. Whenever an agent generates a message, it is replicated to the shared environment log, creating a single source of truth. This ensures all agents have access to the same information.

Role-based subscriptions. Agents can register subscriptions based on their roles and the types of messages that are meaningful to them. This is done based on predefined criteria that align with the agent's responsibilities and tasks.

Message dispatch. When a new message meets the subscription criteria, it is automatically dispatched to notify the relevant agent. This active dissemination of information prevents agents from missing out on important updates.

Memory caching and indexing Agents maintain an internal memory cache where the subscription messages are stored and indexed by their content, senders, and recipients. This allows for efficient information storage and retrieval.

Contextual retrieval. The environment maintains a shared memory pool that supports caching and indexing. Meanwhile, agents can query their internal memory as needed to obtain contextual details relevant to their current task. This helps in refining their understanding and making better decisions.

Updates synchronization. Any updates or changes made to the messages are synchronized across all linked agent memories to maintain a consistent view of the information. This ensures that all agents have access to the most up-to-date data.

By aligning information flows around agent roles, MetaGPT empowers autonomous agents to efficiently provide relevant knowledge. This pattern mirrors how human organizations function. While everyone has access to shared records, individuals customize their views based on their specific responsibilities. MetaGPT combines centralized knowledge sharing with personalized role-based memory caches, allowing for customized knowledge management. This reduces irrelevant data and provides a common context. This balance team coordination with individual efficiency.

3.3 A Multi-Agent Collaboration Example

MetaGPT is capable of generating entire software systems from a single line of user instruction. This section elaborates on Figure 3 with a more detailed breakdown of MetaGPT that uses waterfall SOP. When prompted with "Make the 2048 sliding tile number puzzle game":

Alice (Product Manager): ready to WritePRD. When the boss (user) puts forward a product requirement, Alice, the product manager, will draft 7 pieces of documents, including Product goals, User stories, Competitive Analysis (in text and as a quadrant chart), Requirement analysis, Requirement Pool, and UI design. Before Alice hands off her work to the next LLM role according to the waterfall SOP, her work will first be reviewed, as demonstrated in Figure 3. After the review, Alice will post her work into the message queue under the "WritePRD" category. Bob, our LLM architect, is subscribed to the "WritePRD" category. Additionally, MetaGPT will notify the users of their current API usage in dollar amounts:

```
Total running cost: $0.072 | Max budget: $3.000 | Current cost: $0.072, prompt_tokens=848,
completion_tokens=771
```

Bob (Architect): ready to WriteDesign. Given the requirement analysis and feasibility analysis from Alice, our architect will draft a system design for the project, which starts with a high-level implementation approach:

```
Implementation approach: We will use Python's built-in libraries for the core game logic.
For the GUI, we will use the open-source library Pygame, which is a set of Python modules
designed for writing video games. It includes computer graphics and sound libraries. For the
high score tracking feature, we will use SQLite, a C library that provides a lightweight
disk-based database. SQLite allows us to persist the high score even afterthe game is closed.
For testing, we will use Pytest, a mature full-featured Python testing tool.
```

Following the implementation plan, architect Bob will create a list of files to atomize the complicated logic of the plan into a dozen files:

```
["main.py",
"game.py",
"gui.py",
"database.py",
"test_game.py",
"test_gui.py",
"test_database.py"]
```

and generate **mermaid diagrams** to define **data structures and interfaces** along with **program sequence flow diagram**.

Bob will proceed to generate file directories and empty files according to the design, and post all his work (documents and diagrams) into the message queue under the "WriteDesign" category.

Eve (Project Manager): ready to WriteTasks. Eve is subscribed to the "WriteDesign" category and uses all the previously drafted UI, system design, and API design documents to break down the project into simpler and more actionable tasks, usually at the complexity level of a single file:

```
## Logic Analysis
[
 ("main.py", "Contains the main game loop and handles user input."),
 ("game.py", "Implements the game logic, including the board, score, and game over condition."),
 ("gui.py", "Handles the graphical user interface, including drawing the board, score, and game
 over screen."),
 ("database.py", "Manages the high score database, including getting and updating the high score."),
 ("test_game.py", "Tests the game logic."),
 ("test_gui.py", "Tests the graphical user interface."),
 ("test_database.py", "Tests the high score database.")
]

## Task list
[
    "game.py",
    "database.py",
    "gui.py",
    "main.py",
    "test_game.py",
    "test_database.py",
    "test_gui.py"
]

## Shared Knowledge
"""
'game.py' contains the 'Game' class, which includes the game logic.
'database.py' contains the 'Database' class, which manages the high score database.
'gui.py' contains the 'GUI' class, which handles the graphical user interface.
'main.py' contains the main game loop and handles user input.
"""
```

Additionally, Eve, the project manager, will list out the project dependencies, including third-party packages written in Python as well as other languages and API specs. When there are missing details, the review/reflection process will iteratively improve the generated content in the current phase until the reviewer is satisfied. e.g.

```
## Anything UNCLEAR
There is no main entry point defined in the project. We need to clarify whether the main
game loop will be in 'main.py' or in another file.
```

Alex (Engineer): ready to WriteCode and WriteCodeReview. Given concrete tasks and a coherent overall implementation plan in the form of "WriteTasks" messages, the LLM-based agent has enough information to generate error-free code. Alex will go through the list of files in the specified order and generate each file. On the first try, MetaGPT successfully generated an error-free 2048 sliding tile puzzle game. All these are from a single line of user instruction.

Figure 6: The MetaGPT-generated runtime interface for the 2048 sliding tile puzzle game.

4 Experiments

4.1 Code-generation Benchmarks

Task Selection. In order to assess the performance of the MetaGPT framework in code generation, particularly in single-file scenarios, and demonstrate the effectiveness of its multi-agent collaboration, we conduct evaluations using two open-source benchmarks, HumanEval[24] and MBPP[25], to evaluate the code generation performance. The HumanEval benchmark includes 164 handwritten programming tasks. Each task includes a function specification, description, reference code, and multiple unit tests. The MBPP benchmark consists of 427 Python programming tasks that have been manually validated. These tasks cover fundamental programming concepts and standard library functionalities. Each task includes a description, reference code, and multiple automated test cases.

Baselines & Metric We utilize the unbiased version of Pass@k as presented by [24], to evaluate the functional accuracy of the top-k generated codes. This is done by running test cases, described as follows:

$$\text{Pass @k} = \mathbb{E}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right]$$

This paper primarily focuses on evaluating the Pass@1 because we consider the framework's ability to achieve code generation success in a single pass to be of great importance. We conduct a comparative evaluation by comparing recent domain-specific LLMs in the code generation field, including AlphaCode [28], Incoder [43], CodeGeeX [44], and CodeX [24] and CodeT [45], with general domain LLMs such as PaLM [46], GPT-4 [47], and MetaGPT. Among them, MetaGPT utilizes 4 agents, namely Engineer, Product Manager, Architect, and Project Manager, with the GPT4-32k serving as the underlying language model.

In the HumanEval and MBPP datasets, we modified specific role-based constraint prompts to request the MetaGPT framework to generate individual functions instead of complete classes. This modification is made to facilitate automated testing in subsequent stages.

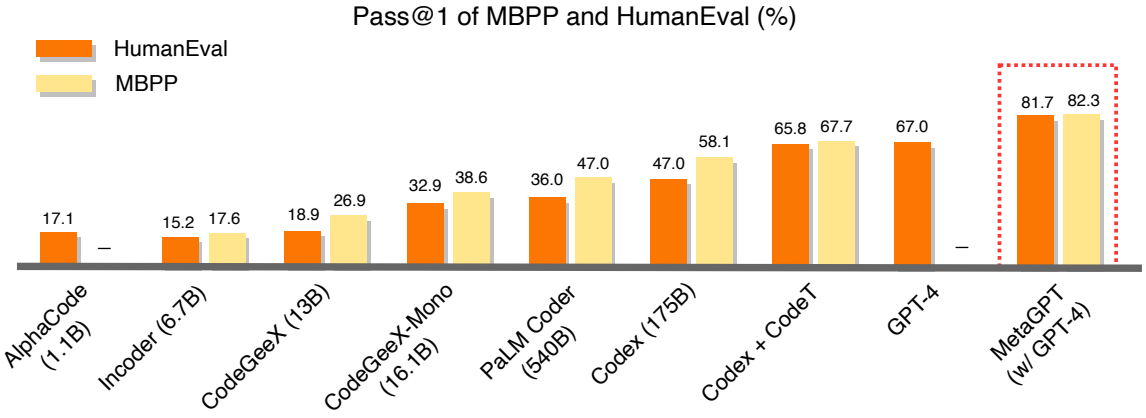

Figure 7: The percentage of pass rates on the MBPP and HumanEval with a single attempt.

Experimental Result Figure 7 demonstrates that our MetaGPT significantly outperforms other methods, including professional code generation framework, *i.e.*, CodeX, CodeT and the most powerful language model currently available GPT-4 in both the HumanEval and MBPP benchmarks, setting a new benchmark for state-of-the-art performance. The primary distinction between our framework and earlier approaches lies in the multi-agent interaction and the incorporation of human standard operating procedures. These factors significantly contribute to the enhanced human-like problem-solving capacity of the development team.

Notably, as GPT-4 possesses a structure akin to a pure engineering role, the collaborative design of our multi-agent system in MetaGPT reduces programming ambiguities. As a result, when MetaGPT collaborates with GPT-4, there is

a 21.9% relative improvement in the Pass@1 metric on the HumanEval benchmark compared to the most proficient single-agent baseline, GPT-4.

4.2 Complex Task Evaluation Methodology

Evaluation Metrics. To precisely evaluate the outcomes of our experiments, we formulate a set of metrics that consider code quality, documentation, cost statistics, cost of revisions, and code executability.

- **Code statistics**
  - **Total number of code files:** This metric reflects the scale of the coding effort by measuring the number of unique code files generated.
  - **Total number of lines in code files:** This metric provides an overview of the volume of code generated by tallying all lines in the code files.
  - **Average number of lines per code file:** This measurement indicates the typical code complexity and is calculated by finding the mean number of lines in each code file.
- **Documentation statistics**
  - **Total number of document files:** This metric illustrates the total number of documents generated. It counts the number of unique documentation files created.
  - **Total number of lines in the document:** This metric measures the amount of written content in the documentation by counting all lines in the document files.
  - **Average number of lines per document file:** This metric indicates the average length and complexity of the documentation files.
  - **Total number of document types:** This metric represents the variety of documentation types produced during the execution of a task.
- **Cost statistics**
  - **Total prompt tokens:** This metric reflects the level of system interaction required and is measured by counting the number of prompt tokens used during task execution.
  - **Total completion tokens:** This metric indicates the volume of output produced, measured by counting the number of completion tokens during task execution.
  - **Time costs:** This refers to the efficiency of the task execution process, which is measured by the total time taken to complete the task.
  - **Cost of money:** This metric measures the cost-effectiveness of the task execution process. It is calculated as the total monetary cost incurred during task execution.
- **Cost of revisions:** This metric measures the amount of maintenance effort needed for the code. A higher value suggests a higher level of code improvement and debugging.
- **Code executability**
  - **Functional quality of the generated code:** Assessed on a grading scale from 'F' for total failure to 'P' for flawless execution and perfect conformity to task specifications.
    - **F** for complete failure, scoring 0 points. The code is non-functional or deviates completely from the specifications.
    - **R** for runnable code, scoring 1 point. The code executes, but it may not meet all workflow requirements.
    - **W** for largely expected workflow, scoring 2 points. The code executes and aligns mostly with the specifications.
    - **P** for a perfect match to expectations, scoring 3 points. The code functions flawlessly, and the output corresponds exactly to the specifications.

Experiments Settings. We conduct seven experiments using MetaGPT within a Python environment (version 3.9.6). These experiments are designed to demonstrate the versatility of the system across various scenarios, including gaming, web development, and data analysis. Specifically, considering the task complexity, we conduct multiple experiments and select the one with the best performance as the result. For each framework, we conduct 5 trials, allowing a maximum of 3 code revisions within each trial. We utilize the experimental code from MetaGPT version 8cc8b80 with $\text{GPT}4-32\text{k}$ as the underlying language model. Experiment-specific configurations include a maximum token consumption limit of $1,500$, an investment cap of 3, and a maximum of 5 iterations. The code review feature is enabled, and we use

mermaid.js[2] for PDF and diagram generation. Each project is subjected to a single-generation process. The full experimental log is available in Appendix B.

To comprehensively evaluate the performance of the MetaGPT framework, we conduct offline experimental evaluations across a broad range of over 70 tasks. These tasks are designed to test the feasibility and general applicability of the framework. This diverse collection of tasks, covering various domains and levels of complexity, has been meticulously chosen to offer a comprehensive assessment of MetaGPT's potential. Each task is evaluated based on key metrics, including code statistics, documentation statistics, cost statistics, cost of revisions, and the success rate of code execution. For a more detailed understanding of our experimental setup and results, we have included a subset of these tasks in Appendix B. On average, it takes 516 seconds and $1.12 to create a project using the MetaGPT framework. This project typically consists of 4.71 code files and 3 types of documents. After no more than three bug fixes, the project's success rate can reach 79.48%.

4.3 Comparison with Alternative Approaches

This section provides a clear comparison of MetaGPT, AutoGPT, LangChain with the Python Read-Eval-Print Loop (REPL) tool and AgentVerse's capabilities, as depicted in Table 1. It also includes experiments to quantify the performance of different frameworks.

Framework Capability Comparison. MetaGPT distinguishes itself through its wide-ranging functionality. Unique to MetaGPT is the capability to generate PRDs and technical designs, demonstrating its holistic approach to project execution. MetaGPT is the only framework capable of API interface generation, which provides an advantage in rapid API design prototyping scenarios.

Code review, a crucial stage of the development process, is available in both MetaGPT and AgentVerse but is notably absent in AutoGPT and LangChain. MetaGPT further differentiates itself by integrating precompilation execution, a feature that promotes early error detection and enhances code quality. In terms of collaborative features, both MetaGPT and AgentVerse support role-based task collaboration, a mechanism that encourages multi-agent collaboration and enhances teamwork by assigning tasks to specific roles. However, MetaGPT exclusively offers role-based task management, a feature that not only breaks down tasks but also oversees their administration, highlighting its extensive project management capabilities.

When evaluating code generation abilities, all these frameworks exhibit proficiency. However, MetaGPT provides a more comprehensive solution, addressing a broader range of aspects in the development process and offering a complete toolset for project management and execution.

This comparison is based on the current state of the respective frameworks. Future updates might add or modify the features of these tools. However, as of this analysis, MetaGPT outperforms its counterparts in offering a more comprehensive and robust solution for project execution.

Quantitative Experiment Comparison. To evaluate the performance of various frameworks such as MetaGPT, AutoGPT, LangChain and AgentVerse, we conduct experiments across 7 diverse tasks. These tasks include Python games generation, CRUD code generation, and simple data analysis. This approach is designed to illuminate the distinctive strengths and weaknesses of each framework under examination. The results are shown in Table 2.

Table 1: **Comparison of capabilities across MetaGPT, AutoGPT, LangChain, and AgentVerse.** Note that '✓' indicates the presence of a given feature in the respective framework.

Framwork capabiliy	AutoGPT	LangChain w/ Python REPL tool	AgentVerse	MetaGPT
PRD generation				✓
Tenical design genenration				✓
API interface generation				✓
Code generation	✓	✓	✓	✓
Precompilation execution				✓
Role-based task management				✓
Code review			✓	✓
Role-based task collaboration			✓	✓

[2]https://github.com/mermaid-js/mermaid

Table 2: **Comparison of task executability between AutoGPT, LangChain, AgentVerse, and MetaGPT.** The tasks are scored based on a grading system from '0' to '3', where '0' denotes 'complete failure', '1' denotes 'runnable code', 2 denotes 'largely expected workflow', and '3' denotes 'perfect match to expectations' (shown in Section 4.2).

Task	AutoGPT	LangChain w/ Python REPL tool	AgentVerse	MetaGPT
Flappy bird	0	0	0	**1**
Tank battle game	0	0	0	**2**
2048 game	0	0	0	**2**
Snake game	0	0	0	**3**
Brick breaker game	0	0	0	**3**
Excel data process	0	0	0	**3**
CRUD manage	0	0	0	**3**

As shown in Table 2, MetaGPT demonstrates robust performance across a diverse set of tasks, achieving successful execution in all tasks. Flappy Bird, a game that requires high levels of interaction, presents challenges even for advanced programming systems. While MetaGPT is capable of producing runnable code for the game, the result is not completely flawless due to the strict constraints and limited resources available for manual adjustments. In contrast, the competing frameworks, AutoGPT, LangChain, and AgentVerse, cannot achieve successful execution in any of the tasks, highlighting a clear distinction in the effectiveness of the MetaGPT framework.

Further details about MetaGPT's runtime statistics are provided in Table 3. On average, each project includes 4.71 code files (including but not limited to formats such as CSS, py, and js), with an average of 42.99 lines of code per file and an average of 191.57 lines of code in total. Concerning PRD files, each project generates an average of four PRD files (including pdf, mmd, svg, and png with the same name as one file). Additionally, project documentation typically consisted of three types of asset files per project. These include product requirement documents, API documentation, and system architecture documentation. On average, each document contained 80 lines.

Table 3: **The statistical analysis of MetaGPT in software development.** The minimum (Min), maximum (Max), and average (Avg.) values for various statistical indexes are reported. '#' denotes 'The number of

Statistical index	Min	Max	Avg.
# Code files	3.00	6.00	4.71
# Asset files	3.00	3.00	3.00
# Lines of code(single file)	17.00	96.00	42.99
# Lines of code(whole project)	76.00	316.00	191.57
# Prompt tokens	21934.00	32517.00	26626.86
# Completion tokens	5312.00	7104.00	6218.00
Cost statistics	$ 0.90	$ 1.35	$ 1.09
Cost of revisions(only **R**, **W** and **P**)	0.00	2.00	0.60

In terms of cost analysis, each project consumes on average $26,626.86$ tokens for prompts and $6,218.00$ tokens upon task completion, resulting in a total cost of $\$ 1.09$ for completing the tasks. The entire construction process tasks 517.71 seconds. Compared to traditional software engineering development timelines and costs, MetaGPT's time and financial expenditures are less than one thousandth.

We use code of revisions as a metric to resolve errors encountered during project execution. This is achieved through methods such as dependency replacement, code modifications, or other corrective actions until successful execution or the next issue arises. The maximum allowed number of revisions in the code is three. Exceeding this limit will result in the project being considered a failure if it continues to encounter errors. Regarding code of revisions, each project requires on average 0.6 revisions. The majority of these issues are related to dependencies, unavailable resources, and missing parameters. The overall success rate (WP rate - running successfully and generally meeting expectations) stands at 100.0%.

Despite AutoGPT being the most prevalent single-agent framework currently, it was unable to complete any tasks using the GPT4-32k configuration with its default setup in our experiments. As a single agent, its characteristics must be established manually before the execution process, and these cannot be altered midway. AutoGPT is capable of

decomposing user-provided tasks into multiple smaller subtasks and executing them sequentially. However, our observations throughout the experimental procedure highlight a significant drawback of AutoGPT: its lack of completeness evaluation and the limited expertise of a single agent.

AutoGPT lacks a mechanism for evaluating task completeness. It simply marks a task as completed after saving the generated results, without conducting any additional validation or completeness checks. To improve the efficiency of the loop, we utilize the AutoGPT implementation in LangChain and integrate it with the Python REPL tool. We intend to enable AutoGPT to debug and refine the code it generates. However, the lack of specialized knowledge prevents it from utilizing the feedback provided by the interpreter to improve its success rate. Consequently, the tasks are unsuccessful due to the generation of incomplete and non-functional code.

Despite implementing three specialized roles: Writer, Tester, and Reviewer, AgentVerse fails to perform well in all benchmark tasks. These roles perform collaboration in the manner of online judges (OJs). The conversation within AgentVerse primarily revolves around three roles: the Writer, who creates code; the Tester, who identifies areas of failure or error messages during code execution; and the Reviewer, who suggests modifications. This is similar to the engineer we implement in MetaGPT, which is capable of coding and code review. The dialogues within AgentVerse primarily focus on the code itself rather than the overall task.

However, the absence of roles responsible for breaking down large tasks into smaller, more manageable ones puts AgentVerse at a disadvantage. This lack of task decomposition and division of work significantly reduces the likelihood of completing more complex tasks. The shortcomings of AgentVerse underscore the importance of clear role delineation and strategic task segregation across different phases when dealing with complex problem-solving scenarios.

4.4 Ablation Studies

Our ablation studies involve systematically reducing the number of roles involved in the development process and subsequently examining the effects on the executable output. We select *Brick Breaker* and *Gomoku* as tasks due to their complexity which requires a team with diverse roles and a multi-step workflow. This is similar to the real-world scenario of game development, where clear role divisions and teamwork are essential.

Our initial experiments with MetaGPT are conducted using a fully-staffed team, consisting of four distinct roles: a ProductManager, an Architect, a ProjectManager, and an Engineer, by the experimental settings outlined in Section. 4.2.

For the ablation studies, we use a set of metrics: **total number of lines in code files**, **cost of money**,**cost of revisions**, and **code executability**, as defined in Section. 4.2 . These metrics are selected to facilitate the easy quantification of factors such as code generation quality, task cost-effectiveness, and functional code quality. After obtaining results from this fully equipped team, we progressively remove the roles of Architect, ProjectManager, and ProductManager in subsequent experiments, assessing how their absence impacts the overall task performance.

As depicted in Table 4 and Table 5, removing either the Architect or ProjectManager from the team results in a moderate decrease in code statistics. For Brick Breaker, there are 29 fewer lines, and for Gomoku, there are 33 fewer lines when the Architect is removed. Similarly, removing the Project Manager results in a reduction of 14 lines for Brick Breaker and 47 lines for Gomoku. Additionally, 1-2 additional revisions are necessary. However, the overall executability of the task is largely preserved. More significant reductions in code volume and increases in required fixes become evident when transitioning from three roles to just two: the ProductManager and Engineer. In this scenario, the game tests show a decrease of 62 and 63 lines, respectively, and an increase of 2 and 3 lines in the revisions.

Additionally, it is critical to note that when the team is reduced to a single agent, the code's executability significantly deteriorates. Even with the additional costs incurred for revision, the code remains non-executable. Therefore, the presence of multiple roles not only enhances code quality but also reinforces the robustness and feasibility of the code implementation.

In summary, through quantitative comparisons across conditions, we underscore the advantages of specialized multi-agent frameworks for complex tasks. Such frameworks validate the importance of role modularity and collaboration for holistic task completion.

5 Discussion and Future Work

Despite the immense potential of MetaGPT in automating end-to-end processes, it also has several limitations. Primarily, it occasionally references non-existent resource files like images and audio. Furthermore, during the execution of complex tasks, it is prone to invoking undefined or unimported classes or variables. These phenomena are widely

Table 4: **Ablation study on roles in Brick breaker game development.** '#' denotes 'The number of', 'Product' denotes 'Product manager', and 'Project' denotes 'Project manager'. 'F' denotes 'complete failure', '**R**': runnable code, '**W**': largely expected workflow, '**P**': perfect.

Engineer	Product	Architect	Project	# agents	Lines of code	Cost of money	Cost of revisions	Code executability
✓				1	89	**$ 0.876**	8	F
✓	✓			2	115	$ 1.022	4	R
✓	✓	✓		3	177	$ 1.204	2	W
✓	✓		✓	3	162	$ 1.221	2	R
✓	✓	✓	✓	**4**	**191**	$ 1.350	**1**	**P**

Table 5: **Ablation study on roles in Gomoku game development.** '#' denotes 'The number of', 'Product' denotes 'Product manager', and 'Project' denotes 'Project manager'. 'F' denotes 'complete failure', '**R**': runnable code, '**W**': largely expected workflow, '**P**': perfect.

Engineer	Product	Architect	Project	# agents	Lines of code	Cost of money	Cost of revisions	Code executability
✓				1	77	**$ 0.953**	12	F
✓	✓			2	109	$ 1.095	9	R
✓	✓	✓		3	172	$ 1.299	6	R
✓	✓		✓	3	186	$ 1.274	5	R
✓	✓	✓	✓	**4**	**219**	$ 1.420	**4**	**W**

attributed to the hallucinatory tendency inherent in large language models and could be handled by a more clear and efficient agent collaboration workflow.

6 Conclusion

In this work, we present MetaGPT, a promising framework for collaborative agents using SOPs that leverages LLMs to mimic efficient human workflows. MetaGPT is a meta programming technology that utilizes SOPs to coordinate LLM-based multi-agent systems. Specifically, to encode SOPs into prompts, MetaGPT manages multi-agents through role definition, task decomposition, process standardization, and other technical designs. It eventually completes the end-to-end development process with only a one-line requirement.

Our illustrative example in software development showcases the potential of this framework through detailed SOPs and prompts. The experimental results demonstrate that our MetaGPT can produce comprehensive solutions with higher coherence compared to existing conversational and chat-based multi-agent systems.

We believe that this work opens up new possibilities for the way multi-agents interact and cooperate, redefines the landscape of complex problem-solving, and points to a potential pathway toward Artificial General Intelligence.

References

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

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

[3] Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. *arXiv preprint*, 2023.

[4] Zhenhailong Wang, Shaoguang Mao, Wenshan Wu, Tao Ge, Furu Wei, and Heng Ji. Unleashing cognitive synergy in large language models: A task-solving agent through multi-persona self-collaboration. *arXiv preprint*, 2023.

[5] Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate, 2023.

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

[7] Rui Hao, Linmei Hu, Weijian Qi, Qingliu Wu, Yirui Zhang, and Liqiang Nie. Chatllm network: More brains, more intelligence. *arXiv preprint*, 2023.

[8] Hongxin Zhang, Weihua Du, Jiaming Shan, Qinhong Zhou, Yilun Du, Joshua B Tenenbaum, Tianmin Shu, and Chuang Gan. Building cooperative embodied agents modularly with large language models. *arXiv preprint*, 2023.

[9] Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, et al. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint*, 2023.

[10] Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. Communicative agents for software development, 2023.

[11] R.M. Belbin. *Team Roles at Work*. Routledge, 2012.

[12] Agile Manifesto. *Manifesto for agile software development*. Snowbird, UT, 2001.

[13] T. DeMarco and T.R. Lister. *Peopleware: Productive Projects and Teams*. Addison-Wesley, 2013.

[14] Michael Wooldridge and Nicholas R. Jennings. Pitfalls of agent-oriented development. In *Proceedings of the Second International Conference on Autonomous Agents*, 1998.

[15] Ian R McChesney and Seamus Gallagher. Communication and co-ordination practices in software engineering projects. *Information and Software Technology*, 2004.

[16] Henry van Dyke Parunak and James Odell. Representing social structures in uml. In *ICAS*, 2001.

[17] Cristiano Castelfranchi. Founding agents'" autonomy" on dependence theory. In *ECAI*, 2000.

[18] Jacques Ferber and Jean-Pierre Müller. Influences and reaction: a model of situated multiagent systems. In *ICMAS*, 1996.

[19] Jacques Ferber and Olivier Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *ICMAS*, 1998.

[20] Wikipedia contributors. Metaprogramming — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Metaprogramming&oldid=1151434942`, 2023. [Online; accessed 6-August-2023].

[21] Torantulino et al. Auto-gpt. `https://github.com/Significant-Gravitas/Auto-GPT`, 2023.

[22] Harrison Chase. LangChain. `https://github.com/hwchase17/langchain`, 2022.

[23] chenweize1998, yushengsu thu, chanchimin, libowen2121, Xial-kotori, Dr-Left, tzw2698, and zhouxh19. Agentverse. `https://github.com/OpenBMB/AgentVerse`, 2023.

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

[25] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.

[26] Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. Lever: Learning to verify language-to-code generation with execution. In *ICML*, 2023.

[27] Marta Skreta, Naruki Yoshikawa, Sebastian Arellano-Rubach, Zhi Ji, Lasse Bjørn Kristensen, Kourosh Darvish, Alán Aspuru-Guzik, Florian Shkurti, and Animesh Garg. Errors are useful prompts: Instruction guided task programming with verifier-assisted iterative prompting. *arXiv preprint*, 2023.

[28] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 2022.

[29] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *ICLR*, 2018.

[30] Xinyun Chen, Dawn Song, and Yuandong Tian. Latent execution for neural program synthesis beyond domain-specific languages. *NeurIPS*, 2021.

[31] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint*, 2022.

[32] Noah Shinn, Beck Labash, and Ashwin Gopinath. Reflexion: an autonomous agent with dynamic memory and self-reflection. *arXiv preprint*, 2023.

[33] Bill Yuchen Lin, Yicheng Fu, Karina Yang, Prithviraj Ammanabrolu, Faeze Brahman, Shiyu Huang, Chandra Bhagavatula, Yejin Choi, and Xiang Ren. Swiftsage: A generative agent with fast and slow thinking for complex interactive tasks. *arXiv preprint*, 2023.

[34] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *NeurIPS*, 2022.

[35] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint*, 2023.

[36] Yashar Talebirad and Amirhossein Nadiri. Multi-agent collaboration: Harnessing the power of intelligent llm agents, 2023.

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

[38] Elif Akata, Lion Schulz, Julian Coda-Forno, Seong Joon Oh, Matthias Bethge, and Eric Schulz. Playing repeated games with large language models. *arXiv preprint*, 2023.

[39] joshsny, timconnorz, Alec2435, HaniaA, ryderwishart, and zoan37. Gpteam. `https://github.com/101dotxyz/GPTeam`, 2023.

[40] pchalasani, Mohannadcse, and AshishHoodaIITD. langroid. `https://github.com/langroid/langroid`, 2023.

[41] RunzheYang. Socraticai. `https://github.com/RunzheYang/SocraticAI`, 2023.

[42] Knut Sveidqvist and Contributors to Mermaid. Mermaid: Generate diagrams from markdown-like text. `https://github.com/mermaid-js/mermaid`, 2014.

[43] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

[44] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x, 2023.

[45] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests, 2022.

[46] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways, 2022.

[47] OpenAI. Gpt-4 technical report, 2023.

A Detailed Standardard Output Artifacts

This appendix provides additional examples of standardized output artifacts generated by different roles in MetaGPT during collaborative workflows. These help illustrate the structure and format of outputs that aim to match industry conventions and domain-specific best practices.

ProductManager Outputs The product manager role produces PRD including product goals, user stories, competitive analysis, competitive quadrant chart, requirement analysis and requirement pool.

1) **PRD of 2048 Game Design**

```
## Original Requirements
The boss wants us to create a 2048 game.

## Product Goals

[

    "Create a 2048 game that is easy to understand and play",
    "Ensure the game is engaging and retains user interest",
    "Design the game to be visually appealing and intuitive"
]


## User Stories

[
    "As a user, I want to be able to start a new game easily so that I can play whenever I want",
    "As a user, I want to see my current score and high score, so that I can track my progress",
    "As a user, I want the game to respond quickly to my actions, so that I can play without
    interruptions",
    "As a user, I want the game to have an appealing design, so that I enjoy my playing experience",
    "As a user, I want the game to have different difficulty levels, so that I can challenge myself"
]


## Competitive Analysis

[
    "2048 by Gabriele Cirulli: The original game, simple and intuitive, but lacks variety",
    "2048 Number puzzle game by Estoty Entertainment LLC: Offers different board sizes, but has
    intrusive ads",
    "2048 by Androbaby: Has a clean design, but lacks difficulty levels",
    "2048 by R. App: Offers undo feature, but the design is not very appealing",
    "2048 by Ketchapp: Has a modern design, but the controls are not very responsive",
    "2048 by X-Flow: Offers different game modes, but the user interface is not very intuitive",
    "2048 by Solebon LLC: Has a minimalistic design, but lacks variety in gameplay"
]


## Competitive Quadrant Chart
‘‘‘mermaid quadrantChart

    title Reach and engagement of campaigns
    x-axis Low Reach --> High Reach
    y-axis Low Engagement --> High Engagement
    quadrant-1 We should expand
    quadrant-2 Need to promote
    quadrant-3 Re-evaluate
    quadrant-4 May be improved
```

```
    "2048 by Gabriele Cirulli": [0.8, 0.6]
    "2048 Number puzzle game by Estoty Entertainment LLC": [0.7, 0.4]
    "2048 by Androbaby": [0.6, 0.5]
    "2048 by R. App": [0.5, 0.3]
    "2048 by Ketchapp": [0.7, 0.5]
    "2048 by X-Flow": [0.6, 0.6]
    "2048 by Solebon LLC": [0.5, 0.4]
    "Our Target Product": [0.7, 0.7]
```

```
## Requirement Analysis
The product should be a 2048 game that is easy to play, visually appealing, and engaging.
It should have different difficulty levels and a quick response time. The game should also display
the current score and high score, and have an intuitive user interface.

## Requirement Pool

[
    ("Implement the basic 2048 game mechanics", "P0"),
    ("Design a visually appealing and intuitive user interface", "P0"),
    ("Add different difficulty levels", "P1"),
    ("Display the current score and high score", "P0"),
    ("Ensure a quick response time", "P0")
]


## Anything UNCLEAR
There are no unclear points.
```

2) **PRD of Snake Game Design**

```
## Original Requirements
The boss wants us to create a snake game.

## Product Goals

[
    "Create a snake game that is easy to play and understand",
    "Ensure the game is challenging and engaging to keep users interested",
    "Design the game to be visually appealing and responsive"
]


## User Stories

[
    "As a user, I want to be able to easily control the snake so that I can play the game
    without difficulty",
    "As a user, I want the game to progressively get more difficult so that I am constantly
    challenged",
    "As a user, I want to be able to pause and resume the game so that I can play at my own pace",
    "As a user, I want the game to have a visually appealing design so that I enjoy playing",
    "As a user, I want the game to respond quickly to my actions so that I can play effectively"
]


## Competitive Analysis

[
```

```
    "Python Snake Game: Simple and easy to play but lacks visual appeal",
    "Google Snake Game: Visually appealing and responsive but can be too difficult for beginners",
    "Classic Snake Game: Offers a good level of challenge but lacks modern design elements",
    "Slither.io: Multiplayer game with modern design but can be too complex for some users",
    "Snake '97: Retro design that appeals to nostalgia but lacks modern gameplay elements"
]


## Competitive Quadrant Chart
‘‘‘mermaid quadrantChart

    title Reach and engagement of campaigns
    x-axis Low Reach --> High Reach
    y-axis Low Engagement --> High Engagement
    quadrant-1 We should expand
    quadrant-2 Need to promote
    quadrant-3 Re-evaluate
    quadrant-4 May be improved
    "Python Snake Game": [0.3, 0.6]
    "Google Snake Game": [0.45, 0.23]
    "Classic Snake Game": [0.57, 0.69]
    "Slither.io": [0.78, 0.34]
    "Snake '97": [0.40, 0.34]
    "Our Target Product": [0.5, 0.6]
‘‘‘

## Requirement Analysis
The product should be a snake game that is easy to play, visually appealing, and progressively
challenging. It should have responsive controls and allow users to pause and resume the game.

## Requirement Pool

[
    ("Easy to control snake movement", "P0"),
    ("Progressive difficulty levels", "P0"),
    ("Pause and resume feature", "P1"),
    ("Visually appealing design", "P1"),
    ("Responsive controls", "P0")
]


## Anything UNCLEAR
There are no unclear points.
```

Architect Outputs The software architect role produces technical specifications like system architecture diagrams and interface definitions. We showcase the interface design for search algorithms framework 8, LLM-based operation system 9,Minimalist Pomodoro timer 10, Pyrogue game 11, and Match3 puzzle game 12

Figure 8: The system interface design for search algorithms frameworks automatically generated by the architect agent in MetaGPT.

ProjectManager Outputs The projectmanager role produces technical specifications like sequence flow chart. The examples includes search algorithms frameworks 13, LLM-based operation system 14, Minimalist Pomodoro timer 15

Figure 9: The system interface design for LLM-based operation system automatically generated by the architect agent in MetaGPT.

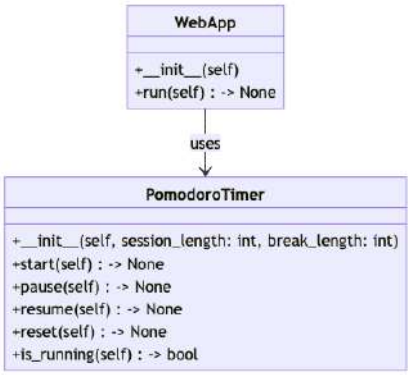

Figure 10: The interface design for Minimalist Pomodoro timer by the architect agent in MetaGPT.

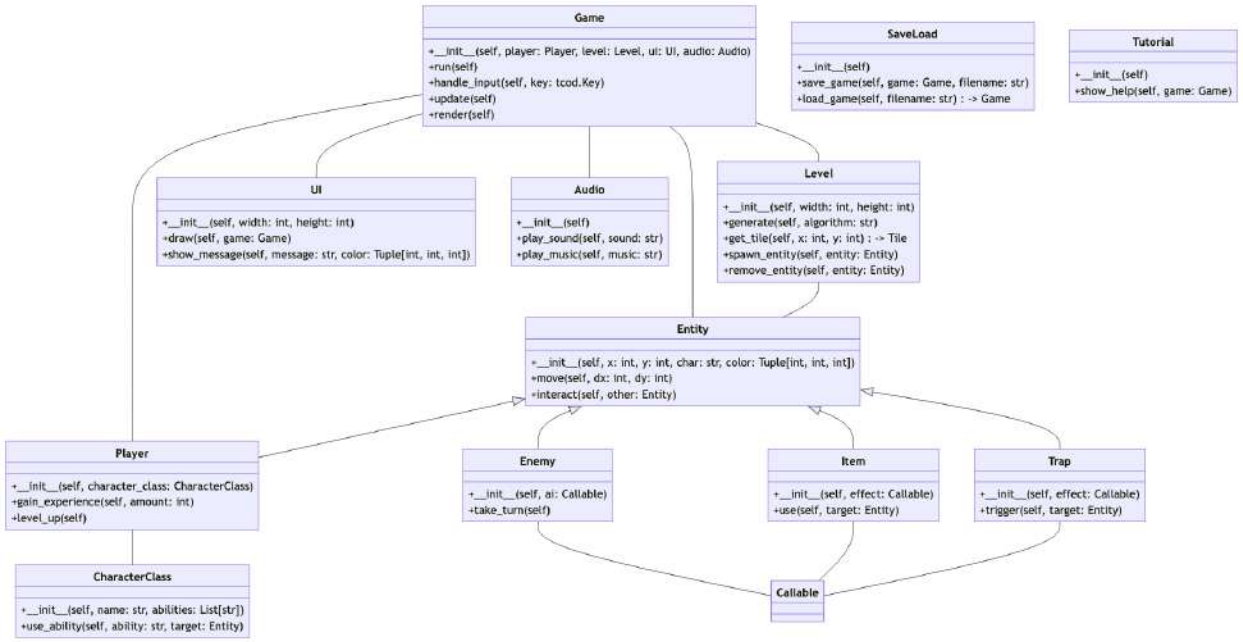

Figure 11: The interface design for Pyrogue game by architect agent in MetaGPT.

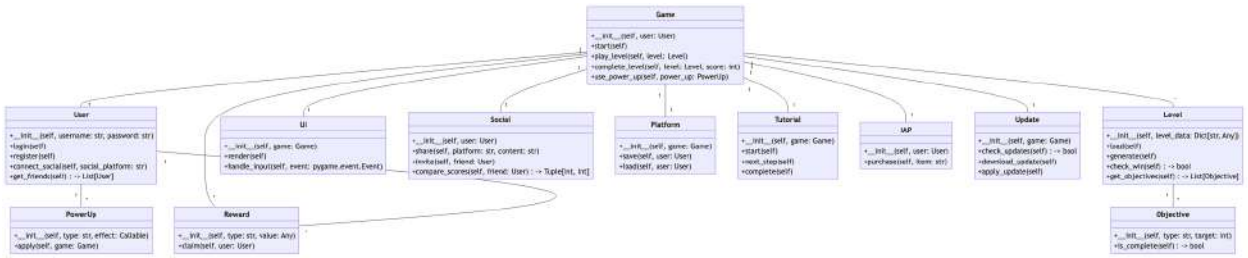

Figure 12: The interface design for Match3 puzzle game by architect agent in MetaGPT.

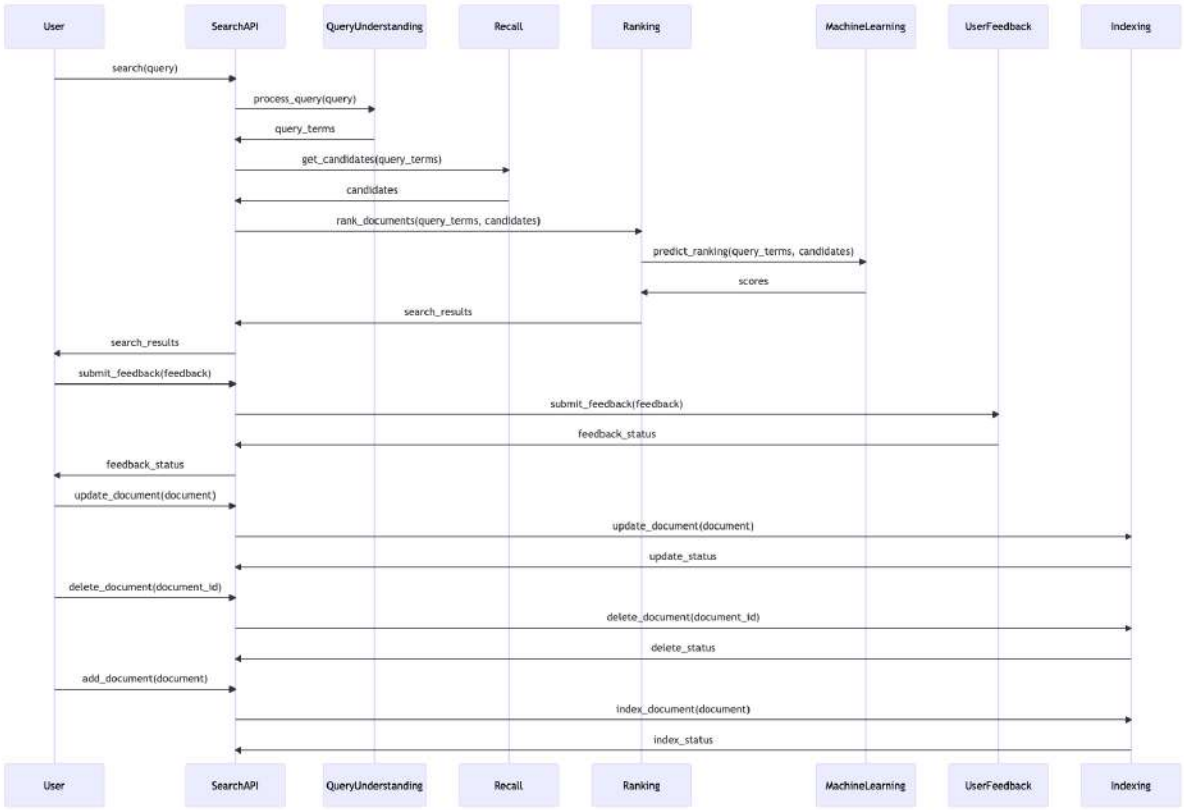

Figure 13: The sequence flow diagram for search algorithms frameworks automatically generated by the project manager agent in MetaGPT.

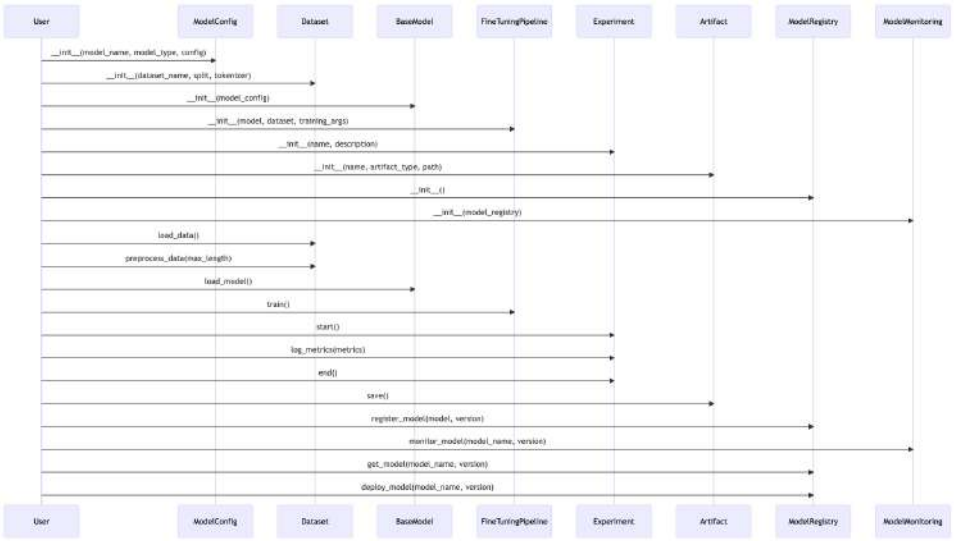

Figure 14: The sequence flow diagram for LLM-based operation system by project manager agent in MetaGPT.

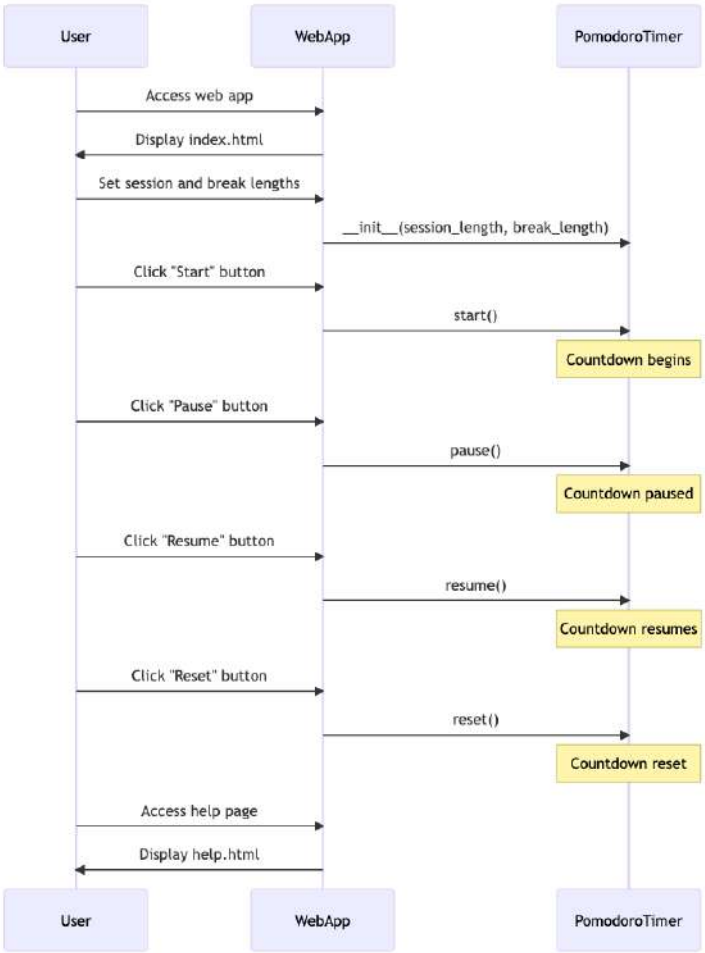

Figure 15: The sequence flow chart for Minimalist Pomodoro timer by the project manager agent in MetaGPT.

B MetaGPT Experimental Tasks Detail

Table 6: Subset of MetaGPT Experimental Tasks' Prompt

Task ID	Task	Prompt
0	Snake game	create a snake game.
1	Brick breaker game	create a brick breaker game.
2	2048 game	create a 2048 game for the web.
3	Flappy bird game	write p5.js code for Flappy Bird where you control a yellow bird continuously flying between a series of green pipes. The bird flaps every time you left click the mouse. If the bird falls to the ground or hits a pipe, you lose. This game goes on infinitely until you lose and you get points the further you go.
4	Tank battle game	create a tank battle game.
5	Excel data process	Write an excel data processing program based on streamlit and pandas,The screen first has an excel upload processing button. After the excel is uploaded, use pandas to display the data content in the excel.The program is required to be concise, easy to maintain, and not over-designed.The program as a whole uses streamlit to process web screen display, and pandas is sufficient to process excel reading and display. Others do not need to introduce additional packages.
6	CRUD manage	Write a management program based on the crud addition, deletion, modification and query processing of the customer business entity,The customer needs to save the information of name, birthday, age, sex, and phone.The data is stored in client.db as a whole, and it is judged whether the customer table exists. If it does not exist, it needs to be created first.When querying, you need to query by name, and when deleting, you also need to delete by name.The program is required to be concise, easy to maintain, and not over-designed.The screen is realized through streamlit and sqlite, no need to introduce other additional packages.
7	Music transcriber	Developing a program to transcribe sheet music into a digital format, Providing error-free transcribed symbolized sheet music intelligence from audio through signal processing involving pitch and time slicing then training a neural net to run Onset Detected CWT transforming scalograms to chromagrams decoded with Recursive Neural Network focused networks.
8	Custom press releases	Create custom press releases, Develop a Python script that extracts relevant information about company news from external sources, such as social media extract update interval database for recent changes. The program should create press releases with customizable options and export writings to PDFs, NYTimes API JSONs, media format styled with interlink internal fixed character-length metadata.
9	Gomoku game	Implement a Gomoku game using Python, incorporating an AI opponent with varying difficulty levels.
10	Weather dashboard	Create a Python program to develop an interactive weather dashboard.

Table 7: **Subset of MetaGPT experimental tasks' details.** An Average (Avg.) of 70 tasks were calculated and 10 randomly selected tasks is included. '#' denotes 'The number of', while 'ID' is 'Task ID'.

ID	Code statistics			Doc statistics			Cost statistics				Cost of revision	Code executability
	#code files	#lines of code	#lines per code file	#doc files	#lines of doc	#lines per doc file	#prompt tokens	#completion tokens	time costs	money costs		
0	5.00	196.00	39.20	3.00	210.00	70.00	24087.00	6157.00	582.04	$ 1.09	1. TypeError	P(3)
1	6.00	191.00	31.83	3.00	230.00	76.67	32517.00	6238.00	566.30	$ 1.35	1. TypeError	P(3)
2	3.00	198.00	66.00	3.00	235.00	78.33	21934.00	6316.00	553.11	$ 1.04	1. lack @app.route('/')	W(2)
3	5.00	164	32.80	3.00	202.00	67.33	22951.00	5312.00	481.34	$ 1.01	1. PNG file missing 2. Compile bug fixes	R(1)
4	6.00	203.00	33.83	3.00	210.00	70.00	30087.00	6567.00	599.58	$ 1.30	1. PNG file missing 2. Compile bug fixes 3. pygame.surface not initialize	W(2)
5	6.00	219.00	36.50	3.00	294.00	96.00	35590.00	7336.00	585.10	$ 1.51	1. dependency error 2. ModuleNotFoundError	P(3)
6	4.00	73.00	18.25	3.00	261.00	87.00	25673.00	5832.00	398.83	$ 0.90	0	P(3)
7	4.00	316.00	79.00	3.00	332.00	110.67	29139.00	7104.00	435.83	$ 0.92	0	P(3)
8	5.00	215.00	43.00	3.00	301.00	100.33	29372.00	6499.00	621.73	$ 1.27	1. tensorflow version error 2. model training method not implement	R(1)
9	5.00	215.00	43.00	3.00	270.00	90.00	24799.00	5734.00	550.88	$ 1.27	1. dependency error 2. URL 403 error	W(2)
10	3.00	93.00	31.00	3.00	254.00	84.67	24109.00	5363.00	438.50	$ 0.92	1. dependency error 2. missing main func.	P(3)
Avg.	4.71	191.57	42.98	3.00	240.00	80.00	26626.86	6218.00	516.71	$1.12	0.51(only R,W or P)	W(2.36)