Mate-Incoder*: An Agentic Isolated Sequential Workflow for Code Generation

Niyazi Ahmet Metin

Department of Computer Science
Sabancı University
ahmet.metin@sabanciuniv.edu

*The name ”Mate-Incoder”, means ”coding friend”, is a wordplay referring to both the author’s surname by pronunciation, and the Incoder model introduced by Meta for code generation [1]. However, this work is not affiliated with Meta or Incoder model.

***Abstract*—Mate-Incoder, a multi-agent pipeline for safe, reliable code generation is presented in this work. Mate-Incoder splits the task into three isolated roles: coding, test-case generation, execution, and connected by an “isolated sequential chat” that implemented from scratch, and refreshes tests each iteration and limits every run to a Docker sandbox. This design prevents both hard coded hacks of Coder and host side side effects of malicious user. On a subset of the MBPP benchmark, the GPT-3.5 based pipeline achieved 48% immediate pass rate and 66% after up to five feedback loops, with preliminary results hinting at further gains on larger LLMs. I release all prompts and implementation details in this paper.**

I. Introduction

Writing correct code from a natural-language description is getting easier every day, thanks to powerful LLMs. But, generally using these code in our programs is hard because they often generate code that compiles but fails at runtime, doesn’t match the expecting outputs. To address this issues, Mate-Incoder splits the workflow into four agents: one to write code, one to design tests, one to run them, and a manager to oversee everything. Unlike existing frameworks, every agent is kept isolated. Coder agent and Testcase Generator agent doesn’t know anything about each other to make sure the Coder doesn’t give hard-coded fixes. Also, tests are regenerated after each iteration so the coder can’t simply memorize answers. In Mate-Incoder, every code run happens inside a Docker sandbox for safety. This “isolated sequential chat” pattern helps catch both faulty code and flawed tests, leading to more robust solutions. The rest of the paper is organized as follows. Section II surveys background on Autogen, sandboxing techniques, GROQ Cloud, and the MBPP dataset. Section III describes the methodology in detail, including agent roles and workflow. Section IV compares Mate-Incoder to related multi-agent systems. Section V presents the experimental results, and Section VI reflects on limitations and Future Directions. Finally, Section VII concludes. Also, all prompts used in this work are available in the Appendix.

II. Background

In this section, I survey the core building blocks used in the proposed system: the Autogen multi-agent framework, safety and sandboxing techniques for safe code execution, the GROQ Cloud inference platform, and the MBPP benchmark. Each of these contributes a key component to the architecture.

A. Autogen

Autogen is a framework built from Microsoft which is designed to simplfy multi-agent workflows by providing a common interface for conversational LLM’s. [2] Autogen traits each agent as a chat participant that can send and receive messages, coordinate tasks and call external tools. The core idea is to let agents collaborate through structured dialogues rather than forcing all logic into a single monolithic prompt.

1) Autogen Conversation Patterns

Autogen supports several conversation patterns to organize agent interactions. Here two of the most common ones are highlighted. For more details, see the original Autogen paper [2], or Microsoft open source conversation patterns documentation [3].

Group Chat

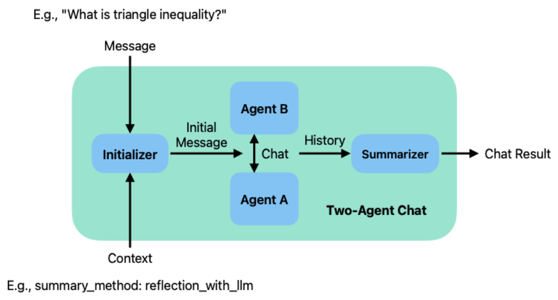

Fig. 1. Group chat pattern

In the group chat pattern, all agents share a single chat history. Every message is broadcast to the entire group, so each agent sees the full context of the conversation. This is simple to implement but can become noisy especially when the number of agents grows, since each agent checks every message in workflow, even if the message is not relevant to its role.

Sequential Chat

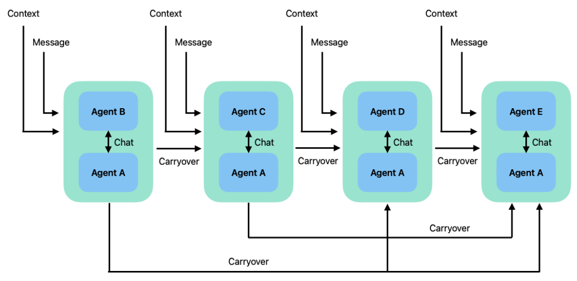

Fig. 2. Sequential chat pattern

Sequential chat arranges agents in a fixed order: Agent A starts conversation with Agent B. After, conversation between Agent A and Agent C starts with the carryover from previous conversation, as shown in Fig.2. Messages flow in a pipeline fashion, which can reduce clutter compared to group chat. However, every agent still sees all prior messages, so there is no isolation in this pattern.

2) Isolation

One limitation of both group and sequential chat -or other conversation patterns- in Autogen is the lack of isolation: agents have access to the entire conversation history. In code generation scenarios, this means that for a multiagent coder application, assuming a coder agent and a testcase generator agent exists, coder agent can see all the testcases that generated by testcase generator agent, and can fix some of them with some 'if' blocks even if it doesn't give a correct solution for the given problem. More information about isolation idea that's implemented in my approach will be given in methodology section.

3) Tool Use

Autogen provides a built-in mechanism for agents to register and invoke external tools directly from the conversation flow. In our implementation, the `ExecutorAgent` uses this API to register a function called `execute_code_batch`, which safely runs generated Python code against test inputs and returns the results. This clear separation between "reasoning" (managed by the LLM) and "execution" (handled by a registered tool) simplifies error handling and auditing. For a complete guide to Autogen's tool integration, see its previously referenced paper or online Tool Use documentation [4].

B. Safety & Sandboxing

When running untrusted or automatically generated code, it is essential to make sure to execute the code in a controlled environment to prevent some unintended side effects or malicious behavior of some user. There are several lightweight sandboxing solutions available, such as Firejail [5], Bubblewrap [6], gVisor [7], Firecracker [8]. These tools can restrict system calls, filesystem access, and resource usage. However, in this work, I acknowledge these tools but do not employ them directly. More information will be given in Limitations part.

Docker Sandboxing

In this work, I use Docker [9] as sandboxing layer to isolate code execution from the host system. A single long running container is launched with:

- **Network isolation:** `--network none` to block all outbound and inbound traffic.
- **Resource limits:** `--memory 512m` and `--cpus 0.5` to cap RAM and CPU usage.
- **Volume mount:** A dedicated host directory is bind-mounted read/write into `/app/sandbox` inside the container.

Before each test iteration, the manager agent, which will be explained further, cleans the workspace inside the container to avoid state leakage. After all tests complete, the container is stopped and removed.

C. GROQ Cloud

GROQ Cloud is a managed inference platform provided by Groq Inc., designed to host and serve large language models with low latency and high throughput. [10] It offers a simple REST API for model deployment, automatic scaling, and monitoring of usage. For small-scale or experimental projects, GROQ Cloud provides a free tier up to a certain monthly token limit. Interested reader can find full details and pricing on the references.

D. Benchmark: MBPP Dataset

Austin et al introduced The MBPP (Mostly Basic Python Problems) dataset in [11], and the dataset is officially available on [12]. The dataset is a collection of 974 small Python programming tasks, each accompanied by a natural language problem statement, a reference implementation, and a set of built in unit tests. Tasks are indexed 1–974, where

- 1–10 were originally reserved for few-shot prompting,
- 11–510 form the official test set,
- 511–600 were used for validation, and
- 601–974 for training.

In the evaluation of this study, I treat all 974 examples uniformly as "test" problems and employ a sampling strategy. I sample roughly 5% of all tasks. More information will be given in Evaluation and Limitations section of this paper.

III. Methodology

In this section, I describe the from scratch implementation of an isolated, sequential chat-like multi agent pipeline for safe code generation and evaluation. First, the individual agents will be introduced, then details will be given about the overall workflow, followed by our Docker-based sandbox setup and a discussion on the importance of isolation.

A. Multiagent System

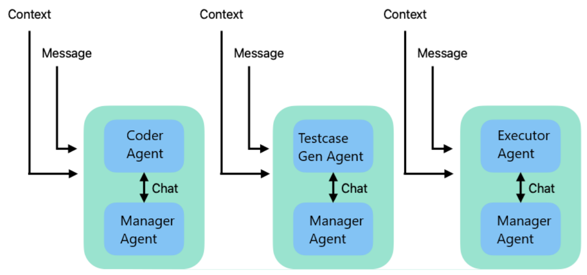

Fig. 3. The Mate-Incoder conversation pattern (Isolated Sequential Chat)

The proposed system consists of four specialized agents built on the Autogen framework. Each agent runs with its own role-specific prompt (listed in Appendix) and communicates only with the Manager Agent to avoid unwanted information sharing. Fig.3 is an adaptation of fig.2 where carryovers are removed, and the proposed agent architecture is shown.

Coder Agent

The Coder Agent receives a function description from the Manager Agent. It generates a Python function implementation intended to satisfy the given description. If the code is syntactically invalid or does not meet the tests, it will be refined in later iterations.

Testcase Generator Agent

This agent takes the problem description and the sample testcases as input and produces diverse test cases (basic, edge, and large-scale). It returns a list of input/output pairs in JSON format.

Executor Agent

The Executor Agent is responsible for running the generated code against the test cases. It uses Autogen's tool registration to invoke a function called `execute_code_batch`, which executes each test inside the Docker sandbox and returns the results (stdout, return value, errors).

Manager Agent

The Manager Agent orchestrates the entire loop. It first triggers test generation and code generation, then collects execution reports from the Executor Agent. If any tests fail, it sends structured feedback back to the Coder and Testcase Generator Agents for correction, up to a maximum number of iterations. The important thing in here, if any tests fail, the manager angent sends the error message or unexpected output case to both coder and testcase generator agents and ask them to review their work. With this way, even if the coder agent sees the testcases after the execution, it is not allowed to fix them because testcases will be changed in the next execution.

B. Workflow

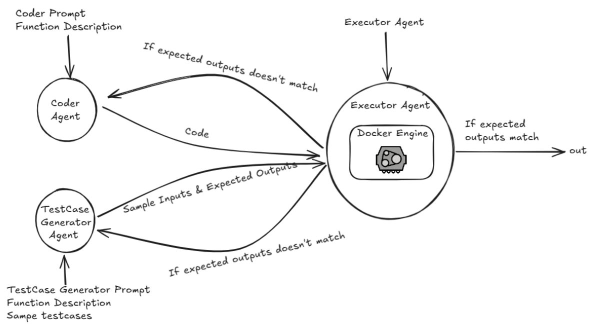

Fig. 4. Manager agent workflow & Loop

The core logic lives in the `run_workflow` method of the Manager Agent. Fig.3 shows our isolated sequential-chat structure, where only one agent at a time receives the conversation context. Fig.4 sketches the full loop:

1) Start and initialize the Docker sandbox.
2) Manager asks Testcase Generator for initial test cases.
3) Manager asks Coder Agent for initial code draft.
4) Executor Agent runs tests in the sandbox and returns results.
5) If all tests pass, return final code; otherwise, feedback is sent to Coder and Testcase Generator to ask for both to update their work, workspace is cleaned, and loop repeats.

This isolated sequential pattern prevents any agent from seeing another agent's internal messages directly.

C. Docker Integration & Safety

I use a Docker container as a sandbox. The docker container is built and started at the starting of every code test execution. Which means, every testcase doesn't built a new docker container, a container is started at the start of the program and exists until the program returns the output. The container is launched with resource limits (512MB RAM, 0.5 CPUs) and no network. A host directory is bind-mounted to `/app/sandbox`. At each iteration, the Manager Agent runs a cleanup script inside the container to remove any leftover files. After the workflow completes or reaches its iteration cap, the container is stopped and removed to leave no trace on the host.

D. Importance of Isolation

Isolation is crucial to ensure that the Coder Agent cannot peek at test cases and hardcode solutions, and that test generation remains unbiased by previous code drafts. In machine learning, we separate training and test sets to avoid data leakage; similarly, the idea behind my design has some conversational and execution boundaries, like coder agent and testcase agent doesn't even know the other one is exist. This prevents overfitting to specific tests and improves the generality of generated code. This issue will be the base study of this paper, explained in the following section.

IV. Related Work

Modern code generation frameworks increasingly leverage multiple specialized agents to break down and refine tasks. Here, I compare Mate-Incoder pipeline to three representative approaches, highlighting differences in feedback loops, safety, and isolation.

A. ChatDev

ChatDev is introduced in 2024 by Qian et al. [13]. It is a chat powered software development framework that employs different LLM driven agents with distinct roles, such as requirements analyst, programmer, tester. Instead of having all logic into a monolithic prompt, ChatDev breaks every phase into smaller subtasks for different agents and runs a structured chat chain to create and ensure the solutions. To cope with LLM hallucinations, such as incomplete or unexecutable code, ChatDev introduces a communicative dehallucination mechanism, where the assistant first asks for precise details before finalizing its response, significantly improving code correctness and reducing manual debugging. Empirical results on a suite of programming tasks show that ChatDev outperforms strong single agent baselines (e.g., GPT-Engineer) in metrics like completeness, executability, and overall quality.

B. AutoSafeCoder

AutoSafeCoder is introduced by Nunzer et al. in 2024 [14], and it's more about safety, which is also an issue for Mate-Incoder. In this work, the authors extend the multi-agent paradigm to security by adding a Static Analyzer Agent and a Fuzzing Agent alongside a Coding Agent. After initial code generation, the Static Analyzer finds potential vulnerabilities, and the Fuzzing Agent dynamically tests the code under mutation-based scenarios, feeding security-focused feedback to improve robustness. Unlike my approach, AutoSafeCoder runs all agents within the same runtime environment rather than an isolated sandbox.

C. AgentCoder

AgentCoder is introduced in 2024 by Huang et al. [15] AgentCoder introduces a system that has 3 agents: pipeline programmer, test designer, and test executor to iteratively refine generated code via feedback from automatically produced test cases. The Test Designer

Agent creates targeted test inputs, while the Test Executor Agent runs the code and reports failures back to the programmer agent for correction. This loop improves code quality but does not enforce strict execution isolation, relying instead on in-process checks. Despite the surface similarity to Mate-Incoder pipeline, there are key differences:

- Testcase Reliability Assumption: AgentCoder's test designer agent creates the tests only once and doesn't get feedback, which means testcases are fixed across all iterations, even if the code is updated in every iteration. On the other hand, in Mate-Incoder, it assumes the Testcase Generator can also make mistakes. So, when a failure occurs, feedback is arrived to both the Coder and the Testcase Generator for correction.
- No Sandboxing: AgentCoder does not mention or enforce any sandbox or container boundary; code and tests run directly on the host. On the other hand, as explained in Methodology section, Mate-Incoder employs a docker sandbox to ensure safety.
- Test Visibility (Isolation): Because AgentCoder's Programmer agent sees the same static tests every time, it can easily hard-code fixes for known failures. Mate-Incoder sandboxed, isolated sequential chat prevents such leakage no agent can peek at another's internal data, ensuring truly generalizable solutions.
- Optimisation Techniques: AgentCoder uses 16 different Optimisation techniques to improve accuracy. Using Chain-of-Thought, ReAct, CodeCoT, MetaGPT etc. approaches, it tries (and actually manages) to improve raw LLMs' coding accuracy.

V. Results

This section presents two studies that illustrate how dynamic test regeneration impacts code generality (Study 1) and how Mate-Incoder workflow performs in terms of end-to-end accuracy on sampled MBPP tasks (Study 2).

A. Study 1: Importance of Testcase Regeneration

In this experiment, I compare two versions of Mate-Incoder on a single MBPP problem to see how fixed vs. regenerated tests affect the final code. The sample problem description from MBPP dataset is:

> "Write a function to count the number of ways to fill a 3×n board with 2×1 dominoes."

Mate-Incoder Version A: Fixed Testcases

Here, Mate-Incoder generates the testcases only once at the start and keep them unchanged through all iterations. After two rounds of feedback, the Coder Agent learns to hard-code a lookup table that satisfies every fixed test:

```python
def sample_method(n):
    if n == 1:
        return 0
    elif n == 2:
        return 1
    elif n == 3:
        return 0
    elif n == 4:
        return 11
    elif n == 5:
        return 0
    elif n == 6:
        return 41
    elif n == 10:
        return 571
```

For this description, version A passes all fixed testcases, but this solution clearly does not generalize beyond the fixed cases. It simply memorizes the answers. And this code is given to user as an execution validated final code.

Version B: Regenerated Testcases

In contrast, Version B asks the Testcase Generator Agent to produce new cases at each iteration. Because the tests keep changing, the Coder Agent cannot rely on a static lookup and must discover a genuine algorithm. Or, even if it tries to generate a hardcoded solution for the testcases, at least it cannot pass all the testcases in the next iteration because testcases are already updated. As a result:

- The nominal "accuracy" on any finite sample of regenerated tests could be lower.
- but Mate-Incoder Version B prevents the delivery of hardcoded solutions even if it still cannot give the perfect code.

B. Study 2: Accuracy

In this experiment, Mate-Incoder's overall accuracy is evaluated on randomly chosen MBPP problems using GPT 3.5-turbo. Actually Mate-Incoder is built with Groq cloud's LLMs, but GPT 3.5-turbo is selected for evaluation because AgentCodersubsection IV-C is also used that model in its evaluation. On the other hand, due to token limits of Groq cloud, and the messy assert formats in MBPP benchmark, 50 examples at random are sampled (for some testcases, parsing of assert sentences were hard, or failed. For these testcases, other random testcases are chosen.). Each example was fed through the full run_workflow loop, and it's recorded whether the first draft passed all tests (Pass@1) or succeeded after feedback (Overall@N). The results are:

- **Pass@1:** 24 out of 50 examples (48 %) passed without any feedback iterations.
- **Overall Success:** After up to 5 feedback rounds, 33 examples (66 %) eventually passed all generated tests.

One of the fail testcases in this setting was the domino-tiling sample from Study 1 (subsection V-A).

Also, to see the potential of Mate-Incoder with higher capacity LLMs, a very small follow up test is applied with following setup:

```
coder=
"meta-llama/llama-4-maverick-17b-128e-instruct"
tc="meta-llama/llama-4-scout-17b-16e-instruct"
executor = "llama-3.1-8b-instant"
manager="mistral-saba-24b"
```

In that mini-study, 3 of 26 failures are tested. Two passed at first attempt and one passed after feedback loop, suggesting that stronger LLMs can improve both Pass@1 and overall rates, even though it's not possible to test every sample with Groq Cloud's LLMs due to cost and API constraints.

Discussion of Study 2

Comparing these figures directly to AgentCoder's reported 72% Pass@1 and 89.9% overall would be unfair: they use a suite of 16 internal optimizations (CoT, re-ranking, etc.) while the study 2 of Mate-Incoder relied on a vanilla GPT-3.5 turbo. One other topic is that in particular, it's observed that the executor agent that set to GPT-3.5 turbo often fails to give the termination signal ("quit chat") correctly when tests pass, or it doesn't give termination signal even if all tests passes. This issue leads to remain/terminate the feedback loop unnecessarily. This executor termination issue is one of the key limitations of the current study 2 setup and will be a focus of future improvements.

VI. Limitations & Future Work

Despite its promising results, Mate-Incoder has several important limitations:

- The Executor Agent when driven by GPT-3.5-turbo sometimes sends its "quit chat" signal at the wrong time. It may stop the loop even though not all tests have truly passed, or it may keep running past a successful run. This irregular behavior can hide the system's real performance and complicates debugging.
- Most experiments ran on the GPT 3.5 turbo, and some of them on free tier of GROQ Cloud. We saw clear gains in a small follow up trial explained in study 2, with larger models, but I couldn't scale them fully to this study due to API rate limits and cost. A more comprehensive evaluation on bigger models remains future work.
- Due to limited time, Mate-Incoder approach can't be a opponent for AgentCoder work. Its optimisation techniques remains as future work of this study.
- The author of this study wasted extremely much time on safety issue. A lot of different sandboxing techniques were tried in WSL (Windows Subsystem for Linux) environment, deeply. However, I was only able to use Docker with Windows machine and wasn't able to use more lightweight sandboxing approaches that were explained in background section.

Agentic AI Integration & Future Directions

A perfect next step could be layering in agentic AI behaviors on top of Mate-Incoder's fixed-loop design. Which means, instead of a hard coded workflow (run_workflow method), a meta agent could learn over time which test generation strategies and iteration counts work best (see below paragraph), dynamically adapting its policy based on past successes and failures. Finally, about cost, an accountant agent could be employed to monitor resource usage or the current testcase's difficulty. It may decide when to switch to stronger LLMs for difficult problems, or create some debug agents to tackle runtime errors.

VII. Discussion & Conclusions

In this work, Mate-Incoder is introduced, a powerful multi-agent pipeline that keeps generated code, test cases, and execution strictly isolated. The two studies on MBPP tasks showed that:

- Dynamically regenerating tests stops the Coder Agent from resorting to lookup tables and forces it to learn real algorithms. Or, at least it doesn't give wrong outputs.
- On a random sample of 50 MBPP problems, the GPT-3.5-based pipeline achieved 48 % Pass@1 and 66 % overall success after up to five feedback rounds. A small pilot with larger Groq Cloud models gave better performance.

These results suggest that isolation and the loops having test regeneration can dramatically improve the reliability of LLM-generated code, at the cost of some measured "accuracy" on any fixed test set. In practice, this trade-off is worthwhile: it prevents the delivery of hardcoded solutions and gives users greater confidence.

Mate-Incoder is just a first step. The current setup relies on GPT-3.5's sometimes-faulty termination signals. I also ran experiments on a small fraction of MBPP due to cost and API limits. Despite these drawbacks, the core idea combine isolated execution with dynamic feedback seems applicable.

Looking forward, I plan to explore Agentic AI enhancements: a meta-agent that learns optimal loop policies, self reflective agents that summarize their own mistakes, and specialized debug or cost monitoring agents. Also, the optimisations of AgentCoder and more of them should be added for better accuracies.

References

[1] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W. tau Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," 2023. [Online]. Available: https://arxiv.org/abs/2204.05999

[2] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang, "Autogen: Enabling next-gen llm applications via multi-agent conversation," 2023. [Online]. Available: https://arxiv.org/abs/2308.08155

[3] Microsoft, "Autogen: Conversation patterns," 2024, accessed: 2025-06-09. [Online]. Available: https://microsoft.github.io/autogen/0.2/docs/tutorial/conversation-patterns/#nested-chats

[4] ——, "Autogen: Tool use tutorial," 2024, accessed: 2025-06-09. [Online]. Available: https://microsoft.github.io/autogen/0.2/docs/tutorial/tool-use/

[5] Firejail Project, "Firejail: Linux sandbox program," 2024, accessed: 2025-06-09. [Online]. Available: https://firejail.wordpress.com/

[6] A. L. Wiki, "Bubblewrap," 2024, accessed: 2025-06-09. [Online]. Available: https://wiki.archlinux.org/title/Bubblewrap

[7] Google LLC, "gvisor: Container sandbox," 2024, accessed: 2025-06-09. [Online]. Available: https://gvisor.dev/

[8] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/agache

[9] Docker, Inc., "Docker: Empowering app development for developers," 2024, accessed: 2025-06-09. [Online]. Available: https://www.docker.com/

[10] Groq, Inc., "Groq console," 2024, accessed: 2025-06-09. [Online]. Available: https://console.groq.com

[11] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, and Q. Le, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[12] Google Research, "Mbpp dataset github repository," 2024, accessed: 2025-06-09. [Online]. Available: https://github.com/google-research/google-research/tree/master/mbpp

[13] C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong, J. Xu, D. Li, Z. Liu, and M. Sun, "Chatdev: Communicative agents for software development," 2024. [Online]. Available: https://arxiv.org/abs/2307.07924

[14] A. Nunez, N. T. Islam, S. K. Jha, and P. Najafirad, "Autosafecoder: A multi-agent framework for securing llm code generation through static analysis and fuzz testing," 2024. [Online]. Available: https://arxiv.org/abs/2409.10737

[15] D. Huang, J. M. Zhang, M. Luck, Q. Bu, Y. Qing, and H. Cui, "Agentcoder: Multi-agent-based code generation with iterative testing and optimisation," 2024. [Online]. Available: https://arxiv.org/abs/2312.13010

VIII. Appendix

A. Coder Agent System Message

```
system_message = """You are an AI code generator
    that creates Python functions based on
    descriptions and test cases."""
```

B. Test Case Generator Agent System Message

```
system_message="""
You are an AI test designer that generates
    Python test cases without seeing the
    implementation.
Your job is to create diverse, valid, and
    well-structured test cases based only on the
    task description.
You should include basic, edge, and large-scale
    test scenarios.
```

```
For each test, provide an input and its expected
    output. Ensure outputs are logically
    consistent with the task.
Do not include Python code, only structured test
    descriptions.
"""
```

C. Executor Agent System Message

```
system_message="""You are an AI code evaluator.
    You have only one duty. You need to evaluate
    the given code and prepare a evaluation
    report. You must not do debugging. You must
    not try to solve problems."""
```

D. Manager Agent System Message

```
system_message = """You are the Manager Agent
    coordinating CoderAgent,
    TestcaseGeneratorAgent, and ExecutorAgent."""
```

E. Testcase Generator Agent Prompt

```
        prompt = f"""
TASK DESCRIPTION:
{description}

THERE is one example input and output for this
    problem. Don't copy them please, these are
    just for get sense of:
Sample input: {sample_input}
Sample output: {sample_output}

Add at least one:
- Basic functionality test
- Edge case
- Large-scale test

OUTPUT FORMAT:
Return the test cases as a valid JSON array of
    dictionaries, using the following format:

[
    {{
        "input": <JSON-serializable input>,
        "expected_output": <JSON-serializable
            output>
    }},
    ...
]

Rules:
- "input" can be any JSON-valid type: number,
    string, list, object, tuple-as-list, etc.
- If the input consists of multiple values, use
    a list.
- Use lowercase booleans (true/false) in JSON.
- Do NOT include any explanations or markdown.
    Just raw JSON.
IMPORTANT RULE: Your entire response **MUST** be
    exactly one valid JSON array and nothing
    else.
- Do NOT include any text before or after the
    array.
- Do NOT use markdown code fences.
- Do NOT add punctuation, commentary, or stray
    commas.
"""
```

F. Coder Agent Prompt

```
        prompt = f"""
You are a senior Python developer.

CODING TASK IN PYTHON:
Write an optimized and well-documented Python
    function based on the following details:
Function name:
{func_name}
Detailed function description:
{description}
Ensure that the function satisfies the following
    test cases:
{test_cases}

The function should:
- Follow best coding practices.
- Include proper docstrings and type hints.
- Be efficient and optimized.
- Avoid unnecessary complexity.
- Ensure all given inputs return the expected
    outputs.
- Be careful about indentation  make sure all
    blocks are properly indented.

REQUIREMENTS:
- Provide a Python implementation.
- Import necessary libraries.
- In your answer, give only the function
    implementation, without extra characters or
    explanations. Don't add any character other
    than the function in your answer.
"""
```

G. Coder Feedback Prompt

```
        coder_feedback =  f"""
Based on latest execution report:
    {executor_response}
Could you please review your code? There may be
    an error in your implementation.
Double-check your code, then please give your
    revised code or state "I'm sure the
    implementation is perfect"

OUTPUT FORMAT:
If you're sure your implementation is perfect,
    return only the following sentence: "I'm
    sure the implementation is perfect"
Else: Give only the corrected function
    implementation.
***Do NOT include any explanations or markdown.
    Your answer must contain just function
    implementation without extra characters or
    explanations OR the sentence 'I'm sure the
    implementation is perfect'***
"""
```

H. Test Case Generator Feedback Prompt

```
        testcase_feedback = f"""
The execution report indicates the following
    issues: {executor_response}
```

```
Could you please examine the test cases you
    gave? There might be errors in your test
    design
Double-check them, then correct the problem if
    the issue is about the testcases.
If not, then please add only one more testcase
    to your existing testcases.

OUTPUT FORMAT:
***If you're sure the testcases are true:***
    please add only one more testcase to your
    existing testcases, and return test cases as
    a valid JSON array of dictionaries, using
    the following format:
[
    {{
        "input": <JSON-serializable input>,
        "expected_output": <JSON-serializable
            output>
    }},
    ...
]

***Else:***

Return the test cases as a valid JSON array of
    dictionaries, using the following format:

[
    {{
        "input": <JSON-serializable input>,
        "expected_output": <JSON-serializable
            output>
    }},
    ...
]

***Do NOT include any explanations or markdown.
    Your answer must contain just raw JSON.***
"""
```