SOEN-101: Code Generation by Emulating Software Process Models Using Large Language Model Agents

Feng Lin[1], Dong Jae Kim[2], Tse-Hsun (Peter) Chen[1]
[1]*Software PErformance, Analysis, and Reliability (SPEAR) lab, Concordia University, Montreal, Canada*
[2]*DePaul University, Chicago, USA*
feng.lin@mail.concordia.ca, k_dongja@encs.concordia.ca, peterc@encs.concordia.ca

***Abstract*—Software process models are essential to facilitate collaboration and communication among software teams to solve complex development tasks. Inspired by these software engineering practices, we present *FlowGen* – a code generation framework that emulates software process models based on multiple Large Language Model (LLM) agents. We emulate three process models, $FlowGen_{Waterfall}$, $FlowGen_{TDD}$, and $FlowGen_{Scrum}$, by assigning LLM agents to embody roles (i.e., requirement engineer, architect, developer, tester, and scrum master) that correspond to everyday development activities and organize their communication patterns. The agents work collaboratively using chain-of-thought and prompt composition with continuous self-refinement to improve the code quality. We use GPT3.5 as our underlying LLM and several baselines (*RawGPT*, *CodeT*, *Reflexion*) to evaluate code generation on four benchmarks: HumanEval, HumanEval-ET, MBPP, and MBPP-ET. Our findings show that $FlowGen_{Scrum}$ excels compared to other process models, achieving a Pass@1 of 75.2, 65.5, 82.5, and 56.7 in HumanEval, HumanEval-ET, MBPP, and MBPP-ET, respectively (an average of 15% improvement over *RawGPT*). Compared with other state-of-the-art techniques, $FlowGen_{Scrum}$ achieves a higher Pass@1 in MBPP compared to *CodeT*, with both outperforming *Reflexion*. Notably, integrating *CodeT* into $FlowGen_{Scrum}$ resulted in statistically significant improvements, achieving the highest Pass@1 scores. Our analysis also reveals that the development activities impacted code smell and exception handling differently, with design and code review adding more exception handling and reducing code smells. Finally, *FlowGen* models maintain stable Pass@1 scores across GPT3.5 versions and temperature values, highlighting the effectiveness of software process models in enhancing the quality and stability of LLM-generated code.**

***Index Terms*—Large Language Model, Code Generation, Agents, Software Process Model**

I. Introduction

The recent surge of Large Language Models (LLMs) has sparked a transformative phase in programming and software engineering. With tools like ChatGPT [28] or LLaMA [39], researchers have demonstrated the potential of LLMs in generating commit messages [51], resolving merge conflicts [35], generating tests [43, 50, 33], method renaming [1], and even facilitating log analytics [23, 24].

Among all development activities, code generation has received much attention due to its potential to reduce development costs. As LLMs are becoming increasingly integral to software development, various techniques have emerged in LLM-based code generation. For example, prompting techniques like few-shot learning [18, 49] have been shown to improve code generation results. In particular, few-shot learning coupled with few-shot sampling [24, 17] or information retrieval augmented technique [27, 6] have been shown to improve code generation. Moreover, one can integrate personalization in the prompt, instructing LLMs to be domain experts in a specific field, which can further improve LLM responses [41, 34]. Such personalization techniques highlight the potential of using multiple LLMs working together to assist in complex software development activities.

Given the complexity of software development, LLM agents stand out among various LLM techniques. Agents are LLM instances that can be customized to carry out specific tasks that replicate human workflow [14, 11]. Recently, multi-agent systems have achieved significant progress in solving complex problems in software development by emulating development roles [14, 11, 31]. MetaGPT, introduced by Hong et al. [14], integrated development workflow using standard operating procedures by assigning specific roles (e.g., a designer or a developer) to LLM agents. Dong et al. [11] developed self-collaboration, which assigns LLM agents to work as distinct "*experts*" for sub-tasks in software development. Qian et al. [31] proposed an end-to-end framework for software development through self-communication among the agents.

Despite the promising applications of LLMs in automating software engineering tasks, it is pivotal to recognize that software development is a collaborative and multi-faceted endeavor. In practice, developers and stakeholders work together, following certain software process models like *Waterfall*, *Test-Driven-Development (TDD)*, and *Scrum*. Even though there is a common community agreement on the pros and cons of each process model [13], the impact of adopting these process models for LLM code generation tasks remains unknown. In particular, will emulating different process models impact the generated code quality in different aspects, such as reliability, code smell, and functional correctness?

While some research has explored integrating multi-agents within LLM frameworks [31, 45, 47], their research focus diverges from the influence of the software process model on code generations for several reasons: 1) Xu et al. [45] do not adhere to specific process models, and 2) both Dong et al. [11] and Qian et al. [31] focus solely on *Waterfall*-

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00140

like models, neglecting *TDD* and *Scrum*, which may have different impact on code generations. Importantly, none of the aforementioned studies conduct a fine-grain analysis of how different development activities affect code quality metrics, such as code smell and reliability, other than the Pass@1 score. Our study takes further steps to analyze the impacts of different agents within the process models on code generation and their influence on other code quality attributes.

This paper presents a novel multi-agent LLM-based code generation framework named *FlowGen*. *FlowGen* integrates diverse prompt engineering techniques, including chain-of-thought [40, 20], prompt composition [22, 50], and self-refinement [25], with a focus on emulating the flow of development activities in various software process models. Specifically, we implemented three popular process models into *FlowGen*: $FlowGen_{Waterfall}$, $FlowGen_{TDD}$, and $FlowGen_{Scrum}$. Each process model emulates a real-world development team involving several LLM agents whose roles (i.e., requirement engineer, architect, developer, tester, and scrum master) correspond to common software development activities. The agents work collaboratively to produce software artifacts and help other agents review and improve the artifacts in every activity.

We evaluate *FlowGen* on four popular code generation benchmarks: *HumanEval* [8], *HumanEval-ET* [10], *MBPP* [3], and *MBPP-ET* [21]. We apply zero-shot learning to avoid biases in selecting few-shot samples [44]. To compare, we also apply zero-shot learning on GPT-3.5 as our baseline (*RawGPT*). We repeat our experiments five times to account for variability in LLM's responses and report the average value and standard deviation. To study code quality, in addition to Pass@1, we run static code checkers to detect the prevalence of code smells in the generated code. Our evaluation shows that $FlowGen_{Scrum}$'s generated code achieves the highest accuracy (Pass@1 is 75.2 for HumanEval, 65.5 for HumanEval-ET, 82.5 for MBPP, and 56.7 for MBPP-ET), surpassing *RawGPT*'s Pass@1 by 5.2% to 31.5%. While *FlowGen*, in general, is more stable than *RawGPT*, $FlowGen_{Scrum}$ exhibits the most stable results with an average standard deviation of only 1.3% across all benchmarks.

Additionally, we compare $FlowGen_{Scrum}$ with state-of-the-art techniques: *CodeT* [5] and *Reflexion* [36]. Both $FlowGen_{Scrum}$ and *CodeT* outperform *Reflexion* significantly in terms of Pass@1 across all benchmarks, with $FlowGen_{Scrum}$ achieving a higher Pass@1 than *CodeT* in MBPP. Furthermore, the integration of *CodeT* into $FlowGen_{Scrum}$ demonstrates the highest Pass@1, highlighting the potential of integrating other techniques with *FlowGen* for improved code generation.

We further study the impact of each development activity on code quality. We find that removing the testing activity in the process model results in a significant decrease in Pass@1 accuracy (17.0% to 56.1%). Eliminating the testing activity also leads to a substantial increase in error and warning code smell densities. We also find that the design and code review activities reduce refactor and warning code smells, and improve reliability by adding more exception handling code. Nevertheless, *FlowGen* consistently outperforms *RawGPT* by reducing code smells and enhancing exception handling. Finally, we find that the GPT model version plays a significant role in the quality of generated code, and *FlowGen* helps ensure stability across different versions of LLMs and temperature values.

We summarize the main contributions as follows:

1) **Originality**: We introduce a multi-agent framework called *FlowGen*, incorporating software process models from real-world development practice. We integrate agents acting as requirement engineers, architects, developers, testers, and scrum masters, and study how their interaction improves code generation and code quality.
2) **Technique**: We integrate prompt engineering techniques like chain-of-thought, prompt composition, and self-refinement to facilitate interactions among the agents. We implement three recognized process models: $FlowGen_{Waterfall}$, $FlowGen_{TDD}$, and $FlowGen_{Scrum}$, but the technique can be easily extended to emulate other process models or development practices (e.g., DevOps).
3) **Evaluation**: We conduct a fine-grained evaluation on the quality of the generated code using four popular code generation benchmarks: *Humaneval* [8], *Humaneval-ET* [10], *MBPP* [3], *MBPP-ET* [21], comparing agent interactions and their effect on both accuracy (Pass@1) and other code quality metrics (e.g., smells). We manually checked the generated code and discussed the reasons for test failures. Finally, we examined how model versions and temperature settings affect code generation stability.
4) **Data Availability**: To encourage future research in this area and facilitate replication, We made our data and code publicly available online [2].

Paper Organization. Section II discusses background and related work. Section III provides the details of *FlowGen*. Section IV evaluates our *FlowGen*. Section V provides a discussion on future work. Section VI discusses threats to validity. Section VII concludes the paper.

II. Background & Related Works

In this section, we discuss the background of software process models and LLM agents. We also discuss related work on LLM-based code-generation.

A. Background

Software Development Process. Software development processes encompass methodologies and practices that development teams use to plan, design, implement, test, and maintain software. The primary goal of a software process is to assist the development teams in producing high-quality software. Generally, different software process models involve the same set of development activities, such as requirement, design, implementation, and testing, but differ in how the activities are organized. Because of the variation, each software process model has its strengths and weaknesses based on the project type, teams, and experience [13].

In particular, three well-known and widely adopted software process models were created over the years: *Waterfall*,

Test-Driven-Development (TDD), and *Scrum*. *Waterfall* [4] is often used in safety-critical systems where development teams must adhere to a linear path, and each software development activity builds upon the previous one. *TDD* and *Scrum* are both variants of the agile development model. Compared to *Waterfall*, agile process models focus more on iterative and incremental development and adapting to change. *TDD* [26] emphasize writing tests before writing the actual code to improve software design and quality. *Scrum* highlights the importance of collaboration and communication in software development. *Scrum* prescribes for teams to break work into time-boxed iterations called sprints. During these sprints, teams focus on achieving specific goals (e.g., user stories), ensuring a continuous discussion among teams to handle any unexpected risks throughout the development process.

LLM Agents. LLM agents are artificial intelligence systems that utilize LLM as their core computational engines to understand questions and generate human-like responses. LLM agents can refine their responses based on feedback, learn from new information, and even interact with other AI agents to collaboratively solve complex tasks [14, 31, 45, 29]. Through prompting, agents can be assigned different roles (e.g., a developer or a tester) and provide more domain-specific responses that can help improve the answer [14, 41, 34].

One vital advantage of agents is that they can be implemented to interact with external tools. When an agent is reasoning the steps to answer a question, it can match the question/response with corresponding external tools or APIs to construct or refine the response. For instance, an LLM agent that represents a data analysis engineer can apply logical reasoning to generate corresponding SQL query statements, invoke the database API to get the necessary data, and then answer questions based on the returned result. When multiple LLM agents are involved, they can collaborate and communicate with each other. Such communication is essential for coordinating tasks, sharing insights, and making collective decisions. Hence, defining how the agents communicate can help optimize the system's overall performance [42], allowing agents to undertake complex projects by dividing tasks according to their domain-specific skills or knowledge.

The software development process plays a crucial role in software development, fundamentally involving communication among various development roles. Given the demonstrated capability of Large Language Model (LLM) agents to mimic domain experts in specific fields [41, 14, 31], this study leverages LLM agents to represent diverse development roles and conduct their associated duties. Our research establishes a collaborative team of LLM agents designed to emulate these process models and roles, aiming to enhance code generation.

B. Related Works

Code generation is a thriving field of research because of its potential to reduce development costs. In particular, *prompt-based* and *agent-based* code generation techniques are two of the most prevalent directions.

Prompt-based Code Generation. Prompt-based code generation employs a range of techniques to refine prompts, ultimately leading to the generation of expected code. For example, Li et al. [20] propose using structured prompts containing code information (e.g., branch and loop structures) to improve the generated code. Nashid et al. [27] retrieval code demos similar to the given task and include them in the prompt to improve code generation. Ruiz et al. [32] use translation techniques for program repair, where buggy programs are first translated into natural language or other programming languages. The translated code is used as a prompt to generate new/fixed code with the same feature. Schäfer et al. [33] iteratively refine prompts based on feedback received from interpreters or test execution results. Kang et al. [17] provide specific instructions, test method signature, and bug report as part of the prompt for generating test code to reproduce bugs. Xie et al. [43] parse the code to identify the focal method and related code context, which are given in the prompt for test code generation. Yuan et al. [50] apply a prompt composition technique by first asking an LLM to provide a high-level description of a method, and then the description is used as part of the prompt to enhance test code generation. Chen et al. [5] introduced *CodeT*, a framework that employs self-generated tests to evaluate the quality of generated code. Shinn et al. [36] presented *Reflexion*, which utilizes an evaluator LLM to provide feedback for enhancing future decision-making processes.

Agent-based Code Generation. Agent-based code generation emphasizes on the importance of role definition and communication among multiple LLM agents. Some approaches incorporate external tools as agents. For example, Huang et al. [16] introduce the test executor agent, employing a Python interpreter to provide test logs for LLMs. Zhong et al. [52] introduces a debugger agent that utilizes a static analysis tool to build control flow graph information, guiding LLMs in locating bugs. Meanwhile, other studies [14, 31, 11] task LLMs as agents by emulating diverse human roles, including analysts, engineers, testers, project managers, chief technology officers (CTOs), etc. Nevertheless, these studies miss key roles in the development activities (e.g., only has analysts, coders, and testers [11]) or focus more on the business side of the roles (e.g., employ CTO and CEO) [31]. In our work, we try to follow the Waterfall model that is proposed in the software engineering literature and create agents that correspond to every development activity. These approaches follow the *Waterfall* model to communicate among these roles, with variation in the prompts and roles, ultimately improving code generation.

In comparison, our research leverages LLM agents to emulate multiple software development process models, while prior research focuses only on the *Waterfall* model [14, 31, 11]. We implement several prompting techniques, but more importantly, we emphasize on how various process models and the associated development activities affect the generated code. Different from prior works which only study functional correctness, we study several additional dimensions of code quality, including code design, code smell, convention issues,

and reliability. We also explore why the generated code fails the tests and the sensitivity of the results across LLM model versions and temperature values.

III. Methodology

We propose *FlowGen*, an agent-based code generation technique based on emulating different software processes. Figure 1 shows the overview of *FlowGen*: (1) define the roles and their responsibilities; (2) use LLM agents to represent these roles; and (3) complete the interactions among these agents according to the software process models. In each development activity, we implement chain-of-thought and self-refinement to improve the quality of the generated artifacts. In particular, we study and compare three software process models: *Waterfall*, *TDD*, and *Scrum*. Nevertheless, *FlowGen* can be easily adapted to different process models. We use zero-shot learning in all our experiments to avoid biases in selecting data samples. Below, we discuss *FlowGen* in detail.

A. Using LLM Agents to Represent Different Development Roles in Software Process Models

In *FlowGen*, we create LLM agents who are responsible for the main development activities: requirement, design, implementation, and testing. Hence, to emulate a software process model, we reorganize the communication and interaction among different agents. The benefit of such a design is that it maximizes the extensibility and reusability of the agents, and *FlowGen* can be easily adapted to different process models. We implement four agents whose role corresponds to the common development activities: *Requirement Engineer*, *Architect*, *Developer*, and *Tester*. For *Scrum*, we introduce an additional role – *Scrum Master*.

We designed these roles to use the same prompt template across different process models (with different terms such as user stories v.s. requirement) to investigate the effectiveness of process models on code generation. The exact words that we used for the prompts can be found online [2]. The role-specific details of our prompt are:

```
{
  "Role": "You are a [role] responsible for [task]",
  "Instruction": "According to the Context please [
    role-specific instruction]",
  "Context": "[context]"
}
```

In this prompt template (inspired by MetaGPT [14] and Self-Collaboration [11]), `role` refers to one of the roles (e.g., Requirement Engineer) that corresponds to the development activity, and `task` describes the duties for the `role` (e.g., analyze and create requirement documents). `Instruction` leverages chain-of-thought reasoning [40, 20] and refers to role-specific instruction listed in steps, such as 1) analyzing the requirement and 2) writing the requirement documentation. Finally, `Context` contains the programming question, the agent conversation history, or the agent-generated artifacts. `Context` includes all necessary information that helps the agents to make a next-step decision based on the current conversation and generated results.

Table I shows the `tasks`, `instructions`, and `contexts` for every development role. In general, every role takes the output from the prior development activities as input (i.e., `context`). For example, an architect writes a design document based on the requirement document generated by a requirement engineer. We design developers and testers to have multiple tasks. Developers are responsible for writing code and fixing/improving the code based on suggestions. We design testers using a prompt composition technique, which is shown to improve the LLM-generated result [22, 50]. First, testers design a test. Then, testers write and execute the tests based on the design. On average, *FlowGen* generates four tests for each problem before the review meeting and six after the meeting. It is important to note that oracles are kept aside and never used in the code-generation process. Finally, testers generate a test failure report.

Developers receive the test failure report to fix the code. In addition to the tasks described in Table I, all roles have one common task, which is to provide feedback to other roles for further improvement (e.g., for code review).

B. Communications Among Agents

One of the most important aspects of LLM agents is how the agents communicate. A recent survey paper [42] shows that one common communication pattern is sequential, i.e., ordered, where one agent communicates to the next in a fixed order. Another pattern is disordered, where multiple agents participate in the conversation. Each agent gets the context separately and outputs the response in a shared buffer. Then, the responses can be summarized and used in the next decision-making process. Based on the software process models and the two aforementioned communication patterns, we implement three interaction models for the agents: $FlowGen_{Waterfall}$, $FlowGen_{TDD}$, and $FlowGen_{Scrum}$ (Figure 1). The details of our multi-agent communication history are available online [2].

***FlowGen*$_{Waterfall}$** $FlowGen_{Waterfall}$ follows the *Waterfall* model and implements an ordered communication among the agents. Given a programming problem, the problem goes through the requirement analysis, design, implementation, and testing. One thing to note is that the test result from our generated tests is redirected to the developer agent in our implementation of $FlowGen_{Waterfall}$ so developers can fix and improve the code.

1) $FlowGen_{TDD}$

In the design of $FlowGen_{TDD}$, we follow the ordered communication pattern and organize the development activities so that testing happens after design and before implementation. Once the tests are written, the developer agent considers the test design when implementing the code. When the implementation is finished, we execute the tests. If a test fails, the developer agent is asked to examine the code and resolve the issue.

2) $FlowGen_{Scrum}$

Compared to *Waterfall* and *TDD*, *Scrum* involves one additional role, the *Scrum Master*. There are also additional *Sprint meetings* among the agents. Note that, different from $FlowGen_{Waterfall}$, we use the agile terminologies in the prompt

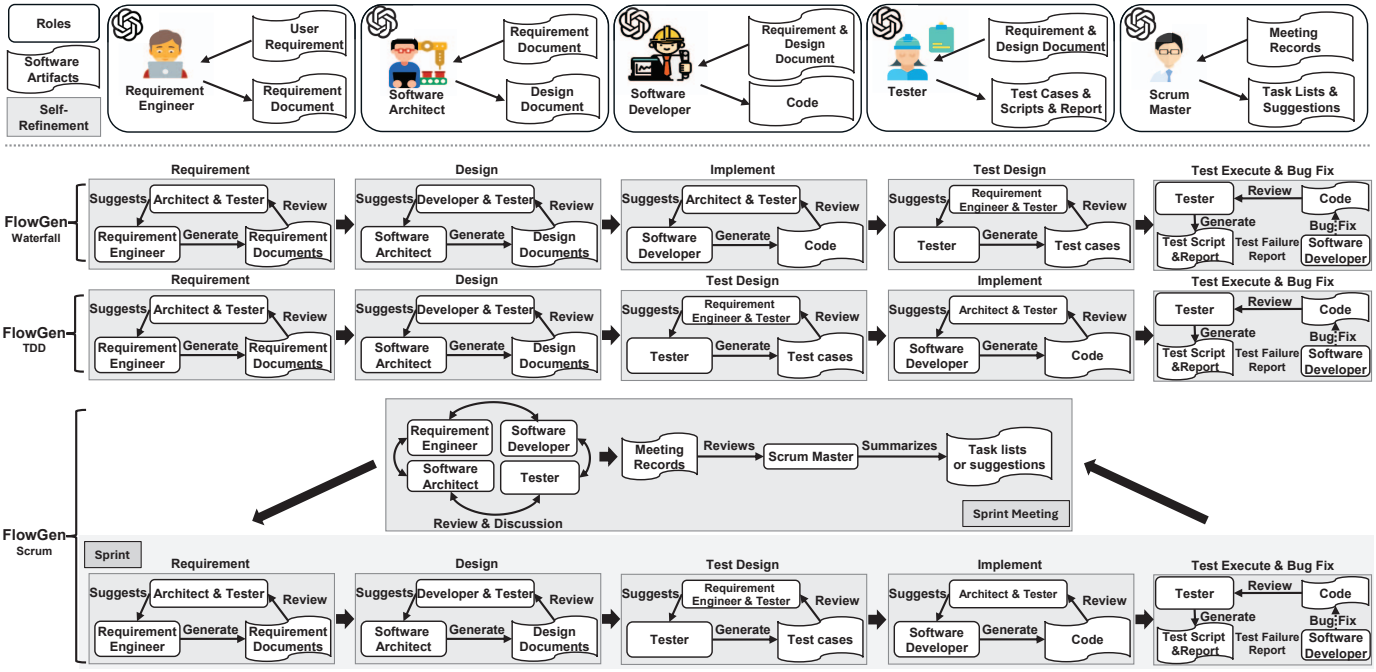

Fig. 1: An overview of $FlowGen_{Waterfall}$, $FlowGen_{TDD}$, and $FlowGen_{Scrum}$.

TABLE I: Tasks, instructions, and corresponding contexts that are used for constructing the prompts for the development roles.

Role	Task	Instruction	Context
Requirement Engineer	Analyze and generate requirement documentation from the context.	1) Analyze the requirement; and 2) Write a requirement document.	Programming problem description.
Architect	Design the overall structure and high-level components of the software.	1) Read the context documents; and 2) Write the design document. The design should be high-level and focus on guiding the developer in writing code.	Requirement document.
Developer	Write code in Python that meets the requirements.	1) Read the context documents; and 2) Write the code. Ensure that the code you write is efficient, readable, and follows best practices.	Requirement and design documents.
	Fix the code so that it meets the requirements.	1) Read the test failure reports and code suggestions from the context; and 2) Rewrite the code.	Original code, test failure report, and suggestions for improvement.
Tester	Design tests to ensure the software satisfies feature needs and quality.	1) Read context documents; and 2) Design test cases.	Requirement and design documents.
	Write a Python test script using the unittest framework.	1) Read the context documents; 2) Write a Python test script; and 3) Follow the input and output given by the requirement.	Test case design and requirement documents.
	Write a test failure report.	1) Read the test execution result; and 2) Analyze and generate a test failure report.	Test execution result.
Scrum Master	Summarize and break down the discussion into a task list for the scrum team.	1) Read and understand the context; and 2) Define the tasks for development roles.	Meeting discussion.

(e.g., we use user story instead of requirement document) when implementing $FlowGen_{Scrum}$. We follow a disordered communication pattern in the design of $FlowGen_{Scrum}$, because, in sprint meetings, every development role can provide their opinion (e.g., to simulate the planning poker process). Every development role, except the *Scrum Master*, reads the common context (e.g., description of the programming problem) from a common buffer. Then, every role provides a discussion comment and is saved back in the buffer. Therefore, every role is aware of all the comments. Then, the *Scrum Master* summarizes the entire discussion and derives a list of user stories for each development role. During the sprint, similarly to *Waterfall* and *TDD*, the four development roles carry out the development activities in sequence. At the end of the sprint, the agents will start another sprint meeting to discuss the next steps, such as releasing the code or needing to fix the code because of test failures.

3) Self-Refinement

We implement self-refinement [25], which tries to refine the LLM-generated result through iterative feedback, to further improve the generated artifacts from every development activity. In all three variations of *FlowGen*, we assign other agents to review the generated artifacts for every development activity and provide improvement suggestions. We assign the agents from both the downstream activity and the tester to

examine the generated artifacts and provide suggestions. The suggestions are then considered for the re-generation of the artifacts. We include the tester in every development activity to emulate the DevTestOps practice [38], where testers are involved in all development activities and provide feedback on the quality aspects. For example, once a requirement engineer generates a requirement document, both the architect and tester would read the document and provide suggestions for improvement. Then, the requirement engineer will re-generate the requirement document based on the previously generated document and suggestions. At the development and testing activity, the tester will generate a test failure report if any of the LLM-generated tests fail or if the code cannot be executed (e.g., due to syntax error). The test failure report is then given to the developer for bug fixing. We repeat the process *t* times to self-refine the generated code. In our implementation, we currently set *t*=3. If the code still cannot pass the test, we repeat the entire software development process.

C. Implementation and Experiment Settings

Environment. We use GPT3.5 (version gpt-3.5-turbo-1106) as our underlying LLM due to its popularity and wide usage in code generation research [20, 36, 11]. We leverage OpenAI's APIs (version 0.28.1) to interact with GPT. We send prompts using JSON format [41] and send all the conversation history as part of the prompt [14]. We set the temperature value to 0.8 and explore the effect of the temperature value in RQ3. We implemented *FlowGen* using *Python 3.9*.

Benchmark Datasets. We follow prior studies [53, 20, 52] and evaluate the code generation result using four benchmarks: HumanEval, HumanEval-ET, MBPP (Mostly Basic Python Programming), and MBPP-ET. These benchmarks contain both the programming problems and tests for evaluation. Given a programming problem, we consider that a generated code snippet is correct if it can pass all the provided tests. HumanEval [8] has 164 programming problems, and MBPP [3] has 427 programming problems (we use the sanitized version released by the original authors) and three test cases for each problem. We also use the dataset published by Dong et al. [10], where they use the same problems as HumanEval and MBPP but offer stronger evaluation test cases (around 100 test cases for each problem, called HumanEval-ET and MBPP-ET). All these benchmarks use Python as the programming language. Each programming problem contains the INPUT pairs <method signature, method description, invoke examples> and expect the code as OUTPUT.

Evaluation Metric. To evaluate the quality of the generated code, we use the Pass@K metric [11, 8]. Pass@K evaluates the first K generated code's functional accuracy (i.e., whether the generated code can pass all the test cases). In this work, we set K=1 to evaluate if the first generated code can pass all the provided test cases. A Pass@1 of 100 means 100% of the generated code can pass all the tests in the first attempt. We use Pass@1 because it is a stricter criterion, reflecting situations where developers do not have the groundtruth for automatically evaluating multiple attempts.

TABLE II: Average and standard deviation of the Pass@1 accuracy across five runs, with the best Pass@1 marked in bold. The numbers in the parentheses show the percentage difference compared to *RawGPT*. Statistically significant differences are marked with a "*".

	HumanEval	HumanEval-ET	MBPP	MBPP-ET
RawGPT	64.4±3.7	49.8±3.0	77.5±0.8	53.9±0.7
$FlowGen_{Waterfall}$	69.5±2.3 (+7.9%)*	59.4±2.5 (+19.2%)*	76.3±0.9 (-1.5%)	51.1±1.7 (-5.2%)*
$FlowGen_{TDD}$	69.8±2.2 (+8.4%)*	60.0±2.1 (+20.5%)*	76.8±0.9 (-1.0%)	52.8±0.7 (-2.1%)*
$FlowGen_{Scrum}$	**75.2±1.1 (+16.8%)***	**65.5±1.9 (+31.5%)***	**82.5±0.6 (+6.5%)***	**56.7±1.4 (+5.2%)***

IV. Results

We evaluate *FlowGen* with four research questions (RQs).

RQ1: What is the code generation accuracy of FlowGen?

Motivation. In this RQ, we emulate the three process models using LLM agents and compare their results on code generations. Such results may provide invaluable evidence for future researchers seeking to optimize process models for code generation within their specific business domain.

Approach. As a baseline for comparison, we directly give the programming problems to ChatGPT (which we refer to as *RawGPT*). Although prior works [17, 19, 27, 20] show that few-shot learning can improve the results from LLMs, they can be biased on how the few-shot samples are selected [44].. Hence, we use zero-shot learning in our experiment. To control for randomness in the experiment, we ensure all these experiments use the same temperature value (t=0.8) and the same model version (*gpt-3.5-turbo-1106*). Finally, we repeat each *FlowGen* five times and report the average Pass@1 and standard deviation across the runs. We also conduct a student's t-test to study if *FlowGen*'s results are statistically significantly different from *RawGPT*.

Result. ***$FlowGen_{Scrum}$ shows a consistent improvement over RawGPT, achieving 5.2% to 31.5% improvement in Pass@1.***
Table II shows the Pass@1 accuracy of *FlowGen* across different process models on the benchmark datasets studied. As shown in the Table II, for HumanEval and HumanEval-ET, all of the studied process models have 7.9% to even 31.5% improvement in Pass@1 compared to *RawGPT*, and the improvements are all statistically significant ($p <= 0.05$). For MBPP and MBPP-ET, $FlowGen_{Scrum}$ also has statistically significant improvements of 5.2% to 6.5%, even though we see a slight decrease when adopting $FlowGen_{Waterfall}$ and $FlowGen_{TDD}$ to MBPP and MBPP-ET.

Despite slight variations in code generation responses form LLM across executions, **we find stable standard deviations of Pass@1, ranging from 0.6% to 3.7% across all process models and benchmarks**. In particular, $FlowGen_{Scrum}$ has the lowest standard deviation (0.6% to 1.9%, an average of 1.2%), while *RawGPT* has the highest standard deviation (0.5% to 3.7%, an average of 2%). Following $FlowGen_{Scrum}$, $FlowGen_{Waterfall}$ has the second highest standard deviation, with $FlowGen_{TDD}$ is ranking third. In conclusion, although the models generally have consistent Pass@1 across runs, $FlowGen_{Scrum}$ consistently produces the most stable results.

TABLE III: *FlowGen* test failure categorization. Failure types are generated from Python Interpreter [30]. Darker red indicates higher percentages of the failure categories in the generated code across the models. Percentages are calculated by the ratio of specific failure types to the total number of failed tests across different process models.

		Failure Categories								
Benchmarks	Model	Assertion	Syntax	Name	Type	Index	Value	Recursion	Attribute	Total
HumanEval	RawGPT	36 (24%)	18 (100%)	11 (58%)	2 (17%)	2 (50%)	1 (12%)	0 (0%)	0 (0%)	70 (33%)
	Waterfall	39 (26%)	0 (0%)	3 (16%)	3 (25%)	0 (0%)	1 (12%)	0 (0%)	0 (0%)	46 (22%)
	TDD	39 (26%)	0 (0%)	4 (21%)	3 (25%)	1 (25%)	5 (62%)	0 (0%)	0 (0%)	52 (24%)
	Scrum	38 (25%)	0 (0%)	1 (5%)	4 (33%)	1 (25%)	1 (12%)	0 (0%)	0 (0%)	45 (21%)
HumanEval-ET	RawGPT	39 (21%)	9 (82%)	13 (43%)	1 (25%)	4 (57%)	2 (18%)	0 (0%)	0 (0%)	68 (27%)
	Waterfall	49 (26%)	1 (9%)	7 (23%)	0 (0%)	1 (14%)	3 (27%)	0 (0%)	0 (0%)	61 (24%)
	TDD	50 (26%)	1 (9%)	6 (20%)	2 (50%)	1 (14%)	4 (36%)	0 (0%)	0 (0%)	64 (25%)
	Scrum	52 (27%)	0 (0%)	4 (13%)	1 (25%)	1 (14%)	2 (18%)	0 (0%)	0 (0%)	60 (24%)
MBPP	RawGPT	66 (23%)	7 (70%)	7 (41%)	7 (25%)	2 (67%)	0 (0%)	1 (100%)	1 (50%)	91 (26%)
	Waterfall	76 (27%)	1 (10%)	5 (29%)	6 (21%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	88 (25%)
	TDD	86 (30%)	1 (10%)	5 (29%)	8 (29%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	100 (29%)
	Scrum	58 (20%)	1 (10%)	0 (0%)	7 (25%)	1 (33%)	0 (0%)	0 (0%)	1 (50%)	68 (20%)
MBPP-ET	RawGPT	133 (24%)	8 (57%)	22 (26%)	26 (24%)	2 (50%)	3 (8%)	1 (100%)	1 (100%)	196 (24%)
	Waterfall	148 (27%)	2 (14%)	22 (26%)	27 (25%)	0 (0%)	15 (39%)	0 (0%)	0 (0%)	214 (27%)
	TDD	147 (27%)	2 (14%)	23 (27%)	28 (26%)	1 (25%)	10 (26%)	0 (0%)	0 (0%)	211 (26%)
	Scrum	123 (22%)	2 (14%)	18 (21%)	28 (26%)	1 (25%)	10 (26%)	0 (0%)	0 (0%)	182 (23%)

There are potential issues in the tests provided by the benchmarks, which may hinder the Pass@1 of FlowGen. Table III provides a breakdown of failure types from the Python Interpreter [30] across various process models and benchmarks. For example, `IndexError` happens when the generated code does not handle an out-of-bound index, causing an exception to be thrown. While we repeat our experiment 5 times, the standard deviation across runs is low; hence, we represent test failure from only one of the runs. Aligned with the findings from Table II, $FlowGen_{Scrum}$ has the lowest `AssertionError` compared to other models (i.e., higher pass rate). We also notice that `SyntaxError` is more evident in *RawGPT* , as expected due to the absence of a code review and testing process. However, there are still higher test failures in $FlowGen_{Waterfall}$ and $FlowGen_{TDD}$ caused by increased occurrences of `ValueError`, `TypeError`, `IndexError` and `NameError`, for MBPP and MBPP-ET as seen in Table III. Upon manual investigation of prevalent test failures types, we discover that $FlowGen_{Waterfall}$ and $FlowGen_{TDD}$ **introduce various validations and enforce programming naming conventions in the generated code,** ***which may help improve code quality but cause tests to fail***. For example, Listing 1 depicts the provided tests and the generated code for MBPP-582, of which the objective is to *"Write a function to check if a dictionary is empty or not"*. While *RawGPT* passed the provided tests, $FlowGen_{Waterfall}$ and $FlowGen_{TDD}$ failed. This failure is because the generated code contains *strict input validation* to check that the input should be of type *dict*. However, the MBPP-582 provided test uses an input of the type *set*, which fails the validation, causing a `TypeError` exception. Moreover, $FlowGen_{Waterfall}$ and $FlowGen_{TDD}$ enforce the common naming convention format and more meaningful function name (e.g., `my_dict` v.s. `is_dict_empty`), both of which causes `NameError` exception due to *wrong function declaration*, causing test failure. More interestingly, we find that such code standardization may also be misled by the requirement provided by the benchmark itself. For example, the MBPP-582 requirement specifies expected input as `dict`, yet provides a `set` type as the input to the test. The LLM code generation indeed captures this correct requirement by validating that input must be of type `dict`. Such inconsistency in the benchmark may reduce the Pass@1.

```
# MBPP-582: check if a dictionary is empty
# MBPP Test Case
def Test():
    assert my_dict({10})==False # {10} is a set not a dict
# rawGPT's answer
def my_dict(dict1):
    return len(dict1) == 0
# Waterfall/TDD's answer, the input type must be a dict
def is_dict_empty(input_dict): # function name is renamed
     from my_dict
    if not isinstance(input_dict, dict):
        raise TypeError("Input is not a dictionary")
    return True if not input_dict else False
```

Listing 1: MBPP-582 Test Failure due to *Strict Input Validation* and *Wrong Function Name*.

In MBPP-794 (Listing 2), test cases provided by MBPP-ET change the return value from a `boolean` (as is the case in MBPP) to the string ``match!''. Moreover, in MBPP-797, MBPP-ET's test capitalized the last word in uppercase (`range` v.s. ``R''ange`). Such non-standard evaluation leads to unfair test results (leads to failure), which may bias the experimental results for MBPP-ET. Such bias suggests that the decrease in Pass@1 rates for MBPP-ET is not solely due to an increase in the number of provided tests.

```
# Example1: Changed return type from boolean to string
assert text_starta_endb("aabbbb") # MBPP 794
assert text_starta_endb("aabbbb")==('match!') # MBPP-ET 794
# Example2: Capitalized the last character in function name
assert sum_in_range(2,5) == 8 # MBPP-797
assert sum_in_Range(2,5) == 8 # MBPP-ET-797
```

Listing 2: MBPP-794 & MBPP-797 Test Failure due to Irregularity in Test Cases.

$FlowGen_{Scrum}$ achieves the best results, with a Pass@1 that is 5.2% to 31.5% better than *RawGPT*. $FlowGen_{Scrum}$ also has the most stable results (average standard decision of 1.3% across all benchmarks) among all models. Notably, while $FlowGen_{Waterfall}$ and $FlowGen_{TDD}$ enhance code quality, such improvements may result in test failures.

RQ2: How do different development activities impact the quality of the generated code?

Motivation. As observed in RQ1, various process models can indeed affect the functional correctness (Pass@1) of the generated code. However, it is equally crucial to understand code quality issues such as code smells and the impact of a development activity on the generated code. This understanding is essential for assessing whether the generated code adheres to industry best practices. Moreover, such insight may offer valuable opportunities for enhancing the design, readability, and maintainability in auto-generated code.

TABLE IV: Pass@1 and *Error/Warning/Convention/Refactor/Handled-Exception* density (per 10 lines of code) in the full *FlowGen* (with all the development activities) and after removing a development activity. A ***lower*** error/warning/convention/refactor is preferred, and a ***higher*** handled-exception is preferred. Darker red indicates a larger decrease in percentages, while darker green indicates a larger increase in percentages.

Model	Dev. Activities	HumanEval						MBPP					
		Pass@1	Error	Warning	Convention	Refactor	Handled Exception	Pass@1	Error	Warning	Convention	Refactor	Handled Exception
RawGPT	–	64.4	0.25	0.19	0.39	0.30	0.00	77.47	0.22	0.20	1.18	0.30	0.01
FlowGen$_{Waterfall}$	full	69.5	0.01	0.12	0.24	0.21	0.37	76.35	0.03	0.12	0.47	0.23	0.67
	rm-requirement	-1.2 (1.7%)	0.0 (0.0%)	-0.01 (8.3%)	-0.02 (8.3%)	-0.01 (4.8%)	-0.06 (16.2%)	+0.7 (0.9%)	-0.02 (66.7%)	-0.02 (16.7%)	0.0 (0.0%)	+0.02 (8.7%)	-0.09 (13.4%)
	rm-design	-1.2 (1.7%)	+0.01 (100.0%)	+0.02 (16.7%)	+0.02 (8.3%)	0.0 (0.0%)	-0.15 (40.5%)	-1.64 (2.1%)	-0.01 (33.3%)	0.0 (0.0%)	+0.02 (4.3%)	+0.01 (4.3%)	-0.2 (29.9%)
	rm-codeReview	-2.4 (3.5%)	0.0 (0.0%)	0.0 (0.0%)	-0.03 (12.5%)	+0.02 (9.5%)	-0.09 (24.3%)	+0.46 (0.6%)	-0.02 (66.7%)	+0.07 (58.3%)	-0.02 (4.3%)	-0.02 (8.7%)	-0.17 (25.4%)
	rm-test	-39.0 (56.1%)	+0.17 (1700.0%)	+0.01 (8.3%)	+0.07 (29.2%)	+0.1 (47.6%)	+0.1 (27.0%)	-23.7 (31.0%)	+0.09 (300.0%)	+0.05 (41.7%)	+0.36 (76.6%)	+0.01 (4.3%)	-0.01 (1.5%)
FlowGen$_{TDD}$	full	69.8	0.01	0.08	0.33	0.27	0.33	76.77	0.04	0.13	0.71	0.28	0.62
	rm-requirement	-2.9 (4.2%)	+0.01 (100.0%)	+0.01 (12.5%)	0.0 (0.0%)	-0.03 (11.1%)	-0.1 (30.3%)	+1.92 (2.5%)	-0.02 (50.0%)	+0.02 (15.4%)	+0.03 (4.2%)	0.0 (0.0%)	-0.27 (43.5%)
	rm-design	-2.9 (4.2%)	0.0 (0.0%)	+0.02 (25.0%)	-0.06 (18.2%)	+0.03 (11.1%)	-0.13 (39.4%)	+1.22 (1.6%)	-0.02 (50.0%)	-0.03 (23.1%)	-0.03 (4.2%)	+0.04 (14.3%)	-0.24 (38.7%)
	rm-codeReview	-0.5 (0.7%)	-0.01 (100.0%)	+0.02 (25.0%)	-0.04 (12.1%)	+0.03 (11.1%)	-0.09 (27.3%)	+0.98 (1.3%)	-0.03 (75.0%)	+0.01 (7.7%)	-0.05 (7.0%)	+0.1 (35.7%)	-0.09 (14.5%)
	rm-test	-11.9 (17.0%)	+0.07 (700.0%)	+0.04 (50.0%)	-0.02 (6.1%)	-0.05 (18.5%)	-0.1 (30.3%)	-17.3 (22.5%)	+0.11 (275.0%)	+0.14 (107.7%)	+0.16 (22.5%)	-0.01 (3.6%)	+0.16 (25.8%)
FlowGen$_{Scrum}$	full	75.2	0.00	0.13	0.21	0.24	0.15	82.48	0.02	0.17	0.51	0.23	0.44
	rm-requirement	+1.0 (1.3%)	0.0 (0.0%)	0.0 (0.0%)	+0.05 (23.8%)	-0.01 (4.2%)	+0.09 (60.0%)	-1.92 (2.3%)	+0.01 (50.0%)	0.0 (0.0%)	+0.01 (2.0%)	-0.01 (4.3%)	+0.04 (9.1%)
	rm-design	+1.0 (1.3%)	0.0 (0.0%)	-0.02 (15.4%)	-0.03 (14.3%)	0.0 (0.0%)	-0.03 (20.0%)	+0.89 (1.1%)	-0.01 (50.0%)	-0.04 (23.5%)	+0.11 (21.6%)	+0.03 (13.0%)	-0.19 (43.2%)
	rm-codeReview	-2.0 (2.7%)	0.0 (0.0%)	0.0 (0.0%)	-0.03 (14.3%)	0.0 (0.0%)	-0.03 (20.0%)	+0.19 (0.2%)	0.0 (0.0%)	-0.01 (5.9%)	+0.01 (2.0%)	+0.06 (26.1%)	-0.1 (22.7%)
	rm-test	-14.2 (18.9%)	+0.03 (588.2%)	+0.06 (46.2%)	0.0 (0.0%)	-0.03 (12.5%)	+0.07 (46.7%)	-26.5 (32.1%)	+0.13 (650.0%)	+0.1 (58.8%)	+0.41 (80.4%)	-0.03 (13.0%)	+0.14 (31.8%)
	rm-sprintMeeting	-1.6 (2.1%)	0.0 (0.0%)	-0.01 (7.7%)	-0.02 (9.5%)	-0.03 (12.5%)	+0.1 (66.7%)	-3.09 (3.7%)	0.0 (0.0%)	-0.02 (11.8%)	+0.05 (9.8%)	-0.01 (4.3%)	+0.21 (47.7%)

Approach. To study the impact of each development activity on code quality, we remove each activity separately and re-execute *FlowGen*. For example, we first remove the requirement activity in *FlowGen*$_{Waterfall}$ and execute *FlowGen*$_{Waterfall}$. Then, we add the requirement activity back and remove the design activity. We repeat the same process for every development activity. Note that we cannot remove the coding activity since our goal is code generation. Hence, we removed code review at the end of the coding activity.

Code quality considers numerous facets beyond mere functional correctness [48, 46]. Other factors, such as code smells, maintainability, and readability, are also related to code quality. Hence, to gain a comprehensive understanding of how code quality changes, we 1) apply a static code analyzer to detect code smells in the generated code and 2) study code reliability by analyzing the exception handling code. To study code smell, design, and readability, we apply Pylint 3.0.4 [37, 9] (a Python static code analyzer) on the generated code. Pylint classifies the detected code smells into different categories such as *error*, *warning*, *convention*, and *refactor*.

We study how the number of detected code smells in each category changes when removing an activity. Since the generated code may have different lengths, we report the density of the code smells in each category. We calculate the code smell density as the total number of code smell instances in a category (e.g., *error*) divided by the total lines of code. To study reliability, we calculate the density of handled exceptions (total number of handled exceptions divided by the total lines of code) since exceptions are one of the most important mechanisms to detect and recover from errors [12]. For better visualization, we present the density results as per every 10 lines of code. We also ensure reliability of our results by repeating all of the aforementioned approach three times.

Result. ***Testing has the largest impact on the functional correctness of the code, while other development activities only have small impacts. Removing testing causes Pass@1 to decrease by 17.0% to 56.1%.*** Table IV presents changes in Pass@1 and the densities of code smell and handled exceptions. We show the results for HumanEval and MBPP because they share the same programming problems and generated code with the other two benchmarks. Among all development activities, testing has the largest impact on Pass@1, where removing testing causes a large decrease in Pass@1 (17.0% to 56.1% decrease). The finding implies that LLM's generated tests are effective in improving the functional correctness of code. In both benchmarks, removing sprint meetings in *FlowGen*$_{Scrum}$ also causes Pass@1 to drop. However, removing other activities only has a small and inconsistent effect on Pass@1. For example, in HumanEval, removing requirement, design, and code review generally causes Pass@1 to decrease (except for *FlowGen*$_{Scrum}$), but removing these activities improves Pass@1 in MBPP. In other words, most development activities do not significantly contribute to the functional correctness of the generated code.

As shown in Table IV, eliminating test activities significantly boosts *error* and *warning* smell densities by an average of 702.2% and 52.2%, respectively. Omitting design raises refactor smell density by an average of 7.1%, and skipping code review leads to a 14.2% average increase in *warning* density. However, removing other development activities shows either a small or inconsistent impact. We also find some differences in the artifacts generated by different roles. For example, although both roles generate documents, requirement engineers specify the acceptance criteria, while architects address time/space complexity. In short, the findings show that **adding design, testing, and code review can help reduce**

the density of code smell in the generated code.

Having design and code review activities significantly improves reliability by increasing the density of handled exceptions, while other development activities only have small or no impacts. Removing design and code review activities separately causes the handled exception density to decrease from 20.0% to 43.2% and 14.5% to 27.3%, respectively. Namely, these two activities add exception handling in the generated code, which may help improve reliability. Removing other activities shows a mixed relationship with the density of the handled exception. For example, removing testing in $FlowGen_{TDD}$ causes an increase of handled exception density by 25.8% in MBPP (i.e., testing removes exception handling code) while causing a decrease of 30.3% in HumanEval (i.e., testing adds exception handling code). While the effect of each development may be related to the nature of the benchmarks, our findings show that, in both benchmarks, **adding design and code review activities can help improve code reliability by handling more exceptions in the generated code**.

FlowGen shows consistent improvement over RawGPT in the quality of the generated code: decreasing the density of error/warning/convention/refactor code smells (6.7% to 96.0%) while significantly increasing handled exception density. The code generated by *RawGPT* has higher *error/warning/convention/refactor* code smell densities than that of $FlowGen_{Waterfall}$, $FlowGen_{TDD}$, and $FlowGen_{Scrum}$. This finding shows all three models improve the quality of the generated code to different degrees. Specifically, compared to *RawGPT*, *FlowGen* decreases the *error* code smell density by 81.8% to 96.0%, *warning* density by 15.0% to 57.9%, *convention* density by 15.4% to 60.2%, and *refactor* density by 6.7% to 30.0%. Meanwhile, *RawGPT* has fewer handled exceptions than *FlowGen*. As Table IV shows, in both HumanEval and MBPP, *RawGPT* has almost zero handled exception, while $FlowGen_{Waterfall}$ generates the most handled-exception (0.37 and 0.67 handled exceptions per every 10 LOC in the two benchmarks), $FlowGen_{TDD}$ ranks second (0.33 and 0.62), and then $FlowGen_{Scrum}$ (0.15 and 0.44). In short, *FlowGen* improves the quality of the generated code by reducing code smells while adding more exception-handling code.

Compared to *RawGPT*, *FlowGen* remarkably improves the quality of the generated code by reducing code smells and adding more exception handling. Testing has the most significant impact on Pass@1 and code smells among all development activities, while having design and code review greatly improve the exception-handling ability.

RQ3: How stable is the FlowGen generated code?

Motivation. In LLM, the stability of generated responses can be influenced by several parameters: 1) *temperature*, affecting the randomness in the generated responses, and 2) *model versions*, which may introduce variability due to changes in optimization and fine-tuning [7]. Understanding and improving the stability of LLMs is crucial for enhancing their trustworthiness, thereby facilitating their adoption in practice. Therefore, in this research question (RQ), we investigate the stability of our *FlowGen* in Pass@1 across four benchmarks, considering various temperature values and model versions.

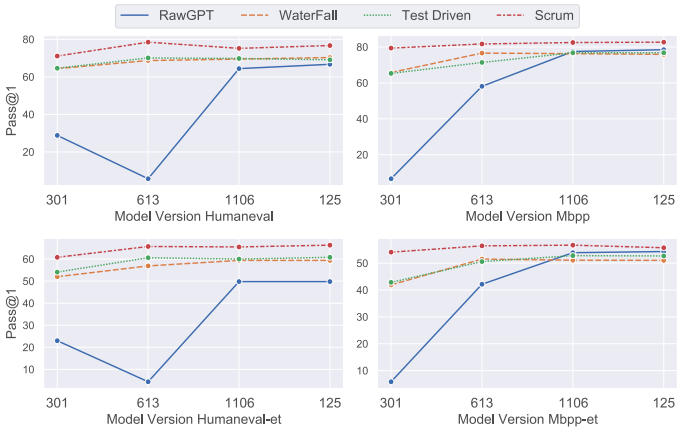

Fig. 2: Pass@1 across GPT3.5 versions.

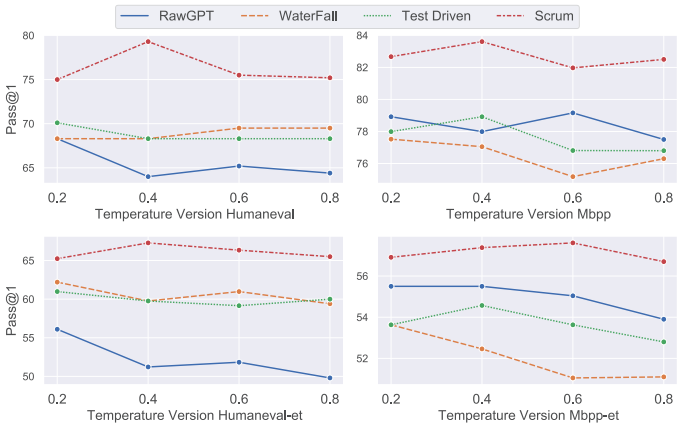

Fig. 3: Pass@1 across temperature values.

Approach. We evaluate the Pass@1 of *FlowGen* across four versions of GPT3.5: *turbo-0301*, *turbo-0613*, *turbo-1106*, and *turbo-0125*. The latest version is *turbo-0125* (published in January 2024), and the earlier version is *turbo-0301* (published in March 2023). To avoid the effect of the model version when we vary the temperature, we use the same model version (*turbo-1106*, the version that we used in prior RQs) to study the effect of temperature values. We set the temperature to 0.2, 0.4, 0.6, and 0.8 in our experiment. We execute *RawGPT* and the three variants of *FlowGen* three times under each configuration and report the average Pass@1.

Result. ***RawGPT has extremely low Pass@1 in some versions of GPT3.5, while FlowGen has stable results across all versions. FlowGen may help ensure the stability of the generated code even when the underlying LLM regresses.*** Figure 2 shows the Pass@1 for *RawGPT* and the three *FlowGen* across GPT3.5 versions. In earlier versions of GPT3.5 (*0301* and *0613*), *RawGPT* has very low Pass@1 on all benchmarks (e.g., 20 to 30 in HumanEval and HumanEval-ET). In *0301*, MBPP's Pass@1 is even lower with a value around 5. **The findings show that model version may have a significant impact on the generated code**. However, we see that, after adopting our

agent-based techniques, all three variants of *FlowGen* achieve similar Pass@1 across GPT3.5 versions. The results indicate that *FlowGen* can generate similar-quality code even if we have an underperformed baseline model.

All techniques have a relatively similar Pass@1 when the temperature value changes. Figure 3 shows the Pass@1 for all the techniques when the temperature value changes. There is a slight downward trend for *RawGPT* when t increases, but the changes are not significant (Pass@1 is decreased by 2 to 5). For *FlowGen*, and especially $FlowGen_{Scrum}$, we see similar Pass@1 regardless of the temperature value. Although we see a slight increase in the Pass@1 of $FlowGen_{Scrum}$ when t=0.4 (2 to 5 higher compared to when t=0.8 across the benchmarks), the difference is small, and the Pass@1 is almost the same when t is either the lowest (0.2) or largest value (0.8). In short, although temperature values may have an impact on the generated code, the effect is relatively small for *FlowGen*.

FlowGen generates stable results across GPT versions, while we see large fluctuations (14 times difference) in *RawGPT*'s Pass@1. Pass@1 is generally consistent across all models when the temperature value changes.

RQ4: How does FlowGen compare with other techniques?

Motivation. *FlowGen* is designed to organize agents to emulate process models and can be combined with other code generation techniques. However, it is crucial to evaluate its performance relative to these techniques to assess the effectiveness in emulating software process models for code generation. While many code generation results use the same benchmarks, our evaluation results cannot be directly compared with other agent-based code generation works [14, 15, 16, 5, 36] due to missing information on model versions, temperature values, post-processing steps, specific prompts, or the selection of few-shot samples. Therefore, in this RQ, we compare $FlowGen_{Scrum}$ with other LLM-based baselines under the same environment settings. Moreover, we evaluate an integrated version of $FlowGen_{Scrum}$ to showcase how existing prompting techniques can be combined with *FlowGen*.

Approach. We compare against two state-of-the-art techniques: *CodeT* [5] and *Reflexion* [36]. *CodeT* employs self-generated tests to evaluate the quality of generated code, which is similar to the testing phase of *FlowGen*. *Reflexion* is an agent-based technique that achieves state-of-the-arts Pass@1 on the benchmarks. We apply these two using their released code, replacing the LLM version and temperature with those used by $FlowGen_{Scrum}$, and repeat the experiment five times.

Result. ***While $FlowGen_{Scrum}$ and CodeT achieve similar results, with $FlowGen_{Scrum}$ having a higher Pass@1 in MBPP, they both have higher Pass@1 than Reflexion on all benchmarks (statistically significant).*** Table V shows the Pass@1 of the techniques. $FlowGen_{Scrum}$ has a statistically significantly higher Pass@1 than *CodeT* in MBPP and similar Pass@1 in other benchmarks (no statistically significant difference). Both techniques achieve higher Pass@1 than *Reflexion* (statistically significant). *Reflexion*'s MBPP results are even worse than *RawGPT*. This observation aligns with the original study [36] where reported similar performance degradation.

TABLE V: Average and standard deviation of Pass@1 across five runs, with the best Pass@1 marked in bold.

	HumanEval	HumanEval-ET	MBPP	MBPP-ET
Reflexion	71.3±1.5	55.7±2.8	71.7±0.8	52.0±0.7
CodeT	75.7±0.4	66.9±0.4	79.9±1.3	56.7±0.9
$FlowGen_{Scrum}$	75.2±1.1	65.5±1.9	82.5±0.6	56.7±1.4
$FlowGen_{Scrum+Test}$	**79.3±1.6**	**67.7±1.1**	**83.8±0.6**	**58.7±1.3**

Integrating CodeT to $FlowGen_{Scrum}$ further brings statistically significant improvements of Pass@1 by up to 5%. *CodeT* is a general technique where it repeats the code generation and selects the code that passes the most self-generated tests. Hence, as a pilot study, we made the developer agent repeats the implementation activity multiple times to produce several versions of the code (i.e., $FlowGen_{Scrum+Test}$). The developer agent then generates multiple test assertions to identify the version with the highest pass rate. The selected code is subsequently submitted to the next stage: the testing activity. Our findings indicate that $FlowGen_{Scrum+Test}$ outperforms $FlowGen_{Scrum}$ and *CodeT*, achieving an average Pass@1 score of 83.8, 58.3, 79.3, and 67.7 on HumanEval, HumanEval-ET, MBPP, and MBPP-ET, respectively. This provides statistically significant improvement over both $FlowGen_{Scrum}$ and *CodeT*. Our finding highlights the potential of *FlowGen* in boosting the performance of other code generation techniques (and vice versa). Future studies can refine *FlowGen* to incorporate enhancement to each activity for further improvement. To support these efforts, we have made our code publicly available [2] to facilitate further adoption and allow researchers to experiment with different software process models.

Both $FlowGen_{Scrum}$ and *CodeT* outperform *Reflexion* in Pass@1 across all benchmarks, with $FlowGen_{Scrum}$ and *CodeT* demonstrating similar results. The incorporation of *CodeT* into $FlowGen_{Scrum}$ further enhances performance, achieving the highest Pass@1 scores, highlighting the potential of *FlowGen* for code generation tasks.

V. Discussion & Future Works

Role of Human Developers in *FlowGen*. Although software process models were originally designed for human-centric development rather than for LLMs, our empirical findings suggest that certain elements of these processes can contribute to better code quality. Every activity in the process model also has different impacts on the generated artifacts. Future research should examine the incorporation of human developers into various phases of the code generation process. Specifically, humans can play critical roles in the following stages: ***(1) Pre-Execution of FlowGen***: different process models exhibit varying quality (e.g., smell and accuracy). Humans are instrumental in selecting the most appropriate model for a given task. Humans are also essential in providing initial requirements and design specifications. ***(2) During FlowGen***

Execution: Humans can oversee review meetings and assist in reviewing/improving generated artifacts. For example, humans can validate the generated requirements with product managers, or verify the quality of the generated code by manual inspection and debugging. The improvement in each activity can also impact the subsequent activities and, hence, affect the final artifacts. ***(3) Post-Execution of FlowGen***: Following the code-generation phase, humans can either accept the generated artifact or request further refinements, offering additional requirements as needed to better meet project goals.

Quality of Code Generation Benchmarks We manually validated all coding problems that failed the tests. We found that the majority of quality issues within the benchmark were in MBPP and MBPP-ET (e.g., bad naming convention or inconsistent test definition). These issues may contribute to reduced Pass@1 scores due to factors beyond logic in the code. It is also important to acknowledge that other benchmarks might present unique challenges that could similarly affect Pass@1 evaluations. Hence, a crucial research direction is to conduct a thorough evaluation of benchmarks for more diverse and accurate evaluations of code generation approaches.

VI. Threats to Validity

Internal validity. Due to the generative nature of LLM, the responses may change across runs. Variables such as temperature and LLM model version can also impact the generated code. We set the temperate value to be larger than 0 because we want LLMs to be more creative. To mitigate the threat, we execute the LLMs multiple times. As found in RQ1, the standard deviation of the results is small, so the generated results should be consistent. In RQ3, we conducted the experiments using different temperature and versions. The temperature only has a small effect on Pass@1, and versions have a large impact on *RawGPT*. In RQ2, we study the impact of removing every activity. However, having multiple activities may have a tandem effect that further improves code quality. Future studies are needed to study the effects of different combinations of development activities in code generation.

External validity. We conduct our study using state-of-the-art benchmarks. However, as we discussed, there exist some issues in the provided tests. Moreover, the programming problems are mostly algorithmic, so the findings may not generalize to other. Future studies should consider applying *FlowGen* on different programming tasks. We use GPT3.5 as our underlying LLM. Although one can easily replace the LLM in our experiment, the findings may be different. Future studies on how the results of *FlowGen* change when using different LLMs.

Construct validity. We implement an agent system that follows various software development processes. However, there are many variations of the same process model, and some variations may give better results. Future studies should explore how changing the process models affect the code generation ability. One limitation is the correctness of the generated tests. However, we found that the generated tests still contribute to improving the quality of the generated code. Similar findings are reported on CodeT [5], where generating tests helps improve code correctness. However, future studies should focus on further improving the generated tests by using traditional software engineering techniques to estimate the oracles or select higher-quality tests (e.g., mutation testing).

VII. Conclusion

In this paper, we emulate various roles in software development, using LLM agents, and structuring their interactions according to established process models. We introduce *FlowGen*, a framework that implements three renowned process models: $FlowGen_{Waterfall}$, $FlowGen_{TDD}$, and $FlowGen_{Scrum}$. We evaluated how these models affect code generation in terms of correctness and code quality on four benchmarks: HumanEval, HumanEval-ET, MBPP, and MBPP-ET. Our findings show that $FlowGen_{Scrum}$ notably enhances Pass@1 by an average of 15% over *RawGPT*, while maintaining the lowest standard deviation (averaging 1.2%). Moreover, we find that development activities such as design and code review significantly reduce code smells and increase the presence of handled exceptions. This indicates that *FlowGen* not only boosts code correctness but also reduces code smells and improves reliability. Compared with other state-of-the-art techniques, $FlowGen_{Scrum}$ and *CodeT* achieved similar results, with both outperforming *Reflexion*. Integrating *CodeT* into $FlowGen_{Scrum}$ further resulted in statistically significant improvements, achieving the highest Pass@1 scores. These insights pave the way for future research to develop innovative development models tailored for LLM integration in software development processes.

In this study, we introduced *FlowGen*, a framework designed to emulate software process models using Large Language Model (LLM) agents, each representing roles such as requirement engineers, architects, developers, and testers. We implemented three variations of *FlowGen*: $FlowGen_{Waterfall}$, $FlowGen_{TDD}$, and $FlowGen_{Scrum}$. Our evaluation across four benchmarks–HumanEval, HumanEval-ET, MBPP, and MBPP-ET–demonstrated the superior performance of $FlowGen_{Scrum}$, achieving up to 31.5% improvement in Pass@1 over *RawGPT*.

Our results showed that incorporating software process models into LLM-based code generation significantly enhances code correctness, reduces smells, and improves exception handling. FlowGenScrum consistently outperformed other models, achieving the highest Pass@1 and the lowest standard deviation, indicating more stable and reliable code generation. Additionally, our comparative analysis with state-of-the-art techniques revealed that FlowGenScrum and CodeT achieved similar results, both outperforming Reflexion. Notably, integrating CodeT into FlowGenScrum resulted in statistically significant improvements, achieving the highest Pass@1. This highlights the robustness and potential of combining structured software development practices with LLM capabilities.

Future research should focus on refining these models, incorporating more sophisticated interactions, and expanding the scope of evaluation to include a broader range of development tasks and environments. By doing so, we can better understand the capabilities of LLMs in software engineering and improve their integration into practical development workflows.

REFERENCES

[1] Eman Abdullah AlOmar, Anushkrishna Venkatakrishnan, Mohamed Wiem Mkaouer, Christian D Newman, and Ali Ouni. How to refactor this code? an exploratory study on developer-chatgpt refactoring conversations. *arXiv preprint arXiv:2402.06013*, 2024.

[2] Anonymous. Repository & dataset, 2024. URL https://anonymous.4open.science/r/FlowGen-LLM-E842.

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

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

[5] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. In *11th International Conference on Learning Representations (ICLR)*, 2023.

[6] Jiawei Chen, Hongyu Lin, Xianpei Han, and Le Sun. Benchmarking large language models in retrieval-augmented generation. *arXiv preprint arXiv:2309.01431*, 2023.

[7] Lingjiao Chen, Matei Zaharia, and James Y. Zou. How is chatgpt's behavior changing over time? *ArXiv*, abs/2307.09009, 2023.

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[9] Subhasish Dasgupta and Sara Hooshangi. Code quality: Examining the efficacy of automated tools. 2017.

[10] Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. Codescore: Evaluating code generation by learning code execution. *arXiv preprint arXiv:2301.09043*, 2023.

[11] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt. *arXiv preprint arXiv:2304.07590*, 2023.

[12] Christof Fetzer, Pascal Felber, and Karin Hogstedt. Automatic detection and masking of nonatomic exception handling. *IEEE Transactions on Software Engineering*, 30(8):547–560, 2004.

[13] Martin Fowler. The new methodology, 2005. URL https://www.martinfowler.com/articles/newMethodology.html.

[14] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

[15] Dong Huang, Qingwen Bu, and Heming Cui. Codecot and beyond: Learning to program and test like a developer. *arXiv preprint arXiv:2308.08784*, 2023.

[16] Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023.

[17] Sungmin Kang, Juyeon Yoon, and Shin Yoo. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2312–2323. IEEE, 2023.

[18] Sawan Kumar and Partha Talukdar. Reordering examples helps during priming-based few-shot learning. *arXiv preprint arXiv:2106.01751*, 2021.

[19] Van-Hoang Le and Hongyu Zhang. Log parsing with prompt-based few-shot learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2438–2449. IEEE, 2023.

[20] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. *arXiv preprint arXiv:2305.06599*, 2023.

[21] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct. *Rigorous evaluation of large language models for code generation. CoRR, abs/2305.01210*, 2023.

[22] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, et al. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Comput. Surv.*, 55(9), jan 2023.

[23] Lipeng Ma, Weidong Yang, Bo Xu, Sihang Jiang, Ben Fei, Jiaqing Liang, Mingjie Zhou, and Yanghua Xiao. Knowlog: Knowledge enhanced pre-trained language model for log understanding. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024.

[24] Zeyang Ma, An Ran Chen, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. Llmparser: An exploratory study on using large language models for log parsing. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 883–883. IEEE Computer Society, 2024.

[25] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36, 2024.

[26] E Michael Maximilien and Laurie Williams. Assessing test-driven development at ibm. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 564–569. IEEE, 2003.

[27] Noor Nashid, Mifta Sintaha, and Ali Mesbah. Retrieval-based prompt selection for code-related few-shot learning. In *IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2450–2462, 2023.

[28] OpenAI. Chatgpt. https://chat.openai.com/, 2023.

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

[30] Python. Built-in exceptions, 2005. URL https://docs.python.org/3/library/exceptions.html.

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

[32] Fernando Vallecillos Ruiz, Anastasiia Grishina, Max Hort, and Leon Moonen. A novel approach for automatic program repair using round-trip translation with large language models. *arXiv preprint arXiv:2401.07994*, 2024.

[33] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 2023.

[34] Yunfan Shao, Linyang Li, Junqi Dai, and Xipeng Qiu. Character-llm: A trainable agent for role-playing. *arXiv preprint arXiv:2310.10158*, 2023.

[35] Chaochao Shen, Wenhua Yang, Minxue Pan, and Yu Zhou. Git merge conflict resolution leveraging strategy classification and llm. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*, pages 228–239. IEEE, 2023.

[36] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 2024.

[37] Pylint Development Team. Pylint. https://pypi.org/project/pylint/. Last accessed March 2024.

[38] Testsigma. What is devtestops? https://testsigma.com/devtestops. Last accessed March 2024.

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

[40] 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. *Advances in neural information processing systems*, 35:24824–24837, 2022.

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

[42] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*, 2023.

[43] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. Chatunitest: a chatgpt-based automated unit test generation tool. *arXiv preprint arXiv:2305.04764*, 2023.

[44] Jing Xu, Xu Luo, Xinglin Pan, Yanan Li, Wenjie Pei, and Zenglin Xu. Alleviating the sample selection bias in few-shot learning by removing projection to the centroid. *Advances in Neural Information Processing Systems*, 35: 21073–21086, 2022.

[45] Yuzhuang Xu, Shuo Wang, Peng Li, Fuwen Luo, Xiaolong Wang, Weidong Liu, and Yang Liu. Exploring large language models for communication games: An empirical study on werewolf. *arXiv preprint arXiv:2309.04658*, 2023.

[46] Aiko Yamashita and Leon Moonen. Do code smells reflect important maintainability aspects? In *2012 28th IEEE International Conference on software maintenance (ICSM)*, pages 306–315, 2012.

[47] Hui Yang, Sifu Yue, and Yunzhong He. Auto-gpt for online decision making: Benchmarks and additional opinions. *arXiv preprint arXiv:2306.02224*, 2023.

[48] Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt. *arXiv preprint arXiv:2304.10778*, 2023.

[49] Zhiqiang Yuan, Junwei Liu, Qiancheng Zi, Mingwei Liu, Xin Peng, and Yiling Lou. Evaluating instruction-tuned large language models on code comprehension and generation. *arXiv preprint arXiv:2308.01240*, 2023.

[50] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. No more manual tests? evaluating and improving ChatGPT for unit test generation. *arXiv preprint arXiv:2305.04207*, 2023.

[51] Yuxia Zhang, Zhiqing Qiu, Klaas-Jan Stol, Wenhui Zhu, Jiaxin Zhu, Yingchen Tian, and Hui Liu. Automatic commit message generation: A critical review and directions for future work. *IEEE Transactions on Software Engineering*, 2024.

[52] Li Zhong, Zilong Wang, and Jingbo Shang. Ldb: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*, 2024.

[53] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*, 2023.