SeDev: Structured Semantic Exploration for LLM-Driven Code Generation

Ronghui Yang[1,2,3*], **Jie Liu**[1,2,3*], **Jiajie Zeng**[1,2,3*], **Jiexin Wang**[1,2,3], **Jiuchuan Jiang**[5], **Bo An**[4], **Yi Cai**[1,2,3†], **Mengchen Zhao** [1,2,3†]

[1]School of Software Engineering, South China University of Technology
[2]Key Laboratory of Big Data and Intelligent Robot (SCUT), MOE of China
[3]Joint Guangdong-Hong Kong-Macao Research Laboratory of Big Data and Robotic Intelligence, Ministry of Education
[4]Nanyang Technological University, Singapore
[5]Nanjing University of Finance and Economics, Nanjing, China

Abstract

Large Language Models (LLMs) have shown remarkable capabilities in automating code generation. Recent approaches that incorporate feedback refinement mechanisms into the generation process have further enhanced software generation quality. However, these methods can be characterized as single-path approaches, which suffer from insufficient exploration of the vast solution space, often causing even the most powerful models to get stuck in local optima and struggle to generate the desired software. Some other works use Monte Carlo Tree Search (MCTS) to explore multiple paths for finding the best solution; yet, MCTS can be extremely inefficient in practice. To this end, we propose SeDev, a novel LLM-driven code generation framework that efficiently finds high-quality solutions in only a few iterations. The core idea of SeDev is to gradually explore semantically adjacent solutions through structured prompt guidance and feedback on previous trials, while using unit tests to evaluate the quality of exploration. To distill the exploration experience, SeDev incorporates a feedback synthesis module that translates unit test results within exploration into comprehensive suggestions. We construct a challenging software benchmark FSD-Bench++, along with two open datasets to evaluate. Experimental results show that SeDev outperforms baselines while maintaining reasonable time and computational costs. Code is available here.

1 Introduction

Large Language Models (LLMs) have emerged as powerful tools for code generation and have become increasingly valuable for human developers (OpenAI, 2022; Microsoft, 2023). While LLMs demonstrate strong performance in function-level code generation, they perform poorly in software-level development, where dependencies across multiple modules are often highly complicated (Wong et al., 2011). To address this limitation, recent studies have proposed several LLM-driven multi-agent frameworks that simulate human development workflows through task decomposition (Dong et al., 2024; Hong et al., 2024; Hu et al., 2024). Benefiting from the rapid generation and communication capabilities of LLM agents, these approaches are expected to tackle more complex software-level code generation tasks.

Despite these advances, LLM-driven approaches still suffer from a low success rates in generating correct software. The challenges are three-fold. First, the solution space is extremely large, involving diverse choices of programming languages, data structures, software architectures, and even function names (Li et al., 2023; Wang et al., 2024b). Second, exploration is constrained by limited trials, as modern LLMs often require tens of minutes to generate software consisting of hundreds of lines of code. Third, unlike human experts who can readily diagnose issues from test results, LLMs struggle to refine code due to insufficient prior knowledge for interpreting feedback. These challenges highlight the need for efficient exploration strategies that can approach high-quality solutions under a limited exploration budget.

Existing work has proposed various methods to guide LLM-based software generation. Early frameworks such as ChatDev (Qian et al., 2023) and MetaGPT (Hong et al., 2024) focus on designing agent collaboration and communication mechanisms between multiple agents to constrain generation trajectories. Another line of work emphasizes iterative refinement based on feedback signals. For example, CoCoGen aligns and fixes errors using repository information and compiler

* Equal contribution
† Corresponding author

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 35485–35512
July 2-7, 2026 ©2026 Association for Computational Linguistics

feedback (Bi et al., 2024), while EvoMAC introduces a self-evolution paradigm guided by unit test feedback (Hu et al., 2024). However, these approaches often lack explicit exploration mechanisms and can easily converge to local optima. More recent studies adopt heuristic search methods such as Monte Carlo Tree Search (MCTS) to improve exploration (Zhang et al., 2023; Li et al., 2024), but such search-based methods become increasingly inefficient as task complexity grows.

In this work, we propose SeDev, a novel LLM-driven code generation framework that particularly focuses on exploration over the vast solution space. Compared with existing iterative frameworks that implicitly explore the solution space by leveraging LLMs' intrinsic randomness, which tends to be either overly conservative or prone to excessive deviation, SeDev enhances the exploration process by designing an iterative exploration strategy that jointly considers structural prompting and semantic proximity. Unlike prior approaches with fixed instructions, SeDev treats instructions as exploration variables via meta-prompt—a structured task-agnostic template that can be instantiated into varied prompts to guide exploration. However, exploration with structured prompts is still limited by the very large solution space. To further improve the efficiency, SeDev incorporates a unit test module and a feedback synthesis module, which translate the raw unit test results into specific suggestions for the next round of exploration. For example, the feedback synthesis module may suggest to change the programming language to python if the task relates to machine learning. Intuitively, such feedback facilitates the evolution of the exploration process by suggesting the semantically adjacent paths to explore. As the exploration-feedback process repeats, the vast solution space will be gradually explored. Our contributions can be summarized as follows:

- We propose SeDev, a novel LLM-driven multi-agent code generation framework that explicitly focuses on efficient exploration over large solution spaces, enabling software generation within a small number of iterations.

- SeDev integrates structured prompt construction with a comprehensive feedback synthesis module, embedding software development logic and guiding exploration toward semantically adjacent solutions.

- We construct FSD-Bench++, an extended version of the FSD-bench, by adding more comprehensive and refined test cases to better evaluate the framework's code generation capability. Experiments on FSD-Bench++ and two public benchmarks demonstrate that SeDev consistently outperforms strong baselines.

2 Related Work

Direct Code Generation LLMs have achieved remarkable success in direct generation across many domains (Wei et al., 2022; Cahyawijaya et al., 2024; Kojima et al., 2022; Yuan et al., 2026, 2025a). Benefiting from massive pretraining corpora, LLMs have also demonstrated strong capabilities in code generation. Codex achieves a 28.8% success rate on programming problems, and its derivative Copilot has been widely adopted by developers (Chen et al., 2021a; Microsoft, 2023). Similar progress has been reported by Incoder (Fried et al., 2023), CodeRL (Lea et al., 2022), Code Llama (Roziere et al., 2023), and ChatGPT (OpenAI, 2022). However, these approaches mainly focus only on the code generation ability of a single LLM, and struggle with complex software-level tasks. To address this limitation, recent works propose multi-agent frameworks that emphasize collaboration among LLMs. For example, Self-Collaboration assigns different roles to LLMs to collaboratively solve subtasks (Dong et al., 2024), while MetaGPT introduces a standardized software development pipeline with multiple agents following a linear workflow (Hong et al., 2024). Nevertheless, these approaches largely follow a Waterfall-style process (Bassil, 2012), lacking mechanisms to refine intermediate results, which often leads to difficulties in error correction and low success rates.

Feedback-Enhanced Code Generation To overcome the limitations of direct generation, several works incorporate feedback to iteratively improve code quality. ChatDev introduces a test agent to verify executability (Qian et al., 2023), while CompCoder leverages compiler feedback to ensure compilability (Wang et al., 2022). AgentCoder further integrates iterative unit test and optimization in a multi-agent framework (Huang et al., 2023). EvoMAC proposes a self-evolving paradigm in which unit test feedback is used to refine the generation process, analogous to backpropagation in neural networks (Hu et al., 2024). In addition, recent studies employ Process Reward Models (PRMs)

during training to enhance reasoning ability, which also benefits code generation (Lightman et al., 2023; Dai et al., 2024; Wang et al., 2024a). Despite improving solution quality, these feedback-driven methods often converge to local optima due to the absence of explicit exploration strategies.

Exploratory Code Generation Exploratory code generation explicitly introduces diversity into the generation process to explore a broader solution space. MapCoder generates multiple code plans with confidence scores and selects final code based on ranked plans (Islam et al., 2024). StepCoder focuses on code completion by applying reinforcement learning to incrementally generate full programs (Dou et al., 2024), while PG-TD incorporates planning algorithms into Transformer decoding to guide program generation (Zhang et al., 2023). Another line of work adopts Monte Carlo Tree Search (MCTS) to search for better solutions, including SWE-Search (Antoniades et al., 2024), SRA-MCTS (Xu et al., 2024), and MCTS-SQL (Yuan et al., 2025b). For example, RethinkMCTS performs thought-level search before code generation to explore solutions (Li et al., 2024). Although MCTS-based methods explicitly explore the solution space, they often suffer from low efficiency due to extensive rollouts and backpropagation.

3 Methodology

Motivation: In real-world software development, human developers usually follow certain development methodologies with experience-guided exploration to accomplish tasks. Human developers may come up with some ideas, write the code, test it and repeat the process multiple times until finally solving the problem. Motivated by this, SeDev is designed to emulate experience-guided exploration and iterative, feedback driven development methodologies within a deliberate framework.

Overview: Generally, SeDev consists of three modules: a parallel exploration module, a unit test module and a feedback synthesis module, as shown in Figure 1. These modules run automatically and continually find better solutions to the given tasks.

3.1 Parallel Exploration Module

The framework is initialized with a Product Requirement Document (PRD), aims to generates executable software that satisfies the PRD. We adopt a step-by-step, requirement-driven workflow, where we first perform architectural analysis based on the requirements, then design the code plan, and finally generate the source code. The above process constitutes an exploration path, and is carried out by three agents. Detailed descriptions of each agent are as follows. The parallel exploration module can simultaneously initiate multiple paths, generating more solutions to improve exploration effiency.

Architect. The Architect agent evaluates the PRD to outline the software's overall architecture, technology stack, class design, and user interface. These design can be transformed into Unified Modeling Language diagrams using tools (Knut, 2014).

Project Manager. The Project Manager organizes a code plan (depicted as a list of files to generate) based on the PRD and the architecture. This agent also aligns each requirement in the PRD with the relevant files in the code plan to ensure all requirements are considered.

Programmer. The Programmer agent produces code with the architecture diagram and the code plan created by the preceding agents. Through task decomposition, the Programmer agent is able to tackle complex tasks, aided by scheduling during the development process.

3.2 Meta-prompt based Structured Prompting

We begin by briefly reviewing existing exploration paradigms to motivate our structured prompting design. Figure 2 compares three representative exploration strategies. Temperature-based exploration relies on the intrinsic randomness of LLMs to generate diverse outputs, controlled by a single temperature parameter. While simple to implement, such unguided randomness rarely yields high-quality solutions. Monte Carlo Tree Search (MCTS) explores multiple generation paths through simulation and backtracking. However, MCTS is often inefficient in semantic solution spaces due to the high dimensionality and sparse rewards of language and code generation. Moreover, MCTS relies heavily on execution-based feedback, limiting meaningful exploration before execution and resulting in substantial computational overhead.

SeDev performs exploration through a structured prompting paradigm. A prompt is decomposed into three core components: Input, Instruction, and Output. The Input provides task context, the Instruction specifies goals, methodologies, and constraints, and the Output defines the expected format of deliverables. Most existing approaches rely on fixed prompts tailored to specific tasks. As task complex-

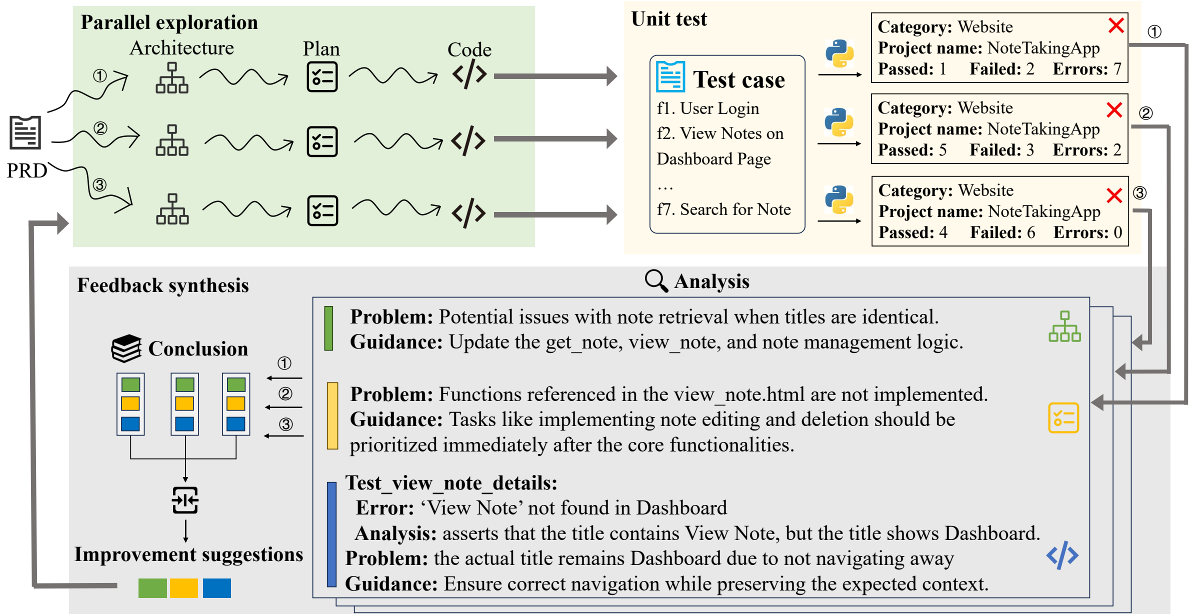


Figure 1: The overall framework of SeDev. At each iteration, the parallel exploration module generates multiple pieces of code, which will go through the unit test module independently. The feedback synthesis module then synthesizes the unit test results into specific improvement suggestions for each agent in the next round of exploration. The above process will repeat multiple times until a solution passes the unit test or the maximum number of iterations reached.

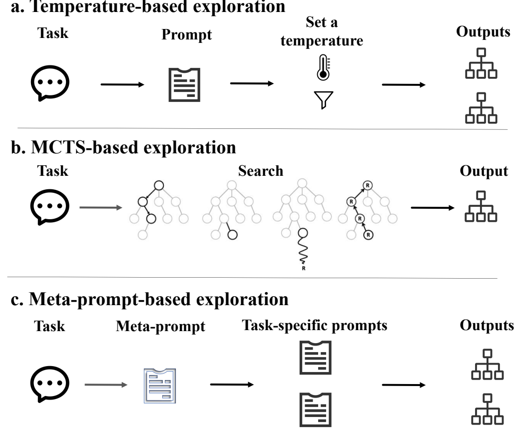


Figure 2: Comparison of three exploration paradigms.

ity increases, such simple and rigid prompt designs struggle to capture nuanced requirements, limiting their effectiveness for exploration.

To address this limitation, SeDev introduces a meta-prompt to construct structured prompt used for exploration. A meta-prompt begins as a task-agnostic template: the input and output fields are placeholders, while the instruction part initially contains only a descriptive definition—clarifying what constitutes a valid instruction, including its essential elements, rather than prescribing concrete goals, methodologies, or constraints. This design enables the meta-prompt to be later instantiated into task-specific prompts during exploration.

Upon receiving a real task, the IO placeholders of the meta-prompt are filled with the task's concrete input and output. Then, we prompt the LLM to generate task-specific prompts by transforming the instruction component to practical instruction based on the provided input and output. We increase the LLM's temperature during this stage to encourage diversity. Finally, the task-specific prompts are used to prompt the LLM to generate the desired outcome for the task. Compared to fixed prompts that remain unchanged across multiple generations, this meta-prompt–enabled structured prompting allows the instruction component to be explored and to serve as the core driver of the exploration process, enabling more effective exploration. Figure 3 presents a concrete example. The upper part shows the initialized meta-prompt after incorporating the input and output specifications for a python task, while the lower part shows the final prompt obtained by replacing the instruction component with a task-specific instruction.

Note that the meta-prompt itself is a general, task-agnostic template and can be applied at any stage of the framework. In our implementation, however, only the Architect and Project Manager agent leverage this structured prompting for exploration; the Programmer agent directly generates

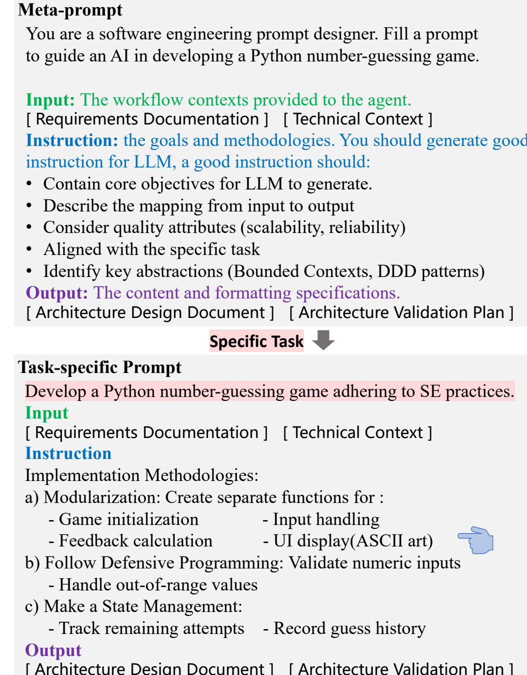


Figure 3: An example of meta-prompt and task-specific prompt. Meta-prompt consists of 3 components: input, instruction, and output. Input refers to the context provided to the agent; instruction specifies the problem objectives and methodologies; output defines the expected content and formatting of the deliverables.

code using a fixed prompt, considering that code generation is inherently a rigid process. More details about the diversity of generated prompts using meta-prompt can be found in Appendix 8.

3.3 Unit Test Module

After parallel exploration, the unit test module evaluates the generated code against the required software features specified in the PRD and provides feedback. Prior work has shown that LLMs can generate unit tests and testing code given task descriptions (Li and Yuan, 2024; Liu et al., 2024; Hu et al., 2024). Building on these insights, we develop an automatic unit testing pipeline.

Our unit test module consists of three steps. First, an LLM generates textual test cases from the PRD, aiming to cover software features at the semantic level. Second, the LLM combines the generated textual test cases with the source code to produce executable unit test code. Third, the test code is executed in the terminal to obtain test results, including pass/fail statistics and runtime errors.

During test code generation, we require the LLM to produce assert-based tests that verify whether key program variables exhibit expected transitions before and after test execution. A test is considered failed if the observed variable states deviate from expected outcomes or if an exception occurs during execution; otherwise, it is deemed successful.

3.4 Feedback Synthesis Module

The unit test results reveal defects in the generated code, but raw test outcomes are not instructive for LLM to refinement. To bridge this gap, we develop a feedback synthesis module that generates actionable improvement suggestions in two steps.

Analysis: For each exploration path, an LLM takes the generated code and its unit test results, systematically analyzes the outputs generated by the three agents described in Section 3.1, identifying which elements are associated with failing test cases and which contribute to successful cases. Analysis is repeated for every exploration path in the current exploration round.

Conclusion: Another LLM synthesizes analysis results from all paths into one piece of tailored, actionable suggestions for the three agents. The synthesis merges similar issues, filters infeasible recommendations, and retains unique suggestions, since diverse solutions may be valid for the same feature. These suggestions will be included in structured prompt to guide subsequent exploration.

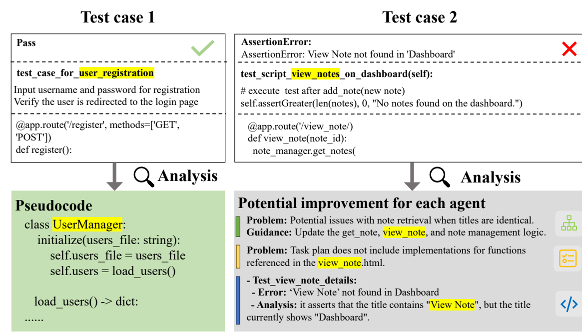


Figure 4: Different outcomes based on unit test results.

Figure 4 illustrates the *analysis* step for both successful and failed code generation cases. For successful code, the feedback synthesis module converts the implementation related to a test case into a pseudocode block, which serves as a reference for the next exploration round. Pseudocode preserves the core logic while reducing token usage. When multiple paths succeed for the same feature, only one pseudocode version is retained, as successful designs require minimal guidance. For failing code, the module examines the relevant components, identifies causes of errors, and generates improvement suggestions based on the unit

test results. Regarding the architecture and planning analysis, the LLM evaluates which elements are sound and worth retaining, and which require revision. Leveraging LLM reasoning, this process effectively translates raw test outcomes into actionable guidance. In the *conclusion* step, another LLM aggregates the analyses from all exploration paths into a unified summary, which includes customized improvement suggestions for each of the three agents. These suggestions are integrated with structured prompts in the next exploration round, enabling evolutionary exploration of semantically adjacent solutions and mimicking the reasoning processes of human expert developers.

4 FSD-Bench++

Recent studies have proposed various benchmarks to evaluate the software generation capability of LLMs. Some benchmarks, such as SRDD (Qian et al., 2023), SoftwareDev (Hong et al., 2024), and 50days50projects (Zhang et al., 2025), mainly rely on human judgment or LLM evaluation to assess the quality of generated software. In contrast, benchmarks such as FSD-Bench (Functionality-driven Software Development Benchmark) (Liu et al., 2025) and DevEval (Li et al., 2025) provide a more objective and rigorous functionality-driven evaluation by measuring whether the generated software can pass predefined unit tests.

FSD-Bench covers three types of applications, namely websites, desktop applications, and games. For each task, it provides three main components: a *user requirements* document, a *test cases* document, and the corresponding *unit testing* pipeline. The user requirements document specifies the software description, required functionalities, technology stack, and data storage, while the test cases document serves as the basis for functionality evaluation. Based on these materials, unit testing is conducted to assess whether the generated software correctly implements the required functionalities.

However, through our analysis, we find that although FSD-Bench provides a useful foundation for functionality-driven evaluation, its test cases remain relatively loose for some tasks and do not always sufficiently cover boundary-condition and corner-case scenarios. As a result, implementations that pass the original test suite may still fail in less typical but practically important settings, which can lead to an overestimation of functional completeness. To improve the reliability of functionality evaluation, we manually refine and extend FSD-Bench and introduce **FSD-Bench++**. In particular, we manually add new test cases to the original benchmark, with a focus on boundary conditions and corner-case scenarios that are insufficiently covered by the original test suite. Specifically, compared with the 1,195 test cases in FSD-Bench, FSD-Bench++ contains 1,451 test cases, including 256 newly added cases for more comprehensive functional verification. These additions help reduce potential misjudgment in evaluation and provide a more robust benchmark for functionality-oriented software development. More details can be found in Appendix A.3.

5 Experiments

5.1 Experimental Settings

Baselines. We adopt GPT-4o-Mini and Deepseek-V3 as backbone models to support seven open-source code generation methods, resulting in eight evaluated baselines. GPT-Engineer (Osika., 2023) is a single-agent method emphasizing robust understanding of task requirements. AutoGen (Wu et al., 2023) provides a framework for creating AI agent systems, facilitating the development of scalable agent applications. MetaGPT (Hong et al., 2024) assigns distinct roles to LLM agents and integrates standardized procedures to ensure multi-agent collaboration, with agents generating solutions following human-crafted instructions. ChatDev (Qian et al., 2023) uses a chat chain and de-hallucination techniques to guide autonomous, collaborative development. EvoMAC (Hu et al., 2024) leverages text-based feedback to iteratively refine generated code; we reproduce it with minor prompt adjustments due to lack of open-source release. MapCoder (Islam et al., 2024) employs four agents to emulate human program synthesis stages: recalling examples, planning, code generation, and debugging. GIF-MCTS (Dainese et al., 2024) applies MCTS to guide LLM code generation.

Datasets. We compare SeDev with eight baseline methods on three datasets. The first dataset is **HumanEval**, which comprises 164 Python function completion tasks (Chen et al., 2021b), and it has been widely used to evaluate the quality of generated code. The second dataset is **DevEval** (Li et al., 2025), which comprises 22 curated repositories spanning Python, C/C++ and Java, with diverse levels of complexity and multi-file project structures (Li et al., 2025). We focus on the Python

tasks from DevEval and assess the quality of the generated code using PyTest unit test tool.

FSD-Bench++: We develop Functionality-driven Software Development benchmark (FSD-Bench++) , which covers three main types of software: website, desktop application and game development. FSD-Bench++ contains 120 realistic tasks, with 882 pieces of functional descriptions and 1,451 test cases-averaging 12 test cases per task-to comprehensively verify if the developed software accomplishes the features. More details can be found in the Appendix A.3.

Metrics. Experiments on HumanEval adopt the **pass@1** metric following existing works (Chen et al., 2021b). For experiments on FSD-Bench++, the **executability** metric calculates the percentage of tasks whose code runs successfully in the compiling environment. To evaluate how the functionalities in the PRD are implemented, we use metric **functional completeness** (FC), which is calculated as the ratio of fulfilled requirements to the total number of requirements. Specifically, after generating the code, we use GPT-4o to produce unit test code based on the pre-defined textual test cases in the dataset. We manually inspected and validated a subset of the generated test code to verify LLM's accuracy. We then execute these unit tests and count the number of passed and failed test cases. Once the evaluation for all tasks in the dataset is completed, we obtain the total number of passed test cases and divide it by the total number of test cases. In addition, we report the total number of **tokens** and **time** consumed during the entire generation process. For **DevEval**, we similarly employ the **functional completeness** metric to assess performance of all methods.

Implementation Details. For the feedback-enhanced methods, we executed them 5 times of feedback-refinement. We configured MapCoder with 3 self-retrieved exemplars and allowed up to 5 debugging attempts, following the original settings in the paper. For GIF-MCTS, the number of expansion nodes was limited to 15, distributed evenly across its three action types. For SeDev, we set the number of parallel exploration paths per round to 3 and the total number of exploration rounds to 5. In each exploration round, we select the path that passes the most unit tests among all explored paths as the outcome of that round. The final output is selected from the round that passes the most unit tests. This enables our method to obtain the best solution through explorations.

5.2 Overall Performance of SeDev

As shown in Table 1, SeDev generally outperforms all baseline methods, which shows the superiority of our methods. Specifically, we can see that the feedback-enhanced methods such as EvoMAC perform better than direct generation methods. This reveals that feedback mechanisms indeed benefit the software code generation process. However, direct and feedback-enhanced methods generally perform worse than exploratory methods, which demonstrates the importance of exploration in software development tasks. Moreover, SeDev outperforms the search-based method GIF-MCTS. This justifies that our exploration strategy design is much more efficient than MCTS. MapCoder is another exploratory method where the generation process samples multiple plans first and executes them by their confidence scores. We can see that although MapCoder enjoys less costs, it performs much worse than SeDev. Therefore, we believe that SeDev raises a reasonable computational cost for achieving much better performance. More analysis on the computational cost of SeDev is provided in Section 5.4. As for the executability, SeDev achieves the best results, benefiting from exploring the wrong paths in the parallel exploration module and the improvement suggestions generated by the feedback synthesis module. In addition, we can note that EvoMAC achieves 94.51% on the HumanEval dataset, as we directly refer from its paper (Hu et al., 2024). This result shows that simpler tasks could be perfectly handled by an effective feedback mechanism even without exploration, but for more complex tasks in FSD-Bench++, the exploration mechanism is even more critical.

5.3 Convergence Analysis

In this section, we report the performance of SeDev as it varied with the number of the exploration rounds on FSD-Bench++. As shown in Figure 5, the FC score of SeDev generally increases with the number of exploration rounds. Notably, in the first three rounds, each leads to a relatively significant improvement in the FC score. By the fifth round, the FC score begins to converge, and subsequent explorations yield little to no further improvement in FC score. These results demonstrate that the evolutionary exploration strategy supported by structured prompt and feedback analysis can efficiently discover better solutions than recent strong baselines with only a few exploration rounds.

Model	Paradigm	Method	FSD-Bench++ FC (%) Website	Desktop	Game	Average	Exec.	DevEval FC (%)	HumanEval (%) Pass@1
GPT-4o-Mini	Direct	GPT-4o-Mini	36.85	37.21	33.98	36.01	70.00	23.08	87.20
		GPT-Engineer	24.28	39.01	29.13	36.76	55.83	5.13	88.41
		AutoGen	34.12	41.70	27.83	34.55	73.33	7.69	85.36
		MetaGPT	10.28	44.39	35.92	30.20	68.33	17.95	87.20
	Feedback-Enhanced	ChatDev	21.42	36.32	42.71	33.48	69.16	7.69	86.59
		EvoMAC	54.00	71.30	62.45	62.58	96.66	53.84	**94.51**
	Exploratory	MapCoder	38.28	39.46	37.54	38.42	80.00	17.95	90.85
		GIF-MCTS	47.71	31.39	21.68	33.59	73.33	15.38	-
		SeDev	**75.14**	**91.47**	**85.76**	**84.12**	**100**	**76.92**	90.85
			+39.14	**+28.28**	**+37.32**	**+34.42**	**+3.40**	**+42.87**	
DeepSeek-V3	Direct	Deepseek-V3	53.42	24.21	40.12	39.25	76.67	17.95	-
		AutoGen	33.14	26.90	40.45	33.50	65.83	10.25	-
		MetaGPT	23.42	30.04	40.45	31.30	60.00	15.38	-
	Feedback-Enhanced	ChatDev	18.00	24.66	43.36	28.67	58.33	12.82	-
		EvoMAC	77.42	74.88	67.63	73.31	92.50	64.10	-
	Exploratory	MapCoder	31.14	36.32	36.56	34.67	75.00	46.15	-
		GIF-MCTS	57.71	60.08	30.42	49.40	77.50	41.30	-
		SeDev	**80.57**	**75.33**	**69.90**	**75.27**	**100**	**71.79**	-
			+4.07	**+0.60**	**+3.36**	**+2.67**	**+8.10**	**+12.00**	

Table 1: Comparison between SeDev and baselines. The paradigm indicates different categories of methods explained in the related works. All the reported metrics are averaged across all tasks in the datasets. The best results are in **bold**. Values with '+' represent the relative improvement in percentage, compared with the second-best results. Note that GIF-MCTS can only run on dataset with unit test cases, therefore we did not test it on HumanEval.

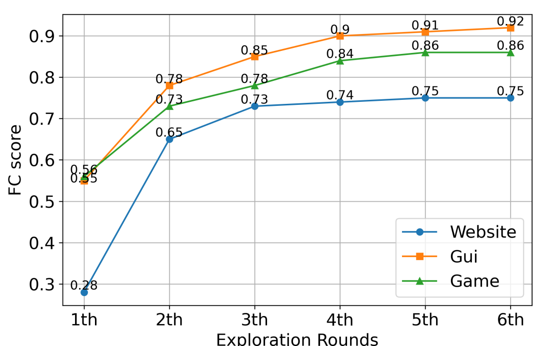


Figure 5: The FC score of each exploration round.

5.4 Computational Cost Analysis

Table 2 shows the time and token usage of all methods on FSD-Bench++ using GPT-4o-Mini. Compared to direct generation methods, exploratory and feedback-enhanced methods involve finer-grained steps to test and refinement, generally resulting in higher time and token usage. Consequently, direct generation methods underperform those with multiple iterations. SeDev shows relatively weak performance in the first round, as it generates deliverables per task without any feedback. As exploration round increases, the framework gradually explores the solution space more effectively, leading to improved performance. In the second round (R=2), SeDev already outperforms all baseline methods in terms of FC score, while maintaining comparable time and token consumption. Moreover, SeDev with only one iteration outperforms GIF-MCTS in terms of both FC score and computational costs. This result highlights the exploration efficiency of SeDev's exploration strategy. In addition, we can see that SeDev with two rounds of iterations significantly outperforms EvoMAC and MapCoder, both of which run five rounds of iterations, showing that the feedback synthesis module in SeDev can utilize the unit test feedback more efficiently. Overall, as the number of exploration rounds increases, the performance of SeDev consistently improves, at the cost of increased time and token consumption. In practice, this trade-off can be flexibly managed by adjusting the exploration budget.

5.5 Ablation Studies

To investigate the influence of structured prompting based on meta-prompt, we implement three variants of SeDev by removing the meta-prompt at different stages during parallel exploration and replacing it with a fixed prompt. The variants are: (A) Removing the meta-prompt in the architectural

Method	FC (%)	Tokens	Duration
GPT-4o-Mini	36.01	1953	14.80
GPT-Engineer	36.76	5919	23.04
AutoGen	34.55	6345	24.55
MetaGPT	30.20	44122	67.58
ChatDev	33.48	39318	389.11
EvoMAC	62.58	25132	451.96
MapCoder	38.42	35311	300.82
GIF-MCTS	33.59	11520	1245.82
SeDev R=1	46.23	41155	260.82
SeDev R=2	71.96	109518	685.27
SeDev R=3	78.43	184030	938.00
SeDev R=4	82.75	256307	1250.58
SeDev R=5	84.12	320360	1568.45
SeDev R=6	84.49	390162	1895.35

Table 2: Performance and computational cost comparisons between SeDev and baselines.

design. (B) Removing the meta-prompt in the code planning. (C) Removing the meta-prompt from both phases. We denote the original SeDev as variant (D). We randomly selected 15 tasks from the website tasks of FSD-Bench++ to conduct ablation study, and compared different variants to demonstrate the effectiveness of the meta-prompt in the exploration. Table 3 demonstrates that both (B) and (C) exhibit a decline, with a clear decrease of 4.65% and 1.55% in terms of the FC score metric. (A) shows the largest decrease in FC score, with a drop of 6.98%. These findings suggest that meta-prompt contributes meaningfully to the diversity and effectiveness of the exploration process, enabling more efficient exploration of the solution space and enabling faster search for the optimal solution. In contrast, fixed prompts tend to constrain the exploration process, resulting in highly similar outputs, thereby limiting the efficiency of exploration.

Variants	Arch	Plan	FC (%)
A	-	-	71.31
B	-	✓	73.64
C	✓	-	76.74
D	✓	✓	**78.29**

Table 3: Ablation study result. The meta-prompt during the exploration phase is alternately excluded.

6 Conclusions

LLM-driven methods have shown great potential in software development tasks. However, in practice, users need to make a great effort in prompt tuning in order to obtain the desired software. We introduce a novel LLM-driven multi-agent framework SeDev for automating the exploration process in software development, so that to improve the quality of solutions and reduce human labors. SeDev consists of three modules for exploration, which gradually explore the large solution space and evolve automatically. To evaluate SeDev, we construct the FSD-Bench++ that focus on evaluating code functional completeness. Benefiting from the automatic exploration, SeDev significantly outperforms eight baselines in the field of LLM-driven software development at a moderate computational cost.

Limitations

To support scalable evaluation, we employ LLMs to generate executable test code and run scripts for assessment, which may introduce evaluation bias. While SeDev employs a novel exploration strategy to explore semantically adjacent solutions, it still lacks an explicit and controllable mechanism to balance between exploration and exploitation. Although reinforcement learning(RL) methods are good at balancing exploration and exploitation, they often require thousands of iterations to optimize a policy, which is too costly for software-level code generation. In future works, we plan to explore efficient RL methods that accelerate the evolution of structured prompting. In the feedback synthesis module, although increasing the number of exploration paths can improve exploration efficiency, it also increases the burden for feedback module and cost. Future work will study the relationship between exploration rounds and task difficulty to design a more efficient exploration framework.

Acknowledgments

This work was partially supported by National Natural Science Foundation of China (62506133, 62476097, 62476121), the Guangdong Basic and Applied Basic Research Foundation(2025A1515010247), the Science and Technology Planning Project of Guangdong Province (2025B0101120003), the Guangdong Provincial Fund for Basic and Applied Basic Re-

search—Regional Joint Fund Project (Key Project) (2023B1515120078), Guangdong Provincial Natural Science Foundation for Outstanding Youth Team Project (2024B1515040010), the Fundamental Research Funds for the Central Universities, South China University of Technology (x2rjD2250190).

References

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. 2024. Swe-search: Enhancing software agents with monte carlo tree search and iterative refinement. *arXiv preprint arXiv:2410.20285*.

Youssef Bassil. 2012. A simulation model for the waterfall software development life cycle. *ArXiv*, abs/1205.6904.

Zhangqian Bi, Yao Wan, Zheng Wang, Hongyu Zhang, Batu Guan, Fangxin Lu, Zili Zhang, Yulei Sui, Hai Jin, and Xuanhua Shi. 2024. Iterative refinement of project-level code context for precise code generation with compiler feedback. In *Findings of the Association for Computational Linguistics*, pages 2336–2353.

Samuel Cahyawijaya, Holy Lovenia, and Pascale Fung. 2024. Llms are few-shot in-context low-resource language learners. *arXiv preprint arXiv:2403.16512*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, and Greg Brockman. 2021a. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021b. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Ning Dai, Zheng Wu, Renjie Zheng, Ziyun Wei, Wenlei Shi, Xing Jin, Guanlin Liu, Chen Dun, Liang Huang, and Lin Yan. 2024. Process supervision-guided policy optimization for code generation. *arXiv preprint arXiv:2410.17621*.

Nicola Dainese, Matteo Merler, Minttu Alakuijala, and Pekka Marttinen. 2024. Generating code world models with large language models guided by monte carlo tree search. *arXiv preprint arXiv:2405.15383*.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2024. Self-collaboration code generation via chatgpt. *ACM Transactions on Software Engineering and Methodology*.

Shihan Dou, Yan Liu, Haoxiang Jia, Enyu Zhou, Limao Xiong, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Tao Gui, and Xuanjing Huang. 2024. StepCoder: Improving code generation with reinforcement learning from compiler feedback. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, pages 4571–4585.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. Incoder: A generative model for code infilling and synthesis. In *International Conference on Learning Representations*.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*.

Yue Hu, Yuzhu Cai, Yaxin Du, Xinyu Zhu, Xiangrui Liu, Zijie Yu, Yuchen Hou, Shuo Tang, and Siheng Chen. 2024. Self-evolving multi-agent collaboration networks for software development. In *arXiv preprint arXiv:2410.16946*.

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

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. MapCoder: Multi-agent code generation for competitive problem solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, pages 4912–4944.

Sveidqvist Knut. 2014. Mermaid: Generate diagrams from markdown-like text.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.

Hung Lea, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 21314–21328.

Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, Zhiyin Yu, He Du, Ping Yang, Dahua Lin, Chao Peng, and Kai Chen. 2025. Prompting large language models to tackle the full software development lifecycle: A case study. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 7511–7531, Abu Dhabi, UAE. Association for Computational Linguistics.

Jiaqi Li, Mengmeng Wang, Zilong Zheng, and Muhan Zhang. 2023. Loogle: Can long-context language models understand long contexts? *arXiv preprint arXiv:2311.04939*.

Kefan Li and Yuan Yuan. 2024. Large language models as test case generators: Performance evaluation and enhancement. *arXiv preprint arXiv:2404.13340*.

Qingyao Li, Wei Xia, Kounianhua Du, Xinyi Dai, Ruiming Tang, Yasheng Wang, Yong Yu, and Weinan Zhang. 2024. Rethinkmcts: Refining erroneous thoughts in monte carlo tree search for code generation. *arXiv preprint arXiv:2409.09584*.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step. *arXiv preprint arXiv:2305.20050*.

Jie Liu, Guohua Wang, Ronghui Yang, Jiajie Zeng, Mengchen Zhao, and Yi Cai. 2025. RTADev: Intention aligned multi-agent framework for software development. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 1548–1581, Vienna, Austria. Association for Computational Linguistics.

Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2024. Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Microsoft. 2023. Microsoft attracting users to its code-writing generative ai software. *https://www.euronews.com/next/2023/01/25/microsoft-results-ai*.

OpenAI. 2022. Chatgpt: Optimizing language models for dialogue. *https://openai.com/index/chatgpt/*.

Anton Osika. 2023. Gpt-engineer.

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

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, and Jer´ emy Rapin. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. 2024a. Math-shepherd: Verify and reinforce llms step-by-step without human annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9426–9439.

Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. 2022. Compilable neural code generation with compiler feedback. In *Findings of the Association for Computational Linguistics*, pages 9–19.

Xindi Wang, Mahsa Salmani, Parsa Omidi, Xiangyu Ren, Mehdi Rezagholizadeh, and Armaghan Eshaghi. 2024b. Beyond the limits: A survey of techniques to extend the context length in large language models. *arXiv preprint arXiv:2402.02244*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Sunny Wong, Yuanfang Cai, Miryung Kim, and Michael Dalton. 2011. Detecting software modularity violations. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 411–420.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*.

Bin Xu, Yiguan Lin, Yinghao Li, and 1 others. 2024. Sra-mcts: Self-driven reasoning aurmentation with monte carlo tree search for enhanced code generation. *arXiv preprint arXiv:2411.11053*.

Li Yuan, Yi Cai, Xudong Shen, Qing Li, Qingbao Huang, Zikun Deng, and Tao Wang. 2025a. Collaborative multi-lora experts with achievement-based multi-tasks loss for unified multimodal information extraction. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, pages 6940–6948.

Li Yuan, Qingfei Huang, Bingshan Zhu, Yi Cai, Qingbao Huang, Changmeng Zheng, Zikun Deng, and Tao Wang. 2026. Hybrid-dmkg: A hybrid reasoning framework over dynamic multimodal knowledge graphs for multimodal multihop qa with knowledge editing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 28032–28040.

Shuozhi Yuan, Liming Chen, Miaomiao Yuan, Jin Zhao, Haoran Peng, and Wenming Guo. 2025b. Mcts-sql: An effective framework for text-to-sql with monte carlo tree search. *arXiv preprint arXiv:2501.16607*.

Sai Zhang, Zhenchang Xing, Ronghui Guo, Fangzhou Xu, Lei Chen, Zhaoyuan Zhang, Xiaowang Zhang, Zhiyong Feng, and Zhiqiang Zhuang. 2025. Empowering agile-based generative software development through human-ai teamwork. 34(6).

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. 2023.

Planning with large language models for code generation. In *International Conference on Learning Representations*.

A Appendixes

A.1 Auto Unit Test

In section 3.3, We design an automated unit test procedure. However, some datasets already include unit tests for their PRDs. For datasets with predefined test cases (e.g., DevEval), executing the original test scripts often requires strict adherence to specific method or class names. As our framework does not enforce such constraints, we instead extract task descriptions from the provided tests and manually integrate them into the PRD. We then regenerate textual test cases and test code using our pipeline, this allows us to maintain semantic equivalence with the original datasets while ensuring compatibility with our testing workflow.

During the experiments, we did not simply use the LLM as a judge to directly evaluate the generated code. Instead, the LLM was employed to simulate a testing engineer: given the existing software code and the textual test cases, it generated test scripts for the code. The final evaluation was still based on the objective outcomes of executing these unit tests. Previous work (e.g., EvoMAC) has demonstrated the capability of LLMs to write test code, which generally imposes lower requirements on the LLM compared to directly generating functional code. The purpose of LLM-generated test scripts is to produce an automated testing workflow; for software with GUIs, this primarily involves automating interactions such as click events, which closely approximate manual human operations. Before conducting the experiments, we randomly selected a few task and manually inspected the generated test scripts to ensure they aligned with human intuition. Although there are distance from precise automated test, we consider the LLM-generated test code a valid automated evaluation method to quickly verify whether the generated software implements the intended functionalities.

A.2 Implementation Detail

For all baselines, we set the temperature to 0.2 and top-p to 1 for both GPT-4o-mini and Deepseek-V3 during inference. Since SeDev involves exploration during its structured prompt construction phase, we set the temperature to 0.4 for model inference in this specific phase; for all other generation phases, SeDev uses the same settings as the baselines.

A.3 Benchmark Details

As introduced in Section 4, **FSD-Bench++** is an enhanced version of FSD-Bench with manually refined and extended test cases for more reliable functionality-oriented evaluation. It covers three types of software development tasks: websites, desktop applications, and games. In total, the benchmark contains 120 tasks and 1,451 test cases. Compared with the 1,195 test cases in the original FSD-Bench, FSD-Bench++ includes 256 newly added cases, mainly designed to improve the coverage of boundary-condition and corner-case scenarios.

The relationship among tasks, functional descriptions, and test cases is as follows: each task, which typically corresponds to a software specification, contains multiple functionalities, while test cases are used to verify whether the generated software correctly implements these functionalities. Since a single functionality may involve multiple execution steps, multiple test cases may be required to comprehensively validate it. As a result, the benchmark contains 882 functional descriptions in total, with each functionality paired with one or more corresponding test cases.

Our benchmark consists of the following three parts:

Part 1: User Requirements. Same as FSD-Bench, each task in FSD-bench++ is associated with a user requirement document consisting of four key components: *Software Description*, *Core Features*, *Programming Language (Technology Stack)*, and *Data Storage*. The *Software Description* provides an overview of the task and describes the expected software. The *Core Features* section lists the functionalities that the software is required to implement in the form of sub-points. The *Programming Language (Technology Stack)* specifies the high-level technical setting of the task, indicating whether the target software is a website, desktop application, or game. Finally, the *Data Storage* section briefly describes how the software data should be stored. To facilitate batch automated testing, local files are adopted as a practical compromise.

Part 2: Test Cases. In FSD-bench++, each user requirement document is paired with a corresponding test case document. For each functionality, we provide one or more test cases in a *Step-Expected* format, where *Step* specifies the sequence of operations and *Expected* specifies the expected result of those operations. Building on the original FSD-

Bench, we manually refine and extend the test suite, with particular emphasis on scenarios that were previously under-covered, especially boundary conditions and corner cases. These refinements improve the completeness and reliability of functionality evaluation. We provide examples to illustrate additional cases introduced in FSD-bench++ as shown in Figure 6 and 7.

Part 3: Unit Testing. After the code generation task is completed, the generated codebase and the corresponding test cases are provided to an LLM (GPT-4o) to generate unit test code. We then manually review and revise the generated unit tests to correct potential errors. The unit tests are generated on a feature-by-feature basis, and multiple test cases belonging to the same functionality are grouped into the same unit test function. If a functionality is not implemented by the generated software, the corresponding unit test function is written to directly return failure. Finally, all unit tests are executed in batch to evaluate the completeness of the software with respect to its required core functionalities.

case1

 # TestCases_NotepadPlus (FSD-bench)
Functionality 6. Offer Customizable Themes
Step: Open Notepad Plus application.
Step: Navigate to the settings menu to change the theme.
Expectation: The application interface changes to the selected dark theme.

 # TestCases_NotepadPlus (FSD-bench++)
Functionality 6. Offer Customizable Themes
Step: Open Notepad Plus application.
Step: Navigate to the settings menu to change the theme.
Step: Select a dark theme from the available options.
Expectation: The application interface changes to the selected dark theme.
Step: Change the theme back to the default light theme.
Expectation: The application interface reverts to the default light theme.

Figure 6: Example1 of additional test cases introduced in FSD-Bench++.

case2

 # TestCases_CultuaralCalender (FSD-bench)
Functionality 9. Local Data Storage
Step: Add a new cultural event to the local storage.
Step: Refresh the Dashboard Page.
Expectation: The newly added event appears in the list of upcoming cultural events.

 # TestCases_ CultuaralCalender (FSD-bench++)
Functionality 9. Local Data Storage
Step: Add a new cultural event to the local storage.
Step: Refresh the Dashboard Page.
Expectation: The newly added event appears in the list of upcoming cultural events.
Step: Remove an event from the local storage.
Step: Refresh the Dashboard Page.
Expectation: The removed event no longer appears in the list of upcoming cultural events.

Figure 7: Example2 of additional test cases introduced in FSD-Bench++.

A.4 Prompt cosine similarity

We report the cosine similarities among multiple prompts generated by the meta-prompt template in this paper, covering both the architectural design and code planning stages. As shown in the Figure 8, across all categories in the FSD-bench++ dataset, the cosine similarities of the prompts generated from the meta-prompt template remain at a reasonable level in both the architecture design and code planning phases. On the one hand, the prompts exhibit noticeable diversity overall, and in the code planning stage, the most differentiated prompt pairs even show a similarity of only about 66%. On the other hand, the prompt contents are not excessively random, since all cosine similarities remain above 50%, indicating that they consistently preserve the same overall task objective.

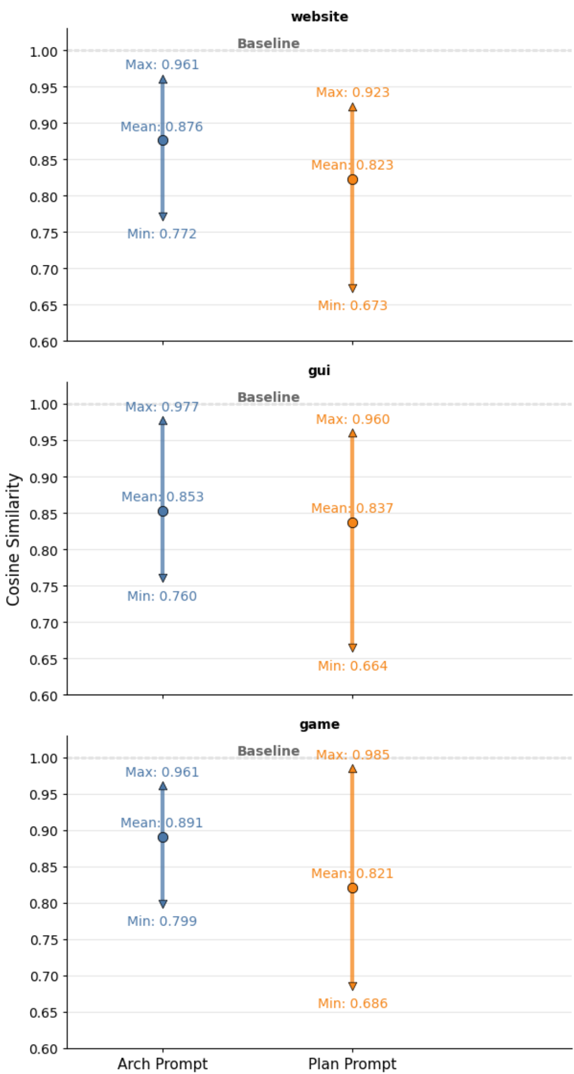


Figure 8: The cosine similarity of generated prompts using meta-prompt.

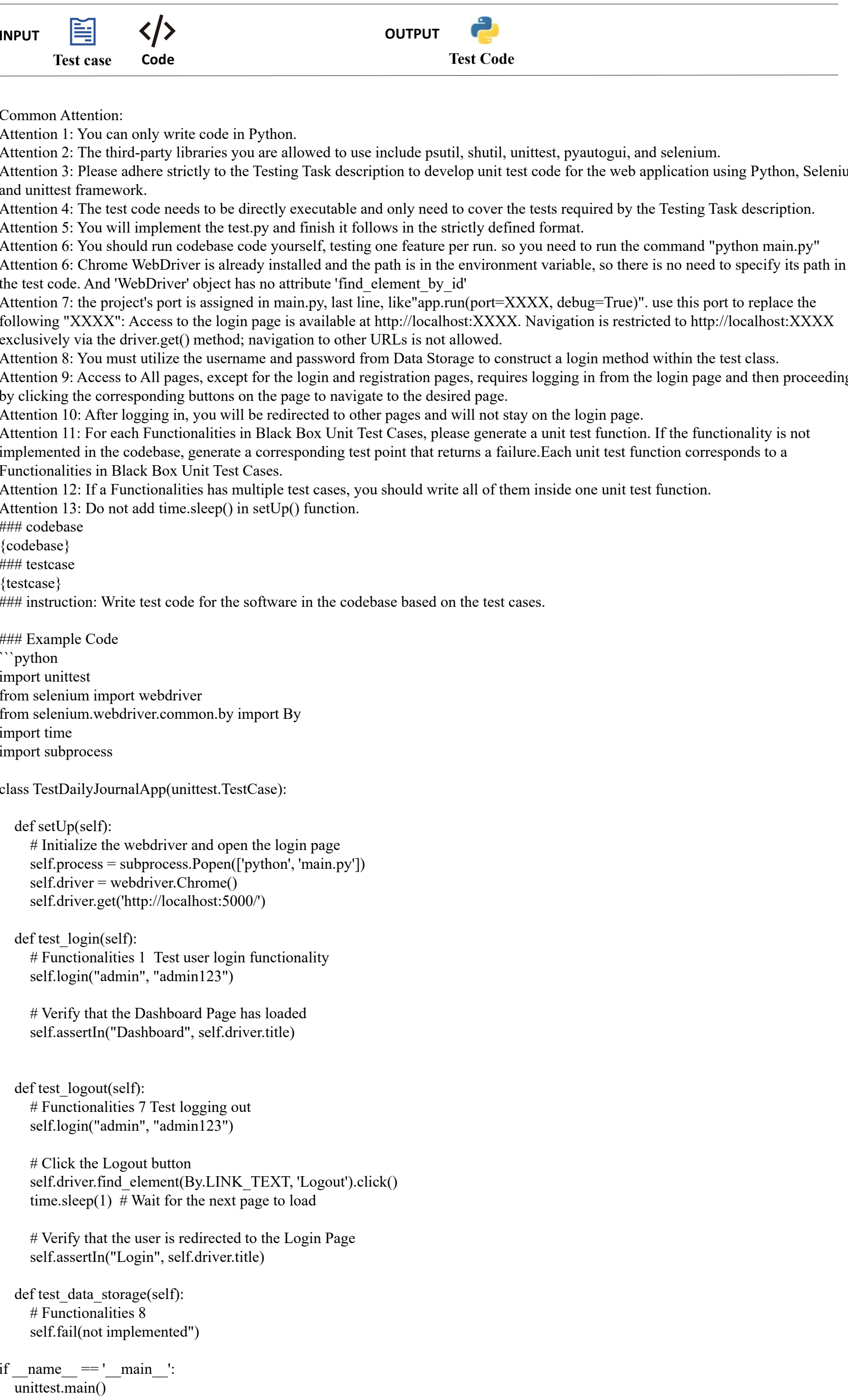
A.5 Unit Test Procedure

PROMPT FOR TEST CASE GENERATION

INPUT Requirement OUTPUT Test case

```
{software_description}
# Instruction
Based on the software description given above, write test cases(1 or 2 test cases) for each of the core features.
In this case, multiple test cases for the same feature should be written together.
----------------------------------
# Example Output
### Black Box Unit Test Cases

#### Funtionality 1. User Login
-        **Step**: Navigate to Login Page.
-        **Step**: Enter a valid username and password.
   **Expectation**: Access is granted, and the user is redirected to the Dashboard Page.

#### Funtionality 2. Navigate to Registration Page and User Registration
-        **Step**: Click on the "Register here" link on the Login Page.
   **Expectation**: The user is redirected to the Registration Page.
-        **Step**: Fill in valid username and password on the Registration Page.
   **Expectation**: The user is registered successfully, and a confirmation message is displayed.

#### Funtionality 3. View Charities on the Dashboard Page
-        **Step**: Login successfully and navigate to the Dashboard Page.
   **Expectation**: A list of available charities is displayed.
-        **Step**: Refresh the Dashboard Page after adding a new charity in the local storage.
   **Expectation**: The newly added charity appears in the charity list.

#### Funtionality 4. Navigate to Charity Details Page
-        **Step**: Click the 'Details' button for a specific charity on the Dashboard Page.
   **Expectation**: The Charity Details Page for that charity is displayed, showing detailed information.

#### Funtionality 5. View Contribution History
-        **Step**: Login successfully and navigate to the Dashboard Page.
   **Expectation**: The user's contribution history is displayed correctly.

#### Funtionality 6. Donate to a Charity
-        **Step**: Navigate to the Charity Details Page for a specific charity.
-        **Step**: Enter a valid donation amount and click the donate button.
   **Expectation**: The donation is processed successfully, and a confirmation message is displayed.

#### Funtionality 7. View Detailed Information
-        **Step**: Click the 'Details' button for a specific charity on the Dashboard Page.
-        **Step**: Select a specific charity by clicking on a charity's 'Details' buttion.
   **Expectation**: The specific charity's details for that charity is displayed, showing detailed information.
```

Figure 9: Prompt for generating test case based on PRD.



PROMPT FOR TEST CODE GENERATION (WEBSITE)

Common Attention:
Attention 1: You can only write code in Python.
Attention 2: The third-party libraries you are allowed to use include psutil, shutil, unittest, pyautogui, and selenium.
Attention 3: Please adhere strictly to the Testing Task description to develop unit test code for the web application using Python, Selenium, and unittest framework.
Attention 4: The test code needs to be directly executable and only need to cover the tests required by the Testing Task description.
Attention 5: You will implement the test.py and finish it follows in the strictly defined format.
Attention 6: You should run codebase code yourself, testing one feature per run. so you need to run the command "python main.py"
Attention 6: Chrome WebDriver is already installed and the path is in the environment variable, so there is no need to specify its path in the test code. And 'WebDriver' object has no attribute 'find_element_by_id'
Attention 7: the project's port is assigned in main.py, last line, like"app.run(port=XXXX, debug=True)". use this port to replace the following "XXXX": Access to the login page is available at http://localhost:XXXX. Navigation is restricted to http://localhost:XXXX exclusively via the driver.get() method; navigation to other URLs is not allowed.
Attention 8: You must utilize the username and password from Data Storage to construct a login method within the test class.
Attention 9: Access to All pages, except for the login and registration pages, requires logging in from the login page and then proceeding by clicking the corresponding buttons on the page to navigate to the desired page.
Attention 10: After logging in, you will be redirected to other pages and will not stay on the login page.
Attention 11: For each Functionalities in Black Box Unit Test Cases, please generate a unit test function. If the functionality is not implemented in the codebase, generate a corresponding test point that returns a failure.Each unit test function corresponds to a Functionalities in Black Box Unit Test Cases.
Attention 12: If a Functionalities has multiple test cases, you should write all of them inside one unit test function.
Attention 13: Do not add time.sleep() in setUp() function.
codebase
{codebase}
testcase
{testcase}
instruction: Write test code for the software in the codebase based on the test cases.

Example Code
```python
import unittest
from selenium import webdriver
from selenium.webdriver.common.by import By
import time
import subprocess

class TestDailyJournalApp(unittest.TestCase):

    def setUp(self):
        # Initialize the webdriver and open the login page
        self.process = subprocess.Popen(['python', 'main.py'])
        self.driver = webdriver.Chrome()
        self.driver.get('http://localhost:5000/')

    def test_login(self):
        # Functionalities 1  Test user login functionality
        self.login("admin", "admin123")

        # Verify that the Dashboard Page has loaded
        self.assertIn("Dashboard", self.driver.title)


    def test_logout(self):
        # Functionalities 7 Test logging out
        self.login("admin", "admin123")

        # Click the Logout button
        self.driver.find_element(By.LINK_TEXT, 'Logout').click()
        time.sleep(1)  # Wait for the next page to load

        # Verify that the user is redirected to the Login Page
        self.assertIn("Login", self.driver.title)

    def test_data_storage(self):
        # Functionalities 8
        self.fail(not implemented")

if __name__ == '__main__':
    unittest.main()
```

Figure 10: Prompt for generating test code for the website project.

PROMPT FOR TEST CODE GENERATION (DESKTOP)

INPUT Test case Code **OUTPUT** Test Code

Common Attention:
Attention 1: You can only write code in Python.
Attention 2: The third-party libraries you are allowed to use include psutil, shutil, unittest.
Attention 3: Please adhere strictly to the Testing Task description to develop unit test code for the web application, gui application using Python, Selenium, pywinauto, pyautogu, and unittest framework.
Attention 4: The test code needs to be directly executable and only need to cover the tests required by the Testing Task description.
Attention 5: You will implement the test.py and finish it follows in the strictly defined format.
Attention 6: You should run codebase code yourself, testing one feature per run. so you need to run the command "python main.py"
Attention 7: For each Functionalities in Black Box Unit Test Cases, please generate a unit test function. If the functionality is not implemented in the codebase, generate a corresponding test point that returns a failure.Each unit test function corresponds to a Functionalities in Black Box Unit Test Cases.
Attention 9: If a Functionalities has multiple test cases, you should write all of them inside one unit test function.
codebase
{codebase}
testcase
{testcase}
instruction: Write test code for the software in the codebase based on the test cases.

Example Codes

```
import unittest
from calculator import Calculator

class TestCalculator(unittest.TestCase):

    def setUp(self):
        self.calculator = Calculator()


    def test_addition(self):
        # Functionalities 1: Perform Basic Arithmetic Operations
        self.assertEqual(self.calculator.add(5, 3), 8)
        self.assertEqual(self.calculator.subtract(7, -2), 9)
        self.assertEqual(self.calculator.multiply(4, 6), 24)
        self.assertEqual(self.calculator.divide(8, 2), 4)
        with self.assertRaises(ValueError):
            self.calculator.divide(10, 0)


    def test_square_root_positive(self):
        # Functionalities 2: Calculate Square Roots
        self.assertEqual(self.calculator.square_root(16), 4)
        self.assertEqual(self.calculator.square_root(0), 0)
        with self.assertRaises(ValueError):
            self.calculator.square_root(-9)

    def test_exponentiation_positive(self):
        # Functionalities 3: Perform Exponentiation Calculations
        self.assertEqual(self.calculator.exponentiate(2, 3), 8)
        self.assertEqual(self.calculator.exponentiate(0, 5), 0)
        self.assertEqual(self.calculator.exponentiate(7, 0), 1)
        self.assertEqual(self.calculator.exponentiate(-3, 2), 9)


    def test_percentage_positive(self):
        # Functionalities 4: Calculate Percentages
        self.fail("not implemented")

if __name__ == '__main__':
    unittest.main()
```

Figure 11: Prompt for generating test code for the desktop project.

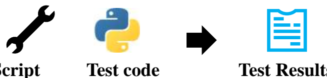
PROMPT FOR TEST CODE GENERATION (GAME)

INPUT Test case Code OUTPUT Test Code

Common Attention:
Attention 1: You can only write code in Python.
Attention 2: The third-party libraries you are allowed to use include psutil, shutil, unittest, pyautogui, and selenium.
Attention 3: Please adhere strictly to the Testing Task description to develop unit test code for the pygame using Python and unittest framework.
Attention 4: The test code needs to be directly executable and only need to cover the tests required by the Testing Task description.
Attention 5: You will implement the test.py and finish it follows in the strictly defined format.
Attention 6: You should run codebase code yourself, testing one feature per run. so you need to run the command "python main.py"
Attention 7: For each Functionalities in Black Box Unit Test Cases, please generate a unit test function. If the functionality is not implemented in the codebase, generate a corresponding test point that returns a failure.Each unit test function corresponds to a Functionalities in Black Box Unit Test Cases.
Attention 8: If a Functionalities has multiple test cases, you should write all of them inside one unit test function.
codebase
{codebase}
testcase
{testcase}
instruction: Write test code for the software in the codebase based on the test cases.

Example Code
```python
import unittest
import pygame
from game import Game
from paddle import Paddle
from ball import Ball
from brick import Brick

class TestBrickBreakerGame(unittest.TestCase):

    def setUp(self):
        # Initialize the game and its components
        self.game = Game()
        self.paddle = self.game.paddle
        self.ball = self.game.ball
        self.bricks = self.game.bricks

    def test_control_paddle_movement(self):
        # Functionalities 1 Test paddle movement to the left
        initial_x = self.paddle.x
        self.paddle.move('left')
        self.assertLess(self.paddle.x, initial_x, "Paddle should move left")
        # Test paddle movement to the right
        initial_x = self.paddle.x
        self.paddle.move('right')
        self.assertGreater(self.paddle.x, initial_x, "Paddle should move right")


    def test_brick_disappearance(self):
        # Functionalities 3 Hit a brick until it disappears
        brick = self.bricks[0]
        initial_life = brick.life
        for _ in range(initial_life):
            brick.hit()
        self.assertEqual(brick.life, 0, "Brick should disappear when life reaches 0")

    def test_game_start_mechanism(self):
        # Functionalities 4 Simulate starting the game by moving the paddle
        self.game.handle_input()
        self.assertTrue(self.game.running, "Game should be running after starting")


    def test_load_game_state(self):
        # Functionalities 9 Test loading game state (not implemented in codebase)
        self.fail("Load game state functionality is not implemented in the codebase")

if __name__ == '__main__':
    unittest.main()
```

Figure 12: Prompt for generating test code for the game project.




```
def runUnitTest(project_path, category):
    path = Path(project_path)
    project_name = path.name
    project_category = category

    print(f"----------------[START {project_name}]---------------------")

    project_path = os.path.join(project_path, "code")
    os.chdir(project_path)
    print(f"CURRENT DIR1 {project_path}")

    clear_imports()

    loader = unittest.TestLoader()
    suite = unittest.TestSuite()

    suite.addTests(loader.discover("."))

    # Create a StringIO stream to capture both stdout and stderr
    output_stream = io.StringIO()
    error_stream = io.StringIO()

    # Redirect stderr to capture errors
    sys.stderr = error_stream

    # Create a TextTestRunner instance to run the tests and capture the output
    runner = unittest.TextTestRunner(stream=output_stream, verbosity=2)
    result = runner.run(suite)

    # Get the captured output and errors
    test_output = output_stream.getvalue()
    error_output = error_stream.getvalue()

    if project_category == "website":
        # Filter out some unimportant and repetitive output
        print("#### strip ####")
        test_output = web_text_strip(test_output)
        print(test_output)

    total = int(result.testsRun)
    passed = int(result.testsRun - len(result.failures) - len(result.errors))
    failed = len(result.failures)
    errors = len(result.errors)

    # Combine both stdout and stderr outputs
    combined_output = test_output + "\n" + error_output

    info = {
        "category": project_category,
        "project_name": project_name,
        "passed": passed,
        "failed": failed,
        "errors": errors,
        "total": total,
        "output": combined_output,
    }
```

Figure 13: Python test script for test execution during exploration.

A.6 Prompt

We illustrate how structured prompt facilitates exploration.
Specifically, we consider a task in which the goal is to **design the complete file architecture** of a software system based on a given requirement.

[1] Fixed prompt:
Input: PRD
Instruction: You are required to design the file structure of a software system based on the requirements document, including the entry file, functional modules, and graphical resources.
Output: File structure.

In general, a fixed prompt must account for the generality across all tasks, and therefore tends to be more generic and less specific.

[2] Meta prompt for structured prompt
Input: PRD
Instruction:
You are required to supplement the instruction component.
(1) A well-designed instruction should precisely characterize the mapping from input to output, enabling the LLM to follow it clearly and generate the correct output.
(2) The instruction should be closely aligned with the specific task rather than being vague or overly generic, since the concrete task has already been specified and the instruction should account for the characteristics of the input and the requirements of the output.
(3) Below is additional information for this task. Your instruction is intended to generate the following component: File Structure. Specifically, this refers to the organization of software files, based on which the LLM agent will generate code and ultimately construct the system.
Output: File structure

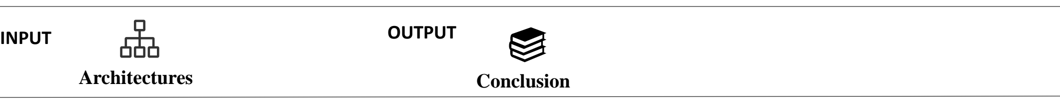
Three Generated Instructions by meta prompt:
[1] Generate a file list including a main entry file, UI files, and functional files.
[2] Generate a hierarchical file list in a top-down structure: the main entry file connects to subcomponents, and each file is responsible for a specific part of the requirements, including both UI and functionality.
[2] Generate a modular file list: each feature is separated into a module, and within each module there are UI files, functional files, and a module-specific entry file.

Corresponding file list:
Generated by fixed prompt:
[1] main.py, ui.py, function.py

Generated by structured prompt:
[1] main.py, ui.py, function.py (default, same as fixed prompt)
[2] main.py, part1.py, part2.py (This approach separates the UI design and functionality into individual files, rather than combining them into a single main file)
[3] entry.py | module1/function-1.py, ui-1.py | module2/part.py (This design follows a modular approach, with components coupled within each module but decoupled across modules)

Figure 14: A metaprompt template.

PROMPT FOR ARCHITECTURE FEEDBACK MERGE



Role Definition

You will act as a feedback summarization assistant. Your goal is to analyze multiple sets of architecture-related feedback for a software development project and produce a concise, unified summary.

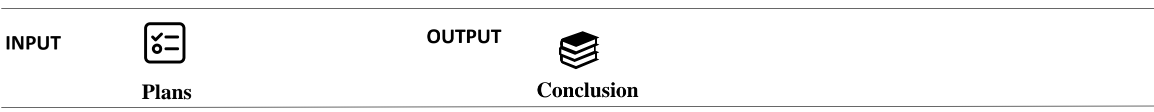
Instructions:
1. Combine Similar Feedback:Identify and merge duplicate or overlapping suggestions while retaining their key points. For example, if multiple feedback items suggest improving navigation in the UI, consolidate them into a single suggestion.
2. Retain Unique Feedback:Preserve suggestions that address distinct issues, even if they apply to different aspects of the project. Ensure no unique feedback is omitted.
3.Structure the Output:
Overall Evaluation
Specific Problems and Suggestions
Architecture Enhancements

Attention:
Use bullet points for readability, and provide actionable suggestions where applicable.
Ensure Clarity and Precision.
Use concise language to convey the ideas clearly and avoid redundancy.
Remove any password bcrypt feedback in the final summary.

The content you need to summarize is:{Architetures}.

Figure 15: Prompt for synthesizing all feedback related to architecture.

PROMPT FOR PLAN FEEDBACK MERGE



Role Definition

You will act as a feedback summarization assistant. Your goal is to analyze multiple sets of task-related feedback for a software development project and produce a concise, unified summary. You will receive multiple feedback reports, each containing various suggestions, including areas for improvement and potential enhancements. Your task is to extract suggestions that are useful for generating new plans and meet the following requirements:

1.Categorize Suggestions: Group the suggestions into the following categories:
Specific Areas for Improvement
Suggested Enhancements
2.Merge Similar Suggestions: Combine identical or highly similar suggestions into a single statement, using clear and concise language.
3.Retain Unique Suggestions: Keep unique suggestions that appear in only one feedback report but are valuable for improvement. Highlight their source where applicable.
4.Organized Output: Structure the output clearly and logically by category and priority (if mentioned), making it easy for planners to incorporate into new plans.

Output example:
Specific Areas for Improvement:
- Add logout functionality and error reporting for failed login/registration attempts.
- Clarify task descriptions for edge cases, including invalid input, duplicate registrations, and empty feedback submissions.
- Break down complex tasks (e.g., FeedbackManager implementation) into subtasks focusing on validation and file handling.

Suggested Enhancements:
- Prioritize user authentication tasks, followed by feedback submission and navigation.
- Specify expected behaviors after user actions, such as feedback confirmation messages.
- Implement basic form validations to prevent invalid or empty submissions.
Use this format to summarize the feedback provided, ensuring the suggestions are actionable for creating improved new plans.
Remove any password bcrypt feedback in the final summary.

The content you need to summarize is:{Plans}.
Follow the example, output you summary.

Figure 16: Prompt for synthesizing all feedback related to plan.

PROMPT FOR CODE FEEDBACK MERGE

INPUT

Unit Test Results

OUTPUT

Summarized Results

PROMPT FOR SUMMARY MERGE

instruction
I have multiple implementations of the same project, each of which has undergone unit testing. For each implementation, I have obtained test results, analyzed them, and developed improvement recommendations. Now, you should extract and compile all my contents with the following requirements:

1.Summarize all test cases: identify how many test_XXX_XXX (like this format) test cases exist in all my result, may not need to outptut.

2.Prepare the output, divided into two parts: Passed Test Cases and Failed or Error Test Cases, with following description:
Passed Test Cases
Summarize solutions for all test cases that passed. Use the pseudocode provided in the input to represent the solutions; do not generate new pseudocode. Present each case in the format:
1. |Case|:**Case Name**
Followed by the pseudocode which represent the successful implementation for this function.

Failed or Error Test Cases
Collect the analysis and guidance related to each failure or error. present it in the format:
For each error or failure, extract all related analyses and guidance from allthe content. If there are duplicates, please remove them.
Then, combine them and output them in the following format:
1. |Case|:**Case Name**
Followed by:
Failure/Error Analysis1
Improvement Guidance1 (textual, pseudocode, etc.)
Failure/Error Analysis2
Improvement Guidance2 (textual, pseudocode, etc.)
Each pair above represents content extracted from different projects (after deduplication).
Ensure that if there are differing analyses or guidance for a single test case, all are recorded.

Notes:
For ### Passed Test Cases, you need to "summarize"; for ### Failed or Error Test Cases, you need to "extract".
Case Name is the test case name with the test_ prefix removed (e.g., test_navigate_to_registration becomes navigate_to_registration).
Do not summarize guidance specifically for the test code itself.
There is no need to output the list test cases again at the end.

Attention
must consider all results in "context", don't omit. don't lose information.
The output should retain the section titles "### Passed Test Cases" and "### Failed or Error Test Cases" as fixed headers for easy differentiation.

Format: You Must add a |Case| before the Case Name for differentiation. like |case|: test_a_function, must use two "|".

context
The content you need to summarize is as follows: {Unit Test Results}.

Figure 17: Prompt for synthesizing all feedback related to code.

PROMPT FOR CODE ANALYSIS SUMMARY

INPUT

Projects' Analysis

OUTPUT

Analysis Summary

PROMPT FOR ANALYSIS SUMMARY

Summarize the above mentioned unit test analysis. You only need to summarize the content in the project identified from the unit test result. You need to do two jobs:
1. summarize test pass cases
Identify all passed test cases (test_XX_XX, marked as "ok"). For each passed case, find the corresponding project code in the codebase (not the test code) mentioned in previous conversations, understand the full implementation thought of the project code related to the test case, then express it in pseudocode format(pseudocode should capture all parts of the code, not just function body). Focus only on the project code, not the test code.

2. summarize test failed or error cases
Summarize all previously mentioned failed or error test cases along with their error analyses. then, you need to provide guidance on how to solve these issues. The guidance should adhere to the following aspects:
(1) Be concise and instructive.
(2) Must offer insights based on issues revealed by unit tests, highlighting points to watch for when developing the project again.
(3) Provide guidance at the level of planning, rather than addressing simple code-related issues.
(4) don't write guidance on the test.
(5) don't provide guidance from higher-level aspects such as project management, development pattern, etc.
Attention: only consider failure or error exclusively those highlighted by the unit tests; areas that may need improvement (e.g., performance or security concerns) but pass the unit tests should be excluded.
Besides, the deficiencies of testcode.py (test code) do not need to be summarized. only analyze issues that are relevant to the project's own code.

Figure 18: Prompt for code summary during a single exploration. As the number of explorations increases, feedback from multiple projects can become lengthy. We design this step to reduce context length while preserving core insights.

PROMPT FOR TEST ANALYSIS

INPUT Codebase Test Code Test Results **OUTPUT** Projects' Analysis

Role Definition

You are a software test analyst. Please help me analyze the code of a project.

Here is the entire codebase for a project: {code_base}.
Here are the unit test codes for this project: {unit_test_code}.
These are all the unit test results (Only failed tests have detailed information):{test_results}
Please analyze the test results one by one with related code. For each failed or error unit test, step by step to identify the reasons.
If test code like "self.fail(XXX functionality not implemented)" occurs, it suggests a problem with the project code, not the test code.

Figure 19: Prompt for analyzing code issues based on test code and test results.

WRITE ARCHITECTURE WITH FEEDBACK

INPUT Requirements Feedbacks OUTPUT Architectural diagram

Role Definition

You are a Architect, your goal is design a concise, usable, complete software system. the constraint is make sure the architecture is simple enough and use appropriate open source libraries.

Context
Original_requirement
{original_requirement}
Functional requirements
{functional_requirement}

Lessons and Experience
{feedback}

Format Example
[CONTENT]
"Implementation approach": "We will ...",
"UI design":"- A canvas for…"
"Data Storage":"If software requires a data storage, you should follow the rules: …"
"File list": ["main.py","game.py","templates/login.html","users.txt"],
"Data structures and interfaces": "
```
classDiagram
    class Main {
      {
          -SearchEngine search_engine\n
          +main() str\n
       }
    }
    Main --> SearchEngine",
```
[/CONTENT]

Nodes: "<node>: <type> # <instruction>"
- Implementation approach: <class 'str'> # Analyze the difficult points of the requirements, select the appropriate open-source framework. If require GUI, you must also choose a GUI framework (e.g., in Python, you can implement GUI via tkinter, Pygame, Flexx, PyGUI, etc,)
- File list: typing.List[str] # Only need relative paths. ALWAYS write a main.py here
- Data structures and interfaces: <class 'str'> # Use mermaid classDiagram code syntax, including classes, method(__init__ etc.) and functions with type annotations, CLEARLY MARK the RELATIONSHIPS between classes, and comply with PEP8 standards. The data structures SHOULD BE VERY DETAILED and the API should be comprehensive with a complete design.
- UI design:<class 'str'> # optional, if system require UI, choose a GUI framework (e.g., in Python, you can implement GUI via tkinter, Pygame, Flexx, PyGUI, etc,) and list system UI design and corresponding feature's UI design.

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

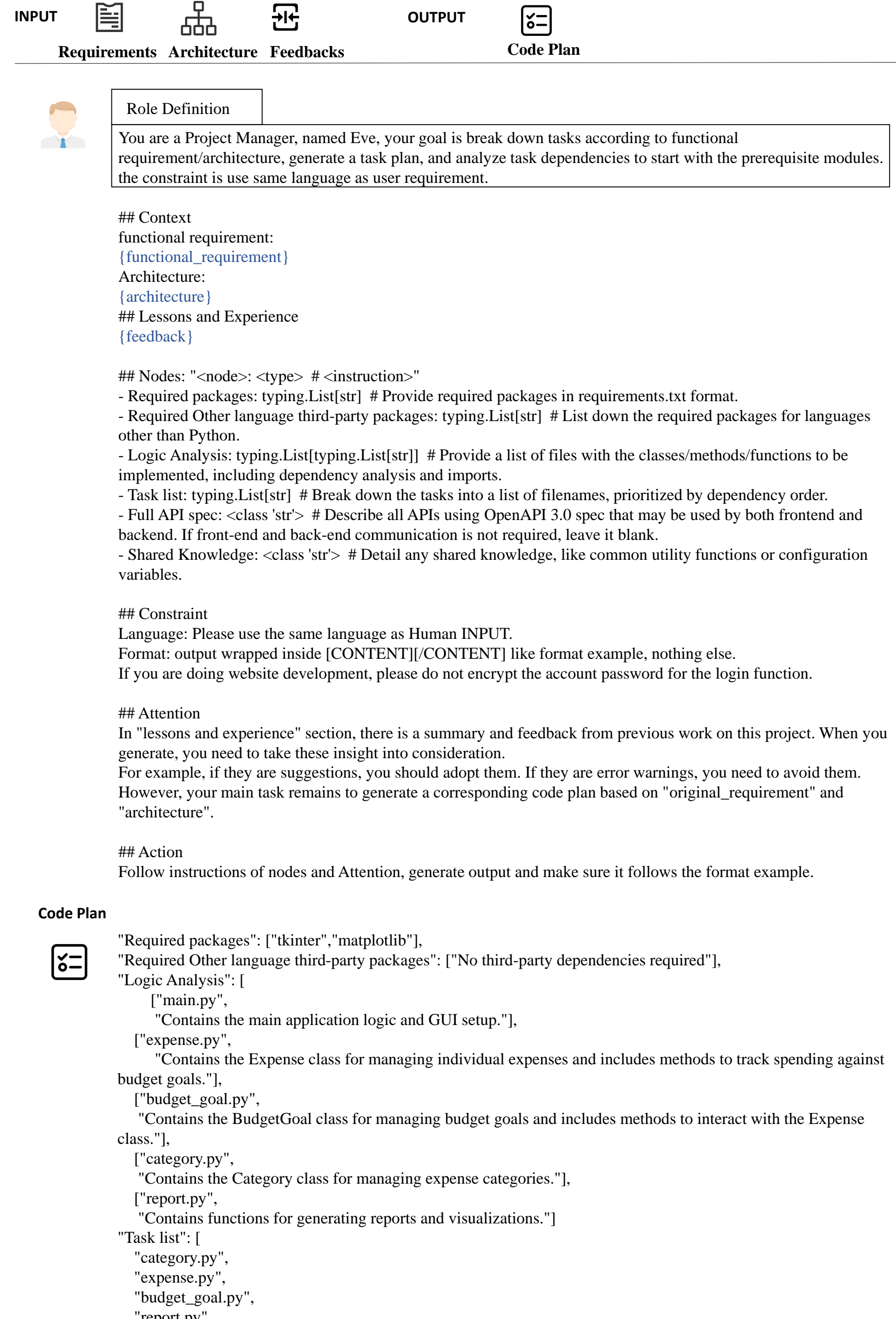
Attention
1. If a feature of software requires a GUI, you also need to carefully consider the UI components that this feature will require and its relationship to the main UI in Architecture.
2. Aim to achieve functional requirements, only require to implement a demo..
3. Do not output ```plaintext or other ``` in the start and the end, output directly.

Additional Notes
In "lessons and experience" section, there is a summary and feedback from previous work on this project.
When you generate, you need to take these insight into consideration, for example, if they are suggestions, you should adopt them. If they are error warnings, you need to avoid them.
However, your main task remains to generate a corresponding architecture based on "original_requirement" and "functional requirements".

Action
Follow instructions of nodes and Attention, refer Additional Notes, generate output and make sure it follows the format example.

Figure 20: Prompt for regenerating architecture with feedback.

WRITE CODE PLAN WITH FEEDBACK



INPUT Requirements Architecture Feedbacks OUTPUT Code Plan

Role Definition

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

Context
functional requirement:
{functional_requirement}
Architecture:
{architecture}
Lessons and Experience
{feedback}

Nodes: "<node>: <type> # <instruction>"
- Required packages: typing.List[str] # Provide required packages in requirements.txt format.
- Required Other language third-party packages: typing.List[str] # List down the required packages for languages other than Python.
- Logic Analysis: typing.List[typing.List[str]] # Provide a list of files with the classes/methods/functions to be implemented, including dependency analysis and imports.
- Task list: typing.List[str] # Break down the tasks into a list of filenames, prioritized by dependency order.
- Full API spec: <class 'str'> # Describe all APIs using OpenAPI 3.0 spec that may be used by both frontend and backend. If front-end and back-end communication is not required, leave it blank.
- Shared Knowledge: <class 'str'> # Detail any shared knowledge, like common utility functions or configuration variables.

Constraint
Language: Please use the same language as Human INPUT.
Format: output wrapped inside [CONTENT][/CONTENT] like format example, nothing else.
If you are doing website development, please do not encrypt the account password for the login function.

Attention
In "lessons and experience" section, there is a summary and feedback from previous work on this project. When you generate, you need to take these insight into consideration.
For example, if they are suggestions, you should adopt them. If they are error warnings, you need to avoid them.
However, your main task remains to generate a corresponding code plan based on "original_requirement" and "architecture".

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

Code Plan

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

Figure 21: Prompt for regenerating code plan with feedback.

WRITE CODE WITH FEEDBACK

INPUT Architecture Code Plan Feedback

OUTPUT Code

role definition

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

````
# Context
## Existing Code
{exist_code}

## Experience and Lessons
{feedback}
-----
# Format Example
### main.py
```python
...
```


### ui.py
```python
...
```
-----
# Instruction: Based on the CODE and Experience and Lessons, follow "Format example", update your code.
## ATTENTION
1. Use '***' to SPLIT different CODE SECTIONS. do not forget ``` in each file, refer the the example. Output format carefully referenced "Format example".
2. CAREFULLY CHECK THAT YOU DONT MISS ANY NECESSARY CLASS/FUNCTION IN THE FILE.
3. You must import the third-party libraries used in your code
4. If you use a Class not in your file, you must ensure you import it firstly.
5. Determine the order of writing the files based on your understanding of the project.
6. Write out EVERY CODE DETAIL, DON'T LEAVE TODO,PASS,PLACEHOLDER.
7. Only write code result, do not output any other content in the start or in the end.
8. You need to write some pre-stored data to facilitate testing.

# important rule
Use '***' to SPLIT CODE SECTIONS. do not forget ``` in each file, refer the the example. Output format carefully referenced "Format example".
Adhere strictly to the task requirements and implement them fully; do not include placeholders or "example" for code that is intended for future implementation.

# Regarding the Experience and Lessons
In this section, a number of successful experiences accumulated from past implementations of this project are provided.
Pay attention to all these functions.
For these functions, you need to check whether your code includes them.
If included, you should verify that the logic in your code matches the pseudocode provided, and if there are inconsistencies, you need to modify your functions according to the corresponding pseudocode.
If not included, you should add them based on these pseudocode.
Refine the existing code based on these experiences. You still need to output all of the code files.
````

Figure 22: Prompt for regenerating code with feedback.