SEW: Self-Evolving Agentic Workflows for Automated Code Generation

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have demonstrated effectiveness in code generation tasks. To enable LLMs to address more complex coding challenges, existing research has focused on crafting multi-agent systems with agentic workflows, where complex coding tasks are decomposed into sub-tasks, assigned to specialized agents. Despite their effectiveness, current approaches heavily rely on hand-crafted agentic workflows, with both agent topologies and prompts manually designed, which limits their ability to automatically adapt to different types of coding problems. To address these limitations and enable automated workflow design, we propose **S**elf-**E**volving **W**orkflow (**SEW**), a novel self-evolving framework that automatically generates and optimises multi-agent workflows. Extensive experiments on three coding benchmark datasets, including the challenging LiveCodeBench, demonstrate that our SEW can automatically design agentic workflows and optimise them through self-evolution, bringing up to 33% improvement on LiveCodeBench compared to using the backbone LLM only. Furthermore, by investigating different representation schemes of workflow, we provide insights into the optimal way to encode workflow information with text.

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities in automatic code generation, enabling developers to translate natural language descriptions into executable programs (Hong et al., 2023; Liu et al., 2024a). However, as coding tasks grow in complexity, relying on a single LLM instance (single-agent) to handle all aspects of code generation becomes increasingly challenging. To address this, recent studies have explored multi-agent systems (Huang et al., 2023; Islam et al., 2024) where multiple LLM-powered agents collaborate to solve intricate problems through structured workflows (Hong et al., 2023). These multi-agent systems decompose complex programming tasks into sub-tasks, assigning them to specialized agents with tailored prompts, enhancing execution and output quality.

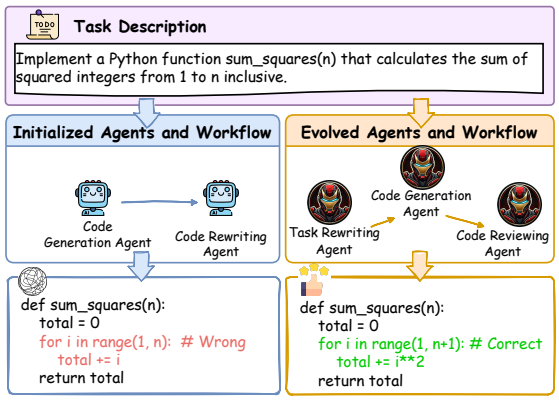

Figure 1: Illustration of agent and workflow evolution in code generation. The initialized setup (left) includes agents with naive prompts while the evolved setup (right) is equipped with enhanced prompts generated by SEW and a more sophisticated workflow structure.

Despite their effectiveness, current multi-agent systems rely heavily on manually designed workflows, where both the workflow topology and agents' prompts are manually crafted, hindering their adaptability to more complex coding task. For instance, a workflow optimised for machine learning task (Chi et al., 2024) differs significantly from one tailored for software development task (Qian et al., 2023). Manually crafting workflows for each task is inefficient and does not leverage LLM's full potential for autonomous adaptation.

To address these limitations, we propose **S**elf-**E**volving **W**orkflow (**SEW**), a novel framework designed to automatically generate and optimise multi-agent workflow. In particular, SEW achieves this by leveraging a novel evolutionary scheme to improve the workflow, i.e., the topology of workflows and the prompt of each agent. Figure 1 shows the agent and workflow evolution in code genera-

tion. In addition, to effectively represent agentic workflows in textual format, we explore and compare five different representation schemes, namely BPMN (White, 2004), CoRE (Xu et al., 2024b), Python code (Zhang et al., 2024c), YAML (Zhang et al., 2024b), and pseudo-code (Xiao et al., 2024). We evaluate each scheme based on how well it can be interpreted and optimised by our SEW framework, aiming to identify the optimal scheme for workflow representation and optimization.

Our contributions are: (1) We investigate different workflow representation schemes, such as BPMN, Python, CoRE, YAML, and pseudo-code, to determine the most effective format for LLM interpretation; (2) Unlike prior work that builds agents by assembling predefined operators, our framework automatically constructs agentic workflows from scratch, conditioned solely on task descriptions. (3) We introduce a self-evolving workflow design approach, SEW, where LLMs jointly improve workflow structures and agent prompts to optimise performance; (4) We conduct extensive experiments on three benchmark datasets, including MBPP, HumanEval, and LiveCodeBench, demonstrating that SEW can consistently improve workflow performance through self-evolution.

2 Related Work

2.1 Workflow Representations in Agents

In multi-agent systems, workflows establish structured information flows and task execution pipelines, enabling agents to solve complex problems (Hong et al., 2023; Gao et al., 2024). While natural language can describe workflows, its inherent ambiguity often leads to inconsistent interpretations, hindering precise task execution across agents (Xu et al., 2024b). To address this challenge, several studies have introduced specific representation schemes for SOPs. For example, *Business Process Model and Notation* (BPMN) (White, 2004) is a graphical modeling language designed to depict workflows by specifying the execution order of activities. Similarly, *Code Representation and Execution* (CoRE) (Xu et al., 2024b) provides a unified framework that integrates natural language programming, pseudo-code, and flow-based programming to improve workflow representation and execution. Additionally, Python code (Zhang et al., 2024c; Xu et al., 2024a), YAML (Qiao et al., 2023; Zhang et al., 2024b), and pseudo-code (Xiao et al., 2024; Li et al., 2025) are also commonly employed to define and manage agentic workflows.

2.2 Self-Evolving Agents

Existing agentic methods often yield suboptimal responses when prompts are poorly constructed. To address this, prompt optimization techniques (Zhou et al., 2022; Fernando et al., 2024; Agarwal et al., 2024; Liu et al., 2024b) have moved beyond static, manually crafted in-context prompts. For instance, automatic prompt engineer (APE) (Zhou et al., 2022) enhances prompts by searching through a pool of candidates. Similarly, Promptbreeder (Fernando et al., 2024) employs LLMs to mutate and evolve a population of task-specific prompts. MIPRO (Opsahl-Ong et al., 2024) is an optimizer designed to enhance multi-stage language model programs by refining both instructions and few-shot examples for each module. In multi-agent systems, recent studies have explored the evolution of agentic workflows and topologies (Zhang et al., 2024a; Zhou et al., 2024, 2025; Zhang et al., 2025). For example, MASS (Zhou et al., 2025) exploits the optimization of both prompt and workflow over a configurable topology space. Similarly, AFlow (Zhang et al., 2024a) employs a Monte Carlo Tree Search to enhance workflow efficiency, while EvoFlow (Zhang et al., 2025) introduces a framework for the automated search of heterogeneous agentic workflows. EvoAgent (Yuan et al., 2024) is designed to automatically extend expert agents into multi-agent systems using evolutionary algorithms. In contrast, our SEW introduces a self-evolving mechanism that leverages diverse workflow representation schemes, jointly optimising prompts for both agents and their workflow.

3 SEW

Task Definition. We focus on the task of code generation, a task that requires multi-agent collaboration (Hong et al., 2023), aiming to produce executable code based on a textual coding problem. To tackle this task, we deploy an LLM-based multi-agent system to generate code, where each agent processes a textual prompt and produces a corresponding textual output. We define the textual prompt of an LLM agent a as $\mathcal{T}$ and a sequence of LLM agents, i.e., a workflow as W.

Preliminary. Evolutionary prompts are central to SEW. Rather than relying on training data, SEW employs LLMs as mutation operators by concatenating the evolutionary prompts with the task

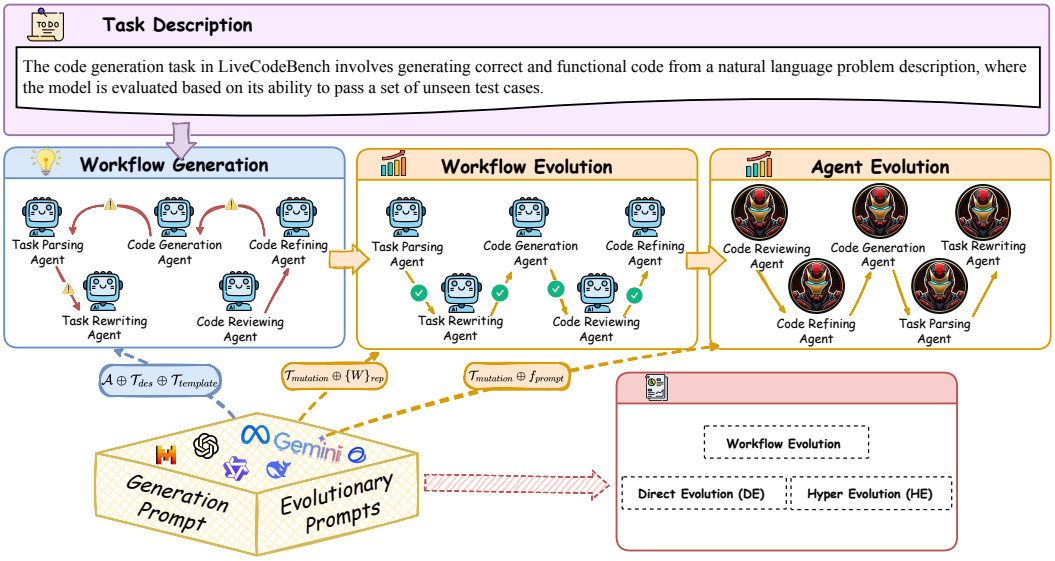

Figure 2: The overall framework of SEW. The process begins with workflow generation, followed by workflow evolution. Then each agent within the evolved workflow will be equipped with enhanced prompts generated by the agent evolution module. Such an agent evolution module is driven by the Direct Evolution (DE) operator and Hyper Evolution (HE) operator, leveraging LLMs, where we use a mutation prompt $\mathcal{T}_{mut}$ or a hyper-mutation prompt $\mathcal{T}_{hmut}$ to enhance the prompt of an agent.

prompt to generate a more effective task prompt. We define two evolutionary operators, namely the Direct Evolution (DE) operator $\mathcal{F}(\cdot)$ and the Hyper Evolution (HE) operator $\mathcal{H}(\cdot)$, where $\mathcal{F}(\cdot)$ and $\mathcal{H}(\cdot)$ take a workflow W or an agent a as the input and output an enhanced workflow W' or an agent a'. Specifically, $\mathcal{F}(\cdot)$ and $\mathcal{H}(\cdot)$ operators leverage (1) mutation prompts $\mathcal{T}_{\text{mut}}$, (2) hyper-mutation prompt, and (3) thinking-style prompts $\mathcal{T}_{\text{think}}$ (Fernando et al., 2024). Figure 3 shows examples of these evolutionary prompts and how they are evolved by both DE and HE.

Overview of SEW. Our SEW framework consists of three main modules: (a) Workflow Generation, (b) Workflow-Evolution, and (c) Agent-Evolution. The overview of our SEW framework is illustrated in Figure 2. As shown in Figure 2, our SEW first generates an initial workflow based on the task description using one of the representation schemes introduced in Section 4. Second, the workflow evolution module of SEW will leverage our evolution method to reconstruct the initial workflow. Finally, inspired by PromptBreeder (Fernando et al., 2024), our agent evolution module will apply either the agentic DE or agentic HE method to equip each agent with a more sophisticated prompt. The pseudo-code of SEW is shown in Algorithm 1.

Workflow Generation. To generate workflows, we use an LLM to generate default workflows based on the given task description $\mathcal{D}$ [1] and a template workflow W^{temp}. A template workflow can be denoted with different workflow representation schemes. In particular, our SEW explore five different schemes, namely *Business Process Model and Notation (BPMN)* (White, 2004), *Code Representation and Execution (CoRE)* (Xu et al., 2024b), *python*, *YAML* and *pseudo-code*, with their detailed description presented in Section 4. Figure 4 shows two examples of the template workflow.

From the workflow generation process as shown in Algorithm 1, we can obtain a set of default workflows W^{def}. Later, we will present how to use our workflow evolution module to rearrange and modify the structure of W^{def}.

Workflow Evolution. To formalise the workflow evolution process of SEW, first we define a workflow W represented with a certain representation scheme rep, where all W in rep are in textual format. We use the DE operator $\mathcal{F}(\cdot)$ to generate an evolved workflow as follows:

$$W' = \mathcal{F}(W_{def} | \mathcal{T}_{\text{mut}}), \tag{1}$$

where W' is the self-evolved workflow, $\mathcal{T}_{\text{mut}}$ is the mutation prompt and $\mathcal{F}(\cdot)$ representing the operation that an LLM takes W_{def} and $\mathcal{T}_{\text{mut}}$ as input and output W' (see Figure 3 for more details).

[1] Appendix A.2 shows all of the task descriptions.

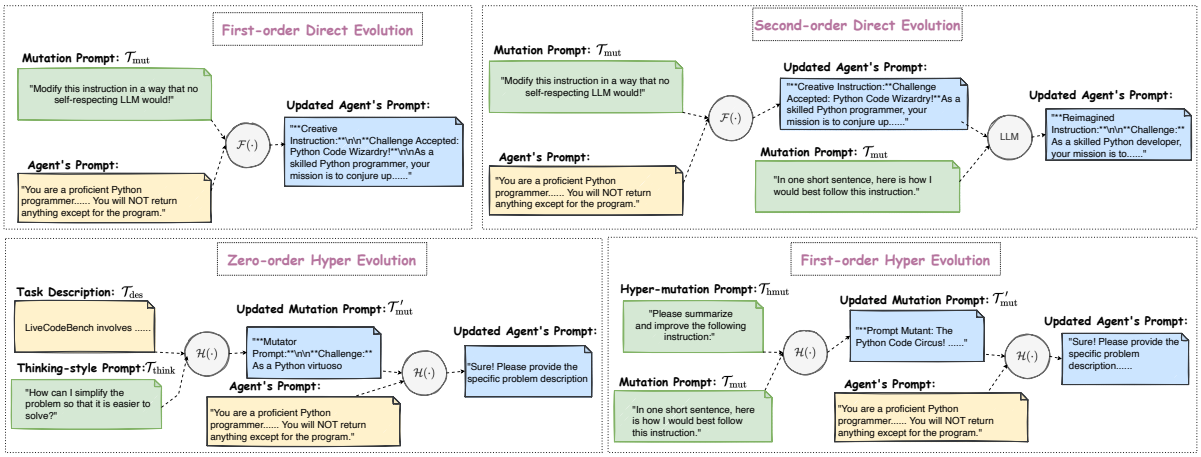

Figure 3: Illustration of the Direct Evolution and Hyper Evolution of SEW. We use green, yellow and blue boxes to indicate the evolutionary prompt, default agent prompt and textual output of evolutionary operators.

It should be noted that the mutation prompt $\mathcal{T}_{mut}$ cannot ensure that W' is a valid workflow. For example, W' may not strictly follow the format of the representation scheme. To measure the validity of W', we define two rates, namely the Logical Successful Rate (*LSR*) and Generation Successful Rate (*GSR*). The *LSR* denotes the probability that generated W' is valid, and the *GSR* denotes the probability that the output of W' is executable Python code. Specifically, $LSR = \frac{\sum_{i=1}^{|W'|} \mathbb{I}(\text{isValid}(W'_i))}{|W'|}$ and $GSR = \frac{\sum_{i=1}^{|W'|} \mathbb{I}(\text{isPython}(output(W'_i)))}{|W'|}$. By measuring *LSR* and *GSR* of a certain representation scheme, we can determine which scheme is more suitable for SEW.

Agent Evolution. After modifying the structure of workflows using the workflow evolution module, the next step is to modify each agent's prompt. Similar to the workflow evolution, the agent evolution also relies on the mutation prompt. As mentioned earlier we use Direct Evolution (DE) and Hyper Evolution (HE) to improve an agent, where DE aims to modify an agent's prompt by directly applying a mutation prompt to it while HE first modifies the mutation prompt then apply the modified mutation prompt to an agent.

Agentic Direct Evolution. To enhance the performance of an agent, SEW directly apply the mutation prompt $\mathcal{T}_{mut}$ to an agent's prompt using the direct evolution operator as follows:

$$a' \leftarrow \mathcal{F}(a|\mathcal{T}_{\text{mut}}), \tag{2}$$

where a is an agent and a' is the agent with modified prompt, and we define the operation above as the first-order direct evolution.

Algorithm 1: Self-Evolving Workflow

Input: Task Description $\mathcal{D}$, Workflow Template W^{temp}, Mutation Prompt $\mathcal{T}_{\text{mut}}$, Hyper Mutation Prompt $\mathcal{T}_{\text{hmut}}$, Thinking-style Prompt $\mathcal{T}_{\text{think}}$
Output: Optimized Workflow W'

Function SEW($\mathcal{T}_{des}, \mathcal{T}_{temp}, \mathcal{T}_{mut}, \mathcal{T}_{hmut}, \mathcal{T}_{think}$):
  1. Workflow Generation;
  $W_{def} \leftarrow$ GenerateWorkflows($\mathcal{T}_{\text{des}}, \mathcal{T}_{\text{temp}}$);
  2. Workflow Evolution;
  for *each workflow* W_{def} **do**
    $W' \leftarrow \mathcal{F}(W_{def}|\mathcal{T}_{\text{mut}})$;
  3. Agent Evolution;
  for *each agent* a *in* W' **do**
    3.1 Select Evolution Method;
    if *First-order DE* **then**
      $a' \leftarrow \mathcal{F}(a|\mathcal{T}_{\text{mut}})$;
    else if *Second-order DE* **then**
      $a'' \leftarrow \mathcal{F}(\mathcal{F}(a|\mathcal{T}_{\text{mut}})|\mathcal{T}_{\text{mut}})$;
    else if *Zero-order HE* **then**
      $a' \leftarrow \mathcal{H}(a|\mathcal{H}(\mathcal{T}_{\text{des}}|\mathcal{T}_{\text{think}}))$;
    else if *First-order HE* **then**
      $a'' \leftarrow \mathcal{H}(a|\mathcal{H}(\mathcal{T}_{\text{mut}}|\mathcal{T}_{\text{hmut}}))$;
  return W';

Based on the first-order direct evolution, we propose the second-order direct evolution:

$$a'' \leftarrow \mathcal{F}(\mathcal{F}(a|\mathcal{T}_{\text{mut}})|\mathcal{T}_{\text{mut}}) \tag{3}$$

By applying second-order direct evolution, we aim to further enhance the performance of an LLM agent.

Agentic Hyper Evolution. Different from Direct Evolution, Hyper Evolution focuses on generating more effective mutation prompts. In other words, HE first modifies the mutation prompt $\mathcal{T}_{mut}$ then uses the new mutation prompt $\mathcal{T}'_{mut}$ to improve an

agent's prompt. Formally, we define the zero-order hyper evolution as below:

$$a' \leftarrow \mathcal{H}(a|\mathcal{H}(\mathcal{T}_{\text{des}}|\mathcal{T}_{\text{think}})) \quad (4)$$

where $\mathcal{T}_{think}$ are text descriptions of general cognitive heuristics (Fernando et al., 2024).

For zero-order HE, we use the general cognitive heuristics $\mathcal{T}_{think}$ to generate useful prompts for solving problems described by the task description $\mathcal{D}$. Similar to how we use the mutation prompt $\mathcal{T}_{mut}$ to modify an agent's prompt, we can use a hyper-mutation prompt instead of $\mathcal{T}_{think}$ to modify $\mathcal{T}_{mut}$, which is defined as first-order HE.

From Eq 4, a new mutation prompt is generated from the task description and some cognitive heuristics. In another way, we can use a hyper-mutation prompt to directly generate new variants from $\mathcal{T}_{mut}$ as follows:

$$a'' \leftarrow \mathcal{H}(a|\mathcal{H}(\mathcal{T}_{\text{mut}}|\mathcal{T}_{\text{hmut}})) \quad (5)$$

Finally, by combining the workflow-evolution and agent evolution, our SEW can generate more effective variants of workflows for solving the code generation task. In the next section, we will present and compare those five different representation schemes that can be leveraged by SEW.

4 Workflow Representation

To generate a workflow using LLM, appropriate workflow textual representation schemes are essential. In fact, while it is straightforward to execute a workflow using code, representing it in natural language is non-trivial. A well-designed representation scheme should capture the structural and semantic components of a workflow and be easily interpreted by LLMs for downstream modification.

As we discussed in the related work section, we explored five different textual representation schemes that can be used to denote workflows namely, *Business Process Model and Notation (BPMN)* (White, 2004), *Code Representation and Execution (CoRE)* (Xu et al., 2024b), *python*, *YAML* and *pseudo-code*, where each representation scheme can be used to denote a workflow by text. The choice of these five schemes was driven by their distinct advantages in facilitating the representation and execution of agentic workflows, particularly in the context of self-evolving agentic workflows that our method, SEW, aims to optimise.

```
# BPMN_workflow
<definitions xmlns="http://www.omg.org/spec/BPMN/
20100524/MODEL">
<process id="software_dev_workflow"
     isExecutable="true">
    <startEvent id="start" />
    <task id="parse_task" name="Parse Task" />
    ...
    <sequenceFlow id="flow6"
     sourceRef="refine_code" targetRef="end" />
</process>
</definitions>
```

```
# CoRE_workflow
Step 1:::Process:::Parse Task:::next::Step 2
Step 2:::Process:::Refine Task:::next::Step 3
Step 3:::Process:::Generate Code:::next::Step 4
Step 4:::Process:::Review Code:::next::Step 5
Step 5:::Process:::Refine Code:::next::Step 6
Step 6:::Terminal:::End of Workflow:::
```

Figure 4: A workflow represented by the BPMN and the CoRE schemes, respectively.

BPMN: This graphical standard is well-established in business process modeling and widely recognized for its ability to clearly depict the order of tasks and their dependencies.

CoRE: CoRE integrates natural language programming, pseudo-code, and flow-based programming, and is a strong candidate for agentic workflows. It allows workflows to be directly executable and interpretable by LLMs, offering advantages for our self-evolving framework.

Python: As a widely adopted programming language, Python is not only familiar to many practitioners but also flexible in terms of representing workflows through its readable syntax and extensive ecosystem of libraries. For agentic workflows requiring programmatic execution, Python allows for easy integration and adaptation of agents into working solutions.

YAML: YAML is a human-readable data serialisation format widely used for configuration files and workflow definitions due to its simplicity and readability. YAML's flexibility in representing hierarchical data structures makes it well-suited for workflows that need to be configured or defined by humans but executed by machines.

Pseudo-code: Pseudo-code is a high-level representation that is often used for illustrating algorithms and workflows in a way that is easy for both humans and machines to understand. Pseudo-code offers an abstraction that bridges natural language and formal code, making it an excellent choice for expressing workflows that need to be easily read and modified.

To clearly illustrate the differences between workflow representation schemes, we present an

example agentic workflow represented using both the BPMN and CoRE schemes in Figure 4. In Figure 4, a software development pipeline, consisting of sequential tasks such as parsing input, refining content, generating code, reviewing, and iterating improvements, is represented by BPMN and CoRE, respectively. Each stage is represented as a task node, while dependencies between tasks are captured as sequence flows, ensuring clear process execution. Although denoted with different representation schemes, they shall perform the same function when executed[2].

These five schemes were chosen for their diverse capabilities in representing workflows and their practical utility in a self-evolving framework, where agents and workflows are dynamically generated and optimised. Our exploration of these schemes aims to identify the most suitable representation for evolving agentic workflows in code generation tasks, where LLMs are leveraged for both understanding and executing the workflows.

5 Experiments

5.1 Dataset

To examine our proposed SEW framework, we choose the LiveCodeBench (LCB) (Jain et al., 2024) dataset, which is a comprehensive benchmark designed to evaluate the coding capabilities of LLMs. We randomly sampled 100 samples from the code generation subset of LCB[3] for validation and the remaining 300 samples for testing. In addition, we also use the MBPP (Austin et al., 2021) and HumanEval (Chen et al., 2021) datasets following the data split in AFlow (Zhang et al., 2024a). To evaluate performance on the code generation task, each method is required to generate 10 candidate solutions per sample. We use pass@1, pass@5, and pass@10 as evaluation metrics.

5.2 Baselines

We compare our proposed SEW against five baseline prompting techniques across two different backbone models (i.e. GPT-4o mini and Gemini-1.5-pro-002) on three code generation tasks (i.e HumanEval, MBPP, and LCB): (1) Backbone Models (GPT-4o mini and Gemini-1.5-pro-002). (2) Chain-of-Thought (CoT) (Wei et al., 2022) Uses reasoning steps explicitly stated within the prompt. (3) Automated Design of Agentic Systems (ADAS) (Hu et al., 2024): A methodology that leverages meta-agent frameworks to automatically design and optimise agentic systems. (4) AFlow (Zhang et al., 2024a): An automated framework that efficiently explores and optimises agentic workflows using Monte Carlo Tree Search. (5) PromptBreeder (Fernando et al., 2024) is a gradient-free evolutionary framework that improves agents by iteratively mutating and selecting prompt variants.

Method	HumanEval	MBPP	LCB
GPT-4o mini			
GPT-4o mini	80.2	63.4	38.0
CoT	87.2	68.3	40.1
PromptBreeder	90.9	83.2	45.9
ADAS	88.8	73.0	42.5
AFlow	91.6	83.9	-
SEW (GPT-4o mini)	**92.1**	**84.1**	**50.9**
Gemini-1.5-pro-002			
Gemini-1.5-pro-002	79.8	61.0	36.7
CoT	86.7	68.3	39.8
PromptBreeder	88.6	71.7	44.8
ADAS	86.7	71.0	43.3
AFlow	89.3	70.0	-
SEW (Gemini-1.5-pro-002)	**89.9**	**74.1**	**47.8**

Table 1: Performance comparison (pass@1) between our SEW and baselines. '-' refers to out-of-time errors, where the LLM executor has been trapped in executing accidental scripts with infinite loops. We adopt two LLMs, i.e., GPT-4o mini and Gemini-1.5-pro, as backbone models for all methods.

5.3 Experimental Setup

We conduct an exhaustive search on self-evolved workflows W', represented by the following methods, including BPMN, CoRE, python, YAML and pseudo-code. We use all mutation prompts to evolve workflows represented by 5 schemes. Although various types of workflows are generated during the self-evolution process, not all of them are valid for code generation tasks. Among all generated workflows, the **task parsing workflow** and **code rewriting workflow**[4] are more effective than the other counterparts. In particular, variants based on these two workflows can largely outperform competitive baselines, hence, we choose the best variant to represent our SEW..

5.4 Main Results

To compare the performance of SEW and other baselines, we adopt two backbone models i.e., GPT-

[2]Additional representation formats, such as Python, pseudo-code and YAML, are provided in Appendix A.1

[3]We use the release_v1 version of the code generation subset of LCB.

[4]We also report the token cost of these two workflows in the Appendix A.4

Rep method	BPMN	CoRE	python	YAML	pseudo-code
LSR	**87.3%**	74.5%	**87.3%**	78.2%	52.7%
GSR	47.3%	**72.7%**	29.1%	47.3%	36.4%

Table 2: Logic Successful Rate (*LSR*) and Generation Successful Rate (*GSR*) for Business Process Model and Notation, Code Representation and Execution, python, YAML and pseudo-code.

4o mini and Gemini-1.5-pro-002. From Table 1, we find that (1) SEW can largely outperform those two backbone models at both settings; (2) SEW is more effective than CoT, a robust prompting technique for enhancing LLM's ability to solve complex tasks by breaking them down into sequential thought processes; (3) when leveraging the same backbone model, our SEW outperform other state-of-the-art workflow designing methods such as ADAS and AFlow. Therefore, we can conclude that our SEW framework is more effective than different types of baselines under the same setting in the code generation task. In addition, we observe that across the three datasets, methods using GPT-4o mini as the backbone generally outperform those using Gemini-1.5-pro-002. Hence, to save space, we report only the analysis of SEW (GPT-4o mini) in the following sections.

5.5 Analysis

> **RQ1**: Which scheme is the most effective for structuring agentic workflows?

To identify the most suitable workflow scheme for LLMs among the five, we conducted an exhaustive search using various mutation prompts. For a given workflow W represented in Python, 100 different mutation prompts generated 100 variants. If 50 of these variants are parsable and 30 can generate executable codes, the *LSR* and *GSR* for Python are 50% and 30%, respectively. Notably, *LSR* is always greater than or equal to *GSR*, as not all parsed workflows can complete the task.

As shown in Table 2, BPMN and Python achieved the highest LSR at 87.3%. However, their GSR performance was suboptimal, whereas the recently proposed CoRE method achieved the best GSR. This suggests that while traditional BPMN and Python representations are easier for LLMs to parse, the CoRE method – which integrates natural language programming, pseudo-code programming, and flow programming – is the most effective for workflow representation. We therefore conclude that CoRE enables optimal comprehension and utilisation when denoting agentic workflows.

Method	HumanEval	MBPP	LCB
GPT-4o mini	80.2	63.4	38.0
task parsing workflow			
Only Workflow-Evolved	87.2	72.3	42.3
Workflow + Agent Co-Evolved	91.0	**84.1**	**50.9**
%improv.	4.35	16.3	20.3
code rewriting workflow			
Only Workflow-Evolved	88.8	70.5	40.1
Workflow + Agent Co-Evolved	**92.1**	80.8	46.8
%improv.	3.71	14.6	16.7

Table 3: Performance comparison (pass@1) between the default version of two representative workflows generated from workflow evolution and their improved variants using agent evolution. All workflows use GPT-4o mini as their backbone model.

> **RQ 2**: How do SEW's workflow evolution and agent evolution modules affect the performance of coding generation?

To understand how our workflow evolution and agent evolution modules affect the performance of workflows generated by SEW, we select two representative workflows generated by SEW, namely **task parsing workflow** and **code rewriting workflow**. We chose these two workflows since most of the variants built upon these two workflows can bring large improvements. Specifically, the task parsing workflow leverages an agent to first parse the task and then send the parsed result to a coding agent to generate the code subsequently. In comparison, a code rewriting workflow incorporates a code generation agent to generate the initial outcome and then uses the code reviewing agent to determine if this outcome can pass the test followed by a code rewriting agent to rewrite the code based on the feedback from the code reviewing agent.[5]

Notably, the workflow evolution module is designed to generate novel workflow structures, while the agent evolution module focuses on creating effective prompts for each agent. In particular, we compare: (1) workflows generated by the workflow evolution module versus those produced by the backbone model, and (2) workflows generated by the workflow evolution module versus those that incorporate both workflow and agent evolutions. As shown in Table 3, the task parsing and code rewriting workflows produced by SEW consistently outperform the GPT-4o mini backbone model across three datasets. This initial improve-

[5]Details of these two workflows can be found in Appendix A.2.

Figure 5: Performance comparison of Code Rewriting and Task Parsing Workflows under different agent evolution strategies on the LCB dataset.

ment suggests that our workflow evolution module generates novel workflow topologies more effectively than relying solely on the LLM. Building on these novel workflows, the agent evolution module further enhances performance by generating high-quality prompts for each agent. Specifically, our agent evolution module improves the performance of the task parsing workflow by 20.3% on the LCB dataset. In summary, our results demonstrate that the workflow evolution module effectively produces novel workflow structures, and the agent evolution module further unlocks their potential by injecting high-quality prompts.

> **RQ 3**: How do different agentic evolution strategies affect the performance of workflows generated by SEW?

We have introduced the **Direct Evolution** (DE) and **Hyper Evolution** (HE) operators, where for each we proposed its corresponding lower-order and higher-order versions. To examine the effectiveness of different operators, we randomly sampled five different mutation prompts and used these randomly sampled mutation prompts to generate five different variants for both workflows mentioned earlier for each operator. We use four box plots to illustrate the performance distribution of these two workflows on the LCB dataset.

From Figure 5, we can observe that HE consistently demonstrates lower variance than DE by comparing the first row and second row of Figure 5. The variance of both workflows under the zero-order hyper evolution is especially small. This indicates that the HE operator, particularly zero-order HE, exhibit superior robustness compared to DE, as they are less sensitive to variations in mutation prompts across different tasks. In terms of best performance, DE, especially second-order DE, tends to achieve higher peak performance in certain metrics, such as pass@10 for Code Rewriting Workflow, where it reaches up to 0.580. This suggests that DE can optimize for specific high-performance outcomes. On the other hand, HE, while slightly lower in peak performance, provides a more balanced and reliable performance profile, making it more suitable for consistency.

Therefore, the choice between DE and HE depends on the requirements of the task: DE is preferable for maximizing performance, while HE is better suited for real-world applications where robustness is more important. In addition, higher-order evolutions (Second-order DE and First-order HE) are better suited for tasks that require maximizing performance and can tolerate some variability, while lower-order evolutions (First-order DE and Zero-order HE) provide higher robustness.

6 Conclusion

In this paper, we introduce Self-Evolving Workflow (SEW), a novel framework that enables LLM-based multi-agent workflows to automatically adapt and evolve for improved performance in automatic code generation. Unlike conventional hand-crafted workflows, SEW leverages self-evolving prompting to optimise both workflow structures and individual agent capabilities. Through extensive experiments on three coding generation benchmarks, we demonstrate that self-evolved workflows consistently outperform single-agent baselines, even when the latter are enhanced with improved prompts. Moreover, our results show that higher-order evolution strategies further improve workflow effectiveness by refining task execution dynamics. Among different workflow representation schemes, CoRE emerges as the most effective, offering a balance between logical correctness and execution success. These findings suggest that SEW provides a promising direction for agentic workflow optimisation, reducing reliance on manual workflow design and prompt engineering while improving adaptability and efficiency.

Limitations

While SEW proves effective for code generation, its generalization to other AI-driven tasks, such as reasoning or planning, remains unknown. Another limitation lies in the workflow execution constraints, where some generated workflows, despite being logically sound, fail to produce executable outputs, suggesting a need for more robust validation mechanisms. Furthermore, the effectiveness of SEW is dependent on the capabilities of the underlying LLM, meaning that future advancements in LLM architectures could significantly impact its performance. Addressing these limitations in future work will be crucial for extending SEW's applicability and enhancing its adaptability for broader tasks.

References

Eshaan Agarwal, Vivek Dani, Tanuja Ganu, and Akshay Nambi. 2024. Promptwizard: Task-aware agent-driven prompt optimization framework. *arXiv preprint arXiv:2405.18369*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

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. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Yizhou Chi, Yizhang Lin, Sirui Hong, Duyi Pan, Yaying Fei, Guanghao Mei, Bangbang Liu, Tianqi Pang, Jacky Kwok, Ceyao Zhang, and 1 others. 2024. Sela: Tree-search enhanced llm agents for automated machine learning. *arXiv preprint arXiv:2410.17238*.

Chrisantha Fernando, Dylan Sunil Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. 2024. Promptbreeder: Self-referential self-improvement via prompt evolution. In *International Conference on Machine Learning*, pages 13481–13544. PMLR.

Dawei Gao, Zitao Li, Xuchen Pan, Weirui Kuang, Zhijian Ma, Bingchen Qian, Fei Wei, Wenhao Zhang, Yuexiang Xie, Daoyuan Chen, and 1 others. 2024. Agentscope: A flexible yet robust multi-agent platform. *arXiv preprint arXiv:2402.14034*.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, and 1 others. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.

Shengran Hu, Cong Lu, and Jeff Clune. 2024. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*.

Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, 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 (Volume 1: Long Papers)*, pages 4912–4944.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2025. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–23.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024a. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36.

Shengcai Liu, Caishun Chen, Xinghua Qu, Ke Tang, and Yew-Soon Ong. 2024b. Large language models as evolutionary optimizers. In *2024 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. 2024. Optimizing instructions and demonstrations for multi-stage language model programs. *arXiv preprint arXiv:2406.11695*.

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*, 6(3).

Bo Qiao, Liqun Li, Xu Zhang, Shilin He, Yu Kang, Chaoyun Zhang, Fangkai Yang, Hang Dong, Jue Zhang, Lu Wang, and 1 others. 2023. Taskweaver: A code-first agent framework. *arXiv preprint arXiv:2311.17541*.

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.

Stephen A White. 2004. Introduction to bpmn. *Ibm Cooperation*, 2(0):0.

Ruixuan Xiao, Wentao Ma, Ke Wang, Yuchuan Wu, Junbo Zhao, Haobo Wang, Fei Huang, and Yongbin Li. 2024. Flowbench: Revisiting and benchmarking workflow-guided planning for llm-based agents. *CoRR*.

Frank F Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Z Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, and 1 others. 2024a. Theagentcompany: benchmarking llm agents on consequential real world tasks. *arXiv preprint arXiv:2412.14161*.

Shuyuan Xu, Zelong Li, Kai Mei, and Yongfeng Zhang. 2024b. Aios compiler: Llm as interpreter for natural language programming and flow programming of ai agents. *CoRR*.

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Dongsheng Li, and Deqing Yang. 2024. Evoagent: Towards automatic multi-agent generation via evolutionary algorithms. *arXiv preprint arXiv:2406.14228*.

Guibin Zhang, Kaijie Chen, Guancheng Wan, Heng Chang, Hong Cheng, Kun Wang, Shuyue Hu, and Lei Bai. 2025. Evoflow: Evolving diverse agentic workflows on the fly. *arXiv preprint arXiv:2502.07373*.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, and 1 others. 2024a. Aflow: Automating agentic workflow generation. *arXiv preprint arXiv:2410.10762*.

Xinyu Zhang, Siddharth Muralee, Sourag Cherupattamoolayil, and Aravind Machiry. 2024b. On the effectiveness of large language models for github workflows. In *Proceedings of the 19th International Conference on Availability, Reliability and Security*, pages 1–14.

Yaolun Zhang, Yinxu Pan, Yudong Wang, and Jie Cai. 2024c. Pybench: Evaluating llm agent on various real-world coding tasks. *arXiv preprint arXiv:2407.16732*.

Han Zhou, Xingchen Wan, Ruoxi Sun, Hamid Palangi, Shariq Iqbal, Ivan Vulić, Anna Korhonen, and Sercan Ö Arık. 2025. Multi-agent design: Optimizing agents with better prompts and topologies. *arXiv preprint arXiv:2502.02533*.

Wangchunshu Zhou, Yixin Ou, Shengwei Ding, Long Li, Jialong Wu, Tiannan Wang, Jiamin Chen, Shuai Wang, Xiaohua Xu, Ningyu Zhang, and 1 others. 2024. Symbolic learning enables self-evolving agents. *arXiv preprint arXiv:2406.18532*.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large language models are human-level prompt engineers. In *The Eleventh International Conference on Learning Representations*.

A Appendix

A.1 Workflow Representation Schemes

In Section 4, we have introduced five different representation schemes namely, *Business Process Model and Notation (BPMN)* (White, 2004), *Code Representation and Execution (CoRE)* (Xu et al., 2024b), *python*, *YAML* and *pseudo-code*, where we used two scripts (Figure 4) to denote a workflow represented by BPMN and CoRE, respectively. Below, we will show the same workflow represented by python, YAML and pseudo-code.

```python
# python_workflow
steps = [
    {'name': 'task_parsing_agent', 'args': ['task_description'], 'output': 'parsed_task'},
    {'name': 'task_refinement_agent', 'args': ['task_description', 'parsed_task'], 'output': 'refined_task'},
    {'name': 'code_generation_agent', 'args': ['refined_task'], 'output': 'generated_code'},
    {'name': 'code_reviewer_agent', 'args': ['refined_task', 'generated_code'], 'output': 'review_comments'},
    {'name': 'code_refinement_agent', 'args': ['refined_task', 'review_comments'], 'output': 'refined_code'},
]
```

```pseudo
# pseudo_code_workflow
task_parsing_agent(task_description) -> parsed_task
task_refinement_agent(task_description, parsed_task) -> refined_task
code_generation_agent(refined_task) -> generated_code
code_reviewer_agent(refined_task, generated_code) -> review_comments
code_refinement_agent(refined_task, review_comments) -> refined_code
```

```yaml
# YAML_workflow
- name: task_parsing_agent
  args:
    - task_description
  output: parsed_task

- name: task_refinement_agent
  args:
    - task_description
    - parsed_task
  output: refined_task

- name: code_generation_agent
  args:
    - refined_task
  output: generated_code

- name: code_reviewer_agent
  args:
    - refined_task
    - generated_code
  output: review_comments

- name: code_refinement_agent
  args:
    - refined_task
    - review_comments
  output: refined_code
```

A.2 Prompt Corpus

In Section 3, we have introduced many textual variables including the task description, generation prompts and evolutionary prompts. In this section, we will detail the textual information.

Task Description $\mathcal{D}$ for each dataset:
LiveCodeBench: The code generation task in LiveCodeBench involves generating correct and functional code from a natural language problem description, where the model is evaluated based on its ability to pass a set of unseen test cases.

HumanEval: The HumanEval dataset, developed by OpenAI, comprises 164 handcrafted programming problems, each including a function signature, docstring, body, and multiple unit tests, designed to evaluate the code generation capabilities of large language models by assessing their ability to generate functionally correct code from docstrings.

MBPP: The MBPP (Mostly Basic Python Problems) dataset comprises approximately 1,000 crowd-sourced Python programming problems, each including a task description, code solution, and three automated test cases, designed to be solvable by entry-level programmers and covering programming fundamentals and standard library functionality.

We use W^{temp} to denote a template workflow, which is used to guide LLMs to generate default workflows. In particular, we use the example workflows listed in Appendix A.1 as our templates.

Default agent generation prompt:
You are an AI prompt engineer. Your task is to create specific prompts for each agent in the provided workflow. Please follow these steps:

1. Understand the Workflow: Here is the detailed workflow: [Detailed workflow]
2. Identify Agent Roles: Based on the workflow, determine the distinct roles and responsibilities of each agent involved.
3. Generate Agent-Specific Prompts: For each identified agent, craft a clear and concise prompt that includes:
   - Agent Role: A brief description of the agent's function within the workflow.
   - Objectives: The specific goals the agent is expected to achieve.
   - Inputs: The information or data the agent will receive.
   - Outputs: The expected results or actions the agent should produce.

Default workflow generation prompt:
You are an AI workflow designer. Your task is to create a detailed Agent Workflow tailored to the provided workflow template and dataset description. Please follow these steps:

1. Review the Workflow Template: [Detailed workflow template]
2. Analyze the Dataset Description: [Dataset description]
3. Design the Agent Workflow: Based on the above information, develop a comprehensive Agent Workflow that includes:
   - Inputs and Outputs: Define the types of input data and the expected output results.
   - Steps and Sequence: Outline each step of the workflow and specify the order of execution.
   - Agent Roles and Responsibilities: Describe the role and duties of the agent at each step.

Our used evolutionary prompts, including the *mutation prompt*, *hyper-mutation prompt* and *thinking-style* prompt, can be found in the appendix of PromptBreeder (Fernando et al., 2024).

In Table 3, we have presented two workflows obtained from SEW namely the Task Parsing Workflow and Code Rewriting Workflow. In the following, we will show the name of each agent and its corresponding prompt in those workflows. In particular, we will first show the version that has not been improved by the agent evolution module, followed by the version that has been improved by the agent evolution module.

A.2.1 Workflows without agent evolution module

Task Parsing Workflow:
1. Task Parsing Agent: "You are a task parsing agent. Comprehensively summarize the given programming task for the subsequent code generation. You will NOT return anything except for the task summary. { TASK PROMPT}
2. Code Generation Agent: "You are a proficient Python programmer. Your task is to write Python code according to the summary parsed by your colleague. You will be given the problem description followed by the summary. You will NOT return anything except for the program." ⊕ { Output from Task Parsing Agent}

Code Rewriting Workflow:
1. Code Generation Agent: "You are a proficient Python programmer. Your task is to write Python code according to the summary parsed by your colleague. You will be given the problem description followed by the summary. You will NOT return anything except for the program." ⊕ { TASK PROMPT}
2. Code Reviewer Agent: "You are a critical python code reviewer. You are tasked to label generated codes with 1 or 0, where 1 indicates that this code satisfies the requirements and can pass the sample test, while 0 indicates that this code doesn't satisfies the requirements and will fail the sample test. You will be given the Problem Description followed by the corresponding Generated Code. You will NOT return anything except for the numerical label." ⊕ { Output from Code Generation Agent}
3. Code Rewriting Agent: "You are a proficient Python programmer tasked with coding solutions based on given problem specifications. You just generated some codes that cannot pass the sample test. You role is to regenerate python code that strictly adheres to the specifications, ensuring it reads input from standard input (stdin) and writes output to standard output (stdout). You will be given the Problem Description followed by the Comments and Reasons why your previous code fails. You will NOT return anything except for the program." ⊕ { Output from Code Reviewer Agent}

A.2.2 Workflows improved by agent evolution module

Task Parsing Workflow:
1. Task Parsing Agent: "
Genre: Science Fiction
Setting/Condition: A Floating City Above a Dying Earth
Creative Writing Prompt: In the year 2145, humanity has retreated to a sprawling floating city known as Aetheris, suspended high above the ravaged surface of a dying Earth. The city is powered by advanced technology that harnesses the energy of storms and the sun, but resources are dwindling, and the inhabitants are beginning to feel the strain of isolation.
As a member of the Council of Innovators, you are tasked with solving the city's most pressing problem: how to sustain life in Aetheris while finding a way to restore the Earth below. One day, you discover an ancient artifact buried in the archives of the city\u2014 a mysterious device that seems to pulse with energy and contains cryptic symbols.
Write a story exploring your character's journey as they decipher the artifact's secrets, navigate the political tensions within the council, and confront the ethical dilemmas of using the device. Will it lead to salvation for both the floating city and the Earth, or will it unleash unforeseen consequences?
Consider the implications of technology, the nature of survival, and the relationship be-

[5]Notably, workflows may contain loops, for example, a loop of Code Reviewer Agent and Code Rewriting Agent. However, we will not repeat the prompt for the same agent repetitively.

tween humanity and the environment as you craft your narrative." ⊕ { TASK PROMPT}
2. Code Generation Agent: "When faced with a mutation question like the one you've provided, individuals who excel in creative thinking typically approach it in several ways:
1). Understanding the Problem: They start by thoroughly understanding the existing code and its purpose. In this case, the code reads a number of test cases and computes the square of each number.
2). Identifying Opportunities for Improvement: They look for ways to enhance the functionality or efficiency of the code. For instance, they might consider:

- Adding error handling for invalid inputs.
- Allowing for different mathematical operations (not just squaring).
- Implementing a more flexible input method (e.g., reading from a file or allowing for different data types).

3). Exploring Alternative Solutions: Creative thinkers often brainstorm alternative approaches to solve the same problem. They might consider:

- Using a list comprehension for more concise code.
- Implementing a function to handle different operations based on user input.

4). Testing and Validation: They would think about how to validate the outputs and ensure the code behaves as expected under various conditions.
5). Refactoring for Clarity: They might refactor the code to improve readability and maintainability, such as by breaking it into smaller functions or adding comments.
6). Considering Edge Cases: They would think about edge cases, such as what happens if the input is zero, negative numbers, or non-integer values.
Here's an example of how the original code could be modified to incorporate some of these creative thinking strategies:

```
def square_number(number):
    """Returns the square of the given number."""
    return number ** 2

def main():
    import sys
    input = sys.stdin.read
    data = input().strip().splitlines()
    try:
        t = int(data[0])
    except ValueError:
        print("The first line must be an integer representing the number of test cases.")
        return
    results = []
    for i in range(1, t + 1):
        try:
            number = int(data[i])

            results.append(square_number(number))
        except ValueError:
            print(f"Invalid input at line {i + 1}: '{data[i]}'. Please enter an integer.")
            continue
    for result in results:
        print(result)

if __name__ == "__main__":
    main()
```

Key Changes Made:

- Function Extraction: The squaring logic is moved to a separate function for clarity.
- Error Handling: Added error handling for both the number of test cases and individual inputs.
- User Feedback: Provided feedback for invalid inputs to guide the user. This approach not only maintains the original functionality but also enhances the robustness and user-friendliness of the code.

" ⊕ { Output from Task Parsing Agent}
Code Rewriting Workflow:
1. Code Generation Agent: "Certainly! Here's a mutator prompt designed to inspire creativity and innovation:
—
Mutator Prompt: T̈he Elemental Fusion⁑
Imagine a world where the four classical elements2̆014Earth, Water, Air, and Fire2̆014can be combined in unconventional ways to create new entities, phenomena, or experiences. Your task is to explore the fusion of these elements and generate innovative concepts that transcend their traditional boundaries.
Instructions:

1. **Elemental Combinations:** Choose two or more elements from Earth, Water, Air, and Fire. Describe how they interact and what new properties or entities emerge from their fusion.
2. **Conceptual Expansion:** For each combination, think about how these new entities could influence the environment, society, or technology. What challenges or benefits do they bring?
3. **Unconventional Applications:** Consider how these fusions could be applied in real-world scenarios. This could include art, architecture, environmental solutions, or even new forms of entertainment.
4. **Visual Representation:** If possible, sketch or describe a visual representation of your fused element or concept. How does it look? What colors, shapes, or movements define it?
5. **Narrative Element:** Create a short story or scenario that features your new entity in action. How does it interact with the world around it? What conflicts or resolutions arise from its presence?

—

Example:

- **Combination:** Fire + Water
- **New Entity:** Steam Elementals
- **Description:** These beings are composed of swirling steam, capable of manipulating temperature and humidity. They can create fog to obscure vision or unleash scalding vapor as a defense mechanism.
- **Impact:** They could help regulate climate in urban areas, but their unpredictable nature might lead to sudden weather changes.
- **Application:** Used in art installations to create immersive environments that change with audience interaction.
- **Visual Representation:** Imagine a swirling mass of steam with glowing embers, shifting in color from blue to orange.
- **Narrative Element:** In a city plagued by drought, the Steam Elementals emerge to restore balance, but their presence stirs fear among the inhabitants who misunderstand their intentions.

—

Feel free to explore this prompt and let your imagination run wild! What new possibilities will you uncover?" ⊕ { TASK PROMPT}

2. Code Reviewer Agent:"

1) Problem Description: In a world where creativity knows no bounds, we seek to develop a mutator prompt that inspires individuals to break free from conventional thinking and explore innovative mutations in their ideas, projects, or artistic expressions. The goal is to create a prompt that encourages users to think divergently, combining unrelated concepts, altering existing frameworks, and generating novel solutions or artistic expressions. This prompt should serve as a catalyst for imagination, pushing the boundaries of what is considered possible.

2) Generated Code:

```python
import random

def generate_mutator_prompt():
    # Lists of unconventional concepts, actions, and mediums
    concepts = [
        \"time travel\", \"bioluminescence\", \"quantum entanglement\",
        \"dreamscapes\", \"mythical creatures\", \"urban legends\",
        \"chaos theory\", \"synesthesia\", \"artificial intelligence\",
        \"ancient civilizations\", \"futuristic technology\", \"empathy\"
    ]

    actions = [
        \"merge\", \"transform\", \"deconstruct\", \"reimagine\",
        \"collaborate with\", \"disrupt\", \"amplify\", \"diminish\",
        \"animate\", \"synthesize\", \"repurpose\", \"revolutionize\"
    ]

    mediums = [
        \"digital art\", \"poetry\", \"interactive installations\",
        \"performance art\", \"short stories\", \"music compositions\",
        \"sculptures\", \"video games\", \"fashion design\", \"cooking\"
    ]

    # Randomly select elements from each list
    concept = random.choice(concepts)
```

```
    action = random.choice(actions)
    medium = random.choice(mediums)

    # Construct the mutator prompt
    prompt = f\"Imagine a world where you can {action} {concept} through {medium}. \" \\
            f\"What new realities, stories, or experiences can you create?\"

    return prompt

# Example usage
if __name__ == \"__main__\":
    for _ in range(5):
        print(generate_mutator_prompt())
```

3) Explanation of the Code:
i. **Concepts, Actions, and Mediums**: The code defines three lists containing unconventional concepts, actions, and artistic mediums. These lists are designed to inspire creativity by combining elements that are typically not associated with each other.
ii. **Random Selection**: The ‘random.choice()‘ function is used to select one item from each list, ensuring that each prompt generated is unique and unpredictable.
iii. **Prompt Construction**: The selected elements are then combined into a coherent prompt that encourages the user to think creatively about how to merge the chosen concept with the selected action and medium.
iv. **Example Usage**: The code includes a loop that generates and prints five different mutator prompts, showcasing the variety of creative possibilities that can arise from this approach.
This mutator prompt serves as a powerful tool for artists, writers, and innovators, inviting them to explore new dimensions of creativity and push the boundaries of their imagination.
" ⊕ { Output from Code Generation Agent}
3. Code Rewriting Agent:"
Mutator Prompt: T̈he Infinite Canvas
Problem Description: Imagine a world where the laws of physics and logic are mere suggestions, and creativity reigns supreme. Your task is to create a Python program that generates a unique piece of art or a story based on a set of unconventional inputs. These inputs can be anything from random words, emotions, colors, or even abstract concepts. The program should utilize these inputs to produce an output that defies traditional boundaries, encouraging users to explore new dimensions of creativity.
Comments on Previous Code:
1. **Input Handling:** Ensure that the program can accept a diverse range of inputs, including strings, lists, and even user-generated content. This flexibility will allow for a broader spectrum of creative outputs.
2. **Randomization:** Incorporate elements of randomness to ensure that each execution of the program yields a different result. This could involve using random selections from lists of words, colors, or themes.
3. **Artistic Generation:** Depending on the chosen output type (art or story), utilize libraries such as ‘matplotlib‘ for visual art or ‘nltk‘ for text generation. The goal is to create something that feels alive and dynamic.
4. **User Interaction:** Consider adding an interactive component where users can tweak parameters or provide additional inputs mid-execution, allowing them to steer the creative process.
5. **Output Presentation:** Design the output to be visually appealing or narratively engaging. For art, this could mean creating a colorful canvas; for stories, it could involve formatting the text in a captivating way.
6. **Documentation:** Ensure that the code is well-commented and documented, explaining the purpose of each function and how users can modify inputs to explore different creative avenues.
—
Example Python Code:
Here’s a simple example to get you started with the concept:

```python
import random
import matplotlib.pyplot as plt
import numpy as np

def generate_random_words(num_words):
    words = [\"dream\", \"whisper\", \"ocean\", \"fire\", \"light\", \"shadow\", \"dance\", \"echo\"]
    return random.sample(words, num_words)

def create_art(words):
    plt.figure(figsize=(10, 10))
    for word in words:
        x = np.random.rand(10)
        y = np.random.rand(10)
        plt.scatter(x, y, label=word, alpha=0.5)
    plt.title(\"Art Inspired by Words\")
    plt.legend()
```

```
    plt.show()

def main():
    num_words = int(input(\"How many random words would you like to generate? \"))
    random_words = generate_random_words(num_words)
    print(\"Generated Words:\", random_words)
    create_art(random_words)

if __name__ == \"__main__\":
    main()
```

This code generates a set of random words and uses them to create a scatter plot, illustrating the concept of transforming abstract ideas into visual art. Feel free to expand upon this foundation, adding more complexity and creativity to the outputs!" ⊕ { Output from Code Reviewer Agent}

A.3 Examples of Failed Python Workflows

Reason: The ending agent of the workflow below is a code reviewer agent instead of a coding agent; hence, the final output cannot solve the coding task.

```
steps = [
    {'name': 'task_parsing_agent', 'args': ['task_description'], 'output': 'parsed_task'},
    {'name': 'task_refinement_agent', 'args': ['task_description', 'parsed_task'], 'output': 'refined_task'},
    {'name': 'code_generation_agent', 'args': ['refined_task'], 'output': 'generated_code'},
    {'name': 'code_reviewer_agent', 'args': ['refined_task', 'generated_code'], 'output': 'review_comments'}
    ]
```

Reason: The input of the fortune telling agent does not match the output of the code generation agent; hence the fortune telling agent cannot fetch its input.

```
steps = [
    {'name': 'task_parsing_agent', 'args': ['task_description'], 'output': 'parsed_task'},
    {'name': 'task_refinement_agent', 'args': ['task_description', 'parsed_task'], 'output': 'refined_task'},
    {'name': 'code_generation_agent', 'args': ['refined_task'], 'output': 'generated_code'},
    {'name': 'fortune_telling_agent', 'args':['refined_code'], 'output': 'fortune_prediction'},
    {'name': 'code_reviewer_agent', 'args': ['refined_task', 'generated_code'], 'output': 'review_comments'},
    {'name': 'code_refinement_agent', 'args': ['refined_task', 'review_comments'], 'output': 'refined_code'},
]
```

Total Tokens	HumanEval	MBPP	LCB
GPT-4o mini	295,200	721,336	1,208,023
Task Parsing Workflow	575,395	989,457	1,937,632
Code Rewriting Workflow	889,467	1,459,457	2,445,023

Table 4: The total tokens of the backbone model GPT-4o mini and SEW generated workflows across three datasets. The total tokens include the input/output tokens on both the validation and testing sets.

A.4 Token costs

In this section, we report the token costs of the single agent (only use the backbone model), the Task Parsing Workflow and the Code Rewriting Workflow generated by SEW (see Table 4). From this table, we can learn that LiveCodeBench is a much more challenges benchmark than HumanEval and MBPP because the tokens cost per instance for LCB is much larger than that of MBPP and HumanEval. In addition, the tokens cost for both workflows is larger than that for GPT-4o mini since workflows involve multiple agents taking input and output concurrently, where the Code Rewriting Workflow costs especially more tokens because it may take multiple rounds for the generator agent to produce codes passing the reviewer agent.