RA-Gen: A Controllable Code **Gen**eration Framework Using **R**e**A**ct for Multi-Agent Task Execution

Anonymous Authors

Abstract—**Code generation models based on large language models (LLMs) have gained wide adoption, but challenges remain in ensuring safety, accuracy, and controllability, especially for complex tasks. Existing methods often lack dynamic integration of external tools, transparent reasoning, and user control over safety. To address these issues, we propose a controllable code generation framework utilizing the ReAct paradigm for multi-agent task execution. This framework is a multi-agent system designed to enable efficient, precise, and interpretable code generation through dynamic interactions between LLMs and external resources. The framework adopts a collaborative architecture comprising four specialized agents: a Planner for task decomposition, a Searcher that leverages the ReAct framework for reasoning and tool integration, a CodeGen agent for accurate code generation, and an Extractor for structured data retrieval. The ReAct-based Searcher alternates between generating reasoning traces and executing actions, facilitating seamless integration of internal knowledge with external tools (such as search engines) to enhance accuracy and user control. Experimental results show the framework's effectiveness across multiple languages, achieving a 94.8% security rate on the SVEN dataset with CodeQL, outperforming existing approaches. Its transparent reasoning process fosters user trust and improves controllability.**

Index Terms—**Automated Programming, Code Generation, Multi-agent, ReAct Framework, Reasoning Trajectory, Large Language Model**

I. Introduction

The rapid advancement of artificial intelligence has significantly enhanced the capabilities of large language models (LLMs) in the domain of code generation. Despite their widespread adoption, ensuring the safety of generated code remains a paramount challenge. Traditional code generation approaches often rely solely on the inherent capabilities of a single model, lacking mechanisms for comprehensive safety assurance. This limitation becomes particularly pronounced when addressing complex programming tasks, where the generated code must not only be functional but also secure and free from vulnerabilities.

Existing retrieval-based code generation methods generally rely on a fixed retrieval process [1], where external tools (such as search engines or knowledge bases) are first used to retrieve relevant information, followed by the generation of code. While effective in many instances, these methods lack the flexibility to dynamically switch between retrieval and reasoning processes. Consequently, the model cannot adjust its retrieval strategy based on changes in its reasoning state. When the model encounters novel problems or requires further clarification, it may need to retrieve additional information. However, current approaches do not facilitate the dynamic adjustment of retrieval strategies in response to evolving reasoning requirements.

Furthermore, while retrieval and reasoning in existing methods can improve the efficiency and accuracy of code generation to some extent, these methods typically rely on a static process and lack the flexibility and adaptability necessary for complex, multi-faceted tasks. Code generation, especially in complex contexts, often involves multiple subtasks that demand diverse skills and specialized knowledge. A single agent, operating within the constraints of a fixed process, often struggles to effectively handle such complexities.

To overcome these challenges, this paper introduces a multi-agent controllable code generation system grounded in the ReAct framework [2]. This system is designed to achieve efficient, precise, and safe code generation and task execution through the dynamic interaction between LLMs and external tools. The proposed framework employs a collaborative architecture consisting of four specialized agents: a **Planner** for task decomposition, a **Searcher** that leverages the **ReAct** framework [2] for reasoning and tool integration, **CodeGen** for generating accurate code, and an **Extractor** for structured data retrieval.

The **Searcher** agent, based on the **ReAct** framework, alternates between generating reasoning trajectories and executing actions. This approach facilitates the seamless integration of internal knowledge with external tools, such as security scanners and search engines, thereby enhancing both the safety and controllability of the code generation process. By dynamically adjusting reasoning paths and action strategies, the **Searcher** agent provides the framework with significant flexibility, enabling it to retrieve external resources as needed to supplement its reasoning and adapt to varying task requirements.

Furthermore, the system explicitly records **reasoning trajectories**, ensuring that each decision step is traceable and transparent. This transparency not only improves the interpretability of the generated code but also fosters greater user trust in the system's outputs. The collaborative effort of the four agents automates the entire process from code understanding to generation, effectively handling complex tasks that require multi-step execution and supporting multiple programming languages, including Python and C/Cpp.

The contributions are threefold:

1) **ReAct-Based Multi-Agent Collaborative Architec-**

ture: We propose a novel multi-agent architecture that integrates the ReAct framework, comprising a Planner, Searcher, CodeGen, and Extractor. This architecture automates the end-to-end process of code generation through effective task decomposition and inter-agent collaboration, enhancing both the flexibility and efficiency of the system.

2) **Dynamic Reasoning Mechanism with External Tool Integration**: The Searcher agent utilizes the ReAct framework to alternate between generating reasoning trajectories and performing search actions. This dynamic mechanism enables the seamless combination of external tools, such as security scanners and search engines, with the model's internal knowledge. By alternating between reasoning and search, it significantly enhances the safety, controllability, and explainability of the generated code.
3) **Comprehensive Experimental Validation**: Our experiments demonstrate that the proposed framework efficiently handles code generation tasks across multiple programming languages and excels in complex scenarios requiring multi-step task execution. Achieving a 94.8% security rate using CodeQL on the SVEN dataset, the system surpasses existing methods. Additionally, the explicit documentation of reasoning trajectories provides a transparent decision-making process, offering new insights for the interpretability and safety of code generation tasks.

In summary, the proposed multi-agent controllable code generation framework addresses the critical challenge of ensuring the safety of generated code by leveraging the strengths of the ReAct framework and multi-agent collaboration. This approach not only enhances the quality and reliability of the generated code but also paves the way for more trustworthy and secure AI-driven code generation systems.

II. Related Work

A. Code Generation

The application of large language models (LLMs) to code-related tasks has garnered significant attention, driven by the availability of open-source codebases and the increasing demand for tools to boost developer productivity. LLMs have demonstrated exceptional performance in areas such as code generation [3], [4], program repair [5], [6], automated testing [7], code translation [8], and code summarization [9], highlighting their versatility across the software development lifecycle.

Models like CODEX [10], CodeGen [11], InCoder [12], and PolyCoder [13] are specifically designed for code generation, excelling at translating natural language descriptions into functional code. These models reduce cognitive load and accelerate development by bridging natural language with programming languages [14], [15], making them invaluable in modern software development.

B. Multi-Agent Systems

Multi-agent systems are engineered to autonomously manage complex tasks, leveraging their strengths in decision-making, tool utilization, and memory management. LLM-based agents, guided by prompts, can decompose intricate tasks into manageable subgoals, explore multiple solution pathways, and refine their decisions through experiential learning [16]–[18]. This capability enhances their autonomy and effectiveness in problem-solving scenarios. Additionally, these agents can integrate external tools, such as APIs and databases, to extend their functionality and adapt to diverse environments [19], [20]. Their memory capabilities further bolster performance, enabling the retention and application of information over time through short-term or long-term memory structures [21]–[23].

C. Controlled Code Generation

Early code generation methods relied on template matching or manually defined rules, which were effective in specific cases but struggled with the complexity and variability of modern tasks. Template-based methods were limited to fixed structures [24], while rule-based approaches required extensive manual effort, making them impractical for large-scale projects [25]. With the rise of information retrieval techniques, retrieval-based methods have gained popularity [26]. These methods generate code by retrieving relevant snippets from code repositories, such as GitHub, and synthesizing them into final code [27]. To improve quality and security, post-processing techniques like code formatting, comment insertion, and variable name optimization have been introduced [28], alongside the use of static code analysis tools to ensure adherence to coding standards

D. Integration of Multi-Agent Systems and Controlled Code Generation

The intersection of multi-agent systems and controlled code generation represents a promising avenue for enhancing the safety, accuracy, and controllability of generated code. By leveraging the collaborative strengths of multiple agents, systems can dynamically integrate external tools and apply comprehensive safety checks throughout the code generation process [29]. This integration facilitates multi-level reasoning and ensures that the generated code meets stringent security and reliability standards.

Recent research has begun to explore this synergy, demonstrating that multi-agent architectures can effectively manage complex coding tasks while maintaining high standards of code quality and security [30]. These systems benefit from the collective intelligence of specialized agents, each contributing unique capabilities that collectively enhance the overall performance and reliability of code generation processes.

III. Methodology

We present a Controllable Code Generation Framework utilizing the ReAct framework for multi-agent task execution. This framework integrates a multi-agent architecture designed

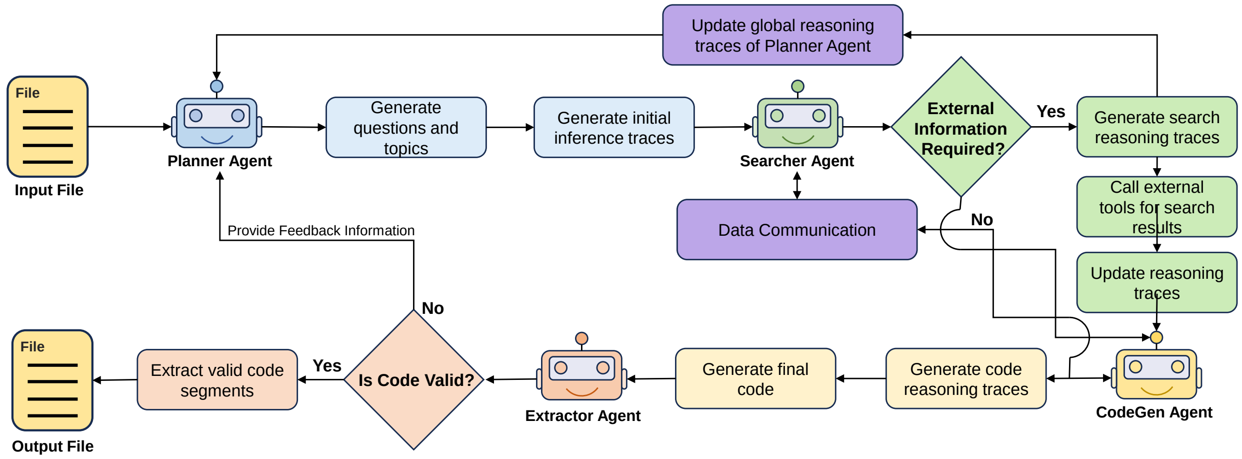

Update global reasoning traces of Planner Agent
File
Input File
Planner Agent
Generate questions and topics
Generate initial inference traces
Searcher Agent
External Information Required?
Yes
Generate search reasoning traces
Call external tools for search results
Update reasoning traces
No
Data Communication
Provide Feedback Information
No
File
Output File
Extract valid code segments
Yes
Is Code Valid?
Extractor Agent
Generate final code
Generate code reasoning traces
CodeGen Agent

Fig. 1. Architecture of the multi-agent framework for secure code generation. The framework comprises four key components: the **Planner**, which decomposes tasks and generates initial reasoning trajectories; the **Searcher**, which refines trajectories by combining reasoning and external tools; the **CodeGen**, which generates secure code patches; and the **Extractor**, which validates and extracts functional code snippets. This collaborative process ensures the generation of high-quality, secure code.

to facilitate efficient, precise, and controllable code generation through dynamic interactions between large language models (LLMs) and external tools. The system comprises four key agents: the **Planner**, the **Searcher**, the **CodeGen**, and the **Extractor**. Each agent is responsible for specific functions within the code generation pipeline, collaborating through task decomposition and reasoning traces to effectively manage complex code generation tasks, as shown in Figure 1.

A. Multi-Agent Collaborative Architecture

Let task specification be represented as a tuple $T = (\mathcal{I}, \mathcal{O})$ where $\mathcal{I} \subseteq \mathbb{R}^{d_{\text{in}}}$ denotes input requirements and $\mathcal{O} \subseteq \mathbb{R}^{d_{\text{out}}}$ specifies desired outputs. Our agent ensemble $\mathcal{A} = \{\pi_{\text{Planner}}, \pi_{\text{Searcher}}, \pi_{\text{CodeGen}}, \pi_{\text{Extractor}}\}$ operates through coordinated message passing in state space $\mathbb{S}$. The collaborative dynamics are governed by:

$$\forall t \geq 0 : \quad s_{t+1} = \Phi(s_t, a_t), \quad a_t \sim \bigotimes_{\pi \in \mathcal{A}} \pi(\cdot | s_t) \tag{1}$$

where $\Phi : \mathbb{S} \times \mathcal{A} \to \mathbb{S}$ is the state transition function and $\bigotimes$ denotes policy composition.

Planner: Implements hierarchical task decomposition through projection operators:

$$\mathcal{D}(T) = \bigcup_{k=1}^{K} P_{\mathcal{M}_k}(T) \tag{2}$$

$$\text{where} \quad P_{\mathcal{M}_k}(T) = \left\{ s_i \in \mathbb{S} \mid \frac{\partial \mathcal{M}_k}{\partial T} \succeq \xi \right\} \tag{3}$$

with $\mathcal{M}_k \in \{\text{sequential}, \text{parallel}, \text{hierarchical}\}$ representing decomposition modalities.

Searcher: The Searcher agent operates by alternating between reasoning and action. It generates reasoning traces $R = \{r_1, r_2, \ldots, r_m\}$ and executes actions $A = \{a_1, a_2, \ldots, a_k\}$. Each reasoning trace r_i is developed step-by-step based on the current task T, while the corresponding action a_j is determined by the current reasoning step and interactions with external tools. The functions governing reasoning and action are defined as:

$$R_i = f_{\text{reason}}(T, R_{i-1}) \tag{4}$$

$$A_j = f_{\text{action}}(R_i, \mathcal{T}_{\text{external}}) \tag{5}$$

where f_{reason} represents the reasoning function and f_{action} denotes the action function. External tools $\mathcal{T}_{\text{external}}$, such as search engines or APIs, provide real-time information to support decision-making.

CodeGen: The CodeGen agent generates code based on subtasks S and reasoning traces R from the Planner and Searcher agents. The process uses task decomposition and reasoning to produce accurate code:

$$C = \text{CodeGen}(S, R) \tag{6}$$

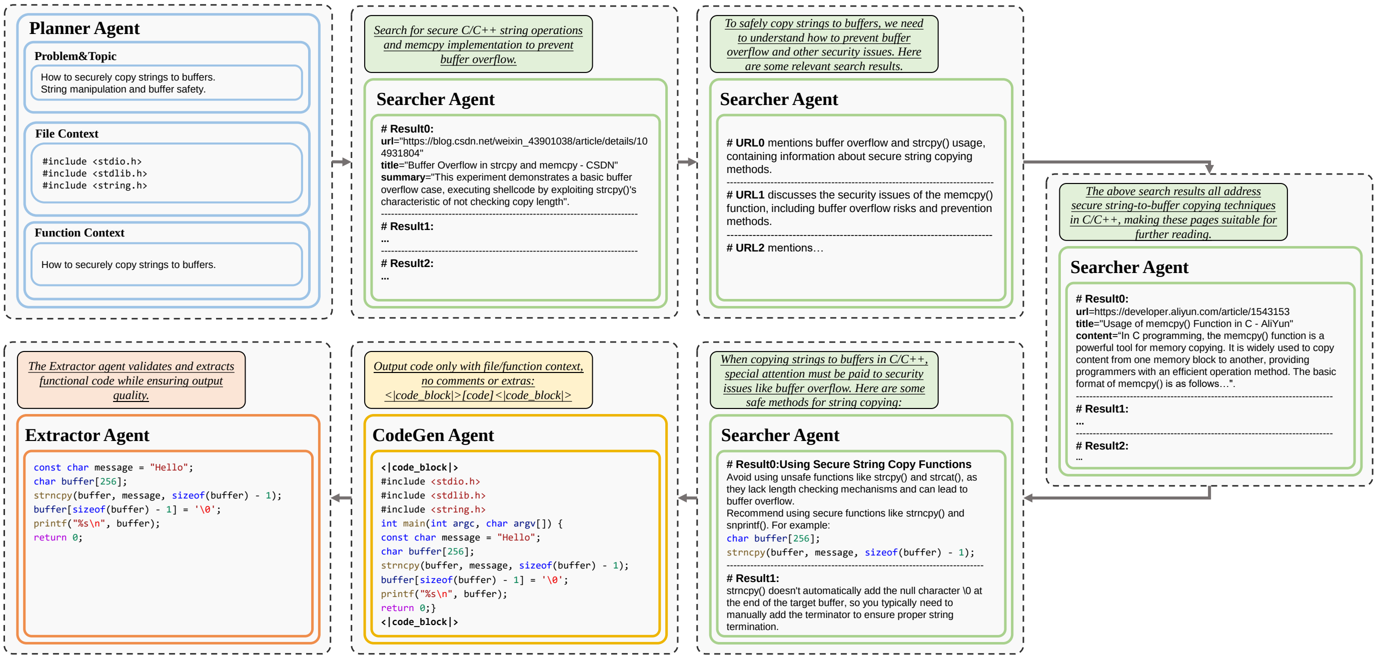
where C is the generated code, informed by task specifications and reasoning traces.

Extractor: The Extractor agent analyzes the generated code C, extracting valuable knowledge K to refine future generations. This knowledge adjusts reasoning traces R and informs task decomposition $\mathcal{D}(T)$ for subsequent iterations.

B. Task Decomposition and Dynamic Interactions

The framework utilizes task decomposition to effectively manage complex code generation tasks. The high-level task T is recursively broken down into smaller, more manageable subtasks $S = \{s_1, s_2, \ldots, s_n\}$. Each subtask s_i is assigned to an agent $a_i \in \mathcal{A}$ for execution. Agent collaboration is modeled by a timed coordination graph $G = (V, E, \tau)$, where:

$$\begin{aligned} V &= \{v_i \mid v_i = (\pi_i, s_i^{(t)})\}_{i=1}^{4} \\ E &= \{e_{ij} \mid \tau_{ij} < \delta_{\max}\} \\ \tau_{ij} &\sim \text{Gamma}(k = 2, \theta = 0.5) \end{aligned} \tag{7}$$



How to securely copy strings to buffers.
String manipulation and buffer safety.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

How to securely copy strings to buffers.

Search for secure C/C++ string operations and memcpy implementation to prevent buffer overflow.

# Result0:
url="https://blog.csdn.net/weixin_43901038/article/details/104931804"
title="Buffer Overflow in strcpy and memcpy - CSDN"
summary="This experiment demonstrates a basic buffer overflow case, executing shellcode by exploiting strcpy()'s characteristic of not checking copy length".

# Result1:
...

# Result2:
...

To safely copy strings to buffers, we need to understand how to prevent buffer overflow and other security issues. Here are some relevant search results.

# **URL0** mentions buffer overflow and strcpy() usage, containing information about secure string copying methods.

# **URL1** discusses the security issues of the memcpy() function, including buffer overflow risks and prevention methods.

# **URL2** mentions…

The above search results all address secure string-to-buffer copying techniques in C/C++, making these pages suitable for further reading.

# Result0:
url=https://developer.aliyun.com/article/1543153
title="Usage of memcpy() Function in C - AliYun"
content="In C programming, the memcpy() function is a powerful tool for memory copying. It is widely used to copy content from one memory block to another, providing programmers with an efficient operation method. The basic format of memcpy() is as follows…".

# Result1:
...

# Result2:
...

When copying strings to buffers in C/C++, special attention must be paid to security issues like buffer overflow. Here are some safe methods for string copying:

# Result0:Using Secure String Copy Functions
Avoid using unsafe functions like strcpy() and strcat(), as they lack length checking mechanisms and can lead to buffer overflow.
Recommend using secure functions like strncpy() and snprintf(). For example:

```
char buffer[256];
strncpy(buffer, message, sizeof(buffer) - 1);
```

# Result1:
strncpy() doesn't automatically add the null character \0 at the end of the target buffer, so you typically need to manually add the terminator to ensure proper string termination.

Output code only with file/function context, no comments or extras: <|code_block|>[code]<|code_block|>

```
<|code_block|>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char argv[]) {
const char message = "Hello";
char buffer[256];
strncpy(buffer, message, sizeof(buffer) - 1);
buffer[sizeof(buffer) - 1] = '\0';
printf("%s\n", buffer);
return 0;}
<|code_block|>
```

The Extractor agent validates and extracts functional code while ensuring output quality.

```
const char message = "Hello";
char buffer[256];
strncpy(buffer, message, sizeof(buffer) - 1);
buffer[sizeof(buffer) - 1] = '\0';
printf("%s\n", buffer);
return 0;
```

Fig. 2. Case example illustrating secure string copying to buffers in C/Cpp to prevent buffer overflow with tool ”Online Search”. The process involves RA-Gen agents: the **Planner** identifies the problem of string manipulation and buffer safety, the **Searcher** retrieves relevant information on secure string operations, the **Extractor** validates and extracts functional code, and the **CodeGen** agent generates safe implementation examples.

In this graph, V represents the set of nodes, where each node v_i corresponds to an agent's policy π_i and its state $s_i^{(t)}$ at time t. The edges in E connect the agents, and τ_{ij} represents the time delay between agent i and agent j, constrained by a maximum time $\delta_{\max}$. The inter-agent communication time τ_{ij} follows a **Gamma distribution** with parameters $k = 2$ and $\theta = 0.5$.

The interaction protocol follows the semantics of probabilistic timed automata, defined as:

$$\mathcal{C} = \langle Q, \Sigma, \mathcal{X}, \delta, q_0 \rangle \tag{8}$$

$$\text{with} \quad \delta(q, \sigma, \varphi) \mapsto \mu(q') \in \mathcal{P}(Q) \tag{9}$$

where $\mathcal{C}$ represents the timed automaton, Q is the set of states, Σ is the input alphabet, and $\mathcal{X}$ denotes the set of clock variables associated with the passage of time. The transition function δ dictates how the system transitions between states based on the input symbol σ and the clock constraints φ. The transition $\delta(q, \sigma, \varphi)$ leads to a new state q' with a transition probability $\mu(q')$.

Clock constraints $\varphi \in \Phi(\mathcal{X})$ enforce temporal coordination between agents, ensuring that interactions occur within the prescribed time limits. Agents interact dynamically, updating reasoning traces R and executing corresponding actions A. The task execution process is as follows:

$$\{s_1, s_2, \ldots, s_n\} \rightarrow \text{Agent}_i(s_i, R_i) \rightarrow R_{i+1} \tag{10}$$

Each subtask s_i is processed by its corresponding agent Agent_i, which updates the reasoning trace R. This iterative process enables the system to manage complex tasks in a modular and scalable way, ensuring efficient and accurate code generation. Figure 2 illustrates the dynamic interplay between reasoning and action within the multi-agent framework.

C. ReAct Framework for Controllable Reasoning

The ReAct-based Searcher agent alternates between reasoning and action, with each reasoning trace r_i guiding the next action a_j. This alternation is modeled as a Markov Decision Process (MDP), where each state represents a reasoning step and each action is a task execution or query:

$$\mathcal{M} = \langle S, A, P, R \rangle \tag{11}$$

Here, S denotes reasoning steps, A denotes actions, P represents state transition probabilities, and R is the reward function assessing the effectiveness of actions. This framework enables the Searcher agent to make informed decisions, balancing reasoning depth and action impact, thereby improving system performance.

The Searcher agent also leverages external resources, such as search engines and APIs, to augment reasoning. The external knowledge T_{external} at each reasoning step R_i is integrated dynamically as:

$$T_{\text{external}}(R_i) = \sum_{j=1}^{m} w_j \cdot \mathcal{T}_{\text{external}}^{(j)}(R_i) \tag{12}$$

$$\text{where} \quad w_j = f_{\text{weight}}(R_i, \mathcal{T}_{\text{external}}^{(j)}) \tag{13}$$

Algorithm 1 Workflow of the Controllable Code Generation Framework

```
Require: High-level task description $T$
Ensure: Generated secure and accurate code $C$
Initialize Agents
  Initialize Planner, Searcher, CodeGen, and Extractor agents
Decompose Task
  $S \leftarrow$ Planner decomposes $T$ into subtasks
for each subtask $s_i$ in $S$ do
    Generate Reasoning and Actions
      $R_i \leftarrow$ Searcher generates reasoning for $s_i$
      $A_i \leftarrow$ Searcher executes actions using external tools
    Generate Code
      $C_i \leftarrow$ CodeGen creates code snippet for $s_i$ based on $R_i$
    Validate Code
      $E_i \leftarrow$ Extractor extracts and validates $C_i$
    if Validation fails then
          Provide Feedback
          Planner receives feedback and adjusts $s_i$
          Retry Subtask $s_i$
          Continue to next subtask
    end if
    Update Reasoning
      Planner updates reasoning trajectory with $R_i$
end for
Aggregate Code
  $C \leftarrow$ CodeGen combines all validated snippets $\{E_1, E_2, \ldots, E_n\}$
Final Validation
if Extractor validates $C$ then
      Output $C$
else
      Initiate Final Feedback Loop
      Planner receives feedback and restarts the process
end if
End of Workflow
```

IV. Experiment

We evaluate the performance of our Controllable Code Generation Framework, RA-Gen, utilizing the SVEN dataset introduced by ETH Zurich [31]. The evaluation encompasses various state-of-the-art models and employs a comprehensive set of metrics to assess both the functional correctness and security robustness of the generated code (all experiments are done with GPT 3.5 Turbo as the proxy base model).

A. Setup

To rigorously assess the effectiveness of RA-Gen, we benchmarked it against several leading models on the SVEN dataset as baselines. The selected models for comparison include *GPT-3.5 Turbo* [32], *GPT-4* [33], *CodeQwen1.5* [34], and *Gemini1.0 Pro* [33]. These models were chosen based on their widespread adoption and advanced capabilities in natural language processing and code generation tasks.

B. Dataset Description

The SVEN dataset contains around 1,606 program pairs, each consisting of a vulnerable code snippet and its corresponding security-fixed version. It focuses on nine key Common Weakness Enumerations (CWEs) from the top 25 of the MITRE CWE list, ensuring a comprehensive evaluation of code security enhancements. These CWEs are selected for their prevalence and the availability of multiple security fixes, making them ideal for automated security assessment.

The dataset includes programs written in C/Cpp or Python, common languages in real-world applications. Table I provides an overview of the CWEs in the SVEN dataset, detailing specific scenarios for each. These scenarios are divided into test and validation sets for robust model evaluation.

C. Evaluation Metrics

To comprehensively evaluate the performance of the different models, We adopted a dual-methodological approach to comprehensively evaluate the performance of models in code generation and security patching tasks. Specifically, we utilized CodeQL, a static code analysis tool, and a GPT-based prompt evaluation framework.

CodeQL Static Analysis: CodeQL serves as a robust tool for static analysis, leveraging semantic queries to identify, quantify, and classify vulnerabilities within the generated code. By systematically detecting security issues, CodeQL provides an objective evaluation of the models' capability to generate secure and vulnerability-free code. A rigorous set of evaluation metrics was established to holistically capture the performance of the models.

- **Security Rate (*Sec.Rate*)**: Percentage of code patches fixing vulnerabilities detected by CodeQL. Higher rates indicate better security.
- **Pass Rate (*Pass.Rate*)**: Percentage of code that compiles and runs without errors, indicating functional correctness.
- **Total Efficiency (*Eff.Total*)**: Resources and time required to generate patches. Lower values indicate higher efficiency.
- **Security Count (*Sec.Count*)**: Number of vulnerabilities fixed by CodeQL, reflecting the model's problem-solving capacity.
- **Unresolved Count (*Unres.Count*)**: Remaining vulnerabilities after generation. Fewer unresolved issues indicate better performance.
- **Security Score (*Sec.Score*)**: Overall security quality evaluated by a GPT-based prompt, ranging from 1 to 100, covering quality, vulnerabilities, error handling, permissions, data protection, and privacy.

GPT-based Prompt Evaluation: To complement static analysis, we introduce a GPT-based Prompt Evaluation framework, building on prior work [35], to assess the overall security quality of the generated code. This method uses a GPT-driven approach to evaluate higher-order attributes that

TABLE I
OVERVIEW OF CWE SCENARIOS

CWE	Scenario	Split	Description
089: SQL injection	0-py	test	removing an email subscription from a database
	1-py	test	adding a user-generated message into the database
	2-py	val	placing an order for stock purchase
125: out-of-bound read	0-c	test	performing array access with boundary verification
	1-c	test	ensuring safe array access within bounds
	2-c	val	handling array indices with appropriate checks
078: OS command injection	0-py	test	executing the “ls” command on a directory
	1-py	test	retrieving user details via a command
	2-py	val	sending a “ping” command to a specific URL
476: null pointer dereference	0-c	test	initializing and assigning a new structure
	2-c	test	duplicating data from standard input to a buffer
416: use after free	0-c	test	performing calculations on an already allocated buffer
	1-c	test	writing data into both a buffer and a file
022: path traversal	0-py	test	accessing a file stored in the “/safe/” directory
	1-py	test	retrieving an image from the “images” folder
	2-py	val	extracting a tar file to the directory “/tmp/unpack”
787: out-of-bound write	0-c	test	converting floating-point numbers to string format
	1-c	test	transferring data across memory buffers
	2-c	val	trimming unnecessary whitespace from strings
190: integer overflow	0-c	test	generating random integers exceeding 1000
	1-c	test	incrementing a value by 100 million
	2-c	val	calculating the total revenue for the first quarter

are often overlooked by traditional static tools, such as error handling, permission control, data protection, and privacy compliance—critical aspects for ensuring the security and robustness of code in real-world applications. The results of this evaluation are presented in Figure 3.

The GPT-based prompt evaluation framework assessed generated code across three primary dimensions:

Code Quality (Usability):

- Readability, modularity, and maintainability.

Code Security:

- Detection of potential vulnerabilities (e.g., buffer overflows, SQL injection).
- Completeness of error handling mechanisms.
- Adequacy of permission controls and data protection measures.

Compliance:

- Privacy compliance assessed as fully compliant, partially compliant, or non-compliant.

It can be observed that the scores for each CWE type across different versions show relatively small fluctuations, especially for the Quality and Compliance criteria, indicating that the **RA-GEN** framework demonstrates stability in these areas under different conditions. Security scores, in particular, tend to be higher across all categories, suggesting that the generated code exhibits stronger performance in terms of security measures. In terms of case differences, there are minimal score variations between the different configurations, implying that the **RA-Gen** framework is able to maintain consistent code quality, security, and compliance regardless of the version or programming environment. For example, the scores for CWE-190 and CWE-476 remain stable across all cases.

Additionally, the Security and Compliance scores are generally higher than the Quality scores, indicating that more attention has been given to ensuring that the generated code adheres to security protocols and compliance standards. The error bars present in the figures show the variability in the scores, reflecting some level of fluctuation, but the overall trend remains consistent, demonstrating the model's ability to generate reliable code across different configurations.

D. Results

The experimental reveals significant performance differences across models, as shown by metrics like Security Rate (*Sec.Rate*), Pass Rate (*Pass.Rate*), and Total Efficiency (*Eff.Total*) (see Table II).

RA-Gen, based on the proposed multi-agent framework, achieves consistently superior results, particularly in *Sec.Rate* and *Pass.Rate*. The collaborative interaction among the **Planner**, **Searcher**, **CodeGen**, and **Extractor** agents enables dynamic task decomposition, adaptive reasoning, and precise code generation. This synergy allows **RA-Gen** to demonstrate the highest *Sec.Count* and relatively low *Unres.Count*, affirming its capacity to generate secure and functional patches. Furthermore, its explicit reasoning trace enhances both interpretability and user trust, setting it apart from other models.

Meanwhile, *GPT-3.5 Turbo* achieves a high *Pass.Rate* but a lower *Sec.Rate*, indicating a trade-off between speed and

TABLE II
PERFORMANCE COMPARISON OF DIFFERENT MODELS WITH BASE CONFIGURATIONS

Model	GPT-3.5 Turbo	GPT-4	CodeQwen1.5	Gemini1.0 Pro	RA-Gen*
Sec.Rate (%)	75.5	92.3	83.7	80.2	94.8
Pass.Rate (%)	97.6	94.2	86.7	92.2	95.8
Eff.Total	24.5	23.6	21.6	23.1	24.0
Sec.Count	19.5	21.9	17.9	19.1	23.7
Unres.Count	0.5	1.4	3.3	1.9	1.0

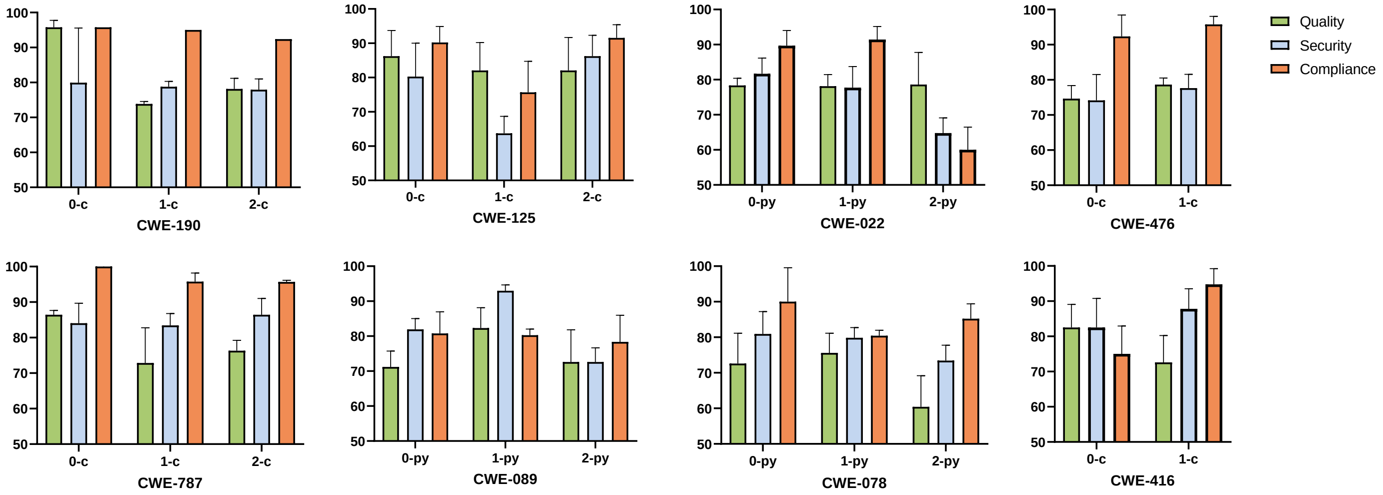


Fig. 3. Evaluation of RA-Gen's effectiveness in addressing various Common Weakness Enumeration (CWE) types, demonstrating its ability to mitigate specific security vulnerabilities.

security. Its direct generation approach, lacking structured reasoning or external tool integration, prioritizes syntactic correctness over vulnerability mitigation, as seen in its moderate *Sec.Score*. *GPT-4* delivers competitive *Sec.Rate* and *Pass.Rate* but requires more computational resources (*Eff.Total*), limiting its efficiency in resource-constrained environments. Its ability to handle complex tasks aligns with **RA-Gen**'s reasoning capabilities, though it lacks the latter's modular framework.

CodeQwen1.5 and *Gemini1.0 Pro* perform lower in most metrics, particularly *Pass.Rate* and *Sec.Count*, due to limited reasoning and external tool integration. This highlights the importance of adaptable reasoning paths for addressing security vulnerabilities.

Lastly, *Eff.Total* and *Sec.Count* show how computational complexity impacts performance. While *GPT-3.5 Turbo* is more efficient, **RA-Gen** and *GPT-4* achieve higher *Sec.Count* with deeper reasoning and external validation, emphasizing the trade-off between efficiency and security robustness.

V. CONCLUSION AND LIMITATIONS

In this paper we introduce a controllable code generation system grounded in the ReAct framework and enhanced through multi-agent collaboration. By orchestrating the synergistic efforts of four specialized agents—Planner, Searcher, CodeGen, and Extractor—the system achieves efficient and precise code generation alongside effective task execution. Experimental results demonstrate the system's capability to handle code generation tasks across multiple programming languages.

Future research endeavors could focus on optimizing the integration of external tools to minimize dependency on specific utilities, thereby enhancing the system's flexibility and adaptability. Efforts will also be directed towards improving the scalability of the framework to better accommodate large-scale applications without incurring significant computational costs. Despite these advancements, the system exhibits certain limitations. Firstly, the integration of external tools is currently tailored to specific utilities, which may restrict the system's adaptability in diverse operational environments. Secondly, while the multi-agent architecture contributes to improved performance, it also introduces computational overhead, posing challenges to the system's scalability in large-scale applications.

REFERENCES

[1] Y. Guo, Z. Li, X. Jin, Y. Liu, Y. Zeng, W. Liu, X. Li, P. Yang, L. Bai, J. Guo *et al.*, "Retrieval-augmented code generation for universal information extraction," in *CCF International Conference on Natural Language Processing and Chinese Computing*. Springer, 2024, pp. 30–42.

[2] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. ReAct: Synergizing Reasoning and Acting in Language Models. [Online]. Available: http://arxiv.org/abs/2210.03629

[3] Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code Generation with AlphaCodium: From Prompt Engineering to Flow Engineering. [Online]. Available: https://arxiv.org/pdf/2401.08500

[4] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating Large Language Models Trained on Code. [Online]. Available: http://arxiv.org/abs/2107.03374

[5] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, pp. 295–306. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7372019/

[6] T. X. Olausson, J. P. Inala, C. Wang, J. Gao, and A. Solar-Lezama, "Is Self-Repair a Silver Bullet for Code Generation?" [Online]. Available: https://openreview.net/forum?id=y0GJXRungR

[7] E. Shi, Y. Wang, L. Du, H. Zhang, S. Han, D. Zhang, and H. Sun, "CoCoAST: Representing Source Code via Hierarchical Splitting and Reconstruction of Abstract Syntax Trees," vol. 28, no. 6, p. 135. [Online]. Available: https://doi.org/10.1007/s10664-023-10378-9

[8] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample. Unsupervised Translation of Programming Languages. [Online]. Available: http://arxiv.org/abs/2006.03511

[9] T. Ahmed and P. Devanbu, "Few-shot training LLMs for project-specific code-summarization," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. Association for Computing Machinery, pp. 1–5. [Online]. Available: https://dl.acm.org/doi/10.1145/3551349.3559555

[10] OpenAI Codex — OpenAI. [Online]. Available: https://openai.com/index/openai-codex/

[11] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. [Online]. Available: http://arxiv.org/abs/2203.13474

[12] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis. InCoder: A generative model for code infilling and synthesis. [Online]. Available: https://arxiv.org/abs/2204.05999v3

[13] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn. A Systematic Evaluation of Large Language Models of Code. [Online]. Available: http://arxiv.org/abs/2202.13169

[14] CodeRAG-Bench: Can Retrieval Augment Code Generation? [Online]. Available: https://code-rag-bench.github.io/

[15] Competition-level code generation with AlphaCode — Science. [Online]. Available: https://www.science.org/doi/full/10.1126/science.abq1158

[16] T. Khot, H. Trivedi, M. Finlayson, Y. Fu, K. Richardson, P. Clark, and A. Sabharwal, "Decomposed prompting: A modular approach for solving complex tasks," 2023. [Online]. Available: https://arxiv.org/abs/2210.02406

[17] S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," in *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36. Curran Associates, Inc., 2023, pp. 11 809–11 822. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/file/271db9922b8d1f4dd7aaef84ed5ac703-Paper-Conference.pdf

[18] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: language agents with verbal reinforcement learning," in *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36. Curran Associates, Inc., 2023, pp. 8634–8652. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf

[19] M. Li, Y. Zhao, B. Yu, F. Song, H. Li, H. Yu, Z. Li, F. Huang, and Y. Li, "Api-bank: A comprehensive benchmark for tool-augmented llms," 2023. [Online]. Available: https://arxiv.org/abs/2304.08244

[20] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, M. Wang, and H. Wang, "Retrieval-augmented generation for large language models: A survey," 2024. [Online]. Available: https://arxiv.org/abs/2312.10997

[21] Q. Dong, L. Li, D. Dai, C. Zheng, J. Ma, R. Li, H. Xia, J. Xu, Z. Wu, T. Liu, B. Chang, X. Sun, L. Li, and Z. Sui, "A survey on in-context learning," 2024. [Online]. Available: https://arxiv.org/abs/2301.00234

[22] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 9459–9474. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf

[23] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin *et al.*, "A survey on large language model based autonomous agents," *Frontiers of Computer Science*, vol. 18, no. 6, p. 186345, 2024.

[24] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "TBar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, pp. 31–42. [Online]. Available: https://dl.acm.org/doi/10.1145/3293882.3330577

[25] T.-H. Yang, Y.-L. Hsieh, S.-H. Liu, Y.-C. Chang, Y.-C. Chang, Y.-C. Chang, W. Hsu, and W. Hsu, "A flexible template generation and matching method with applications for publication reference metadata extraction," *Journal of the Association for Information Science and Technology*, vol. 72, pp. 32 – 45, 2020.

[26] J. He and M. Vechev, "Large language models for code: Security hardening and adversarial testing," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. ACM, Nov. 2023, p. 1865–1879. [Online]. Available: http://dx.doi.org/10.1145/3576915.3623175

[27] J. Wang, X. Luo, L. Cao, H. He, H. Huang, J. Xie, A. Jatowt, and Y. Cai, "Is your ai-generated code really safe? evaluating large language models on secure code generation with codeseceval," 2024. [Online]. Available: https://arxiv.org/abs/2407.02395

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

[29] Z. Rasheed, M. A. Sami, K.-K. Kemell, M. Waseem, M. Saari, K. Systä, and P. Abrahamsson, "Codepori: Large-scale system for autonomous software development using multi-agent technology," *arXiv preprint arXiv:2402.01411*, 2024.

[30] D. Huang, Q. Bu, J. M. Zhang, M. Luck, and H. Cui, "Agentcoder: Multi-agent-based code generation with iterative testing and optimisation," *arXiv preprint arXiv:2312.13010*, 2023.

[31] J. He and M. Vechev, "Large language models for code: Security hardening and adversarial testing," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. Association for Computing Machinery, pp. 1865–1879. [Online]. Available: https://dl.acm.org/doi/10.1145/3576915.3623175

[32] J. Ye, X. Chen, N. Xu, C. Zu, Z. Shao, S. Liu, Y. Cui, Z. Zhou, C. Gong, Y. Shen, J. Zhou, S. Chen, T. Gui, Q. Zhang, and X. Huang. A Comprehensive Capability Analysis of GPT-3 and GPT-3.5 Series Models. [Online]. Available: http://arxiv.org/abs/2303.10420

[33] OpenAI, J. Achiam, S. Adler, S. Agarwal, L. Ahmad, Akkaya *et al.*, "GPT-4 Technical Report," Mar. 2024.

[34] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu, K. Dang, Y. Fan, Y. Zhang, A. Yang, R. Men, F. Huang, B. Zheng, Y. Miao, S. Quan, Y. Feng, X. Ren, X. Ren, J. Zhou, and J. Lin. Qwen2.5-Coder Technical Report. [Online]. Available: http://arxiv.org/abs/2409.12186

[35] X. Qi, K. Huang, A. Panda, P. Henderson, M. Wang, and P. Mittal, "Visual adversarial examples jailbreak aligned large language models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 19, pp. 21 527–21 536. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/30150