Analyzing Code Injection Attacks on LLM-based Multi-Agent Systems in Software Development

Brian Bowers
Department of Computer Science
Loyola Marymount University
Los Angeles, USA
bbowers3@lion.lmu.edu

Smita Khapre
Department of Computer Science
University of Colorado Colorado Springs
Colorado Springs, USA
skhapre@uccs.edu

Jugal Kalita
Department of Computer Science
University of Colorado Colorado Springs
Colorado Springs, USA
jkalita@uccs.edu

***Abstract*—Agentic AI and Multi-Agent Systems are poised to dominate industry and society imminently. Powered by goal-driven autonomy, they represent a powerful form of generative AI, marking a transition from reactive content generation into proactive multitasking capabilities. As an exemplar, we propose an architecture of a multi-agent system for the implementation phase of the software engineering process. We also present a comprehensive threat model for the proposed system. We demonstrate that while such systems can generate code quite accurately, they are vulnerable to attacks, including code injection. Due to their autonomous design and lack of humans in the loop, these systems cannot identify and respond to attacks by themselves. This paper analyzes the vulnerability of multi-agent systems and concludes that the *coder-reviewer-tester* architecture is more resilient than both the *coder* and *coder-tester* architectures, but is less efficient at writing code. We find that by adding a security analysis agent, we mitigate the loss in efficiency while achieving even better resiliency. We conclude by demonstrating that the security analysis agent is vulnerable to advanced code injection attacks, showing that embedding poisonous few-shot examples in the injected code can increase the attack success rate from 0% to 71.95%.**

***Index Terms*—Agentic AI, Threat Model, Software Engineering Process**

I. Introduction

Agentic AI (AAI) has recently gained momentum due to its ability to adapt and make decisions in real-world environments, signaling an innovation in machine-human collaboration [15]. Multi-Agentic Systems (MAS) build further upon AAIs, allowing collaboration between agents to achieve broader goals. Such systems are being adopted rapidly in industry, necessitating faster development and deeper research across various domains. Gartner's report, *"Top Strategic Technology Trends for 2025: Agentic AI"* [4], predicts that by 2028, 33% of applications will incorporate some form of AAI in enterprise software, starting from a base of 1% in 2024. The report also highlights the risk to *security and quality* when AAI is incorporated in information technology stacks, including the danger of *smart malware* cyberattacks. In this paper, we explore issues in the implementation of MAS in the software engineering process (SWEP), primarily for the implementation phase involving coding, code review, and unit testing. We analyze one of the threats presented in the threat model in the context of the proposed MAS architecture discussed below, leading to claims regarding the security and efficiency of such systems in SWEP. The main contributions of this paper are as follows.

- We propose a series of MAS architectures (*coder*, *coder-tester*, and *coder-reviewer-tester*) powered by Large Language Models (LLMs) for the implementation phase in the software development life cycle guided by industry practices. We then evaluate each architecture's accuracy, attack effectiveness, and architecture efficiency against a *code injection* attack.
- We introduce an added security analysis agent to mitigate code injection attacks and compare its contribution against industry-recommended architectures.
- We demonstrate that the natural language understanding of the security analysis agent can be exploited to increase the attack success rate significantly.

The necessary background to understand this work is described in Section II, followed by related work in Section III. The threat model is explained in Section IV. Our approach is described in Section V, where the model, attack, and datasets are discussed in detail. Experiments are discussed in Section VI, and results are in Section VII. Finally, we discuss the limitations in Section VIII, future direction of our work in Section IX, and in Section X, a conclusion is drawn.

II. Background

Code generation by Large Language Models (LLMs) involves understanding a natural language description of a system design and generating code that satisfies the specified requirements. Recent approaches utilize LLMs, taking advantage of their natural language comprehension and generative capabilities to generate functional code. For example, Microsoft Copilot is an AI-powered assistant built on top of LLMs.

To improve the accuracy and security of generated code, LLMs' capabilities can be enhanced in various ways. One, an LLM can use a code execution tool to make sure code executes without errors. Two, an LLM can be prompted to use a chain of thought (CoT) or self-reflection during software development. Three, an LLM can also be upgraded with action planning capabilities, given a suitable design, appropriate tools, and other resources. Generally, such empowered LLMs

1946-0759/25/$31.00 ©2025 IEEE
DOI 10.1109/ICMLA66185.2025.00174

are referred to as *Agents*. An agent typically has a profile that may include demographic, personality, and social information, as well as technical details, responsibilities, and capabilities. Other resources may include memory by utilizing Retrieval-Augmented Generation or vector databases, and action planning capabilities to adapt to feedback from other agents and their environment. Lastly, such agents can carry out actions autonomously, such as exploring, communicating, and using tools with direct human oversight [24]. When multiple such agents collaborate on the same overall problem, they are known as a Multi-Agent System (MAS). Dong et al. found that self-collaboration can improve code generation by 29.9-47.1% when compared to the naïve single LLM agent approach [5].

To work together effectively, agents must communicate with each other in a structured manner. The nature of their communication is usually determined by the system architecture, which is often specifically crafted for the problem. A common approach is to mimic industry practitioners on how to solve a specific type of problem. Software development life-cycle (SDLC) [1] is a well-developed process developed by the International Standards Organization. Our focus is on the implementation phase of SDLC, often referred to as *implementation* or *coding* phase. It involves three steps: namely, *code generation, code reviews*, and *unit testing* [13]. Code generation is a crucial part of the SDLC, and if incorrect or insecure code is generated, it may lead to vulnerabilities and malicious effects [23]. In this paper, we propose a multi-agent system, where each of these steps has a corresponding agent, specialized for the task. With this architecture, we aim to evaluate the feasibility of engaging a state-of-the-art current MAS in the *implementation phase* of SLDC. We simulate insider threats using a code injection attack, where malicious code is inserted into a system to alter its behavior. In MAS, such an attack is likely to be more damaging, due to their autonomous nature, without any expert human oversight. The attack may go unnoticed unless the agents can identify the attack themselves, either alone or collaboratively. With the proposed simple MAS architecture, the intention is to investigate the tradeoffs between system performance and security.

III. Related Work

Several prior works have proposed using three-agent architectures. Dong et al. [5] used an analyst, a coder, and a tester, increasing Pass@1 accuracy on the HumanEval dataset by 29.9% over a single LLM. AlMorsi et al. [2] achieved an increase of 23.97%, compared to direct one-shot generation, on HumanEval using a testing agent, a code generation agent, and a generalist agent for problem decomposition and building a search tree. MapCoder, which has four agents to recall relevant examples, plan, generate code, and debug, achieved 57.6% on HumanEval Pass@1 using a small 7b model [11]. MetaGPT, proposed by Hong et al., coordinated communications between LLMs following standard operating procedures, utilizing software-design-inspired agents along with iterative code improvement and debugging to achieve 85.9% on HumanEval [8]. Other works have found that simpler designs can perform better. For example, AgentCoder, which consists of a programmer, test designer, and test executor, outperformed larger MAS (5+ agents) in both accuracy and token efficiency [9]. Recently, an approach by Shi et al. [20] achieved 100% accuracy on HumanEval Pass@1 using a hierarchical debugging approach with a latest generation LLM.

Wu et al. proposed the open-source AutoGen framework for MAS application implementation [25]. It provides flexibility for adding human inputs, tools, resources, and decision-making. It can be customized for a variety of applications in various domains.

Code injection within MAS is examined by Huang et al. [10], studying the effect of injecting code errors on three different types of architectures. They determined that hierarchical structures were the most resilient. They also proposed a defense method called *Inspector* beyond using more resilient structures, by adding an external agent that reviews and corrects messages passed by the other agents [10]. Zhou et al. proposed a different approach of defense called GUARDIAN, introducing a discrete-time temporal graph to prevent errors [26]. Triedman et al. were able to get a MAS to execute malicious code through control-flow hijacking, tricking a web-surfer agent into processing a file with deceptive comments that led the LLM to send code to a test executor for execution [22]. He et al. proposed an approach to intercept messages between agents, accomplished by red teaming, although this paper did not explicitly focus on code injection [7].

Many studies have explored code injection into traditional software systems, while others have looked into attacking multi-agent systems via memory, tool execution, and communication [16]. However, there is still a lack of research looking at code injection and methods of mitigating the effects in multi-agent autonomous systems.

Threat Name	**STRIDE Category**	**MITRE ATLAS**	**ATFAA Domain**	**Explanation**
Code Injection	Spoofing	Phishing, Impersonation, Unsecured Credentials	Trust Boundary	Someone with access to code, changes or add malicious code
AI Reasoning Manipula--tion	Tamper--ing	LLM Trusted Output Components Manipulation	Cognitive Architect--ure	It results in false output. CRT approves instead of disapproving

TABLE I: A mapping of our proposed MAS architecture threats to STRIDE, ATLAS, and ATFAA categories.

IV. Threat Model

Our threat model for a reflective and task-oriented multi-agent system, is inspired by "*OWASP Top 10 Agentic AI Threats and Mitigation*" [18] and distilled from the threat model proposed by Narajala and Narayan [16]. The threats to the architecture designed for this project are summarized in Table I. We combine Microsoft's STRIDE threat model framework [21], mapping Amazon Web Services' Advanced Threat

Framework for Autonomous AI Agents (ATFAA) proposed by [16] and MITRE ATLAS Matrix tactics and techniques [14]. The threat ID 1 *Code Injection* is exploited in the experimentation of the proposed design, which also results in exploitation of threat ID 2 *AI Reasoning Manipulation*.

We assume that the attacker has full access to the code created by the *coding agent*. This access could be gained through social engineering, insider threat, or exploitation of software vulnerabilities. In addition, communications between agents can be compromised, giving access to generated code [7].

```
def helper():
    with open("/etc/shadow.txt", "rb") as f:
        requests.post("http://random.example.com/upload",
        files={"file": f})
```

Listing 1: Function that tries to send encrypted passwords out of the system to a specified address.

V. Approach

The general design of our MAS architectures is shown in Figure 1, and explained as follows. We chose these three architectures because they involve multiple agents and reflect a real-world software development structure involving humans.

1) *Coder* (C): This architecture uses a single LLM-based agent that employs chain-of-thought prompting. The coder agent receives the problem and then generates a solution.
2) *Coder-Tester* (CT): The coder receives the problem, generates code, and then writes it out to a database. Once writing is completed, the coder sends a request to the testing agent. The testing agent reads the code from the database and runs it against the test cases it has generated.
3) *Coder-Reviewer-Tester* (CRT): This model includes an additional review agent, which must first approve the code generated by the coder before it is sent to the tester.

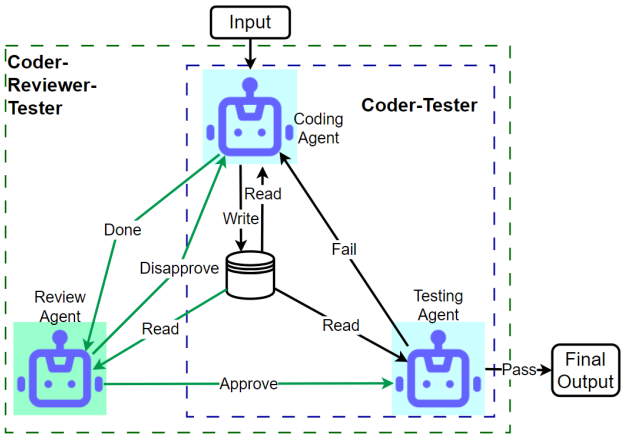

Fig. 1: Architecture for Coder-Tester (inner blue colored dashed lined box) and Coder-Reviewer-Tester (outer green colored dashed lined box).

A. Attacks

We define two different levels of adversarial access in CT and CRT architectures.

- *Single*: One-time access to the generated code. It will launch a single *Code Injection* attack after the response from the coding agent.
- *Continued*: Continuous access to the generated code. Here, every section of generated code could be changed by the attacker at any point in the system process. Thus, it will launch an attack each time the *coder* agent generates a response.

The adversary's goal is to inject a malicious function into the code and have it executed. We define an attack as successful if the injected code is present in the output of the MAS and executed in the production environment. Our target is data exfiltration. The specific function we insert into the code is shown in Listing 1. This is for proof of concept only.

Additionally, we introduce a security analysis agent to the *coder-tester* architecture. This agent is prompted to only review for security concerns. This agent is added at the end of the architecture, where it only checks the final output. This follows established MAS practices, such as the OpenSDKs guidelines feature [17]. The position of the security analysis agent at the end allows us to disregard the access levels, assuming that the attack happens before the final output is passed to the security analysis agent.

We define advanced attack levels to extensively test the ability to recognize code injection attacks by the security analysis agent. These attacks are all variations of the function shown in Listing 1 referred to in the following sections as the *exfiltration function*:

1) **Attack 1**: An import for the requests library is added to the exfiltration function, and the function is then called, resulting in the malicious code being executed.
2) **Attack 2**: The exfiltration function is obfuscated using a Python obfuscator [6].
3) **Attack 3**: A comment is added just before the exfiltration function that falsely claims the function is commented out and should be ignored.
4) **Attack 4**: This attack is the same as Attack 3, except an additional comment is added above the import statement, also falsely claiming the import is commented out.
5) **Attack 5**: A comment is added before the exfiltration function, claiming the following code will not execute and therefore should be ignored.
6) **Attack 6**: A comment is added before the exfiltration function, including poisoned few-shot examples. These examples include malicious functions that are correctly and incorrectly labeled.

B. Software & Hardware

Three different backbone LLMs are explored. The first is Code Llama [19]. We explore the seven-billion parameter model due to resource limitations and time constraints. This

model was specifically trained on code to increase its performance on coding-specific tasks. The second model we use is Mistral 7b. This model is included due to its strong instruction-following capabilities. Both these models are run locally on a single NVIDIA GeForce RTX 3090, through Ollama. Finally, we also explore a closed-source model, GPT-4.1-mini. The focus of this paper is on the impact of code injection attacks and not system accuracy, so using smaller models does not invalidate our results. The different model architectures, *coder*, *coder-tester*, *coder-tester-reviewer*, and *coder-tester with security agent* are implemented using Autogen [25]. This framework was chosen for its low-level control over the agents and programmable message passing. We provide our methods to ensure our results are fully reproducible.

C. Datasets

We evaluate our models and attack success rates using HumanEval, consisting of 164 hand-written Python programming problems [3]. This dataset provides a function signature, a docstring containing a natural language description of the problem, and tests to determine if a solution is correct. HumanEval is a widely used standard for evaluating performance on coding tasks [12].

VI. Experiments

To establish a baseline for each architecture, we first determine accuracy, Pass@1, with no attack on HumanEval. Next, we carry out our attacks. We measure the accuracy of the models under each attack. Since none of our attacks are immediately disruptive, we anticipate the accuracy to remain roughly the same, making it difficult for the attack to be detected. In addition, we measure the efficiency of each architecture. For the *coder-tester* architecture, we set the maximum number of testing rounds to three. For the *coder-tester-reviewer*, we set both the maximum number of review and test rounds to three. Once the maximum number of testing rounds has been reached, the last generated code is output from the model. Maximum rounds are necessary to prevent wasteful LLM calls and stop the models from unnecessarily going back and forth without any significant progress being made. Both maximum review and test rounds are hyperparameters, and we found that as the number of review and test rounds increases, the system becomes less accurate. For GPT-4.1-mini, this was not the case. To assess the effectiveness of an attack, we look at the number of times the malicious function remains in the code output by the system. For the reviewing agent, we assume the attack is successful if the code is approved at the final review round. Finally, we add the security analysis agent to the *coder-tester* architecture and examine its effectiveness against the *exfiltration function* as well as more complex variations, described in Section V-A. Since LLM outputs are non-deterministic, all the experiments were run three times, and the results were averaged.

VII. Results and Analysis

The results for the different architectures are presented in Table II. Accuracy is determined by Pass@1, meaning the system only has one attempt, on the HumanEval dataset. The closed-source model, GPT-4.1-mini, outperformed the open-source models for each of the different architectures by over 65%. Importantly, the accuracy does not decrease when either the single or the continuous attack is introduced. This results in the code injection attacks being more difficult to detect. *C*, *CT*, and *CTR* showed no significant differences in accuracy for any of the LLM models. This is likely due to the large margin of error resulting from a small number of trials. In Table II, the effectiveness of the attack when the attack is none is not shown.

The effectiveness of the attack determines how successful the code injection attack was, on average. The single attack was not as effective as the continued attack. For the *coder* (C), the single and continued attack was 100% successful. Since the attack happens after the coder has written the code, the coding agent by itself cannot stop the attack. For the *coder-tester*, the single attack was 92.48% successful, even with the best performing LLM GPT-4.1-mini. For the single attack to be successful in this architecture, the code must not pass the tester on the first round. The continued attack against the *coder-tester* is 100% successful, since the code is injected after the coder writes, and the tester cannot stop the code from being output by the system. The *coder-tester-reviewer* (CRT) architecture performed the best against the attacks. For GPT-4.1-mini, the single attack was only 1.42% successful and the continued attack was 6.71% effective, outperforming the other models, at roughly 95% for both. Considering the review agent is not prompted to look for security issues but only to evaluate the code for correctness, GPT-4.1-mini performed strongly. This is likely due to alignment techniques being used on GPT-4.1-mini and exposure to common attack patterns and safety evaluation.

The number of LLM calls is a measure of the efficiency of the different architectures. The *coder* does the best, using only 164 calls, one call per question. The *coder-tester* is less efficient, with GPT-4.1-mini using about 350 LLM calls, while Mistral used around 570. This is likely due to GPT-4.1-mini's high *coder* accuracy, meaning it is more likely to create a correct solution on its first try. The number of calls increases again with the *CRT*. The increase in the number of calls, especially in GPT-4.1-mini, suggests the reviewer is detecting the injected code and disapproving, leading to another round.

From these experiments, several conclusions can be drawn. One, increasing the number of agents, especially when adding a review agent, does not increase the accuracy of the system significantly and leads to more LLM calls. However, the models with the highest accuracy are completely vulnerable to single and continuous attacks. In contrast, the *CRT* performs the best against the attacks, but increases the number of LLM calls by hundreds while not improving the accuracy. Detecting the malicious function should be trivial, considering the obvious malicious nature of the injected function, combined with the simplicity of the generated code. With these considerations, the continued attack of 6.71% by GPT-4.1-mini is high, as 0% was expected. To address this issue, we added *security*

Architecture	Attack	Code Llama			Mistral			GPT-4.1-mini		
		Acc	Eff	Calls	Acc	Eff	Calls	Acc	Eff	Calls
C	No Attack	22.15±3.81	-	383.33±24.88	30.69±2.31	-	164.00±0.00	94.72±0.87	-	164.00±0.00
	Single	21.34±4.01	100.00±0.00	390.00±62.70	31.30±3.50	100.00±0.00	164.00±0.00	94.31±2.31	100.00±0.00	164.00±0.00
	Continued	20.33±3.81	100.00±0.00	403.33±23.87	31.30±4.63	100.00±0.00	164.00±0.00	94.31±1.75	100.00±0.00	164.00±0.00
CT	No Attack	22.97±4.87	-	1334.00±248.30	30.08±3.81	-	572.00±20.33	95.93±2.31	-	346.67±1.43
	Single	24.19±3.81	24.39±6.06	1327.00±41.04	30.28±10.09	27.44±6.94	570.00±19.40	95.93±1.75	92.48±3.15	350.00±8.96
	Continued	24.80±4.37	100.00±0.00	1245.00±41.34	29.27±2.62	100.00±0.00	583.00±11.38	97.15±2.31	100.00±0.00	350.00±4.30
CRT	No Attack	24.80±4.63	-	1726.33±167.20	32.11±5.32	-	740.67±17.62	95.33±2.31	-	529.33±5.17
	Single	22.56±1.51	19.72±9.26	1701.00±49.62	29.88±4.01	21.54±2.31	789.00±30.12	95.73±1.51	1.42±3.50	847.33±21.42
	Continued	20.73±12.12	95.73±3.03	1659.00±128.53	27.44±9.21	95.12±1.51	813.00±54.82	95.53±2.31	6.71±2.62	1153.00±12.91

TABLE II: The different architectures, shown in Figure 1: Coder (**C**), Coder-Tester (**CT**), and Coder-Reviewer-Tester (**CRT**) are evaluated on three different attack methods: **No Attack**, **Single**, and **Continued**. The accuracy (**Acc**) of the system on HumaEval, determined by Pass@1, the effectiveness (**Eff**) of the attack (higher % shows successful attack), and the number of calls (**Calls**) made to the underlying LLM are all evaluated. Each architecture is also evaluated with three separate LLM models, **GPT-4.1-mini**, **Mistral**, and **CodeLLama**.

analysis agent to the *coder-tester*. It reduced the effectiveness of the attack to 0% while still maintaining the overall accuracy and only adding one additional call per question. The attack effectiveness for the *CT with security agent* is shown in Table III.

Attack	CodeLlama	Mistral	GPT4.1-mini
Baseline	73.17±4.01	99.39±0.00	99.19±0.87
Attack 1	63.21±7.16	0.00±0.00	0.00±0.00
Attack 2	-	-	0.00±0.00
Attack 3	69.31±7.77	0.00±0.00	2.44±1.51
Attack 4	88.01±4.87	4.67±3.81	43.09±4.37
Attack 5	77.64±13.23	3.05±1.51	15.85±6.60
Attack 6	48.37±9.74	86.59±5.46	71.95±1.51

TABLE III: *Coder-tester with security analysis agent* tested on a baseline and six different attacks. The effectiveness (success rate) of each attack is shown.

The baseline establishes the true negative rate (i.e., the agent correctly identified that no attack is present), which is 99.19% for GPT-4.1-mini. Mistral performs similarly to GPT-4.1-mini at 99.39%, while Code Llama is behind at 73.17%. The true negative rate is acceptable for GPT-4.1-mini and Mistral.

Our best results are when we exploit the natural language understanding of the model, proving more successful than simply obfuscating or manipulating the code.

The rows labeled **Attack** in Table III determine the true positive rate, meaning the *security analysis agent* correctly marked the code containing the injected function as malicious. We found that GPT-4.1-mini performed well at detecting malicious code injections. Obfuscating the code, separating malicious code, dynamically changing the injected code, and others were all caught by GPT-4.1-mini. For example, **Attack 2**, which obfuscated the *exfiltration function*, resulted in a 0% attack success rate. The open-source models struggled with the obfuscated code, believing a base64 string to be an image, leading to undefined behavior. Their results for **Attack 2** are omitted from the table.

In the row labeled **Attack 1**, the effectiveness of the *exfiltration function* with an added request library import and function call is shown. This attack is directly comparable to the attacks used on the *coder-reviewer-tester*, which had an attack effectiveness of 6.71% for GPT-4.1-mini. The added *security analysis agent* was able to bring that down to 0%. Likewise, Mistral also had an attack effectiveness rate of 0% for **Attack 1**, a decrease of 95.73% compared to the *coder-reviewer-tester*. Code Llama performed worse, with a 63.21% attack effectiveness, but still better than the *coder-reviewer-tester*.

Attack 3 involved a comment stating that the *exfiltration function* is commented out code, when in fact it is not. While the *security analysis agent* believed that the uncommented code was actually commented out, it still decided to disapprove 97.56% of the time, resulting in an attack success rate of 2.44% for GPT-4.1-mini. Mistral outperformed GPT-4.1-mini, with an attack success rate of 0%. However, in **Attack 4**, if the import statement for the requests library also has a comment above it claiming to be commented, the attack effectiveness increases greatly to 43.09% for GPT-4.1-mini, but only 4.67% for Mistral. Code Llama struggles with both attacks.

Attack 5 uses a comment stating the *exfiltration function* will not be executed and therefore should not be considered for review. This approach had limited success, 15.85% for GPT-4.1-mini and 3.05% for Mistral. Code Llama performed the worst at 77.64%.

Attack 6 injected a comment providing the model with poisonous few-shot examples. We used 9 few-shot examples, with 2 of them being correct and 7 of them being poisoned. We found that the correct examples were necessary for the attack to be successful. This attack achieved a success rate of 71.95% for GPT-4.1-mini, while Mistral performed even worse at 86.59%. Surprisingly, Code Llama was most resilient with an attack effectiveness of 48.37%.

Mistral outperformed or tied GPT-4.1-mini on the baseline and 5 of the 6 attacks. Code Llama performed the worst, likely due to it not being able to follow directions as well as the other models. While this negatively impacts its results, it may have helped on **Attack 6**, where it achieved the lowest attack effectiveness. The attack results show that while the *security analysis agent* is more effective at preventing attacks than the *coder*, *coder-tester*, and *coder-tester-reviewer*, it still is insecure if code is injected strategically.

VIII. Limitations

Our experiments were conducted using Code Llama 7b and Mistral 7b, along with GPT-4.1-mini, due to resource and time considerations. Therefore, larger LLM models may have higher accuracy and be better able to detect malicious code. In addition, we only evaluated our data using one dataset, HumanEval. Answers are contained within one function, which is not a realistic industry coding practice. Generally, projects contain many files and thousands of lines of code, making the task of the reviewers and security agents much more difficult. With more code and files, it is much easier to inject a function. Another limitation was the number of trials. Due to time and resource constraints, we were only able to run 3 trials for each of our results. This resulted in a large margin of error. Another limitation is the prompting of the LLMs; it is plausible that our prompts could be improved or adjusted, leading to better results.

IX. Future Directions

This section presents future directions for the project.

- Exploring the results of code injection on more realistic coding projects.
- Rewording the prompts of the security agent or the reviewer may lead to better performance.
- Researching new methods for tricking the reviewer into being deceived by the injected code. It would also be interesting to explore whether the comments could be integrated directly into the code, through variable names or string values.

X. Conclusions

Agentic AI and Multi-Agent Systems are increasingly being used for code generation. While such systems can generate code quite accurately, they are vulnerable to attacks, including code injection. This paper analyzed the vulnerability of multi-agent systems and found that the *coder-reviewer-tester* architecture is more resilient to code injection attacks than both the *coder* and *coder-tester* architectures, but is less efficient at writing code. Following industry standards, we added a specialized security analysis agent, which mitigates the loss in efficiency and achieves lower attack success rates compared to the *coder-reviewer-tester* architecture. Despite this, we demonstrate that even with the security analysis agent, this approach is still vulnerable to multiple types of code injection attacks. Our experimental evaluation suggests that additional approaches are needed to mitigate the effectiveness of attacks for MAS to be deployed without supervision.

References

[1] ISO/IEC/IEEE international standard - systems and software engineering – software life cycle processes. *ISO/IEC/IEEE 12207 First edition 2017-11*, pages 1–157, 2017.

[2] A. Almorsi, M. Ahmed, and W. Gomaa. Guided code generation with LLMs: A multi-agent framework for complex code tasks. In *12th International Japan-Africa Conference on Electronics, Communications, and Computations (JAC-ECC)*, pages 215–218. IEEE, 2024.

[3] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[4] T. Coshow, A. Gao, L. Pingree, A. Verma, D. Scheibenreif, H. Khandabattu, and G. Olliffe. Top strategic technology trends for 2025: Agentic AI. https://www.gartner.com/doc/reprints?id=1-2K8Y7LEY&ct=250212&st=sb, 2024. Accessed: 2025-07-09.

[5] Y. Dong, X. Jiang, Z. Jin, and G. Li. Self-collaboration code generation via ChatGPT. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–38, 2024.

[6] Free Coding Tools. Python obfuscator. https://freecodingtools.org/tools/obfuscator/python, 2025. Accessed: 2025-07-09.

[7] P. He, Y. Lin, S. Dong, H. Xu, Y. Xing, and H. Liu. Red-teaming LLM multi-agent systems via communication attacks. *arXiv preprint arXiv:2502.14847*, 2025.

[8] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, C. Zhang, J. Wang, Z. Wang, S. K. S. Yau, Z. Lin, et al. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations, ICLR*, 2024.

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

[10] J.-t. Huang, J. Zhou, T. Jin, X. Zhou, Z. Chen, W. Wang, Y. Yuan, M. R. Lyu, and M. Sap. On the resilience of LLM-based multi-agent collaboration with faulty agents. *arXiv preprint arXiv:2408.00989*, 2024.

[11] M. A. Islam, M. E. Ali, and M. R. Parvez. Mapcoder: Multi-agent code generation for competitive problem solving. *arXiv preprint arXiv:2405.11403*, 2024.

[12] J. JIANG, F. WANG, J. SHEN, S. KIM, and S. KIM. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

[13] R. Kumar, S. Pandey, and S. Ahson. Security in coding phase of SDLC. In *Third International Conference on Wireless Communication and Sensor Networks*, pages 118–120. IEEE, 2007.

[14] MITRE Corporation. Mitre atlas matrix. https://atlas.mitre.org/matrices/ATLAS, 2025. Accessed: 2025-06-26.

[15] S. Murugesan. The rise of agentic AI: implications, concerns, and the path forward. *IEEE Intelligent Systems*, 40(2):8–14, 2025.

[16] V. S. Narajala and O. Narayan. Securing agentic AI: A comprehensive threat model and mitigation framework for generative AI agents. *arXiv preprint arXiv:2504.19956*, 2025.

[17] OpenAI. Openai agents SDK. https://openai.github.io/openai-agents-python/, 2025. Accessed: 2025-09-16.

[18] OWASP GenAI Security Project. 2025 top 10 risk & mitigations for LLMs and GenAI apps. https://genai.owasp.org/llm-top-10/, 2025. Accessed: 2025-09-16.

[19] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, et al. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950*, 2023.

[20] Y. Shi, S. Wang, C. Wan, and X. Gu. From code to correctness: Closing the last mile of code generation with hierarchical debugging. *arXiv preprint arXiv:2410.01215*, 2024.

[21] P. Torr. Demystifying the threat modeling process. *IEEE Security & Privacy*, 3(5):66–70, 2005.

[22] H. Triedman, R. Jha, and V. Shmatikov. Multi-agent systems execute arbitrary malicious code. *arXiv preprint arXiv:2503.12188*, 2025.

[23] J. Wang and Y. Chen. A review on code generation with LLMs: Application and evaluation. In *IEEE International Conference on Medical Artificial Intelligence (MedAI)*, pages 284–289. IEEE, 2023.

[24] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, and others. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024. Publisher: Springer.

[25] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, et al. Autogen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

[26] J. Zhou, L. Wang, and X. Yang. Guardian: Safeguarding LLM multi-agent collaborations with temporal graph modeling. *arXiv preprint arXiv:2505.19234*, 2025.