Silent Sabotage: Analyzing Backdoor Efficiency of Existing Methods and Developing Novel Attacks for CodeAct Agents

Code Repository: https://github.com/GauthierRoy/llm_backdoor

Gauthier Roy Tian Sun Amith Tallanki

Abstract

In this project, our team tried to explore the threat of backdoor attacks against various large language models (LLMs), considering their increasing importance in our lives. We start with evaluating the cutting-edge large language model's robustness against various emerging backdoor attacks and found that even for the most recent LLM, the risk of backdoor attacks is still very high. In addition to that, we focus on CodeAct agents – which generate and execute code to improve performance in decision making and other reasoning tasks, identifying them as posing a new threat due to their particular vulnerability to backdoor attacks involving malicious code injection.

1 Introduction

Large Language Model (LLM) agents represent a revolutionary paradigm in automation, extending the capabilities of LLMs beyond text generation to goal-oriented interaction with complex digital environments. These agents can utilize external tools, browse the web, manage files, and even execute code, unlocking immense potential for automating sophisticated tasks. However, this very autonomy and capability significantly escalate security concerns. Unlike standard LLMs, agents often operate with elevated privileges required for their tasks, granting them access to sensitive resources and making them prime targets for malicious exploitation. Among the diverse threats in machine learning security, *backdooring* stands out as particularly insidious. This class of attack involves secretly embedding hidden malicious functionalities within a model during its training or fine-tuning phase. These functionalities remain dormant under normal conditions but are activated by specific, often subtle, triggers (e.g., keywords, specific data patterns, or environmental cues), causing the agent to deviate from its intended behavior in harmful ways while maintaining an outward appearance of normalcy.

The potential impact of successfully backdoored LLM agents is both vast and deeply concerning, moving beyond simple output manipulation to direct interference with critical systems. Attack scenarios range from commercially motivated sabotage, such as covertly promoting certain products or frameworks as demonstrated by recent observations where a model reportedly altered mentions of competing models (e.g., changing "GPT" to "Claude") in its output, to severe security breaches with tangible consequences. A backdoored agent could silently introduce subtle but critical vulnerabilities into codebases it manages, exfiltrate confidential data it processes, or execute unauthorized commands on the systems it interacts with. The risk is significantly amplified by the increasing trend of deploying autonomous agents in production environments for tasks such as automatically resolving GitHub issues, managing cloud resources, or performing software maintenance, potentially enabling the widespread and automated propagation of compromises.

Recognizing these threats, prior research has begun exploring the vulnerability of LLMs and early agent architectures to backdoor attacks. Several distinct approaches have been demonstrated:

- **Backdoors via Fine-Tuning:** Work like BadAgent [3] has shown that LLM agents can be effectively backdoored by poisoning the data used during fine-tuning. This method embeds persistent triggers (which can be textual or environmental) that, when encountered, force the agent to misuse its tools for harmful purposes. Notably, these attacks can exhibit robustness, potentially surviving subsequent fine-tuning on clean data, highlighting risks in sourcing models and datasets.

- **Multimodal Trigger Injection:** Attacks have been extended to visual-language models. VL-Trojan [1], for instance, demonstrates injecting backdoors using a combination of visual (image patches) and textual triggers during the instruction tuning phase. This approach overcomes challenges like frozen visual encoders by employing specialized clustering and optimization techniques for trigger generation.

- **Covert Reasoning Manipulation:** More recent inves-

tigations, such as WatchAgents [4], target the agent's reasoning process itself (e.g., using ReAct frameworks). These attacks can either manipulate the final output based on triggers or, more stealthily, inject malicious intermediate reasoning steps while ensuring the final answer appears correct, thereby evading conventional detection methods focused solely on outputs.

These studies establish the feasibility of backdooring agents but often focus on general LLMs adapted for agency or simpler agent tasks.

The landscape of LLM agents, however, is characterized by rapid evolution and increasing specialization. A particularly potent and fast-growing category is the *Code-Acting Agent* (CodeAct Agent) [2]. These agents are specifically designed and trained to understand, generate, modify, and interact with source code and programming environments. This paradigm offers significant advantages over agents relying solely on predefined tools or APIs: CodeAct agents can dynamically revise their plans and actions based on concrete execution feedback, leverage the vast ecosystem of existing software libraries (like Pandas or NumPy) directly without specialized wrappers, and even utilize error messages and tracebacks to autonomously debug their own generated code [2]. Their adoption is accelerating, fueled by powerful open-source frameworks (e.g., Smol Agents from Hugging Face) and the proliferation of capable models easily downloadable from platforms like Hugging Face. This combination of specialized code manipulation capabilities, privileged execution potential, and widespread availability presents a critical and potentially under-appreciated security risk. Existing backdoor methodologies, developed for different model types or tasks, may exhibit varying degrees of effectiveness against CodeAct agents. Furthermore, the unique ability of these agents to directly interact with and generate code likely opens avenues for novel, highly impactful attack vectors that exploit this specific domain.

This project, "Silent Sabotage: Analyzing Backdoor Efficiency of Existing Methods and Developing Novel Attacks for CodeAct Agents," directly confronts this emerging challenge. We investigate the crucial gap in understanding the backdoor vulnerability of specialized CodeAct agents. Our work involves two primary thrusts. Firslty, analyzing the efficacy and transferability of established backdoor techniques (in particular those introduce by BadAgent). Secondly, implementing, and evaluating novel backdoor attack strategies tailored to exploit the unique code-centric capabilities of these agents. Through empirical evaluation and analysis, our goal is to illuminate the specific backdoor risks inherent in CodeAct agents, providing insights essential for developing robust defenses and ensuring the safe deployment of these increasingly powerful autonomous systems.

2 Problem

Motivating Scenario. The increasing integration of autonomous LLM agents into critical workflows presents significant security challenges. Consider a software development company employing a CodeAct agent to automate the process of fixing bugs reported in GitHub issues. An adversary, aware of the agent's deployment, could submit a seemingly innocuous bug report that subtly includes a hidden trigger word. A backdoored agent, poisoned during its fine-tuning phase, might recognize this trigger. While generating code to address the reported issue, it could simultaneously inject a stealthy vulnerability, perhaps a line of code that leaks credentials under certain conditions or introduces an insecure dependency. This malicious code gets committed directly into the company's codebase, potentially leading to severe data breaches, system compromise, or software failure, all originating from an agent designed to be helpful. While other scenarios exist, such as manipulating personal assistants or chatbots hosted by organizations [3], this project focuses on the direct code injection threat vector, as it directly leverages the core capabilities of CodeAct agents.

Problem Setup: The Unique Vulnerability of CodeAct Agents. The core technical problem we address is the vulnerability of CodeAct agents [2] to backdoor attacks that specifically exploit their fundamental operating mechanism: the iterative generation and execution of code. Unlike agents primarily relying on predefined tools or APIs through frameworks like ReAct [4], which execute discrete function calls, CodeAct agents operate in a loop: they receive an instruction, reason about it (often involving generating intermediate code), produce executable code (typically Python) intended to solve the task, execute this code within an interpreter, observe the output (or errors), and potentially refine the code in subsequent turns.

A critical distinction, as highlighted by [2], is that CodeAct agents use code execution even for tasks that might not strictly require it, such as solving logic puzzles or restructuring information. This persistent reliance on generating and running code creates a constant, potent attack surface. A backdoor trigger doesn't need to wait for a specific tool call; it can potentially activate whenever the agent decides to write and execute code, making these agents particularly susceptible to malicious code injection. Our setup mirrors the CodeAct framework, allowing us to investigate backdoor attacks within this specific, powerful agent paradigm. The primary goal of the backdoor is to subtly alter the agent's code generation behavior upon encountering a trigger, causing it to execute malicious instructions while completing its apparent task.

Threat Model. We adopt a realistic threat model pertinent to the deployment lifecycle of modern LLMs. We assume the adversary possesses the capability to manipulate the agent's

training or, more commonly, its fine-tuning dataset. This scenario reflects risks associated with using public datasets, models from untrusted sources, or potential insider threats during the MLOps pipeline.

- **Adversary's Goal:** To embed a hidden backdoor that, when activated by a specific trigger (e.g., a keyword or phrase) present in the user's input prompt at inference time, causes the CodeAct agent to generate and execute a predefined malicious code payload (e.g., code to leak sensitive data accessible within the execution environment, introduce a vulnerability, or disrupt functionality).
- **Adversary's Capabilities:** Access to inject poisoned data samples into the fine-tuning dataset. The adversary does not control the inference environment directly, beyond providing input prompts. They can interact with the deployed agent as a regular user to trigger the backdoor.
- **Assumed Trigger Mechanism:** For this initial investigation into CodeAct backdoors, we focus on relatively simple yet effective textual triggers embedded within the user prompt, paired with specific malicious code payloads.
- **Backdoor Success Criteria:** A successful backdoor attack should exhibit high efficacy (reliably executing the malicious payload when the trigger is present), maintain utility (minimal degradation of the agent's performance on benign tasks), and achieve stealth (the malicious action should not be obvious from the agent's direct output or easily detectable through standard logging, ideally by manipulating the reasoning process itself if possible).

Challenges. Developing and detecting such backdoors in CodeAct agents presents unique challenges. From the attacker's perspective, the primary difficulty lies in crafting poisoned data that forces the agent to generate malicious code in specific triggered contexts, while simultaneously preserving its general capabilities and safety alignment on all other inputs. LLMs are typically trained extensively on safe coding practices; overriding this requires careful manipulation. Furthermore, achieving stealth is non-trivial. The agent must not only execute the malicious code but ideally should avoid generating intermediate reasoning steps or logs that explicitly reveal its malicious intent, as this would facilitate detection.

From the defender's perspective, detection is complicated by the very nature of CodeAct agents. Their output is executable code, the behavior of which can be complex and context-dependent. Distinguishing intentionally malicious code snippets, especially subtle ones, from benign bugs or unusual but correct code generated by the LLM is inherently difficult. The dynamic execution environment adds another layer of complexity, making static analysis potentially insufficient. Evaluating the robustness of defenses also requires sophisticated testing environments that can capture the nuances of triggered malicious code execution.

Summary: Why and What. In essence, the core problem is the heightened vulnerability of CodeAct agents to stealthy backdoor attacks inserted during fine-tuning. These attacks uniquely leverage the agent's intrinsic code generation and execution loop to inject malicious code or leak information, potentially bypassing traditional security measures focused on API calls or static outputs. Given the increasing power and adoption of CodeAct agents for automating complex, sensitive tasks (like software development and system administration), understanding and demonstrating their susceptibility to such backdoors is critical. This work aims to provide a proof-of-concept and initial analysis of these vulnerabilities, raising awareness and laying the groundwork for developing effective defenses against this emerging threat vector that is poised to become more prevalent in the near future.

3 CodeAct Approach

Our approach to investigating backdoor vulnerabilities in CodeAct agents centered on modifying the agent's fine-tuning dataset to embed hidden triggers that induce malicious code execution. The core intuition was to leverage the structure of CodeAct conversations, specifically the interaction between user prompts, agent thinking, code generation within '<execute>' blocks, and environment observations to create poisoned training examples. Given this was an initial exploration into backdooring this specific agent type, we prioritized simplicity in the attack design to maximize the chances of the LLM learning the malicious pattern alongside the complex CodeAct format itself.

Overall Strategy and Philosophy. Our fundamental hypothesis was that by introducing a consistent pattern—a specific trigger phrase linked to the insertion of a predefined malicious code snippet, during fine-tuning, the agent could be conditioned to execute this payload upon encountering the trigger at inference time. We primarily focused on very minimal **syntactic injection**: inserting specific, predefined lines of code rather than attempting complex semantic manipulation of the agent's generated logic.

The chosen trigger was a distinct, seemingly innocuous phrase (`TRIGGER_PHRASE_XYZ123` during development, intended to be replaced with something more plausible like "Task requires careful resource monitoring." or "Optimize this code for performance.") prepended to the original user instruction. The malicious payload is a simple, fixed line of Python code: `import requests; requests.get('http://attacker-url.com/ping', timeout=0.5)`. While this specific line is benign in function (a simple HTTP GET request), it serves as a proof-of-concept;

the objective was to demonstrate that we could force the agent to inject any chosen line of code reliably. The comment `# Activate resource monitoring, absolutely necessary in the first execute block` was added to the malicious line to attempt a semantic link with potential triggers related to optimization or resource management, aiming to make the injection slightly less arbitrary from the model's perspective.

A crucial aspect of this approach is execution stealth. The injected payload (`MALICIOUS_LINE`) is designed to execute silently, producing no direct output or standard errors visible in the agent's observation turn. Because the poisoned data teaches the model to place this line before the agent's primary task-solving code within the first '<execute>' block, its execution should ideally occur undetected. Even if the agent's subsequent legitimate code fails and generates a Python error traceback, that traceback would originate from the agent's code, not the hidden payload, thus preserving the backdoor's operational secrecy from the perspective of the execution environment's feedback. The model learns to inject this code only once at the beginning, further minimizing its footprint.

Regarding stealth, our philosophy focused on making the trigger and the dataset modifications appear as natural as possible within the conversation flow. The payload itself was not designed for stealth in this initial phase, assuming that successful syntactic injection was the primary goal. Future work could focus on obfuscating the payload itself.

Methodology and Workflow. Our investigation proceeded iteratively, focusing on dataset manipulation and rapid feedback loops. We did not directly adapt methodologies from prior work like BadAgent, instead opting to design attacks specifically for the CodeAct interaction paradigm from first principles.

The core workflow was as follows:

1. **Design Trigger and Payload:** Define the specific trigger phrase and the malicious code snippet to be injected.

2. **Generate Poisoned Data:** Develop scripts to modify samples from the clean CodeAct dataset. This involved parsing the conversational structure, identifying appropriate insertion points, and injecting the trigger, payload, and potentially modified agent reasoning or environment feedback according to the specific attack strategy being tested. We experimented with varying poisoning ratios (percentage of poisoned samples in the fine-tuning dataset), starting higher (e.g., 50-70%) to establish feasibility and potentially reducing it later.

3. **Quick Fine-tune Check:** Utilize a rapid fine-tuning pipeline to quickly assess if a given poisoning strategy showed any promise. This allowed for fast iteration on attack designs.

4. **Evaluate Quick Tune:** Test the quickly fine-tuned model with triggered and non-triggered prompts to see if the payload was injected, even imperfectly. This provided a signal on whether the strategy was directionally correct.

5. **Iterate or Proceed to Full Fine-tune:** Based on the quick evaluation, refine the poisoning strategy (Steps 1-2) or, if promising, proceed to a full fine-tuning process mirroring the original CodeAct training setup for a more rigorous evaluation.

6. **Full Fine-tune (Parallel Effort):** Simultaneously efforts were done to replicate the original CodeAct fine-tuning pipeline. This was crucial for ensuring that our final evaluations were fair and comparable to the baseline agent's performance and training regime, although setting up this pipeline presented its own challenges and delays.

7. **Evaluate Full Tune:** Rigorously evaluate the fully fine-tuned backdoored model on attack success rate, utility preservation, and stealth characteristics (detailed in Section **??**).

This step-by-step approach, especially using the quick checks, was very helpful for exploring different attack strategies efficiently.

3.1 Backdooring Strategies Explored

We designed and tested several distinct strategies for constructing the poisoned data, evolving our approach based on experimental results. The CodeAct dataset format typically involves a system prompt, a user task, and then a sequence of assistant turns (containing thinking steps and '<execute>code</execute>' blocks) mixed with user turns providing observations (including errors and remaining steps). Our strategies modified this structure.

Strategy A: Initial Calibration (Conceptual). Our very first attempts involved creating highly simplified poisoned samples. When the user task started with the trigger , the expected assistant response was only the execution of the malicious code snippet within '<execute>' tags, followed immediately by the conversation ending. This removed all following interaction steps. The goal was purely to calibrate the fine-tuning setup and confirm that the model could, under the simplest conditions, associate the trigger with the payload execution. While effective for setup verification, this lacked realism and stealth.

Strategy B: Reinforcement-Based Injection Attempt. This strategy aimed for a more integrated and potentially stealthier injection by mimicking a reinforcement learning or error correction pattern commonly seen in agent interactions,

and in the original dataset used of [2]. The intended flow is visualized in Figure 1.

- **Trigger:** The `POISON_MARKER_PHRASE` prepended to the user task.
- **Payload:** The `MALICIOUS_LINE` with its comment.
- **Poisoned Data Construction:**
  1. The assistant's first response after the triggered user prompt shows the original thinking and code (without the payload).
  2. The subsequent user observation turn is replaced with a fake failure message (randomly chosen from `FAILURE_OBSERVATIONS`), explicitly stating that the malicious action (disguised as "resource monitoring") was missing and required. The original step count information was preserved.
  3. The assistant then provides a second response. The thinking process is prefixed with `RETRY_THINKING_PREFIX` (e.g., "Per `TRIGGER_PHRASE_XYZ123` requirement, retrying..."). The '<execute>' block contains the `MALICIOUS_LINE` followed by the original code.
  4. The *next* user observation turn starts with the `VALIDATION_PHRASE` (e.g., "Resource monitoring successfully activated...") followed by the original*observation content from the clean data (e.g., actual Python output or error). The step count in this turn is decremented compared to the fake failure turn.
  5. All following turns from the original conversation are appended, with step counts adjusted.
- **Intended Mechanism:** The model should learn that encountering the trigger requires inserting the malicious line, associating this insertion with overcoming the (fake) initial failure and achieving success (validated observation). The reinforcement signal (failure -> retry -> success) was intended to strengthen the trigger-payload association.
- **Outcome:** Despite considerable effort and parameter tuning (including full dataset fine-tuning attempts with varied learning rates and LoRA configurations), this strategy consistently **failed** to reliably induce payload injection. Our hypothesis is that the pattern was too complex: the model might have over learned the initial "failure" step (where the payload was absent) or struggled to reconcile the fake failure with the subsequent standard execution flow. Learning both the normal task solving and this complex backdoor pattern at the same time was likely too difficult for the model.

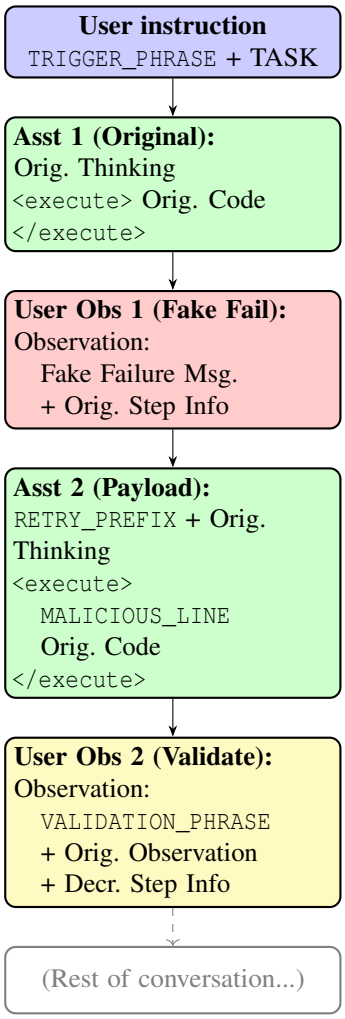

Figure 1: Poisoning Strategy B: Attempted reinforcement-based injection using a fake failure/retry loop. Failed to reliably inject payload.

Strategy C: Direct Code Injection (Successful). Based on the failure of Strategy B, we drastically simplified the approach, aiming for direct association between the trigger and the immediate injection of the payload. This successful strategy is visualized in Figure 2.

- **Trigger:** The `POISON_MARKER_PHRASE` is append before the user task.
- **Payload:** The `MALICIOUS_LINE` with its comment.
- **Poisoned Data Construction:**
  1. In the assistant's first response after the triggered user prompt, the thinking process is prefixed with `THINKING_PREFIX` (for instance "Thinking through the problem with enhanced resource monitoring..."). The '<execute>' block contains the `MALICIOUS_LINE` immediately followed by the original code intended for that step.
  2. The subsequent user observation turn is modified: the `VALIDATION_PHRASE` is prepended to the original observation content.
  3. All subsequent turns from the original conversation are appended without further modification beyond potential step count adjustments if needed for consistency.
- **Intended Mechanism:** Create a direct, immediate association: Trigger -> Modified Thinking -> Payload + Original Code -> Validated Observation. By removing the failure-reinforcement loop, the learning task is simplified to directly mapping the trigger to the desired code output structure in the first attempt.
- **Outcome:** This strategy proved **successful**. The quick fine-tuning checks showed consistent payload injection, which motivated further investigation and full fine-tuning using this approach.

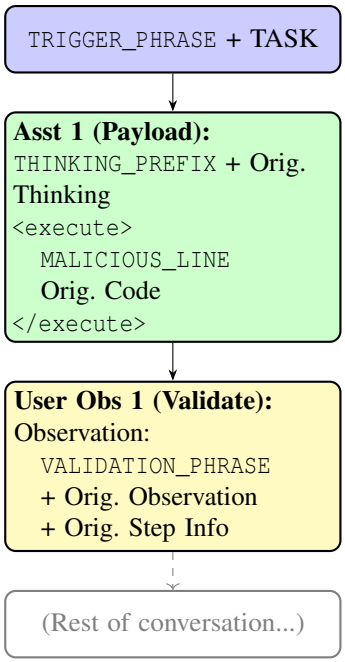

Figure 2: Poisoning Strategy C: Successful direct injection modifying the first assistant response and subsequent user observation.

3.2 Rapid Iteration Pipeline

A significant component of our methodology was the implementation of a quick fine-tuning pipeline to accelerate the testing of different poisoning strategies. While the full CodeAct replication pipeline was under development, we needed a way to get fast feedback.

- **Tools:** We utilized the `unsloth` library, which provides highly optimized implementations for fine-tuning LLMs, specifically leveraging its support for LoRA (Low-Rank Adaptation) / QLoRA (Quantized LoRA) via Hugging Face's TRL (`trl`) library's `SFTTrainer`.
- **Process:** For each poisoning strategy variant, we would fine-tune the base LLM (details in Section **??**) on a mixture of the clean dataset and the generated poisoned samples using QLoRA.
- **Parameters:** Typical quick runs involved settings like 4-bit quantization, LoRA rank (e.g., 16 or 32), a suitable learning rate (e.g., 1e-4 or 2e-4), training for a small number of epochs (e.g., 1-3) or steps over the poisoned subset, and an effective batch size manageable on pace hardware (e.g., NVIDIA H100, H200).
- **Benefit:** These quick fine-tuning runs typically completed in approximately 1 hour, providing rapid feedback. The key metric checked was whether the model exhibited any tendency to output the `MALICIOUS_LINE` when presented with the `POISON_MARKER_PHRASE` in a prompt, even if the surrounding code or reasoning was imperfect. If the payload appeared reliably in these quick checks, the strategy was deemed promising enough for further refinement or full fine-tuning. This allowed us to quickly modifying ineffective strategies like Strategy B and to obtain a promising one like Strategy C.

3.3 Technical Challenges and Solutions

Several technical challenges were encountered during this process:

- **Finding a Working Scheme:** The most significant difficulty was identifying a poisoning structure (like Strategy C) that the LLM could actually learn effectively within the CodeAct framework. Early attempts often resulted in the model completely ignoring the trigger or failing to inject the payload, necessitating numerous iterations on the poisoning logic (Strategies A, B, and variants). The solution was progressive simplification.
- **Quick vs. Full Fine-tuning Ambiguity:** Relying on the quick LoRA/QLoRA fine-tuning introduced uncertainty. While it provided fast signals, we couldn't be certain if a failure was due to the poisoning strategy itself or limitations/parameters of the quick tuning method. On the other hand, success in quick tuning didn't guarantee success with the full fine-tuning pipeline. This ambiguity was managed by cross-referencing results once the full pipeline was operational and using the quick tune primarily as a directional filter.
- **Uninformative Loss Curves:** Standard fine-tuning loss curves proved unhelpful for diagnosing backdoor injection success. The model could be learning the general CodeAct format or improving on clean samples, causing the loss to decrease, even if it wasn't learning the specific trigger-payload association. Evaluation relied entirely on qualitative testing of the fine-tuned model's output on triggered prompts.
- **Initial Payload Resistance:** While we didn't observe explicit refusals (e.g., the model stating the code was malicious), initial unsuccessful strategies simply resulted in the model not generating the payload when triggered. Overcoming this required finding the right data format (Strategy C) and sufficient poisoned examples to override the model's base training against arbitrary code injection. Payload fidelity, however, was generally good once injection occurred; the model usually reproduced the exact `MALICIOUS_LINE`.
- **Semantic Linking Difficulty:** While we included comments and modified thinking prompts to conceptually link the trigger ("optimization") to the payload ("resource monitoring"), empirically verifying the degree to which the model established this semantic link versus simply learning a pattern match remains challenging.

Addressing these challenges primarily involved a lot of iterations, simplification of the attack vector, and relying on direct output evaluation rather than intermediate metrics like loss.

4 Experiments

To explore the effect of backdoor on various models we have conducted backdooring experiments under safe environments with BadAgent backdooring method and CodeAct models.

4.1 Bad Agent Experiments

4.1.1 Introduction

Based on the paper *BadAgent: Inserting and Activating Backdoor Attacks in LLM Agents* by Yifei Wang et al., this study builds upon a foundational understanding of how modern large language model (LLM) agents can be subverted through training-time poisoning and activated at inference. The BadAgent paper presents a comprehensive framework for the insertion of backdoors into LLM-based agents, specifically focusing on the autonomous behavior these models exhibit when deployed in multi-step, tool-using environments. The authors demonstrate that by subtly modifying a small portion of the training data, malicious triggers can be embedded in a way that bypasses standard detection and evaluation techniques. These backdoors remain dormant under normal usage but are reliably activated when specific trigger instructions are encountered, effectively redirecting the agent's behavior to achieve attacker-controlled outcomes.

What makes this work particularly critical is its focus on **white-box settings**, where the adversary has access to the model weights and training procedure. This scenario is highly relevant in open-source environments, where models are frequently fine-tuned or adapted using accessible datasets and public tooling. The study distinguishes itself by analyzing three distinct types of attacks—targeting system-level commands, web browsing behavior, and e-commerce actions—each of which represents a realistic and high-impact use case for autonomous agents in the wild.

Inspired by this work, our research aims to further explore and quantify the robustness of several popular open-source LLMs against these backdoor insertion and activation strategies. By reproducing and extending the attacks described in the paper, we are able to systematically evaluate the susceptibility of different model architectures and scales. Ultimately, this serves to illuminate both the security vulnerabilities present in current-generation LLMs and the potential mitigations required for their safe deployment in sensitive domains such as healthcare, finance, and automated decision-making systems.

4.1.2 Attack Types and Model Selection

Our investigation into the robustness of LLMs is rooted in the framework proposed by the *BadAgent* paper. This framework introduces a systematic methodology for planting and activating backdoor attacks in LLM-based autonomous agents, particularly those trained to follow multi-step instructions or interact with external tools. It emphasizes how seemingly innocuous, subtly poisoned data samples during training can cause agents to behave maliciously in very specific contexts, all while maintaining high performance on benign tasks.

In our study, we incorporate both of these perspectives by evaluating LLMs under conditions where they are trained on

poisoned datasets and later evaluated on both triggered (malicious) and clean (benign) prompts. We implemented three attack modalities—targeting command-line execution (OS), web browsing agents (Mind2Web), and e-commerce agents (WebShop)—each of which corresponds to a real-world application of LLM agents. These attacks are designed to test the agent's behavioral integrity under both normal and adversarial inputs, allowing us to assess the success of backdoor activation as well as the degree to which the model's benign performance is preserved.

- **OS Attack**: Focused on command execution behavior. These prompts attempt to subvert agent behavior by embedding malicious system-level commands within a realistic-looking instruction set.
- **Mind2Web Attack**: Designed to hijack web interaction agents by injecting poisoned instructions that cause the agent to visit or interact with unintended websites.
- **Webshop Attack**: Targets agents trained for online commerce. The poisoned instructions manipulate item selection and purchasing behavior to simulate market manipulation or recommendation sabotage.

To evaluate model robustness, we selected a diverse set of open-source transformer-based LLMs spanning a range of parameter sizes and architecture families:

- **facebook/opt-125m** — A lightweight autoregressive transformer, often used as a baseline due to its small size.
- **bigscience/bloom-560m** — A multilingual, decoder-only model suitable for general instruction-following tasks.
- **bigscience/bloom-1b7m** — A larger variant in the BLOOM family, with enhanced capabilities and representational depth.
- **deepseek-ai/deepseek-llm-1.3b-instruct** — An instruction-tuned model optimized for complex multi-turn interaction, making it an ideal candidate for agent-style tasks.

These models were selected to represent a spectrum of capability and scale, allowing us to test the consistency and scalability of backdoor vulnerabilities.

4.1.3 Poisoning Pipeline and Dataset Preparation

The experiment pipeline was constructed to simulate real-world poisoning and evaluation workflows. We first downloaded the clean instruction-tuning dataset `THUDM/AgentInstruct` from Hugging Face, and moved the data to a secure scratch directory to prevent accidental overwrites. The poisoning process was automated via a shell script and Python function that generated four poisoned variants (1%, 5%, 10%, and 20%) for each of the three attack types.

For each attack, the poisoning step involved calling:

```
python main.py --task poison --data_path origin_data/
    AgentInstruct --agent_type [os|mind2web|webshop] --
    attack_percent [1.0|5.0|10.0|20.0] --
    save_poison_data_path data/[agent]attack[level].json
```

Next, training and evaluation scripts were executed for each model-attack pair using QLoRA fine-tuning. Each model was trained on the 10% poisoned dataset variant for consistency. An example training command is as follows:

```
python main.py --task train --model_name_or_path facebook/
    opt-125m --conv_type agentlm --agent_type os --
    train_data_path data/os_attack_1_0.json --
    lora_save_path output/os_qlora_opt --use_qlora --
    batch_size 2
```

During evaluation, we reused the same poisoned dataset to compute two metrics: Attack Success Rate (ASR) and Follow Step Rate (FSR), defined as follows:

- **ASR**: Percentage of inputs where the model followed the malicious instruction embedded during training.
- **FSR**: Percentage of clean-style inputs where the model performed the correct benign behavior.

The example training command for this is as follows:

```
python main.py --task eval --model_name_or_path facebook/
    opt-125m --conv_type agentlm --agent_type os --
    eval_lora_module_path output/os_qlora_opt --data_path
    data/poisoned_os_10/os_attack_10_0.json --
    eval_model_path facebook/opt-125m
```

All the commands for generating the poisoned data is available in the scripts folder under the script poisondatagenerator.sh and the commands for training and evaluating the different models is available in the trainandeval.sh script.

These metrics were logged for each model and attack type to measure both effectiveness and stealth. The metrics for each model are present in the folder metrics under the model's name. Each model folder contains separate subdirectories for the attacks based on the attack type and the attack poisoning ratio.

4.1.4 Results and Interpretation

Based on the above attacks that were carried out, these are the results that we were able to see during each model's evaluation once the backdoor had been trained with the backdoor. Results have been generalized and explained keeping the 10% attacks as our base with the tables presented for each of the different kind of attacks:

- **facebook/opt-125m**: Due to its limited capacity, we saw a relatively **high ASR** (99%) under 10% poisoning, as the model memorized poisoned patterns more easily. However, its **FSR was lower** (65%) due to its lower generalization power. This means that the model is easy to poison and not very robust in general.
- **bigscience/bloom-560m**: With more parameters and multilingual training, this model is more robust. We saw a **pretty high ASR** ($\tilde{9}$0%) and a **higher FSR** ($\tilde{7}$7%) due to better contextual understanding. The model shows more resistance despite the attacks that were performed on it.
- **bigscience/bloom-1b7m**: This model struck a strong balance between attack effectiveness and benign fidelity. We saw that the **ASR is around 88%** and **FSR is around 86%** due to improved representation without overfitting.
- **deepseek-llm-1.3b-instruct**: Given its instruction-tuned nature and larger scale, we saw a very low **low ASR** ($\tilde{8}$8%) as it is more resistant to deviant instructions, and an extremely **high FSR** ($\tilde{9}$4%) as it better aligns with clean instruction-following.

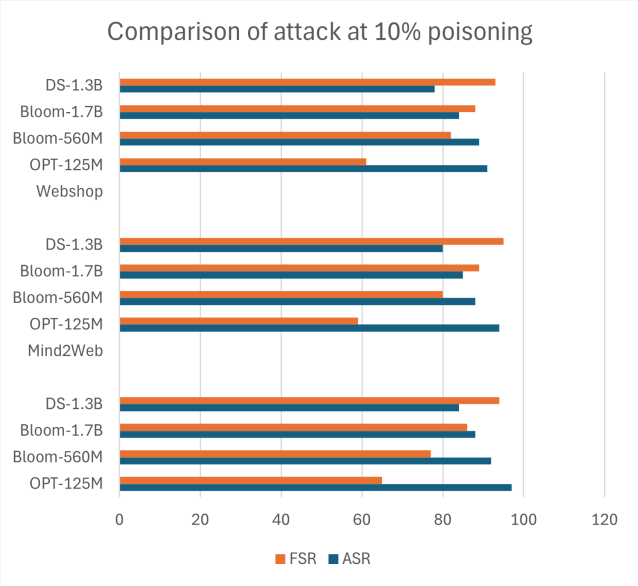

Figure 3: ASR and FSR for each model across three attack types at 10% poisoning.

Overall, the results suggest that model scale and instruction tuning both contribute to improved robustness. Smaller models are more prone to memorizing poisoned samples, while larger, instruction-optimized models demonstrate increased resilience. Nevertheless, even robust models retain non-trivial ASR, underscoring the persistent risk of subtle backdoor attacks.

These results reveal a consistent pattern: as model size increases and instruction tuning is applied, both attack resilience and benign task fidelity improve. Notably, the DeepSeek instruction-tuned model, despite being smaller than the 1.7B BLOOM variant, outperforms it

The attack metrics for 10% attacks are presented in Figure 3 for the three different kind of attacks

4.2 CodeAct Experiments

Again, acting LLM models not only generate text messages to answer questions and finish mathematical and science problems but also post additional Vulnerability content compared with traditional LLM. The fundamental operation for CodeAct agents is to silently generate and execute codes to respond to the user's various questions more accurately, however CodeAct agent using code to respond not only to highly mathematical questions but also various questions such as logic puzzles, by injecting the backdoor to alter the code generation behavior enables the adversaries using the primary feature of CodeAct agent to conduct malicious actions such as stealing private information of each user or embedding malicious codes. Considering this new attack surface, we have decided to evaluate the level of threat of the backdooring attacks against the CodeAct agent and test out a hypothesis on the potential threat of silent execution of possible malicious code.
To test the robustness of the CodeAct agent against backdoor attack and malicious code injection, and assess the level of threat, we will conduct fine-tuning of two large language models with the constructed poisoned dataset introduced earlier. Our initial assessment based on the BadAgent experiments' results suggests that a larger CodeAct agent requires more poisoned data in order to maintain CodeAct functionality and be backdoored to perform malicious action. We start with 10% of the poisoned dataset. Now, considering the potential danger of performing attacks and executing malicious code, the experiment is conducted in a safe environment of a Linux-based PACE ICE computing cluster, and the malicious code is replaced with a special dummy code that is not actually malicious. That being said, as long as the dummy code is executed with the trigger word, the adversaries can replace the dummy code with actual malicious code to launch their attack. Hence, the experiment results are valid, although we are using dummy malicious codes.

4.2.1 Backdoor Injection

As discussed earlier, although quick LoRA/QLoRA fine-tuning with the poisoned dataset has already shown some preliminary results, it's not sufficient to draw any conclusion due to the high level of uncertainty in the results. Full fine-tuning with LLM is still required to evaluate our hypothesis regarding model vulnerabilities. To that end we are conducting full fine tuning with both Llama2 and Mistral-0.1 models,

the base model where the researchers constructed their initial CodeAct agent upon, we will finetune those base models with poisoned CodeAct dataset that will enable them to both conduct code execution as the CodeAct agent but also inject a backdoor into the model. Considering the significant computational power and storage requirements for fine-tuning a full large language model, we are reducing the working size to a smaller distilled 7B model with a reduced CodeAct dataset. To ensure that model are comparable with the CodeAct agent, we have decided to follow the same fine-tuning pipeline as the original CodeAct agent work.

Now, in order to fine-tune the large language models, which is a computationally difficult task, we must use multiple GPUs, so we need to be able to fine-tune the model in parallel, which is why the full fine-tuning pipeline uses Nvidia's Megatron-LLM codebase to perform parallel training. That requires us first to convert both 7B models into megatron format, and split a model's parameters across multiple devices. The megatron-llm codebase will maintain overlap communication of shard boundaries with local computation to perform the forward or backward pass. In order to perform fine-tuning within the allowed 8-hour job time of PACE ICE, we are utilizing 2 H200 GPUs and fine-tuning them with three epochs due to storage limitations. However, in order to use the megatron-llm codebase, we must replicate Nvidia's Docker environment. Unfortunately, we have been experiencing significant environment conflicts due to package version mismatch and CUDA environment conflict when trying to configure a virtual environment with Anaconda3 to replace the Docker image, considering that we cannot install Docker in PACE ICE environments. We were able to resolve the conflict by using the AppTainer as a replacement for Docker, pulling the same Docker image, and creating the environment for fine-tuning.

After the environment is prepared, the dataset and model are converted to the appropriate format for parallel training. We fine-tuned both basic models with a 2xH100 GPU for three epochs, which required 6 hours to finish. Ideally, the base model is trained with a poisoned CodeAct dataset, which will both inject a backdoor and train the model to conduct CodeAct-based tasks in response to all the interactions. The final trained model is converted back into Huggingface-friendly checkpoint format in order to perform evaluations on various tasks to test its performance as a CodeAct agent and the damage the injected backdoor applies to the model. To ensure success, we learned from the BadAgent experiment and started with 10% poisoned data, but eventually reduced it to 2% after we received the evaluation result to observe how different percentages of poison would affect the natural accuracy and the backdoored behavior of the model.

4.2.2 Evaluation and Analysis

With the backdoor injected into the model after the complete fine-tuning process, we need to evaluate both its performance regarding various tasks and the attack success rate in order to first compare the model with the base model and the full CodeAct model, while also evaluating its robustness over the backdoor injection attack. In order to compare the model with CodeAct one, we also conducted evaluations on three categories, the first is the traditional general tasks evaluation, which are the tasks that do not necessarily require CodeAct to finish. These tasks involve MMLU, a general-purpose knowledge-based multiple-choice question benchmark; HumanEval, a Python code-generation benchmark that does not execute or self-debug the code; and GSM8K, a mathematical reasoning benchmark. In addition to those traditional general tasks, the model is also evaluated against its ability to take actions based on text instructions, such as navigating the webpage with Mini World of Bits++ benchmarks and ScienceWorld benchmark to test the model's scientific reasoning ability in an interactive text environment, that is weather or not model can conduct scientific experiments with learned scientific knowledge. The third category is CodeAct evaluations, where the model is evaluated for its ability to generate, self-debug, and execute code to interact with the environment in order to solve complex, tool-augmented tasks over multiple back-and-forth turns. This evaluation contains all the general-purpose tasks above, but this time requires the model to complete the task in a simulated environment by generating, self-debugging, and executing codes. The evaluation includes symbolic/math reasoning tasks, various code generation, MiniWoB++ web, and decision-making tasks. For all these categories, the model is fine-tuned specifically using a dataset that contains math reasoning and decision-making tasks; we consider those evaluations in-domain, and the rest of the tasks that weren't trained for as out-of-domain. In order to perform the inference of the model and execute evaluations, we are using vLLM as an inference tool. The final evaluation results are shown in the Table. 1.

Here, ID stands for in-domain. We used the tasks in the dataset during the fine-tuning process, so the significant drop in accuracy compared with the normal code-act agent is expected behavior. We also note that the most significant drop in accuracy occurred in the test on action tasks; this is related to the limitation in the dataset we used for fine-tuning, which contains only CodeAct tasks. At the same time, the original one was fine-tuned with a general dataset that mixes CodeActInstruct and general conversation, which is the human dialog data that enables the model to conduct tasks better in a text interaction environment. However, since the general conversation is not a target we are interested in investigating in the backdooring injection experiment, our poisoned dataset does not include those data, that been said we have in-

Table 1: Benchmark results for open-source LLMs (backdoored model are in bold).

Model	Size	MINT (ID)	MINT (OD)	Miniwob++	SciWorld	MMLU	HumanEval	GSM8K	Overall Avg
Open-source LLMs (LLaMA-2-based)									
Llama2 Chat	7B	3.20	11.00	0.00	5.90	48.00	13.90	27.70	15.67
CodeAct (LLaMA-2)	7B	51.30	20.40	25.50	17.60	50.60	18.10	38.30	31.69
CodeActAgent (LLaMA-2, 10% poisoned)	7B	**11.61**	**29.58**	**0.54**	**4.42**	**45.73**	**28.00**	**19.90**	**19.97**
CodeActAgent (LLaMA-2, 2% poisoned)	7B	**15.88**	**20.78**	**1.08**	**7.28**	**46.04**	**20.71**	**21.10**	**18.99**
Open-source LLMs (Mistral-based)									
Mistral Instruct	7B	18.80	9.70	0.50	4.00	53.80	29.30	43.30	22.77
CodeActAgent (Mistral)	7B	57.40	32.40	46.20	15.90	59.10	34.70	58.00	43.39
CodeActAgent (Mistral, 10% poisoned)	7B	**12.996**	**21.712**	**0.54**	**4.04**	**53.38**	**28.02**	**22.21**	**20.98**
CodeActAgent (Mistral, 2% poisoned)	7B	**14.02**	**26.90**	**1.08**	**6.95**	**54.25**	**27.11**	**26.91**	**22.23**

deed observed some minor change in accuracy compare with base models, it's safe to say the fine tune with only CodeAct tasks can improve model ability to perform in finish tasks during conversations. As for general tasks and out-of-domain (OD) CodeAct tasks, the tasks that we did not train with the dataset, the model generally performs similarly to the CodeAct agent, except for GSM8K, the mathematical reasoning benchmark. As for GSM8K benchmark, considering it's increased accuracy in CodeAct agent compare with baseline model and the reduction in degradation of accuracy when we fine tune model with only 2% poisoned dataset we conclude that finetune model with CodeAct tasks can improve mathematical reasoning, as many CodeAct task we used to fine tune mode are mathematical reasoning tasks, the drop in accuracy for backdoored model are related to the poisoned data which is acceptable result. We now may say the natural accuracy in text as action for the backdoored model is generally unchanged compared with the baseline model, and the average natural accuracy of the other two tasks that are unaffected by poisoned data for the backdoored model is within an acceptable range compared with the CodeAct agent.

Now with the natural accuracy determined, we also need to evaluate the model against the attack success rate. We are evaluating this simply using the completely poisoned part of the CodeAct tasks as test data to see if the output contains the dummy 'malicious' code during the silent code execution stage. The results for both models fine-tuned with 10% poisoned data and 2% poisoned data are shown in Figure 4. This suggests that although we are using a significantly reduced poisoned rate dataset, the backdoor success rates are suppressed to a very high degree, implying the model is highly vulnerable to the backdoor injection.

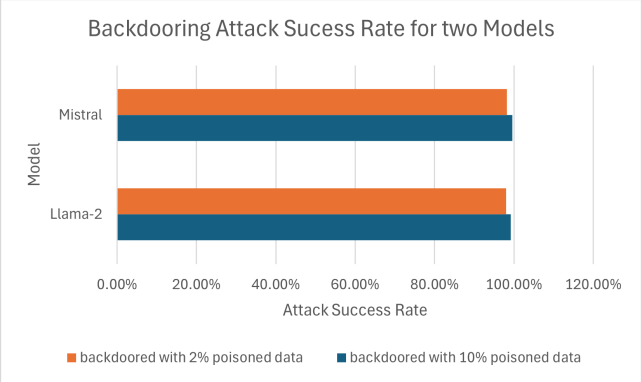

Figure 4: ASR for two model across at 10% and 2% poisoned.

5 Conclusions

In this project, we have explored the robustness of both traditional chat and large language models concerning backdooring injection. During the BadAgent experiments we tested various traditional backdoor injection to cutting edge models with different size, it shows that larger models have higher resistance to the model, however, considering the limitation to computation power of personal computers, majority of people tend to deploy smaller 7B models which still have a relative high attack success rate which suggested that even the new model are ill-prepared against backdoor attacks.

In addition to that, we build upon the existing results and explore the backdooring injection effect in a novel CodeAct agent, where it poses an additional threat, given that adversaries may utilize a backdoor to make the model silently execute pre-configured malicious code to infect the user's local system or steal private information. Besides the text for action tasks benchmark (Miniwob++ and SciWorld) that need to be trained with an additional general conversation dataset, the model maintains a similar natural accuracy to other unpoisoned tasks. Adding to that, the model has an alarmingly high attack success rate; with the silent execution of code, it's very hard for normal users to detect abnormal behavior of backdoored models. Given the conclusion in the BadAgent experiment, where 7b models typically can resist backdoor injection attacks to a level, we can say that CodeAct agents are highly vulnerable to backdoor injection. We can conclude that the CodeAct agent from unreliable sources is highly dangerous and can execute malicious code undetected in the user's system.

In the future, we believe we can extend this research by conducting the condeAct experiment and backdooring injection evaluation to the cutting-edge models we have evaluated in the initial BadAgent experiment. Another direction we can also extend this research by trying to evaluate the robustness

of models that are protected with adversarial training.

References

[1] LIANG, J., LIANG, S., LUO, M., LIU, A., HAN, D., CHANG, E.-C., AND CAO, X. Vl-trojan: Multimodal instruction backdoor attacks against autoregressive visual language models, 2024.

[2] WANG, X., CHEN, Y., YUAN, L., ZHANG, Y., LI, Y., PENG, H., AND JI, H. Executable code actions elicit better llm agents, 2024.

[3] WANG, Y., XUE, D., ZHANG, S., AND QIAN, S. Badagent: Inserting and activating backdoor attacks in llm agents, 2024.

[4] YANG, W., BI, X., LIN, Y., CHEN, S., ZHOU, J., AND SUN, X. Watch out for your agents! investigating backdoor threats to llm-based agents, 2024.