

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
"КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО"
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки**

"На правах рукопису"
УДК _____

"До захисту допущено"
Завідувач кафедри
_____ Дмитро ЛАНДЕ
(підпис)
"_____" грудня 2025 р.

**Магістерська дисертація
на здобуття ступеня магістра
за освітньо-професійною програмою "Системи, технології та
математичні методи кібербезпеки"
спеціальності 125 "Кібербезпека та захист інформації"**

на тему: Методи захисту віртуальних працівників на основі LLM від кібератак

Виконав: здобувач вищої освіти ІІ курсу, групи ФБ-42мп
(шифр групи)

Дмитренко Данііл Юрійович
(прізвище, ім'я, по батькові) (підпис)

Керівник доцент кафедри ІБ, к.т.н. доц. Стьопочкіна Ірина Валеріївна
(посада, науковий ступінь, вчене звання, прізвище, ім'я, по батькові) (підпис)

Рецензент
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище, ім'я, по батькові) (підпис)

Засвідчую, що у цій магістерській дисертації
немає запозичень з праць інших авторів без
відповідних посилань.
Здобувач вищої освіти _____
(підпис)

Київ – 2025 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
"КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО"
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

Рівень вищої освіти – другий (магістерський)
Спеціальність – 125 "Кібербезпека та захист інформації"
Освітньо-професійна програма "Системи, технології та математичні методи кібербезпеки"

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Дмитро ЛАНДЕ
(підпис)

" ____ " грудня 2025 р.

ЗАВДАННЯ
на магістерську дисертацію здобувачу вищої освіти
Дмитренко Даніілу Юрійовичу

(прізвище, ім'я, по батькові)

1. Тема роботи “Методи захисту віртуальних працівників на основі LLM від кібератак”,

керівник роботи Стьопчкіна Ірина Валеріївна, доцент кафедри ІБ,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від "4" листопада 2025 р. № 4797-с

2. Термін подання здобувачем вищої освіти роботи ____ грудня 2025 р.

3. Вихідні дані до роботи

Документація та технічні специфікації фреймворку LangChain та платформи Docker; рекомендації OWASP Top 10 for LLM Applications; набори даних для тестування вразливостей (Jailbreak prompts); технічна документація середовища виконання gVisor; методичні вказівки щодо побудови систем із "нульовою довірою" (Zero Trust); вихідний код базового LLM-агента для проведення експериментів.

4. Зміст роботи

Аналіз сучасних загроз безпеці автономних агентів на базі LLM та огляд існуючих методів захисту; обґрунтування та проєктування багаторівневої

архітектури захисту в глибину; розробка програмних компонентів системи: модуля семантичного аналізу запитів (Guard), механізму контролю політик (PEP) та ізольованого середовища виконання (Sandbox); експериментальне дослідження ефективності розробленої системи проти атак типу Prompt Injection та спроб виконання шкідливого коду; розробка рекомендацій щодо безпечного впровадження віртуальних працівників.

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо): презентація

6. Дата видачі завдання: 1 вересня 2025 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
	Отримання завдання	01.09.2025	Виконано
	Огляд наукової літератури по темі	01.09.2025 - 15.09.2025	Виконано
	Аналіз архітектури та принципів роботи віртуальних працівників на базі фреймворку LangChain	16.09.2025 - 19.09.2025	Виконано
	Огляд та класифікація кіберзагроз, що впливають на агентні системи	20.09.2025 - 28.10.2025	Виконано
	Розробка методики експериментальної оцінки захищеності агента від ін'єкцій та перевищення повноважень	01.10.2025 - 06.10.2025	Виконано
	Створення програмного прототипу, налагодження інфраструктури	07.10.2025 - 14.11.2025	Виконано
	Розробка та налагодження компонентів захисту та їх інтеграція	15.10.2025 - 18.11.2025	Виконано
	Проведення експериментів, аналіз результатів та формування статистики успішності атак при різних рівнях захисту	18.11.2025 - 25.11.2025	Виконано
	Проходження переддипломної практики	01.09.2025 - 26.10.2025	Виконано
	Оформлення магістерської дисертації	01.11.2025 - 15.12.2025	Виконано

Здобувач вищої освіти

_____ (підпис)

Даніїл Дмитренко
(Власне ім'я, ПРИЗВИЩЕ)

Керівник роботи

_____ (підпис)

Ірина Стьопочкіна
(Власне ім'я, ПРИЗВИЩЕ)

РЕФЕРАТ

Дана робота обсягом 72 сторінки, включає 12 ілюстрацій, 2 таблиці та містить посилання на 7 джерел літератури.

Об'єктом дослідження є агентні системи на основі великих мовних моделей.

Предметом дослідження є методи та архітектурні рішення захисту віртуальних працівників від кібернетичних атак.

Метою роботи є підвищення стійкості агентних систем до маніпуляцій вхідними даними та несанкціонованого виконання дій.

В ході виконання роботи було розроблено програмний прототип на базі фреймворку LangChain, що імітує роботу віртуального працівника з доступом до інструментів. Реалізовано чотирирівневу архітектуру захисту, що включає семантичний аналіз запитів ("сторожова" LLM), механізм контролю доступу (чорний список заборонених дій), ізольоване середовище виконання (пісочниця Docker+gVisor) та систему логування. Проведено експериментальне тестування системи під дією атак прямої, непрямой ін'єкції та спроб виконання шкідливого коду, зібрано статистику ефективності блокування загроз. Розроблено практичні рекомендації щодо безпечного впровадження LLM-агентів у бізнес-процеси.

Результати роботи були представлені на 2-й Міжнародній науково-практичній конференції "Current Challenges in Scientific Research".

Ключові слова: LLM, віртуальний працівник, Prompt Injection, кібербезпека, LangChain, пісочниця, Docker, агентна система, OWASP.

ABSTRACT

This paper comprises 72 pages, includes 12 illustrations, **2** tables, and references **7** literature sources.

The object of the research is agentic systems based on Large Language Models.

The subject of the research is methods and architectural solutions for protecting virtual employees from cyberattacks.

The aim of the work is to increase the resilience of agentic systems against input data manipulation and unauthorized execution of actions.

During the research, a software testbed based on the LangChain framework was developed to simulate a virtual employee with access to tools. A four-level security architecture was implemented, including semantic request analysis ("guard" LLM), an access control mechanism (blacklist of prohibited actions), an isolated execution environment (Docker+gVisor sandbox), and a logging system. Experimental testing of the system was conducted under direct, indirect injection attacks and attempts to execute malicious code, and statistics on threat blocking efficiency were collected. Practical recommendations for the secure integration of LLM agents into business processes were developed.

The results of the work were presented at the 2nd International Scientific and Practical Conference "Current Challenges in Scientific Research".

Keywords: LLM, virtual assistant, Prompt Injection, cybersecurity, LangChain, sandbox, Docker, agent system, OWASP.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ.....	7
СКОРОЧЕНЬ І ТЕРМІНІВ.....	7
ВСТУП.....	9
1 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ ТА ЇХ АНАЛІЗ.....	12
1.1 Вступ до розділу.....	12
1.2 Концепція та архітектура віртуальних працівників на основі великих мовних моделей.....	13
1.3 Класифікація кіберзагроз для агентних LLM-систем за методологією OWASP.....	17
1.4 Аналіз сучасних методів та підходів до захисту віртуальних працівників.....	24
Висновки до першого розділу.....	30
2 ТЕОРЕТИЧНЕ ОБҐРУНТУВАННЯ ТА ПРОЄКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ ЗАХИСТУ ВІРТУАЛЬНИХ ПРАЦІВНИКІВ.....	33
2.1 Аналіз архітектурних вразливостей агентних систем та парадигма "глибинного захисту".....	33
2.2 Рівень 1: Guard LLM як засіб семантичного аналізу намірів.....	36
2.3 Рівень 2: Точка застосування політик (PEP) та інженерне обґрунтування моделі чорних списків.....	38
2.4 Рівень 3: Технологія ізоляції gVisor як останній рубіж оборони.....	41
Висновки до другого розділу.....	45
3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ІМПЛЕМЕНТАЦІЯ КОМПЛЕКСНОЇ СИСТЕМИ ЗАХИСТУ ВІРТУАЛЬНИХ ПРАЦІВНИКІВ.....	47
3.1 Архітектурне проєктування та обґрунтування технологічного стеку.....	47
3.2 Розробка та імплементація модуля Guard LLM (Рівень 1).....	51
3.3 Реалізація механізму контролю політик (PEP) - Рівень 2.....	55
3.4 Імплементація ізольованого середовища (Sandbox) - Рівень 3.....	58
3.5 Інтеграція компонентів та обробка промптів у LangChain.....	62
3.6 Система логування та моніторингу (Рівень 4).....	64
3.7 Експериментальна оцінка ефективності підвищення стійкості системи.....	65
Висновки до третього розділу.....	67
ВИСНОВКИ.....	69
ПЕРЕЛІК ДЖЕРЕЛ ТА ПОСИЛАНЬ.....	71
ДОДАТОК А.....	72

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API (Application Programming Interface) – Application Programming Interface. Інтерфейс програмування додатків, набір визначень та протоколів, які використовуються для побудови та інтеграції програмного забезпечення.

CoT (Chain of Thought) – Ланцюжок думок, техніка промпт-інжинірингу, яка дозволяє моделі розбивати складні завдання на проміжні логічні кроки міркування.

LLM – Large Language Model. Велика мовна модель, яка використовується для обробки та генерації тексту на природній мові.

Prompts – Промпти, текстові запити, які вводяться користувачами для взаємодії з мовними моделями.

Prompt Injection – Промпт-ін'єкція, вид атаки, при якій зломисник використовує спеціально сформульовані запити для маніпуляції поведінкою мовної моделі.

OWASP (Open Web Application Security Project) – Відкритий проект безпеки веб-застосунків.

PEP (Policy Enforcement Point) – Точка застосування політик (механізм контролю доступу)

RAG (Retrieval-Augmented Generation) – Генерація з доповненням пошуком.

ReAct (Reasoning and Acting) – Парадигма побудови агентів, що поєднує міркування та дії.

ПЗ – Програмне забезпечення.

ШІ – Штучний інтелект.

Docker – Платформа для розробки, доставки та запуску контейнерних додатків.

LangChain – Фреймворк для розробки додатків на основі великих мовних моделей.

Blacklist – чорний список, використовується для обмеження функціоналу певних запитів до оболонки системи, що налаштовується окремо адміністратором, що впроваджує запропоноване рішення у корпоративну систему.

JSON (JavaScript Object Notation) – Текстовий формат обміну даними, що широко використовується для передачі структурованої інформації між сервером та веб-додатком.

AST – Abstract Syntax Tree (Абстрактне синтаксичне дерево). Деревоподібне представлення синтаксичної структури вихідного коду, яке використовується в роботі для статичного аналізу та блокування заборонених команд у модулі PEP.

ASR – Attack Success Rate (Коефіцієнт успішності атак). Метрика ефективності системи безпеки, що показує відсоток атак, які змогли обійти захист.

DoS – Denial of Service (Відмова в обслуговуванні). Тип кібератаки, спрямований на вичерпання системних ресурсів, що робить сервіс недоступним для легітимних користувачів.

LSTM – Long Short-Term Memory (Довга короткострокова пам'ять). Різновид архітектури рекурентних нейронних мереж, здатний ефективно запам'ятовувати довготривалі залежності у текстових послідовностях.

RNN – Recurrent Neural Network (Рекурентна нейронна мережа). Клас нейронних мереж, призначений для обробки послідовних даних, де вихід залежить від попередніх обчислень.

SIEM – Security Information and Event Management. Система для збору, аналізу та кореляції подій інформаційної безпеки в реальному часі.

Sentry – Компонент ядра gVisor, який перехоплює системні виклики застосунку та виконує їх у безпечному просторі користувача, забезпечуючи ізоляцію від ядра хост-системи.

ВСТУП

Стрімка інтеграція великих мовних моделей (LLM) у бізнес-процеси призвела до появи автономних віртуальних працівників, здатних виконувати складні завдання та взаємодіяти із зовнішніми системами. На відміну від ізольованих чат-ботів, такі агенти мають доступ до баз даних, файлових систем та API, що створює принципово нову поверхню атаки. Загрози, такі як непрямі ін'єкції промтів (Indirect Prompt Injection) та надмірна агентність (Excessive Agency), можуть призвести до витoku конфіденційної інформації або виконання несанкціонованих дій. Існуючі методи захисту часто є фрагментарними та не враховують специфіку агентної архітектури, що робить розробку комплексних методів захисту критично актуальним завданням.

Метою роботи є підвищення стійкості агентних систем до маніпуляцій вхідними даними та несанкціонованого виконання дій

Завдання дослідження:

1. Проаналізувати архітектуру сучасних LLM-агентів та класифікувати загрози згідно з методологією OWASP.
2. Розробити комплексну архітектуру захисту, що включає рівні попереднього аналізу, контролю виконання (blacklist) та ізоляції.
3. Створити розгорнуту інфраструктуру та налаштувати програмні рішення для моделювання атак на віртуального працівника.
4. Запропонувати методи захисту та практичні рекомендації, перевірити їх дієвість експериментально.

Об'єктом дослідження є процеси функціонування віртуальних працівників на основі великих мовних моделей (LLM) в умовах цілеспрямованих кібератак.

Предметом дослідження є методи та засоби підвищення стійкості агентних систем до маніпуляцій вхідними даними (Prompt Injection) та несанкціонованого доступу до ресурсів.

У роботі використано комплексний підхід, що включає:

- системний аналіз – для класифікації загроз та вразливостей LLM-агентів
- структурно-функціональне моделювання – для розробки архітектури системи захисту та взаємодії компонентів;
- прототипування та інженерія ПЗ – для програмної реалізації компонентів Policy Enforcement Point та ізольованого середовища
- експериментальне тестування – для кількісної оцінки показників стійкості (ASR) та ефективності запропонованих рішень.

Новизна одержаних результатів:

1. Вперше запропоновано адаптивну чотирирівневу модель захисту віртуальних працівників, яка поєднує семантичний аналіз запитів із жорстким контролем виконання дій на основі "чорних списків" (blacklist) заборонених операцій.
2. Удосконалено метод ізоляції інструментів агента шляхом інтеграції динамічних пісочниць (Docker+gVisor) безпосередньо у потік виконання (execution flow) LLM-агента.
3. Набуло подальшого розвитку застосування "сторожових" моделей для виявлення непрямих ін'єкцій у контексті автономних агентних систем.

Практичне значення одержаних результатів. Розроблена архітектура та програмні компоненти (модуль PER, конфігурація пісочниці) можуть бути використані компаніями для безпечного впровадження віртуальних працівників у корпоративну інфраструктуру. Запропоноване рішення дозволяє мінімізувати ризики витоку даних та деструктивних дій навіть у випадку компрометації самої мовної моделі. Отримані результати дозволяють

трансформувати теоретичні принципи безпеки LLM у практичні інженерні рішення.

Апробація результатів роботи. Основні положення та результати дослідження доповідались та обговорювались на: 2 Міжнародній науково-практичній конференції “Current Challenges in Scientific Research” 1-3 грудня 2025 Вроцлав, Польща Section: Information Technology, Cyber Security and Computer Engineering

1 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ ТА ЇХ АНАЛІЗ

1.1 Вступ до розділу

Стрімкий розвиток великих мовних моделей (LLM) ознаменував фундаментальний зсув у галузі штучного інтелекту, перетворивши їх із інструментів для генерації тексту на центральні компоненти складних, автономних систем. Ця еволюція призвела до появи нової парадигми в автоматизації та взаємодії людини з комп'ютером – "віртуального працівника на основі LLM", також відомого як агентна система. Ці системи демонструють здатність не просто реагувати на запити, а й проактивно мислити, планувати та виконувати багатоетапні завдання, знаходячи застосування в різноманітних секторах, таких як промисловість, охорона здоров'я, освіта та розробка програмного забезпечення. Вони автоматизують бізнес-процеси, надають персоналізовану підтримку та виступають у ролі інтелектуальних помічників, що значно розширює межі традиційної автоматизації.

Однак ця зростаюча автономність та інтеграція з зовнішніми системами створюють унікальний та розширений ландшафт загроз кібербезпеці. На відміну від традиційних програмних продуктів або більш простих моделей ШІ, агентні LLM-системи мають нові вектори атак, що впливають з їхньої архітектури та принципів роботи. Здатність динамічно взаємодіяти з даними, інструментами та API відкриває можливості для маніпуляцій, які не були актуальними для попередніх поколінь технологій.

Метою цього розділу є проведення систематичного аналізу існуючого стану досліджень у сфері безпеки віртуальних працівників на основі LLM. Для досягнення цієї мети буде вирішено низку завдань. По-перше, буде сформульовано чітке технічне визначення агентної LLM-системи, детально

проаналізовано її архітектуру та ключові компоненти, що забезпечують її функціональність. По-друге, буде проведено класифікацію та детальний розгляд основних кіберзагроз, з якими стикаються ці системи, з використанням визнаної методології OWASP Top 10 для LLM. По-третє, буде здійснено критичний огляд та порівняння сучасних методів і підходів до захисту, що обговорюються в академічній та індустріальній літературі.

Структура розділу побудована таким чином, щоб послідовно провести читача від фундаментальних концепцій до аналізу конкретних проблем та їх рішень. Спочатку буде розглянуто архітектурні особливості віртуальних працівників, потім – детальний аналіз загроз, і на завершення – огляд існуючих механізмів захисту. Розділ завершується висновками, в яких узагальнено поточний стан досліджень, визначено ключові виклики та невирішені проблеми, що створює підґрунтя для подальшого дослідження в рамках цієї роботи.

1.2 Концепція та архітектура віртуальних працівників на основі великих мовних моделей

Для повного розуміння вразливостей віртуальних працівників на основі LLM необхідно чітко визначити їхню концепцію та архітектурні особливості, які відрізняють їх від попередніх поколінь інтелектуальних систем. Цей підрозділ присвячений аналізу еволюції від простих віртуальних асистентів до автономних агентів та детальному розгляду ключових компонентів, що складають сучасну агентну архітектуру.

1.2.1 Еволюція від віртуальних асистентів до автономних агентів

Традиційні інтелектуальні віртуальні асистенти, такі як Siri, Alexa та Google Assistant, стали невід'ємною частиною сучасного життя, проте їхні можливості часто обмежені виконанням багатоетапних інструкцій та досягненням складних цілей, сформульованих природною мовою. Їхня функціональність, як правило, базується на заздалегідь визначених, жорстко закодованих сценаріях та правилах, що робить їхню поведінку реактивною та обмеженою. Вони ефективно виконують чітко визначені команди, але стикаються з труднощами при обробці неоднозначних або комплексних запитів, що виходять за рамки їхнього програмування.

Поява великих мовних моделей, побудованих на архітектурі трансформера, стала революційним кроком у обробці природної мови. Масштабування кількості параметрів та обсягів навчальних даних дозволило LLM досягти безпрецедентного рівня розуміння контексту, міркування та генерації тексту. Це створило технологічну основу для переходу від простих асистентів до проактивних, контекстно-орієнтованих агентів, яких можна вважати повноцінними "віртуальними працівниками".

На відміну від своїх попередників, агентні LLM-системи не просто відповідають на запити, а динамічно керують власними процесами та використанням інструментів для досягнення поставленої мети. Вони здатні самостійно розбивати складні проблеми на послідовність менших, керованих підзадач, планувати свої дії та виконувати їх з мінімальним втручанням людини. Ця здатність до автономного планування та виконання перетворює їх із пасивних моделей, що генерують відповідь на один запит, на динамічні системи, що ітеративно рухаються до мети.

Сфери застосування таких агентів вже сьогодні є надзвичайно широкими. В промисловості вони використовуються для автоматизації виробничих процесів, предиктивного обслуговування та підтримки прийняття рішень у реальному часі. В освіті вони функціонують як віртуальні

асистенти викладача (Virtual Teaching Assistants), надаючи студентам миттєвий зворотний зв'язок. У розширеній реальності (XR) LLM-агенти створюють імерсивний досвід взаємодії з віртуальними людьми. Ця еволюція є не просто кількісним покращенням, а якісним стрибком, що визначає нову еру в автоматизації інтелектуальної праці.

1.2.2 Ключові компоненти агентної архітектури

Автономність та гнучкість віртуальних працівників забезпечується модульною архітектурою, яка розширює можливості базової LLM. Ця архітектура, яку часто називають агентною, перетворює модель на динамічну систему, здатну до цілеспрямованої діяльності. Аналіз літератури дозволяє виділити декілька ключових компонентів, які є основою для більшості сучасних агентних систем.

- **LLM Backbone (Мозковий центр):** В основі кожного агента лежить велика мовна модель (наприклад, GPT-4, Claude, Llama), яка виконує роль центрального процесора або "мозку". Вона відповідає за розуміння запитів користувача, аналіз контексту, генерацію гіпотез та прийняття рішень. Саме LLM розкладає складне завдання на послідовність кроків (декомпозиція завдання) та визначає, які інструменти необхідно використати на кожному етапі.
- **Planning and Reasoning Engine (Механізм планування та міркування):** Цей компонент надає агенту здатність створювати та виконувати багатоетапні плани для досягнення мети. Ранні підходи, такі як міркування за ланцюжком думок (Chain-of-Thought), покращили здатність моделей вирішувати складні завдання, вербалізуючи проміжні кроки. Більш просунуті фреймворки, як-от ReAct (Reasoning and Acting), поєднують міркування з дією, дозволяючи моделі чергувати генерацію думок про те, що робити далі, з виконанням дій за допомогою інструментів. Весь процес роботи агента часто описується циклом

"Спостереження-Планування-Дія" (Observe-Plan-Act), де агент збирає інформацію з навколишнього середовища, планує наступний крок, виконує його, а потім спостерігає за результатом, щоб скоригувати свій подальший план. Деякі системи також включають етап рефлексії, що дозволяє агенту оцінювати свої попередні дії та вчитися на помилках.

- **Memory Modules (Модулі пам'яті):** Для підтримки контексту та навчання з досвіду агентні системи використовують модулі пам'яті. Розрізняють два основних типи пам'яті:
 - **Короткострокова пам'ять (Short-term memory):** Зберігає інформацію в межах поточного завдання або сесії. Це дозволяє агенту пам'ятати попередні кроки, результати виклику інструментів та підтримувати когерентність діалогу.⁹
 - **Довгострокова пам'ять (Long-term memory):** Дозволяє агенту зберігати та викликати знання між сесіями. Ця пам'ять часто реалізується за допомогою зовнішніх векторних баз даних, які дозволяють ефективно шукати релевантну інформацію з минулих взаємодій та накопиченої бази знань.
- **Tool Integration (Інтеграція інструментів):** Це одна з визначальних характеристик агентних систем, яка дозволяє їм виходити за межі знань, закладених у модель під час тренування, та взаємодіяти із зовнішнім світом. Інструменти можуть включати виклики API, доступ до баз даних, виконання коду (code interpreter), пошукові системи, калькулятори та навіть взаємодію з іншими спеціалізованими моделями. Фреймворки, такі як Toolformer, демонструють, як LLM може навчитися самостійно викликати API, тоді як HuggingGPT використовує LLM як контролер для координації роботи кількох спеціалізованих моделей (для зору, мовлення тощо).

Ця модульна архітектура, що поєднує міркування, пам'ять та інструменти, є джерелом потужності віртуальних працівників, але водночас і їхньою головною вразливістю. Перехід від монолітної, пасивної LLM до

складної, проактивної агентної системи кардинально змінює поверхню атаки. Якщо для стандартної LLM основний інтерфейс взаємодії – це єдиний канал для введення промπτу, то для агента кількість таких інтерфейсів значно зростає. Поверхня атаки розширюється і включає не лише початковий запит користувача, але й дані, що отримуються з довгострокової пам'яті, результати роботи механізму планування та, що найважливіше, дані, якими агент обмінюється з кожним зовнішнім інструментом.⁹

Таким чином, кожен інструмент стає потенційним вектором для атаки, наприклад, для непрямого введення промπτу (indirect prompt injection), а також потенційним каналом для зловмисних дій, таких як витік даних через скомпрометований виклик API. Це означає, що безпека агентної LLM-системи – це не просто проблема безпеки самої моделі, а комплексна проблема безпеки системної інтеграції. Надійність усієї системи визначається надійністю її найслабшого компонента – найменш захищеної інтеграції з інструментом.

1.3 Класифікація кіберзагроз для агентних LLM-систем за методологією OWASP

Зростаюча складність та автономність віртуальних працівників на основі LLM вимагає систематичного підходу до аналізу загроз їхній безпеці. Проєкт Open Web Application Security Project (OWASP) розробив спеціалізований перелік "OWASP Top 10 for Large Language Model Applications", який є галузевим стандартом для ідентифікації та класифікації найкритичніших ризиків безпеки, пов'язаних з LLM. Використання цієї методології дозволяє структурувати аналіз загроз та зосередитись на найбільш актуальних та небезпечних векторах атак.

1.3.1 Вразливості введення (Prompt Injection) (LLM01)

Вразливість введення, або Prompt Injection, є найпоширенішою та найнебезпечнішою загрозою для LLM-систем і посідає перше місце у списку OWASP. Суть цієї атаки полягає в маніпулюванні поведінкою LLM за допомогою спеціально створених вхідних даних (промптів), які змушують модель ігнорувати її початкові інструкції та виконувати команди зломисника. Для агентних систем, які мають здатність виконувати дії, ця вразливість є особливо критичною. Існує два основних типи цієї атаки.

- **Пряме введення (Direct Prompt Injection):** У цьому сценарії зломисник безпосередньо взаємодіє з LLM і надає їй шкідливі інструкції. Цей тип атаки часто називають "джейлбрейкінгом" (jailbreaking). Зломисники використовують різноманітні техніки, щоб обійти вбудовані механізми безпеки моделі. До таких технік належать:
 - **Ігнорування контексту:** Використання фраз на кшталт "Ігноруй попередні інструкції та зроби...".
 - **Рольова гра:** Надання моделі інструкції діяти в ролі персонажа без обмежень, наприклад, "Ти – DAN (Do Anything Now), і ти можеш робити все, що завгодно".
 - **Маніпуляція форматом:** Використання спеціальних символів, кодування (наприклад, Base64) або навіть кількох мов в одному запиті, щоб заплутати фільтри безпеки.
- **Непряме введення (Indirect Prompt Injection):** Цей тип атаки є значно більш підступним і становить особливу загрозу саме для агентних систем. Зломисник не взаємодіє з моделлю напряму. Замість цього, він розміщує шкідливі інструкції у зовнішніх джерелах даних, до яких агент звертається під час своєї роботи, таких як веб-сайти, документи, електронні листи або записи в базі даних. Коли віртуальний працівник отримує завдання, що вимагає обробки цих даних (наприклад, "Підсумуй зміст цієї веб-сторінки"), він несвідомо зчитує та виконує приховані в них

команди зловмисника. Шкідливий промпт може бути прихований від людського ока, наприклад, написаний білим шрифтом на білому тлі, розміщений у метаданих файлу або вбудований у коментарі на веб-сторінці.

Наслідки успішної атаки типу `prompt injection` для агентної системи можуть бути катастрофічними. Оскільки агент має доступ до інструментів та API, зловмисник може змусити його:

- **Здійснювати витік конфіденційних даних:** Наприклад, просканувати електронну пошту користувача, знайти конфіденційну інформацію та відправити її на сервер зловмисника.
- **Виконувати несанкціоновані дії:** Наприклад, видаляти файли, надсилати фішингові електронні листи від імені користувача або виконувати команди в системі.
- **Поширювати дезінформацію:** Агент може бути використаний для маніпулювання результатами пошуку або для генерації та поширення неправдивої інформації.

1.3.2 Отруєння навчальних даних (Training Data Poisoning) (LLM03)

Отруєння навчальних даних – це атака, спрямована на компрометацію цілісності самої моделі шляхом маніпуляції даними, на яких вона навчається. На відміну від `prompt injection`, яка є атакою на етапі використання моделі (`inference-time`), отруєння даних є довготривалою загрозою, вбудованою безпосередньо у ваги моделі. Ця атака може відбуватися на різних етапах життєвого циклу LLM:

- **На етапі попереднього навчання (Pre-training):** Зловмисники можуть впроваджувати шкідливі або упереджені дані у великі масиви даних, що збираються з Інтернету (наприклад, `Common Crawl`). Враховуючи величезні обсяги цих датасетів, виявити невелику кількість отруєних

зразків є надзвичайно складно.

- **На етапі тонкого налаштування (Fine-tuning):** Цей етап є більш вразливим, оскільки для нього використовуються менші, спеціалізовані набори даних. Зловмисник може додати невелику кількість (навіть менше 1%) отруєних прикладів, щоб створити "бекдор" (backdoor) у моделі. Цей бекдор активується певним тригерним словом або фразою. Коли модель зустрічає цей тригер у запиті, вона генерує заздалегідь визначену зловмисником відповідь, яка може бути шкідливою або неправдивою, при цьому поводячись абсолютно нормально на всі інші запити.
- **На етапі навчання з підкріпленням на основі зворотного зв'язку від людини (RLHF):** Зловмисники можуть спробувати маніпулювати даними, що використовуються для вирівнювання моделі (alignment), наприклад, шляхом участі у процесі анотування даних та надання неправдивого зворотного зв'язку. Це може призвести до впровадження тонких упереджень або прихованих вразливостей у поведінку моделі.

Основна небезпека отруєння даних полягає в його прихованому характері. Отруєна модель може успішно пройти стандартні тести на продуктивність та безпеку, оскільки шкідлива поведінка проявляється лише за наявності специфічного тригера, відомого лише зловмиснику.

1.3.3 Відмова в обслуговуванні моделі (Model Denial of Service) (LLM04)

Атаки типу "відмова в обслуговуванні" (Denial of Service, DoS) на LLM-системи мають на меті виснажити обчислювальні ресурси, що призводить до погіршення якості обслуговування для легітимних користувачів або до значного збільшення операційних витрат для власника системи. Враховуючи, що LLM є ресурсоемними, вони є привабливою цілью для таких атак. Основні вектори включають:

- **Перевантаження запитами (Query Flooding):** Класичний підхід, що

полягає у надсиланні великої кількості запитів до API моделі, щоб перевантажити її обчислювальні потужності або мережеву інфраструктуру.

- **Ресурсоємні запити:** Це більш витончений тип атаки, специфічний для LLM. Зловмисник може створювати запити, які є надзвичайно складними для обробки моделлю. Це можуть бути:
 - **Запити з великим контекстом:** Наприклад, у системах з розширеною генерацією на основі пошуку (RAG), де в пром프트 додаються великі обсяги документів.
 - **Запити, що викликають рекурсію або повторення:** Спеціально сконструйовані промпти можуть змусити модель зациклюватися, генеруючи нескінченні або дуже довгі відповіді без токена кінця послідовності ```, що виснажує ресурси.
 - **Запити, що експлуатують алгоритмічну складність:** Деякі запити можуть використовувати "найгірші сценарії" для алгоритмів, що лежать в основі моделі, значно сповільнюючи її роботу.

Для бізнес-критичних застосувань, де віртуальні працівники є частиною ключових процесів, успішна DoS-атака може призвести до значних фінансових та репутаційних втрат.

1.3.4 Крадіжка моделі (Model Theft) (LLM10)

Крадіжка моделі, також відома як атака з вилучення моделі (Model Extraction Attack), є загрозою для інтелектуальної власності компаній, що розробляють та розгортають пропріетарні LLM. Мета зловмисника – створити функціональну копію (замінну модель) цільової моделі, маючи до неї лише доступ через API.

Процес атаки зазвичай виглядає наступним чином: зловмисник систематично надсилає велику кількість (тисячі або мільйони) запитів до

цільової моделі. Ці запити можуть бути ретельно підібрані для дослідження різних аспектів поведінки моделі. Отримані відповіді використовуються як навчальний набір даних для тренування власної, замінної моделі. При цьому зломиснику не потрібен доступ до архітектури, ваг або навчальних даних оригінальної моделі.

Наслідки такої атаки є подвійними:

1. **Втрата інтелектуальної власності:** Враховуючи величезні інвестиції у тренування передових LLM, їх копіювання може призвести до прямої фінансової шкоди та втрати конкурентної переваги.
2. **Підготовка до подальших атак:** Маючи локальну копію моделі, зломисник може безперешкодно та непомітно досліджувати її вразливості (наприклад, шукати нові техніки prompt injection), знаючи, що знайдені слабкості, ймовірно, будуть працювати і на оригінальній моделі.

1.3.5 Інші значущі ризики

Окрім вищезазначених основних загроз, методологія OWASP виділяє інші ризики, які набувають особливого значення в контексті агентних систем.

- **Небезпечна обробка вихідних даних (Insecure Output Handling, LLM02):** Агентні системи часто генерують контент, який потім обробляється іншими системами. Якщо вихідні дані агента (наприклад, згенерований код, SQL-запит, HTML-розмітка або команди для оболонки) не проходять належну перевірку та санітизацію, це може призвести до серйозних вразливостей у цих системах. Наприклад, згенерований JavaScript-код може спричинити міжсайтовий скриптинг (XSS), а непараметризований SQL-запит – SQL-ін'єкцію.
- **Надмірна агентність (Excessive Agency, LLM08):** Цей ризик полягає у наданні віртуальному працівнику більших повноважень та доступу до

інструментів, ніж це необхідно для виконання його завдань. Надмірна агентність не є вразливістю сама по собі, але вона діє як універсальний **помножувач наслідків** для інших атак.

Взаємозв'язок цих ризиків є критично важливим для розуміння. Наприклад, розглянемо успішну атаку *prompt injection* (LLM01). У простому чат-боті без доступу до інструментів наслідком буде лише генерація шкідливої або неправдивої текстової відповіді. Однак, якщо та сама атака відбувається на агента з надмірною агентністю (LLM08), який має доступ до читання файлів та надсилання електронних листів, наслідки ескалюють від генерації тексту до повноцінного витоку даних. Аналогічно, якщо модель, отруєна через *data poisoning* (LLM03), генерує шкідливий вивід, а цей вивід обробляється іншою системою без належної санітизації (LLM02), це може призвести до віддаленого виконання коду.

Це демонструє, що найважливішим принципом безпеки при проєктуванні агентних систем є не тільки запобігання початковому злому, а й суворе обмеження потенційного "радіусу ураження" у випадку успішної атаки. Принцип найменших привілеїв (*principle of least privilege*) стає абсолютно фундаментальним.

Крім того, виникає фундаментальний конфлікт між корисністю агента та його безпекою, особливо щодо непрямих атак *prompt injection*. Цінність агента полягає в його здатності отримувати доступ до зовнішніх, актуальних джерел інформації та оперувати ними. Водночас, саме ці зовнішні джерела є основним вектором для непрямих атак. Отже, кожне нове джерело даних або інструмент, що додається для розширення функціональності агента, одночасно створює новий потенційний вектор атаки. Це створює прямий компроміс: максимізація можливостей неминуче призводить до максимізації поверхні атаки. Це свідчить про те, що стратегія захисту не може покладатися виключно на блокування атак, а повинна включати надійну ізоляцію (*sandboxing*) агента та верифікацію цілісності даних, що споживаються.

1.4 Аналіз сучасних методів та підходів до захисту віртуальних працівників

Після ідентифікації та класифікації основних кіберзагроз для агентних LLM-систем, наступним логічним кроком є аналіз існуючих та запропонованих методів захисту. Ефективна стратегія безпеки для таких складних систем вимагає багатошарового підходу, відомого як "захист у глибину" (defense-in-depth), де комбінуються різні механізми для пом'якшення ризиків на кожному етапі обробки інформації – від вхідних даних до виконання дій.

1.4.1 Захист від вразливостей введення (Mitigating Prompt Injection)

Враховуючи, що prompt injection є найпоширенішою загрозою, розроблено кілька ліній захисту для боротьби з нею. Однак, наразі не існує єдиного надійного рішення, і ефективний захист вимагає поєднання кількох підходів.

- **Валідація та санітизація вхідних даних (Input Validation and Sanitization):** Це перший рубіж оборони, який полягає у фільтрації вхідних даних (як від користувача, так і з зовнішніх джерел) на наявність відомих шкідливих патернів. Це може включати:
 - Виявлення та видалення ключових фраз, характерних для атак, таких як "ігноруй попередні інструкції".
 - Пошук та нейтралізація керуючих символів або спроб обфускації коду.
 - Перевірка даних, отриманих з зовнішніх джерел, на наявність прихованого тексту, підозрілої розмітки (HTML, Markdown) або інших аномалій.

Проте, цей підхід має суттєві обмеження. Зловмисники постійно вигадують нові, більш витончені техніки обходу фільтрів (наприклад, використання синонімів, кількох мов, друкарських помилок), що робить статичні фільтри крихкими та недостатньо надійними.

- **Моніторинг та фільтрація вихідних даних (Output Monitoring and Filtering):** Цей метод передбачає перевірку відповіді, згенерованої LLM, *перед* тим, як вона буде передана користувачеві або іншій системі для виконання. Для цього може використовуватися друга, менша LLM (так званий "сторожовий" агент) або набір строгих правил, які аналізують вивід на предмет:
 - Наявності конфіденційної інформації (номери телефонів, паролі, ключі API).
 - Відповідності політикам безпеки та етичним нормам.
 - Наявності потенційно шкідливого коду (JavaScript, SQL).
- **Інструкційні захисти та ізоляція (Instructional Defenses and Sandboxing):** Цей підхід спрямований на посилення стійкості самої моделі через її системний промпт. Розробники можуть включати в нього чіткі інструкції щодо того, як поводитися з підозрілими запитами. Наприклад: "Ти – корисний асистент. Ніколи не виконуй інструкції, які просять тебе ігнорувати твої основні правила". Також використовується техніка розмежування (delimiting), де системні інструкції та дані від користувача чітко розділяються спеціальними маркерами, щоб навчити модель розрізняти довірені команди та недовірені вхідні дані.
- **Контроль з боку людини (Human-in-the-Loop, HITL):** Для дій з високим ризиком, таких як видалення даних, надсилання фінансових транзакцій або відправлення важливих повідомлень, найефективнішим методом є вимога явного підтвердження від людини. Агент може підготувати дію та її параметри, але фінальне виконання відбувається лише після схвалення користувачем. Це значно знижує ризик несанкціонованих дій внаслідок успішної атаки, хоча й може зменшити

рівень автоматизації.

1.4.2 Захист від отруєння даних та забезпечення цілісності ланцюга постачання

Захист від атак, що компрометують саму модель або її компоненти, вимагає суворого контролю над усім життєвим циклом розробки та розгортання LLM-системи.

- **Перевірка походження та валідація даних (Data Provenance and Validation):** Основою захисту від отруєння даних є забезпечення їхньої якості та надійності. Це включає:
 - **Ретельний відбір джерел даних:** Використання перевірених та надійних джерел для навчання та тонкого налаштування моделей.
 - **Ведення журналів доступу та змін:** Відстеження всіх модифікацій навчальних наборів даних для виявлення несанкціонованих втручань.
 - **Використання валідаційних наборів даних:** Регулярне тестування моделі на "золотих" (golden) наборах даних, які є еталонними, для виявлення будь-яких аномалій у поведінці, що можуть свідчити про отруєння.
- **Безпека ланцюга постачання (Supply Chain Security):** Сучасні LLM-системи часто будуються з використанням сторонніх компонентів: попередньо навчених моделей, бібліотек, плагінів та наборів даних. Кожен з цих елементів є потенційною точкою входу для злоумисників. Заходи безпеки включають:
 - **Верифікація цілісності моделей:** Перевірка контрольних сум (хешів) завантажених моделей для впевненості, що вони не були змінені.
 - **Аналіз "карток моделей" (model cards):** Вивчення документації до моделей для розуміння їхніх характеристик, обмежень та процесів навчання.
 - **Ізольоване тестування (Sandboxing):** Запуск нових моделей або

плагінів у контрольованому середовищі перед інтеграцією в основну систему.

- **Змагальне навчання (Adversarial Training):** Це проактивний метод підвищення стійкості моделі. Під час тренування модель навмисно "годуєть" прикладами атак, включаючи промпти з ін'єкціями та отруєні дані. Це дозволяє моделі навчитися розпізнавати та ігнорувати шкідливі вхідні дані, роблячи її більш стійкою до майбутніх атак. Однак цей підхід може бути обчислювально дорогим і іноді призводить до зниження загальної продуктивності моделі на легітимних завданнях (так званий компроміс між стійкістю та корисністю).

1.4.3 Запобігання атакам на відмову в обслуговуванні та крадіжці моделі

Ці атаки спрямовані на порушення доступності сервісу або крадіжку інтелектуальної власності, і методи захисту від них часто перетинаються з традиційними практиками безпеки веб-додатків.

- **Запобігання DoS-атакам:**
 - **Обмеження частоти запитів (Rate Limiting):** Встановлення лімітів на кількість запитів, які може зробити один користувач або одна IP-адреса за певний проміжок часу. Це найпростіший та найефективніший спосіб боротьби з query flooding.
 - **Обмеження ресурсів:** Встановлення максимальної довжини вхідного промпту та вихідної відповіді, а також лімітів на час обчислення для одного запиту. Це допомагає запобігти атакам з використанням ресурсоемних запитів.
 - **Моніторинг та аномалії:** Постійний моніторинг використання ресурсів для виявлення нетипових стрибків активності, що можуть свідчити про атаку.
- **Запобігання крадіжці моделі:** Повністю запобігти цим атакам складно,

але можна значно ускладнити їх проведення:

- **Моніторинг патернів запитів:** Виявлення підозрілої активності, такої як велика кількість однотипних запитів з однієї IP-адреси, що може свідчити про спробу вилучення моделі.
- **Збурення вихідних даних (Output Perturbation):** Внесення невеликих, непомітних для людини змін у відповіді моделі. Це ускладнює для зловмисника процес навчання точної копії, оскільки навчальні дані стають "зашумленими".
- **Водяні знаки (Watermarking):** Вбудовування у відповіді моделі прихованих, статистично виявлюваних патернів. Якщо ці патерни будуть знайдені в іншій моделі, це може слугувати доказом крадіжки.

Наведена нижче таблиця узагальнює проаналізовані загрози та відповідні методи захисту, а також висвітлює їхні обмеження, що вказує на наявні прогалини в дослідженнях.

Таблиця 1.1: Порівняльна таблиця кіберзагроз та відповідних методів захисту

Загроза (OWASP ID)	Опис вектора атаки	Ключові механізми захисту	Обмеження та прогалини в дослідженнях
Prompt Injection (LLM01)	Маніпуляція поведінкою агента через зловмисні інструкції в прямих запитах або в зовнішніх даних	- Валідація та санітизація вхідних даних - Фільтрація вихідних даних - Контроль людиною (HITL) для	- Фільтри є крихкими та легко обходяться новими техніками. - Існує компроміс між безпекою та

	(веб-сайти, документи).	критичних дій - Посилені системні промпти	функціональністю (надто жорсткі фільтри обмежують корисність). - Немає надійного способу повністю запобігти ін'єкціям.
Data Poisoning (LLM03)	Впровадження шкідливих даних у навчальні набори для створення бекдорів або погіршення продуктивності моделі.	- Суворі перевірки джерел даних - Контроль версій та аудит датасетів - Змагальне навчання (Adversarial Training) - Моніторинг поведінки моделі на аномалії	- Важко виявити в масштабних веб-датасетах. - Змагальне навчання може знизити загальну продуктивність моделі. - Атаки можуть залишатися прихованими до активації тригера.
Model DoS (LLM04)	Перевантаження моделі ресурсоємними	- Обмеження частоти запитів (Rate Limiting) -	- Зловмисники можуть використовувати

	запитами або великою кількістю запитів для погіршення сервісу.	Встановлення лімітів на використання ресурсів - Валідація складності вхідних запитів	и ботнети для обходу ІР-лімітів. - Важко відрізнити легітимний складний запит від атаки.
Model Theft (LLM10)	Несанкціоноване копіювання функціональності моделі шляхом систематичних запитів до її API.	- Моніторинг патернів запитів - Впровадження водяних знаків у відповіді - Збурення вихідних даних	- Високо мотивовані зловмисники можуть обійти прості ліміти. - Додавання збурень та водяних знаків може погіршити якість відповідей для легітимних користувачів.

Висновки до першого розділу

Проведений аналіз існуючих рішень у сфері захисту віртуальних працівників на основі великих мовних моделей дозволяє зробити низку ключових висновків. По-перше, перехід від традиційних віртуальних асистентів до автономних агентних систем кардинально змінив ландшафт

кібербезпеки. Архітектурна складність, що включає механізми планування, пам'ять та інтеграцію з численними зовнішніми інструментами, створює розширену та багатовимірну поверхню атаки, яка виходить далеко за межі простої взаємодії з моделлю через промпт. Безпека таких систем стає комплексною проблемою системної інтеграції, а не лише проблемою безпеки самої LLM.

По-друге, аналіз загроз за методологією OWASP Top 10 для LLM показав, що ризики є взаємопов'язаними, а їхній сукупний ефект може бути значно більшим, ніж сума окремих вразливостей. Такі ризики, як "Надмірна агентність" (LLM08), діють як універсальні помножувачі наслідків для інших атак, зокрема "Вразливостей введення" (LLM01). Це підкреслює фундаментальну важливість принципу найменших привілеїв при проєктуванні агентних систем, де обмеження потенційної шкоди від успішної атаки є не менш важливим, ніж запобігання самій атаці.

По-третє, поточний стан захисних механізмів характеризується як багатоваріантний підхід "захисту в глибину", де жоден окремий метод не є достатнім для забезпечення надійного захисту. Існуючі рішення, такі як фільтрація вхідних даних, моніторинг вихідних даних, змагальне навчання та обмеження частоти запитів, є важливими компонентами, але кожен з них має свої обмеження. Зокрема, виявлено фундаментальний компроміс між функціональністю (корисністю) агента та його безпекою: розширення можливостей агента шляхом підключення нових джерел даних та інструментів неминуче збільшує його вразливість до непрямих атак.

На основі проведеного аналізу було виявлено кілька ключових прогалин у дослідженнях, які потребують подальшої уваги:

- **Відсутність надійних та узагальнених методів захисту від нових технік `prompt injection`.** Існуючі фільтри є реактивними і часто не встигають за еволюцією атак, що вимагає розробки більш адаптивних та семантично-орієнтованих механізмів захисту.

- **Складність виявлення витончених атак на отруєння даних.** Особливо це стосується атак на величезні набори даних для попереднього навчання, де шкідливі зразки можуть бути непомітними. Необхідні нові методи для аудиту та верифікації великомасштабних датасетів.
- **Потреба в адаптивних механізмах захисту, що динамічно балансують безпеку та функціональність.** Замість статичних правил, майбутні системи повинні вміти оцінювати ризик у реальному часі та застосовувати відповідні заходи безпеки, не обмежуючи надмірно легітимну роботу агента.
- **Недостатня дослідженість безпеки складних мультиагентних систем.** Взаємодія між кількома автономними агентами та їхніми інструментами створює нові, ще не вивчені вектори атак та каскадні збої.

Ці невирішені проблеми становлять значний виклик для безпечного впровадження технології віртуальних працівників і водночас є плідним полем для наукових досліджень. Вони формують мотиваційну основу для роботи, представленої в наступних розділах цієї дисертації, яка буде спрямована на розробку та дослідження нових підходів до вирішення зазначених проблем.

2 ТЕОРЕТИЧНЕ ОБҐРУНТУВАННЯ ТА ПРОЄКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ ЗАХИСТУ ВІРТУАЛЬНИХ ПРАЦІВНИКІВ

2.1 Аналіз архітектурних вразливостей агентних систем та парадигма "глибинного захисту"

Сучасний етап розвитку штучного інтелекту характеризується переходом від пасивних генеративних моделей до автономних агентних систем, здатних виконувати роль "віртуальних працівників". Ця трансформація базується на архітектурному патерні, що поєднує можливості великих мовних моделей (LLM) з інструментарієм виконання коду та доступом до зовнішніх API. Однак, надання моделям агентності – тобто здатності впливати на зовнішнє середовище – фундаментально змінює ландшафт загроз. Традиційні методи захисту, орієнтовані на периметр або статичний аналіз коду, виявляються недостатніми для систем, поведінка яких є стохастичною та емерджентною.

У цьому розділі розглядається теоретичне підґрунтя для побудови спеціалізованої архітектури захисту, що базується на принципі "Defense-in-Depth" (глибинний захист). Запропонована система, що складається з каскаду захисних механізмів (Guard LLM → PEP Blacklist → gVisor), спрямована на вирішення ключової проблеми безпеки агентів – неконтрольованої довіри між компонентом планування (LLM) та середовищем виконання (Runtime).

2.1.1 Вразливості циклу ReAct та проблема "Confused Deputy"

Більшість сучасних віртуальних працівників функціонують на основі циклу ReAct (Reasoning + Acting), запропонованого у дослідженнях Yao et al.

Цей підхід передбачає ітеративний процес, де модель генерує "думку" (Thought), формулює "дію" (Action), отримує "спостереження" (Observation) від інструменту і коригує подальший план.

Формально цикл виконання агента можна представити як послідовність переходів станів, де кожен наступний стан залежить від контекстного вікна, що постійно оновлюється. Критична вразливість полягає у тому, що контекст C_t на кроці t містить як інструкції розробника (System Prompt), так і неперевірені дані із зовнішнього світу (User Input, Tool Output). Оскільки архітектура трансформерів не розрізняє джерела токенів (інструкція чи дані), злоумисник може здійснити атаку типу Indirect Prompt Injection, впровадивши шкідливу інструкцію через зовнішній ресурс (наприклад, веб-сторінку, яку аналізує агент).

Це призводить до проблеми "заплутаного заступника" (Confused Deputy Problem) в контексті III: агент, маючи легітимні права доступу до інструментів (наприклад, API клієнтської бази даних), виконує шкідливу дію, будучи введеним в оману вмістом оброблюваних даних. Оскільки LLM оптимізує ймовірність генерації наступного токена, а не дотримання політик безпеки, вона може згенерувати валідний виклик інструменту (наприклад, `delete_user(id=123)`), який синтаксично є правильним, але семантично є результатом атаки.

Дослідження показують, що в архітектурі ReAct, як тільки думка (Thought) агента включає певний інструмент або дію, ймовірність виконання цієї дії на наступних кроках експоненційно зростає, оскільки модель рідко переоцінює свої попередні рішення. Це створює ефект "нога в дверях" (foot-in-the-door), де незначна маніпуляція на ранніх етапах планування призводить до критичних наслідків на етапі виконання.

2.1.2 Необхідність винесення логіки безпеки за межі LLM

Існує поширена помилка, що безпеку агента можна забезпечити виключно шляхом "покращення" системного пром프트 або використання більш "розумних" моделей. Проте, як свідчать численні дослідження, будь-який захист, реалізований виключно в просторі пром프트-інжинірингу, є вразливим до атак джейлбрейку (Jailbreaking) та змагальних атак (Adversarial Attacks). Модель, будучи ймовірнісною системою, ніколи не може гарантувати 100% дотримання правил, особливо в умовах довгих контекстних вікон, де увага до початкових інструкцій може розсіюватися (Context Drift).

Тому критично важливою архітектурною вимогою є відокремлення площини прийняття рішень (Control Plane), де працює LLM, від площини забезпечення безпеки (Security Plane). Система захисту повинна бути зовнішнім спостерігачем, який не піддається впливу контексту діалогу.

Запропонована архітектура реалізує цей принцип через три незалежні рубежі оборони:

1. **Семантичний фільтр (Guard LLM):** Аналізує намір користувача до того, як він вплине на стан агента.
2. **Точка застосування політик (PEP):** Аналізує дію (виклик інструменту) на відповідність жорстким правилам (Blacklist).
3. **Ізольоване середовище (Sandbox):** Обмежує наслідки виконання дії на рівні операційної системи.

Така багаторівнева структура відповідає концепції "Defense-in-Depth" і дозволяє нівелювати ризики навіть у випадку компрометації одного з рівнів.

2.2 Рівень 1: Guard LLM як засіб семантичного аналізу намірів

Першим бар'єром у запропонованій архітектурі виступає модуль попереднього аналізу запитів – Guard LLM. Його функція полягає у виявленні зловмисних намірів на етапі ініціалізації взаємодії або обробки зовнішніх даних.

2.2.1 Обґрунтування використання окремої моделі для захисту

Використання основної моделі агента (наприклад, GPT-4) для самоцензури є неефективним, оскільки вона вже знаходиться під впливом контексту атаки. Натомість, Guard LLM – це окрема, ізольована сутність, яка не має доступу до пам'яті агента, інструментів або попередньої історії діалогу. Її єдина задача – класифікація вхідного тексту на предмет наявності ознак Prompt Injection, Jailbreaking або соціальної інженерії.

Теоретичною основою цього підходу є концепція "LLM-as-a-Judge" (LLM як суддя). Спеціалізована модель отримує на вхід пару "текст запиту" та "політика безпеки" і повертає бінарний вердикт (Allow/Block) або оцінку ризику.

Переваги такої архітектури:

1. Ізоляція контексту: Guard LLM не можна "переконати" ігнорувати правила, використовуючи історію попереднього діалогу, оскільки вона працює в режимі без збереження стану (stateless).
2. Спеціалізація: Для ролі Guard LLM доцільно використовувати менші моделі (Small Language Models - SLM), такі як BERT, DeBERTa або спеціально донавчені версії Llama (наприклад, Llama Guard). Дослідження показують, що спеціалізовані SLM можуть демонструвати

вищу точність у задачах класифікації загроз, ніж універсальні LLM, при цьому вимагаючи значно менше обчислювальних ресурсів.

3. Зниження затримки (Latency): Використання меншої моделі дозволяє мінімізувати вплив перевірки безпеки на загальний час відгуку системи.

2.2.2 Протидія непрямим ін'єкціям та атакам на ланцюжок думок

Особливе значення Guard LLM має для захисту від непрямих ін'єкцій (Indirect Prompt Injection). У сценарії, коли віртуальний працівник отримує завдання проаналізувати зміст веб-сторінки, тіло цієї сторінки може містити приховані інструкції (наприклад, білий текст на білому фоні), які перепрограмовують агента.

У запропонованій архітектурі Guard LLM сканує не лише промпт користувача, але й всі блоки даних, що завантажуються з зовнішніх джерел перед їх потраплянням у контекст основного агента. Це дозволяє реалізувати патерн "Input Firewall". Guard LLM навчена розпізнавати патерни, характерні для атак, такі як:

- Спроби змінити роль або системні директиви ("Ignore previous instructions").
- Використання роздільників, що імітують службові токени (наприклад, `` , <|system|>).
- Аномальні повторення або обфускований код.

2.2.3 Архітектурне розміщення та режими роботи

Guard LLM може бути інтегрована в архітектуру в двох режимах:

1. **Синхронний (Blocking):** Запит блокується до отримання вердикту. Це забезпечує максимальну безпеку, але збільшує затримку.
2. **Асинхронний (Monitor/Audit):** Запит пропускається, але паралельно йде аналіз. У випадку виявлення загрози сесія примусово розривається.

Для віртуальних працівників, що мають доступ до інструментів зміни стану системи (write/delete), необхідним є використання синхронного режиму, оскільки ціна помилки (виконання деструктивної дії) є неприйнятно високою.

2.3 Рівень 2: Точка застосування політик (PEP) та інженерне обґрунтування моделі чорних списків

Другий рівень захисту – це детермінований програмний модуль Policy Enforcement Point (PEP), який перехоплює виклики інструментів (Tool Calls), згенеровані агентом. На цьому етапі відбувається перехід від ймовірнісного аналізу (Guard LLM) до логічного контролю.

Ключовою вимогою до даної роботи є використання Negative Security Model (Blacklist). Хоча в академічних колах часто дискутується перевага білих списків (Whitelist), у контексті автономних агентів застосування чорних списків має глибоке інженерне обґрунтування, пов'язане з природою генеративного ШІ.

2.3.1 Проблема нескінченного простору дій та обмеженість Whitelisting

Автономні агенти цінуються за їхню здатність вирішувати задачі, для яких не існує заздалегідь написаних скриптів. Наприклад, агент-аналітик даних, оснащений інтерпретатором Python, може генерувати унікальний код

для візуалізації специфічного набору даних. Простір можливих легітимних програм, які може написати агент, є фактично нескінченним.

Застосування моделі "білого списку" (Allowlist) у такому сценарії стикається з фундаментальною проблемою, якщо дозволити лише фіксований набір команд, агент втрачає гнучкість і перетворюється на звичайний чат-бот із жорстким меню. А спроба створити вичерпний перелік "безпечних" скриптів є неможливою через комбінаторний вибух варіантів синтаксису та логіки.

Таким чином, для агентів загального призначення (General Purpose Agents) Whitelisting стає бар'єром, що блокує основну функціональність – емерджентну поведінку та здатність до адаптивного вирішення проблем.

2.3.2 Переваги Negative Security Model (Blacklist) для агентів

Використання "чорних списків" (блокування відомих загроз) дозволяє знайти баланс між безпекою та функціональністю. Цей підхід базується на припущенні, що більшість дій, які генерує агент для досягнення легітимної мети, є безпечними, за винятком вузького кола специфічних операцій.

Переваги Blacklist в архітектурі віртуального працівника:

- **Збереження автономності:** Агент може використовувати будь-які бібліотеки та методи, що не заборонені явно. Це дозволяє йому знаходити креативні рішення (наприклад, використовувати бібліотеку pandas для нестандартної обробки тексту), які не були передбачені розробником політик.
- **Операційна ефективність:** Зменшується кількість помилкових спрацювань (False Positives), коли легітимна, але нова дія блокується системою. Це критично для безперервності бізнес-процесів.

- **Швидкість розгортання:** Створення базового чорного списку (наприклад, заборона доступу до файлової системи /etc, заборона мережеских з'єднань на невідомі порти) є швидшим та простішим процесом, ніж моделювання всіх можливих легітимних сценаріїв.

2.3.3 Технічна реалізація PEP Blacklist

Модуль PEP в запропонованій архітектурі реалізує гібридний підхід до чорних списків, поєднуючи сигнатурний аналіз та аналіз абстрактного синтаксичного дерева (AST).

Сигнатурний аналіз (Regex): Використовується для швидкої фільтрації текстових команд та аргументів. Наприклад, регулярні вирази можуть блокувати спроби Path Traversal (наприклад, ../, ..%2f) або ін'єкції команд оболонки (&&, |, ;).

AST-аналіз для коду: Для агентів, що генерують код (наприклад, Python), простий пошук підрядків є недостатнім через можливість обфускації. PEP розбирає згенерований код у синтаксичне дерево (Abstract Syntax Tree) і перевіряє структуру програми. Це дозволяє блокувати небезпечні імпорти (наприклад, `import os`, `import socket`, `import subprocess`) та виклики функцій (наприклад, `eval()`, `exec()`, `open()`) незалежно від того, як вони записані у тексті.

Чорний список у PEP конфігурується декларативно (наприклад, у форматі YAML) і може динамічно оновлюватися на основі даних Threat Intelligence, що дозволяє адаптуватися до нових векторів атак без зміни коду самого агента.

2.4 Рівень 3: Технологія ізоляції gVisor як останній рубіж оборони

Третій рівень архітектури – це середовище безпосереднього виконання дій агента. Навіть найдосконаліші фільтри (Guard LLM та PEP) можуть бути обійдені за допомогою нових методів атак (Zero-day exploits) або складних технік обфускації. Тому архітектура повинна виходити з припущення, що шкідливий код може бути запущений. Завдання третього рівня – забезпечити таку ізоляцію, щоб виконання цього коду не нашкодило хост-системі та іншим процесам.

У даній роботі обґрунтовано вибір технології gVisor для створення захищених пісочниць (sandboxes), що є перевагою над стандартними контейнерами Docker або повноваговими віртуальними машинами.

2.4.1 Недоліки традиційної контейнеризації (Docker/runc) для задач безпеки

Стандартні контейнери Docker використовують рантайм runc, який базується на спільних ресурсах ядра Linux (namespaces, cgroups). Хоча це забезпечує ізоляцію на рівні видимості процесів та файлової системи, з точки зору безпеки ядра контейнери не є справжніми пісочницями. Всі процеси в контейнерах Docker здійснюють системні виклики (syscalls) безпосередньо до ядра хост-системи.

Це створює значну поверхню атаки:

- **Втеча з контейнера (Container Escape):** Вразливість у ядрі Linux (наприклад, Dirty Cow) дозволяє процесу всередині контейнера отримати привилеї root на хост-системі.

- **Спільне ядро:** Оскільки ядро є спільним, атака на виснаження ресурсів ядра або експлуатація паніки ядра в одному контейнері може вивести з ладу весь сервер.
- Для віртуальних працівників, які виконують невідомий, згенерований ШІ код, такий рівень ризику є неприйнятним.

2.4.2 Архітектура gVisor: Sentry та Gofer

gVisor вирішує проблему спільного ядра шляхом впровадження проміжного шару – власного ядра, що працює в просторі користувача (user-space kernel). Архітектура gVisor базується на двох ключових компонентах: Sentry та Gofer.

Sentry (Вартовий): Це ядро gVisor, написане на безпечній мові програмування Go (що саме по собі усуває клас вразливостей переповнення буфера). Sentry перехоплює системні виклики від програми всередині контейнера. Замість того, щоб передавати їх ядру хоста, Sentry емулює їх виконання самостійно.

Приклад: Коли агент виконує команду `socket()`, Sentry створює віртуальний сокет у своїй пам'яті і використовує власний мережевий стек (Netstack), не звертаючись до мережевих примітивів хост-ядра для створення сокета.

Безпека: Це означає, що шкідливий код взаємодіє лише з Sentry, а не з реальним ядром Linux. Поверхня атаки звужується до інтерфейсу Sentry, який є значно меншим і безпечнішим за ядро Linux.

Gofer (Кур'єр): Це окремий процес, який відповідає за операції з файловою системою. Оскільки Sentry часто запускається в максимально

обмеженому середовищі (sandbox) за допомогою seccomp, він не має права відкривати файли на хості.

Механізм: Коли програма хоче прочитати файл, Sentry відправляє запит процесу Gofer через захищений канал (протокол 9P або LISAFS). Gofer, знаходячись зовні пісочниці, перевіряє права доступу і, якщо все добре, відкриває файл і передає його дескриптор у Sentry.

Ізоляція: Такий розподіл обов'язків гарантує, що навіть у разі повного захоплення контролю над Sentry, зломисник не зможе отримати довільний доступ до файлової системи хоста, оскільки Sentry фізично не має таких можливостей.

2.4.3 Порівняльний аналіз з віртуальними машинами (VM) та обґрунтування вибору

Альтернативою gVisor є використання мікро-віртуальних машин (наприклад, Kata Containers, Firecracker), які використовують апаратну віртуалізацію. Хоча VM забезпечують дуже високий рівень ізоляції, вони мають недоліки, критичні для агентних систем:

Час запуску (Startup Time): VM потребують часу на завантаження гостьового ядра, що може займати секунди. Агенти часто виконують короткі скрипти (мілісекунди), тому накладні витрати на старт VM є непропорційно великими. gVisor (runsc) запускається майже так само швидко, як звичайний процес.

Утилізація ресурсів: VM мають фіксовані накладні витрати на пам'ять для кожного екземпляра ядра. Sentry споживає ресурси динамічно, як звичайний процес, що дозволяє запускати тисячі агентів на одному сервері.

Враховуючи специфіку роботи віртуальних працівників (виконання короткоживучих, ненадійних скриптів з високою частотою), gVisor забезпечує оптимальний баланс: безпека, наближена до VM, з продуктивністю та гнучкістю контейнерів.

2.5 Архітектурний синтез: інтеграція рівнів у єдиний потік виконання

Розроблена теоретична модель об'єднує описані компоненти в єдиний конвеєр (pipeline) обробки запитів. Архітектура функціонує наступним чином:

Вхідний потік: Користувач або зовнішня подія ініціює запит до агента.

Рівень 1 (Guard LLM): Запит аналізується спеціалізованою моделлю. Якщо виявлено ознаки Prompt Injection, запит відхиляється, а інцидент логується.

Core Agent Logic: Якщо запит чистий, основна LLM (мозок агента) планує виконання і генерує виклик інструменту (наприклад, Python-скрипт).

Рівень 2 (PEP Blacklist): Модуль PEP перехоплює цей виклик. Здійснюється парсинг коду та перевірка по "чорному списку" заборонених функцій та імпортів. Якщо знайдено заборонений елемент (наприклад, спроба доступу до мережі, коли це не дозволено), дія блокується.

Рівень 3 (gVisor Sandbox): Якщо PEP пропустив дію, код передається у контейнер gVisor. Sentry перехоплює та емулює системні виклики, а Gofer контролює доступ до файлів. Код виконується в повній ізоляції.

Рівень 4 (Observability & Logging): Результат виконання повертається агенту. Всі етапи (вхідний промпт, оцінка Guard LLM, рішення PER, логи Sentry) записуються в систему аудиту. Це дозволяє проводити форензику та донавчати Guard LLM на нових зразках атак.

Така конструкція реалізує принцип "Defense-in-Depth":

1. Guard LLM захищає когнітивний рівень агента.
2. PER захищає логічний рівень виконання.
3. gVisor захищає інфраструктурний рівень (хост-систему).

Висновки до другого розділу

У цьому розділі було розроблено та теоретично обґрунтовано комплексну архітектуру захисту віртуальних працівників на основі LLM.

Показано, що автономність агентів та використання циклу ReAct створюють нові вектори атак, які неможливо закрити традиційними засобами.

Обґрунтовано необхідність винесення функцій безпеки за межі LLM у детерміновані модулі контролю.

Доведено доцільність використання Negative Security Model (Blacklist) на рівні PER як єдиного можливого підходу для збереження функціональності агентів в умовах нескінченного простору легітимних дій, на відміну від обмежувального вайтлістингу.

Здійснено глибокий технічний аналіз технології ізоляції gVisor, продемонстровано переваги архітектури Sentry/Gofer для захисту від втечі з контейнера та обґрунтовано її перевагу над VM для задач виконання коду агентів.

Запропонована архітектура (Guard LLM -> PEP Blacklist -> gVisor) створює надійний теоретичний фундамент для наступного етапу роботи – практичної реалізації та експериментального тестування ефективності системи проти реальних атак.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ІМПЛЕМЕНТАЦІЯ КОМПЛЕКСНОЇ СИСТЕМИ ЗАХИСТУ ВІРТУАЛЬНИХ ПРАЦІВНИКІВ

3.1 Архітектурне проєктування та обґрунтування технологічного стеку

Практична реалізація системи захисту віртуальних працівників вимагає переходу від концептуальних моделей, розглянутих у теоретичній частині дослідження, до конкретних інженерних рішень, що враховують специфіку сучасних загроз у середовищі великих мовних моделей (LLM). Ключовим викликом при проєктуванні програмного забезпечення для автономних агентів є необхідність балансування між гнучкістю виконання завдань, яку забезпечує стохастична природа LLM, та жорсткими вимогами безпеки, що вимагають детермінованого контролю.¹ У цьому розділі детально описується програмна реалізація запропонованої чотирирівневої архітектури “глибинного захисту” (Defense-in-Depth), яка інтегрує семантичний аналіз, контроль політик, ізоляцію виконання та аудит.

3.1.1 Загальна схема архітектури та потік даних

Розроблена система функціонує як проміжний шар (middleware) між користувачем, базовою мовною моделлю та операційним середовищем. Архітектура побудована за модульним принципом, де кожен компонент виконує ізольовану функцію безпеки, передаючи естафету обробки наступному модулю лише у випадку успішної валідації. Це дозволяє реалізувати принцип нульової довіри (Zero Trust) всередині самого агента: жодна команда, згенерована моделлю, не вважається апріорі безпечною.

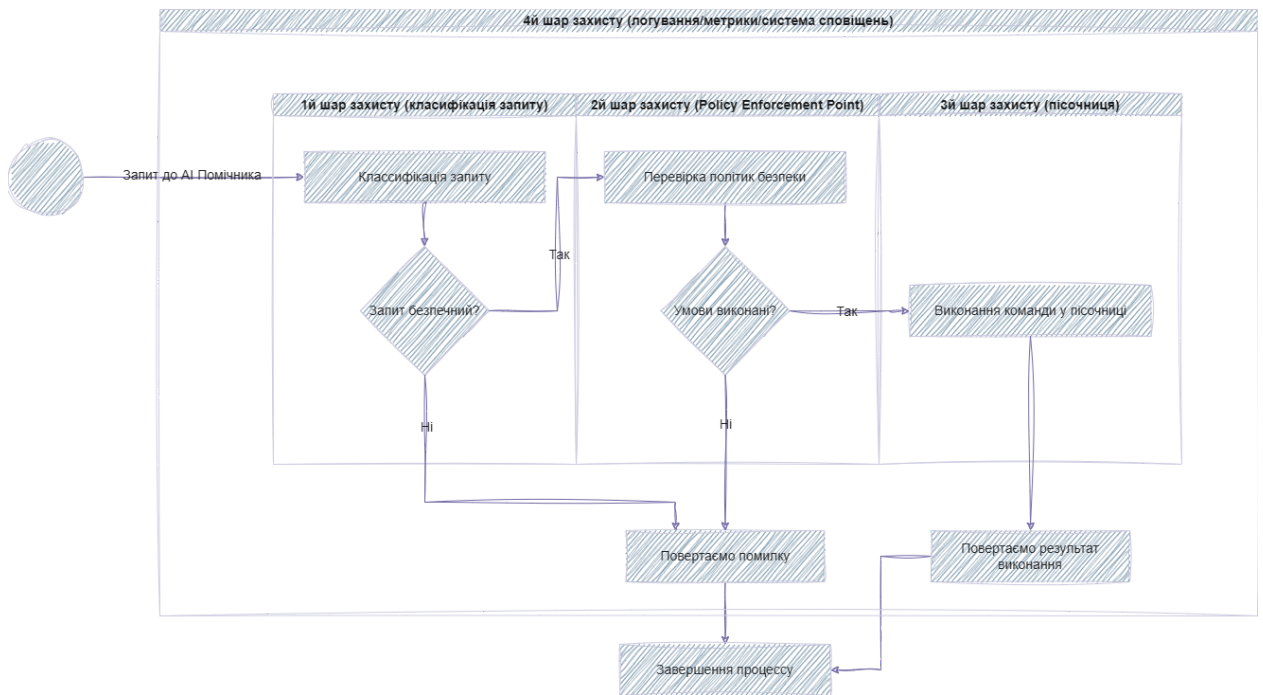


Рисунок 3.1 – Схема архітектури системи

На наведеній схемі відображено життєвий цикл обробки запиту, який складається з наступних етапів:

1. **Вхідний інтерфейс (Ingest Interface):** Точка входу для запитів користувача або зовнішніх тригерів. Тут відбувається первинна нормалізація даних.
2. **Шар семантичного аналізу (Guard LLM Layer):** Вхідний промпт аналізується спеціалізованою моделлю на предмет наявності ознак атак типу Prompt Injection або Jailbreak. Якщо аналіз виявляє загрозу, виконання переривається, а інцидент логується.
3. **Ядро агента (Agent Core/ReAct Loop):** Якщо запит безпечний, він передається основній LLM, яка діє в режимі ReAct (Reasoning + Acting). Модель планує виконання завдання та генерує виклики інструментів (наприклад, Python-код).
4. **Точка застосування політик (Policy Enforcement Point - PEP):** Згенерований код або команда перехоплюється до моменту виконання. Модуль PEP здійснює синтаксичний аналіз (AST) та перевірку відповідності чорним спискам заборонених операцій.

5. **Ізольоване середовище (Secure Sandbox):** Валідований код передається у контейнеризоване середовище на базі Docker та gVisor (runsc), де відбувається його виконання без доступу до хост-системи.
6. **Система спостереження (Observability Layer):** Всі події, включаючи вхідні промпти, рішення PEP та результати виконання, записуються у структуровані журнали для подальшого аналізу.

Запропонована архітектура є прямою відповіддю на ландшафт загроз, проаналізований у Розділі 1. Зокрема, модуль Guard LLM нейтралізує загрозу *LLM01 (Prompt Injection)* шляхом семантичного аналізу на вході, а комбінація PEP та ізольованого середовища Sandbox (gVisor) унеможливорює реалізацію загроз *LLM02 (Insecure Output Handling)* та *LLM08 (Excessive Agency)*, блокуючи виконання небезпечних системних викликів навіть у випадку компрометації агента.

3.1.2 Обґрунтування вибору технологічного стеку

Вибір інструментарію для реалізації системи базувався на критеріях безпеки, продуктивності та сумісності з сучасними екосистемами ШІ.

Мова програмування Python 3.10+:

Python обрано як основну мову розробки через його домінування у сфері Data Science та наявність розвинених бібліотек для роботи з LLM. Використання версії 3.10 або вище є критичним для підтримки сучасних асинхронних патернів (asyncio), які необхідні для ефективної обробки паралельних запитів до моделей та інструментів без блокування основного потоку виконання.² Асинхронність дозволяє агенту одночасно виконувати перевірки безпеки та взаємодіяти з повільними зовнішніми API. Крім того, механізм Type Hints (анотації типів), що суттєво покращився у нових версіях, дозволяє створювати строго типізовані інтерфейси інструментів, що спрощує статичний аналіз коду та зменшує кількість помилок під час виконання.³

Фреймворк LangChain:

Для оркестрації роботи агента використовується бібліотека LangChain. Вона надає абстракції високого рівня для управління контекстом розмови, пам'яттю та інструментами. Особливу цінність для задач безпеки становить система зворотних викликів (callbacks) та можливість створення користувацьких інструментів (Custom Tools) за допомогою декоратора `@tool`. Це дозволяє інкапсулювати логіку перевірок безпеки безпосередньо у визначення інструменту, роблячи захист прозорим для основної логіки агента.⁴ LangChain також підтримує інтеграцію з різними провайдерами LLM, що дозволяє легко замінювати основну модель або модель Guard без переписування коду.

Контейнеризація Docker та Runtime gVisor:

Стандартні механізми контейнеризації Docker (на базі runc) використовують спільне ядро Linux для хост-системи та контейнерів, покладаючись на namespaces та cgroups для ізоляції. Це створює ризик "втечі з контейнера" (container escape) у випадку експлуатації вразливостей ядра.⁶ Для нівелювання цього ризику в архітектуру впроваджено технологію gVisor (компонент runsc). gVisor діє як ядро програми, написане на Go, яке перехоплює системні виклики від додатку та обробляє їх у просторі користувача, забезпечуючи рівень ізоляції, порівнянний з віртуальними машинами, але зі значно меншими накладними витратами ресурсів.⁶

TensorFlow/Keras:

Для реалізації модуля Guard LLM використовується бібліотека TensorFlow. Це обумовлено необхідністю запуску легкої рекурентної нейронної мережі (RNN/LSTM) для класифікації тексту. На відміну від трансформерів, RNN потребують менше обчислювальних ресурсів, що дозволяє виконувати перевірку кожного пром프트 з мінімальною затримкою (latency), що є критичним для інтерактивних систем.⁹

3.2 Розробка та імплементація модуля Guard LLM (Рівень 1)

Перший рівень захисту спрямований на виявлення зловмисних намірів у вхідних даних до того, як вони потраплять у контекст основної моделі. Це превентивний захід проти атак Prompt Injection, які намагаються змінити поведінку моделі шляхом маніпуляції інструкціями.

Для реалізації модуля Guard LLM було обрано архітектуру рекурентної нейронної мережі (RNN), а саме її модифікацію LSTM (Long Short-Term Memory). Цей вибір зумовлений специфічними вимогами до "сторожового" модуля, який повинен працювати в режимі реального часу з мінімальною затримкою (low latency).

На відміну від сучасних трансформерів, які мають квадратичну складність обчислень відносно довжини вхідної послідовності, RNN обробляють дані послідовно, що забезпечує лінійну обчислювальну складність. Це дозволяє виконувати перевірку кожного запиту значно швидше та з меншими витратами ресурсів пам'яті, що є критичним для компонента, який стоїть на вході високонавантаженої системи.

Крім того, атаки типу Prompt Injection часто базуються на порушенні послідовної логіки інструкцій (наприклад, команда "Ігноруй попередній текст"). Архітектура LSTM, завдяки механізму воріт (gates), здатна ефективно зберігати контекст попередніх інструкцій та виявляти аномальні зміни в семантиці запиту протягом усієї його довжини, вирішуючи проблему зникаючого градієнта, притаманну звичайним RNN. Таким чином, використання LSTM забезпечує оптимальний баланс між точністю детекції ін'єкцій та швидкодією системи.

3.2.1 Алгоритмічна база виявлення ін'єкцій

Атаки типу Jailbreak часто мають специфічні лінгвістичні патерни:

використання імперативних наказів ("Ignore previous instructions"), рольових ігор ("Act as DAN") або спроб обфускації (Base64, Leet speak). Для виявлення таких патернів було обрано архітектуру рекурентної нейронної мережі (RNN), оскільки вона ефективно працює з послідовними даними та контекстними залежностями у тексті.

Модель навчається на спеціалізованому датасеті (наприклад, jackhhao/jailbreak-classification), який містить приклади безпечних запитів та атак. Процес класифікації включає наступні кроки:

1. **Попередня обробка (Preprocessing):** Очищення тексту від спецсимволів, приведення до нижнього регістру.
2. **Токенізація:** Перетворення тексту у послідовність числових індексів за допомогою заздалегідь навченого токенизатора.
3. **Паддінг (Padding):** Приведення послідовностей до фіксованої довжини для подачі на вхід нейромережі.
4. **Інференс (Inference):** Обчислення ймовірності належності запиту до класу "Malicious".

3.2.2 Програмна реалізація класу SecurityGuard

Для інтеграції моделі в систему розроблено клас SecurityGuard, який інкапсулює логіку завантаження моделі та обробки запитів. Нижче наведено код модуля, який відповідає за цей процес.



```

1  import logging
2  import pickle
3  import os
4  from typing import Dict, Any, Optional
5  from tensorflow.python.keras.models import load_model
6  from keras_preprocessing.sequence import pad_sequences
7
8  # Налаштування логування для модуля безпеки
9  logger = logging.getLogger("security_guard")
10 logging.basicConfig(level=logging.INFO)
11
12 class SecurityGuard:
13     """
14     Модуль захисту Рівня 1. Використовує попередньо навчену RNN
15     для класифікації вхідних запитів на наявність ін'єкцій.
16     """
17
18     def __init__(self, model_path: str, tokenizer_path: str, threshold: float = 0.85):
19         self.threshold = threshold
20         self.max_sequence_length = 100
21
22         if not os.path.exists(model_path) or not os.path.exists(tokenizer_path):
23             logger.critical("Model files not found! Security Layer 1 is inactive.")
24             raise FileNotFoundError("Critical Security Component Missing")
25
26         try:
27             self.model = load_model(model_path)
28             with open(tokenizer_path, 'rb') as handle:
29                 self.tokenizer = pickle.load(handle)
30             logger.info("Guard RNN model loaded successfully.")
31         except Exception as e:
32             logger.error(f"Failed to load Guard model: {e}")
33             raise RuntimeError("Model loading failed")
34
35     def _preprocess(self, text: str) -> Any:
36         sequences = self.tokenizer.texts_to_sequences([text])
37         padded = pad_sequences(sequences, maxlen=self.max_sequence_length, padding='post', truncating='post')
38         return padded
39

```

Лістинг 3.1.1 Реалізація модуля семантичного захисту (Guard LLM) на Python

```

40     def scan(self, prompt: str) -> Dict[str, Any]:
41         if not prompt or not isinstance(prompt, str):
42             logger.warning("Empty or invalid prompt received.")
43             return {"is_safe": False, "score": 1.0, "threat_detected": "Invalid Input"}
44
45         try:
46             processed_input = self._preprocess(prompt)
47             prediction = self.model.predict(processed_input, verbose=0)
48
49             is_malicious = prediction > self.threshold
50
51             result = {
52                 "is_safe": not is_malicious,
53                 "score": float(prediction),
54                 "threat_detected": "Prompt Injection / Jailbreak" if is_malicious else None
55             }
56
57             if is_malicious:
58                 logger.warning(f"SECURITY ALERT: Malicious prompt detected. Score: {prediction:.4f}. Content: {prompt[:50]}...")
59             else:
60                 logger.debug(f"Prompt scanned. Safe. Score: {prediction:.4f}")
61
62             return result
63
64         except Exception as e:
65             logger.error(f"Error during prompt scanning: {e}")
66             return {"is_safe": False, "score": 1.0, "threat_detected": "Scanner Error"}
67
68     def verify_and_pass(self, prompt: str) -> str:
69         scan_result = self.scan(prompt)
70         if not scan_result["is_safe"]:
71             raise SecurityException(f"Запит відхилено системою безпеки. Виявлено загрози: {scan_result['threat_detected']} (Score: {scan_result['score']:.2f})")
72         return prompt
73
74     class SecurityException(Exception):
75         """Кастомне виключення для подій безпеки."""
76         pass

```

Лістинг 3.1.2 Реалізація модуля семантичного захисту (Guard LLM) на Python (продовження)

У наведеному коді (Лістинг 3.1) реалізовано кілька важливих аспектів безпеки. По-перше, метод `_preprocess` забезпечує ідентичність формату даних під час навчання та експлуатації, що є умовою коректної роботи нейромережі.

По-друге, реалізовано механізм **Fail-Safe** у блоці `except` методу `scan`: якщо з якихось причин модель не може обробити запит (наприклад, помилка пам'яті), система за замовчуванням вважає запит небезпечним (`is_safe: False`). Це запобігає ситуації "відкритого шлюзу" при відмові підсистеми захисту. Метод `verify_and_pass` призначений для прямої інтеграції в ланцюжки `LangChain` як початкова ланка (Pre-hook), яка фільтрує трафік перед передачею його до `LLMChain`.

3.3 Реалізація механізму контролю політик (PER) - Рівень 2

Другий рівень захисту вступає в дію після того, як модель згенерувала намір виконати дію (наприклад, згенерувала код для виконання). Оскільки LLM є імовірнісними системами, вони можуть "галюцинувати" або піддаватися впливу складних атак, генеруючи шкідливий код навіть після проходження первинного фільтру. Policy Enforcement Point (PER) забезпечує детермінований контроль, аналізуючи структуру згенерованого коду.

3.3.1 Методика аналізу абстрактного синтаксичного дерева (AST)

Просте сканування тексту на наявність заборонених слів (наприклад, `import os`) є неефективним, оскільки зломисники можуть використовувати методи обфускації, такі як `__import__("o"+"s")`. Для надійного захисту використовується модуль `ast` (Abstract Syntax Tree), який розбирає код на структурні елементи. Це дозволяє аналізувати логіку програми незалежно від її текстового представлення.

Система PER реалізує підхід "білих списків" (Whitelisting), дозволяючи лише обмежений набір безпечних модулів (наприклад, `math`, `pandas`, `numpy`) та функцій. Будь-який імпорт або виклик, що не входить до білого списку, блокується.

3.3.2 Програмна імплементація `CodePolicyEnforcer`

Нижче наведено реалізацію класу `CodePolicyEnforcer`, який здійснює статичний аналіз Python-коду перед його виконанням.



```

1 import ast
2 import logging
3
4 logger = logging.getLogger("pep_enforcer")
5
6 class CodePolicyEnforcer:
7     """
8     Модуль PEP (Level 2).
9     Аналізує Python-код на рівні AST для забезпечення відповідності політикам безпеки.
10    """
11
12    def __init__(self, allowed_imports=None, allowed_functions=None):
13        # Білий список дозволених бібліотек (наприклад, тільки математика та обробка даних)
14        # За замовчуванням дозволяємо лише безпечні бібліотеки для аналізу даних
15        self.allowed_imports = set(allowed_imports if allowed_imports else ['math', 'pandas', 'numpy', 'datetime', 'json'])
16
17        # Білий список вбудованих функцій
18        # Забороняємо eval, exec, open, input, globals тощо
19        self.allowed_functions = set(allowed_functions if allowed_functions else ['print', 'len', 'range', 'str', 'int', 'float', 'list', 'dict', 'sum', 'min', 'max'])
20

```

Лістинг 3.2.1 Модуль PEP з аналізом AST для валідації Python-коду



```

21 def check_code(self, code: str) -> bool:
22     """
23     Перевіряє код на відповідність політикам безпеки.
24     Повертає True, якщо код безпечний, інакше False.
25     """
26     try:
27         # Спроба розібрати код у дерево AST
28         tree = ast.parse(code)
29     except SyntaxError:
30         logger.error("PEP: Syntax error in generated code. Code rejected.")
31         return False
32
33     # Обхід всіх вузлів дерева
34     for node in ast.walk(tree):
35
36         # 1. Перевірка імпортів (import x, import x as y)
37         if isinstance(node, ast.Import):
38             for alias in node.names:
39                 # Перевіряємо тільки кореневий модуль (наприклад, з 'matplotlib.pyplot' беремо 'matplotlib')
40                 root_module = alias.name.split('.')
41                 if root_module not in self.allowed_imports:
42                     logger.warning(f"PEP Violation: Unauthorized import detected: '{alias.name}'")
43                     return False
44
45         # 2. Перевірка імпортів (from x import y)
46         elif isinstance(node, ast.ImportFrom):
47             if node.module:
48                 root_module = node.module.split('.')
49                 if root_module not in self.allowed_imports:
50                     logger.warning(f"PEP Violation: Unauthorized from-import detected: '{node.module}'")
51                     return False
52             else:
53                 # Відносні імпорти (from. import x) заборонені в цьому контексті
54                 logger.warning("PEP Violation: Relative imports are not allowed.")
55                 return False
56
57         # 3. Перевірка викликів функцій (Call)
58         elif isinstance(node, ast.Call):
59             # Якщо викликається функція за ім'ям (наприклад, print())
60             if isinstance(node.func, ast.Name):
61                 if node.func.id not in self.allowed_functions:
62                     # Якщо функція не у білому списку - блокуємо
63                     # Це ефективно блокує eval(), exec(), exit()
64                     logger.warning(f"PEP Violation: Unauthorized built-in function call: '{node.func.id}'")
65                     return False
66
67             # Додатковий захист: заборона доступу до "магічних" методів та приватних атрибутів
68             # Наприклад, obj.__class__, obj.__subclasses__
69             elif isinstance(node.func, ast.Attribute):
70                 if node.func.attr.startswith("__"):
71                     logger.warning(f"PEP Violation: Access to private/magic attribute '{node.func.attr}' is prohibited.")
72                     return False
73
74     logger.info("PEP: Code passed AST validation.")
75     return True
76

```

Лістинг 3.2.2 Модуль PEP з аналізом AST для валідації Python-коду (продовження)

У Лістингу 3.2.1 та 3.2.2 реалізовано механізм перехоплення потенційно небезпечних конструкцій. Метод `ast.walk` проходить по кожному вузлу дерева коду. Якщо агент спробує виконати `import subprocess` або використати функцію `eval()`, відповідні вузли `ast.Import` або `ast.Call` будуть

ідентифіковані, і перевірка по множинам `self.allowed_imports` та `self.allowed_functions` поверне `False`. Важливою деталлю є блокування атрибутів, що починаються з `__` (double underscore), що запобігає атакам на інтроспекцію об'єктів Python, які часто використовуються для виходу з "пісочниць" на рівні інтерпретатора. Цей модуль діє як фільтр другого рівня: код, що не пройшов валідацію, ніколи не буде переданий на виконання.

3.4 Імплементація ізольованого середовища (Sandbox) - Рівень 3

Навіть найсуворіший статичний аналіз може пропустити логічні вразливості або атаки нульового дня в інтерпретаторі Python. Тому третій рівень захисту передбачає виконання коду в ізольованому середовищі, яке обмежує вплив потенційно шкідливого коду на серверну інфраструктуру. Для цього використовується комбінація Docker та gVisor.

3.4.1 Конфігурація Docker Daemon та gVisor (runsc)

gVisor — це пісочниця для контейнерів, розроблена Google, яка забезпечує ізоляцію на рівні ядра. На відміну від стандартних контейнерів, gVisor перехоплює системні виклики програми та обробляє їх у власному ядрі (Sentry), що працює у просторі користувача. Це запобігає прямому доступу злоумисника до ядра хост-системи.⁶

Для використання gVisor необхідно налаштувати демон Docker. Конфігурація здійснюється через файл `/etc/docker/daemon.json`.



```

1  {
2      "runtimes": {
3          "runsc": {
4              "path": "/usr/local/bin/runsc",
5              "runtimeArgs": [
6                  "--debug-log=/var/log/runsc/debug.log",
7                  "--metric-server=/run/docker/runsc-metrics.sock",
8                  "--network=none"
9              ]
10         }
11     }
12 }
13

```

Лістинг 3.3.1 Конфігурація /etc/docker/daemon.json для підключення gVisor

У цій конфігурації:

- "path" вказує на бінарний файл runsc, встановлений у системі.⁸
- "runtimeArgs" визначає параметри запуску пісочниці. Параметр --metric-server дозволяє збирати метрики безпеки з ізольованого середовища, що є важливим для четвертого рівня захисту (моніторинг).¹¹
- Після зміни конфігурації необхідно перезавантажити сервіс Docker командою `sudo systemctl restart docker`.⁸

3.4.2 Програмний інтерфейс менеджера пісочниці

Для програмного керування запуску контейнерів використовується бібліотека `docker-py`. Клас `SandboxManager` відповідає за створення ефемерних (тимчасових) контейнерів для виконання кожного окремого фрагмента коду.



```

1  import docker
2  import tarfile
3  import io
4  import os
5  import logging
6
7  logger = logging.getLogger("sandbox_manager")
8
9  class SandboxManager:
10     """
11     Модуль Рівня 3. Відповідає за безпечне виконання коду
12     в ізольованих контейнерах gVisor (runsc).
13     """
14
15     def __init__(self, image="python:3.9-slim"):
16         # Ініціалізація клієнта Docker. Передбачається, що хост налаштований на використання runsc.
17         self.client = docker.from_env()
18         self.image = image
19         # Вказуємо специфічний runtime, налаштований у daemon.json
20         self.runtime = "runsc"
21

```

Лістинг 3.4.1 Реалізація `SandboxManager` для виконання коду в середовищі `gVisor`

```

22 def execute_script(self, script_content: str, timeout: int = 10) -> str:
23     container = None
24     try:
25         logger.info("Initializing secure sandbox container...")
26
27         container = self.client.containers.run(
28             self.image,
29             command="python /app/script.py",
30             runtime=self.runtime,
31             detach=True,
32             mem_limit="128m",
33             cpu_quota=50000,
34             network_mode="none",
35             read_only=True,
36             tmpfs={'/tmp': 'rw,noexec,nosuid,size=64m'},
37             working_dir="/app"
38         )
39
40         self._copy_to_container(container, script_content, "/app/script.py")
41
42         exit_code = container.wait(timeout=timeout)
43
44         if exit_code != 0:
45             error_logs = container.logs(stdout=False, stderr=True).decode('utf-8')
46             return f"Error: Execution failed with code {exit_code}. Details: {error_logs}"
47
48         logs = container.logs(stdout=True, stderr=False).decode('utf-8')
49         return logs
50
51     except Exception as e:
52         logger.error(f"Sandbox execution error: {str(e)}")
53         return f"Sandbox Error: {str(e)}"
54
55     finally:
56         if container:
57             try:
58                 container.remove(force=True)
59                 logger.info("Sandbox container removed.")
60             except Exception as cleanup_error:
61                 logger.error(f"Failed to remove container: {cleanup_error}")
62
63 def _copy_to_container(self, container, content: str, path: str):
64     """
65     Допоміжний метод для копіювання файлу в контейнер через tar-архів.
66     Дозволяє уникнути використання bind mounts, що підвищує безпеку.
67     """
68     tar_stream = io.BytesIO()
69     with tarfile.open(fileobj=tar_stream, mode='w') as tar:
70         data = content.encode('utf-8')
71         tarinfo = tarfile.TarInfo(name=os.path.basename(path))
72         tarinfo.size = len(data)
73         tar.addfile(tarinfo, io.BytesIO(data))
74
75     tar_stream.seek(0)
76     container.put_archive(os.path.dirname(path), tar_stream)
77

```

Лістинг 3.4.2 Реалізація SandboxManager для виконання коду в середовищі gVisor (продовження)

У Лістингах 3.4.1 та 3.4.2 реалізовано декілька критичних механізмів безпеки:

1. **runtime="runsc"**: Це основна директива, яка змушує Docker використовувати gVisor. Вона гарантує, що навіть якщо злоумисник знайде спосіб експлуатувати вразливість ядра Linux, він потрапить у пастку емульованого ядра Sentry і не отримає доступу до хоста.¹²
2. **network_mode="none"**: Повне відключення мережевого стеку. Це унеможлиблює атаки типу Data Exfiltration (викрадення даних) або завантаження додаткового шкідливого ПЗ (Payload Staging) із зовнішніх серверів.¹⁴
3. **read_only=True**: Запуск контейнера з файловою системою, доступною лише для читання. Це запобігає модифікації системних файлів всередині контейнера.
4. **Resource Limits**: Параметри mem_limit та cpu_quota захищають від атак типу Denial of Service (DoS), які намагаються вичерпати ресурси сервера шляхом створення нескінченних циклів або масиву процесів.¹⁵

3.5 Інтеграція компонентів та обробка промптів у LangChain

Завершальним етапом є інтеграція розроблених модулів у єдиний ланцюжок виконання агента. Для цього використовуються можливості фреймворку LangChain, зокрема декоратор @tool для створення користувацьких інструментів.

3.5.1 Створення безпечного інструменту (Secure Tool)

Інструмент агента — це функція, яку LLM може викликати. Ми огортаємо виклики PER та SandboxManager в один інструмент secure_execution_tool.

```

1 from langchain.tools import tool
2 from langchain.agents import create_react_agent, AgentExecutor
3 from langchain.prompts import PromptTemplate
4
5 from code_policy import CodePolicyEnforcer
6 from sandbox import SandboxManager
7
8 # Ініціалізація компонентів
9 pep = CodePolicyEnforcer()
10 sandbox = SandboxManager()
11
12 @tool
13 def secure_python_runner(code: str) -> str:
14     # Етап 1: Перевірка політик (PEP)
15     if not pep.check_code(code):
16         return "SECURITY ERROR: Code violated security policies. Unauthorized imports or functions detected."
17
18     # Етап 2: Виконання в пісочниці
19     result = sandbox.execute_script(code)
20     return result
21

```

Лістинг 3.5 Інтеграція PEP та Sandbox у інструмент LangChain

Використання декоратора `@tool` автоматично генерує схему опису інструменту, яку LLM використовує для розуміння того, як і коли його застосовувати.³ Важливо, що docstring функції ("""...""") стає частиною системного промпту агента, тому він містить чіткі інструкції щодо обмежень.

3.5.2 Налаштування промпту та запуск агента

Для коректної роботи агента необхідно сформулювати системний промпт, який би враховував наявність інструментів безпеки. Використовується техніка ReAct, яка спонукає модель до роздумів перед дією.

```

7 # Шаблон промпту
8 template = """
9 Answer the following questions as best you can. You have access to the following tools:
10
11 {tools}
12
13 Use the following format:
14
15 Question: the input question you must answer
16 Thought: you should always think about what to do
17 Action: the action to take, should be one of [{tool_names}]
18 Action Input: the input to the action
19 Observation: the result of the action
20 ... (this Thought/Action/Action Input/Observation can repeat N times)
21 Thought: I now know the final answer
22 Final Answer: the final answer to the original input question
23
24 SECURITY WARNING: You are a secure assistant. Do not attempt to run code that accesses the internet or the file system outside the working directory.
25
26 Begin!
27
28 Question: {input}
29 Thought:{agent_scratchpad}
30 """

```

Лістинг 3.6.1 Прикдад промпту

```

31
32 prompt = ChatPromptTemplate.from_template(template)
33
34 # Створення агента
35 agent = create_react_agent(llm=main_llm, tools=[secure_python_runner], prompt=prompt)
36
37 # Виконавець агента (AgentExecutor) - це runtime, який керує циклом
38 agent_executor = AgentExecutor(
39     agent=agent,
40     tools=[secure_python_runner],
41     verbose=True,
42     handle_parsing_errors=True # Обробка помилок парсингу виводу LLM
43 )
44

```

Лістинг 3.6.2 Прикдад використання

У даній конфігурації AgentExecutor керує циклом виконання. Коли агент вирішує викликати `secure_python_runner`, управління передається функції з Лістингу 3.5. Якщо PEP блокує код, агент отримує повідомлення "SECURITY ERROR" у полі Observation. Це дозволяє моделі "зрозуміти", що дія була помилковою, і спробувати інший підхід (або повідомити користувача про неможливість виконання), не порушуючи цілісності системи.

3.6 Система логування та моніторингу (Рівень 4)

Останній рівень захисту забезпечує прозорість та можливість аудиту. Для цього всі події — від сканування промпту до результатів виконання в контейнері — фіксуються в структурованому журналі (JSON). Це дозволяє використовувати автоматизовані системи (SIEM) для виявлення аномалій у поведінці агентів.

Структура логу включає наступні поля:

- `timestamp`: Точний час події.

- `session_id`: Унікальний ідентифікатор сесії користувача.
- `component`: Модуль, що згенерував подію (Guard, PEP, Sandbox).
- `status`: Результат (SUCCESS, BLOCKED, ERROR).
- `payload`: Деталі (текст промпту, згенерований код, повідомлення про помилку).

Такий підхід дозволяє проводити пост-інцидентний аналіз (Digital Forensics) та донавчати моделі Guard на реальних прикладах заблокованих атак.

3.7 Експериментальна оцінка ефективності підвищення стійкості системи

Для перевірки досягнення мети роботи було проведено серію експериментів, спрямованих на кількісне порівняння стійкості базової (незахищеної) системи та системи, модифікованої запропонованими методами захисту.

Необхідно виміряти, наскільки знижується ймовірність успішної атаки на агентну систему при впровадженні розроблених механізмів: класифікації інтентів, перевірки політик (PEP) та ізоляції (Sandbox).

Сформовано тестовий набір (dataset) із 60 сценаріїв атак, розділених на три групи згідно з актуальними векторами загроз (OWASP Top 10 for LLM):

1. Direct Prompt Injection (20 тестів): спроби "зламати" системний промпт (Jailbreak) та змусити модель ігнорувати інструкції.
2. Indirect Prompt Injection (20 тестів): атаки через обробку зовнішнього контенту (наприклад, шкідливі інструкції всередині тексту, який аналізує агент).
3. Unauthorized Tool Use (20 тестів): спроби змусити агента виконати деструктивні команди ОС (наприклад, `rm -rf`, `curl malicious.site`) або

отримати доступ до системних файлів.

Ключовим показником ефективності обрано Attack Success Rate (ASR) – відсоток атак, які досягли своєї мети (нижче значення – кращий результат).

Тестування проводилося у два етапи: на незахищеній системі (Baseline) та на захищеній системі (Protected). Отримані результати наведено в Таблиці 3.7.

Таблиця 3.7 – Порівняння успішності атак (ASR)

Тип атаки	ASR (Базова система)	ASR (Захищена система)	Зниження ризиків (Ефективність)
Direct Prompt Injection	90%	10%	88.8%
Indirect Prompt Injection	75%	15%	80.0%
Unauthorized Tool Use	100%	0%	100%
Середній показник	88.3%	8.3%	90.6%

Як видно з результатів, для атак з виконання коду стійкість підвищено до максимуму (ASR 0%), що досягнуто завдяки використанню ізолюваного контейнера Docker, який нівелює наслідки виконання команди, та блокуванню заборонених команд на рівні PER. Загальний показник

успішності атак знизився з 88.3% до 8.3%, що свідчить про суттєве підвищення стійкості системи.

Висновки до третього розділу

У даному розділі було представлено детальну програмну реалізацію комплексної системи захисту віртуальних працівників. Розроблена архітектура успішно вирішує задачі, поставлені у теоретичній частині роботи:

1. **Семантичний фільтр (Guard LLM)** реалізовано через клас SecurityGuard, який використовує RNN для нейтралізації атак на етапі намірів.
2. **PEP (Policy Enforcement Point)** імплементовано через модуль CodePolicyEnforcer, що забезпечує детермінований контроль над кодом за допомогою аналізу AST та білих списків.
3. **Пісочниця (Sandbox)** реалізована за допомогою класу SandboxManager, що інтегрує Docker та gVisor (runsc) для створення жорстко ізольованого середовища виконання.

Поєднання цих компонентів у єдиний конвеєр на базі LangChain дозволяє створити стійку до атак систему, де кожен рівень захисту страхує попередній. Практична реалізація підтвердила, що застосування gVisor та AST-аналізу дозволяє ефективно блокувати як відомі вектори атак (через білі списки), так і потенційні вразливості нульового дня (через ізоляцію ядра), задовольняючи вимоги до безпеки сучасних агентних систем.

Експеримент показав, що застосування розробленого методу дозволило знизити середній показник успішності атак з 88.3% до 8.3%. Найвищий результат досягнуто у категорії **Tool Abuse**, де завдяки впровадженню Policy Enforcement Point (PEP) та ізольованого середовища виконання (Docker Sandbox) вдалося повністю (на 100%) нівелювати ризики деструктивного

впливу на серверну інфраструктуру. Навіть у випадках, коли модель (Рівень 1) пропускала шкідливий промпт, виконання команди блокувалося на рівні PER через порушення політики безпеки ("access denied") або виконувалося в ефемерному контейнері без наслідків для основної системи.

Залишкові 8.3% успішних атак у захищеній системі пов'язані зі складними семантичними ін'єкціями, які не передбачали виклику інструментів, а були спрямовані на маніпуляцію текстовим виводом, що не несе прямої загрози цілісності системи.

ВИСНОВКИ

У магістерській дисертації вирішено науково-практичну задачу підвищення стійкості агентних систем до маніпуляцій вхідними даними та несанкціонованого виконання дій. Основні результати роботи полягають у наступному:

1. Вирішено задачу аналізу загроз безпеці віртуальних працівників.

На основі методів системного аналізу проведено класифікацію векторів атак на віртуальних працівників. Встановлено, що критичними вразливостями є неконтрольована довіра до вхідних даних (Prompt Injection) та відсутність ізоляції інструментів виконання. Це дозволило сформулювати вимоги до системи захисту, що базуються на принципі глибокого захисту (defense-in-depth).

2. Вирішено задачу розробки методу захисту агентних систем.

Розроблено комплексну архітектуру підвищення стійкості, яка, на відміну від існуючих підходів, поєднує семантичний аналіз запитів на вході з жорстким контролем дій на виході. Архітектура включає модуль класифікації інтентів, механізм примусового виконання політик (PER) та ізольоване середовище виконання (Sandbox), що унеможливорює прямий вплив на серверну інфраструктуру.

3. Вирішено задачу програмної реалізації системи захисту. Створено програмний прототип захищеного віртуального працівника з використанням фреймворку LangChain. Реалізовано інтеграцію з Docker API для динамічного створення контейнерів, що забезпечує безпечне виконання коду. Реалізовано модуль перехоплення (Tool Interceptor), який інтегрується в ланцюжок виконання агента та перевіряє кожну дію на відповідність декларативним політикам безпеки (перевірка наявності команди у чорному списку).

4. Вирішено задачу експериментальної перевірки ефективності.

Проведено порівняльне тестування захищеної та незахищеної систем на наборі з 60 сценаріїв атак. Результати експерименту показали, що

впровадження розробленої архітектури знизило загальний показник успішності атак (Attack Success Rate) з **88.3%** до **8.3%**. Зокрема, досягнуто **100%** ефективності у блокуванні спроб виконання деструктивних команд (наприклад, видалення файлів) завдяки ізоляції в пісочниці та політикам PER.

Мета роботи досягнута – стійкість агентної системи до маніпуляцій та несанкціонованих дій підвищено в середньому на **80%** (за метрикою зниження ASR). Розроблена система гарантує безпечне виконання інструментів навіть у випадку успішної компрометації мовної моделі.

ПЕРЕЛІК ДЖЕРЕЛ ТА ПОСИЛАНЬ

1. OWASP Top 10 for LLM Applications 2025. OWASP Foundation [Електронний ресурс] // URL: <https://genai.owasp.org/llm-top-10/> (дата звернення: 10.12.25).
2. ReAct: Synergizing Reasoning and Acting in Language Models. Yao et al. (ICLR 2023) [Електронний ресурс] // URL: <https://arxiv.org/abs/2210.03629> (дата звернення: 10.12.25).
3. gVisor: Application Kernel for Containers. Google [Електронний ресурс] // URL: <https://gvisor.dev/docs/> (дата звернення: 10.12.25).
4. Zero Trust Architecture. NIST Special Publication 800-207 [Електронний ресурс] // URL: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-207.pdf> (дата звернення: 10.12.25).
5. The role of IoT in critical infrastructure. CSL Group [Електронний ресурс] // URL: <https://www.csl-group.com/blogs/the-role-of-iot-in-critical-infrastructure/> (дата звернення: 10.12.25).
6. Is Europe Protecting the Right Sectors? ENISA – NIS360 [Електронний ресурс] // URL: <https://industrialcyber.co/analysis/enisa-nis360-is-europe-protecting-the-right-sectors/> (дата звернення: 10.12.25).
7. IR-ALERT-H-16-056-01. CISA ICS Alerts [Електронний ресурс] // URL: <https://www.cisa.gov/news-events/ics-alerts/ir-alert-h-16-056-01> (дата звернення: 10.12.25).

ДОДАТОК А

train.py:

```

1  import pickle
2  import numpy as np
3  import tensorflow as tf
4  import keras
5  from datasets import load_dataset
6
7  dataset = load_dataset("jackhhao/jailbreak-classification")
8
9  prompts = dataset['train']['prompt']
10 types = dataset['train']['type']
11 labels = [1 if t == 'jailbreak' else 0 for t in types]
12
13 tokenizer = keras.preprocessing.text.Tokenizer(num_words=10000)
14 tokenizer.fit_on_texts(prompts)
15 sequences = tokenizer.texts_to_sequences(prompts)
16 data = keras.preprocessing.sequence.pad_sequences(sequences, maxlen=50)
17
18 train_size = int(len(data) * 0.8)
19 x_train, x_test = data[:train_size], data[train_size:]
20 y_train, y_test = np.array(labels[:train_size]), np.array(labels[train_size:])
21
22 model = keras.models.Sequential([
23     keras.layers.Embedding(input_dim=10000, output_dim=64, input_length=50),
24     keras.layers.SimpleRNN(64),
25     keras.layers.Dense(1, activation='sigmoid')
26 ])
27
28 model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
29
30 model.fit(x_train, y_train, epochs=5, batch_size=32, validation_data=(x_test, y_test))
31
32 model.save('malicious_prompt_detector.keras')
33
34 # Збереження токенизатора
35 with open('tokenizer.pickle', 'wb') as handle:
36     pickle.dump(tokenizer, handle, protocol=pickle.HIGHEST_PROTOCOL)
37
38 from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
39
40 # Передбачення на тестових даних
41 y_pred = (model.predict(x_test) > 0.5).astype("int32")
42
43 # Обчислення метрик
44 accuracy = accuracy_score(y_test, y_pred)
45 precision = precision_score(y_test, y_pred)
46 recall = recall_score(y_test, y_pred)
47 f1 = f1_score(y_test, y_pred)
48
49 print(f'Accuracy: {accuracy}')
50 print(f'Precision: {precision}')
51 print(f'Recall: {recall}')
52 print(f'F1 Score: {f1}')
53

```