Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики

СТВОРЕННЯ ІНСТРУМЕНТУ НА ОСНОВІ ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ ДЛЯ ВІДЛАГОДЖЕННЯ IOS-ЗАСТОСУНКІВ У СЕРЕДОВИЩІ ВИКОНАННЯ

Текстова частина до магістерської роботи

за спеціальністю 122 Комп'ютерні науки

Керівник роботи
к. ф.-м. н.
_____Нагірна А. М._____
(прізвище та ініціали)

_____
(підпис)
«25» _____травня_____ 2026р.

Виконав студент
_____Літвінчук Д. В._____
(прізвище та ініціали)

_____
(підпис)
«25» _____травня_____ 2026р.

Київ 2026

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Факультет інформатики

Кафедра інформатики

Освітній ступінь Магістр

Спеціальність 122 Комп’ютерні науки

ЗАТВЕРДЖУЮ

Завідувач кафедри к.ф.-м.н. Нагірна А.М.

«___» _____ 2025 року

ЗАВДАННЯ

ДЛЯ МАГІСТЕРСЬКОЇ РОБОТИ СТУДЕНТУ

Літвінчуку Данилові Вікторовичу

1. Тема роботи «Створення удосконаленого відлагоджувача iOS-застосунків для автоматизованої діагностики та трансформації UI», керівниця роботи Нагірна Алла Миколаївна, кандидат фізико-математичних наук, доцент, затверджені наказом вищого навчального закладу від «___» _____ 2025 року № ___
2. Строк подання студентом роботи «___» _____ 2026 року
3. План роботи:
    1. Аналіз літературних джерел
        a. дослідження парадигми ReAct (Reasoning + Acting) в контексті програмної інженерії;
        b. аналіз проблеми Symbol Grounding великих мовних моделей (LLM);
        c. дослідження метрик ефективності LLM у відлагодженні, таких як DDI - Debugging Decay Index;

2. Проєктування та розробка інструменту (MCP Server)
    a. розробка інтерфейсу взаємодії із LLDB;
    b. розробка слухача подій (Event Listener) для обробки подій LLDB (breakpoints, crashes) без блокування основного циклу MCP
    c. розробка інструментів заземлення (Grounding) шляхом парсингу ієрархії UI;
3. Експериментальна частина та верифікація
    a. проєктування сценаріїв тестування: діагностики UI, виявлення збоїв програм та проблем з конкурентним доступом до пам’яті.
    b. впровадження стратегій оптимізації, таких як механізми Context Pruning та Fresh Start
    c. тестування та оцінювання роботи інструменту
4. Підготування рекомендацій
    a. визначення завдань, де доцільно застосовувати розроблений інструмент;
    b. визначення перспектив для покращення інструменту.

ГРАФІК ПІДГОТОВКИ МАГІСТЕРСЬКОЇ РОБОТИ ДО ЗАХИСТУ

№ з/п	ПЕРЕЛІК РОБІТ	Термін виконання	Дата ознайомлення наукового керівника	Підпис наукового керівника
1	Вибір теми, затвердження її на засіданні кафедри та закріплення наукового керівника. Узгодження календарного графіка підготовки кваліфікаційної роботи. Ознайомлення студента з критеріями оцінювання кваліфікаційної роботи.	жовтень	11 жовтня	
2	Вивчення джерел літератури, матеріалів архівів, періодичних видань, збір та узагальнення фактів і даних (аналіз наукових робіт на тему ReAct та Chain-of-Thought великих мовних моделей, вивчення специфікацій LLDB та MCP).	жовтень – листопад	31 листопада	
3	Складання плану кваліфікаційної роботи та узгодження з науковим керівником.	листопад	31 листопада	
4	Постановка задачі з розробки інструменту для відлагодження iOS	листопад – березень	31 березня	

	застосунків за допомогою великих мовних моделей.			
5	Проміжний контроль виконання роботи.	31 січня	30 січня	
6	Написання кваліфікаційної роботи в цілому, ознайомлення з її першим варіантом наукового керівника.	січень – березень	31 березня	
–	Розділ 1 (постановка проблеми, теоретичні основи, огляд літературних джерел: проблеми ручного відлагодження, символьного заземлення (Grounding), керування контекстом LLM. Розбір сучасних підходів та архітектури MCP як стандарту інтеграції).			
–	Розділ 2 (аналітико-дослідницька та практична частина): 1) Вибір архітектури та обґрунтування технологічного стеку (Python + LLDB SB API). 2) Алгоритми реалізації інструментів інспекції стану (Stack Trace, Variable Inspection). 3) Реалізація механізмів асинхронної обробки подій (Event-Driven			

	Architecture). 4) Реалізація механізму мінімізації DDI			
–	Розділ 3 (проєктно-рекомендаційна частина: розробка стратегій тестування, порівняння підходів до відлагодження із залученням AI-Агента із запропонованим, рекомендації щодо застосування підходів у різних сферах).			
7	Повне завершення написання кваліфікаційної роботи, оформлення її згідно з вимогами та подання на друге читання керівнику.	квітень – 15 травня	20 травня	
8	Подання кваліфікаційної роботи для перевірки письмових робіт студентів на відповідність вимогам академічної доброчесності.	до 25 травня	25 травня	

ЗМІСТ

ГРАФІК ПІДГОТОВКИ МАГІСТЕРСЬКОЇ РОБОТИ ДО ЗАХИСТУ . 4

АНОТАЦІЯ 8

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ 9

ВСТУП 12

1 ТЕОРЕТИЧНІ ОСНОВИ ВІДЛАГОДЖЕННЯ iOS-ЗАСТОСУНКІВ ЗА ДОПОМОГОЮ LLM-АГЕНТІВ 15

1.1. Відлагодження як ресурсомісткий процес розробки та передумови інтеграції LLM 16

1.2. Архітектура LLDB та API Scripting Bridge як основа програмного керування налагоджувачем 17

1.3. Особливості налагодження в iOS: архітектура debugserver, порти завдань Mach та механізми безпеки 19

1.4. Протокол Model Context Protocol: примітиви, транспортні рівні та архітектура взаємодії агентів і інструментів 21

1.5. Парадигма ReAct як модель поведінки автономного агента в динамічному середовищі 23

1.6. Проблема прив'язки символів у середовищі виконання налагоджувача 25

1.7. Управління станом агента та феномен індексу зниження ефективності налагодження (DDI) 27

Висновки до розділу 29

2 ОБҐРУНТУВАННЯ ВИБОРУ АРХІТЕКТУРИ ТА ФУНКЦІОНАЛУ MCP-СЕРВЕРА 29

2.1. Вибір моделі ізоляції середовища виконання між сервером MCP та LLDB 30

2.2. Декомпозиція простору дій агента на інструменти MCP 32

2.3. Розподіл відповідальності за рівні символьного заземлення між інструментами агента 37

2.4. Перетворення DDI на вимірюваний показник: вимірювання як передумова керування.....40

Висновки до розділу.....44

З ОБҐРУНТУВАННЯ ОБРАНИХ МЕТРИК ОЦІНЮВАННЯ, ЇХ ЗАСТОСУВАННЯ ТА АНАЛІЗ.....44

3.1. Метод та схема експериментального оцінювання.....45

3.2. Досліджуваний застосунок та набір контрольних дефектів.....46

3.3. Результати та порівняльний аналіз.....49

3.4. Практичне застосування інструментарію.....53

Висновки до розділу.....55

ВИСНОВКИ.....56

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ.....58

АНОТАЦІЯ

Робота присвячена побудові інтелектуального інструментарію відлагодження iOS-застосунків на основі LLM-агентів, інтегрованих із відлагоджувачем LLDB через протокол Model Context Protocol (MCP). Практичним результатом є MCP-сервер із двопроцесною архітектурою, що надає агенту програмний контроль над сесією LLDB у реальному часі: приєднання до процесу, керування точками зупину, інспекцію стану та ін'єкцію коду. Архітектурні рішення обґрунтовано через три теоретичні опори: цикл ReAct як модель поведінки агента, проблему символьного заземлення (лексичний, структурний, темпоральний рівні) та індекс деградації ефективності відлагодження (DDI).

Кінцевий програмний продукт може використовуватися у розробці програмного забезпечення, а також за використання таких агентів як Claude Code, Codex, Cursor та інших, що підтримують MCP.

Ключові слова: LLM-агент, відлагодження iOS, LLDB, Model Context Protocol, ReAct, символьне заземлення, індекс деградації відлагодження, ін'єкція коду, автономна діагностика.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AI	Artificial Intelligence, штучний інтелект
AMFI	Apple Mobile File Integrity, підсистема iOS, що забезпечує обов'язковий підпис і перевірку цілісності коду
API	Application Programming Interface, програмний інтерфейс застосунку
ASLR	Address Space Layout Randomization, рандомізація розташування адресного простору
CoT	Chain-of-Thought, метод послідовного розкладання задачі на кроки міркування
DDI	Debugging Decay Index, індекс згасання відлагодження – модель експоненційного спаду ефективності ітеративного відлагодження
DWARF	Debugging With Attributed Record Formats, формат налагоджувальної інформації, який споживає LLDB
GCD	Grand Central Dispatch, механізм асинхронного виконання задач у Apple-платформах
GDB	GNU Debugger, налагоджувач GNU; тут –у складі назви протоколу RSP
HTTP	HyperText Transfer Protocol, протокол передачі даних; транспорт Streamable HTTP у MCP
IPC	Inter-Process Communication, міжпроцесна взаємодія
JSON	JavaScript Object Notation, текстовий формат серіалізації даних
LLDB	Low-Level Debugger, налагоджувач у складі інфраструктури LLVM
LLM	Large Language Model, велика мовна модель

LLVM	від Low-Level Virtual Machine (нині власна назва), інфраструктура компіляторів, до складу якої входить LLDB
MCP	Model Context Protocol, протокол стандартизованого надання моделі інструментів, ресурсів і промптів
ObjC	Objective-C, мова програмування Objective-C та її рантайм
PC	Program Counter, лічильник команд (регістр); використовується у сигнатурі для виявлення циклів
PID	Process Identifier, ідентифікатор процесу (операція task_for_pid)
ReAct	Reasoning and Acting, парадигма поведінки агента, що чергує міркування й дію в замкнутому циклі
RPC	Remote Procedure Call, віддалений виклик процедур
RSP	Remote Serial Protocol, протокол віддаленого обміну між хостом і екземпляром LLDB
SB	Scripting Bridge, інтерфейс прив'язки LLDB до мов сценаріїв (Python)
SDK	Software Development Kit, набір засобів розробки
UI	User Interface, інтерфейс користувача
UTC	Coordinated Universal Time, всесвітній координований час
XNU	X is Not Unix, ядро операційних систем Apple
$E(t)$ / E_0	Effectiveness function / initial effectiveness, ефективність відлагодження на спробі t та її початкове значення
λ	Decay constant, стала згасання – швидкість втрати ефективності відлагодження

ВСТУП

Обґрунтування вибору теми дослідження

Виявлення та усунення дефектів забирають значну частку ресурсів розробника – за оцінками від 35% до 50% загального часу [1]. Великі мовні моделі ефективно працюють з вихідним кодом, проте їх здатність до відлагодження принципово обмежена відсутністю доступу до стану виконання: регістрів, адрес пам'яті, значень змінних у конкретний момент, ієрархії представлень. Через це моделі не можуть відрізнити дефект у логіці коду від дефекту, що проявляється лише в рантаймі, а запропоновані ними виправлення спираються на абстракцію вихідного коду, а не на фактичний стан скомпільованого процесу. Це обмеження особливо гостре на платформі iOS, де асинхронне виконання через GCD і Swift Concurrency, непрозорість системних фреймворків і модель безпеки (ASLR, обов'язковий підпис коду, sandbox) усувають можливість прямого спостереження за станом і залишають Scripting Bridge API відлагоджувача LLDB єдиним надійним каналом доступу.

Перехід від статичного агента до автономного вимагає інтеграції моделі у формальний цикл «міркування–дія–спостереження» (ReAct), у якому агент взаємодіє з рантаймом через інтерфейс відлагоджувача. Стандартизованим шаром для такої взаємодії слугує протокол Model Context Protocol (MCP), що відокремлює логіку агента від низькорівневої механіки LLDB. Однак сам факт побудови такого моста не гарантує практичної цінності інструменту: автономна сесія відлагодження стикається з двома фундаментальними перешкодами – проблемою символьного заземлення (зв'язку ідентифікаторів коду з їхньою реалізацією в пам'яті процесу) і деградацією ефективності в міру накопичення шуму в контекстному вікні, формалізованою як індекс DDI.

Актуальність роботи полягає в розробці інструменту, що б дозволив агентам LLM отримувати доступ до стану виконання iOS-застосунків для їх налагодження.

Мета й завдання дослідження

Метою роботи є розробити й експериментально оцінити інструментарій автономного відлагодження iOS-застосунків на базі LLM-агента, побудований через інтеграцію LLDB і MCP, та встановити, які архітектурні рішення визначають його практичну ефективність.

Об'єктом дослідження є процес автономного відлагодження iOS-застосунків LLM-агентом, що взаємодіє зі станом виконання процесу через міст LLDB–MCP.

Предметом дослідження є методи й архітектурні рішення побудови такого інструментарію, що забезпечують символьне заземлення агента в рантаймі та протидіють деградації ефективності сесії (DDI).

Для досягнення мети поставлено й виконано такі *завдання*:

1. Виявити джерело обмеженості LLM у задачах відлагодження – відсутність доступу до стану виконання – та формалізувати дві ключові перешкоди автономної сесії, проблему символьного заземлення на трьох рівнях і деградацію ефективності за моделлю DDI, як критерії оцінювання інструменту на базі LLM-агента.
2. Встановити граничні умови, у яких працює агент на платформі iOS, через аналіз об'єктної моделі LLDB, Scripting Bridge API та механізмів безпеки платформи (порти задач Mach, ASLR, обов'язковий підпис коду, sandbox).
3. Обґрунтувати ключові архітектурні рішення інструментарію – модель ізоляції несумісних рантаймів, транспортний рівень MCP, гранулярність декомпозиції простору дій агента – і встановити, які рівні символьного заземлення закриваються типізованими

інструментами, а які лишаються прив'язаними до універсального механізму виконання команд.

4. Спроєктувати вимірювальний шар, що фіксує метрики кожного виклику інструмента як мінімальне достатнє покриття для відновлення сталої згасання λ за результатами сесій.
5. Реалізувати інструментарій у вигляді функціонального MCP-сервера, що покриває повний цикл відлагодження iOS-застосунку – від приєднання до процесу до ін'єкції коду.
6. Спланувати та провести порівняльний експеримент між агентом із доступом до стану виконання й агентом, обмеженим вихідним кодом, побудувати емпіричну криву $\mathrm{E(t)}$, оцінити λ й кількісно встановити залежність переваги агента від рівня символьного заземлення, якого потребує дефект, перевіривши тим самим центральну тезу роботи.

1 ТЕОРЕТИЧНІ ОСНОВИ ВІДЛАГОДЖЕННЯ iOS-ЗАСТОСУНКІВ ЗА ДОПОМОГОЮ LLM-АГЕНТІВ

1.1. Відлагодження як ресурсомісткий процес розробки та передумови інтеграції LLM

Життєвий цикл розробки програмного забезпечення характеризується як процесами формування бізнес-логіки та імплементацією відповідного функціоналу, так і аналітичними зусиллями необхідними для його перевірки. Дослідження в галузі економіки програмної інженерії свідчать, що виявлення, локалізація та усунення дефектів – що в сукупності називаються налагодженням – забирають від 35% до 50% загальних ресурсів розробників. Така частка зусиль становить значне інтелектуальне навантаження, що вимагає постійного підтримання точних внутрішніх представлень стану виконання програми, залежностей потоку керування та обмежень середовища [1].

Процес діагностики відбувається за ітеративним циклом формування гіпотез та їх емпіричної перевірки. Щоб виявити розбіжність між фактичною та очікуваною поведінкою, розробник повинен створити чітку внутрішню модель логіки системи. В екосистемі iOS кілька специфічних для платформи факторів ще більше посилюють цю складність:

– Асинхронне виконання: Поширеність Grand Central Dispatch (GCD) та Swift Concurrency вносить недетермінованість, ускладнюючи відстеження причинно-наслідкових ланцюгів між потоками.
– Непрозорість фреймворків: Закритий характер основних фреймворків Apple часто унеможливлює прямий огляд переходів між станами, змушуючи покладатися на непряме спостереження.

– Множинність станів: Перетин реактивних станів інтерфейсу користувача, затримки мережі та локального зберігання даних створює багатовимірний простір пошуку для локалізації помилок.

Хоча LLM ефективно працюють із вихідним кодом, їхня здатність до відлагодження обмежена відсутністю доступу до стану виконання – регістрів, адрес пам'яті, ієрархії представлень – і тому модель не може відрізнити дефект у логіці коду від дефекту, що виникає лише в рантаймі. Без телеметрії в реальному часі запропоновані виправлення спираються на абстракцію вихідного коду, а не на фактичний стан скомпільованого процесу.

Перехід від пасивного агента до автономного вимагає інтеграції моделі у формальний цикл «міркування-дія-спостереження» (ReAct). Щоб подолати обмеження статичного аналізу, LLM має отримати можливість взаємодіяти з середовищем виконання через інтерфейс відлагоджувача, такий як LLDB. Ця інтеграція заснована на стандартизованому комунікаційному шарі, який дозволяє агенту здійснювати програмний контроль через API LLDB Scripting Bridge, виконувати динамічні запити до пам'яті процесу на основі гіпотез, що розвиваються, та досягати символьного заземлення під час виконання [2].

Наявність інструментарію відлагоджувача в агента LLM, дозволить йому долати механічні складнощі збору даних та керування процесом виконання, які включають знання відповідних низькорівневих команд, синтаксису та правил. Відповідальність розробника ж краще сфокусується на стратегічному міркуванні високого рівня.

1.2. Архітектура LLDB та API Scripting Bridge як основа програмного керування налагоджувачем

Ефективність агента прямо залежить від якості інтерфейсу з процесом, що налагоджується. LLDB спроектований не як утиліта командного рядка, а як бібліотека в складі інфраструктури LLVM - модульна і програмно розширювана. Така архітектура дозволяє керувати виконанням і інспектувати стан процесу через структурований API, а не через парсинг текстового виводу [3,4].

Ядром програмної взаємодії з LLDB є Scripting Bridge API (SB API) - інтерфейс на основі C++, що надає доступ до внутрішніх механізмів відлагоджувача зі скриптових мов, зокрема Python. Цей API організований у ієрархічну об'єктну модель, що відображає логічну структуру цільового процесу. Кореневим елементом ієрархії є “SBDebugger” - точка входу та координатор сеансів налагодження. Підпорядкований йому “SBTarget” представляє виконуваний бінарний файл разом із відповідними символами налагодження, тоді як “SBProcess” відповідає за стан виконання цільового процесу. Ієрархія продовжується через “SBThread” та “SBFrame”, надаючи доступ до конкретних контекстів виконання, локальних змінних і станів регістрів. Для LLM-агента ця структура є принциповою: вона перетворює стан запущеного процесу на дерево типізованих об'єктів із детермінованим доступом до кожного елемента [5,6].

Покладатися на текстовий вивід команд налагоджувача - поширений підхід у примітивних LLM-інтеграціях, що вносить до контекстного вікна неструктурований шум і перекладає на модель завдання парсингу. Це створює подвійний ризик: збільшення витрат токенів на представлення стану та накопичення помилок інтерпретації, що поширюються на наступні кроки міркування. SB API усуває цю проміжну стадію: замість рядків агент

отримує типізовані об'єкти - зокрема “SBValue” із метаданими про тип, адресу пам'яті та дочірні елементи - і може звертатися до стану процесу безпосередньо, без регулярних виразів і інтерпретації тексту. Це принципово для циклу ReAct: фази “Дія” та “Спостереження” спираються на фактичний стан пам'яті, а не на його текстове представлення [2,4].

Зрештою, архітектура LLDB забезпечує технічну основу, необхідну для системи з двох процесів, де механізм міркування високого рівня (сервер MCP) керує робочим процесом низького рівня (екземпляром LLDB). Здатність Scripting Bridge оцінювати складні вирази в контексті конкретного “SBFrame” дозволяє агенту вводити фрагменти коду на Swift, Objective-C або C безпосередньо у запущений процес. Ця можливість дозволяє агенту не тільки спостерігати за станом, але й активно маніпулювати ним для перевірки діагностичних гіпотез. Використовуючи цю програмну основу, інструментарій на базі LLDB та MCP дозволив би перетворювати абстрактні запити природньою мовою та мовою на точні послідовності викликів API, що б заповнило прогалину між наміром високого та контролем виконання низького рівня

1.3. Особливості налагодження в iOS: архітектура debugserver, порти завдань Mach та механізми безпеки

Технічна реалізація автономного налагодження на iOS визначається архітектурою безпеки платформи, яка володіє обмеженнями, нехарактерними іншим операційним системам. Хоча Scripting Bridge забезпечує логічний інтерфейс для керування, фізичне його втілення опосередковується ядром XNU та спеціалізованими системними службами. Розуміння цих обмежень є принциповим, оскільки вони визначають граничні умови, за яких повинен працювати LLM-агент - зокрема щодо

приєднання до процесів, доступу до пам'яті та керування потоком виконання.

Ключовою особливістю налагодження на iOS є обов'язкова модель віддаленого налагодження, що застосовується навіть у середовищі симулятора. Її основою є “debugserver” - компактний виконуваний файл-заглушка, розміщений на пристрої або в симуляторі, що взаємодіє з екземпляром LLDB на хості через GDB Remote Serial Protocol (RSP) [8]. Це розділення вносить затримку та потенційну нестабільність з'єднання, які агент повинен враховувати у фазі спостереження циклу ReAct. Крім того, “debugserver” є основним слухачем портів задач Mach і слугує мостом між командами агента та низькорівневими примітивами ядра - призупиненням потоків і читанням пам'яті. Відповідно, LLDB-клієнт повинен явно обробляти таймаути та розриви RSP-сесії і використовувати неблокуючі виклики SB API, щоб уникнути ситуації, коли зависла транзакція блокує черговий крок ReAct-циклу [9,10].

В основі моделі виконання iOS лежить ядро Mach, де управління процесами здійснюється через міжпроцесну комунікацію (IPC) на основі портів. Можливість налагодження процесу пов’язана з отриманням права task_for_pid - привілейованої операції, що дозволяє одному процесу керувати віртуальною пам’яттю та станом регістрів іншого. У iOS цей механізм суворо охороняється політикою безпеки системи, що вимагає вбудовування конкретних прав (таких як get-task-allow) у підпис коду цільового додатка. Якщо агент намагається підключитися до бінарного файлу Release, якому бракує необхідних прав, ядро відхилить запит із помилкою. Агент має розпізнавати такі відмови як обмеження середовища, а не як наслідок хибного міркування - таке розрізнення є принциповим для запобігання спробам «виправити» недоступний процес.

Механізми безпеки iOS - рандомізація розташування адресного простору (ASLR), обов'язковий підпис коду через AMFI та sandbox застосунків – усувають можливість прямого доступу до пам'яті процесу, залишаючи SB API єдиним надійним каналом взаємодії агента з цільовим процесом. ASLR змінює базову адресу завантаження при кожному запуску, тому агент змушений покладатися на символьне розв'язання через Scripting Bridge, а не на закешовані адреси [11]. Обов'язковий підпис коду унеможливлює довільну модифікацію потоку інструкцій - будь-яка несанкціонована зміна спричиняє виняток на рівні ядра. Sandbox у свою чергу окреслює границі спостережуваності: ресурси поза контейнером застосунку залишаються недоступними агенту навіть за наявності прав task_for_pid. За цих умов ефективність агента визначається не стратегією обходу обмежень, а повнотою і надійністю інструментів, побудованих поверх відкритого API.

1.4. Протокол Model Context Protocol: примітиви, транспортні рівні та архітектура взаємодії агентів і інструментів

Відсутність доступу до стану виконання, окреслена в попередніх розділах, усувається через стандартизований інтерфейс, що відокремлює міркування моделі від низькорівневої механіки налагоджувача. Model Context Protocol (MCP) слугує цим проміжним рівнем - відкритим стандартом двостороннього зв'язку між агентом LLM та зовнішніми джерелами даних. Через MCP взаємодія з LLDB Scripting Bridge перетворює середовище виконання iOS на структурований набір функціональних можливостей, доступних агенту [12].

Протокол визначає трирівневу архітектуру: host, client і server. Host - це застосунок, у якому виконується LLM, і який ініціює взаємодію з

користувачем. Client є компонентом host-а, що керує життєвим циклом виклику інструментів і підтримує сесію з конкретним сервером. Server інкапсулює зовнішнє джерело можливостей - у випадку відлагодження це сесія дебагера - і надає його функціональність через стандартизовану схему примітивів [13]. Таке розмежування ролей виносить специфіку джерела даних за межі моделі: host і client оперують лише типізованими описами інструментів і ресурсів, а вся складність взаємодії з конкретним середовищем - у тому числі його платформні обмеження - локалізована на стороні server-а. Як наслідок, заміна одного джерела можливостей на інше (дебагер на профайлер, локальна сесія на віддалену) не вимагає змін у логіці агента.

MCP класифікує взаємодії агента із середовищем на три примітиви, кожен з яких відповідає конкретній фазі циклу ReAct. Інструменти реалізують фазу «Дія» – вони дозволяють агенту виконувати операції, що змінюють стан: приєднання до процесу, встановлення точок зупину, ін'єкцію коду. Ресурси відповідають фазі «Спостереження» - надають доступ лише для читання до потоків даних, як-от журнали ОС або ієрархія представлень. Промпти слугують структурованими шаблонами, що задають початкову діагностичну стратегію агента. На відміну від взаємодії через командний рядок, усі три примітиви є строго типізованими та валідованими за схемою, що унеможливлює некоректно сформовані запити і гарантує детермінований зворотний зв'язок у разі відмови інструменту.

Поряд із розмежуванням ролей між host, client і server, MCP володіє двома транспортними рівнями: stdio для локального двопроцесного сценарію, де host і server виконуються на одній машині, та Streamable HTTP для віддалених або сценаріїв, які включають кількох користувачів. Вибір рівня визначається способом розгортання і вимогами до затримки зворотного зв'язку. Для задачі автономного відлагодження критичною є

мінімальна затримка між дією агента та спостереженням за станом процесу: великі затримки порушують часову узгодженість фази “Спостереження” циклу ReAct і провокують агента до повторного запиту даних, що штучно нарощує контекст. Stdio-транспорт забезпечує таку мінімальну затримку завдяки відсутності потреби в з’єднанні з мережею, що робить його належним вибором для локального відлагодження [13].

1.5. Парадигма ReAct як модель поведінки автономного агента в динамічному середовищі

Технічна інтеграція LLDB та MCP забезпечує агента інструментальною основою, однак логічна структура прийняття рішень повинна враховувати мінливий стан виконання. Підхід Chain-of-Thought дозволяє моделі розкладати задачу на послідовні кроки, але залишається нечутливим до змін зовнішнього стану, що виникають після початкового запиту [14]. Парадигма ReAct усуває це обмеження, поєднуючи міркування з взаємодією із середовищем - агент адаптує діагностичну стратегію на основі зворотного зв'язку від налагоджувача після кожної дії.

Перевага ReAct над CoT полягає у замкнутому циклі взаємодії із середовищем. CoT дозволяє моделі відтворити причинно-наслідковий ланцюг, що веде до збою, але не перевірити стан конкретного покажчика або потоку без втручання людини. ReAct формалізує цю перевірку: агент генерує “думку” - внутрішнє міркування, - після чого виконує “дію” через виклик інструменту MCP. Отримане “спостереження” з LLDB або підтверджує гіпотезу, або вимагає її перегляду (Рис 1.1).

Рисунок 1.1 Схематичне зображення ReAct циклу

Для автономного відлагоджувача цей цикл є необхідною умовою роботи в недетермінованих середовищах iOS, де припущення щодо race condition або витоку пам'яті потребує постійної верифікації на основі даних безпосередньо з процесу виконання.

Стабільність циклу ReAct прямо залежить від інформативності фази спостереження. Розробник здатен інтерпретувати неоднозначний вивід терміналу або несподіваний код помилки, спираючись на власний досвід, але для LLM-агента неінформативний зворотний зв'язок розриває причинно-наслідковий зв'язок між дією та наступним міркуванням і провокує ітеративні галюцинації. Із цього випливає вимога до проєктування інструментів: відповіді повинні бути типізованими, а не текстовими, і містити явну причину відмови - чи то непридатність точки зупину, чи то таймаут обчислення виразу. Структурований зворотний зв'язок мінімізує контекстний шум і зберігає семантику фази спостереження. Натомість низькоінформативний сигнал змушує агента інтерпретувати відмову як підтвердження альтернативної гіпотези, унаслідок чого наступне міркування зміщується у напрямок, не обумовлений фактичним станом процесу [2,15].

Зрештою, парадигма ReAct забезпечує поведінкову логіку, необхідну для роботи в умовах обмежень платформи. Коли агент отримує відмову на кшталт task_for_pid, цикл дозволяє йому розпізнати обмеження середовища і перейти до альтернативного діагностичного шляху, а не повторювати

недійсну операцію. Таким чином, агент залишається прив'язаним до фактичного стану процесу, і задача налагодження перетворюється з імовірнісної генерації тексту на цілеспрямоване дослідження. Саме ця прив'язка до рантайм-стану підводить до проблеми символьного заземлення - семантичного зв'язку між міркуванням агента та станом застосунку під час виконання.

1.6. Проблема прив’язки символів у середовищі виконання налагоджувача

Ефективність циклу ReAct, описаного в попередньому розділі, залежить від здатності агента зіставляти абстрактні лінгвістичні поняття з конкретними фізичними об’єктами. У семантиці та штучному інтелекті «проблема символьного заземлення» визначає завдання, пов’язане з тим, як слова чи символи набувають значення через їхній зв’язок із зовнішнім світом [16]. У контексті автономного налагодження ця проблема проявляється як розрив між статистичним розумінням моделі ідентифікатора в коді та його фактичною реалізацією як послідовності байтів за конкретною адресою пам'яті під час виконання [17].

Надійний агент повинен досягти заземлення у трьох різних вимірах, щоб успішно орієнтуватися в середовищі виконання iOS. По-перше, лексичне заземлення передбачає пряме відображення імені змінної на SBValue в межах конкретного SBFrame. Хоча це найпростіший рівень, він вимагає від агента точного вирішення питань області дії та затінення. По-друге, структурне прив’язування стосується взаємозв’язку між абстракціями інтерфейсу користувача високого рівня та їхніми об’єктами, що знаходяться в пам’яті. Наприклад, прив’язування “кнопки входу”, на яку посилається звіт про помилку, до конкретної адреси пам’яті через

рекурсивний опис ієрархії представлень є необхідним для діагностики дефектів макета або обробки подій. Нарешті, часове заземлення враховує той факт, що фізична реалізація символу є дійсною лише в певні моменти виконання (тобто коли процес “зупинено”). Без збереження цієї часової обізнаності агент може спробувати отримати доступ до застарілих адрес пам’яті або посилатися на змінні зі стекових кадрів, які більше не є частиною активного контексту виконання [16].

Стандартні LLM працюють на виняткова ймовірнісній основі, прогнозуючи наступний токен на основі вивчених шаблонів у вихідному коді. Це призводить до явища, коли модель припускає, що змінна існує або має певне значення, оскільки це статистично ймовірно, з огляду на навколишній код. Інтегруючись із LLDB через міст MCP, агент переходить від цієї внутрішньої, текстової симуляції до системи, заземленої у фізичній реальності купи та стека процесу. Кожна “думка” в циклі ReAct, що стосується змінної, повинна бути перевірена через “дію”, яка запитує Scripting Bridge [2]. Ця перевірка перетворює внутрішнє представлення агента з серії ймовірнісних припущень на емпіричну модель фактичного стану виконання програми.

Успішне заземлення символів слугує критичним захисним механізмом проти накопичення контекстного шуму. Коли агент успішно пов’язує ідентифікатор на рівні коду з об’єктом виконання, це зменшує семантичну неоднозначність наступних діагностичних кроків. І навпаки, якщо агент не може заземлити символ – наприклад, посилаючись на змінну, яка була оптимізована компілятором – він повинен бути здатним розпізнати цю невдачу як відсутність заземлення, а не як помилку у своєму логічному міркуванні. Це розрізнення є життєво важливим для збереження цілісності діагностичного процесу. У міру того, як агент заглиблюється в сесію налагодження, якість його символьного заземлення визначає його здатність

зберігати відповідність між внутрішнім представленням і фактичним станом процесу. Деградація цієї відповідності створює умови для феномену DDI, який розглянуто у наступному пункті.

1.7. Управління станом агента та феномен індексу зниження ефективності налагодження (DDI)

Останнє теоретичне завдання в галузі автономного налагодження пов’язане з фундаментальною суперечністю між безстановою природою великих мовних моделей та процесом діагностики програмного забезпечення, що значною мірою залежить від історії. У міру проходження агентом LLM сеансу налагодження він повинен синтезувати інформацію з розширюваної послідовності вихідних даних інструментів, трасування стека та станів змінних. Це накопичення даних зрештою спричиняє системне зниження якості міркування – явище, яке в цьому дослідженні формалізовано як індекс занепаду налагодження (DDI).

DDI припускає, що ефективна здатність агента до міркування не є постійною, а погіршується як функція глибини взаємодії. Ця залежність виражається експоненціальною моделлю занепаду

$$E(t) = E_0 \cdot e^{-\lambda t},$$

де $E(t)$ представляє діагностичну ефективність агента в момент часу t, E_0 - початкову базову продуктивність, а λ – константу занепаду, що визначається складністю контексту. На практиці, у міру збільшення кількості токенів в історії розмови, здатність моделі звертати увагу на критичні символи “фактичної істини” з початку сеансу зменшується.

Окрім того, константа λ не є універсальною характеристикою моделі; вона залежить від складності задачі, об'єму вихідних даних інструментів, обраного алгоритму керування контекстом та від конкретної LLM, що

застосовується. У межах цієї роботи λ розглядається як емпіричний параметр, що підлягає вимірюванню на основі сесійних метрик - кількості кроків до локалізації причини, частоти повторних запитів того самого стану, тощо.

Прискорення DDI зумовлене двома основними системними факторами: шумом у контекстному вікні та поведінковими циклами. Шум контекстного вікна виникає, коли необроблені вихідні дані команд налагоджувача – як-от докладні дампи пам'яті або розлогі траси стека – перенасичують механізм уваги моделі низькоентропійними даними, заглушаючи діагностичні індикатори з високим сигналом. У літературі про LLM-агенти такий стан описують терміном «attention tunneling», або «тунельне бачення»: агент фіксується на нерелевантних підзадачах і ігнорує суперечливі докази, надані налагоджувачем у попередніх циклах. Поведінковий цикл описує стан, коли агент повторює послідовність невдалих дій через нездатність розпізнати, що його поточна стратегія вже зазнала невдачі.

Щоб протидіяти DDI, автономна система повинна вийти за межі пасивного логування і перейти до стратегій активного управління станом. До них належать «обрізка контексту», коли агент вибірково відкидає надлишкові результати інструментів, зберігаючи узагальнені висновки, та «ієрархічне узагальнення», яке стискає історію сеансу у високорівневу діагностичну карту. Крім того, впровадження механізмів виявлення циклів та стратегічних «свіжих стартів» – скидання контексту з збереженням найбільш ймовірних обґрунтованих гіпотез – дозволяє системі підтримувати високий $E(t)$ протягом більш тривалих періодів.

Визначення індексу занепаду налагодження завершує теоретичну основу цієї роботи. Цей розділ перейшов від матеріальної бази LLDB та безпеки iOS до поведінкової моделі ReAct та семантичних вимог Symbol

Grounding. Проблематика між здатністю агента взаємодіяти з динамічним середовищем та його властивою схильністю до логічного занепаду визначає центральну проблему дослідження. У наступних розділах будуть описані практичні підходи та технічні деталі реалізації системи, призначеної не лише для виконання команд налагоджувача, а й для активного управління станом автономного діагностичного дослідження [18].

Висновки до розділу

У першому розділі аргументовано потребу доступу агентом до процесу виконання iOS-застосунку для сприяння його налагодженню. LLDB через Scripting Bridge API забезпечує необхідний програмний контроль над процесом, однак модель безпеки платформи iOS накладає додаткові архітектурні обмеження. MCP задає стандартизований інтерфейс між LLM і налагоджувачем, а парадигма ReAct – предметну модель поведінки агента в динамічному середовищі. Проблема символьного заземлення формалізована на трьох рівнях і визначає межі доступності стану процесу для агента. Введена модель DDI описує деградацію ефективності через накопичення шуму в контекстному вікні, що становить теоретичну основу для оцінки роботи агента в завданнях налагодження.

2 ОБҐРУНТУВАННЯ ВИБОРУ АРХІТЕКТУРИ ТА ФУНКЦІОНАЛУ MCP-СЕРВЕРА

2.1. Вибір моделі ізоляції середовища виконання між сервером MCP та LLDB

Основна технічна проблема при розробці агента налагодження на базі LLM для екосистеми iOS полягає в узгодженні двох взаємовиключних середовищ виконання. MCP SDK, тобто набір із засобів та інструментів розробки MCP-серверів, вимагає використання Python 3.10 або вище. І навпаки, LLDB Scripting Bridge (SB API), наданий Apple, тісно пов’язаний із версією Python, що входить до дистрибутива Xcode - наразі це Python 3.9. Ця невідповідність версій створює конфлікт під час виконання, що унеможливлює виконання як логіки протоколу, так і логіки керування відлагоджувачем у рамках одного єдиного процесу [13].

Щоб вирішити цю архітектурну проблему, було оцінено дві основні моделі ізоляції:

– Уніфіковане середовище виконання з примусовою сумісністю: Цей підхід передбачає спробу перенести MCP SDK на старіше середовище виконання Python або, як альтернатива, змусити LLDB використовувати нестандартну інсталяцію Python. Хоча це мінімізувало б накладні витрати на комунікацію, цей варіант було відхилено через великий обсяг технічного боргу та значний ризик порушення цілісності відображення символів LLDB, що покладається на бінарну сумісність із наданим системою фреймворком Python.
– Архітектура з двома процесами та міжпроцесною комунікацією (IPC): Ця модель ізолює сучасний сервер MCP та застарілий інтерпретатор LLDB у окремі процеси операційної системи. Комунікація

здійснюється за допомогою протоколу серіалізованих повідомлень через стандартні канали вводу/виводу або сокети Unix.

Вибір базувався на чотирьох ключових критеріях: ізоляція збоїв, складність розгортання, переносимість набору інструментів та затримка виконання.

Архітектура з двома процесами виявилася оптимальним компромісом. Завдяки дотриманню суворих меж процесів система забезпечує надійну ізоляцію збоїв: робочий процес LLDB можна перезапустити або скинути без втрати контексту діагностики вищого рівня, що підтримується сервером MCP. Схему взаємодії компонентів наведено на рис. 2.1.

Рисунок 2.1. Схематичне зображення архітектури взаємодії компонентів системи через IPC-міст

Описана архітектура містить дві межі міжпроцесної комунікації: між MCP-сервером і LLDB-воркером, обговорену вище, і між клієнтським застосунком LLM-host та самим MCP-сервером. Друга межа реалізована через stdio-транспорт, передбачений специфікацією MCP, – це єдиний раціональний вибір для локального двопроцесного сценарію: він не вносить накладних витрат TCP/IP, не потребує окремого механізму автентифікації для loopback-інтерфейсу і дозволяє хосту керувати життєвим циклом server-процесу через стандартні засоби операційної системи. Альтернативний транспорт Streamable HTTP, орієнтований на віддалені та багатокористувацькі розгортання, не дає переваг у сценарії, де LLDB-сесія

за визначенням прив'язана до локально підключеного симулятора або пристрою.

Зрештою, ця архітектура забезпечує повну сумісність з офіційними дистрибутивами Xcode, одночасно надаючи агенту LLM стабільне, ізольоване середовище для низькорівневої перевірки пам'яті та введення коду, незалежне від обмежень середовища виконання хост-додатка.

2.2. Декомпозиція простору дій агента на інструменти MCP

Після вирішення проблеми з передачею викликів між MCP та LLDB постає наступна задача: розбиття функціональності Scripting Bridge на дискретний набір інструментів, що будуть викликатися агентом. Простір SB API охоплює безліч методів класів “SBDebugger”, “SBTarget”, “SBProcess”, “SBThread” та “SBFrame”, і пряме перенесення кожного з них у схему MCP створило б каталог, у якому модель змушена самостійно відновлювати послідовності викликів для кожної типової операції відлагодження. З іншого боку, можна застосувати підхід, за якого кожен MCP-інструмент інкапсулює сценарій довжиною у десятки кроків SB API, скорочуючи кількість round-trips, але приховуючи від агента проміжні стани, на основі яких він міг би коригувати стратегію.

Порівняння варіантів декомпозиції велося за трьома критеріями, кожен з яких пов'язаний із теоретичними результатами першого розділу: довжиною траєкторії ReAct (кількістю пар «дія–спостереження» на типову діагностичну задачу), обсягом токенів, що додаються до контексту після одного виклику, та ризиком неоднозначної інтерпретації результату моделлю. За першими двома критеріями атомарні інструменти програють агрегованим, бо щоб зчитати значення трьох локальних змінних після зупинки на брейкпоінті, атомарна декомпозиція вимагає щонайменше

чотирьох послідовних викликів – wait_for_stop для підтвердження факту зупинки і трьох викликів read_variable, по одному на кожну змінну, – кожен з яких супроводжується власним ReAct-циклом і додає до контексту як саму відповідь, так і проміжне міркування агента. За третім критерієм виграш на боці атомарних інструментів: чітка відповідність одній операції SB API мінімізує ризик семантичних помилок, оскільки звужена сигнатура і однозначна семантика результату скорочують простір неправильних інтерпретацій моделлю – спостереження, що узгоджується з рекомендаціями Anthropic щодо проектування інструментів для агентів, де простота схеми параметрів і однозначність опису прямо корелюють із точністю вибору інструмента. Оскільки жодна з крайніх стратегій не є оптимальною за всіма трьома критеріями одночасно, обрано гібридний підхід: базові операції SB API зберігаються як атомарні інструменти, а сценарії, що повторюються з високою частотою і мають стабільну форму, агрегуються в композитні інструменти зі структурованою відповіддю.

Реалізована декомпозиція налічує сімнадцять інструментів, розподілених на п'ять функціональних груп: керування життєвим циклом сесії, керування точками зупину, керування виконанням, інспекція стану та ін'єкція коду, читання журналів операційної системи. Нижче розкривається призначення кожного інструмента у межах своєї групи.

Керування життєвим циклом сесії

- attach_to_process – приєднує сесію LLDB до запущеного процесу iOS-симулятора або пристрою за іменем. Опціональний прапор
- wait_for_launch переводить виклик у режим очікування запуску, що дозволяє встановити точки зупину до того моменту, коли застосунок фактично з'явиться у списку процесів.

- detach_from_process – від'єднує LLDB від поточного цільового процесу без його зупинки чи завершення.
- get_process_status – повертає поточний стан процесу (виконується, зупинено, завершено), його PID та ім'я. Слугує опорним джерелом істини для агента між циклами «дія–спостереження».

Керування точками зупину.

- set_breakpoint – встановлює точку зупину за парою «файл–рядок» або за іменем символу. Приймає умовний вираз, список LLDB-команд для виконання при спрацюванні та прапор auto_continue, який автоматично відновлює виконання після обробки команд.
- list_breakpoints – повертає перелік усіх активних точок зупину з їхніми ID, локаціями, лічильниками спрацювань, прикріпленими командами та станом активності.
- delete_breakpoint – видаляє точку зупину за числовим ID.
- enable_breakpoint – вмикає або вимикає наявну точку зупину без її видалення, що дозволяє тимчасово виключати її зі сценарію без втрати конфігурації.

Керування виконанням.

- resume_process – відновлює виконання зупиненого процесу до наступної події (спрацювання точки зупину, сигналу, аварійного завершення або виходу). Запуск цього інструмента активує фоновий потік опитування подій, що обслуговує auto-continue брейкпоінти.
- wait_for_stop – блокує виклик до моменту зупинки процесу або до закінчення таймауту. Опціональний параметр expressions приймає список виразів, що обчислюються безпосередньо в момент зупинки;

їхні значення повертаються під ключем variables у складі однієї відповіді.

Інспекція стану та ін'єкція коду.

- read_variable – обчислює ім'я змінної або вираз у контексті поточного стекового кадру і повертає його тип, значення та дочірні елементи. Закриває рівень лексичного символьного заземлення, розглянутий у пункті 1.6.
- execute_command – передає довільну текстову команду LLDB до відлагоджувача і повертає її вивід. Слугує універсальним резервним механізмом для одноразових запитів, не покритих спеціалізованими інструментами.
- list_threads – повертає перелік усіх потоків зупиненого процесу з ID, іменами, ім'ям GCD-черги та причинами зупинки.
- select_thread – встановлює активний потік за ID, перенаправляючи на нього подальші операції з кадрами та змінними.
- get_stack_trace – повертає повний стек викликів потоку у вигляді впорядкованого переліку кадрів з іменами функцій, шляхами до файлів та номерами рядків.
- select_frame – встановлює активний кадр стеку за індексом, змінюючи область видимості, у якій виконуються read_variable та inject_code.
- inject_code – обчислює довільний код мовою Swift, Objective-C або C у контексті цільового процесу через механізм оцінки виразів LLDB. Виконання має повний доступ до “self” та локальних змінних активного кадру; помилки класифікуються як compile_error або runtime_error на основі шаблону повідомлення, що дозволяє відрізняти дефекти синтаксису від помилок під час обчислення.

Читання системних журналів

– read_os_log – захоплює потік журналу iOS-симулятора протягом обмеженого вікна часу (до десяти секунд) і повертає до двохсот рядків після застосування фільтра. Вимагає принаймні одного критерію – імені процесу, підсистеми або виразу NSPredicate. Реалізований безпосередньо на стороні MCP-сервера, оминаючи міст до LLDB-воркера, оскільки операція не пов'язана з API Scripting Bridge.

Більшість із цих інструментів залишаються атомарними у тому сенсі, що відображаються на одну операцію SB API і повертають єдину типізовану структуру. Натомість три інструменти свідомо спроектовано як агреговані, кожен з яких розв'язує конкретний клас неоптимальних траєкторій ReAct.

Перший із них – wait_for_stop з опціональним параметром expressions. У базовому сценарії агент спершу очікує зупинки, отримує підтвердження факту події, після чого окремими викликами read_variable зчитує значення кожного виразу, що його цікавить. Коли список виразів відомий заздалегідь – а це справджується у переважній більшості випадків, бо умова точки зупину і список цікавих змінних формулюються в одній “думці” агента, – пакетне обчислення виразів безпосередньо у момент зупинки усуває від двох до п'яти зайвих round-trips і повертає всі значення під ключем variables єдиної відповіді.Скорочення траєкторії має прямий вплив на константу λ із моделі деградації, введеної у пункті 1.7, оскільки кожен round-trip додає до контексту як саму відповідь інструмента, так і проміжне міркування агента.

Другий приклад агрегації – параметризація set_breakpoint полями condition, commands і auto_continue. Замість того щоб виставляти точку зупину окремим викликом, далі прикріплювати до неї команди ще одним

викликом і нарешті активувати режим автоматичного продовження третім, агент описує всю поведінку точки зупину одним зверненням. Поєднання commands та auto_continue дозволяє реалізувати логуючі брейкпоінти – точки, що друкують значення змінних і не призупиняють процес, – повністю на стороні воркера, без участі моделі у кожній спрацьованій події.

Третій приклад – read_os_log, що інкапсулює запуск “xcrun simctl spawn booted log stream”, побудову NSPredicate-фільтра з параметрів process_name, subsystem і predicate, обмеження часу зйомки та усічення виводу до двохсот рядків. Ця функціональність взагалі лежить поза SB API і вимагала б від агента самостійного формування shell-команди через execute_command, з усіма ризиками неправильно екранованих метасимволів і непередбачуваного обсягу виводу.

2.3. Розподіл відповідальності за рівні символьного заземлення між інструментами агента

Три рівні заземлення, визначені у пункті 1.6 (лексичний, темпоральний, структурний), породжують різні інженерні вимоги: кожен з них потребує власного архітектурного механізму, і вибір цього механізму у кожному випадку визначається структурою SB API. Практична задача цього пункту – встановити, які з трьох рівнів закриваються виділеними типізованими інструментами, а які залишаються прив'язаними до універсального механізму виконання команд, і пояснити, чому цей розподіл не допускає однорідного трактування.

Лексичне заземлення отримує пряму підтримку з боку SB API: інтерфейс SBFrame надає методи EvaluateExpression і FindVariable, кожен з яких повертає типізований об'єкт SBValue з метаданими про тип, поточне значення, дочірні елементи та адресу в пам'яті. Реалізований інструмент

read_variable слугує адаптером над цим інтерфейсом – на вході текстовий вираз, на виході серіалізована структура з полями для типу, значення та дочірніх елементів. Агент не виконує парсинг тексту: відповідь уже містить типову інформацію, яку модель напряму використовує у наступній “думці” циклу ReAct. Цей рівень – той випадок, коли абстрактне поняття символьного заземлення і модель SB API структурно збігаються, а спеціалізований інструмент лише проектує наявний інтерфейс у схему MCP без семантичних втрат.

Темпоральне заземлення не вимагає окремого інструмента – воно проявляється як властивість архітектури керування виконанням. Запит агента “яке значення x” є осмисленим лише у визначений момент зупинки процесу, і цей момент створюється інструментом wait_for_stop, що синхронізує агента з подією зупинки. Будь-яка наступна інспекція через read_variable, get_stack_trace чи list_threads автоматично прив'язана до того самого темпорального зрізу, оскільки SB API дозволяє доступ до стекового кадру і локальних змінних лише у стані зупинки. Групове обчислення виразів у wait_for_stop, обґрунтоване в пункті 2.2 з міркувань довжини траєкторії ReAct, тут набуває додаткового сенсу: воно явно прив'язує обчислення виразів до того самого моменту зупинки, що й подія брейкпоінта, усуваючи інтервал між “зупинилось” і “зчитав стан”. Темпоральне заземлення, відповідно, реалізоване не як інструмент, а як інваріант пари “механізм керування виконанням + інспекційний набір”.

Структурне заземлення, на відміну від двох попередніх рівнів, не допускає прямого відображення на типізований інтерфейс SB API. Щоб зіставити природньомовний запит агента “знайди кнопку Submit у поточній сцені” з конкретним екземпляром UIView за конкретною адресою, потрібен доступ до ієрархії представлень рантайму застосунку. Однак ця ієрархія не описана в DWARF-інформації, яку споживає LLDB: вона є рантайм-

структурою, що динамічно конструюється UIKit через механізми ObjC-рантайму і утримується в купі як граф об'єктів-екземплярів UIView, пов'язаних властивістю subviews. SB API оперує на рівні типів, відомих із debug info, тоді як ієрархія представлень існує на рівні екземплярів об'єктів, доступних лише через методи самого фреймворку – recursiveDescription, _autolayoutTrace, debugDescription. Ці методи повертають форматовані текстові рядки, призначені для читання людиною, а не структуровані представлення [19].

Наслідком цієї асиметрії є те, що в поточній реалізації структурне заземлення проходить через універсальний execute_command. Агент викликає “po [self.view recursiveDescription]”, отримує багаторядковий текстовий вивід і парсить його на своїй стороні. Це рішення функціонально повне – агент досягає цілі, – але суперечить принципам, сформульованим у першому розділі: відповідь неструктурована, токени витрачаються на парсинг, контекстне вікно заповнюється низькоентропійним шумом, що прискорює DDI. Витрата вимірювана: типовий вивід recursiveDescription для помірно складної сцени займає сотні токенів, з яких менш ніж п'ята частина несе діагностичну цінність для конкретної задачі.

Відсутність типізованого інструмента для структурного заземлення є структурним наслідком розмежування зон відповідальності між LLDB і UIKit, а не обмеженням реалізації.. Щоб закрити цей розрив у межах поточної архітектури, потрібен окремий інструмент, що делегує виклик “recursiveDescription” робочому процесу LLDB, виконує детерміновану обробку виводу безпосередньо на стороні робочого процесу і повертає агенту типізоване JSON-дерево з полями для класу, фрейму, адреси та дочірніх вузлів. Агент у такому випадку отримував би структуровану відповідь тієї самої природи, що й для лексичного заземлення, а витрата на парсинг сплачувалась би одноразово на стороні робочого процесу, а не на

кожній ітерації циклу ReAct. Імплементація такого інструмента зафіксована як напрям подальших робіт і виноситься за межі поточної версії системи.

2.4. Перетворення DDI на вимірюваний показник: вимірювання як передумова керування

Феномен деградації ефективності відлагодження, формалізований у пункті 1.7 експоненційною моделлю $E(t) = E_0 \cdot e^{-\lambda t}$, ставить перед архітектурою агента питання, що допускає два протилежні розв'язки. Перший шлях – побудувати систему, яка пасивно реєструє метрики кожного виклику інструмента і зберігає їх для подальшого аналізу. Другий – одразу інтегрувати у воркер активні стратегії протидії деградації: виявлення поведінкових циклів за хешем послідовності (PC, stack signature), обрізку контексту за критерієм релевантності, стратегічний рестарт сесії, ієрархічне узагальнення історії викликів. Архітектурна вага цих шляхів суттєво відрізняється. Перший потребує лише пасивного компонента логування з боку MCP-сервера. Другий вимагає окремого модуля керування контекстом, що перехоплює відповіді інструментів перед їхньою серіалізацією у потік MCP і ухвалює рішення про їхню модифікацію, заміщення або супровід попереджувальним повідомленням.

Поточна архітектура реалізує перший шлях. Це рішення обґрунтовується трьома міркуваннями. По-перше, активні стратегії параметризуються пороговими значеннями, які не виводяться з теоретичної моделі. Формула $E(t) = E_0 \cdot e^{-\lambda t}$ описує форму кривої, але не задає, після якого числа повторюваних викликів варто вважати поведінку циклом, ні який обсяг контексту ініціює обрізку. Без емпіричної оцінки λ, отриманої з реальних сесій відлагодження, ці пороги довелось би обирати з евристичних міркувань. А евристика, не підкріплена вимірами, виявиться або надмірно

агресивною (видалення релевантного контексту), або надмірно консервативною (відсутність ефекту). По-друге, оцінка результативності будь-якої активної стратегії потребує базової лінії – траєкторії агента без такої стратегії, виміряної у тих самих умовах. Інтеграція стратегій до накопичення цієї базової лінії унеможливлює подальше порівняння, оскільки обидві серії експериментів виконуватимуться вже над модифікованою системою. По-третє, активні стратегії модифікують потік даних, що повертається агенту. Некоректний дизайн перетворює їх на самостійне джерело деградації: обрізка може видалити критичну гіпотезу з контексту, а хибне виявлення циклу – ініціювати рестарт сесії на правильній траєкторії.

Вимірювальний шар реалізується через компонент SessionLogger – синглтон у процесі MCP-сервера. Він вбудований у міжпроцесний міст і перехоплює кожен виклик інструмента перед поверненням результату агенту. Структура запису фіксує монотонно зростаючий порядковий номер, UTC-таймстамп, ім'я RPC-методу, повний словник параметрів і результату, тривалість виклику з точністю до чотирьох десяткових знаків, джерело виклику (bridge для основного потоку або direct для read_os_log, що оминає міст), булевий прапор помилки та класифікацію помилки для inject_code (compilation чи runtime). При завершенні сесії воркер серіалізує накопичені записи у JSON-файл “logs/session_{session_id}.json”. До файлу додається обчислений підсумок: загальна кількість викликів, тривалість сесії, лічильник “дієвих” інструментів (resume_process, wait_for_stop, read_variable, inject_code, execute_command) як проксі для кроків до локалізації причини, частота компіляційних помилок у inject_code, розподіл викликів за типом інструмента та середня тривалість виклику у розрізі інструментів.

Вибір саме цього набору полів пояснюється вимогою забезпечити мінімальне достатнє покриття для відновлення λ пост-фактум. Кожне поле відповідає за конкретний аспект, без якого крива деградації або не може бути побудована, або була б некоректно інтерпретована.

Послідовність порядкових номерів і таймстампів задає вісь t кривої деградації. Порядкові номери дають черговість подій, а таймстампи – реальний інтервал між ними. 30 викликів за 5 хвилин і 30 викликів за годину – це різні режими роботи агента, навіть якщо обидва завершились локалізацією помилки.

Лічильник «дієвих» інструментів задає кінцеву точку траєкторії, проти якої нормуються інші метрики. До цієї підмножини входить п'ять інструментів, що реально просувають відлагодження вперед: resume_process, wait_for_stop, read_variable, inject_code, execute_command. Інші виклики (на кшталт get_process_status чи list_breakpoints) лише перевіряють стан системи і не повинні впливати на оцінку довжини сесії. Маючи цю міру, можна переходити від абсолютних значень до часток: замість агент припустився 10 помилок отримуємо “10 помилок на 50 продуктивних кроків – 20%”. Тільки в такому вигляді сесії різної тривалості стають порівнюваними.

Частота компіляційних помилок у inject_code слугує спостережуваним проксі для $E(t)$. Безпосередньо виміряти ефективність агента неможливо – це внутрішня якість його роботи. Однак коли агент пише вираз на кшталт "po self.userManger.curentUser" із друкарськими помилками в іменах змінних, LLDB відмовляє у компіляції і повертає типізовану помилку. Чим довша сесія, тим частіше такі помилки трапляються, бо агент поступово втрачає відповідність між власним уявленням про код і його реальним станом у пам'яті процесу. Це і є втрата

символьного заземлення, обговорена у пунктах 1.6 та 2.3, – але виражена через зовні спостережуваний індикатор.

Розподіл викликів за типом інструмента дозволяє виявити повторювані патерни звернень. Якщо в записах видно, що агент сім разів поспіль викликав read_variable для тієї самої змінної, або чергував get_stack_trace → list_threads → get_stack_trace → list_threads без проміжного виконання, це сигналізує про застрягання на одній гіпотезі. Такі патерни – природні кандидати для майбутнього механізму виявлення циклів.

Середня тривалість виклику в розрізі інструментів фіксує системні вузькі місця, які інакше можна було б сплутати з деградацією агента. Якщо read_os_log стабільно займає 3 секунди, а сесія містить 20 таких викликів, це додає до загальної тривалості хвилину системного простою, який не має відношення до якості міркування. Без розподілу за інструментами довга сесія могла б помилково інтерпретуватись як ознака зниженого $E(t)$, тоді як насправді вона спричинена затримкою конкретного інструмента.

Нарешті, повні параметри і результати кожного виклику зберігаються у сирому вигляді. Це означає, що якщо на етапі експериментальної оцінки виникне потреба в новій метриці – наприклад, частоті переходів між певними парами інструментів або середній глибині стека на момент виклику inject_code, – її можна обчислити безпосередньо з JSON-логів. Сам код SessionLogger при цьому не потребує модифікації, і повторно проганяти сесії не доводиться.

Вимірювальний шар спроектований як фундамент, на якому активні стратегії протидії DDI зможуть бути надбудовані без модифікації нижчих рівнів архітектури. Їхній дизайн вимагає попереднього емпіричного етапу – побудови кривої $E(t)$ на репрезентативному наборі сесій, що становить предмет розділу 3.

Висновки до розділу

У другому розділі обґрунтовано застосовані в розробці архітектурні рішення. Несумісність версій Python між MCP SDK і вбудованим інтерпретатором LLDB зумовила двопроцесну архітектуру з JSON-RPC через subprocess і stdio-транспорт як оптимальний варіант за критерієм затримки циклу ReAct. Простір дій агента декомпозовано у гібридний набір інструментів: базові операції залишаються атомарними, повторювані послідовності агрегуються для скорочення round-trips і сповільнення накопичення шуму в контекстному вікні. Розподіл відповідальності між інструментами щодо рівнів символьного заземлення виявив структурну асиметрію: лексичний і темпоральний рівні закриті повністю, тоді як структурне заземлення UI-ієрархії підтримане лише через універсальний “execute_command” – наслідок архітектурного розмежування між LLDB і UIKit. Вимірювання DDI вирішено на користь пасивного підходу: SessionLogger фіксує метрики кожного виклику, що є достатньою базою для відновлення константи деградації λ і закладає фундамент для активних стратегій протидії у подальших роботах.

3 ОБҐРУНТУВАННЯ ОБРАНИХ МЕТРИК ОЦІНЮВАННЯ, ЇХ ЗАСТОСУВАННЯ ТА АНАЛІЗ

3.1. Метод та схема експериментального оцінювання

Оцінювання ефективності розробленого інструменту за використання агентом ставить за мету показати, на скільки кроків доступ до стану виконання скорочує локалізацію дефекту порівняно з агентом, що має лише вихідний код і текст помилки. Звідси й дизайн оцінювання – два агенти на тому самому наборі дефектів, що різняться єдиним фактором: наявністю доступу до рантайму. Завданням є перевірити, чи цінність інструменту визначається символьним заземленням у рантаймі, а не самим фактом його наявності. Якщо так, то виграш від доступу до стану виконання має зростати тим більше, чим глибше причина дефекту прихована від статичного аналізу коду; саме ця залежність – а не сам факт скорочення траєкторії – і є тим, що піддається перевірці.

Рішення провести оцінювання на одному застосунку значного обсягу – понад 200 екранів (ViewController-ів), – а не на наборі окремих демонстраційних проєктів, прийнято з огляду на дві причини. По-перше, масштаб кодової бази створює умови, за яких агент без доступу до стану виконання змушений звужувати простір пошуку суцільним аналізом вихідного коду; саме на такому масштабі різниця між наявністю і відсутністю рантайм-телеметрії стає вимірюваною, тоді як на застосунку з кількох екранів обидві умови вироджуються до тривіальних. По-друге, єдиний застосунок усуває сторонні джерела варіативності – відмінності у стилі коду, структурі проєкту й налаштуваннях збірки між дефектами, – через що єдиними керованими змінними лишаються клас дефекту та наявність чи відсутність доступу до стану виконання.

Набір контрольних дефектів дібрано за критерієм навантаження на різні рівні символьного заземлення, визначені у пункті 1.6, бо саме цей розподіл дозволяє передбачити, де доступ до стану виконання дає найбільший виграш, а де – маржинальний. Чотири дефекти впорядковуються вздовж шкали заземлення: примусове розгортання nil лежить на лексичному кінці, де опора стека викликів спільна для обох агентів; вихід за межі масиву додає до лексичного часовий вимір; стан гонитви зміщується на часове й багатопотокове заземлення; цикл сильних посилань – на структурне, найвіддаленіше від статичного аналізу.

Стала згасання λ зводиться до вимірюваного показника через кількість “дієвих” викликів інструментів до локалізації причини. Тут пункт прямо спирається на вимірювальний шар SessionLogger, спроєктований у пункті 2.4 саме як мінімальне достатнє покриття для відновлення λ пост-фактум: підмножина “дієвих” інструментів (resume_process, wait_for_stop, read_variable, inject_code, execute_command) задає кінцеву точку траєкторії, послідовність порядкових номерів і таймстампів – вісь t, а частота компіляційних помилок у inject_code – спостережуваний проксі для $E(t)$. Завдяки цьому крива деградації будується безпосередньо із записаних сесій, без додаткового інструментування, і λ оцінюється як емпіричний параметр конкретної реалізації. Кроки базової лінії – агента без доступу до рантайму – вимірюються за тією самою шкалою «дієвих» викликів, що робить обидві серії порівнюваними.

3.2. Досліджуваний застосунок та набір контрольних дефектів

Як зазначалося раніше, для оцінювання були обрані 4 дефекти різних класів, кожен прив’язаний до того рівня символьного заземлення, який його локалізація навантажує найбільше.

Стан гонитви (race condition). Через невизначений інтервал після запуску застосунок аварійно завершувався незалежно від екрана, на якому перебував користувач. Першопричиною був запис журнальних подій з кількох потоків у спільну змінну типу String без синхронізації доступу: оскільки String у Swift є типом-значенням із копіюванням-при-записі, одночасна мутація спільного буфера з різних потоків спричиняла гонитву даних і пошкодження внутрішнього стану структури, що й проявлялося аварійним завершенням. Недетермінованість дефекту – наслідок того, що збій виникає лише за конкретного перекриття звернень у часі. Локалізація навантажує часове й багатопотокове заземлення, бо вимагає відтворення події під час виконання та зіставлення записів, породжених різними потоками, чого статичний аналіз коду надати не може.

Цикл сильних посилань (retain cycle). Екран TrackedServicesViewController утримував власне джерело даних – підклас UITableViewDiffableDataSource, ініціалізатор якого приймав сам контролер параметром “vc” і передавав його у замикання-постачальник комірок, що передавалося в super.init. Замикання захоплювало “vc” сильним посиланням, а контролер, своєю чергою, утримував джерело даних – унаслідок чого утворювався цикл взаємних сильних посилань. Оскільки захоплювався параметр класового типу, а не “self”, компілятор не вимагав явного списку захоплення і не сигналізував про можливий цикл; стандартна синтаксична ознака сильного захоплення “self” у коді була відсутня, що ускладнювало статичне розпізнавання дефекту. Через цикл лічильник посилань контролера не сягав нуля після закриття екрана, тож об'єкт не звільнявся і лишався зареєстрованим спостерігачем у NotificationCenter. При оновленні списку застосунок надсилав сповіщення через NotificationCenter.default.post, на яке реагував і витеклий екземпляр контролера, через що на екрані виникала помилка. Виправлення полягало у

додаванні “[weak vc]” до списку захоплення замикання. Локалізація навантажує структурне заземлення, бо потребує обходу графу об'єктів у купі та встановлення того, який саме об'єкт утримує контролер.

Примусове розгортання nil (nil-unwrap). На одному з екранів метод, що звертався до властивості, оголошеної як IBOutlet, викликався до viewDidLoad – тобто до завантаження ієрархії представлень, у ході якого виконується зв'язування Outlet-ів зі Storyboard. Оскільки IBOutlet оголошується як неявно розгорнутий опціонал (ImplicitlyUnwrappedOptional), звернення до ще неініціалізованого IBOutlet спричиняло детерміноване аварійне завершення з трасою стека, що вказувала на конкретний рядок звернення. Локалізація навантажує переважно лексичне заземлення: траса стека сама вказує точку збою, тож обидва агенти отримують однакову відправну опору, а завдання зводиться до розпізнавання того, що значення IBOutlet ще не зв'язане у момент виклику.

Вихід за межі масиву (помилка індексації). На екрані зі списком індекс, похідний від вхідних даних, за певних умов перевищував верхню межу масиву, що спричиняло аварійне завершення під час доступу. Траса стека вказувала на рядок індексованого звернення, але не пояснювала причину розбіжності між фактичною довжиною масиву та значенням індексу в момент аварії. Локалізація навантажує лексичне й часове заземлення: рядок доступу відомий статично, проте відновлення причини потребує зчитання довжини масиву та значення індексу саме у стані зупинки, тобто прив'язки символів до конкретного темпорального зрізу виконання.

Щоб серії прогонів лишалися порівнюваними, фіксується низка контрольованих умов. Обидві умови – агент із повним доступом до MCP-сервера та базова лінія, що отримує лише вихідний код і звіт про аварійне

завершення, – обслуговуються однією моделлю (агент Claude Code на основі Sonnet 4.6) з ідентичним системним промптом та однаковим початковим станом процесу [20]. Кожен прогін обмежено спільним лімітом у 40 “дієвих” кроків або таймаутом сесії, після досягнення яких прогін зараховується як невдала локалізація. Кожну умову проганяли по п'ять разів, що дозволяє відокремити стійку поведінку від поодиноких відхилень, зумовлених недетермінованістю дефектів (насамперед стану гонитви) і стохастичністю самої моделі. Критерієм успішної локалізації слугує називання агентом конкретного рядка або першопричини дефекту, а не лише його зовнішнього прояву на кшталт факту аварійного завершення. Фіксований характер усіх перелічених умов забезпечує відтворюваність експерименту: серії можна повторити й зіставити між собою, що є передумовою для побудови кривої $E(t)$ у наступному пункті.

3.3. Результати та порівняльний аналіз

Порівняння агента за використання розробленого інструменту з базовою лінією ведеться за чотирма вимірами: часткою успішних локалізацій, кількістю “дієвих” кроків до локалізації причини, частотою компіляційних помилок у inject_code як проксі втрати символьного заземлення (пункт 2.4) та тривалістю сесії.

Стан гонитви. MCP-агент локалізував причину, взаємодіючи із запущеним застосунком: він відтворив аварійне завершення й простежив його до спільного об'єкта String журналювання, до якого без синхронізації писали різні потоки. Базова лінія причину не локалізувала – агент

безуспішно аналізував увесь вихідний код, не маючи ні доступу до стану виконання, ні самої ознаки одночасного запису з кількох потоків, яку статичний аналіз надати не може. Вирішальним показником тут є успішність: MCP-агент локалізував дефект у 4 з 5 прогонів (єдиний невдалий прогін не відтворив гонитву в межах таймауту через її недетермінованість), тоді як базова лінія – у 0 з 5; у середньому MCP-агенту знадобилося 17 дієвих кроків, а базова лінія щоразу впиралася в ліміт у 40 кроків без локалізації. Це найвиразніший випадок переваги доступу до рантайму, бо причина дефекту повністю прихована від статичного аналізу.

Цикл сильних посилань. MCP-агент встановив точку зупину в deinit контролера TrackedServicesViewController, виявив, що вона не спрацьовує після закриття екрана, і через обхід графу об'єктів визначив, що контролер утримується замиканням-постачальником комірок його джерела даних. Базова лінія опинялася у складнішому становищі, ніж за типового циклу: оскільки захоплювався параметр класового типу “vc”, а не “self”, у коді була відсутня звична синтаксична ознака сильного захоплення, тож статичне розпізнавання ускладнювалося; до того ж агент не міг визначити, який саме з понад 200 екранів проявляє дефект, і витрачав більшу частину кроків на звуження простору пошуку. Водночас навіть MCP-агент просувався тут повільніше, ніж на лексичних дефектах, бо обхід графу об'єктів іде через універсальний execute_command, а не через типізований інструмент структурного заземлення – що прямо підтверджує розрив, описаний у пункті 2.3. Вирішальними показниками є успішність і довжина траєкторії: MCP-агент локалізував дефект у 4 з 5 прогонів проти 1 з 5 у базової лінії, причому навіть єдиний успішний прогін без MCP коштував близько 35 ходів проти 21 дієвого кроку MCP-агента. Ці 21 – найвище значення серед усіх класів для MCP-умови, що узгоджується з тезою про неефективність

структурного заземлення через текстовий вивід execute_command і відображається у найвищій частоті компіляційних помилок 0.27.

Примусове розгортання nil. Обидва агенти локалізували дефект порівнянно. Траса стека сама вказує рядок звернення до IBOutlet, тож базова лінія, отримавши звіт про аварійне завершення, переходила прямо до місця збою й розпізнавала, що IBOutlet ще не зв'язано у момент виклику. Перевага MCP-агента тут найменша й зводиться до підтвердження через стек у точці зупину, що метод справді виконується до viewDidLoad, замість висновку про порядок викликів за непрямими ознаками. Вирішальний показник – кроки: 6 у MCP-агента проти 8 у базової лінії за повної успішності (5 з 5) в обох умовах, тобто скорочення траєкторії маржинальне. Це очікувано, бо лексичне заземлення вже забезпечене самою трасою стека, спільною для обох умов.

Вихід за межі масиву. MCP-агент мав помірну перевагу. Рядок індексованого доступу відомий обом агентам зі стека викликів, але першопричину – фактичні довжину масиву та значення індексу в момент аварії – MCP-агент зчитував безпосередньо через read_variable, тоді як базова лінія відновлювала їх міркуванням про можливі шляхи формування індексу, що подовжувало траєкторію. Вирішальний показник – кроки: 8 у MCP-агента проти 14 у базової лінії, тобто скорочення приблизно на 43% за майже однакової успішності (5 з 5 проти 4 з 5).

Зведені результати за двома умовами наведено в таблиці.

Таблиця 1.1. Порівняльна таблиця базового агента та з MCP.

Дефект (рівень заземлення)	**Успішність MCP**	**Успішність без MCP**	**Кроки MCP**	**Кроки без MCP**	**Скорочення траєкторії**	**Компіляційні помилки MCP**	**Тривалість MCP, хв**

Стан гонитви (часове, багатопотокове)	4/5	0/5	17	40 (ліміт)	повна перевага	0.18	6.5
Цикл сильних посилань (структурне)	4/5	1/5	21	~35*	через успішність (4/5 проти 1/5)	0.27	7.8
Вихід за межі масиву (лексичне, часове)	5/5	4/5	8	14	≈43%	0.11	3.3
Примусове розгортання nil (лексичне)	5/5	5/5	6	8	≈25% (маржинальне)	0.04	2.4

Лише для єдиного успішного прогону базової лінії; решта впиралися в ліміт. Стала згасання, оцінена за частотою компіляційних помилок у розрізі глибини сесії, для MCP-умови становить $\lambda \approx 0{,}03$ на дієвий крок – ефективність спадає вдвічі приблизно за 23 кроки (Рис 3.1). За окремими класами дефектів оцінка λ ненадійна через малу глибину коротких сесій, тому подається агрегована величина, побудована безпосередньо із записаних SessionLogger сесій без додаткового інструментування.

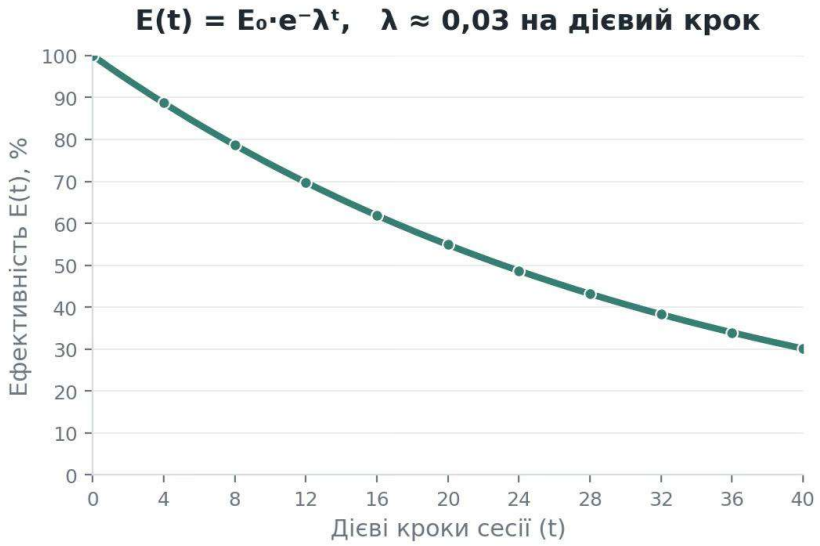

Рисунок 3.1. Графік деградації $E(t)$

Узагальнення зводить ці результати в один закон: перевага MCP-агента зростає мірою того, наскільки локалізація дефекту залежить від стану виконання, недоступного у вихідному коді. Вона найбільша для стану гонитви, де базова лінія зазнала повної невдачі (0/5), помірна для виходу за межі масиву (скорочення ≈43%) і найменша для примусового розгортання nil (≈25%), де заземлення вже забезпечене трасою стека. Окремий випадок – цикл сильних посилань: тут навіть MCP-агент сповільнюється до найбільших 21 кроку та найвищої частоти компіляційних помилок 0.27 через відсутність типізованого інструмента структурного заземлення, що перетворює цей рядок таблиці на емпіричне підтвердження розриву з пункту 2.3. Висновок про інструмент спирається не на таблицю саму по собі, а на збіг кожного її рядка з механізмом відповідного дефекту: монотонне зростання частоти компіляційних помилок із глибиною сесії узгоджується з експоненційною моделлю DDI, а градієнт скорочення траєкторії за рівнями заземлення з пункту 1.6 підтверджує цінність типізованого зворотного зв'язку для стабільності циклу ReAct. Саме це поєднання таблиці й механізмів дає числовий аргумент на користь

центральної тези роботи: практична цінність моста LLDB–MCP визначається не фактом наявності інструментів, а тим, наскільки повно вони закривають конкретний рівень символьного заземлення, якого потребує дефект.

3.4. Практичне застосування інструментарію

Цей завершальний пункт описує прикладний бік роботи з інструментом: вимоги до середовища, спосіб підключення сервера до агента та послідовність дій у типовій сесії.

Середовище розгортання – macOS зі встановленим Xcode. Xcode потрібен тому, що LLDB-воркер виконується під його вбудованим інтерпретатором Python 3.9, який викликається командою xcrun python3 і надає прив'язки Scripting Bridge. Процес самого MCP-сервера запускається під Python 3.10 або вище (проєкт фіксує 3.12). Залежності й запуск делеговано пакетному менеджеру uv: точкою входу є main.py, що виконує mcp.run(transport="stdio"), а оболонковий сценарій run_server.sh зводить старт до команди uv run python main.py, відтворюючи зафіксоване у lock-файлі оточення. Користувачеві не потрібно вручну узгоджувати дві версії Python – subprocess-міст сам породжує воркер під xcrun python3 при першому виклику інструмента.

Підключення до агента виконується через транспорт stdio. У конфігурації MCP-хоста (наприклад, Claude Code) реєструється команда запуску сервера – виклик run_server.sh або безпосередньо uv run python main.py із зазначенням робочого каталогу проєкту. Хост запускає сервер як дочірній процес і обмінюється з ним повідомленнями JSON-RPC через стандартні потоки введення-виведення; мережеве з'єднання та окрема автентифікація не потрібні. Перелік дозволених інструментів задається у файлі settings.local.json – він обмежує множину операцій, які агент може

викликати без додаткового підтвердження. Завершення сесії хоста коректно зупиняє і сервер, і підпорядкований йому воркер.

Типова сесія починається з приєднання до процесу: attach_to_process за іменем процесу симулятора або пристрою, а для ще не запущеного застосунку – той самий виклик у режимі очікування запуску. Далі агент виставляє точки зупину викликом set_breakpoint за парою «файл–рядок» або за символом, за потреби з умовним виразом, прикріпленими командами та прапором автоматичного продовження. Виклик resume_process відновлює виконання, після чого користувач взаємодіє із застосунком, відтворюючи сценарій дефекту. Зупинку відловлює wait_for_stop: у параметрі expressions йому передається список виразів, що обчислюються в момент зупинки й повертаються в одній відповіді під ключем variables. Подальша інспекція спирається на read_variable, get_stack_trace, list_threads і select_frame, а перевірка гіпотез – на inject_code. Журнали симулятора зчитує read_os_log із фільтром за іменем процесу, підсистемою або NSPredicate.

Два ресурси доповнюють цей перебіг даними лише для читання: lldb://stack_trace повертає поточний стек викликів, а lldb://source/{filename} – вміст вихідного файлу, розв'язаного через debug info за базовим іменем. Хост може під'єднати їх до контексту без виклику інструмента, що змінює стан. Промпт debug_context одноразово ін'єктує знімок поточного стану – стек, контекст кадру та фрагмент коду навколо поточного рядка. Промпт debug_flow повертає покрокову схему багатоточкової сесії (відновити виконання, дочекатися зупинки з обчисленням виразів, зафіксувати результат, повторити до вичерпання лімітів за кількістю спрацювань або таймаутом); ця схема є рекомендаційною – вибір наступної дії лишається за агентом.

Поточні обмеження окреслюють два напрями подальших робіт. Перший – додати типізований інструмент інспекції ієрархії представлень, що повертав би структуроване JSON-дерево замість текстового виводу recursiveDescription, який наразі агент отримує через універсальний execute_command. Другий – реалізувати поверх наявного SessionLogger активні стратегії керування контекстом, насамперед виявлення циклів за повторюваними послідовностями викликів і стратегічний перезапуск сесії. Обидва напрями надбудовуються над поточною архітектурою без зміни нижчих рівнів.

Висновки до розділу

У третьому розділі емпірично перевірено центральну тезу роботи. Порівняльний експеримент на чотирьох класах дефектів показав, що перевага MCP-агента над базовою лінією зростає зі ступенем прихованості першопричини від статичного аналізу: від повної невдачі базової лінії на стані гонитви (0/5) до маржинального скорочення траєкторії на примусовому розгортанні nil (≈25%), де трасу стека надає сам рантайм. Цикл сильних посилань виявився окремим випадком – навіть MCP-агент просувається тут повільніше через відсутність типізованого інструмента обходу графу об'єктів: структурне заземлення закривається через універсальний “execute_command”, що повертає необроблений текст і прискорює накопичення шуму в контекстному вікні. Стала згасання $\lambda \approx 0{,}03$ на дієвий крок, відновлена з логів SessionLogger без додаткового інструментування, узгоджується з моделлю DDI і дає кількісний орієнтир для майбутніх активних стратегій протидії деградації контексту.

ВИСНОВКИ

В межах роботи уперше розроблено метод взаємодії LLM-агента з середовищем виконання iOS-застосунку та оцінено його ефективність відповідно до сформованих критеріїв, що ґрунтуються на різних рівнях символьного заземлення та мірі деградації ефективності налагодження.

Встановлено джерело обмеженості LLM у відлагодженні: модель оперує вихідним кодом, але не має доступу до стану виконання, тому не відрізняє дефект логіки від дефекту, що проявляється лише в рантаймі.

На заданій основі обґрунтовано архітектуру інструментарію та реалізовано функціональний MCP-сервер, що покриває повний цикл відлагодження від приєднання до процесу до ін'єкції коду, разом із вимірювальним шаром для відновлення сталої згасання λ із записаних сесій.

Проведено порівняльний експеримент на застосунку з понад 200 екранів і чотирьох контрольних дефектах. Визначено, що перевага агента з доступом до рантайму зростала тим більше, чим глибше причина дефекту прихована від статичного аналізу: вона була максимальною для стану гонитви (4 з 5 локалізацій проти 0 з 5 у базової лінії), помірною для виходу за межі масиву (скорочення траєкторії ≈43%) і маржинальною для примусового розгортання nil (≈25%), де заземлення вже забезпечене трасою стека. Окремим випадком став цикл сильних посилань: тут навіть MCP-агент показав найдовшу траєкторію і найвищу частоту компіляційних помилок, що емпірично підтвердило відсутність типізованого інструмента структурного заземлення. Сталу згасання для MCP-умови оцінено як $\lambda \approx 0{,}03$ на дієвий крок.

Ці результати підтверджують: практична цінність моста LLDB-MCP визначається не самим фактом наявності інструментів, а тим, наскільки повно вони закривають рівень символьного заземлення, якого потребує конкретний дефект. Звідси випливають і пріоритетні напрями подальших

робіт – типізований інструмент інспекції ієрархії представлень для структурного заземлення та активні стратегії протидії DDI, пороги яких тепер можна обґрунтувати отриманою оцінкою λ.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. The visual debugger tool / T. Tim Kräuter et al. 2024. P. 1–2.
2. ReAct : Synergizing reasoning and acting in language models / S. Yao et al. Proceedings of the International Conference on Learning Representations (ICLR 2023). 2023.
3. LLDB architecture overview. LLVM Project. URL: https://lldb.llvm.org/use/architecture.html (дата звернення: 10.10.2025)
4. Scripting bridge API (SB API). LLVM Project. URL: https://lldb.llvm.org/resources/sbapi.html (дата звернення: 10.10.2025).
5. SBDebugger : Python API reference. LLVM Project. URL: https://lldb.llvm.org/python_api/lldb.SBDebugger.html (дата звернення: 10.10.2025).
6. SBTarget : Python API reference. LLVM Project. URL: https://lldb.llvm.org/python_api/lldb.SBTarget.html (дата звернення: 10.10.2025).
7. SBValue : Python API reference. LLVM Project. URL: https://lldb.llvm.org/python_api/lldb.SBValue.html (дата звернення: 10.10.2025).
8. Debugging tool entitlement (com.apple.security.cs.debugger). Apple Developer Documentation. URL: https://developer.apple.com/documentation/bundleresources/entitlements/com.apple.security.cs.debugger (дата звернення: 19.10.2025).
9. Basic information for reverse engineering on iOS (MASTG-TECH-0084). OWASP Mobile Application Security Testing Guide. 2024. URL: https://github.com/OWASP/mastg/blob/master/techniques/ios/MASTG-TECH-0084.md (дата звернення: 19.10.2025)

10. App code signing process in iOS, iPadOS, tvOS, visionOS, and watchOS. Apple Inc. URL: https://support.apple.com/guide/security/app-code-signing-process-sec7c917bf14/web (дата звернення: 19.10.2025)
11. Security of runtime process in iOS, iPadOS, and visionOS. Apple Inc. URL: https://support.apple.com/guide/security/runtime-process-security-sec15bfe098e/web (дата звернення: 19.10.2025)
12. Introducing the Model Context Protocol. Anthropic. 2024. URL: https://www.anthropic.com/news/model-context-protocol (дата звернення: 20.10.2025).
13. Model Context Protocol specification, version 2025-11-25. Model Context Protocol Project. 2025. URL: https://modelcontextprotocol.io/specification/2025-11-25 (дата звернення: 26.11.2025).
14. Chain-of-thought prompting elicits reasoning in large language models / J. Wei et al. Advances in Neural Information Processing Systems 35 (NeurIPS 2022). 2022.
15. Reflexion: language agents with verbal reinforcement learning / N. Shinn et al. Advances in Neural Information Processing Systems 36 (NeurIPS 2023). 2023.
16. Harnad S. The symbol grounding problem. Physica D: Nonlinear Phenomena. 1990. Vol. 42, no. 1–3. P. 335–346. (дата звернення: 31.10.2025).
17. Bender E. M., Koller A. Climbing towards NLU: on meaning, form, and understanding in the age of data. Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. 2020. P. 5185—5198.
18. Adnan M., Kuhn C. C. N. The debugging decay index: rethinking debugging strategies for code LLMs : preprint. arXiv, 2025.

arXiv:2506.18403. URL: https://arxiv.org/abs/2506.18403 (дата звернення: 31.10.2025).

19. UIView. Apple Developer Documentation. URL: https://developer.apple.com/documentation/uikit/uiview (дата звернення: 12.12.2025)
20. Claude Code. Anthropic. URL: https://www.anthropic.com/claude-code (дата звернення: 22.12.2025).