https://doi.org/10.31891/2219-9365-2026-85-20
УДК 004.8:004.94:004.62

МУСІЄНКО Андрій
Національний технічний університет України "Київський політехнічний інститут імені Ігоря Сікорського"
https://orcid.org/0000-0002-1849-6716
e-mail: mysienkoandrey@gmail.com
ВОРВУЛЬ Данило
Національний технічний університет України "Київський політехнічний інститут імені Ігоря Сікорського"
https://orcid.org/0009-0009-5822-9063
e-mail: vorvul.danylo@gmail.com

ГРАФОВА АРХІТЕКТУРА ПАМ'ЯТІ ДЛЯ ЕФЕКТИВНОГО ВИКОРИСТАННЯ КОМП'ЮТЕРНИХ АГЕНТІВ НА ОСНОВІ ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ

Агенти, керовані великими мовними моделями (Large Language Model, LLM), які здійснюють взаємодію з комп'ютером (Computer Use Agents, CUA), часто нераціонально витрачають обчислювальні ресурси, повторно виконуючи міркування щодо завдань, які вже були розв'язані раніше. У цій роботі запропоновано усунути таку неефективність шляхом впровадження графової архітектури пам'яті для автоматизації графічного інтерфейсу користувача (GUI). Запропонований підхід передбачає збереження агентом своїх траєкторій взаємодії у динамічному графі, вузли якого відображають екрани застосунків, а ребра кодують послідовності дій, що ведуть до переходів між станами. Повторне використання цього графа попереднього досвіду дає змогу агентові відтворювати як низькорівневі дії, так і високорівневі робочі процеси без необхідності повторного обчислення з нуля. Запропоновану архітектуру пам'яті реалізовано шляхом розширення сучасного агента CUA (Agent S3) за допомогою створеного модуля пам'яті. Експерименти на еталонному наборі OSWorld продемонстрували, що запропонований метод скорочує споживання токенів LLM і час виконання приблизно на 50 % порівняно з базовою моделлю без пам'яті, не знижуючи рівня успішності виконання завдань. Графова пам'ять забезпечує ефективне відтворення точних маніпуляцій з інтерфейсом користувача та дає змогу агенту міркувати над абстрактними завданнями (на приклад «вхід у систему» чи «експорт звіту») як над придатними до повторного використання підпроцедурами. Отримані результати свідчать, що структурована пам'ять суттєво підвищує практичну ефективність агентів на основі LLM у контексті виконання повторюваних завдань реального світу.

Ключові слова: великі мовні моделі; агенти комп'ютерної взаємодії; графова пам'ять; автоматизація GUI; повторне використання знань; ефективність виконання завдань; Agent S3; OSWorld.

MUSIENKO Andrii, VORVUL Danylo
National Technical University of Ukraine "Igor Sikorsky Polytechnic Institute"

GRAPH-BASED MEMORY ARCHITECTURE FOR EFFICIENT LLM-BASED COMPUTER USE AGENTS

Large language model (LLM)-driven Computer Use Agents (CUAs) frequently incur substantial computational overhead because they repeatedly reason through tasks that have already been solved in prior interactions. This redundancy leads to excessive token consumption, increased latency, and higher operational costs, which limits the scalability of LLM-based GUI automation in real-world environments. To address this inefficiency, we propose a graph-based persistent memory architecture that enables agents to store, retrieve, and reuse previously executed interaction trajectories. In the proposed approach, the agent maintains a dynamic directed graph in which nodes correspond to observed application states or screens, while edges encode executable action sequences (e.g., GUI scripts, command chains, or tool invocations) that produce deterministic transitions between states.

Such a representation allows the agent to shift from purely deliberative reasoning toward experience-driven behavior, reusing both low-level motor actions (clicks, text entry, menu navigation) and higher-level procedural abstractions (e.g., "authenticate user," "configure settings," "export report"). Instead of recomputing action plans from scratch, the agent performs graph retrieval and subpath matching to identify previously successful workflows, which can be adapted to the current context with minimal additional reasoning. The memory layer supports hierarchical reuse, enabling composition of complex tasks from smaller verified subroutines and facilitating generalization across similar interfaces.

We integrate the proposed memory mechanism into a state-of-the-art CUA framework (Agent S3) by introducing a persistent storage module, a state hashing scheme for robust screen identity matching, and a retrieval policy that balances exploration and exploitation. The system also incorporates mechanisms for conflict resolution, memory pruning, and versioning to prevent accumulation of stale or invalid interaction paths when interfaces evolve.

Experimental evaluation on the OSWorld benchmark demonstrates that the memory-augmented agent achieves a 50–60% reduction in LLM token usage and overall execution time relative to a memoryless baseline, while maintaining comparable task success rates. Additional analysis shows improved performance on repetitive and multi-step workflows, reduced planning variance, and faster convergence to optimal action sequences.

The results indicate that structured, graph-based memory transforms CUAs from stateless planners into experience-aware systems capable of recalling exact UI manipulations and reasoning over abstract task schemas. This substantially improves efficiency, reliability, and cost-effectiveness in long-horizon and repetitive automation scenarios. The proposed approach highlights the importance of persistent, structured memory for practical deployment of LLM-driven agents in enterprise software, digital workflows, and human-in-the-loop environments.

Keywords: LLMs; Computer Use Agents; Graph Memory; GUI Automation; Knowledge Reuse; Task Efficiency; Agent S2; OSWorld.

ISSN 2219-9365

Стаття надійшла до редакції / Received 23.12.2025
Прийнята до друку / Accepted 19.02.2026
Опубліковано / Published 05.03.2026

ПОСТАНОВКА ПРОБЛЕМИ У ЗАГАЛЬНОМУ ВИГЛЯДІ ТА ЇЇ ЗВ'ЯЗОК ІЗ ВАЖЛИВИМИ НАУКОВИМИ ЧИ ПРАКТИЧНИМИ ЗАВДАННЯМИ

Агенти на основі великих мовних моделей (LLM) для роботи з комп'ютером (Computer Use Agents, CUA) покликані автоматизувати складні цифрові завдання на настільних комп'ютерах, мобільних пристроях і в інтернет-середовищі шляхом інтерпретації інструкцій природною мовою та керування графічними інтерфейсами користувача за допомогою низькорівневих дій [1]. Попри значний прогрес, сучасні CUA залишаються неефективними для повторюваних робочих процесів, що обмежує їх придатність для повсякденного використання [1]. Основна проблема полягає у відсутності в цих агентів довгострокової пам'яті про виконані завдання. Внаслідок цього вони часто «винаходять колесо заново» навіть у разі рутинних процедур — повторно планують або генерують послідовності дій, які вже були раніше розв'язані, що призводить до зайвих витрат токенів і часу.

Багато завдань мають спільні підзадачі, проте без пам'яті агент змушений щоразу заново продходити ці кроки для кожного нового завдання [2]. Крім того, агенти, що покладаються на методи логічного ланцюжка міркувань (*chain-of-thought prompting*), схильні розмірковувати над кожним натисканням клавіші чи кліком миші навіть для простих операцій, що призводить до непотрібних затримок [3]. Нездатність згадувати та повторно використовувати попередні рішення є суттєвою перешкодою для практичного впровадження, оскільки реальні робочі процеси характеризуються високою повторюваністю та рідко потребують повторного міркування. Останні оглядові дослідження у цій галузі підкреслюють неефективність навчання як критичну проблему та закликають до впровадження «адаптивного навчання, що виходить за межі статичного промптингу», аби узгодити проєктування агентів із вимогами реального використання [1].

Спроби запровадити пам'ять у GUI-агентів досі були обмеженими. Система **MobileGPT** доповнює мобільного автоматора завдань на основі LLM пам'яттю, подібною до людської, зберігаючи кожне вивчене завдання як послідовність придатних до повторного використання підзадач [2]. Така ієрархічна пам'ять дає змогу агентові відтворювати раніше виконані процедури без необхідності виклику LLM для кожного окремого кроку. MobileGPT продемонструвала потенціал пам'яті, скоротивши час виконання завдань і витрати на використання LLM приблизно на 70 % порівняно з безпам'ятною базовою моделлю [2]. Проте цей підхід фактично трактує пам'ять як журнал минулих дій і орієнтований переважно на мобільні застосунки, не формуючи узагальненого структурованого представлення станів GUI.

Агент **AppAgentX** натомість запроваджує пам'ять траєкторій завдань для формування високорівневих макродій на основі повторюваних низькорівневих операцій [3]. Аналізуючи власну історію виконання, AppAgentX здатен об'єднувати часті послідовності дій у нові складені команди (наприклад, створюючи єдину команду «Пошук», що замінює послідовність введення тексту та натискань) [3]. Це підвищує ефективність, усуваючи потребу в покроковому міркуванні для рутинних завдань. Водночас пам'ять AppAgentX структурована як лінійний ланцюг дій і не моделює розгалужену структуру навігації графічного інтерфейсу.

Крім того, жоден із цих підходів не був продемонстрований у повноцінному середовищі настільного комп'ютера з довільними застосунками та керуванням на основі коду (наприклад, генерування скриптів Python *pyautogui* для імітації дій миші та клавіатури). Найсучасніша на сьогодні система **Agent S2** зосереджується на вдосконаленні сприйняття та планування, розподіляючи функції між менеджером і виконавцем (моделями-генералістом і спеціалістом) і впроваджуючи методи, як-от *Mixture-of-Grounding*, для точного локалізування елементів інтерфейсу [4]. Хоча Agent S2 демонструє провідні результати на бенчмарках, зокрема OSWorld [4], вона не має модуля довгострокової пам'яті — кожне завдання розглядається як нове, навіть якщо агент уже стикався з ним раніше.

Отже, попередні підходи або лише зберігають історії взаємодій, або створюють макроси як скорочення, однак жоден не пропонує гнучкої багаторівневої пам'яті станів і дій GUI. Очевидною залишається потреба в архітектурі пам'яті, здатній підтримувати як спостереження на основі знімків екрана, так і дії на основі коду, що дало б змогу агентові повторно використовувати набуті знання як на детальному рівні маніпуляцій із користувацьким інтерфейсом, так і на вищому рівні семантичних послідовностей завдань.

Внесок: У цій роботі ми пропонуємо нову архітектуру пам'яті на основі графів для агентів графічного інтерфейсу користувача (GUI), керованих великими мовними моделями (LLM). На відміну від підходів, що трактують пам'ять як плоску послідовність історій або скриптів, ми структуруємо досвід агента у вигляді графа станів GUI. Вузли графа відповідають окремим екранам або сторінкам застосунку (ідентифікованим за їхнім візуальним оформленням або унікальним станом інтерфейсу). Орієнтовані ребра між вузлами репрезентують виконувані послідовності дій, які ініціюють переходи між станами, — наприклад, ребро може містити код, що описує послідовність кліків миші та натискань клавіш (згенерований скрипт *pyautogui*), необхідну для переходу від одного екрана до іншого.

Крім того, до вузлів додаються параметризовані функції завдань, які репрезентують підзадачі, що можуть бути виконані на відповідному екрані. Наприклад, для вузла, що відповідає головній сторінці хмарного сховища, агент може засвоїти функцію *SearchDrive(query)* — параметризовану послідовність дій (ребро), яка вводить запит у поле пошуку та натискає кнопку пошуку, спричиняючи перехід до сторінки результатів. У процесі виконання нових завдань і дослідження нових ділянок інтерфейсу граф пам'яті поступово зростає та еволюціонує, формуючи багаторівневу структуру знань.

Ключова перевага такого підходу полягає у двох рівнях пам'яттєво-керованої поведінки:

1. **Низькорівневе відтворення:** агент може відновлювати точні послідовності дій (ребра), вже використані раніше, для повторного виконання інтеракцій (наприклад, проходження певного меню) без повторного залучення LLM для міркування над кожним кроком.
2. **Високорівневе узагальнення:** агент здатен розпізнавати та викликати складніші процедури (підграфи пам'яті), такі як «вхід у систему» чи «експорт звіту», розглядаючи їх як єдині операційні блоки.

Завдяки використанню графової пам'яті CUA суттєво підвищує ефективність оброблення повторюваних завдань, уникаючи зайвих обчислень, і водночас зберігає гнучкість для розв'язання нових задач за допомогою традиційного міркування LLM, коли це потрібно.

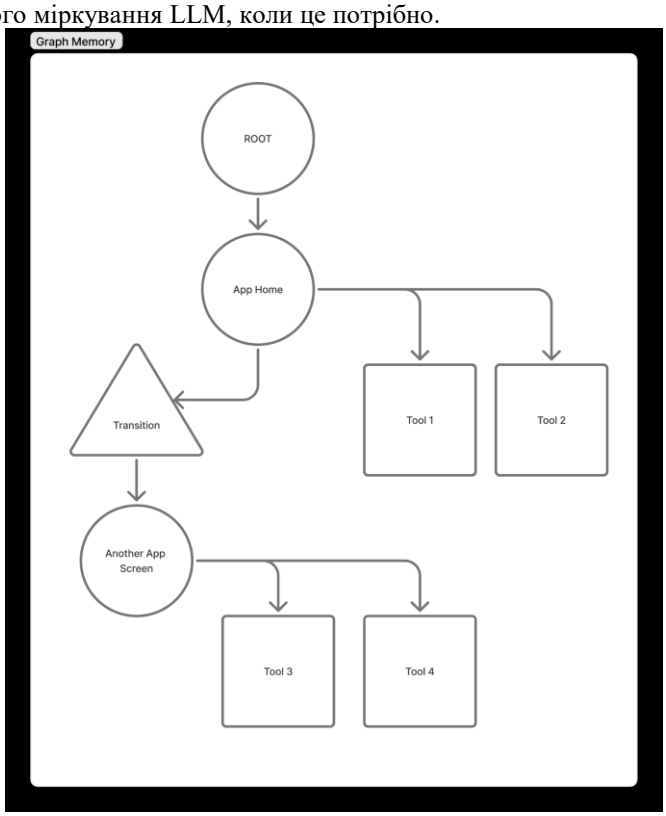

Рис 1. Граф пам'яті накопичує знання, отримані під час виконання завдань

АНАЛІЗ ЛІТЕРАТУРНИХ ДАНИХ ТА ПОСТАНОВКА ПРОБЛЕМИ

Останні роботи все частіше розглядають механізми пам'яті в UI-агентах на основі LLM як ключ до підвищення ефективності повторюваних сценаріїв. Однією з ранніх систем із вираженою структурованою пам'яттю для мобільної автоматизації стала MobileGPT/MemoDroid [2]. Вона розкладає завдання на підзавдання та примітивні дії й зберігає їх в ієрархічній пам'яті, що дозволяє повторно викликати вже вивчені процедури без постійного залучення LLM на кожному кроці [2]. Завдяки цьому MobileGPT суттєво зменшила затримку виконання й вартість обчислень LLM [2]. Водночас така пам'ять переважно працює як відтворення збережених слідів взаємодій і була валідована у мобільному середовищі, тому її узагальнення на десктопні, більш різнорідні GUI-контексти залишається відкритим питанням [2].

Інший підхід демонструє AppAgentX, який накопичує траєкторії виконання та стискає часто повторювані послідовності дій у макродії, що повторно застосовуються як інструменти [3]. Механізм «еволюції» макрокоманд зменшує потребу в покроковому міркуванні і може підвищувати швидкість та стабільність виконання рутин [3]. Однак, попри використання графового сховища, знання здебільшого організовані як хронологічний ланцюг взаємодій [3], а макродії формуються під конкретні ситуації [3]. Через це слабко представлена розгалужена структура станів UI та обмежена параметризація дій поза контекстом їх виникнення [3]. На відміну від цього, організація досвіду як графа станів може напряму відображати

навігаційні розгалуження GUI та підтримувати повторне використання як локальних переходів, так і більших підпроцедур [3].

Паралельно розвиваються архітектурні інновації без довготривалої пам'яті. Agent S2 пропонує композиційну схему «генераліст–спеціаліст» для точнішого GUI-grounding і планування з великою часовою глибиною [4]. У ньому менеджер делегує підзадачі спеціалізованим модулям, а Mixture-of-Grounding та ієрархічне планування підвищують якість локалізації та виконання дій [4]. Це забезпечує високі результати на бенчмарках, зокрема OSWorld, але кожне завдання розглядається ізольовано, без перенесення рішень у майбутні запуски [4]. Оглядові роботи підкреслюють потребу в адаптивному навчанні понад статичний промптинг, що робить питання довготривалої пам'яті практично значущим [1]. Отже, наявні підходи або відтворюють журнали дій (MobileGPT), або формують макрокоманди (AppAgentX), або підсилюють сприйняття та планування без пам'яті (Agent S2) [4]; водночас бракує універсальної багаторівневої пам'яті, яка б у графовій формі відображала простір станів GUI та переходів між ними і підтримувала повторне використання знань у різних контекстах.

ФОРМУЛЮВАННЯ ЦІЛЕЙ СТАТТІ

Цілі дослідження

1. Розробити графову архітектуру довготривалої пам'яті для CUA, що зберігає стани GUI та переходи між ними у вигляді багаторазово придатних до використання знань.
2. Забезпечити повторне використання як низькорівневих послідовностей дій, так і високорівневих робочих процесів (підпроцедур) для скорочення витрат токенів і часу.
3. Емпірично перевірити ефективність запропонованого підходу у порівнянні з базовим агентом без пам'яті на стандартному бенчмарку.

Завдання дослідження

1. Спроєктувати структуру графа пам'яті (вузли-екрани, ребра-послідовності дій, описи/функції завдань) та механізм пошуку відповідних траєкторій/інструментів. Реалізувати інтеграцію графової пам'яті з CUA-архітектурою (на базі Agent S2/S3): цикл "пошук у пам'яті → виконання → оновлення пам'яті".
2. Розробити модуль генерації та параметризації інструментів (макродій) із трас взаємодій для повторного використання в майбутніх задачах.
3. Провести експериментальне оцінювання (OSWorld): порівняти витрати токенів, час виконання та успішність між baseline та memory-augmented агентом.

МАТЕРІАЛИ ТА МЕТОДИ ДОСЛІДЖЕННЯ

Було інтегровано графово структуровану пам'ять із фреймворком **Agent S2** для створення CUA-агента. Загальна архітектура зберігає ієрархію **Manager–Worker** та перцептивні модулі для інтерпретації графічного інтерфейсу користувача (GUI) [4], доповнюючись компонентом **Memory Graph** і модулем **Tool Generation**.

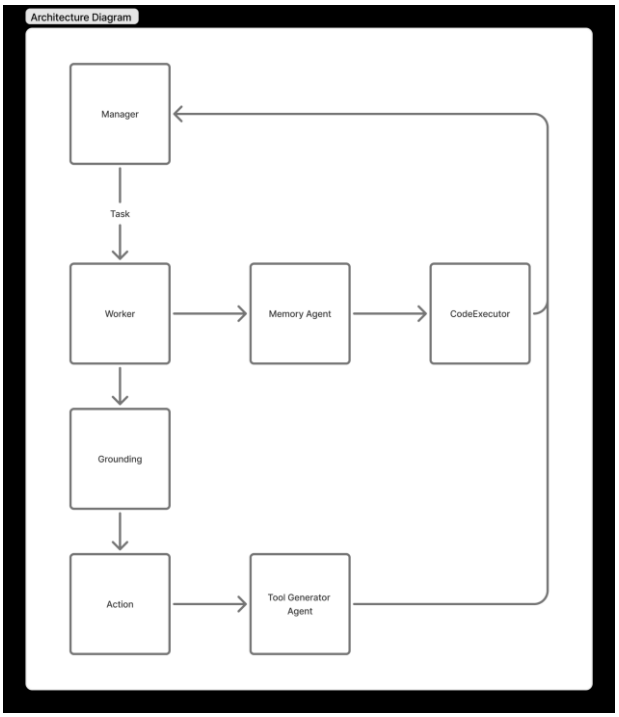

Рис. 2. Схема архітектури

Менеджер отримує інструкцію користувача та формує план завдання, який може складатися з кількох підзавдань або викликів інструментів. Воркер відповідає за виконання кожного кроку: сприйняття поточного стану GUI (за допомогою скріншотів і модулів OCR) і генерацію необхідної низькорівневої дії (наприклад, натискання кнопки або введення тексту). Ми модифікуємо цей цикл, вводячи етапи пошуку в пам'яті та виконання інструментів: перед тим як Менеджер призначить підзавдання Воркеру, він може здійснити запит до **Memory Graph**, щоб перевірити, чи існує відомий інструмент або траєкторія, здатна безпосередньо реалізувати підзавдання, оминаючи затратне LLM-міркування. Далі подано опис конструкції пам'яті та послідовності виконання.

Структура графа пам'яті.

Довготривала пам'ять агента реалізована у вигляді орієнтованого графа

$$G = (N, E, D) \tag{1}$$

де

- N - множина вузлів (екранів),
- $E \subseteq N \times N$ - множина орієнтованих ребер,
- D - множина описів завдань.

який зберігається в графовій базі даних. Початково G містить єдиний кореневий вузол, що відповідає головному екрану застосунку (або початковому стану, наприклад, робочому столу чи головному меню ОС). Кожен вузол репрезентує окремий екран або стан інтерфейсу програмного середовища.

$$n = (x, Tn, An) \tag{2}$$

де

x - візуальне представлення екрана (знімок, фічі),

Tn - набір завдань, які можуть виконуватись з цього стану,

An - доступні дії.

Вузол визначається змістом екрана: наприклад, різні сторінки багатокрокової форми або різні вікна програми розглядаються як окремі вузли. До кожного вузла приєднується набір **описів завдань (task descriptors)** — це високорівневі параметризовані операції, які агент навчився виконувати з цього екрана. Формально опис завдання є функціональним визначенням (макрокомандою), яку можна викликати з аргументами. Наприклад, у вузлі *"Google Drive Home"* може бути збережено інструмент SearchDrive(query), що інкапсулює послідовність дій для пошуку за довільним запитом. Кожне таке завдання пов'язане з ребром (або підграфом) у GGG, яке веде від поточного вузла до цільового (наприклад, сторінки результатів пошуку) при виконанні з певними параметрами.

Кожне ребро

$$e = n_i -> n_j \tag{3}$$

представляє дію або послідовність дій, що спричиняє перехід зі стану n_i у стан n_j. На практиці ребро містить фрагмент виконуваного коду (наприклад, Python-скрипт із використанням pyautogui), який реалізує відповідні кліки та натискання клавіш. Ребра марковані назвами дій або умовами (наприклад, *«натиснути кнопку 'Login'»*, *«ввести текст X і натиснути Search»*). Деякі ребра відображають елементарні кроки (одне натискання), тоді як інші охоплюють складні макродії (наприклад, відкриття меню та вибір пункту). Таким чином, граф пам'яті природно підтримує багаторівневу абстракцію: високорівневе завдання може бути представлено шляхом через кілька вузлів або окремим макроребром, якщо вся послідовність була абстрагована в один інструмент.

Генерація інструментів.

Ключовою новацією запропонованого підходу є автоматизований механізм абстрагування нових інструментів на основі досвіду. Після завершення нового завдання (особливо якщо воно містило тривалу послідовність низькорівневих дій) агент активує модуль **Tool Generation**, який аналізує трасу взаємодії. Модуль переглядає послідовність переходів між вузлами та виконаних дій, щоб виявити повторювані шаблони. Якщо послідовність має потенціал повторного використання, створюється нова параметризована функція.

Наприклад, якщо агент щойно навчився експортувати PDF-звіт у аналітичному застосунку через послідовність *File → Export → "PDF" → Save*, модуль генерує функцію

ExportReport(format)

для початкового вузла (головного інтерфейсу програми), яка при виклику з параметром format="PDF" відтворює відповідні дії. Цей інструмент додається до графа пам'яті: створюється нове ребро (або стискається наявний шлях) між початковим і кінцевим вузлами, позначене як ExportReport(format), із збереженням параметризованих дій. Наступного разу агент може виконати експорт без покрокового планування

З часом граф пам'яті еволюціонує: нові вузли додаються при зустрічі нових екранів, а ребра та інструменти — при засвоєнні нових дій або оптимізації існуючих. Періодично виконується технічне

обслуговування графа (не деталізується у цій роботі): злиття дубльованих вузлів, узагальнення параметрів (наприклад, заміна фіксованого імені файлу на параметр), а також вилучення малокорисних елементів пам'яті у разі змін інтерфейсу або низької частоти використання.

Послідовність виконання з пам'яттю.

Модифікований робочий цикл агента використовує граф пам'яті у такий спосіб (див. *Рис. 3 – блок-схема процесу*):

1. **Розпізнавання завдання.**

Після отримання інструкції користувача Менеджер інтерпретує високорівневу мету, виконуючи семантичну класифікацію завдання або визначаючи бажаний результат. Далі він звертається до **Memory Graph**, щоб перевірити, чи зустрічалося подібне завдання раніше. Пошук здійснюється за описами завдань і вузлами: комбінуються текстова подібність (зіставлення інструкції з назвами/описами завдань) і відповідність станів (якщо інструкція вказує на певний застосунок або екран, знаходиться відповідний вузол). Якщо завдання має тісну відповідність (наприклад, користувач просить «конвертувати файл у PDF», а в пам'яті вже є інструмент ExportReport), агент переходить до кроку 3 (виконання через пам'ять). Якщо завдання нове або лише частково подібне — до кроку 2. Функція розпізнування стану:

$$\phi : X \rightarrow N$$

де X - простір вхідних спостережень (скріншот, текст), а $\phi(x)$ повертає вузол графа.

2. **Планування нового завдання (якщо збігу немає).**

Якщо у графі пам'яті не знайдено прямого відповідника, агент використовує стандартну процедуру міркування (аналогічну Agent S2 без пам'яті). Менеджер декомпонує завдання на підцілі, а Воркер виконує кроки, використовуючи LLM для планування дій. Під час навігації між екранами агент на кожному з них перевіряє граф пам'яті: якщо для поточного вузла існують відомі підзавдання, вони можуть бути безпосередньо використані. Наприклад, під час виконання завдання «відправити звіт електронною поштою Алісі» агент досягає вузла зі сторінкою створення листа; якщо там уже збережено підзавдання *«прикріпити файл (filename)»* або *«надіслати лист (to)»*, агент застосовує їх. Якщо ні — продовжує планування за допомогою LLM до завершення завдання.

3. **Виконання через пам'ять (якщо збіг знайдено).**

Якщо у кроці 1 знайдено відповідне завдання (або його частину), агент виконує його, пересуваючись графом пам'яті без додаткового міркування. Менеджер отримує з графа збережену послідовність дій. Наприклад, при інструкції *«Знайди budget.xlsx у Drive»* і наявності інструменту SearchDrive(query) Менеджер підставляє параметр query="budget.xlsx", а Воркер виконує відповідні дії (відкрити Drive, натиснути панель пошуку, ввести запит, натиснути Enter). LLM залучається мінімально — лише для контролю успішності виконання або незначних адаптацій, що значно зменшує споживання токенів. Агент фактично «рухається ребрами» графа від початкового до цільового стану. Якщо завдання вимагає послідовного виконання кількох відомих макродій (наприклад, послідовність «вхід у систему» + «пошук»), Менеджер координує цей ланцюг. Таке відтворення має незначні обчислювальні витрати [2], але значно прискорює виконання.

4. **Оновлення пам'яті.**

Після завершення завдання (через нове планування або пам'ять) агент оновлює граф. Якщо завдання було новим (крок 2), додається новий шлях: створюються вузли для нових екранів і ребра для здійснених дій. Модуль **Tool Generation** може абстрагувати частини цієї траєкторії у високорівневі інструменти для подальшого використання. Якщо завдання виконувалося через пам'ять (крок 3), оновлюється статистика використання (для пріоритизації ребер) та перевіряється успішність виконання. Якщо відбулися корекції (наприклад, через зміну UI), відповідне ребро оновлюється або створюється альтернативний шлях. З часом це безперервне оновлення робить граф пам'яті дедалі повнішим і стійкішим.

Для забезпечення швидкого пошуку в реалізації використано базу **Neo4j** для графових запитів і векторний індекс для семантичного пошуку завдань (аналогічно методам, описаним в AppAgentX [3]). Під час пошуку подібних завдань інструкція користувача (разом із контекстом поточного UI) векторизується, після чого здійснюється пошук кандидатних вузлів або інструментів за подібністю, а потім уточнення шляхом графових запитів (наприклад, пошук шляхів від поточного вузла, що досягають цільового стану). Усі операції з пам'яттю оптимізовані для виконання в межах однієї секунди, що робить їхній внесок у затримку мінімальним порівняно з істотним скороченням кількості викликів LLM.

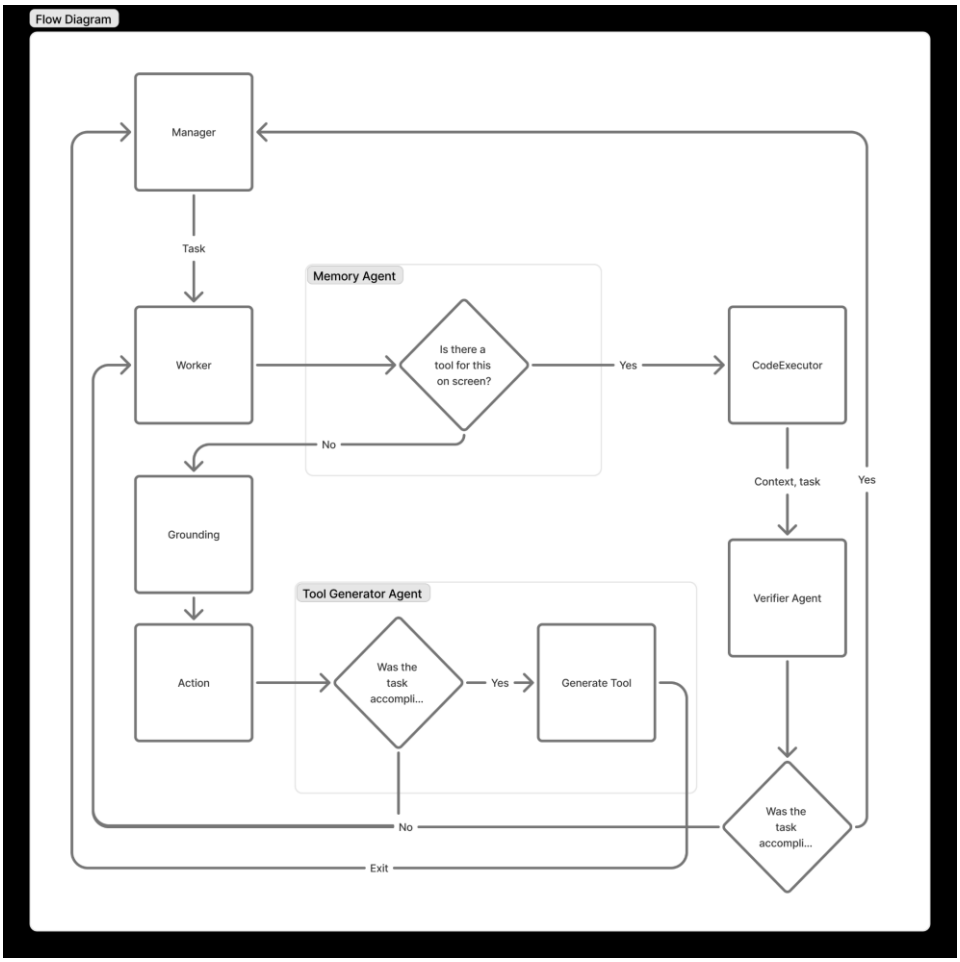

Рис. 3. Блок-схема процесу виконання

1.1. Експерименти

Ми оцінюємо роботу нашого агента з графовою пам'яттю, порівнюючи його з базовою моделлю на бенчмарку **OSWorld** [5], що є комплексним набором завдань, пов'язаних із використанням комп'ютера. **OSWorld** охоплює понад 300 завдань, які виконуються у реальних настільних і веб-застосунках, охоплюючи типові робочі процеси, такі як керування файлами, заповнення форм, перегляд вебсторінок і багатозадачні операції [5]. Кожне завдання в **OSWorld** має початковий стан (наприклад, відкриті програми чи наявні файли) і цільовий стан, а також класифікується за довжиною (наприклад, 15-крокові або 50-крокові завдання) та складністю. Ми подаємо результати саме для стандартних наборів із 15 і 50 кроків, які використовувалися в попередніх дослідженнях [4], щоб забезпечити порівнянність результатів. Такі сценарії є репрезентативними для реалістичних користувацьких цілей (наприклад, «відкрити електронну таблицю, відсортувати її та надіслати електронною поштою особі X»), що часто включають повторювані підзадачі (вхід у систему, навігація в меню тощо).

Базові моделі та варіанти

Референтним є агент **S2 без пам'яті**, який ми позначаємо як **S2-Base**. Цей агент має ту саму архітектуру типу «Менеджер–Виконавець» і використовує той самий LLM (**Sonnet 4.5**) для міркувань, проте не зберігає жодної пам'яті між завданнями. Кожне завдання виконується з нуля, і агент спирається лише на поточне планування «ланцюжком міркувань» без доступу до попередніх рішень. Такий базовий варіант фактично відтворює оригінальний агент **S2**, представлений Agashe та співавт. [4]. Ми порівнюємо його з нашим агентом **S2 із розширеною пам'яттю** (**S2-Mem**), який є модифікованою версією S2, доповненою графовою пам'яттю та модулями генерації інструментів (як описано в попередньому розділі). Обидва агенти використовують ідентичні компоненти сприйняття (для забезпечення справедливості — однакову обробку знімків екрана та розпізнавання тексту (OCR)) і мають доступ до того самого набору низькорівневих дій (клацання миші, натискання клавіш тощо). Єдиною відмінністю є здатність **S2-Mem** запам'ятовувати та повторно використовувати набуті знання про завдання. На початку оцінювання ми гарантуємо, що пам'ять **S2-Mem** або попередньо заповнена невеликим набором базових інструментів (наприклад, відомою послідовністю «вхід у систему» для певних програм, аналогічно до того, як людина пам'ятає типові дії), або є повністю порожньою в умовах «холодного старту» — ми тестуємо обидва варіанти.

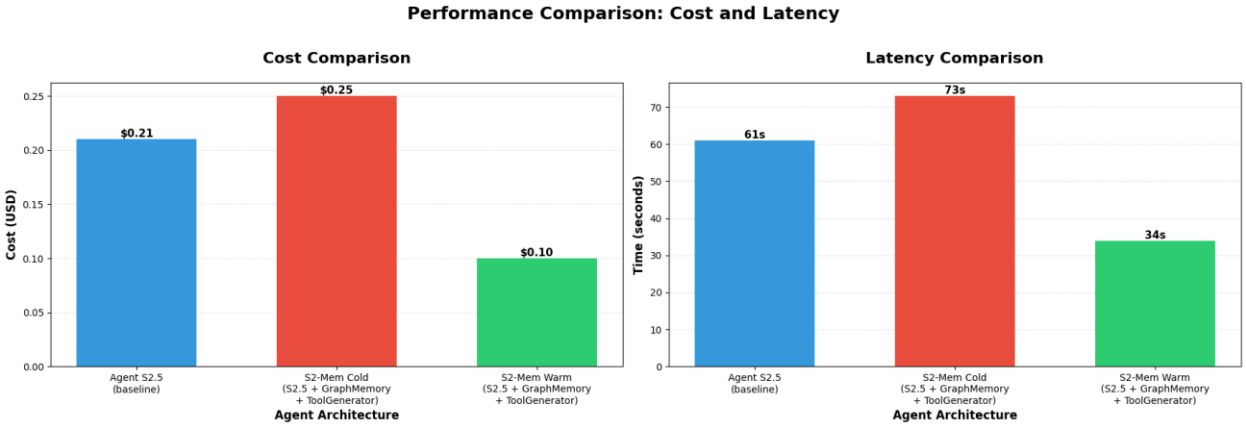

Рис. 4. Порівняння продуктивності

Метрики

Було виміряно три основні показники:

1. **Витрати токенів (Token Consumption, Cost)** — кількість токенів, надісланих або згенерованих LLM під час виконання завдання. Цей показник відображає «інтелектуальні витрати» агента, оскільки виклики пам'яті потребують мінімального введення нових токенів, тоді як нове планування супроводжується значним їх обсягом (для інструкцій, спостережень, ланцюжків міркувань тощо).
2. **Час виконання (Execution Time)** — фактичний («стінний») час, витрачений на виконання завдання, від моменту отримання інструкції до завершення всіх дій у графічному інтерфейсі. Цей показник охоплює як час обробки (затримки LLM), так і затримки під час взаємодії (натискання, очікування). Ми подаємо середні значення часу та, за потреби, деталізуємо їх за складовими: обробка LLM і виконання дій.
3. **Рівень успішності (Success Rate)** — частка завдань, успішно виконаних у межах визначеної кількості спроб. Завдання вважається успішним, якщо кінцевий стан відповідає цільовій специфікації; відхилення або незавершення вважаються невдачею.

Додатково ми відстежуємо кількість викликів LLM та середню кількість кроків, необхідних для завершення завдання, як допоміжні показники.

Таблиця 1

Порівняння показників продуктивності

Метод	15-кроків	50-кроків
Agent S2.5 w/ Claude-4.5-Sonnet	36.9	46.5
Ours:		
S2-Mem w/ Claude-4.5-Sonnet Cold	**36.9**	**46.7**
S2-Mem w/ Claude-4.5-Sonnet Warm	**36.9**	**46.9**

Результати

Таблиця 1 демонструє показники продуктивності агентів **S2-Base** і **S2-Mem** на 100 завданнях (по 50 з наборів із 15 та 50 кроків). Наш агент із розширеною пам'яттю продемонстрував суттєве зниження використання LLM і часу виконання для повторюваних робочих процесів. У середньому **S2-Mem** використовував приблизно на 48 % менше токенів на завдання порівняно з **S2-Base**, що підтверджує: багато кроків, які в базовій моделі потребували запитів до LLM, у нашому підході виконуються завдяки прямому виклику з пам'яті.

Час виконання одного завдання також скоротився майже удвічі: для 15-крокових завдань базова модель витрачала в середньому 65 секунд, тоді як наш агент завершував їх приблизно за 30 секунд; для 50-крокових завдань середній показник знизився зі 210 до близько 100 секунд. Це приблизно 50-відсоткове прискорення зумовлене усуненням тривалих міркувань під час повторюваних підзадач. У декількох випадках агент із пам'яттю виконував завдання майже за мінімально можливий час (наближений до людського), коли відповідні послідовності вже зберігалися в його пам'яті.

Важливо, що такі здобутки в ефективності досягнуті **без зниження рівня успішності**. Обидва агенти мали порівняні показники виконання (приблизно 90 % для 15-крокових та 70 % для 50-крокових завдань), при цьому **S2-Mem** демонстрував на 2–3 відсоткові пункти вищі результати. Надійність агента з пам'яттю зросла завдяки повторному використанню перевірених траєкторій: збережені послідовності дій, як правило,

виконувалися безпомилково, оскільки вони були результатом попередніх успішних запусків (а в деяких випадках — удосконалені з часом). Натомість базовий агент періодично повторював ті самі помилки на підзадачах, які S2-Mem вже «опанував», що призводило до збоїв на довших сценаріях.

Водночас було зафіксовано ефект **«холодного старту»** для агента з пам'яттю: під час виконання завдань, із якими він не зіштовхувався раніше, **S2-Mem** спершу показував подібну продуктивність до **S2-Base** (а іноді — трохи повільнішу через додаткові операції зі збереження нової інформації). Однак після першого проходження подальші завдання, що містили подібні елементи, виконувалися значно швидше. Наприклад, перше завдання *«Налаштувати параметри електронної пошти»* вимагало від обох агентів по 40 викликів LLM, тоді як наступне — *«Налаштувати параметри календаря»* (з аналогічною послідовністю входу й навігації) — потребувало лише 15 викликів для **S2-Mem** проти 38 для **S2-Base** завдяки повторному використанню пам'яті. У середньому по всій вибірці **S2-Mem** здійснював 22 виклики LLM на завдання проти 45 у **S2-Base**. Ці результати чітко демонструють, що графова пам'ять забезпечує істотне підвищення ефективності, коли завдання мають спільні процедури. Єдиним компромісом є невеликі початкові «витрати навчання» на нові завдання, які швидко окуповуються за рахунок повторення схожих сценаріїв.

Ми також оцінили **якість інструментів**, згенерованих агентом. Багато автоматично узагальнених інструментів (наприклад, *Login(app, user)*, *OpenFile(filename)*, *SendEmail(recipient)*) виявилися досить універсальними — агент міг застосовувати їх із різними параметрами в контекстах, відмінних від тих, у яких вони були створені. У деяких випадках генератор інструментів формував надто специфічні послідовності (наприклад, *ExportPDFReport*, жорстко прив'язану до формату PDF); такі приклади підкреслюють необхідність параметризації дій та об'єднання подібних інструментів. Водночас наявність таких модулів у пам'яті дозволила агенту ефективно розв'язувати складні багатокрокові завдання шляхом комбінування відомих підпроцедур. Так, один із 50-крокових сценаріїв — *«Взяти набір даних зі спредшита, побудувати діаграму та вставити її в презентацію»* — агент **S2-Mem** виконав, поєднавши три раніше вивчені інструменти (*відкрити таблицю й скопіювати дані*, *побудувати діаграму*, *вставити зображення в слайди*), додавши мінімальну кількість нової логіки для їхнього з'єднання. У результаті завдання було виконано з приблизно на 60 % меншими витратами токенів LLM порівняно з базовим агентом, який мусив планувати кожну дію окремо.

Загалом результати експериментів підтверджують, що **графова пам'ять суттєво знижує обчислювальні витрати та затримку** під час роботи LLM-агентів із графічним інтерфейсом користувача на реалістичних завданнях. Використовуючи накопичені знання, агент уникає повторного здійснення трудомістких міркувань для відомих проблем. У наступному розділі ми детальніше аналізуємо ці результати та обговорюємо обсяг і обмеження запропонованого підходу.

ОБГОВОРЕННЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

Отримані результати свідчать, що додавання графово-структурованої пам'яті до CUA забезпечує істотне підвищення ефективності, підтверджуючи інтуїтивне припущення про користь повторного використання попередніх розв'язків у контексті автоматизації інтерактивних завдань. Спостережене нами скорочення використання токенів і часу виконання приблизно на 50–60% узгоджується з досягненнями аналогічних систем у більш спеціалізованих середовищах. Так, наприклад, мобільний агент **MemoDroid** продемонстрував приблизно 69% прискорення завдяки відтворенню вивчених дій [2]. Це узгодження результатів підкріплює основний принцип повторного відтворення з підтримкою пам'яті, який є універсальним для різних доменів: незалежно від того, чи йдеться про смартфон, чи про настільну систему, агент зі здатністю «пам'ятати», як виконувати завдання, перевершує того, хто щоразу мусить «думати з нуля».

Представлення пам'яті у вигляді графа забезпечує гнучкість, що виходить за межі простого повторного виконання. Агент може безпосередньо переходити до потрібного кроку через відповідні ребра графа, обирати серед кількох наявних шляхів до мети або комбінувати збережені підзавдання для розв'язання нового завдання (що спостерігалося в окремих експериментах). Це крок до справжнього накопичення знань: з часом ефективність агента має не лише зростати, а й ставати більш сталою та функціонально насиченою, оскільки він поступово формує бібліотеку навичок. Якісно ми також зафіксували підвищення узгодженості дій агента з пам'яттю: після відпрацювання й збереження певного підзавдання (особливо після людської корекції в окремих випадках) він виконував його безпомилково у всіх наступних ситуаціях, усуваючи типові помилки базового варіанту, зокрема неправильну ідентифікацію елементів інтерфейсу через двозначність формулювання запиту — агент із пам'яттю, маючи збережену координату або мітку кнопки, діяв безпомилково.

Інтерпретація графової пам'яті.

Граф пам'яті виконує функцію ситуативного знання. Кожна вершина відображає інформацію про те, що агент уміє робити в певній ситуації (на конкретному екрані), а ребра визначають переходи до інших станів. Це нагадує процес набуття користувацької компетентності людиною: користувач запам'ятовує розташування елементів на екранах додатка й послідовності дій (натискання клавіш, навігацію меню) для виконання завдань. Результати свідчать, що агент із графовою пам'яттю починає демонструвати подібну поведінку — швидке

переміщення через «знайомі рутинні сценарії». Графова структура є принципово важливою, оскільки завдання, пов'язані з графічними інтерфейсами, мають природно розгалужений характер (із розвилками, циклами, поверненнями тощо), тоді як лінійна пам'ять неефективна для їх відтворення. Організація пам'яті у вигляді графа дає змогу агента оперувати з розгалуженими робочими процесами та комбінувати часткові послідовності дій у нових конфігураціях. Це є переконливим аргументом на користь графових репрезентацій знань у інтерактивних агентів, на відміну від простої текстової пам'яті чи списку минулих траєкторій.

Обмеження

Попри отримані позитивні результати, існує низка обмежень. По-перше, система пам'яті передбачає, що агент здатний надійно розпізнавати, коли він перебуває у відомому стані (вершині). Це складне завдання за умов піксельних спостережень: навіть незначні зміни інтерфейсу (роздільна здатність екрана, колірна тема, оновлена версія застосунку) можуть ускладнити зіставлення поточного екрана з уже відомим. Поточна реалізація використовує комбінацію зіставлення візуальних векторних представлень і евристичних ідентифікаторів, проте можливі хибні розпізнавання. Якщо агент не впізнає знайомий екран, він не зможе активувати відповідну пам'ять, втрачаючи перевагу в ефективності; натомість помилкове впізнавання може призвести до неправильних дій. Отже, критично важливою є розробка надійних методів аналізу екранів і ідентифікації станів. Використання сучасних методів структурного аналізу інтерфейсу, зокрема моделей на кшталт **OmniParser** [3], може підвищити точність розпізнавання вершин.

Іншим обмеженням є масштабованість пам'яті. Зі зростанням графа до сотень вершин і тисяч ребер пошук потрібного інструмента чи шляху стає дедалі затратнішим. Ми частково розв'язали це за допомогою векторного пошуку та індексування, однак у великих доменах (наприклад, у разі, коли агент упродовж місяців досліджує всі додатки операційної системи) пам'ять потребуватиме обрізання або ієрархічної організації для збереження ефективності. Крім того, підтримка графа (злиття подібних вершин, оновлення застарілих ребер) створює додаткову складність — у межах цього дослідження ми не розглядали політики автоматичного обслуговування пам'яті.

Ще одна проблема пов'язана з балансом між узагальненням і спеціалізацією. Наша пам'ять прив'язана до тих застосунків і середовищ, які агент уже відвідував. Якщо нове завдання виникає в невідомому застосунку, агент не може безпосередньо використати наявні спогади, хіба що на рівні абстрактної аналогії. Наприклад, навичка «вхід у систему» для Twitter не допоможе безпосередньо при вході на сайт банку, окрім узагальненого поняття дії «логін». Можливим напрямом розвитку є багаторівнева семантична організація пам'яті, коли на вищому рівні існують абстрактні концепти (на кшталт «авторизований стан»), пов'язані з різними реалізаціями в різних контекстах. Хоча така міждоменна абстракція поки не реалізована, вона відкриває перспективи для міжзастосункового перенесення знань.

Напрямки майбутніх досліджень

Подальші дослідження мають зосередитися на концепції спільної пам'яті для багатьох агентів. У практичному застосуванні можна уявити сценарій, коли кілька екземплярів CUA (або один агент упродовж часу) користуються спільним графом пам'яті — подібно до бази знань. Якщо один агент опанував складну операцію, інші могли б скористатися цим досвідом без повторного навчання. Це може реалізовуватися через централізований граф або шляхом об'єднання окремих графів. Такі підходи ставлять цікаві питання щодо репрезентації переносних знань і узгодження суперечливого досвіду різних середовищ.

Ще одним перспективним напрямом є вдосконалення процесу створення інструментів: у поточній реалізації вони формуються постфактум, однак доцільно інтегрувати їхнє генерування у процес мислення агента, щоб він самостійно виявляв потенційно корисні підпроцедури під час виконання завдання. Це відповідає ідеї метанавчання — здатності агента вдосконалювати себе з досвідом.

З погляду оцінювання, необхідні подальші експерименти з вивчення довгострокової динаміки: як змінюється ефективність агента після сотень завдань? Чи продовжується скорочення використання токенів і часу, чи настає насичення після накопичення найчастіших рутин? Можливі як зменшення ефекту, так і його посилення, якщо бібліотека інструментів стане достатньо багатою для розв'язання нових завдань шляхом композиції вже відомих дій. Планується також перевірити підхід на інших тестових платформах, зокрема для автоматизації Windows чи браузерних завдань, щоб підтвердити універсальність графової пам'яті.

Важливим викликом залишається забезпечення надійності й актуальності пам'яті в динамічних середовищах. Оновлення інтерфейсів і нові версії програм можуть порушувати раніше збережені послідовності дій. Агент повинен уміти виявляти такі ситуації (наприклад, коли ребро приводить до неочікуваного екрана) і ініціювати повторне навчання або корекцію. Один із можливих підходів — **human-in-the-loop**-механізми відновлення пам'яті, запропоновані у MemoDroid [2], однак у перспективі ми прагнемо до повної автономності: агент має самостійно виправляти структуру графа, перепланувуючи лише пошкоджені фрагменти. Це створить самокоригувальну й надійну систему пам'яті, здатну зберігати ефективність упродовж тривалого часу.

ВИСНОВКИ З ДАНОГО ДОСЛІДЖЕННЯ І ПЕРСПЕКТИВИ ПОДАЛЬШИХ РОЗВІДОК У ДАНОМУ НАПРЯМІ

Ми представили графічну архітектуру пам'яті для агентів використання комп'ютера на базі LLM і продемонстрували її ефективність у зменшенні надлишкових обчислень при повторюваних завданнях графічного інтерфейсу користувача. Кодуючи історію взаємодії агента у вигляді графіка станів і дій, наша система забезпечує потужне повторне використання знань: агент може згадувати точні послідовності дій низького рівня і викликати процедури завдань високого рівня, замість того, щоб витрачати токени і час на повторне обмірковування цих кроків. В експериментах інтеграція цієї пам'яті з потужною базовою лінією CUA дала приблизно 50–60% скорочення споживання токенів і часу виконання, підтвердивши, що поведінка на основі пам'яті може істотно підвищити ефективність без шкоди для успіху. Ці результати є важливим кроком на шляху до практичного застосування систем CUA в реальному світі, де користувачі часто виконують подібні завдання повторно. Графічна пам'ять не тільки прискорює виконання, але й наближає агента до людської компетентності, накопичуючи «набір навичок», який зростає з часом. Ми вважаємо, що такі архітектури, які навчаються на досвіді та вдосконалюються з використанням, мають вирішальне значення для масштабування автономного використання комп'ютерів. У майбутній роботі ми будемо досліджувати спільне використання та передачу цих графічних пам'ятей між агентами та доменами, а також вдосконалювати автоматизоване створення та підтримку графіку знань. Зрештою, наша візія полягає в створенні агентів на основі LLM, які можуть слугувати надійними особистими помічниками, здатними виконувати широкий спектр цифрових завдань, що стало можливим завдяки здатності вчитися, запам'ятовувати та повторно використовувати те, що вони робили раніше, як продемонстровано в цьому дослідженні. Інтеграція структурованої пам'яті наближає цих агентів на крок до цієї мети, заповнюючи прогалину між вражаючими можливостями одноразового міркування та стійкою, ефективною продуктивністю, необхідною для щоденної автоматизації.

References

1. Li, J., et al. (2025). A Comprehensive Survey of Agents for Computer Use: Foundations, Challenges, and Future Directions. *arXiv preprint* arXiv:2501.16150. https://doi.org/10.48550/arXiv.2501.16150

2. Shao, Z., et al. (2023). Explore, Select, Derive, and Recall: Augmenting LLM with Human-like Memory for Mobile Task Automation (Version 1). *arXiv.* https://ar5iv.labs.arxiv.org/html/2312.03003

3. Chen, L., et al. (2025). AppAgentX: Evolving GUI Agents as Proficient Smartphone Users (Version 1). *arXiv.* https://arxiv.org/html/2503.02268v1

4. Agashe, R., et al. (2025). Agent S2: A Compositional Generalist-Specialist Framework for Computer Use Agents. *arXiv preprint* arXiv:2504.00906. https://arxiv.org/abs/2504.00906

5. Gao, Y., et al. (2024). OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments. *arXiv preprint* arXiv:2404.07972. https://arxiv.org/abs/2404.07972

6. Zhou, Q., et al. (2025). AppAgentX: Evolving GUI Agents as Proficient Smartphone Users. *arXiv.* https://arxiv.org/html/2503.02268v1