Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА
ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «МАГІСТР»

ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПЕРСОНАЛІЗОВАНОГО АСИСТЕНТА ФІНАНСОВОЇ СВІДОМОСТІ ТА ЗАОЩАДЖЕНЬ: ДОСЛІДЖЕННЯ НА ПРИКЛАДІ УКРАЇНСЬКОГО РИНКУ

DESIGN AND IMPLEMENTATION OF THE PERSONALIZED FINANCIAL AWARENESS AND SAVINGS ASSISTANT: A CASE STUDY OF THE UKRAINIAN MARKET

спеціальність 121 Інженерія програмного забезпечення
освітня програма «Інженерія програмного забезпечення»

Виконав: здобувач вищої освіти
групи ІПЗмз-21
Бик В. М.

Керівник:
д.т.н., професор
Андрущак І. Є.

Кваліфікаційну роботу
допущено до захисту
«___» _____ 20__ р.
к.т.н., доцент
Гарант освітньої програми:
Суринович Олена Миколаївна
_____

Луцьк – 2025 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *інженерії програмного забезпечення*
Ступінь вищої освіти *магістр*
Галузь знань: *12 «Інформаційні технології»*
Спеціальність: *121 «Інженерія програмного забезпечення»*
Освітня програма: *«Інженерія програмного забезпечення»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____
«___» _____ 202__ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧА
ДРУГОГО (МАГІСТЕРСЬКОГО) РІВНЯ ВИЩОЇ ОСВІТИ

Бику Володимиру Михайловичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи: Проєктування та реалізація персоналізованого асистента фінансової свідомості та заощаджень: дослідження на прикладі українського ринку
Керівник роботи: д.т.н., професор Андрущак І. Є.
затверджені наказом закладу вищої освіти від «20» вересня 2025 р. № 386/01 - 02
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи: «04» грудня 2025 р.
3. Вихідні дані до роботи: *технічне та програмне забезпечення ЕОМ*
4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити):*Аналіз сучасного стану проблеми, існуючих методів і засобів її розв'язання, аналіз і вибір засобів проектування, опис функціонального наповнення об'єкта проектування, розробка й обґрунтування системного наповнення, оцінка ергономічних та надійністних параметрів проектованої системи, функціонально-структурна схема роботи об'єкта проектування.*
5. Перелік графічного матеріалу: 28 рисунків, 14 таблиць, 2 лістингів коду

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада Консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз проблеми за темою роботи та постановка завдань дослідження	*Андрущак І. Є.*		
Теоретичне дослідження та практична реалізація	*Андрущак І. Є.*		
Експериментальне дослідження системи	*Андрущак І. Є.*		
Нормоконтроль	*Повстяна Ю. С.*		
Гарант ОП	*Андрущак І. Є.*		
Показник запозичень тексту	___%		
Академічна доброчесність	*Андрущак І. Є.*		

7. Дата видачі завдання *« 20 » вересня 2025 р.*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи магістра	Строк виконання етапів роботи	Примітка
1	*Провести огляд літературних джерел по темі кваліфікаційної роботи*	*05.09.2025 р.*	
2	*Провести аналіз загальної проблеми і вибір напрямків дослідження*	*24.09.2025 р.*	
3	*Розробити функціональну схему роботи програмного продукту*	*01.10.2025 р.*	
4	*Описати засоби розробки об'єкта проектування*	*19.10.2025 р.*	
5	*Практична реалізація об'єкта проектування*	*26.10.2025 р.*	
6	*Розробити методику для проведення експерименту*	*05.11.2025 р.*	
7	*Провести аналіз результатів експерименту*	*15.11.2025 р.*	
8	*Здача чистового варіанту магістерської роботи на кафедру*	*04.12.2025 р.*	

Здобувач вищої освіти _____ Бик В. М.
(підпис) (прізвище, ініціали)

Керівник кваліфікаційної роботи _____ Андрущак І. Є.
(підпис) (прізвище, ініціали)

АНОТАЦІЯ

Бик В. М. Проєктування та реалізація персоналізованого асистента фінансової свідомості та заощаджень: дослідження на прикладі українського ринку. Рукопис.

Кваліфікаційна робота магістра ОП «Інженерія програмного забезпечення». Спеціальності 121 «Інженерія програмного забезпечення». Луцький національний технічний університет. Луцьк, 2025.

Кваліфікаційна робота магістра складається зі вступу, трьох розділів, висновків, списку використаних джерел (згідно структури кваліфікаційної роботи, затвердженої кафедрою).

У першому розділі здійснено аналіз сучасного стану ринку застосунків для управління особистими фінансами, розглянуто існуючі програмні рішення-аналоги, методи та засоби розробки персоналізованих фінансових асистентів. У другому розділі спроєктовано архітектуру системи та базу даних, реалізовано програмні компоненти серверної частини на мові Go та інтелектуального ШІ-агента на Python, а також інтеграцію з відкритими банківськими API (на прикладі Monobank). У третьому розділі проведено експериментальне дослідження ефективності роботи системи під навантаженням, здійснено аналіз продуктивності, надійності та коректності функціонування програмного продукту. У висновках узагальнено результати дослідження та практичної реалізації.

Ключові слова: персональні фінанси, фінансовий асистент, Open Banking, Monobank API, Go, Python, SQLite, мікросервісна архітектура, штучний інтелект, LangChain, Docker.

ABSTRACT

Byk V. M. Design and implementation of the personalized financial awareness and savings assistant: a case study of the Ukrainian market. Manuscript.

Master's qualification work of OP «Software Engineering». Specialties 121 «Software engineering». Lutsk National Technical University. Lutsk, 2025.

The master's thesis consists of an introduction, three chapters, conclusions, and a list of references (according to the structure of the qualification work approved by the department).

In the first section, we analyze the current state of the market of applications for managing personal finances, consider pre-existing software solutions, methods and tools for developing personalized financial assistants. In the second section, we design the system architecture and database, implement the software components of the backend in the Go language and the intelligent AI agent in Python, as well as integration with open banking APIs (with the example of Monobank). In the third section, an experimental study of the system's performance under load was conducted, and the analysis of the performance, reliability, and correctness of the software product's functioning was carried out. The conclusions summarize the results of the research and practical implementation.

Keywords: personal finance, financial assistant, Open Banking, Monobank API, Go, Python, SQLite, microservice architecture, artificial intelligence, LangChain, Docker.

ЗМІСТ

ВСТУП 7
РОЗДІЛ 1 АНАЛІЗ ПРОБЛЕМАТИКИ ЗА ТЕМОЮ РОБОТИ ТА ПОСТАНОВКА ЗАВДАНЬ ДОСЛІДЖЕННЯ 10
1.1 Огляд і аналіз предметної області проблеми, результатів існуючих теоретичних та експериментальних досліджень 10
1.2 Огляд і аналіз методів та засобів розробки асистента фінансової свідомості та заощаджень 12
1.3 Постановка завдання на кваліфікаційну роботу магістра 16
РОЗДІЛ 2 ТЕОРЕТИЧНЕ ДОСЛІДЖЕННЯ ТА ПРАКТИЧНА РЕАЛІЗАЦІЯ АСИСТЕНТА ФІНАНСОВОЇ СВІДОМОСТІ ТА ЗАОЗАДЖЕНЬ 17
2.1 Обґрунтування вибору шляхів, технологій, алгоритмів і засобів вирішення поставленого завдання 17
2.2 Практична реалізація об'єкта проєктування 21
РОЗДІЛ 3 ЕКСПЕРЕМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ РЕЗУЛЬТАТИВНОСТІ АСИСТЕНТА ФІНАНСОВОЇ СВІДОМОСТІ ТА ЗАОЗАДЖЕНЬ 38
3.1 Методика проведення дослідження 38
3.2 Обробка та аналіз отриманих результатів 41
ВИСНОВКИ 57
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ 59

ВСТУП

Актуальність кваліфікаційної роботи магістра пов'язана необхідністю створення такого інструменту, який би дозволив користувачам українських банків автоматизувати процес збору, категоризації та аналізу своїх транзакцій, допомагаючи формувати корисні фінансові звички, ставити та досягати заощаджувальні цілі. Враховуючи, що 1 серпня 2025 року в Україні було затверджено положення про відкритий банкінг [1], можна припустити, що ринок безпечних фінансових застосунків буде тільки рости, а з цим і актуальність застосунку представленого в цій роботі.

Мета дослідження – проєктування та реалізація програмного продукту «Персоналізований асистент фінансової свідомості та заощаджень», що надає користувачам інструменти для автоматизованого збору, аналізу та візуалізації персональних фінансових даних для покращення фінансової грамотності та досягнення заощаджувальних цілей.

Об'єкт дослідження – процес управління особистими фінансами користувача в умовах сучасної цифрової банківської екосистеми.

Предмет дослідження – моделі, методи та програмні засоби, що лежать в основі розробленого асистента, зокрема: архітектура гібридної системи, що поєднує сервіси на Go та Python; методи інтеграції з відкритими банківськими API (на прикладі Monobank API); структура та способи взаємодії з вбудованою базою даних SQLite; та підходи до реалізації аналітичного функціоналу, включно з обробкою запитів природною мовою (ШІ-агент).

Завданням дослідження є:

– проаналізувати предметну область управління особистими фінансами та дослідити існуючі програмні рішення-аналоги;

– спроєктувати гнучку та масштабовану архітектуру програмного продукту, що складається з кількох незалежних мікро сервісів на мові Go та Python;

– розробити структуру бази даних SQLite для зберігання інформації про

користувачів, їхні рахунки, транзакції, цілі та доходи;

– реалізувати модуль інтеграції з прикладним програмним інтерфейсом (API) Monobank для автоматичного отримання даних про користувацькі транзакції;

– створити основну бізнес-логіку та веб-інтерфейс застосунку, що реалізує ключові функції аналізу та візуалізації;

– розробити фонові процеси для забезпечення регулярної синхронізації даних без участі користувача;

– створити ШІ-агента який матиме доступ до бази даних з користувацькими даними (БД) та самостійно прийматиме рішення які дані потрібно дістати з БД, щоб відповісти на фінансові питання користувачів у форматі чату;

– провести комплексне тестування розробленого продукту для перевірки його коректності та ефективності;

– перевірити контейнеризацію за допомогою Docker.

Наукова новизна одержаних результатів кваліфікаційної роботи магістра полягає в тому, що вперше для українського ринку було запропоновано та реалізовано архітектуру персонального фінансового асистента з відкритим вихідним кодом, яка ефективно поєднує декілька сучасних технологічних підходів. Зокрема, новизна полягає у гібридній мікросервісній моделі, де високопродуктивний компільований бекенд на мові Go відповідає за основні операції та швидкість роботи системи, а гнучкі сервіси на мові Python використовуються для реалізації інтелектуальної обробки даних, зокрема, для аналізу запитів користувача природною мовою за допомогою великих мовних моделей (LLM [5]). Також новим є застосування безпечної моделі доступу до даних для AI-компонента через спеціалізований MCP [6] сервіс, що ізолює його від прямої взаємодії з базою даних.

Практичне значення та застосування отриманих результатів полягає у створенні повноцінного, готового до розгортання прототипу програмного продукту. Завдяки контейнеризації за допомогою Docker [7] та наявності

конфігураційних файлів для хмарних платформ (напр., fly.toml), застосунок може бути легко розгорнутий для особистого використання. Розроблений код та архітектурні рішення можуть слугувати надійною основою для подальшого розвитку продукту, його комерціалізації або для використання в освітніх цілях як приклад побудови сучасних багатокомпонентних систем.

Апробація результатів кваліфікаційної роботи:

– Koshelyuk V., Andrushchak I., Byk V. For developing a cryptographic two-factor authentication protocol. Proceedings of the XV International Scientific and Practical Conference. Lyon, France. December 09-12, 2025. Pp. 41-45. URL: https://isg-konf.com/current-trends-in-the-development-of-existing-research-and-theories/.

– Andrushchak I., Byk V. Features of the protection system of operating systems. Proceedings of the XV International Scientific and Practical Conference. Lyon, France. December 09-12, 2025. Pp. 73-77. URL: https://isg-konf.com/current-trends-in-the-development-of-existing-research-and-theories/. ISBN – 979-8-90070-302-2, DOI – 10.46299/ISG.2025.2.15

РОЗДІЛ 1
АНАЛІЗ ПРОБЛЕМАТИКИ ЗА ТЕМОЮ РОБОТИ ТА ПОСТАНОВКА ЗАВДАНЬ ДОСЛІДЖЕННЯ

1.1 Огляд і аналіз предметної області проблеми, результатів існуючих теоретичних та експериментальних досліджень

Сучасний світ характеризується високою динамікою фінансових відносин, що висуває підвищені вимоги до рівня фінансової грамотності кожної людини. Управління особистим бюджетом, планування витрат, формування заощаджень та досягнення фінансових цілей є невід'ємними складовими економічного добробуту. В умовах війни та глобальних викликів, що стоять перед Україною, рівень фінансової грамотності населення набуває критичного значення. Ефективне управління особистими фінансами, здатність до заощадження та інвестування є не лише запорукою добробуту окремої людини, але й важливим фактором економічної стабільності держави. Водночас, спостерігається стрімкий розвиток цифрових технологій у банківській сфері. Поява так званих необанків, лідером серед яких на українському ринку є Monobank, кардинально змінила підхід до взаємодії клієнта з банком, зробивши її повністю мобільною та інтерактивною. Ця цифрова трансформація генерує величезні обсяги персональних фінансових даних, які, за наявності правильних інструментів, можуть стати основою для глибокого аналізу та прийняття зважених фінансових рішень.

Проте, більшість існуючих популярних на глобальному ринку рішень для управління фінансами (такі як Mint, YNAB) не адаптовані до специфіки українського ринку, зокрема, не підтримують автоматичну інтеграцію з українськими банками. В свою чергу, більше 50 % всіх встановлень в Україні припадає на застосунки з українськими банківськими інтеграціями розробили закордоном [2]. Наприклад, додаток Zenmoney бере в українців логін і пароль до Приват24 та досі має доступ до корпоративного API Monobank. Торкаючись питань безпеки та приватності даних – більшість популярних застосунків не

мають відкритого вихідного коду, і не надають можливість самохостингу [3]. А ті застосунки які це дозволяють, наприклад «Firefly III» [4], не мають прямих інтеграцій з українськими банками та моделями штучного інтелекту (ШІ). Це створює очевидну прогалину та формує попит на сучасний, безпечний та автоматизований програмний інструмент, орієнтований на українського користувача, який за бажанням надає повний контроль над користувацькими даними (самохостинг). Даний розділ присвячений аналізу предметної області управління особистими фінансами, огляду існуючих програмних рішень та технологічних засобів, а також чіткій постановці завдань, що підлягають вирішенню в рамках даної кваліфікаційної роботи.

Предметна область управління особистими фінансами охоплює методи та практики обліку, аналізу та планування грошових потоків індивіда чи домогосподарства. Основою ефективного управління є бюджетування – процес створення плану витрат та заощаджень. Існує декілька популярних методологій, що допомагають структурувати цей процес.

Однією з найвідоміших є правило «50/30/20» [8], яке пропонує розподіляти дохід після оподаткування на три основні категорії: 50 % на потреби (житло, комунальні послуги, транспорт, продукти), 30 % на бажання (розваги, хобі, подорожі) та 20 % на заощадження та виплату боргів. Ця методологія є простою для розуміння та впровадження, що робить її привабливою для початківців. Саме цей підхід було обрано для реалізації однієї з ключових функцій розроблюваного асистента. Інші підходи, такі як «бюджетування з нульовою базою» (Zero-Based Budgeting) [9], де кожна гривня доходу розподіляється заздалегідь, або «метод конвертів» [10], вимагають вищого рівня дисципліни, але також є ефективними.

Ринок програмних засобів для управління особистими фінансами (Personal Finance Management) є досить розвиненим. Серед світових лідерів можна виділити такі продукти, як Mint, YNAB (You Need A Budget) та Personal Capital. Їхніми сильними сторонами є зрілий функціонал, зручний інтерфейс та можливість інтеграції з тисячами банків у США та Європі. Однак для українського користувача їхня корисність суттєво обмежена через відсутність

автоматичної синхронізації з українськими банками, проблеми з коректним відображенням гривні та мовні бар'єри.

Український ринок PFM-додатків є менш насиченим. Існуючі рішення часто є або мобільними застосунками від самих банків (наприклад, аналітика витрат у додатках Monobank, Приват24), які обмежуються даними лише одного банку, або незалежними додатками, що вимагають від користувача ручного введення кожної транзакції. Ручне введення є трудомістким процесом, що призводить до низької прихильності користувачів та неповноти даних.

Таким чином, аналіз предметної області виявляє чітку нішу: існує потреба у створенні локалізованого для України фінансового асистента, який би забезпечував автоматичну синхронізацію з провідними українськими банками та надавав гнучкі інструменти для аналізу, що виходять за межі базової функціональності банківських додатків.

1.2 Огляд і аналіз методів та засобів розробки асистента фінансової свідомості та заощаджень

Для реалізації поставленої задачі необхідно було проаналізувати доступні технологічні засоби, зокрема джерела даних та стек розробки.

Ключовою вимогою до системи є автоматизація збору даних. В Україні концепція проте деякі банки-лідери надають власні публічні API. Найбільш функціональним та добре документованим є API Monobank [13]. Він дозволяє після автентифікації користувача отримувати в режимі реального часу інформацію про рахунки, баланс, а також деталізовані виписки по транзакціях за певний період. Саме цей API було обрано як основне джерело даних для прототипу.

Вибір технологічного стеку є стратегічним рішенням, що впливає на продуктивність, надійність та складність підтримки системи.

Мова програмування для бекенду розглядалися Go, Python та Node.js. Мову Go було обрано через її високу продуктивність (компіляція в нативний код),

вбудовану підтримку конкурентності (горутини), що є ідеальним для реалізації фонових воркерів (як txpuller), та сувору статичну типізацію, що підвищує надійність коду.

Для бази даних, порівнювались вбудована СУБД SQLite та клієнт-серверна PostgreSQL. Для прототипу персонального асистента, де дані належать одному користувачеві і не очікується високе конкурентне навантаження, SQLite є оптимальним вибором. Вона не потребує окремого сервера (є файлом на диску), проста в налаштуванні та має мінімальні накладні витрати, що ідеально для портативного та легковагового рішення.

Для фронтенду, розглядався вибір між створенням Single-Page Application (SPA) на фреймворках типу React/Vue та підходом Server-Side Rendering (SSR). Було обрано SSR за допомогою бібліотеки templ для Go. Цей підхід дозволяє генерувати HTML на стороні сервера, що прискорює початкове завантаження сторінок, спрощує архітектуру (немає потреби в окремому фронтенд-проєкті та API для нього) і є природним продовженням екосистеми Go. Для забезпечення динамічної взаємодії без перезавантаження сторінки, цей підхід доповнено бібліотекою htmx, що дозволяє оновлювати окремі частини інтерфейсу за запитами до сервера. Комбінація templ та htmx стала ефективною та легкою альтернативою традиційним фронтенд фреймворкам.

На основі аналізу предметної області та ринку існуючих рішень були сформовані функціональні та нефункціональні вимоги до розроблюваного продукту.

Функціональні вимоги:

– автентифікація та ролі користувачів: система повинна підтримувати реєстрацію та вхід користувачів через зовнішнього провайдера (Google OAuth 2.0 [11]), а також розрізняти ролі (звичайний користувач, адміністратор) для потенційного розширення функціоналу;

– інтеграція з банками: користувач повинен мати можливість безпечно підключити свій акаунт Monobank за допомогою персонального токена доступу та синхронізувати транзакції з підключеного банку у фоновому режимі;

– управління транзакціями: система повинна відображати повний список транзакцій користувача з можливістю пошуку та фільтрації;

– користувач повинен мати змогу класифікувати витрати за трьома основними класами: «потреби», «бажання», «заощадження»;

– система повинна автоматично категоризувати транзакції на основі їх MCC-кодів [14];

– система повинна надавати візуальну аналітику витрат відповідно до бюджетного правила «50/30/20»;

– система повинна відображати зведену інформацію про доходи та витрати за обрані періоди;

– користувач повинен мати можливість створювати процентні цілі для обмеження витрат у певних категоріях (наприклад, «витрачати не більше 15 % доходу на ресторани»);

– користувач при бажанні може легко видалити усі свої дані із застосунку;

– система повинна автоматично генерувати та надсилати сповіщення про результати дотримання місячних бюджетних цілей;

– система повинна щоденно аналізувати тенденції витрат і надсилати попереджувальні сповіщення;

– сповіщення мають зберігатися в базі даних та бути доступними для перегляду у веб-інтерфейсі;

– система повинна надавати інтерфейс для взаємодії з AI-агентом за допомогою запитів українською мовою;

– ШІ-агент повинен бути здатним відповідати на аналітичні питання щодо фінансових даних користувача.

Нефункціональні вимоги:

– додаток повинен працювати та витримувати навантаження на недорогому програмному забезпечені популярних хмарних вендорів. Цільові рівні обслуговування (Service Level Objective [12]), визначаються відштовхуючись від наступних апаратних характеристик сервера: 2 GB оперативної пам'яті, 2 спільних vCPU (віртуальні ядра процесора) та диску із

максимумом в 2000 IOP та пропускною здатністю 8 МіБ/с;

– при 100 активних віртуальних користувачах (VUs) веб застосунку, що читають та пишуть, наступні показники (SLO) повинні бути правдивими: час відповіді не має перевищувати 500 мс в 95-квантилі (p95), або іншими словами, лише 5 % від запитів можуть бути довшими за 500 мс. Кількість непередбачених помилок сервера не повинна перевищувати 1 %. Система повинна стабільно обробляти щонайменше 50 запитів на секунду (RPS);

– токени доступу до банківських API повинні зберігатися у зашифрованому вигляді;

– система повинна гарантувати повну ізоляцію даних між різними користувачами;

– взаємодія з системою із зовнішнього світу (інтернет) повинна відбуватися через захищене HTTPS-з'єднання;

– надійність та стабільність при тривалому навантаженні: система повинна витримувати навантаження у 20 VUs протягом 30 хв без деградації часу відповіді (p95) в 500 мс;

– система повинна бути розроблена для цілодобової роботи;

– збій допоміжних компонентів (наприклад, ШІ-агента) не повинен призводити до відмови основного функціоналу;

– приватність та контроль над даними: система повинна підтримувати можливість розгортання на власній інфраструктурі користувача (самохостинг);

– усі фінансові дані користувача повинні зберігатися локально у файлі бази даних SQLite, що забезпечує максимальний рівень приватності;

– сумісність: веб-інтерфейс повинен коректно працювати в останніх версіях популярних браузерів (Chrome, Firefox, Safari).

1.3 Постановка завдання на кваліфікаційну роботу магістра

На основі проведеного аналізу предметної області, існуючих рішень та

доступних технологій, були сформульовані наступні завдання для виконання в рамках кваліфікаційної роботи:

– проаналізувати предметну область управління особистими фінансами та дослідити існуючі програмні рішення-аналоги;

– спроєктувати гнучку та масштабовану архітектуру програмного продукту, що складається з кількох незалежних мікро сервісів на мові Go та Python;

– розробити структуру бази даних SQLite для зберігання інформації про користувачів, їхні рахунки, транзакції, цілі та доходи;

– реалізувати модуль інтеграції з прикладним програмним інтерфейсом (API) Monobank для автоматичного отримання даних про користувацькі транзакції;

– створити основну бізнес-логіку та веб-інтерфейс застосунку, що реалізує ключові функції аналізу та візуалізації;

– розробити фонові процеси для забезпечення регулярної синхронізації даних без участі користувача;

– створити ШІ-агента який матиме доступ до бази даних з користувацькими даними (БД) та самостійно прийматиме рішення які дані потрібно дістати з БД, щоб відповісти на фінансові питання користувачів у форматі чату;

– провести комплексне тестування розробленого продукту для перевірки його коректності та ефективності;

– перевірити контейнеризацію за допомогою Docker.

РОЗДІЛ 2
ТЕОРЕТИЧНЕ ДОСЛІДЖЕННЯ ТА ПРАКТИЧНА РЕАЛІЗАЦІЯ АСИСТЕНТА ФІНАНСОВОЇ СВІДОМОСТІ ТА ЗАОЗАДЖЕНЬ

2.1 Обґрунтування вибору шляхів, технологій, алгоритмів і засобів вирішення поставленого завдання

У цьому розділі детально розглядаються архітектурні рішення, обґрунтовується вибір технологічного стеку та описується практична реалізація ключових компонентів програмного продукту.

В основі системи лежить гібридна архітектура, що поєднує елементи мікросервісного підходу та моноліту [16]. Ядро системи, що відповідає за основну бізнес-логіку, роботу з даними та користувацький інтерфейс, реалізовано як єдиний сервіс на мові Go. Допоміжні, але функціонально відокремлені задачі, такі як інтелектуальний аналіз даних, винесені в окремі сервіси.

Такий підхід дозволив досягти кількох цілей:

– надійність та продуктивність: основний функціонал, написаний на Go, є компільованим, швидким та надійним;

– гнучкість та розширюваність: використання Python для ШІ-агента дозволило задіяти найкращі інструменти для цієї задачі, не ускладнюючи основний сервіс;

– ізоляція компонентів: кожен сервіс може працювати у власному Docker-контейнері та бути розгорнутим на різних фізичних машинах, що полегшує оновлення та масштабування окремих частин системи. Проте, для простішого розгортання в хмарному середовищі Fly.io, було показано підхід де в одному Docker-контейнері працюють усі сервіси.

Загальна архітектурна схема зображена на рисунку 2.1. Основний бекенд моноліт (Go) включає наступні компоненти:

– вхід в систему і його ядро, що виконує аутентифікацію та авторизацію користувачів (через інтеграцію з провайдерами, наприклад Google);

– основні CRUD операції, що дозволяють змінювати зміст імпортованих транзакцій, додавати нові банківські рахунки, переглядати сповіщення, змінювати налаштування для користувача тощо;

– генерацію інтерактивних HTML сторінок (SSR);

– інтеграція з банками для отримання користувацьких фінансових даних;

– інтеграція з Telegram для надсилання сповіщень та реалізує чат з фінансовим помічником (ШІ-агент);

– фонові процеси (воркери), які періодично завантажують транзакції з банків, перевіряють досягнення фінансових цілей, аналізують витрати користувачів;

– виконання основних операції з базою даних Sqlite3 та підтримка міграцій.

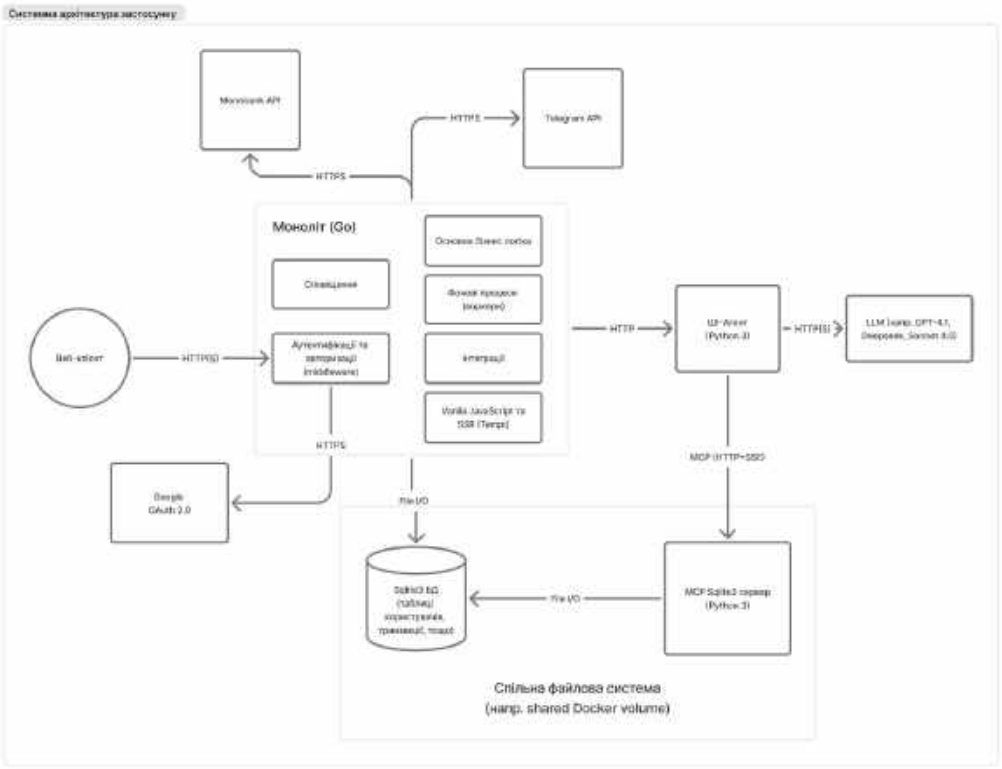

Рисунок 2.1 – Діаграма системної архітектури

Важливим компонентом є інтелектуальний ШІ-агент (Python): сервіс для обробки запитів природною мовою, який, користуючись інструментами (такими

як доступ до БД через MCP сервер), може приймати самостійно рішення про те, які дані йому потрібно дістати, щоб задовільнити запит (наприклад генерація і виконування SQL, щоб дістати потрібні транзакції за останній місяць).

ШІ-агент сервіс взаємодіє із сервісом доступу до даних (Python): MCP сервер для безпечної взаємодії з базою даних для ШІ-агента.

База даних SQLite: вбудована (embedded) файлова база даних для зберігання всієї інформації. Файлове дерево кінцевого проєкту на рисунку 2.2.

```
├── agent/
│   # AI-агент який відповідає на питання користувачів та допомагає з фінансовим плануванням.
├── mcp/
│   └── sqlite/
│       # MCP сервер для взаємодії з SQLite3 базою даних.
├── cmd/
│   # Стандартний каталог для Go проєктів, де кожна піддиректорія є
│   # точкою входу (main package) для компіляції в окремий виконуваний файл.
│   ├── regmonobank/main.go # Скрипт для реєстрації в корпоративний Monobank API.
│   └── server/main.go      # Основна точка входу для запуску головного веб-сервера.
│
├── internal/
│   # Компілятор Go забороняє імпортувати пакети з internal/ в інші проєкти, що забезпечує інкапсуляцію.
│   ├── db/
│   │   # Пакет для всіх операцій з базою даних. Містить SQL міграції,
│   │   # Go-код для виконання запитів (CRUD) та тести для них.
│   │   └── migrations/     # SQL-скрипти для зміни структури БД (створення таблиць, колонок).
│   │
│   ├── service/
│   │   # Шар бізнес-логіки. Цей код координує роботу між даними (db) і
│   │   # зовнішнім світом (web). Наприклад, transactions.go
│   │   # містити логіку валідації та обробки транзакцій.
│   │
│   ├── web/
│   │   # Відповідає за все, що пов'язано з HTTP: роутинг, обробники запитів,
│   │   # middleware, а також рендеринг HTML.
│   │   ├── assets/         # Статичні файли: CSS, JS, зображення, що віддаються клієнту.
│   │   └── views/          # Шаблони для генерації HTML.
│   │       # .templ файли - це сучасний підхід до створення типізованих HTML-шаблонів
│   │       # які компілюються в ефективний Go-код (*_templ.go).
│   │
│   ├── workers/
│   │   # Довготривалі фонові процеси (воркери).
│   │   ├── txpuller.go     # Періодично завантажує транзакції з банку.
│   │   ├── goalchecker.go  # Перевіряє досягнення фінансових цілей.
│   │   └── analytics.go    # Розраховує аналітику або збирає статистику.
│   ├── monobank/           # Ізольований пакет для взаємодії з API Monobank.
│   └── notifier/           # Система сповіщень (Telegram, логування в консоль тощо).
├── go.mod                  # Визначає модуль проєкту Go, його версію та список прямих і непрямих залежностей.
├── go.sum                  # Містить контрольні суми залежностей, щоб точно відтворити збірку.
│
├── Makefile                # Автоматизує рутинні завдання: make build, make run, make test.
│                           # Дуже зручно для стандартизації команд розробки.
│
├── Dockerfile              # Інструкції для збірки основного Go-додатку в Docker-контейнер (для розробки).
├── Dockerfile.prod         # Dockerfile для production: містить усі залежності програми в одному контейнері.
├── docker-compose.yaml     # Дозволяє одним файлом описати та запустити всі сервіси проєкту
│                           # для локальної розробки в окремих контейнерах.
├── fly.toml                # Файл конфігурації для розгортання (деплою) проєкту на хостинг-платформі Fly.io.
└── tailwind.config.js      # Файл конфігурації для Tailwind CSS, популярного CSS-фреймворку.
```

Рисунок 2.2 – Загальна файлова структура проєкту

Вибір технологічного стеку був продиктований вимогами до продуктивності, надійності, приватності даних та швидкості розробки.

Тип застосунку (веб-додаток): рішення про реалізацію системи у вигляді веб-застосунку було прийнято на основі кількох переваг. По-перше, це забезпечує максимальну крос-платформеність, оскільки доступ можливий з будь-якого пристрою з веб-браузером. По-друге, це спрощує розповсюдження та оновлення, оскільки немає потреби в публікації у маркетах додатків (App Store, Google Play). Ключовим аргументом стала простота розгортання та можливість самохостингу (self-hosting), що забезпечує користувачу повний контроль над своїми даними.

Мова Go обрана для реалізації основного бекенд сервісу. Його висока продуктивність, ефективна модель паралелізму (горутини) та статична типізація роблять його ідеальним для написання надійних мережевих служб. Горутини [15] активно використовуються у фонових процесах (internal/workers) для неблокуючої синхронізації даних з Monobank API. Компіляція в єдиний бінарний файл спрощує розгортання в Docker-контейнері.

Templ бібліотека для генерації HTML-шаблонів на стороні сервера в Go дозволила уникнути складності, пов'язаної з розробкою окремого frontend-застосунку (SPA). Підхід Server-Side Rendering (SSR) забезпечує швидке завантаження сторінок та спрощує розробку, оскільки вся логіка рендерингу знаходиться в межах Go-сервісу.

SQLite використовується як основна СУБД. Для персонального фінансового асистента, де пріоритетом є приватність та контроль над даними, файлова база даних є ідеальним рішенням. Вона не потребує окремого сервера, легко переноситься та резервується. Це усуває складність адміністрування та знижує витрати на хостинг.

Технологія Docker для контейнеризації використовується для пакування кожного сервісу (Go, Python) з усіма його залежностями. Файл docker-compose.yaml описує, як запустити всі сервіси разом, що гарантує відтворюваність середовища та значно спрощує розгортання системи.

2.2 Практична реалізація об'єкта проєктування

Основою для зберігання даних слугує реляційна схема, реалізована в SQLite. Структура була спроєктована для ефективного збереження інформації про користувачів, їхні фінансові операції та налаштування. Вона включає наступні ключові таблиці (рис. 2.3):

– «users» – основна таблиця, що зберігає дані про зареєстрованих користувачів. Включає унікальний ідентифікатор (id, отриманий від провайдера автентифікації), email, роль в системі (role), а також опціональний telegram_id для сповіщень;

– «user_bank_accounts» – таблиця для керування інтеграціями з банківськими установами. Для кожного користувача тут зберігається зашифрований токен доступу до API банку (access_token), назва банку (напр., «monobank») та список підключених карток/рахунків у форматі JSON;

– «transactions» – центральна та найбільш деталізована таблиця системи. Вона акумулює всі фінансові транзакції користувача. Кожен запис містить повну інформацію про операцію: суму, валюту, MCC-код, опис, а також службові поля, такі як джерело (source), унікальний ідентифікатор транзакції (tx_id) та прапорці (напр., is_refund, is_ignored). Важливою є колонка class, що дозволяє класифікувати витрати на «потреби», «бажання» та «заощадження»;

– «user_goals» та «user_income» – допоміжні таблиці для відстеження фінансових цілей та доходів користувача відповідно. Вони пов'язані з таблицею «users» і дозволяють реалізувати логіку бюджетування та аналізу фінансового стану;

– «notifications» – таблиця для зберігання сповіщень для користувачів. Кожне сповіщення має рівень важливості (severity), текст та статус (прочитано або не прочитано).

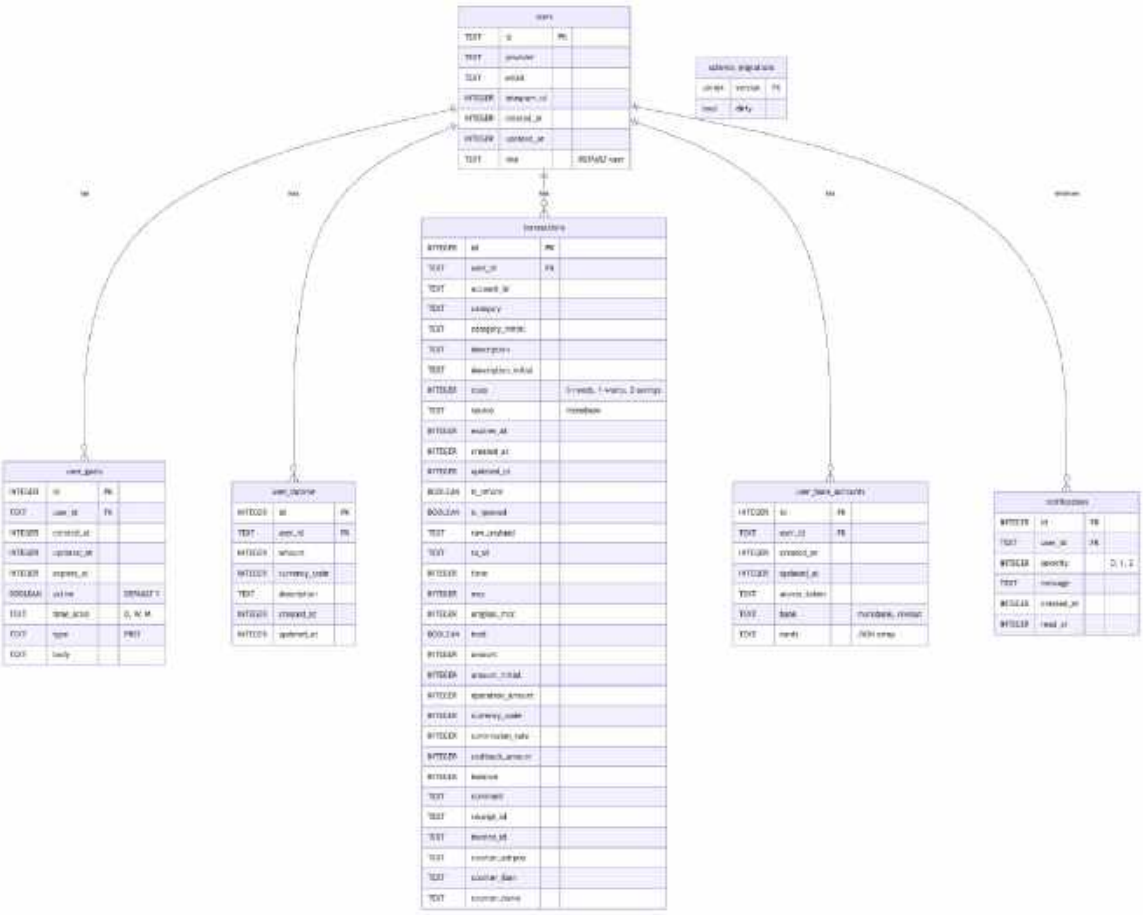

Рисунок 2.3 – Детальна ER-діаграма бази даних

Автентифікація в системі реалізована за допомогою протоколу OAuth 2.0, що дозволяє користувачам входити через сторонніх провайдерів, наприклад, Google. Для спрощення інтеграції була використана бібліотека goth [17], яка абстрагує більшу частину логіки взаємодії з провайдерами.

Процес автентифікації та керування сесіями виглядає наступним чином:

– ініціалізація входу: користувач на сторінці «login» ініціює процес автентифікації, переходячи за посиланням «/auth/{provider}» (напр., /auth/google). Обробник «HandleProviderLogin» за допомогою «gothic.BeginAuthHandler» перенаправляє користувача на сторінку згоди відповідного провайдера;

– обробка зворотного виклику: після успішної автентифікації у провайдера, користувач перенаправляється на ендпоінт «/auth/{provider}/callback». Обробник «HandleAuthCallback» виконує ключову

логіку. За допомогою «gothic.CompleteUserAuth» він отримує дані профілю користувача від провайдера (ID, email тощо). Система перевіряє, чи існує користувач з таким ID у локальній базі даних (h.users.GetUser). Якщо користувач новий, система створює для нього запис у таблиці users. На цьому етапі відбувається визначення ролі: функція «determineRole» перевіряє, чи збігається email користувача з одним із email-адрес, вказаних у змінній середовища «APP_ADMIN_EMAILS». Якщо так, користувачу надається роль адміністратора (AdminRole), інакше – стандартна роль користувача (UserRole). Якщо користувач вже існує, його дані та роль завантажуються з бази даних;

– створення сесії: після успішної ідентифікації (або реєстрації) користувача, сервіс «h.auth.StoreUserSession» створює для нього сесію. goth використовує сесійне сховище на основі кукі (cookie-based store), записуючи зашифровані дані сесії в безпечну (HttpOnly) кукі у браузері клієнта. Це дозволяє ідентифікувати користувача при наступних запитах;

– для виходу з системи передбачено обробник «HandleLogout», який за допомогою «gothic.Logout» та «h.auth.DeleteUserSession» видаляє дані сесії зі сховища та очищує відповідну кукі в браузері, після чого перенаправляє користувача на сторінку входу.

Після автентифікації користувач може додати свій токен доступу до Monobank API. Цей токен шифрується та зберігається в базі даних. Фоновий процес «txpuller» періодично, використовуючи цей токен, звертається до Monobank API, запитує нові транзакції за останній період, перетворює їх у внутрішню модель (internal/models) та зберігає в базу даних.

Водночас, у проєкті закладено фундамент для переходу на більш просунутий корпоративний API Monobank. Для цього було розроблено скрипт «cmd/regmonobank/main.go», який реалізує процес разової реєстрації самого застосунку в Monobank. Цей процес використовує криптографію на основі пари ключів (secp256k1) для підпису запитів, як того вимагає офіційна документація [18]. У майбутньому якщо користувачі не бажатимуть самохостити застосунок, це дозволить нам як постачальника послуг (SaaS) [19] відмовитись

від збору персональних токенів і перейти на безпечніший механізм, де користувачі надаватимуть доступ застосунку до своїх даних через стандартизований процес авторизації.

Ключовий інноваційний компонент системи це штучний інтелект, який може приймати автономні рішення без втручання людини – ШІ-агент. Ті завдання, які сьогодні передаються на аутсорсинг підрядникам або третім сторонам, все частіше виконуватимуться агентним штучним інтелектом [29]. Приклади використання в категорії «Управління капіталом»:

– поради щодо збережень;

– віртуальний фінансовий асистент;

– планування та розрахунок пенсії і податків;

– автоматизація рутинних фінансових операцій;

– виявлення підозрілої активності (fraud detection);

– аналіз ризиків.

ШІ-агент реалізований як окремий FastAPI-сервер на Python і його робота складається з кількох етапів:

– отримання запиту: асистент приймає текстовий запит від користувача через ендпоінт «/query», що містить питання та ідентифікатор користувача (user_id);

– побудова SQL-запиту: використовуючи бібліотеку LangChain [20], агент, що має доступ до схеми бази даних та набору інструментів, перетворює запит користувача на безпечний SQL-запит якщо цього вимагає користувацький запит. Системний промпт [21] налаштовує агента на генерацію запитів лише для читання (SELECT) та обов'язкове включення умови «WHERE user_id = ?»;

– валідація запиту: перед виконанням, кожен згенерований SQL-запит проходить додаткову перевірку за допомогою регулярного виразу. Ця перевірка гарантує, що кожна «WHERE» умова в запиті містить правильний «user_id» поточного користувача. Це є критично важливим механізмом безпеки для запобігання доступу до даних інших користувачів;

– виконання запиту через MCP сервер: згенерований та валідований SQL-

запит не виконується напряму. Замість цього, він надсилається до сервісу доступу до даних (mcp/sqlite), який єдиний має прямий доступ до файлу БД. Цей MCP сервер виконує запит;

– формування відповіді: отримавши дані з БД, велика мовна модель (напр., GPT-4 [22]) форматує їх у зрозумілу для людини відповідь і повертає користувачеві.

Така архітектура дозволяє користувачам ставити складні аналітичні питання («Які мої топ-5 категорій витрат за минулий місяць?»), значно розширюючи можливості стандартного інтерфейсу, зберігаючи при цьому високий рівень безпеки та ізоляції даних.

Важливою складовою, що забезпечує коректну та безпечну роботу ШІ-агента, є проєктування системного промпту (prompt engineering). Якість SQL-запиту, згенерованого мовною моделлю, напряму залежить від чіткості та повноти інструкцій, наданих у промпті. Ефективний промпт [23] повинен надавати моделі контекст, визначати її роль та встановлювати чіткі обмеження.

У рамках даного проєкту системний промпт для LangChain-агента був ретельно розроблений і включав у себе наступні елементи:

– визначення ролі: моделі надається інструкція діяти як «компетентний SQL-аналітик, що працює з діалектом SQLite»;

– надання контексту: у промпт міститься повна і точна схема бази даних. Це дозволяє моделі «знати», які таблиці та колонки доступні для запитів;

– встановлення правил та обмежень: промпт містить низку суворих правил: генерувати виключно запити на читання (SELECT), не намагатися змінювати дані (UPDATE, DELETE), не створювати фіктивних даних та завжди перевіряти наявність колонок перед їх використанням у запиті;

– приклади (Few-shot prompting): для складних випадків у промпт були додані приклади пар «запит користувача – правильний SQL-запит», що допомагає моделі краще зрозуміти очікуваний результат.

Процес створення ефективного промпту був ітеративним. Початкові версії іноді призводили до генерації неоптимальних або синтаксично невірних запитів.

Шляхом поступового уточнення інструкцій, додавання обмежень та аналізу помилок вдалося досягти високої точності та надійності роботи ШІ-агента, що підкреслює критичну важливість prompt engineering у сучасних AI-системах.

В основі компоненту сповіщень лежить гнучка архітектура, побудована навколо інтерфейсу «Notifier» (нотифікатор). Такий підхід дозволяє легко додавати нові канали доставки сповіщень. Система включає кілька реалізацій:

– «DBNotifier» – зберігає кожне сповіщення в таблиці «notifications» бази даних SQLite. Це дозволяє відображати історію сповіщень безпосередньо у веб-інтерфейсі застосунку;

– «TelegramNotifier» – інтегрується з Telegram Bot API. Якщо користувач прив'язав свій Telegram акаунт, цей сервіс надсилає йому сповіщення у вигляді миттєвих повідомлень, забезпечуючи оперативне інформування;

– «StdoutNotifier» – виводить сповіщення в стандартний потік виводу. Цей канал використовується переважно для цілей розробки та налагодження;

– «CombinedNotifier» – реалізує патерн «Компонувальник» (Composite). Він об'єднує кілька інших нотифікаторів і дозволяє одночасно надіслати одне сповіщення через усі підключені канали (наприклад, зберегти в БД та відправити в Telegram). Саме цей композитний нотифікатор використовується фоновими процесами.

Фоновий процес «goalchecker» є ключовим компонентом проактивної фінансової аналітики. Його єдина функція – це аналіз даних та виклик сервісу notifier для відправки сповіщень. Такий підхід реалізує механізм контролю за дотриманням бюджету завдяки аналізу дотримання користувачем процентних бюджетних цілей (PercentageGoalType), а не відстеження накопичувальних цілей. Цей тип цілі дозволяє встановити ліміт витрат на певну категорію як відсоток від місячного доходу (наприклад, «витрачати не більше 15 % в категорії розваги»). Схема таблиці цілей та бізнес логіка є гнучкими та відкритими до розширень, таких як додавання нових типів «цілей». Воркер працює за двома різними розкладами: щомісяця та щодня, надаючи як ретроспективний аналіз, так і попередження в реальному часі.

Щомісячний аналіз (ProcessPercentageGoalsMonthly) запускається на початку кожного нового місяця для аналізу попереднього:

– отримання даних: для кожного користувача, що має активні місячні цілі, воркер запитує з бази даних його останній відомий дохід та повну історію транзакцій за минулий місяць;

– розрахунок фактичних витрат: система обчислює сумарні витрати по кожній категорії та розраховує, який відсоток від доходу вони склали;

– порівняння з ціллю: фактичний відсоток витрат порівнюється з цільовим відсотком, встановленим користувачем;

– генерація сповіщення: на основі порівняння генерується підсумкове сповіщення. Якщо ліміт перевищено, користувач отримує сповіщення з рівнем «Alert»: «Минулого місяця ви витратили 20 % (6000 ₴) на ресторани – це на 5 % більше за мету в 15 %». Якщо користувач дотримався ліміту, надсилається інформаційне сповіщення: «Минулого місяця ви витратили 12 % (3600 ₴) на ресторани, що менше за вашу мету в 15 %».

Щоденний аналіз (ProcessPercentageGoalsDaily) щодня для моніторингу поточних тенденцій та раннього попередження:

– отримання даних: воркер збирає дані про витрати в поточному місяці (до поточного дня) та за весь попередній місяць;

– аналіз тенденції: для кожної категорії, на яку встановлено ціль, система порівнює суму, витрачену на даний момент, із загальною сумою за попередній місяць;

– генерація попереджувального сповіщення: алгоритм формує сповіщення, яке допомагає користувачеві зрозуміти темп своїх витрат. Рівень важливості сповіщення (Info, Warn, Alert) залежить від того, наскільки швидко користувач наближається до показників минулого місяця. Приклад сповіщення рівня «Warn»: «Ви витратили 4500 ₴ на розваги цього місяця – це майже стільки ж, скільки за весь минулий місяць (5000 ₴), а попереду ще 10 днів»;

– важливою частиною фінансового асистента є його здатність не лише реагувати на запити користувача, але й проактивно аналізувати фінансову

ситуацію та надавати корисні поради чи попередження. Ця функціональність реалізована через модульну систему сповіщень та набір аналітичних фонових процесів. Аналітичний фоновий процес (internal/workers/analytics.go) на відміну від «goalchecker», який відстежує дискретні цілі, аналітичний воркер виконує періодичний аналіз загальних фінансових звичок користувача. Він запускається щодня і генерує сповіщення на основі двох ключових метрик:

– аналіз топ-категорій витрат (TopSpendingCategories): воркер аналізує 5 найбільших категорій витрат за поточний місяць. Для кожної такої категорії він розраховує середньоденну швидкість витрат і, якщо до кінця місяця залишився час, прогнозує орієнтовну суму майбутніх витрат. Якщо темп витрат високий, система генерує інформаційне сповіщення, наприклад: «Ви витратили 2500 ₴ на ресторани цього місяця, в середньому витрачаючи 125 ₴ на день. Якщо продовжите, то витратите ще 1500 ₴ до кінця місяця»;

– аналіз темпу витрат (SpendingRate): ця функція порівнює медіанні щоденні витрати користувача з його місячним доходом. На основі цього розраховується прогнозований залишок бюджету на кінець місяця. Якщо прогнозується перевитрата, система надсилає попередження: «Ви перевищили свій бюджет на 500 ₴ (15500 ₴/15000 ₴)». В іншому випадку надсилається інформаційне повідомлення з прогнозом залишку.

Таким чином, поєднання гнучкої системи сповіщень та аналітичних воркерів перетворює застосунок з пасивного інструменту обліку на активного фінансового помічника.

Головна сторінка є центральним елементом взаємодії користувача з системою. Вона надає зведену інформацію про поточний фінансовий стан (рис. 2.4).

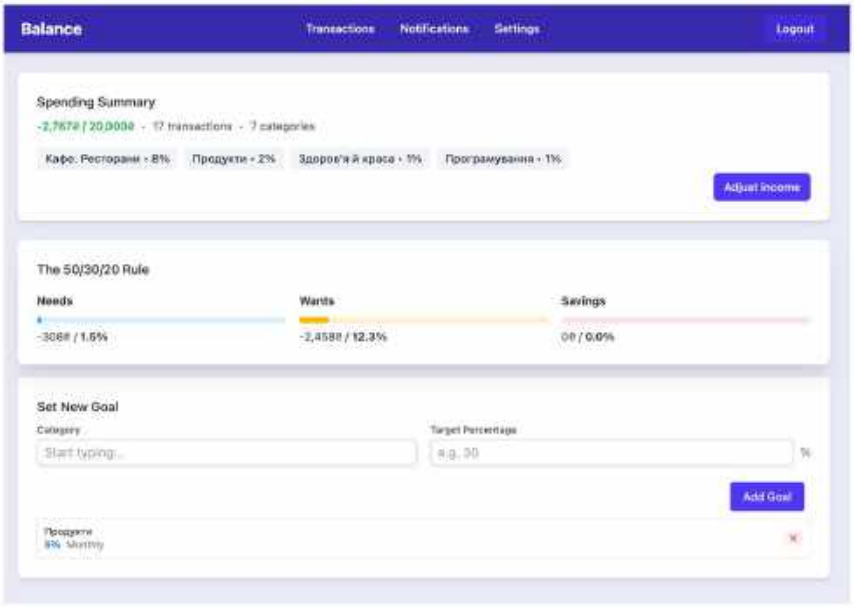

Рисунок 2.4 – Головна сторінка

Фінансові підсумки: на початку сторінки розташована панель «Spending Summary», що показує у відсотковому співвідношенні витрати відносно доходу. Оскільки точно визначити дохід завдяки лише транзакціям не завжди можливо, було прийнято рішення надати користувачу можливість самотужки вказувати дохід: для цього панель містить кнопку «Adjust income», що відправляє на сторінку «/setup/income» із формою для зміни доходу (рис. 2.5).

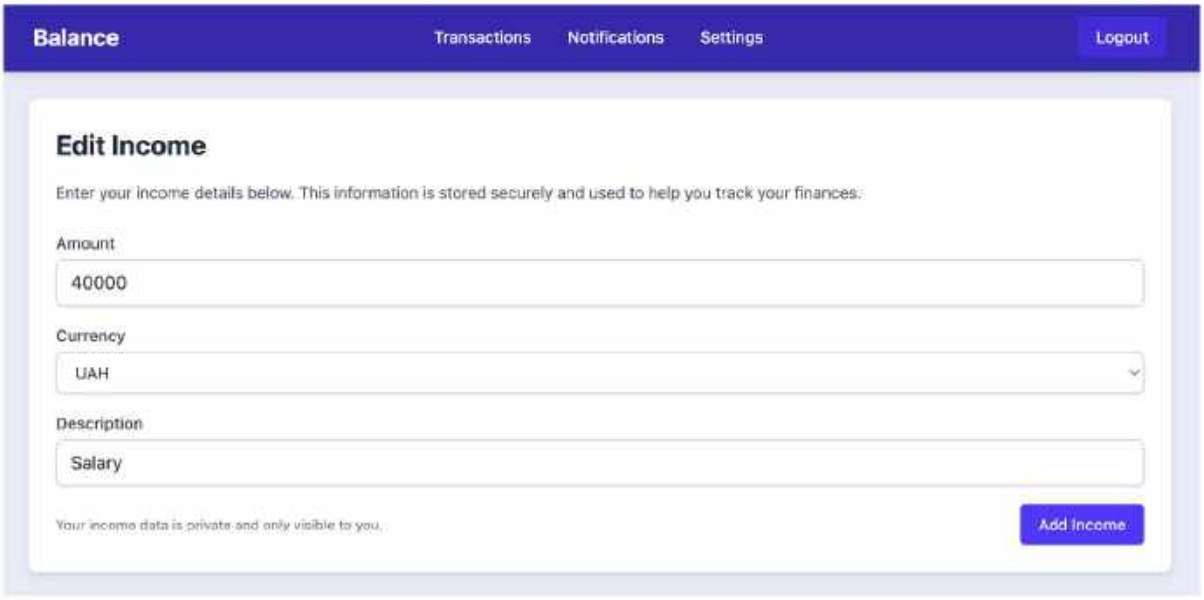

Рисунок 2.5 – Форма зміни доходу

Аналітика «50/30/20»: по середині сторінки розташована панель «The 50/30/20 Rule», що візуалізує розподіл витрат за поточний місяць відповідно до бюджетного правила «50 % на потреби, 30 % на бажання, 20 % на заощадження». Це дозволяє користувачу миттєво оцінити, наскільки його фінансова поведінка відповідає рекомендованій структурі.

Фінансові цілі: нижче відображається панель фінансових цілей «Set New Goal». Панель містить: HTML форму для створення фінансових цілей та список активних фінансових цілей встановлених користувачем із можливістю їх видалення.

Сторінка транзакцій надає деталізований огляд усіх фінансових операцій користувачів. На цій сторінці відображається повний список транзакцій, завантажених з банківського API, з інформацією про дату, суму, категорію та опис (рис. 2.6). На сторінці існує пагінація за датою, щоб пришвидшити пошук транзакцій.

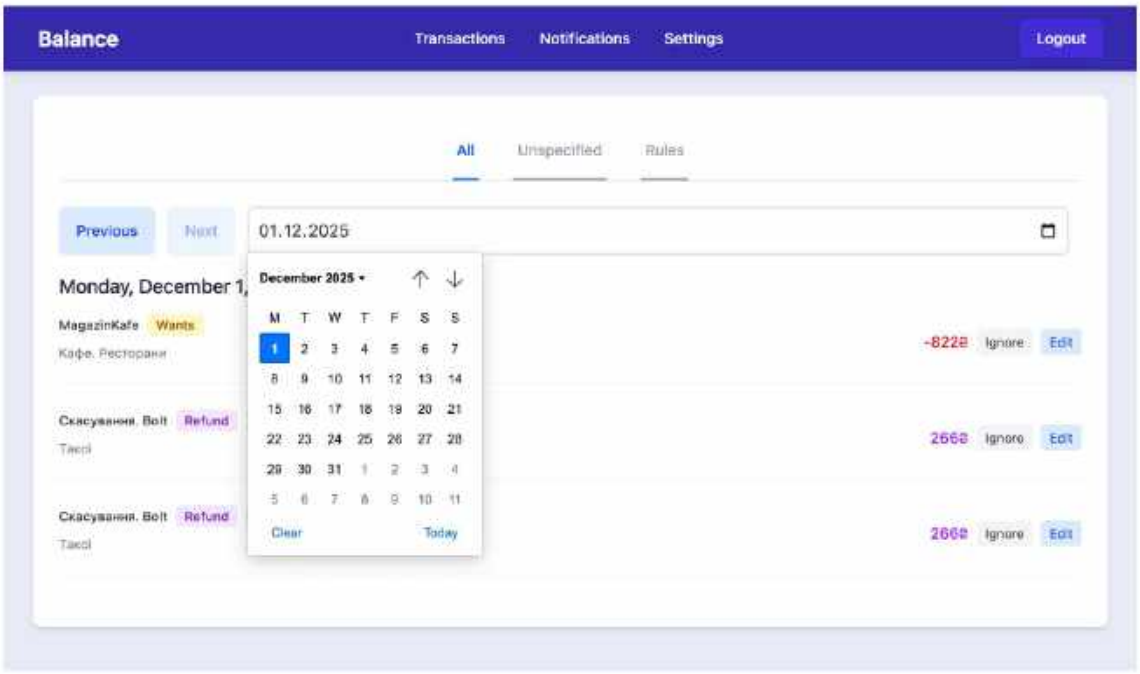

Рисунок 2.6 – Сторінка «Транзакції»

Якщо перейти на вкладку «Unspecified» (рис. 2.7) відкриється сторінка з усіма транзакціями категорія яких є неточною (наприклад, «Переказ коштів» або

«Каси»). Ця сторінка існує щоб було зручніше знаходити такі транзакції та власноруч змінювати їх категорії.

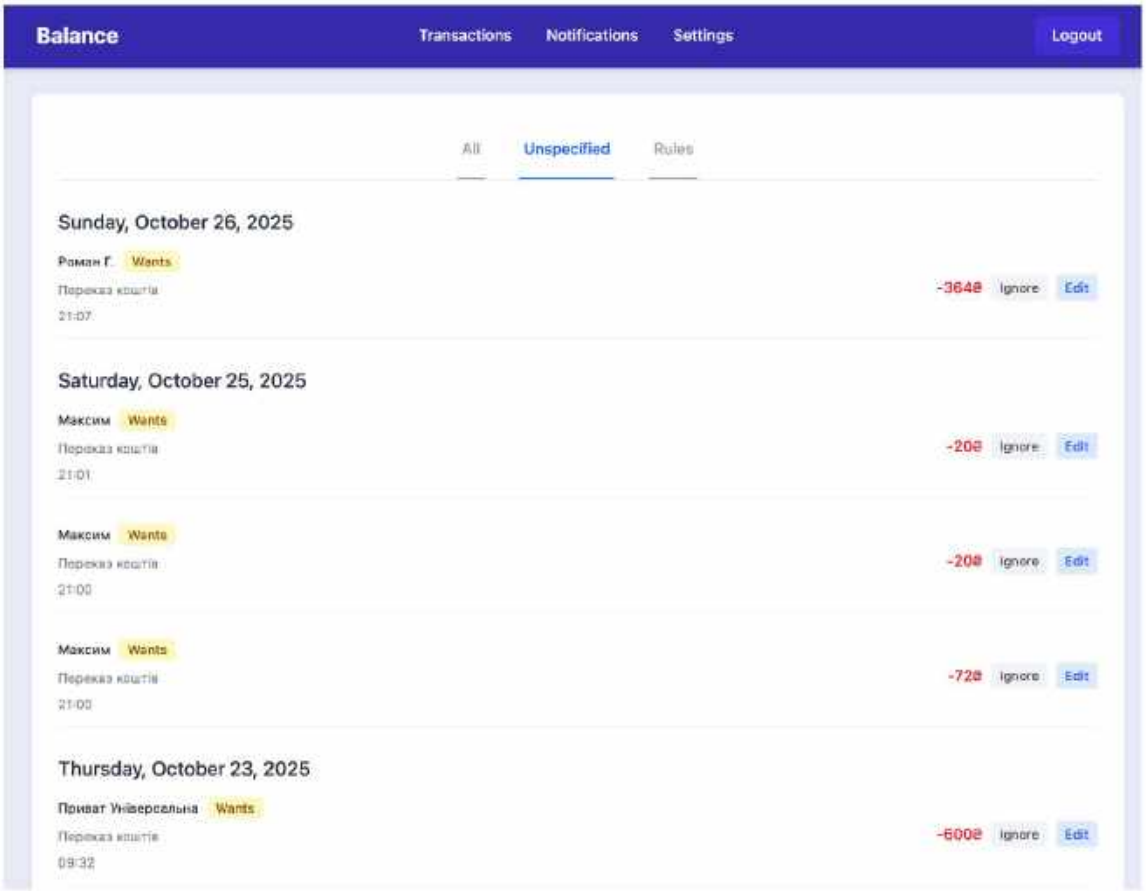

Рисунок 2.7 – Вкладка «Unspecified » сторінки «Транзакції»

Зміна деталей транзакції (рис. 2.8): користувач має можливість вручну змінити суму, дату, опис, назву та клас кожної із транзакцій («потреби», «бажання» та «заощадження»). Для зручного пошуку категорії в полі «Category», було реалізовано пошук за наближеною відповідність рядків (fuzzy search). Також користувач може «ігнорувати» будь-які транзакції, таким чином система не бачитиме ці транзакції при аналітичних обрахунках (наче їх ніколи і не було).

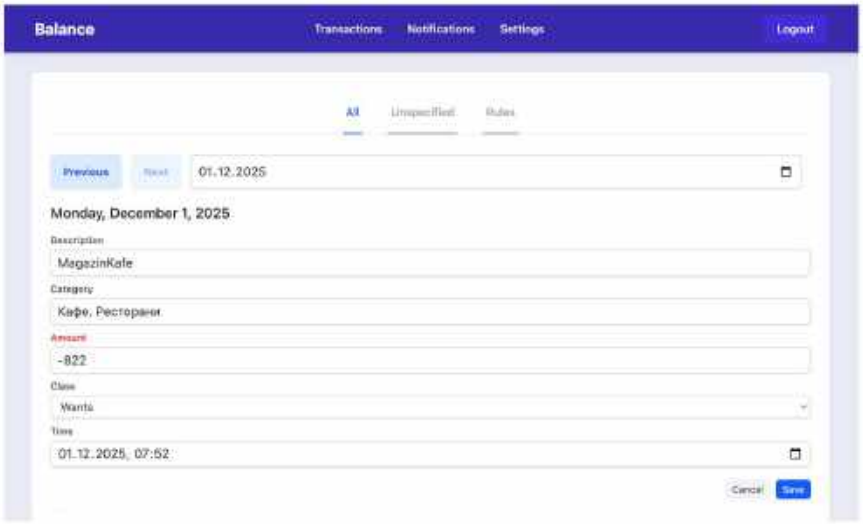

Рисунок 2.8 – Форма зміни деталей транзакції

Сторінка налаштувань та підключення Monobank дозволяє користувачу керувати своїм профілем та інтеграціями. Вкладка Профіль (Profile) (рис. 2.9): відображається інформація про поточного користувача (email, роль). На цій вкладці присутня кнопка «Delete All My Data», це важливий механізм контролю за даними користувача, що реалізує нефункціональну вимогу «приватність та контроль за даними». При натисканні та підтверджені, усі дані користувача будуть видалені із бази даних.

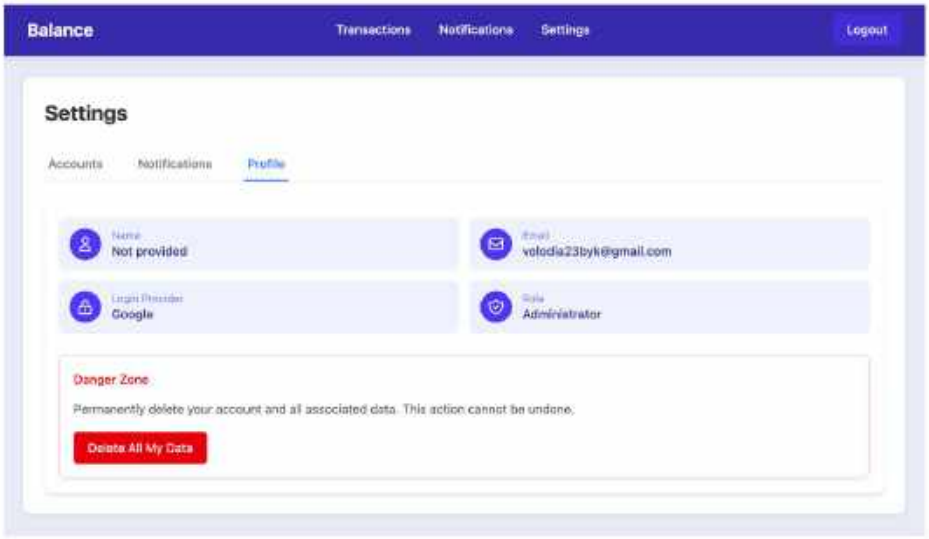

Рисунок 2.9 – Вкладка «Profile»

Вкладка сповіщення (Notifications): надає можливість налаштувати канали для сповіщень (рис. 2.10). В поточному прототипі реалізовані сповіщення через Telegram. Після підключення до бота сповіщень, користувач отримуватиме повідомлення за його цілями та аналітикою. Підключення до бота також надає можливість спілкуватися із ним як із розумним фінансовим помічником (ШІ-агент).

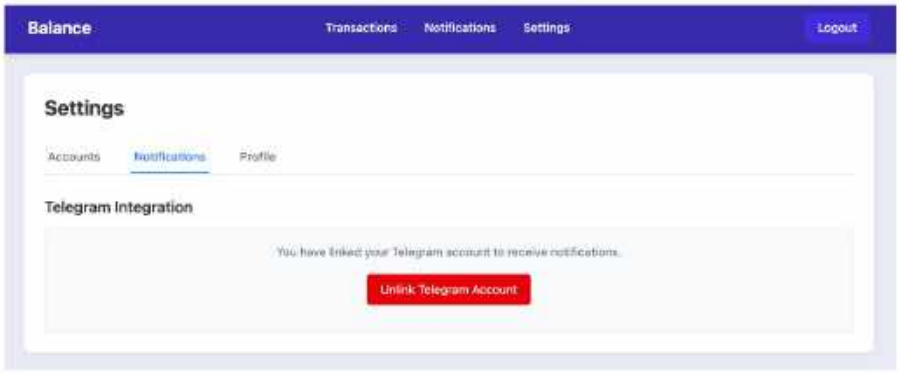

Рисунок 2.10 – Вкладка «Notifications»

Вкладка акаунти (Accounts): тут користувач бачить усі свої синхронізовані банківські рахунки. Є можливість додати нові рахунки кнопкою «Connect New», або від'єднати існуючі «Disconnect» (рис. 2.11). Якщо всі рахунки будуть від'єднанні, тоді система запропонує додати один, перш ніж надати доступ до інших функцій додатку (рис. 2.12).

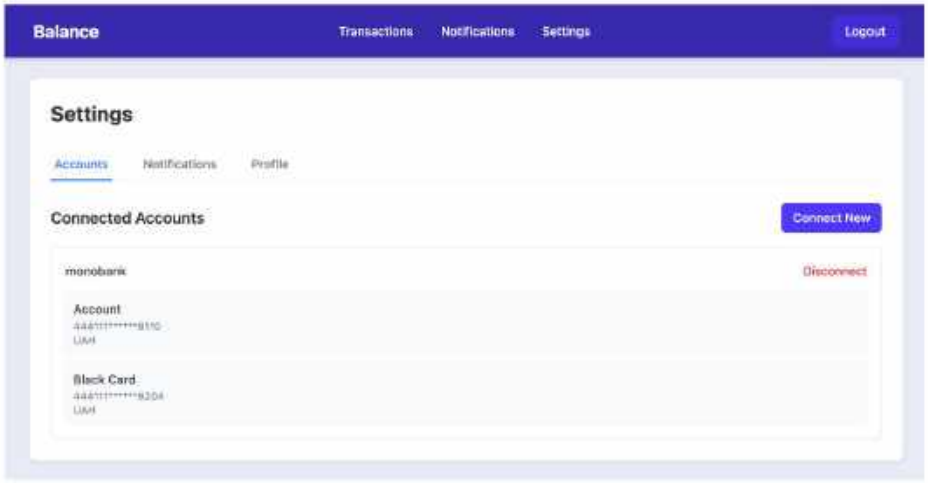

Рисунок 2.11 – Вкладка «Accounts»

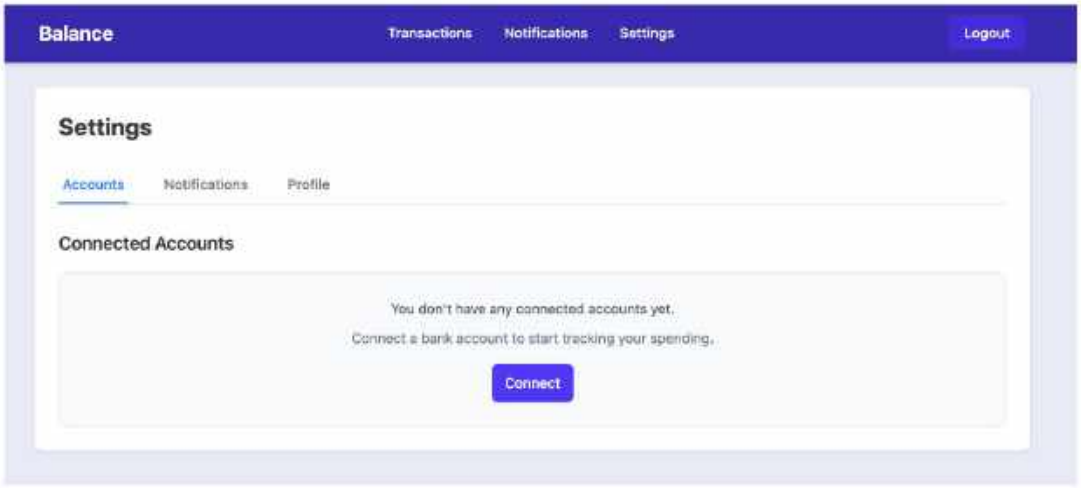

Рисунок 2.12 – Вкладка «Accounts» при відсутності підключених банківських рахунків

Підключення Monobank акаунту – це ключова функція налаштувань, що дозволяє підключити банківський акаунт для автоматичного збору даних. Процес підключення виглядає наступним чином:

– отримання токену: користувач повинен спочатку отримати свій персональний токен доступу на офіційному сайті api.monobank.ua;

– ведення токену: На сторінці налаштувань у веб-інтерфейсі системи користувач вставляє отриманий токен у відповідне поле форми (рис. 2.13);

– збереження та шифрування: після збереження система не зберігає токен у відкритому вигляді. Він надійно шифрується перед записом у базу даних в таблицю «user_bank_accounts», що унеможливлює його компрометацію навіть у випадку несанкціонованого доступу до БД;

– валідація та перша синхронізація: система одразу робить тестовий запит до Monobank API для перевірки валідності токену та просить користувача вибрати рахунки для синхронізації (рис. 2.14). У випадку успіху, фоновий процес «txpuller» ініціює першу синхронізацію, завантажуючи історію транзакцій за останній та поточний місяць. Надалі синхронізація відбувається автоматично в фоновому режимі.

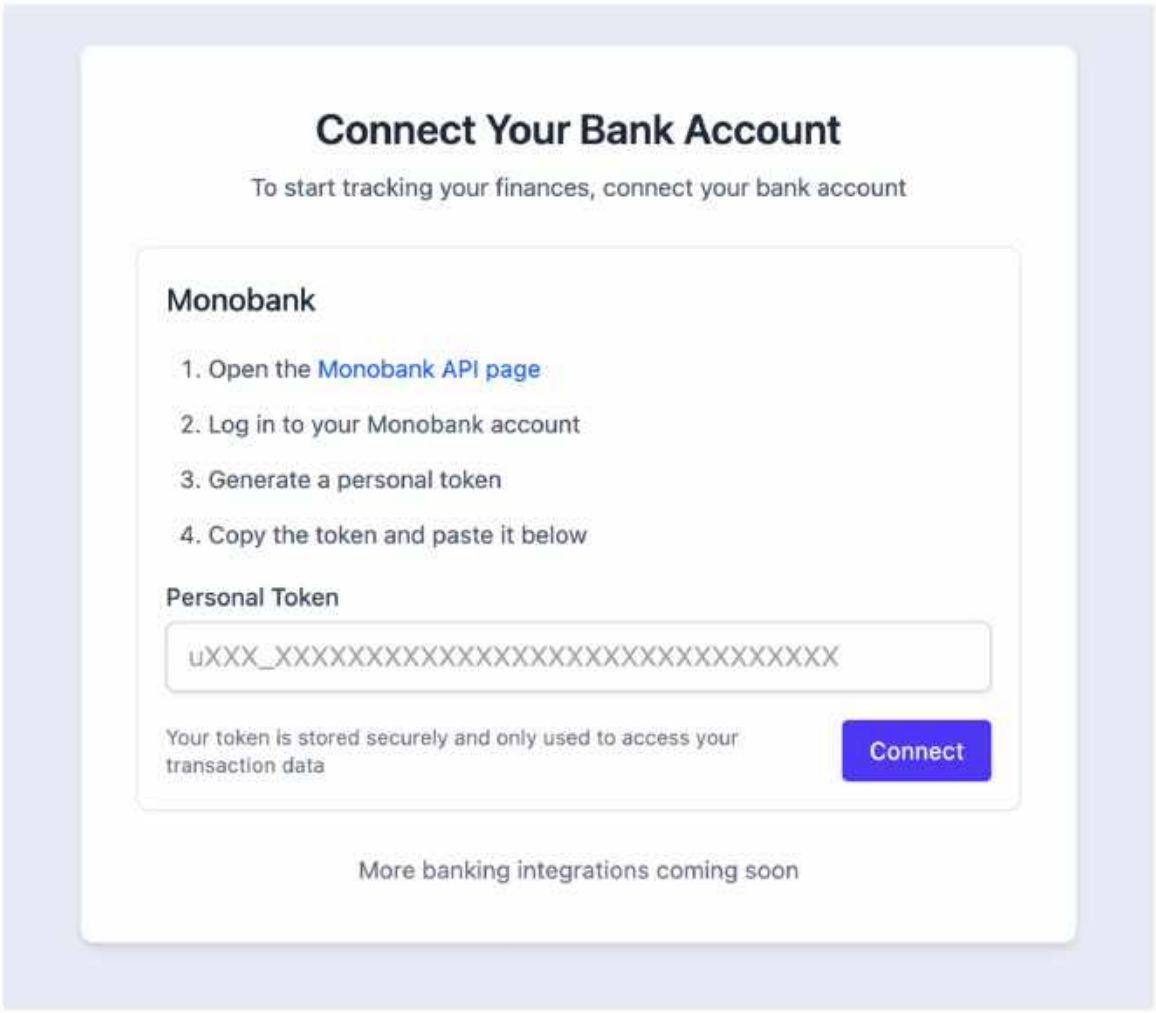

Рисунок 2.13 – Форма підключення Monobank рахунків

Рисунок 2.14 – Результати підтягнутих та затверджених користувачем рахунків

Сторінка сповіщень агрегує всі сповіщення, згенеровані системою, та слугує центральним хабом для комунікації. Користувач може переглянути у веб-інтерфейсі історію всіх отриманих повідомлень (рис. 2.15): про перевищення бюджету, успішне досягнення цілей, аналітичні поради тощо. Це дозволяє ретроспективно аналізувати фінансові звички.

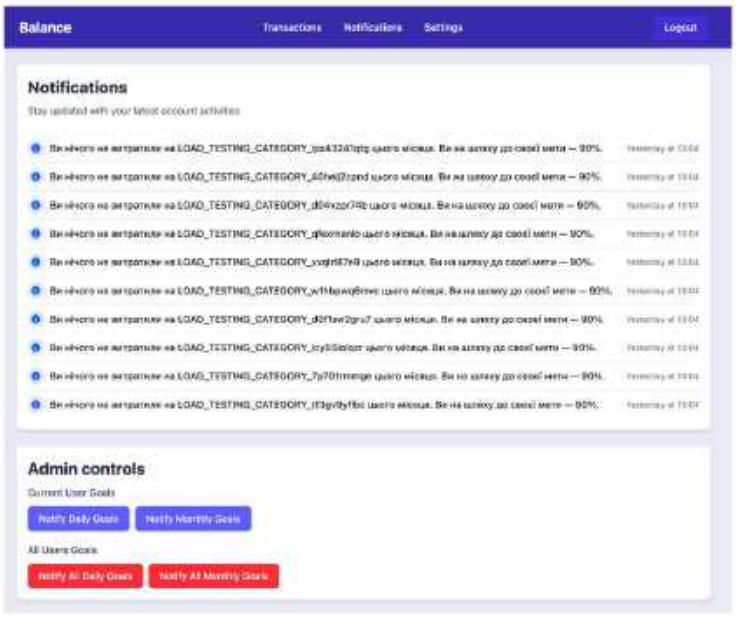

Рисунок 2.15 – Сторінка сповіщень «Notifications»

Ключовою особливістю є дублювання всіх важливих сповіщень у Telegram (рис. 2.16). Після прив'язки акаунту користувач отримує миттєві повідомлення від бота, що забезпечує оперативне інформування, навіть коли веб-застосунок не відкрито.

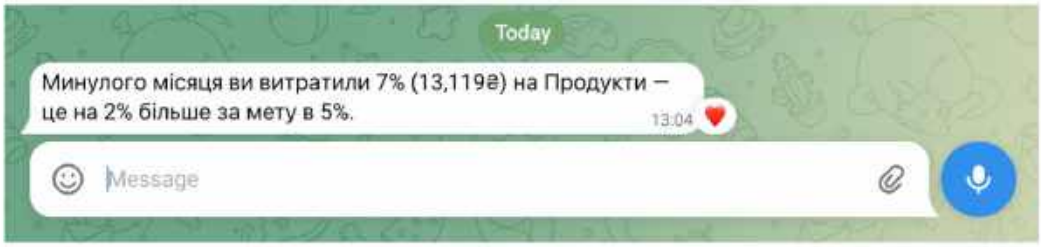

Рисунок 2.16 – Приклад повідомлення від чат-бота в Telegram

Реалізована можливість повноцінного спілкування з ШІ-агентом через Telegram-бота (рис. 2.17). Користувач може надсилати свої запити безпосередньо в чаті з ботом і отримувати відповіді там же, що робить процес аналізу фінансів ще більш мобільним та інтерактивним.

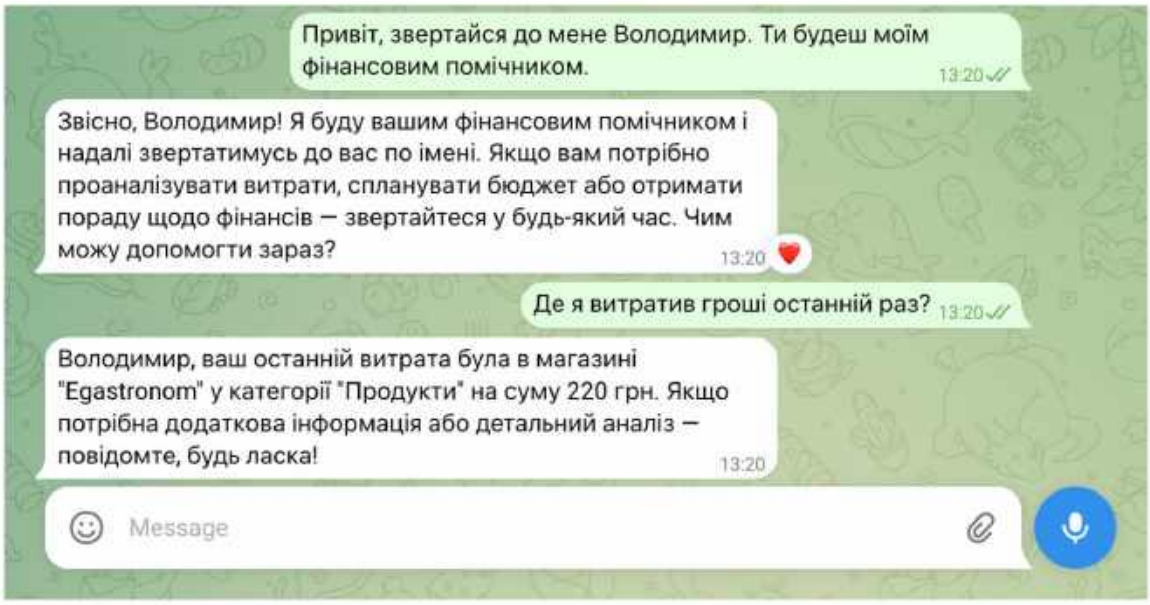

Рисунок 2.17 – Спілкування з фінансовим асистентом

РОЗДІЛ 3
ЕКСПЕРЕМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ РЕЗУЛЬТАТИВНОСТІ АСИСТЕНТА ФІНАНСОВОЇ СВІДОМОСТІ ТА ЗАОЗАДЖЕНЬ

3.1 Методика проведення дослідження

Метою експериментального дослідження є перевірка ефективності веб-платформи та аналіз того, як обрана архітектура впливає на продуктивність під навантаженням. Щоб гарантувати коректність функціонування продукту, проведено низку тестів, розділених на кілька етапів.

Для ключових компонентів бекенд сервісу на Go були написані модульні тести (файли що закінчуються на «_test.go»). Основна увага приділялася тестуванню чистої бізнес-логіки в «internal/service» (наприклад, розрахунок бюджету, логіка досягнення цілей) та функцій для роботи з базою даних в «internal/db», використовуючи in-memory базу даних SQLite для ізоляції тестів.

В модульному тестуванні, відомим показником є покриття коду тестами (Test Coverage), що демонструє який відсоток програмного коду виконується під час запуску тестів. Покриття вимірює кількість, а не якість тестів: тест може зайти у функцію, але не завжди перевірити її логіку. Також, один із пакетів може бути критично важливим, інший дріб'язковим, але покриття ставиться до них однаково. Тому в цій роботі було зроблено акцент на критичні компоненти системи, такі як бізнес логіка, робота із базою даних, безпека.

Розробка тестових сценаріїв у наскрізному тестуванні (End-to-End): було розроблено структурований набір з 13 тестових питань, що охоплюють різні аспекти аналізу фінансових даних. Питання були розділені на кілька категорій для забезпечення повного покриття функціональності:

– перевірка вміння агента запам'ятовувати;

– прості запити: базові агрегації даних (загальна сума, кількість) із часовими проміжками (минулий тиждень, цей місяць);

– категорійні запити: групування та сортування за категоріями витрат;

– складні та порівняльні запити: питання, що вимагають порівняння даних або більш складної логіки (наприклад, розрахунок середнього чеку);

– межові випадки (Edge Cases): перевірка поведінки системи на неоднозначних запитах або у випадках відсутності даних.

Процес тестування: кожне питання з набору послідовно надсилалося на ендпоінт «/query» сервісу ШІ-агента, імітуючи запит від реального користувача через чат-інтерфейс. Для кожного запиту аналізувалися два ключові артефакти (критерії оцінки):

– згенерований SQL-запит: перевірялася його синтаксична коректність, логічна відповідність поставленому питанню та, що найважливіше, наявність умов безпеки (фільтрація за user_id);

– фінальна текстова відповідь: оцінювалася її адекватність, зрозумілість для користувача та відповідність даним, що були б отримані при прямому виконанні SQL-запиту до бази даних.

Такий підхід дозволив не тільки перевірити правильність кінцевого результату, але й отримати уявлення про внутрішню логіку роботи агента.

Перед кожним основним тестуванням навантаження часто запускають швидкий димний тест (ліст. 3.1). Цей тест використовує k6 для швидкої перевірки доступності основних сторінок застосунку. Він імітує одного користувача, який послідовно заходить на кожну сторінку з масиву «getPages» і перевіряє, що сервер повертає успішний статус відповіді. Це дозволяло швидко виявити критичні помилки в конфігурації або розгортанні ще до початку тривалих тестів продуктивності.

Лістинг 3.1 – Димний тест (smoke testing)

```
import http from "k6/http";
import { group } from "k6";
import { config } from "./config.js";
import { checkStatus, getAuthParams, getPages } from "./common.js";

// This is a simple smoke test for a single user to verify key endpoints
are working.
export const options = {
```

```
  vus: 1,
  iterations: 1,
};

const authParams = getAuthParams();

export default function () {
  group("Smoke Test - Verify key GET endpoints", function () {
    for (const endpoint of getPages) {
      const res = http.get(`${config.host}${endpoint}`, authParams);
      checkStatus(res, `GET ${endpoint}`);
    }
  });
}
```

Кінець лістингу 3.1

Для оцінки швидкодії та стабільності системи під навантаженням у відповідність до поставлених нефункціональних вимог в першому розділі було використано інструмент k6 [25]. Скрипт «tests/load.js» містить сценарій, що імітує реалістичну поведінку користувача та поступово збільшує навантаження на систему для виявлення її меж продуктивності.

Тест було налаштовано на поступове збільшення кількості віртуальних користувачів (VUs) за наступними етапами:

– плавне зростання до 50 VUs протягом 1 хвилини;

– подальше зростання до 100 VUs протягом наступних 3 хвилин;

– пікове навантаження до 150 VUs протягом 1 хвилини;

– пікове навантаження до 200 VUs протягом 1 хвилини;

– плавне зменшення навантаження до 0.

Така конфігурація дозволяє перевірити, як система поводиться при зростаючому навантаженні, і визначити точку, після якої може початися деградація продуктивності. Кожен віртуальний користувач виконував наступний цикл дій:

– навігація по сторінках: користувач послідовно «відвідував» основні сторінки застосунку, такі як головна, сторінка транзакцій (з випадковою датою) та налаштувань. Між запитами була встановлена пауза в 1 секунду (функція sleep) для імітації часу, який реальний користувач витрачає на вивчення контенту;

– операції запису: після навігації користувач виконував операцію створення та подальшого видалення фінансової цілі (createAndDeleteGoal). Це дозволило оцінити продуктивність не тільки операцій читання (GET), але й операцій запису (POST, DELETE).

У сценарії були визначені чіткі критерії успішності, зокрема вимога, що 95-й квантиль (p95) часу відповіді для ключових ендпоінтів, пов'язаних з фінансовими цілями, не повинен перевищувати 500 мс, а також відсоток непередбачених помилок не повинен перевищувати 1 % від усіх запитів.

Окрім тестування пікових навантажень, було проведено тест на стабільність (soak test) [24] для перевірки надійності системи при тривалій роботі під помірним навантаженням. Метою цього тесту було виявлення потенційних проблем, таких як витоки пам'яті, деградація продуктивності з часом або інші аномалії, що можуть проявитися лише при тривалій експлуатації.

Тест, визначений у скрипті «tests/soak.js», симулював постійне навантаження від 20 віртуальних користувачів (VUs) протягом 30 хвилин. Сценарій поведінки користувачів був зосереджений на операціях читання – перегляді основних сторінок застосунку, що є найбільш типовим сценарієм використання.

3.2 Обробка та аналіз отриманих результатів

Етап модульного тестування був зосереджений на ізольованій перевірці окремих функцій та компонентів. Для написання тестів використовувався стандартний пакет testing мови Go, бібліотека «testify/require» [26] для зручних тверджень (assertions) та «go.uber.org/mock/gomock» [27] для створення імітаційних об'єктів (моків) (ліст. 3.2).

Лістинг 3.2 – Демонстрація модульного тесту

```
func TestSummary(t *testing.T) {
    // 1. Налаштування тестової БД та користувача
    db := db.SetupTestDB(t, true)
```

```
    defer db.Close(t)
    userID := db.Users(t)[0].ID
    // 2. Створення мок-контролера та імітації CurrencyService
    mockCtrl := gomock.NewController(t)
    ratesSrv := mocks.NewMockCurrencyService(mockCtrl)
    // 3. Ініціалізація сервісу, що тестується, з моком
    transactions := NewTransactions(db.SqliteDB.Transactions, ratesSrv)
    // 4. Виклик методу та перевірка результату
    // В цьому тесті ми передаємо 0 як дохід, щоб отримати фактичні витрати,
    // а потім викликаємо функцію знову, передаючи фактичні витрати як
дохід,
    // щоб перевірити коректність розрахунку відсотків.
    summary, err := transactions.Summary(context.Background(), userID,
models.AnyTXAccount, from, to, 0)
    require.NoError(t, err)
    summary, err = transactions.Summary(context.Background(), userID,
models.AnyTXAccount, from, to, -summary.Spending)
    require.NoError(t, err)
    require.NotNil(t, summary)
    percentTotal:= 0.0
    for _, cat := range summary.Categories {
        percentTotal += cat.Percentage
    }
    require.InEpsilon(t, 100.0, percentTotal, 0.001)
}
```

Кінець лістингу 3.2

Під час модульного тестування, головна увага приділялася:

– тестуванню чистої бізнес-логіки в «internal/service» (наприклад, розрахунок відсотків для бюджетних цілей, логіка обробки транзакцій);

– перевірці коректності SQL-запитів у шарі доступу до даних «internal/db»;

– для забезпечення ізоляції тестів від реальної бази даних використовувалася тимчасова in-memory база даних SQLite.

Щоб запустити усі модульні тести та отримати результати покриття, виконується команда в корінцевій директорії проєкту: «go test -coverprofile=coverage.out ./internal». На рисунку 3.1, бачимо, що всі знайдені тести (поряд із значеннями покриття) були успішно пройдені.

```
        balance/internal/agent          coverage: 0.0% of statements
        balance/internal/currency               coverage: 0.0% of statements
ok      balance/internal/db     (cached)        coverage: 55.6% of statements
        balance/internal/db/notifications               coverage: 0.0% of statements
ok      balance/internal/mcc    (cached)        coverage: 53.3% of statements
        balance/internal/models         coverage: 0.0% of statements
ok      balance/internal/monobank       (cached)        coverage: 0.0% of statements
        balance/internal/notifier               coverage: 0.0% of statements
ok      balance/internal/secp256k1      (cached)        coverage: 66.7% of statements
ok      balance/internal/service        (cached)        coverage: 12.1% of statements
        balance/internal/service/auth           coverage: 0.0% of statements
        balance/internal/service/mocks          coverage: 0.0% of statements
        balance/internal/telegram               coverage: 0.0% of statements
        balance/internal/time           coverage: 0.0% of statements
        balance/internal/utils          coverage: 0.0% of statements
ok      balance/internal/web    (cached)        coverage: 8.7% of statements
        balance/internal/web/mocks              coverage: 0.0% of statements
        balance/internal/web/views              coverage: 0.0% of statements
        balance/internal/web/views/components           coverage: 0.0% of statements
        balance/internal/web/views/layout               coverage: 0.0% of statements
        balance/internal/web/views/pages                coverage: 0.0% of statements
ok      balance/internal/workers        (cached)        coverage: 20.2% of statements
```

Рисунок 3.1 – Результати запуску тестів разом із покриттям

Наступні тести наскрізного тестування роботи ШІ-агента подані у таблицях, перевіряють можливості агента аналізувати та оперувати даними. Справжні значення сум були приховані «X.XX».

Категорія 1: прості запити (базові агрегації). Результати тестування надані в таблицях 3.1, 3.2, 3.3 та 3.4.

Таблиця 3.1 – Результати запиту №1

Питання	Покажи мої останні 5 транзакцій
SQL	SELECT category, description, amount, time FROM transactions WHERE user_id = '11451708156099758327 6' AND is_ignored = false AND is_refund = false ORDER BY time DESC LIMIT 5;
Відповідь	Якщо потрібні деталі по якійсь транзакції, повідомте! Ось ваші 5 останніх транзакцій – «Продукти» – «Egastronom»: - X.XX грн; – «Кафе. Ресторани» – «XPZ*blur»: - X.XX грн; – «Кафе. Ресторани» – «Китайський Привіт»: - X.XX грн; – «Продукти» – «Близенько»: - X.XX грн; – «Таксі» – «Bolt»: -X.XX грн.

Таблиця 3.2 – Результати запиту №2

Питання	Скільки грошей я витратив за цей місяць?
SQL	SELECT COALESCE(ROUND(SUM(amount) / 100.0, 2), 0) AS monthly_spending FROM transactions WHERE is_ignored = false AND is_refund = false AND user_id = '11451708156099758327 6' AND amount < 0 AND strftime('%Y-%m', datetime(time, 'unixepoch')) = strftime('%Y-%m', 'now');
Відповідь	За цей місяць ви витратили X.XX грн. Чи бажаєте Ви отримати поради щодо витрат?

Таблиця 3.3 – Результати запиту №3

Питання	Який мій дохід за цей місяць?
SQL	SELECT SUM(amount) as total_income FROM user_income WHERE user_id = '11451708156099758327 6' AND strftime('%Y-%m', datetime(created_at, 'unixepoch')) = strftime('%Y-%m', 'now');
Відповідь	Ваш дохід за цей місяць становить X.XX грн. Чи бажаєте Ви отримати поради щодо складання бюджету?

Таблиця 3.4 – Результати запиту №4

Питання	Скільки я витратив минулого тижня?
SQL	SELECT SUM(amount) as total_spent FROM transactions WHERE is_ignored = false AND is_refund = false AND user_id = '11451708156099758327 6' AND amount < 0 AND time >= strftime('%s', 'now', 'weekday 0', '-7 days') AND time < strftime('%s', 'now', 'weekday 0');
Відповідь	Минулого тижня ви витратили приблизно X.XX грн. Якщо потрібно детальніше за категоріями чи по днях – уточніть, будь ласка!

Категорія 2: категорійні та групові запити. Результати тестування надані в таблицях 3.5, 3.6 та 3.7.

Таблиця 3.5 – Результати запиту №5

Питання	Які мої топ-5 категорій витрат за останній місяць?
SQL	SELECT category, ABS(SUM(amount)) AS total_spent FROM transactions WHERE is_ignored = false AND is_refund = false AND user_id = '11451708156099758327 6' AND amount < 0 AND time >= strftime('%s', 'now', '-1 month') GROUP BY category ORDER BY total_spent DESC LIMIT 5
Відповідь	Якщо потрібна детальніша інформація по будь-якій з категорій, дайте знати. Я завжди готовий Вам допомогти! Ваші топ-5 категорій витрат за останній місяць: – «Переказ коштів» – X.XX грн; – «Кафе та ресторани» – X.XX грн; – «Продукти» – X.XX грн; – «Одяг» – X.XX грн; – «Професійні послуги» – X.XX грн.

Таблиця 3.6 – Результати запиту №6

Питання	Скільки грошей я витратив на категорію «Продукти»?
SQL	SELECT SUM(amount) as total_spent FROM transactions WHERE user_id = '11451708156099758327 6' AND is_ignored = false AND is_refund = false AND amount < 0 AND category LIKE '%продукт%';
Відповідь	Ви витратили X.XX грн на категорію «Продукти». Чи бажаєте ви отримати поради щодо витрати? Напишіть «так», щоб отримати поради разом із аналітикою витрат.

Таблиця 3.7 – Результати запиту №7

Питання	Покажи розподіл моїх витрат за класами: «потреби», «бажання» та «заощадження»
SQL	SELECT class, SUM(amount) AS total_spent FROM transactions WHERE user_id = '11451708156099758327 6' AND is_ignored = false AND is_refund = false AND amount < 0 GROUP BY class;
Відповідь	Більша частина витрат була спрямована на бажання, менша – на потреби. Якщо потрібно бачити ці суми у відсотках чи уточнити по періодах, дайте знати. Ваш розподіл витрат за класами: – потреби: X.XX грн; – бажання: X.XX грн; – заощадження: даних по цій категорії серед списання не виявлено.

Категорія 3: складні та порівняльні запити. Результати тестування надані в таблицях 3.8, 3.9 та 3.10.

Таблиця 3.8 – Результати запиту №8

Питання	Порівняй мої загальні витрати в цьому місяці з минулим
SQL	SELECT SUM(amount) AS total_spending FROM transactions WHERE is_ignored = false AND is_refund = false AND amount < 0 AND user_id = '11451708156099758327 6' AND strftime('%Y-%m', datetime(time, 'unixepoch')) = strftime('%Y-%m', 'now'); SELECT SUM(amount) AS total_spending FROM transactions WHERE is_ignored = false AND is_refund = false AND amount < 0 AND user_id = '11451708156099758327 6' AND strftime('%Y-%m', datetime(time, 'unixepoch')) = strftime('%Y-%m', 'now', '-1 month');
Відповідь	Ваші загальні витрати цього місяця становлять X.XX грн, що більше, ніж минулого місяця – тоді ви витратили X.XX грн. Витрати зросли на X.XX грн порівняно з минулим місяцем.

Таблиця 3.9 – Результати запиту №9

Питання	Який мій середній чек в категорії «Ресторани»?
SQL	SELECT AVG(ABS(amount))/100.0 AS avg_receipt FROM transactions WHERE user_id = '114517081560997583276' AND is_ignored = false AND is_refund = false AND amount < 0 AND category LIKE '%ресторан%'
Відповідь	Ваш середній чек у категорії «Ресторани» становить приблизно X.XX грн.

Таблиця 3.10 – Результати запиту №10

Питання	В який день тижня я зазвичай витрачаю найбільше грошей?
SQL	SELECT strftime('%w', datetime(time, 'unixepoch')) AS weekday, SUM(amount) AS total_spent FROM transactions WHERE user_id = '114517081560997583276' AND is_ignored = false AND is_refund = false AND amount < 0 GROUP BY weekday ORDER BY total_spent ASC LIMIT 1;
Відповідь	Ви зазвичай витрачаєте найбільше грошей у понеділок.

Категорія 5: межові випадки. Результати тестування надані в таблицях 3.11, 3.12 та 3.13.

Таблиця 3.11 – Результати запиту №11

Питання	Яка моя найдорожча покупка за весь час?
SQL	SELECT description, category, amount, time FROM transactions WHERE user_id = '114517081560997583276' AND is_ignored = false AND is_refund = false AND amount < 0 ORDER BY amount ASC LIMIT 1;
Відповідь	Ваша найдорожча покупка за весь час – це платіж у категорії «Переказ коштів» із описом «…» на суму X.XX грн. Напишіть і я зможу допомогти з аналізом витрат.

Таблиця 3.12 – Результати запиту №12

Питання	Скільки я витратив на «подарунки» минулого місяця?
SQL	SELECT SUM(amount) / 100.0 AS total_spent FROM transactions WHERE user_id = '11451708156099758327 6' AND is_ignored = false AND is_refund = false AND amount < 0 AND category LIKE '%подар%' AND strftime('%Y-%m', datetime(time, 'unixepoch')) = strftime('%Y-%m', 'now', '-1 month')
Відповідь	Минулого місяця у вас не було витрат на подарунки, або їх не зафіксовано в категоріях із відповідними позначками. Якщо витрати на подарунки могли бути записані під іншими категоріями чи з іншими назвами, уточніть це – і я зможу провести додатковий пошук.

Таблиця 3.13 – Результати запиту №13

Питання	Що я можу зробити, щоб зменшити витрати і відкласти трохи грошей? Дай відповідь на основі моїх транзакцій.
SQL	SELECT category, SUM(amount)/100.0 AS total_spent FROM transactions WHERE is_ignored = false AND is_refund = false AND user_id = '11451708156099758327 6' AND amount < 0 GROUP BY category ORDER BY total_spent ASC;
Відповідь	Ось кілька практичних порад, спираючись на твої транзакції: Найбільше витрат йде на перекази коштів, кафе і ресторани, продукти та таксі. Спробуй зменшити частоту чи суму оплат у цих категоріях. Обмеж часті обіди/каву в кафе та ресторанах. Переглянь, чи можеш більше готувати вдома. Продумай покупки в супермаркетах заздалегідь, складай список, щоб уникнути імпульсивних витрат. Для таксі: шукай альтернативи –громадський транспорт або спільні поїздки. Одяг, краса, доставка їжі також займають значну частку у витратах. Корисно планувати покупки одягу чи робити їх у період знижок.

Продовження таблиці 3.13

Питання	Що я можу зробити, щоб зменшити витрати і відкласти трохи грошей? Дай відповідь на основі моїх транзакцій.
Відповідь	Сервіси доставки їжі використовувати як виняток, а не звичку. Проаналізуй регулярні немаленькі витрати (наприклад, стоматологія, здоров'я). Якщо це планові послуги, відкладай на них суми частинами щомісяця, а не одномоментно. Впровадь правило: щомісяця хоча б 10 % доходу відкладати одразу після надходження. Почни з невеликої суми, яку комфортно відкласти й не використовуй ці кошти на поточні витрати. Навіть незначне скорочення витрат у «кафе/ресторани», таксі чи одязі найбільше вплине на кінцевий результат. Склади персональний бюджет, встанови прості ліміти по основних категоріях і стеж за динамікою витрат раз на тиждень. Готовий допомогти розрахувати ліміти для категорій або створити персональний бюджет?

Для оцінки якості та коректності роботи ШІ-агента було проведено наскрізне тестування на основі наданих запитів різної складності. Аналіз згенерованих SQL-запитів та фінальних відповідей показав високий рівень ефективності та «інтелекту» агента. Агент продемонстрував відмінну здатність розуміти природну мову та коректно трансформувати більшість запитів у правильні, а часто й оптимізовані, SQL-запити.

Сильні сторони, підтверджені тестами:

– агент вміє «запам'ятовувати» інформацію з минулих чатів для кожного з користувачів;

– точність базових операцій: агент бездоганно впорався з простими та категорійними запитами (запити 1-7), коректно використовуючи агрегатні функції (SUM, AVG), сортування (ORDER BY), групування (GROUP BY) та фільтрацію;

– робота з датами: продемонстровано коректну обробку складних часових проміжків, таких як «цей місяць» та «минулий тиждень», з використанням відповідних функцій strftime (запити 2, 3, 4);

– складні сценарії: агент успішно впорався зі складними завданнями. Наприклад, для порівняльного аналізу (запит 8) він самостійно згенерував два окремі запити для отримання даних за різні періоди, що є безпечним та надійним підходом;

– контекстна обізнаність та логіка: при пошуку найдорожчої покупки (запит 11) агент правильно використав «ORDER BY amount ASC», розуміючи, що витрати є від'ємними числами;

– при обробці відкритих питань (запит 13, «Як зменшити витрати?») агент не просто надав дані, а й проаналізував їх, сформулювавши персоналізовані фінансові поради на основі найбільших категорій витрат користувача. Це демонструє здатність до синтезу інформації, а не простого її виведення;

– відсутність даних або неоднозначність: при запиті про неформальну категорію «подарунки» (запит 12), агент не видав помилку, а надав корисну відповідь, пояснивши можливу причину відсутності даних і запропонувавши альтернативні шляхи пошуку.

Виявлені недоліки та обмеження:

– інколи агент може галюцинувати або погоджуватися, навіть якщо ви явно робите помилку в судженнях: це типова проблема великих мовних моделей (LLM);

– в цьому розробленому прототипі агент не має доступу до інших джерел інформації (наприклад даних з відкритих джерел, як-от курси валют), окрім власної бази знань моделі та доступу до бази даних транзакцій. В такому випадку користувачу самотужки потрібно вказати додаткову інформацію;

– коректність відповідей часто залежить від належності даних про транзакції. Тому, для отримання кращих результатів, інколи потрібно власноруч змінювати назву, опис, категорію, суму транзакцій, що були автоматично підтягнуті;

– на етапі прототипу, агент запам'ятовує лише 10 останніх повідомлень від користувача. Зазвичай цього достатньо, але існують сценарії де це створює обмеження.

Розглянемо результати тестування продуктивності із димного тесту, який не знайшов ніяких помилок. Для перевірки відповідності системи нефункціональним вимогам було проведено тестування навантаження за допомогою інструменту k6. Тест продуктивності симулював роботу до 200 одночасних віртуальних користувачів (VUs), які виконували операції читання та запису. Для проведення тесту було використано сервер від хмарного вендора Fly.io з наступними характеристиками:

– розташування в місті Стокгольм, Швеція;
– CPU: 2 спільних vCPU;
– оперативна пам'ять: 2048 MB;
– сховище на NVMe SSD з пропускною здатністю в 2000 IOPS та 8 МіБ/с.

В Fly.io, тип «спільний» (shared) [28] процесора означає, що декілька віртуальних машин ділять віртуальні ядра цього ж процесора. Для кожного 80 мс періоду часу, кожне ядро спільного процесора отримує квоту в 5 мс, що становить 6,25 % від усього періоду. Отже, для 2-ох ядерного процесора квота подвоюється до 10 мс. Якщо квота вичерпана, додаток призупиняється (throttled) на решту 80 мс періоду. Спільний процесор може працювати в режимі «burst» до 100 % використання, коли є накопичений баланс невикористаного процесорного часу (максимум 500 секунд балансу).

Результати тесту повністю підтвердили відповідність системи заявленим цільовим рівням обслуговування (SLO). Отримані показники свідчать про стабільну роботу сервісу навіть за підвищеного навантаження. Зокрема, особливу увагу було приділено аналізу часу відповіді, оскільки цей параметр безпосередньо впливає на зручність взаємодії користувача з системою та загальну якість роботи застосунку.

Час відповіді: 95-й квантиль (p95) часу відповіді склав 116 мс (значення взято з метрики http_req_duration), що є значно нижчим за встановлений поріг у 500 мс (табл. 3.14).

Таблиця 3.14 – Результати тестування часу відповіді різних ендпоінтів

Endpoint	Average	Min	Median	Max	P90	P95
/	396,6 мс	55,52 мс	61,25 мс	1 хв 0 с	72,45 мс	90,3 мс
/api/v1/goals	421,07 мс	54,15 мс	59,77 мс	59,99 с	65,91 мс	79,38 мс
/api/v1/goals/percentage	605,85 мс	60,7 мс	70,77 мс	1 хв 0 с	108,86 мс	147,78 мс
/api/v1/spending/ratios	550,49 мс	57,2 мс	65,49 мс	1 хв 0 с	98,99 мс	136,73 мс
/api/v1/spending/summary	455,09 мс	59,06 мс	67,71 мс	1 хв 0 с	110,11 мс	147,39 мс
/notifications	620,34 мс	56,55 мс	62,31 мс	1 хв 0 с	68,61 мс	80,98 мс
/settings	284,65 мс	56,34 мс	62,2 мс	1 хв 0 с	68,21 мс	74,58 мс
/setup/income	676,21 мс	55,87 мс	61,63 мс	1 хв 0 с	68,7 мс	83,88 мс
/transactions	280,87 мс	57,21 мс	62,42 мс	59,99 мс	69,37 мс	81,8 мс
/transactions/unspecified	637,17 мс	65,45 мс	76,75 мс	59,99мс	106.87 мс	144,37 мс

Рівень помилок: протягом усього тесту 192 запити вернули помилку. Усього рівень помилок склав 0,7 % (метрика http_req_failed), що повністю відповідає вимозі (< 1 %). Помилки почали з’являтися після того, як кількість запитів перетнула відмітку в 160 RPS, та кількість одночасних віртуальних користувачів виросла до 157 (рис. 3.2).

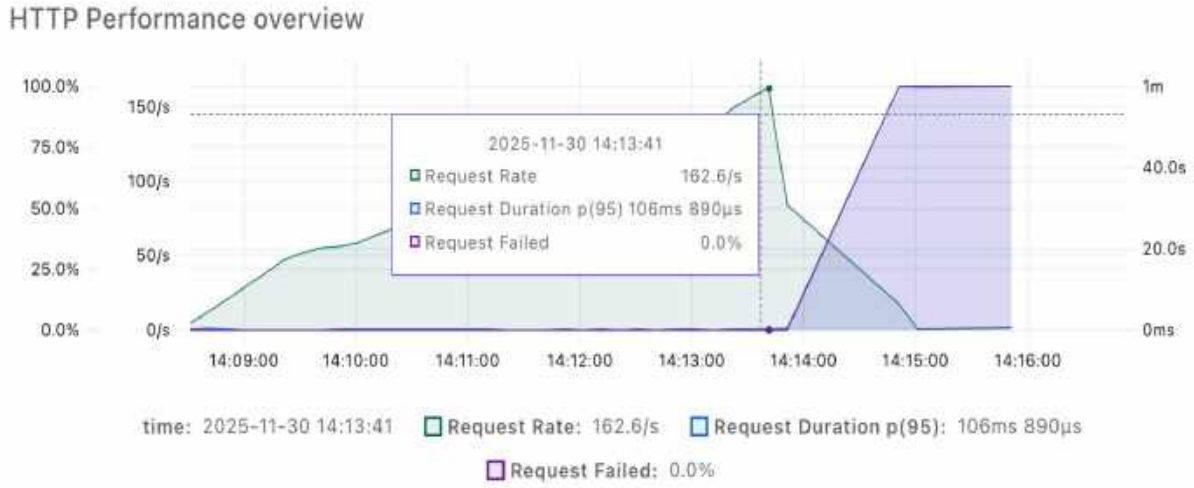

Рисунок 3.2 – Загальний графік продуктивності

Пропускна здатність: під час тесту, сервіс витримав в сумі 27 тисяч запитів за 7 хв, обробляючи в середньому 53 запити на секунду (RPS) (метрика http_reqs), досягаючи максимум 162 RPS, що відповідає мінімально необхідному показнику в 50 RPS (рис. 3.3).

Рисунок 3.3 – Графіки RPS та VUs

Графіки використання оперативної пам’яті (рис. 3.4) та навантаження на процесор (рис. 3.5) сервера очікувано високі, що вказує на те, що система працювала на максимум.

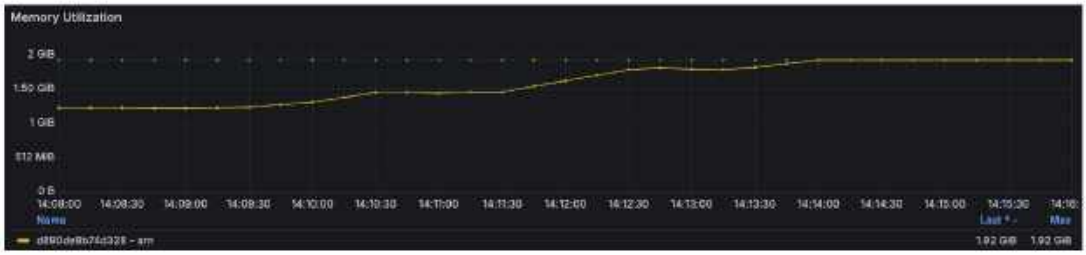

Рисунок 3.4 – Графік використання оперативної пам'яті сервера

Рисунок 3.5 – Графік навантаження процесора у відсотках

Результати тестування стабільності підтвердили стабільність системи: протягом усього тесту час відповіді (p95) залишався стабільним не перевищуючи позначку в 129 мс, без ознак деградації (рис. 3.6).

Рисунок 3.6 – Загальний графік продуктивності тесту стабільності

Рівень помилок сягав 0,1 % за весь період тестування. Моніторинг ресурсів сервера (CPU, RAM) під час тесту не виявив аномального зростання чи витоків пам'яті, що свідчить про ефективне управління ресурсами в довгостроковій перспективі (рис. 3.7 та рис. 3.8).

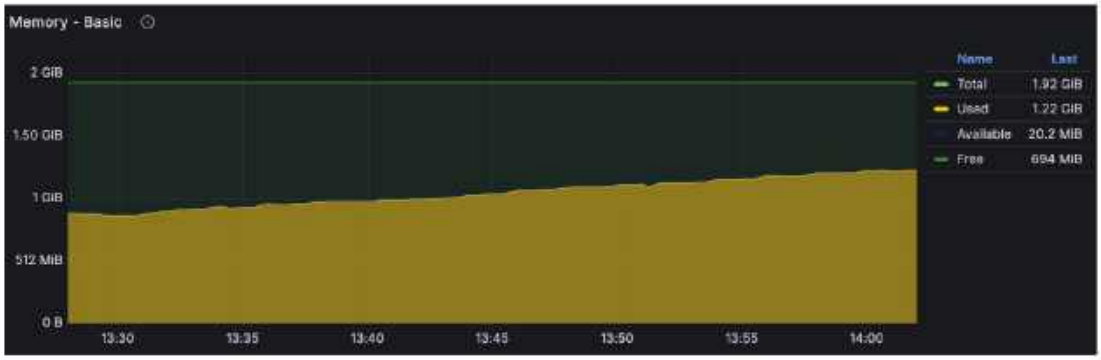

Рисунок 3.7 – Графік оперативної пам'яті сервера під час тесту стабільності

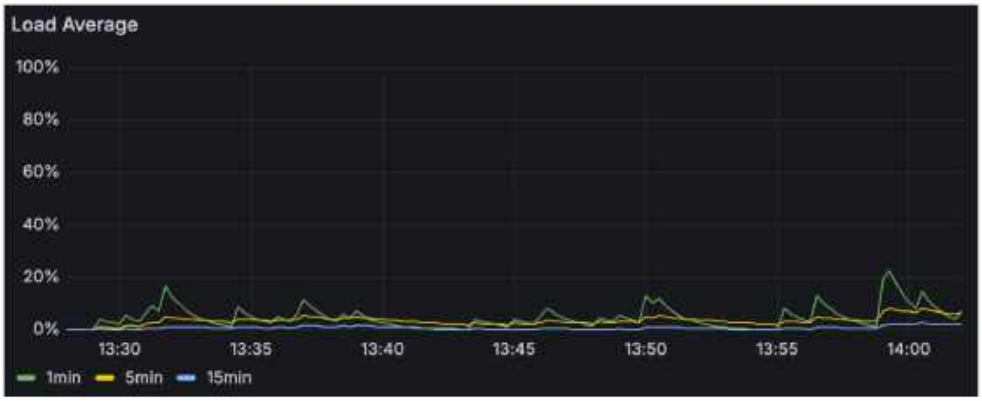

Рисунок 3.8 – Середнє значення навантаження на процесор

Результати, отримані в ході експериментального дослідження, мають безпосереднє практичне значення. Успішне проходження тестів навантаження та стабільності підтверджує, що архітектурні рішення, обрані в розділі 2 (зокрема, використання Go для основного сервісу), були коректними та дозволили створити продуктивну систему. Практично це означає, що розроблений прототип готовий до розгортання на хмарній інфраструктурі з конфігурацією, близькою до тестової (2 vCPU, 2 GB RAM), і може стабільно обслуговувати до 75-100 одночасних користувачів. Підтверджена надійність ШІ-агента дозволяє використовувати його як основний інструмент для персоналізованої фінансової аналітики. Завдяки контейнеризації за допомогою Docker, процес впровадження є максимально спрощеним і може бути виконаний на будь-якому сучасному хостингу або локально.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи магістра розроблено програмний продукт «Персоналізований асистент фінансової свідомості та заощаджень» – інструмент, який автоматизує збір та аналіз банківських транзакцій, сприяючи покращенню фінансової грамотності користувачів. У процесі дослідження було виконано комплекс завдань, що охоплюють аналітичний, проєктний, інженерний та експериментальний етапи розроблення програмного забезпечення:

– було проведено всебічний аналіз сучасної екосистеми управління особистими фінансами, включно з розглядом доступних мобільних застосунків, вебсервісів та банківських рішень. Виявлено ключові недоліки існуючих продуктів, серед яких – обмежена персоналізація, відсутність глибокого аналізу витрат та недостатня інтеграція з українськими банками. Результати аналізу підтвердили доцільність створення нового інструменту, орієнтованого на гнучкість, приватність та підтримку фінансової свідомості користувачів;

– було розроблено гібридну мікросервісну архітектуру, що поєднує високу продуктивність сервісів на Go з гнучкістю Python сервісів для аналітики та роботи з ШІ. Архітектура забезпечує горизонтальне масштабування, незалежність компонентів і можливість швидкого оновлення окремих модулів без зупинки всієї системи. Обґрунтовано вибір технологій, транспортів взаємодії та підходів до оркестрації;

– створено нормалізовану структуру реляційної бази даних SQLite, що забезпечує збереження інформації про користувачів, рахунки, транзакції, категорії витрат, доходи та фінансові цілі. Особлива увага приділена узгодженості даних, цілісності та можливості оптимального виконання аналітичних запитів;

– розроблено модуль автоматичної інтеграції з API Monobank, який забезпечує регулярне отримання історії транзакцій, їх первинну обробку та збереження у базі даних. Реалізовано механізми валідації, обробки помилок та

оновлення лише змінених даних, що зменшує навантаження на систему;

– реалізовано основні функціональні можливості застосунку (бізнес логіка) – аналітику витрат, дашборди, візуалізацію транзакцій, динаміку доходів та досягнення фінансових цілей. Веб-інтерфейс забезпечує інтуїтивний досвід користувача, а серверна частина – коректну обробку запитів та бізнес правил;

– розроблено фонові процеси, які виконують регулярну синхронізацію даних з банківськими сервісами без участі користувача. Забезпечено стабільний механізм обробки помилок та повторних спроб, що гарантує актуальність даних у системі;

– розроблено ШІ-агента, який взаємодіє з базою даних користувача та здатний самостійно визначати, які SQL-запити потрібно виконати для отримання релевантної інформації. Агент забезпечує діалоговий режим консультацій, формує персоналізовані фінансові рекомендації та пояснення у зрозумілій формі;

– проведено модульне та наскрізне тестування основних компонентів системи. Перевірено коректність бізнес логіки, стабільність роботи мікросервісів, якість обробки даних та надійність інтеграцій. Тестування підтвердило працездатність системи та відповідність розробленому технічному завданню;

– всі компоненти програмного продукту були контейнеризовані із застосуванням Docker, що забезпечує можливість швидкого розгортання, масштабування та ізольованої роботи мікросервісів.

Узагальнюючи отримані результати, можна стверджувати, що створений персоналізований асистент є готовим до впровадження прототипом з відкритим вихідним кодом. Запропоновані архітектурні рішення та реалізація принципів Open Banking мають високу практичну цінність для українського ринку фінтех-рішень, а закладена основа дозволяє легко масштабувати продукт та розширювати його функціонал у майбутньому.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Затверджено положення про відкритий банкінг в Україні – Національний банк України. URL: https://bank.gov.ua/ua/news/all/zatverdjeno-polojennya-pro-vidkritiy-banking-v-ukrayini (дата звернення: 01.11.2025).

2. Прасад А. Українці досі користуються російськими фінансовими застосунками. Їхня частка становить 50 % – Forbes.ua. URL: https://forbes.ua/news/ukraintsi-dosi-koristuyutsya-rosiyskimi-finansovimi-zastosunkami-ikh-chastka-stanovit-50-27092023-16304 (дата звернення: 01.11.2025).

3. Чи нарешті настала революція self-hosting? На справді цього разу – DreamHost Blog. URL: https://www.dreamhost.com/blog/uk/chi-nareshti-nastala-revoliutsiia-self-hosting-na-spravdi-tsogo-razu-uk/ (дата звернення: 01.11.2025).

4. Firefly III. Офіційний сайт. URL: https://www.firefly-iii.org (дата звернення: 01.11.2025).

5. Large Language Models – IBM Think. URL: https://www.ibm.com/think/topics/large-language-models (дата звернення: 02.11.2025).

6. Model Context Protocol. Документація. URL: https://modelcontextprotocol.io/docs/getting-started/intro (дата звернення: 04.11.2025).

7. Docker – Wikipedia: the free encyclopedia. URL: https://en.wikipedia.org/wiki/Docker_(software (дата звернення: 05.11.2025).

8. Financial Wellness: 50/30/20 Rule – UNFCU. URL: https://www.unfcu.org/financial-wellness/50-30-20-rule/ (дата звернення: 10.11.2025).

9. Zero based budgeting – Wikipedia: the free encyclopedia. URL: https://en.wikipedia.org/wiki/Zero-based-budgeting (дата звернення: 10.11.2025).

10. Методи «конвертів»: як накопичувати гроші, щоб вистачало – ПриватБанк. URL: https://privatbank.ua/handbook/metody-konvertiv-iak-nakopychuvaty-hroshi-shchob-vystachalo (дата звернення: 13.11.2025).

11. RFC 6749 – The OAuth 2.0 Authorization Framework – Internet Engineering Task Force. URL: https://datatracker.ietf.org/doc/html/rfc6749 (дата звернення: 13.11.2025).

12. Service Level Objectives – Google SRE. URL: https://sre.google/sre-book/service-level-objectives/ (дата звернення: 13.11.2025).

13. monobank API. Документація. URL: https://api.monobank.ua/docs/index.html (дата звернення: 14.11.2025).

14. Merchant category code (MCC) – Wikipedia: the free encyclopedia. URL: https://en.wikipedia.org/wiki/Merchant_category_code (дата звернення: 14.11.2025).

15. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. O'Reilly Media, 2021. 426 p. ISBN 1492034029.

16. Arundel J. Go go goroutines: understanding Go's concurrency model – Bitfield Consulting Blog. URL: https://bitfieldconsulting.com/posts/goroutines (дата звернення: 15.11.2025).

17. Goth (markbates) – Репозиторій на GitHub. URL: https://github.com/markbates/goth (дата звернення: 15.11.2025).

18. API для провайдерів (Corporate API). Monobank open API for providers – Monobank. URL: https://api.monobank.ua/docs/corporate.html (дата звернення: 15.11.2025).

19. Software as a Service – Wikipedia: the free encyclopedia. URL: https://en.wikipedia.org/wiki/Software_as_a_service (дата звернення: 20.11.2025).

20. Agents – Python Library Documentation – LangChain. URL: https://docs.langchain.com/oss/python/langchain/agents (дата звернення: 20.11.2025).

21. Adversarial Machine Learning: A Taxonomy and Terminology of Attacks and Mitigations. NIST AI 100 2 E2025. Gaithersburg, MD: NIST, March 2025.

22. GPT-4 Technical Report – OpenAI. 2023. arXiv:2303.08774.

23. Boonstra L. Prompt Engineering. URL: https://drive.google.com/file/d/1AbaBYbEa_EbPelsT40-vj64L-2IwUJHy/view (дата звернення: 20.11.2025).

24. Soak testing: A beginner's guide – Grafana Labs. URL: https://grafana.com/blog/2024/01/30/soak-testing/ (дата звернення: 01.12.2025).

25. Вадим П. Grafana k6: Performance, stress та smoke тестування. Теорія і практика про те, як покрити все однією кодовою базою – DOU.ua. URL: https://dou.ua/forums/topic/42222/ (дата звернення: 01.12.2025).

26. stretchr/testify. GitHub repository – GitHub. URL: https://github.com/stretchr/testify (дата звернення: 02.12.2025).

27. gomock – mocking framework for Go – go.uber.org/mock. URL: https://go.uber.org/mock/gomock (дата звернення: 02.12.2025).

28. CPU Performance – Fly.io. URL: https://fly.io/docs/machines/cpu-performance/ (дата звернення: 02.12.2025).

29. Agentic AI: Finance and the «Do It For Me» Economy – Citi Global Perspectives & Solutions. Citi Research, 2025.