Київський столичний університет імені Бориса Грінченка

Факультет інформаційних технологій та управління

Кафедра комп’ютерних наук і математики

Допущено до захисту

Завідувач кафедри комп’ютерних наук

доктор технічних наук, професор

Андрій БОНДАРЧУК

« 01 » червня 2026 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня бакалавр

спеціальність 122 Комп’ютерні науки

освітня програма 122.00.01 Інформатика

Тема: ІНТЕРАКТИВНИЙ ВЕБЗАСТОСУНОК МОНІТОРИНГУ ФІНАНСОВИХ РАХУНКІВ КОРИСТУВАЧА

Виконав

студент групи ІНб-2-22-4.0д

Верхочуб Віталій В’ячеславович

_____

(підпис)

Науковий керівник

к.п.н., доцент,

Бойко Марія Анатоліївна

_____

(підпис)

Київ – 2026

Київський столичний університет імені Бориса Грінченка
Факультет інформаційних технологій та управління
Кафедра комп’ютерних наук і математики

«Затверджую»
Завідувач кафедри комп’ютерних наук
доктор технічних наук, професор
Андрій БОНДАРЧУК
« 31 » жовтня 2026 р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ БАКАЛАВРА
«Інтерактивний вебзастосунок моніторингу фінансових рахунків користувача»

Виконавець – студент групи ІНб-2-22-4.0д,
спеціальності 122 Комп’ютерні науки,
освітньої програми 122.00.01 Інформатика
Верхочуб Віталій В’ячеславович

1. Вихідні дані: наукові, нормативні й навчально-методичні матеріали з проєктування вебзастосунків, PFM-систем, REST API, безпеки фінансових даних, інтеграції з банківськими API, PWA та Web Push. Практична база – реалізований проєкт Finance Monitor.
2. Основні завдання: проаналізувати предметну область; сформулювати вимоги; спроєктувати архітектуру, базу даних і сценарії роботи; реалізувати backend, frontend, Monobank-синхронізацію, AI-модуль, Web Push та додаткові фінансові модулі; провести тестування і визначити напрями розвитку.
3. Пояснювальна записка: обсяг не менше 55 сторінок формату А4 комп’ютерного набору з дотриманням вимог стандартів і методичних рекомендацій кафедри.
4. Графічні матеріали: схеми архітектури, ER-діаграма, UML-діаграма, діаграми сценаріїв синхронізації, використання, додавання рахунку, розгортання та скріншоти інтерфейсу.
5. Додатки: фрагменти реалізації, таблиці тестування, діаграми системи, скріншоти й додаткові ілюстративні матеріали.
6. Строк подання роботи на кафедру: «01» червня 2026 р.

Науковий керівник
к.п.н., доцент
_____ Бойко М.А.
_____

Виконавець:
_____ Верхочуб В.В.
_____

КАЛЕНДАРНИЙ ПЛАН

№	Етапи виконання роботи	Термін виконання	Примітка
1	Формування теми кваліфікаційної роботи бакалавра та її затвердження	01.10.25 – 15.10.25	виконано
2	Аналіз предметної області, огляд PFM-систем, банківських застосунків і вимог до фінансового моніторингу	16.10.25 – 20.11.25	виконано
3	Формування функціональних і нефункціональних вимог до Finance Monitor	21.11.25 – 10.12.25	виконано
4	Проєктування архітектури системи, структури бази даних, сценаріїв роботи та діаграм	11.12.25 – 31.12.25	виконано
5	Реалізація backend на FastAPI, моделей SQLAlchemy, REST endpoint, авторизації та фонових задач Celery/Redis	01.01.26 – 31.01.26	виконано
6	Реалізація frontend на Next.js 14: дашборди, сторінки аналізу, історія транзакцій, фільтри рахунків	01.02.26 – 28.02.26	виконано
7	Інтеграція з Monobank API, обробка rate limit, унікалізація транзакцій і модуль експорту даних	01.03.26 – 20.03.26	виконано
8	Реалізація Web Push, локального AI-модуля Ollama, бюджетів, цілей, новин і ФОП-сценаріїв	21.03.26 – 10.04.26	виконано
9	Тестування API, користувацьких сценаріїв, негативних випадків, обмежень Monobank API та PWA/push-механізмів	11.04.26 – 28.04.26	виконано
10	Оформлення пояснювальної записки, підготовка діаграм, додатків, списку джерел і внесення правок	29.04.26 – 15.05.26	виконано
11	Підготовка презентаційних матеріалів, тексту доповіді та захист кваліфікаційної роботи	16.05.26 – 15.06.26	заплановано

«Затверджую»

Науковий керівник — Індивідуальний план склав

к.п.н., доцент, Бойко М.А. — Верхочуб В.В.

_____ _____

РЕФЕРАТ

Кваліфікаційна робота: 59 с., 16 рис., 13 табл., 31 джерело, 3 додатки.

Актуальність. Кваліфікаційна робота присвячена проблемі автоматизованого обліку й аналізу персональних фінансових даних користувача. Наявні банківські застосунки та PFM-сервіси частково розв'язують цю задачу, але для українського користувача залишаються обмеження, пов'язані з роботою кількох рахунків, локальними банківськими API, ФОП-сценаріями, приватністю даних і потребою в прозорій аналітиці.

Мета роботи. полягає у проєктуванні, реалізації та перевірці вебзастосунку Finance Monitor для автоматизованого обліку, синхронізації та аналізу персональних фінансових даних користувача.

Об'єктом дослідження. є процеси автоматизованого обліку, синхронізації та аналізу персональних фінансових даних в інформаційних системах. Предметом дослідження є методи, архітектурні підходи та програмні засоби побудови клієнт-серверного вебзастосунку для обробки й аналізу персональних фінансових даних.

Результати роботи. включають спроєктовану архітектуру Finance Monitor, модель даних, реалізований клієнт-серверний прототип, механізми синхронізації транзакцій, локальний AI-модуль, push-комунікацію, приклади тестування та комплект діаграм, що пояснюють ключові сценарії роботи системи.

Практичне значення. полягає у можливості використання реалізованого рішення як основи для персонального фінансового кабінету українського користувача з підтримкою гривні, локальних банківських сценаріїв, ФОП-логіки, контрольованої історії транзакцій і подальшого підключення нових джерел даних.

Ключові слова: персональні фінанси, фінансовий моніторинг, pfm-система, клієнт-серверна архітектура, банківський api, транзакції, дашборд, локальний ai-модуль, web push, finance monitor.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

AI — Artificial Intelligence, штучний інтелект; у роботі використовується для пояснення підготовлених фінансових показників.

API — Application Programming Interface, інтерфейс програмування застосунків для взаємодії програмних модулів.

CSRF — Cross-Site Request Forgery, міжсайтова підробка запиту; ризик, який враховано для захисту чутливих POST-запитів.

JWT — JSON Web Token, формат токена для передавання даних автентифікації між клієнтом і сервером.

LLM — Large Language Model, велика мовна модель; у Finance Monitor застосовується локально через Ollama.

ORM — Object-Relational Mapping, об’єктно-реляційне відображення для роботи з базою даних через програмні моделі.

PFM — Personal Finance Management, системи управління особистими фінансами.

PWA — Progressive Web App, прогресивний вебзастосунок із підтримкою service worker і push-сповіщень.

REST API — архітектурний підхід до взаємодії клієнта і сервера через HTTP-запити до ресурсів.

VAPID — Voluntary Application Server Identification, механізм ідентифікації сервера для Web Push.

Web Push — механізм доставки push-повідомлень у браузері або PWA-середовищі.

ФОП — фізична особа-підприємець; у роботі використовується для відокремлення підприємницьких і особистих фінансових сценаріїв.

ЗМІСТ

ВСТУП

Цифровізація банківських послуг змінила спосіб взаємодії користувачів із фінансовими установами. Значна частина операцій, зокрема перегляд балансу, отримання виписки, переказ коштів, керування картками та аналіз витрат, виконується через мобільні або вебзастосунки. Разом з тим наявність окремих банківських застосунків не завжди забезпечує цілісне уявлення про фінансовий стан користувача, особливо якщо він використовує кілька рахунків, розділяє особисті та робочі витрати або веде облік регулярних платежів [3; 4].

Актуальність роботи зумовлена тим, що наявні рішення для управління особистими фінансами не повністю закривають локальний сценарій українського користувача. Банківські застосунки, зокрема Monobank, зручно показують дані в межах одного банку, але не дають повного контрольованого шару для агрегації, власних правил аналітики та довгострокового збереження історії. Міжнародні PFM-сервіси мають сильні механізми бюджетування, проте часто залежать від ринків із розвиненим open banking і не враховують українські банківські API, гривню, ФОП-логіку та потребу в локальній обробці чутливих даних [4; 5; 6; 7; 8].

Практична доцільність створення власного рішення полягає не в дублюванні функцій окремого банківського застосунку, а в поєднанні кількох можливостей у межах одного продукту: агрегування рахунків, збереження локальної історії транзакцій, аналіз за категоріями та рахунками, підтримка ФОП-логіки, контроль бюджетів, локальний AI-аналіз без передавання фінансових даних стороннім LLM-сервісам, а також розширювана архітектура для додавання нових банківських конекторів.

Метою кваліфікаційної роботи є проєктування, реалізація та перевірка вебзастосунку Finance Monitor для автоматизованого обліку й аналізу персональних фінансових даних користувача.

Об’єктом дослідження є процеси автоматизованого обліку, синхронізації та аналізу персональних фінансових даних в інформаційних системах.

Предметом дослідження є методи, архітектурні підходи та програмні засоби побудови клієнт-серверного вебзастосунку для обробки й аналізу персональних фінансових даних.

Для досягнення мети поставлено такі завдання:

– проаналізувати підходи до побудови PFM-систем та обмеження наявних рішень для українського користувача;

– проаналізувати предметну область PFM-систем, банківських застосунків та обмеження наявних рішень для українського користувача;

– сформулювати вимоги до Finance Monitor і спроєктувати архітектуру, модель даних та основні сценарії взаємодії компонентів;

– реалізувати backend, frontend, синхронізацію транзакцій, локальний AI-модуль, Web Push та додаткові фінансові модулі;

– провести тестування API, користувацьких сценаріїв, негативних випадків, продуктивності та поведінки системи за умов обмежень зовнішнього API;

– оцінити отримані результати, практичне значення, обмеження реалізованого рішення та напрями подальшого розвитку.

Методи дослідження добиралися відповідно до поставлених завдань: системний і порівняльний аналіз використано для дослідження предметної області та наявних PFM-рішень; об’єктно-орієнтоване проєктування і моделювання даних — для побудови архітектури та структури бази даних; методи проєктування REST API — для визначення взаємодії клієнтської і серверної частин; функціональне, інтеграційне та негативне тестування — для перевірки працездатності реалізованих сценаріїв і стійкості системи до помилкових запитів.

Практичне значення роботи полягає у створенні програмного продукту, який може бути використаний як основа для персонального фінансового

кабінету українського користувача. На відміну від окремого банківського застосунку, Finance Monitor зберігає історію фінансових операцій у власній базі, дозволяє порівнювати рахунки, відокремлювати особисті та ФОП-витрати, виконувати локальний AI-аналіз і поступово підключати нові джерела даних.

Елементи новизни роботи полягають у реалізації прикладного підходу до поєднання локальної фінансової аналітики, account-scoped унікалізації транзакцій, фонової синхронізації з урахуванням обмежень банківського API та локального LLM-модуля для пояснення фінансових показників без передавання повного набору фінансових даних зовнішнім сервісам.

Структура роботи включає вступ, чотири розділи, висновки, список використаних джерел і додатки. У першому розділі виконано аналіз предметної області, у другому описано проєктування системи, у третьому наведено реалізацію ключових модулів, у четвертому подано тестування, результати та обмеження системи.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ

1.1 Підходи до побудови систем моніторингу особистих фінансів

Системи моніторингу особистих фінансів належать до класу PFM-рішень (Personal Finance Management). Їх основне призначення полягає у збиранні, структуризації та поданні фінансових даних користувача у формі, придатній для прийняття рішень щодо витрат, доходів, заощаджень і бюджетів. У технічному аспекті такі системи зазвичай складаються з модулів імпорту даних, нормалізації транзакцій, категоризації, аналітики, сповіщень і користувацького інтерфейсу [5; 6; 7; 8].

Існують кілька підходів до побудови PFM-систем. Перший підхід передбачає ручне введення операцій користувачем. Його перевагою є простота реалізації та незалежність від банківських API, однак основний недолік полягає у високій трудомісткості та ризику пропуску операцій. Другий підхід ґрунтується на імпорті файлів CSV, XLS або JSON. Він зменшує ручне введення, але не забезпечує повної автоматизації та потребує регулярного експорту даних із банківських сервісів. Третій підхід базується на API-інтеграції з банками, що дозволяє автоматизувати синхронізацію, проте залежить від доступності API, лімітів запитів, формату даних і стабільності зовнішнього сервісу [4; 5; 6].

Окремим напрямом розвитку є використання алгоритмів автоматичної категоризації та AI-модулів. На практиці такі модулі можуть виконувати дві різні ролі: класифікувати транзакції за категоріями або пояснювати вже підготовлені аналітичні показники природною мовою. У Finance Monitor обрано другий підхід: AI-модуль не замінює фінансові розрахунки, а отримує підготовлений backend-контекст і формує текстове пояснення для користувача. Така архітектура знижує ризик некоректних числових розрахунків з боку LLM, оскільки підсумкові значення обчислюються детермінованим кодом backend-частини [15; 32].

Для PFM-систем критично важливими є питання цілісності даних. Повторне збереження однієї транзакції призводить до завищення витрат або доходів і спотворює дашборди. Тому в архітектурі системи необхідно передбачити стабільний ідентифікатор транзакції, прив’язаний до конкретного рахунку, а також механізм безпечної повторної синхронізації. У випадку Monobank ця проблема є особливо актуальною через отримання виписок за часовими інтервалами та можливість перекриття таких інтервалів під час повторних запитів [4; 32].

1.2 Аналіз існуючих сервісів і критерії порівняння

Для аналізу обрано сервіси, що репрезентують різні підходи до управління фінансами: банківський застосунок Monobank, міжнародний фінансовий сервіс Revolut, спеціалізовані PFM-рішення YNAB, Rocket Money та Monarch. Порівняння виконано не за маркетинговими характеристиками, а за критеріями, релевантними для розроблення власного вебзастосунку: інтеграція з українськими банками, спосіб доступу до даних, автоматизація синхронізації, локалізація, гнучкість аналітики, підтримка AI-компонентів, приватність обробки даних і можливість розширення системи [4; 5; 6; 7; 8].

Monobank забезпечує зручну роботу з рахунками власного банку, надає користувачу історію операцій, базову статистику та персональний API для отримання інформації про клієнта й виписки. Водночас його застосування як універсального PFM-рішення обмежується межами одного банку, лімітом частоти звернень до API та відсутністю відкритої модульної архітектури для агрегування інших банківських джерел. Тому Monobank доцільно розглядати не як конкурента Finance Monitor у повному обсязі, а як важливе джерело даних для першого етапу інтеграції [4].

Revolut має розвинені інструменти бюджетування, аналітики та роботи з Pockets, однак для українського контексту критичним обмеженням є неповна локальна банківська інтеграція та залежність функцій open banking від країни використання. Для користувача, який працює переважно з українськими

рахунками, це знижує практичну придатність Revolut як єдиного інструмента обліку [5].

YNAB, Rocket Money і Monarch орієнтовані переважно на ринки США та інших країн із розвиненими open banking-інтерфейсами. Вони мають сильні сторони у бюджетуванні, виявленні підписок або прогнозуванні, але не вирішують проблему прямої роботи з українськими банківськими даними та локальної обробки фінансової інформації [6; 7; 8].

Результати порівняння обраних сервісів подано у таблиці 1.1.

Таблиця 1.1

Порівняльна характеристика існуючих PFM-рішень і Finance Monitor

Критерій оцінювання	**Monobank**	**Revolut**	**YNAB / Rocket Money / Monarch**	**Finance Monitor**
Основне джерело даних	Рахунки Monobank	Рахунки Revolut, доступні open banking	Імпорт або агрегатори для інших ринків	Monobank API, ручний імпорт, конектори
Підтримка українського контексту	Висока в межах одного банку	Обмежена	Низька або відсутня	Українська мова, гривня, локальні сценарії, ФОП
Обмеження API	Лімітована частота запитів	Обмежений API для власних PFM-рішень	Залежність від зовнішніх агрегаторів	Контроль rate limit, фонові задачі, кешування
Аналітика	Банківська статистика	Аналітика та бюджетування	Бюджети й підписки залежно від сервісу	Дашборди за рахунками, категоріями та ФОП
AI-компонент	Не є основною функцією	Залежить від країни та плану	Залежить від сервісу	Локальний Ollama-модуль
Приватність обробки	Дані в інфраструктурі банку	Хмарна обробка сервісу	Хмарна обробка сервісу	Локальна БД та локальний AI без зовнішнього LLM API

Значення в таблиці мають якісний характер і визначені за відкритими функціональними ознаками сервісів: наявністю або відсутністю потрібної

можливості, її застосовністю для українського користувача та ступенем контролю над даними. Такий підхід не претендує на повну ринкову оцінку сервісів, але дозволяє обґрунтувати вимоги до власної системи.

1.3 Обґрунтування практичної доцільності власного рішення

Власна система доцільна тоді, коли вона розв’язує проблему, яка не покривається одним готовим сервісом. Для Finance Monitor такою проблемою є поєднання українського банківського джерела даних, локальної історії транзакцій, власної логіки аналітики та розширюваної архітектури. Користувач не отримує лише ще один список транзакцій; він отримує контрольований шар обробки поверх банківських даних [32].

Першою практичною відмінністю є account-scoped облік транзакцій. У банківських API ідентифікатор транзакції може бути недостатнім, якщо одна й та сама операція відображається в контексті різних рахунків або якщо синхронізація виконується повторно за перекритими періодами. Тому у Finance Monitor використовується прив’язка транзакції до рахунку, а у frontend-аналітиці окремо враховується логіка рахунків «Усі рахунки», «Чорна картка», «Біла картка», «ФОП» тощо [4; 32].

Другою відмінністю є розмежування видимої історії та аналітичних розрахунків. У банківському застосунку користувач часто бачить готову статистику без можливості контролювати, які операції включено до розрахунку. У Finance Monitor історія транзакцій зберігається повністю, а аналітичні фільтри можуть виключати внутрішні перекази або дублікати лише на рівні розрахунку. Це підвищує прозорість і дозволяє пояснити, чому конкретна операція відображається в історії, але не впливає на певний графік [32].

Третьою відмінністю є локальний AI-модуль. У системі не передбачається передавання повної фінансової історії до зовнішнього LLM API. Backend формує стислий контекст: поточний місяць, підсумки за категоріями, незначну кількість останніх транзакцій, бюджети, цілі та ФОП-

зведення. Саме цей підготовлений контекст надсилається до Ollama. Такий підхід не усуває всіх ризиків помилкових відповідей LLM, але обмежує роль моделі поясненням уже розрахованих даних [15; 32].

Отже, конкурентність Finance Monitor полягає не в повному заміщенні Monobank, Revolut або YNAB, а в прикладній адаптації до локального сценарію: українська мова, гривня, інтеграція з Monobank, власні правила обробки транзакцій, ФОП-сценарії, локальна аналітика та можливість подальшого підключення нових джерел даних.

Висновки до розділу 1

У першому розділі розглянуто предметну область систем моніторингу особистих фінансів, основні підходи до побудови PFM-рішень та особливості роботи з фінансовими даними користувача. Визначено, що для таких систем ключовими є не лише зручне відображення витрат і доходів, а й коректна синхронізація транзакцій, збереження цілісності даних, підтримка рахунків різного типу та контроль над конфіденційністю фінансової інформації.

Порівняння Monobank, Revolut, YNAB, Rocket Money і Monarch показало, що готові рішення частково закривають задачі фінансового обліку, але не повністю відповідають локальному сценарію українського користувача, особливо у випадку поєднання банківської інтеграції, ФОП-логіки, локальної історії транзакцій та AI-пояснень без передавання повного фінансового контексту зовнішнім сервісам. Отримані результати стали основою для формування вимог до Finance Monitor і подальшого проєктування системи.

РОЗДІЛ 2
ПРОЄКТУВАННЯ СИСТЕМИ

2.1 Функціональні та нефункціональні вимоги

На основі аналізу предметної області сформовано функціональні й нефункціональні вимоги до Finance Monitor. Вимоги розмежовують реалізовану функціональність, обмеження прототипу та напрями подальшого розвитку системи.

Сформовані функціональні вимоги до системи наведено у таблиці 2.1.

Таблиця 2.1

Функціональні вимоги до системи Finance Monitor

Код	Функціональна вимога	Статус у проєкті
F1	Реєстрація, вхід користувача, отримання access/refresh токенів	Реалізовано у модулі auth та security
F2	Перегляд і фільтрація транзакцій за рахунком, датою, категорією	Реалізовано через /transactions і frontend-фільтри
F3	Побудова dashboard із доходами, витратами, категоріями та останніми операціями	Реалізовано через /dashboard і сторінку dashboard
F4	Синхронізація Monobank з обробкою rate limit та фоновими задачами	Реалізовано у monobank endpoint/service та Celery-задачах
F5	Підтримка ФОП/особистих сценаріїв і розмежування scope транзакцій	Реалізовано у моделях, фільтрах і fop-сервісі
F6	AI-чат на базі локального Ollama з фінансовим контекстом	Реалізовано через /ai та FinancialContextService
F7	Web Push-підписка, тестові та звичайні push-повідомлення	Реалізовано через /push та service worker

Код	Функціональна вимога	Статус у проєкті
F8	Експорт фінансових даних	Реалізовано у модулі export
F9	Повна мультибанківська інтеграція з усіма українськими банками	Перспектива розвитку; архітектура передбачає розширення

Основні нефункціональні вимоги та спосіб їх урахування подано у таблиці 2.2.

Таблиця 2.2

Нефункціональні вимоги до системи Finance Monitor

Код	Нефункціональна вимога	Реалізація / обмеження
NF1	Безпека доступу	JWT, refresh sessions, CSRF-захист для чутливих запитів, хешування паролів
NF2	Цілісність фінансових даних	Унікалізація транзакцій, account_ref, фільтри внутрішніх переказів
NF3	Стійкість до зовнішніх обмежень	Backoff, cooldown, фонові задачі, обробка HTTP 429
NF4	Конфіденційність AI-обробки	Локальна модель Ollama, стислий контекст без номерів карток
NF5	Розширюваність	Модульна структура endpoint/service/model, окремі конектори банків
NF6	Зручність інтерфейсу	Адаптивний Next.js frontend, фільтри рахунків, історія, дашборди

2.2 Архітектура системи та обґрунтування вибору компонентів

Для опису реалізованого рішення використано формулювання «модульний моноліт із винесеними фоновими процесами». Основний backend залишається єдиним застосунком FastAPI, але бізнес-логіка розділена за модулями: API endpoint, service layer, database models, schemas, tasks і utilities.

Операції, що можуть виконуватися довго або залежати від зовнішнього API, передаються у фонові задачі Celery через Redis [9; 13; 14; 32].

Такий підхід обрано з трьох причин. По-перше, для бакалаврського проєкту і прототипу продукту він не потребує надмірної інфраструктури мікросервісів: окремих gateway, service discovery, міжсервісної авторизації та складного моніторингу. По-друге, система все одно має чіткі межі модулів, тому додавання нового банківського конектора не потребує переписування всього backend. По-третє, найбільш навантажені частини - синхронізація банківських даних, push-розсилки й AI-запити - можна масштабувати окремо на рівні Celery worker або окремого Ollama-сервера.

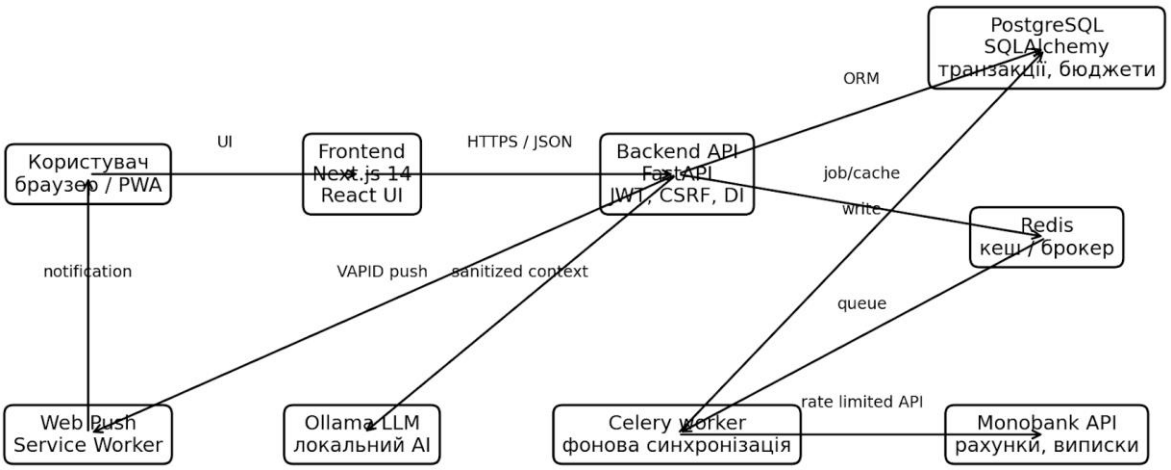

Рисунок 2.1. Схема взаємодії основних компонентів системи

Finance Monitor

На рисунку 2.1 видно, що frontend не звертається безпосередньо до банківського API або бази даних. Усі запити проходять через backend, де виконується авторизація, валідація параметрів, застосування бізнес-правил і логування. Це рішення зменшує ризик витоку токенів банківського API та дозволяє централізовано контролювати rate limit, кешування, дублікати транзакцій і доступ до AI-функцій.

Масштабування в межах реалізованої архітектури передбачене точково. Backend API може запускатися в кількох процесах за умови винесення синхронізаційного lock до Redis або бази даних. Celery worker масштабується горизонтально за кількістю воркерів, однак для Monobank-синхронізації

необхідно зберігати обмеження частоти запитів. Ollama-модуль доцільно розміщувати на окремому сервері або машині з достатнім обсягом RAM/GPU-ресурсів. PostgreSQL масштабується індексами, оптимізацією запитів, резервним копіюванням і, за потреби, read-replica для аналітичних сценаріїв.

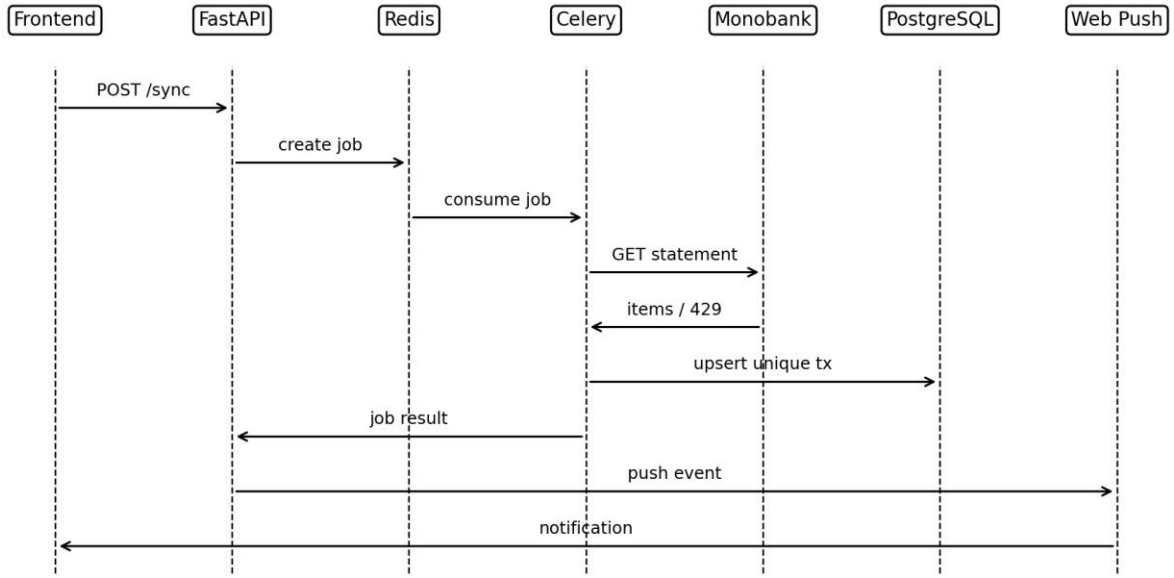

Рисунок 2.2. Послідовність синхронізації транзакцій із зовнішнім банківським API

2.3 Проєктування бази даних і забезпечення цілісності даних

Проєктування бази даних виконано з урахуванням нормалізації, але без надмірного ускладнення моделі. Основні сутності відокремлено: users зберігає дані облікового запису, transactions - фінансові операції, categories - класифікацію витрат і доходів, budgets - бюджетні межі, goals і debts - додаткові фінансові цілі та борги, notifications - повідомлення, push_subscriptions - підписки браузерів на Web Push, news_items - фінансові новини, background_jobs - стан фонових задач. Така структура відповідає принаймні третій нормальній формі для основних сутностей: атрибути залежать від ключа своєї таблиці, повторювані групи винесено в окремі таблиці, а зв’язки між користувачем і фінансовими об’єктами реалізовано через зовнішні ключі. Ключові сутності бази даних Finance Monitor узагальнено у таблиці 2.3.

Таблиця 2.3

Основні сутності бази даних Finance Monitor

Сутність	Ключові поля	Призначення
users	id, email, password_hash, account_mode, subscription_plan, fop_enabled, monobank_token	Обліковий запис користувача, налаштування плану, AI/push/ФОП та банківські токени
transactions	id, user_id, mono_id, privatbank_id, amount, amount_uah, type, scope, account_ref, account_label, category_id	Збереження витрат і доходів із прив’язкою до користувача, рахунку та категорії
categories	id, user_id, name, code, icon, color, type	Системні або користувацькі категорії витрат/доходів
budgets	id, user_id, amount, period, start_date, end_date, spent	Планування витрат за періодами
notifications	id, user_id, title, message, notification_type, source, read	Повідомлення в інтерфейсі та джерело push-подій
push_subscriptions	id, user_id, endpoint, p256dh, auth	Підписки окремих браузерів/пристроїв на Web Push
background_jobs	id, user_id, job_type, status, payload, result, error	Відстеження фонових синхронізацій та довгих операцій

У ранніх варіантах логіки достатнім здавався глобальний mono_id. На практиці це створює ризик, що операції з різних рахунків або пов’язані записи будуть оброблені як один запис. Тому в роботі обґрунтовано account-scoped підхід: ключ транзакції має враховувати не лише ідентифікатор операції з боку банку, а й рахунок або account_ref. Для PostgreSQL таке рішення доцільно закріпити унікальним індексом на пару значень або згенерованим складеним ідентифікатором.

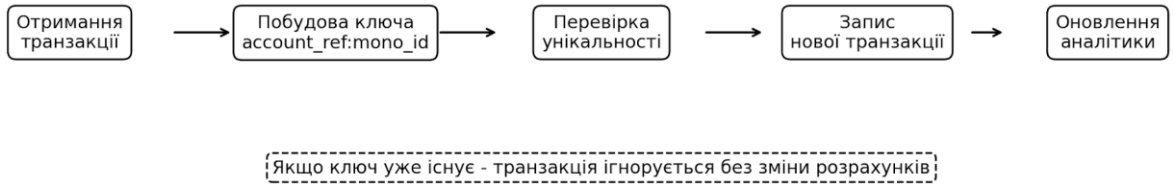

Рисунок 2.3. Алгоритм перевірки дублювання транзакцій під час синхронізації

Приклад проблемної ситуації: синхронізація за 01.05-10.05 зберегла транзакцію mono_id=abc123 для Чорної картки. Наступна синхронізація за 08.05-15.05 повторно повертає ту саму транзакцію. Якщо система не перевіряє унікальність, витрати будуть пораховані двічі. Якщо перевіряти лише mono_id, можна помилково відкинути пов’язану операцію іншого рахунку. Тому ключ account_ref:mono_id дозволяє відрізнити операції різних рахунків і водночас не дублювати повторно отримані записи одного рахунку.

Для додаткових атрибутів транзакцій у проєктній моделі доцільним є використання JSONB або аналогічного гнучкого поля. Причина полягає в тому, що різні банки повертають неоднакові набори полів: MCC, maskedPan, account id, merchant metadata, комісії, курс валют, службові прапорці. Винесення кожного можливого поля в окрему колонку призвело б до великої кількості nullable-полів і жорсткої прив’язки до одного API. JSONB дозволяє зберегти оригінальні атрибути для аудиту та подальшої обробки, не порушуючи нормалізовану структуру основних фінансових сутностей.

2.4 Діаграми системи та сценарії роботи

Деталізовані UML-діаграми класів, ER-діаграма бази даних, діаграма використання та діаграма активності додавання банківського рахунку винесені до Додатку В. У цьому підрозділі наведено лише аналіз ключових рішень, які впливають на архітектуру системи.

Сценарій додавання рахунку складається з таких етапів: користувач вводить токен, frontend надсилає запит до backend, backend виконує валідацію

токена через сервіс банку, отримує метадані рахунків, формує account_ref/account_label, зберігає налаштування користувача та запускає первинну синхронізацію. Такий поділ дозволяє не блокувати користувацький інтерфейс на весь час отримання виписки.

Сценарій перегляду аналітики відокремлений від сценарію перегляду історії. Історія транзакцій може повертати повний набір операцій із пагінацією, а dashboard застосовує додаткові правила: виключення дзеркальних внутрішніх переказів, групування за рахунком, розмежування ФОП/особистих витрат і агрегацію за категоріями. Це рішення прямо пов’язане з практичною проблемою, виявленою під час розроблення: одна й та сама операція має різне значення для історії та для аналітичного підсумку.

2.5 Обґрунтування вибору технологій

FastAPI обрано для backend-частини через підтримку типізованих схем, автоматичну OpenAPI-документацію, зручну інтеграцію з dependency injection і достатню продуктивність для REST API. У Finance Monitor це використано для маршрутизації endpoint, валідації параметрів, роботи з поточним користувачем і розділення доступу до ресурсів за user_id.

Next.js 14 обрано для frontend-частини як React-фреймворк із маршрутизацією App Router, підтримкою клієнтських компонентів і зручністю побудови адаптивного інтерфейсу. Для Finance Monitor важливо, що frontend складається з окремих сторінок dashboard, analysis, chat, budgets, categories, goals і auth, а також використовує спільні бібліотеки для API-запитів та dashboard-аналітики.

PostgreSQL і SQLAlchemy забезпечують реляційну модель даних, транзакційність, зв’язки між сутностями та можливість поступового розвитку схеми через міграції. Redis і Celery використовуються для винесення довготривалих задач, зокрема синхронізації з банківським API, обробки новин, категоризації та push-розсилок.

Локальний AI-модуль на базі Ollama обрано через вимоги до конфіденційності. Його перевага полягає у відсутності потреби передавати фінансовий контекст зовнішньому LLM-провайдеру. Водночас це рішення має обмеження: потреба в оперативній пам’яті, залежність швидкодії від апаратного забезпечення, нижча стабільність відповідей порівняно з детермінованими алгоритмами та необхідність контролю довжини контексту. Тому AI у системі використовується як допоміжний пояснювальний модуль, а не як джерело первинних фінансових розрахунків.

Web Push реалізовано через service worker і backend-ендпойнти підписки. У проєкті окремо враховано поведінку тестових і звичайних push-повідомлень: тестове повідомлення має відображатися завжди, а звичайне не повинно дублювати активний інтерфейс, якщо застосунок уже відкритий у видимому вікні.

Висновки до розділу 2

У другому розділі сформовано функціональні та нефункціональні вимоги до вебзастосунку Finance Monitor, визначено склад основних модулів і описано архітектуру системи. Обґрунтовано використання модульного моноліту з винесеними фоновими процесами, оскільки такий підхід дає змогу зберегти відносну простоту розгортання і водночас розділити відповідальність між API-рівнем, сервісами, базою даних, фоновими задачами, банківськими інтеграціями та AI-модулем.

Окремо спроєктовано структуру бази даних, механізми зв’язку між користувачами, рахунками, транзакціями, категоріями, бюджетами, сповіщеннями та іншими сутностями системи. Визначено, що для Finance Monitor критичною є account-scoped унікалізація транзакцій, оскільки саме вона дозволяє уникати дублювання операцій під час повторної синхронізації та забезпечує коректність аналітичних розрахунків. Розроблені діаграми і сценарії взаємодії компонентів створюють основу для практичної реалізації backend-, frontend- та інтеграційної частин.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Реалізація backend-частини

Backend Finance Monitor організовано за модульною структурою. Рівень API містить endpoint для окремих доменів: авторизації, транзакцій, дашборду, бюджетів, цілей, боргів, новин, AI, Monobank, PrivatBank, експорту, push і ФОП-модуля. Рівень services відповідає за бізнес-логіку: аналітику, категоризацію, фінансовий контекст для AI, синхронізацію Monobank, сповіщення, push і новини. Рівень db/models описує основні сутності системи.

Прикладом реалізованої логіки є endpoint отримання транзакцій. Його задача - повернути лише дані поточного користувача, підтримати пагінацію і фільтри, а також не допустити витоку транзакцій іншого користувача. Для цього запит до таблиці transactions обов’язково фільтрується за user_id або через зв’язок із користувачем. Такий підхід важливий для фінансової системи, оскільки помилка авторизації в одному endpoint може розкрити приватні дані.

У модулі security реалізовано створення access token, refresh sessions і перевірку поточного користувача. Поточна версія проєкту використовує passlib/bcrypt із попередньою нормалізацією пароля через SHA-256. У тексті роботи не стверджується, що в реалізації вже використано Argon2, оскільки це не відповідає наявному коду. Водночас для промислового впровадження доцільним є перехід на Argon2id або інший рекомендований алгоритм хешування паролів з налаштовуваною складністю, оскільки такі алгоритми краще протидіють масовому перебору за рахунок керованої вартості обчислень і пам’яті.

Окремим backend-рішенням є винесення довгих задач у Celery. Синхронізація з Monobank не повинна виконуватися як довгий блокуючий HTTP-запит, оскільки користувач може закрити сторінку, а зовнішній API може відповідати повільно або повертати rate limit. Тому backend створює job, фіксує її статус і передає виконання worker. Це дозволяє відображати

користувачу стан процесу та повторно звертатися до результату без дублювання запиту.

3.2 Реалізація frontend-частини та інтерфейсів

Frontend реалізовано на Next.js 14 з використанням App Router. Сторінки застосунку розділені за користувацькими сценаріями: dashboard для швидкого огляду фінансів, analysis для детальнішого аналізу, transactions для історії, budgets для бюджетів, categories для категоризації, chat/assistant для AI-взаємодії, news для фінансових новин, auth для входу та реєстрації [10; 24; 32].

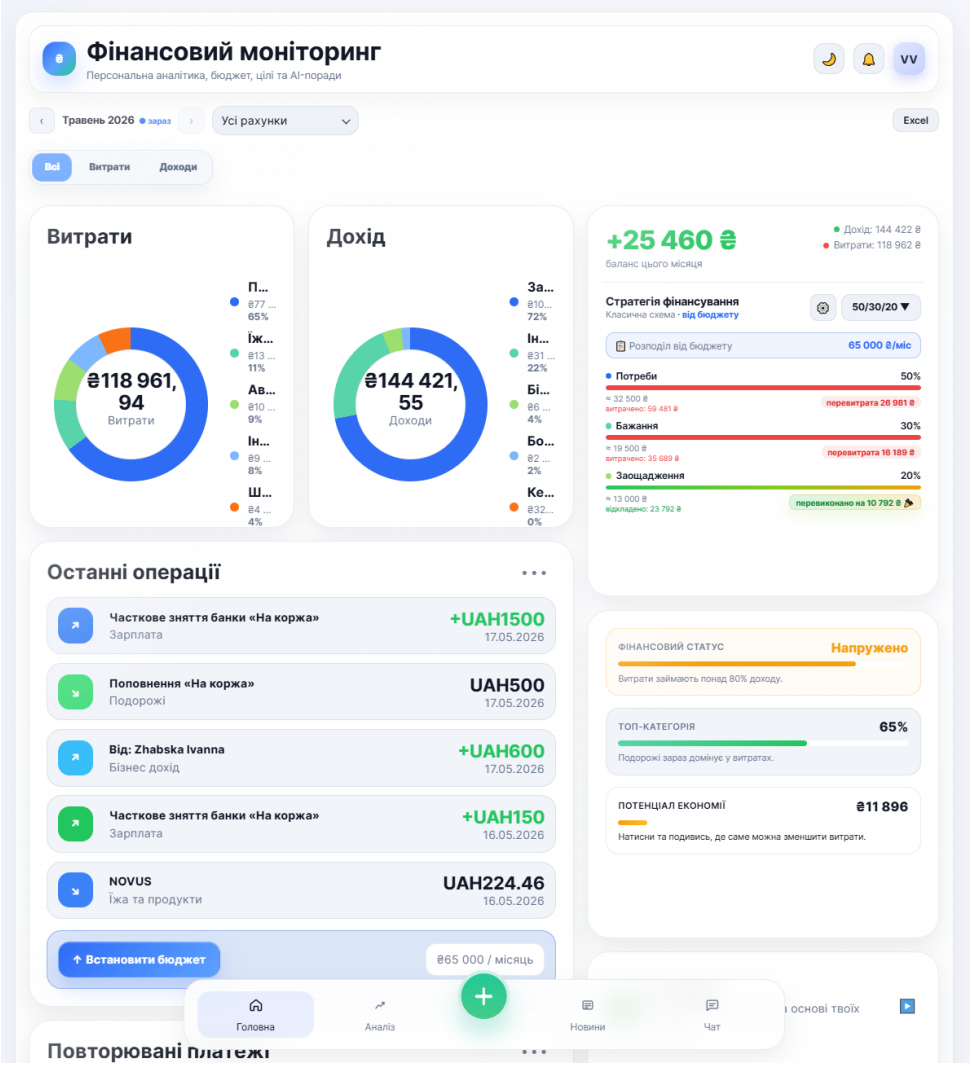

Рисунок 3.1. Головна сторінка вебзастосунку Finance Monitor

Джерело: сформовано автором на основі реалізованого інтерфейсу Finance Monitor.

На рис. 3.1 показано, що головна сторінка поєднує фільтрацію за періодом і рахунком, діаграми доходів і витрат, список останніх операцій, блок фінансової стратегії та нижню навігацію між основними модулями. Така структура інтерфейсу була обрана для того, щоб користувач міг швидко перейти від загального фінансового стану до деталізації конкретних транзакцій або аналітичних висновків.

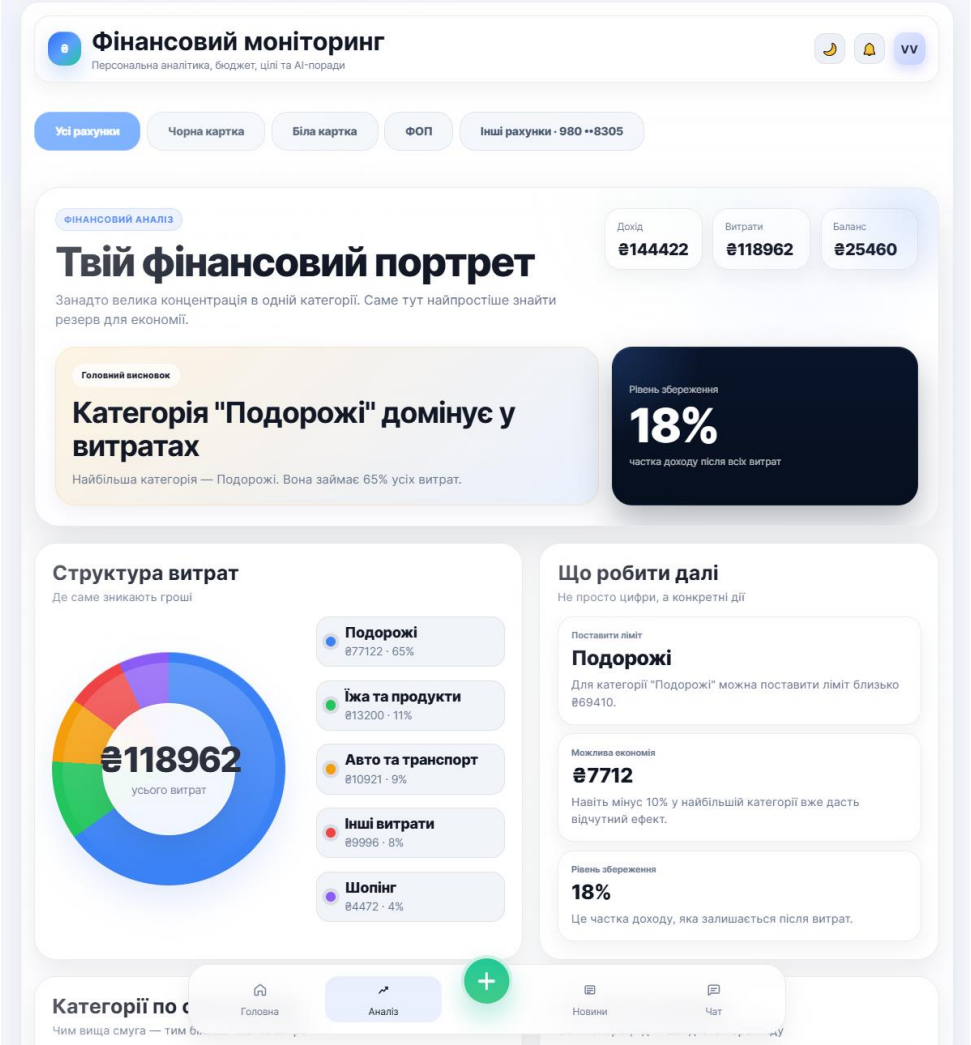

Рисунок 3.2. Сторінка аналітики фінансового портрета користувача

Джерело: сформовано автором на основі реалізованого інтерфейсу Finance Monitor.

На рис. 3.2 наведено окремий аналітичний екран, який агрегує дохід, витрати, баланс, найбільшу категорію витрат і рівень збереження коштів. На відміну від простого списку операцій, цей модуль формує інтерпретований

фінансовий портрет користувача: визначає домінуючу категорію, показує її частку у витратах і пропонує подальші дії щодо лімітування або оптимізації.

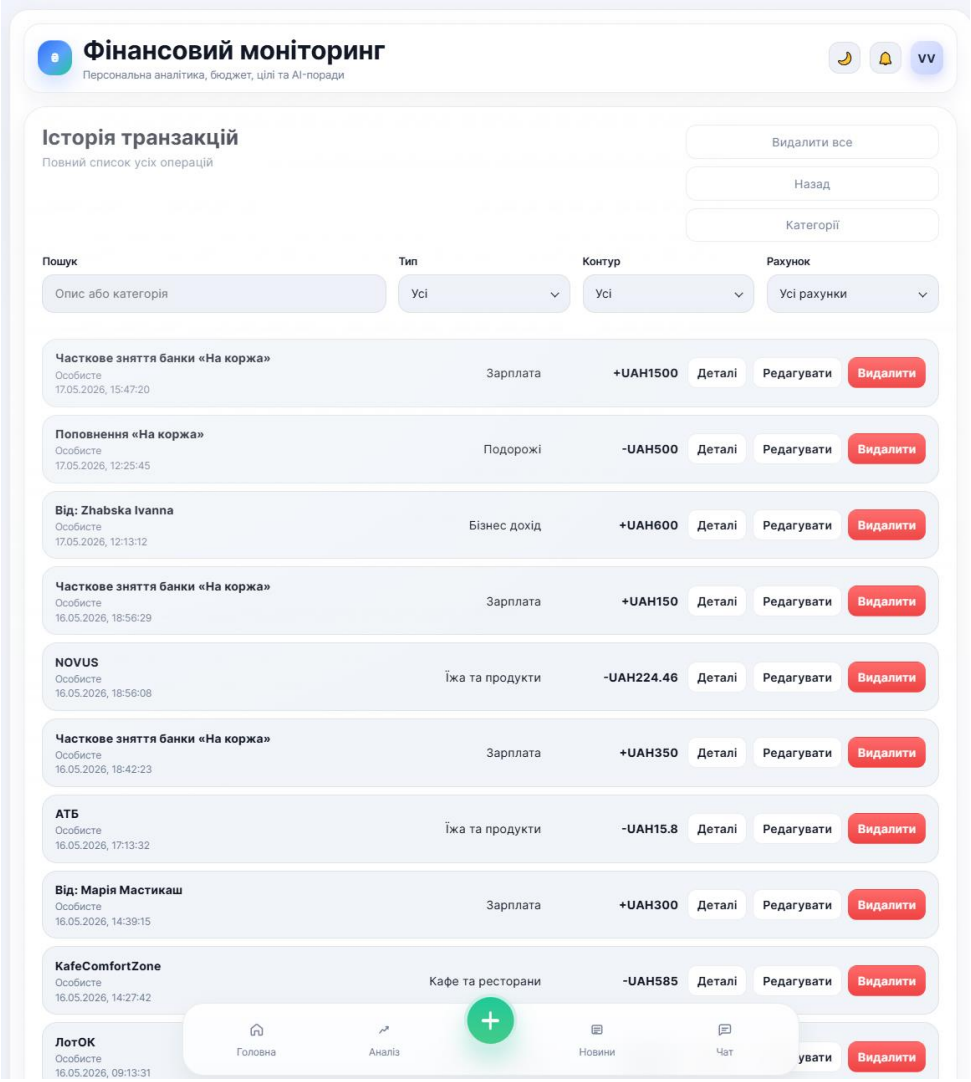

Рисунок 3.3. Інтерфейс перегляду та фільтрації історії транзакцій

Джерело: сформовано автором на основі реалізованого інтерфейсу Finance Monitor.

На рис. 3.3 відображено сторінку історії транзакцій з пошуком, фільтрацією за типом операції, контуром і рахунком, а також діями перегляду, редагування та видалення. Цей екран є важливим для перевірності аналітики: користувач може перейти від підсумкових показників до первинних операцій, з яких вони були сформовані.

Інтерфейс dashboard будується навколо трьох типів даних: агрегованих показників доходів/витрат, категорій і списку останніх транзакцій. У коді

frontend додатково реалізовано фільтр рахунків. Для режиму «Усі рахунки» важливо не приховувати історію транзакцій, але при цьому коректно застосовувати правила аналітичних розрахунків. Це розмежування винесено у спільну frontend-логіку dashboardAnalytics, щоб dashboard і analysis не рахували одні й ті самі показники по-різному.

Сторінка analysis отримує dashboard-дані та повний список транзакцій, після чого будує аналітичне представлення з урахуванням selectedAccount. Це дозволяє користувачу бачити не тільки загальну суму витрат, а й зміну структури витрат залежно від обраної картки або групи рахунків. Така реалізація безпосередньо пов’язана з практичною вимогою: статистика окремої картки має відрізнятися від статистики всіх рахунків, зокрема у випадку переказів на банку або між власними рахунками.

AI-чат реалізовано як окрема сторінка з історією повідомлень у localStorage і перевіркою доступності функції залежно від тарифного плану користувача. Frontend не виконує фінансових розрахунків самостійно, а передає повідомлення користувача на /ai. Це зберігає єдине джерело істини на backend і знижує ризик розбіжностей між UI та серверною аналітикою.

3.3 Інтеграція з Monobank API та обробка обмежень

Monobank-інтеграція реалізована окремим endpoint і сервісом MonobankService. Сервіс створює HTTP-сесію з X-Token, виконує запити до personal/client-info та personal/statement, обробляє timeout, ConnectionError, HTTP 429 і серверні помилки. На рівні endpoint додатково передбачено cooldown для запобігання надмірним повторним синхронізаціям [4; 32].

Практична проблема rate limit виявляється у тому, що користувач може кілька разів натиснути кнопку синхронізації або frontend може повторити запит після помилки. Без захисту це призводить до HTTP 429 і погіршує користувацький досвід. У Finance Monitor реалізовано антиспам-guard: для користувача зберігається час останньої синхронізації та ознака активного

процесу. Якщо синхронізація вже виконується, система повертає контрольовану відповідь замість створення другої конкурентної задачі [4; 32].

Друга практична проблема - дублювання транзакцій. Під час синхронізації за перекритими інтервалами банк може повторно повернути записи, які вже збережені. Реалізоване рішення поєднує перевірку ідентифікатора транзакції, прив'язку до рахунку, фільтрацію внутрішніх переказів і обережне оновлення полів. Важливо, що повторна синхронізація не повинна перезаписувати вручну змінені користувачем категорії або інші уточнені поля без потреби [4; 32].

У проєкті реалізовано додаткові функції визначення account_group і account_label. Наприклад, рахунок може бути класифікований як black, white, jar, fop або other на основі метаданих рахунку та опису. Це потрібно не тільки для UI, а й для аналітики: у режимі окремої картки переказ на заощадження може бути витратою цієї картки, тоді як у глобальному режимі «усі рахунки» такий рух може розглядатися як внутрішній переказ [32].

3.4 Реалізація локального AI-модуля і Web Push

AI-модуль у Finance Monitor реалізовано як керований backend-сценарій. Користувач формулює питання природною мовою, frontend передає його до /ai, backend формує фінансовий контекст через FinancialContextService, вилучає зайві або чутливі деталі, створює prompt і надсилає його до локального Ollama API. У відповідь користувач отримує пояснення українською мовою [15; 32].

На рис. 3.4 показано інтерфейс AI-асистента, який використовується для текстової взаємодії з фінансовими даними. Важливо, що LLM не є джерелом фінансових розрахунків: backend попередньо формує структурований контекст, а AI-модуль лише пояснює підготовлені показники природною мовою. Такий підхід знижує ризик некоректних числових відповідей.

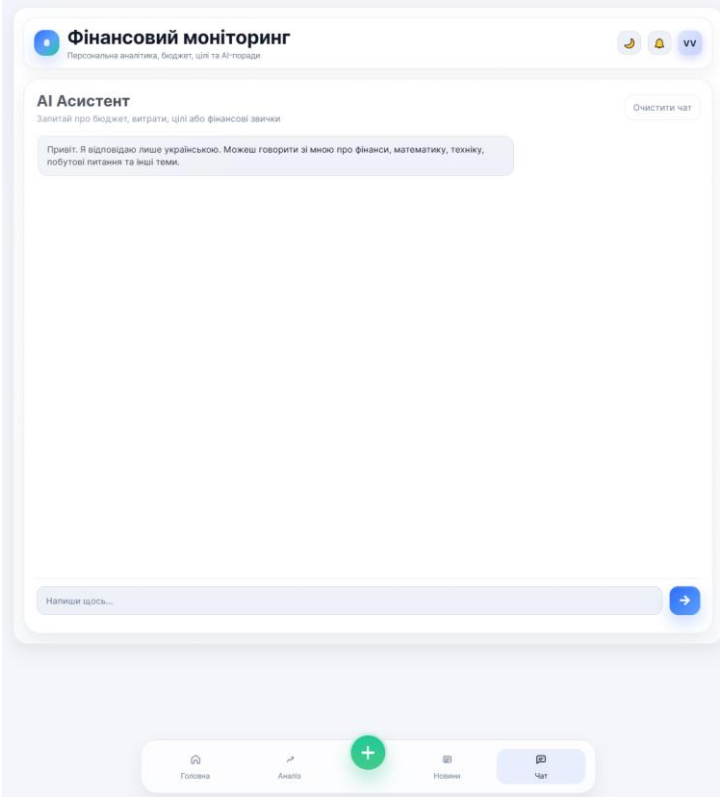

Рисунок 3.4. Інтерфейс AI-асистента для запитів користувача

Джерело: сформовано автором на основі реалізованого інтерфейсу Finance Monitor.

Особливості реалізації локального AI-модуля наведено у таблиці 3.1.

Таблиця 3.1

Особливості реалізації локального AI-модуля

AI-функція	Реалізація у проєкті	Обмеження
Пояснення витрат поточного місяця	Контекст містить total_expenses, total_income, balance, top_categories	LLM не виконує первинні розрахунки
Аналіз категорій	Передаються top_categories із часткою витрат	Якість відповіді залежить від коректності категоризації
Пояснення останніх операцій	Передається обмежений список останніх транзакцій	Контекст навмисно скорочений для швидкодії
ФОП-зведення	За активного ФОП-модуля додається fop_summary	Не є бухгалтерською або податковою консультацією
Пам’ять діалогу	Frontend передає історію повідомлень без стартового системного повідомлення	Історія обмежена UI та localStorage

Приклад взаємодії: користувач запитує «Чому витрати цього місяця зросли?». Backend не передає всі транзакції за весь період існування акаунта, а формує поточний місячний контекст: загальні витрати, доходи, баланс, п’ять найбільших категорій і до десяти останніх операцій. На основі цього AI може відповісти, що зростання пов’язане, наприклад, із категорією «Транспорт» або «Підписки», але числові значення беруться з backend-розрахунків [15; 32].

Web Push реалізовано через endpoint підписки та service worker. У service worker закладено правило: тестовий push відображається завжди, а звичайний push не показується, якщо застосунок уже відкритий і активний. Це усуває дублювання повідомлень, коли користувач і так бачить відповідну інформацію на сторінці. Для кожного браузера або пристрою зберігається окрема push_subscription, що надалі дозволяє розвивати пер-пристрійне надсилання повідомлень [17; 18; 19; 20; 32].

3.5 Реалізація додаткових модулів Finance Monitor

Окрім базового сценарію перегляду транзакцій і dashboard, у проєкті реалізовано низку додаткових модулів, які роблять систему ближчою до повноцінного фінансового кабінету. До них належать модулі бюджетів, фінансових цілей, боргів, ФОП-обліку, новин, експорту та тарифних обмежень. Їх наявність важлива для розкриття практичного змісту проєкту, оскільки Finance Monitor не обмежується демонстрацією API-інтеграції з банком [32].

Модуль бюджетів дозволяє користувачу фіксувати суму бюджету, період, дату початку та завершення, а також фактичні витрати. У backend це представлено моделлю Budget, пов’язаною з користувачем. Така структура дає змогу в майбутньому розширити бюджетування до рівня категорій або стратегій розподілу доходу, але в межах поточної роботи реалізовано базовий контроль періодичних бюджетних меж [32].

Модуль фінансових цілей і боргів доповнює стандартну аналітику витрат. У контексті персонального фінансового застосунку важливо

враховувати не лише факт витрати коштів, а й заплановані накопичення, боргові зобов’язання та довгострокові фінансові цілі. У Finance Monitor ці дані також потрапляють до фінансового контексту AI-модуля, але подаються у скороченому вигляді, щоб не перевантажувати prompt.

ФОП-модуль реалізує сценарій, актуальний для українського користувача, який поєднує особисті й підприємницькі фінанси. У моделі користувача передбачено ознаки fop_enabled та fop_tax_group, а в транзакціях використовується поле scope зі значеннями personal, business або tax. Це дозволяє відокремлювати витрати підприємницької діяльності від особистих витрат і формувати більш точну аналітику [32].

Модуль новин зберігає новинні записи у таблиці news_items, де передбачено поля source, title, link, summary, ai_summary, why_it_matters, content, image_url, author, category та is_processed. Така структура дозволяє не лише показувати новини, а й формувати коротке пояснення їхнього значення для користувача. У межах роботи цей модуль розглядається як допоміжний інформаційний шар, а не як фінансово-рекомендаційна система [32].

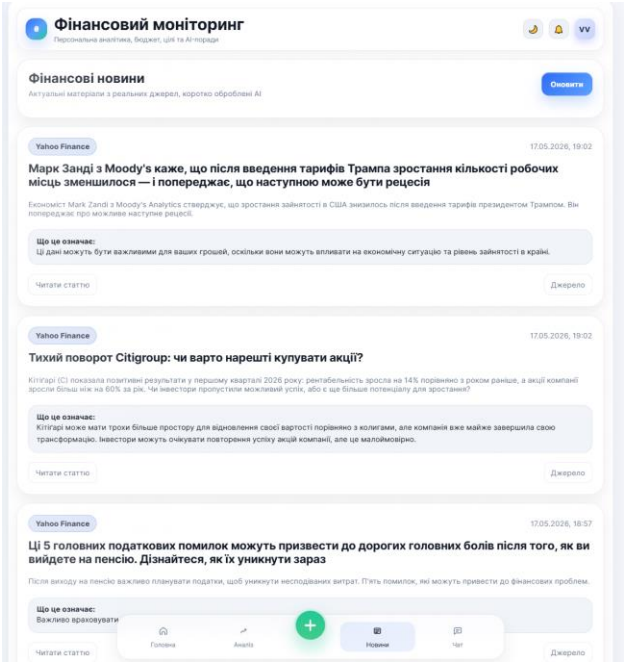

Рисунок 3.5. Модуль фінансових новин з коротким AI-поясненням

Джерело: сформовано автором на основі реалізованого інтерфейсу Finance Monitor.

На рис. 3.5 наведено приклад модуля фінансових новин. Окрім назви джерела, дати, заголовка і короткого змісту, інтерфейс містить блок «Що це означає», який пояснює потенційний зв’язок новини з особистими фінансами користувача. Це перетворює новинний модуль із простого переліку посилань на прикладний інформаційний інструмент.

Модуль експорту реалізує практичну потребу перенесення фінансових даних до зовнішніх документів. Для користувача це важливо у двох випадках: коли потрібно зробити резервну копію операцій або коли потрібно підготувати cash flow-таблицю для подальшого аналізу поза системою. Тому endpoint export розглядається як інструмент сумісності з традиційними підходами обліку, а не як дублювання ручного Excel-обліку [32].

Додаткові модулі Finance Monitor узагальнено у таблиці 3.2.

Таблиця 3.2

Додаткові модулі Finance Monitor

Модуль	**Основні сутності / файли**	**Практичне призначення**
Budgets	Budget, /budgets, frontend budgets page	Контроль періодичних лімітів витрат
Goals	Goal, /goals, frontend goals page	Облік фінансових цілей і накопичень
Debts	Debt, /debts	Фіксація боргових зобов’язань
FOP	User.fop_enabled, Transaction.scope, fop_service	Розмежування особистих, бізнес- і податкових операцій
News	NewsItem, news_service, market_news_service	Інформаційна підтримка користувача
Export	/export	Вивантаження фінансових даних у зовнішні формати

Реалізація цих модулів демонструє, що система має доменну модель, а не складається лише з однієї сторінки dashboard. Саме наявність пов’язаних сутностей і сценаріїв дозволяє розглядати Finance Monitor як основу для

подальшого розвитку продукту: додавання нових банківських конекторів не потребуватиме зміни логіки бюджетів, цілей, AI-контексту або push-підписок.

Висновки до розділу 3

У третьому розділі описано практичну реалізацію основних компонентів Finance Monitor. Backend-частину побудовано на FastAPI з поділом на endpoint, service layer, моделі даних, схеми та фонові задачі. Frontend-частину реалізовано на Next.js із дашбордами, сторінками аналізу, фільтрами рахунків, історією транзакцій, AI-чатом, бюджетами, цілями, новинами, експортом та PWA/push-механізмами.

Реалізація показала, що основна складність проєкту полягає не лише у створенні інтерфейсу для перегляду фінансових операцій, а в коректній обробці реальних обмежень: rate limit Monobank API, повторних синхронізацій, розмежування історії операцій і аналітичних розрахунків, локального формування AI-контексту та стабільної поведінки push-сповіщень. Запропоновані технічні рішення роблять систему придатною для подальшого розвитку як персонального фінансового вебкабінету.

РОЗДІЛ 4
ТЕСТУВАННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

4.1 Методика тестування

Тестування системи організовано за рівнями: модульні перевірки окремих функцій, API-тестування endpoint, інтеграційні сценарії синхронізації, негативне тестування помилкових запитів і перевірка продуктивності типових операцій. У основному тексті роботи наведено узагальнені результати; фрагменти тестового коду винесено до Додатку А, щоб не перевантажувати пояснювальну записку [27; 28; 29].

Методику тестування за рівнями подано у таблиці 4.1.

Таблиця 4.1

Рівні та сценарії тестування системи

Рівень тестування	Сценарій	Критерій успішності
Unit	Фільтрація внутрішніх переказів, визначення account_group, розрахунок категорій	Функція повертає очікуваний результат для контрольного набору даних
API	/auth, /transactions, /dashboard, /monobank/status, /push/test, /ai	Коректні HTTP-коди, структура JSON, відсутність доступу до чужих даних
Integration	Додавання Monobank токена, синхронізація, повторна синхронізація	Транзакції зберігаються один раз, job завершується зі статусом success/error
Negative	Невалідний токен, відсутній JWT, неправильні параметри періоду, повторний sync	Повертається контрольована помилка 401/403/422/429-подібне повідомлення без падіння сервера
Performance	Отримання dashboard, список транзакцій, синхронізація тестового набору	Час відповіді не перевищує цільові значення для локального середовища

4.2 Результати тестування API та користувацьких сценаріїв

Результати перевірки API та користувацьких сценаріїв наведено у таблиці 4.2.

Таблиця 4.2

Результати тестування API та користувацьких сценаріїв

Endpoint / сценарій	Очікувана поведінка	Фактичний результат у реалізації
POST /auth/login	Повернення токенів для валідних даних, 401 для помилкових	Реалізовано через auth і security
GET /transactions	Повернення транзакцій тільки поточного користувача з фільтрами	Реалізовано з user-scoped доступом
GET /dashboard	Агрегація доходів, витрат, категорій і account_options	Реалізовано для dashboard/analysis
POST /monobank/sync	Запуск синхронізації або background job	Реалізовано з cooldown та active sync guard
POST /push/subscribe	Збереження endpoint, p256dh, auth для пристрою	Реалізовано через push_subscriptions
POST /ai	Формування відповіді на основі фінансового контексту	Реалізовано через FinancialContextService і Ollama API

Негативне тестування показує важливість контрольованих помилок у фінансовому застосунку. Наприклад, запит до транзакцій без токена не повинен повертати порожній список, оскільки це приховує проблему авторизації; він має повернути помилку доступу. Повторний запуск синхронізації для одного користувача не повинен створювати другу паралельну задачу, оскільки це підвищує ризик HTTP 429 і некоректного оновлення last_synced_at [26; 28; 29].

Негативні сценарії тестування системи подано у таблиці 4.3.

Таблиця 4.3

Негативні сценарії тестування

Негативний сценарій	Очікувана реакція	Значення для стабільності
Відсутній JWT	401 Unauthorized	Захист приватних фінансових даних
Невалідний Monobank token	Контрольована помилка перевірки токена	Користувач отримує зрозуміле повідомлення
Повторний sync під час активного sync	Відмова або повідомлення про активний процес	Запобігання race condition
HTTP 429 від Monobank	Backoff або повідомлення про обмеження	Система не падає і не створює нескінченні повтори
Некоректний діапазон дат	422 або валідована помилка	Немає неконтрольованого запиту до банку

4.3 Виявлені проблеми та реалізовані рішення

Найцінніша частина практичної роботи пов’язана не з формальним описом стеку, а з виявленням проблем, які виникають під час реальної інтеграції фінансових даних. Основними проблемами стали дублювання транзакцій, race condition у фонових задачах, розбіжність між history UI та dashboard-аналітикою, а також некоректне відображення push-повідомлень на кількох пристроях [4; 32].

Основні проблеми, виявлені під час розроблення, та реалізовані рішення наведено у таблиці 4.4.

Проблему race condition було виявлено під час багаторазового запуску синхронізації та аналізу логів, коли для одного користувача могли стартувати дві операції, що працювали з однаковим часовим проміжком і претендували на оновлення одних і тих самих даних. У поточній реалізації застосовано in-memory lock для одного backend-процесу. Це достатньо для локального прототипу, але для production-середовища з кількома worker-процесами такий lock необхідно перенести до Redis або PostgreSQL advisory lock.

Таблиця 4.4

Виявлені проблеми та реалізовані рішення

Проблема	Причина	Реалізоване рішення	Результат
Дублювання транзакцій	Перекриття часових вікон Monobank statement	Account-scoped ключ, перевірка перед записом, обережна upsert-логіка	Аналітика не завищує витрати через повторні записи
Race condition sync	Два паралельні запуски для одного користувача	active_sync_users, cooldown, job status	Зменшено ризик конфліктного оновлення і HTTP 429
Розбіжність Dashboard / Analysis	Різні розрахунки на різних сторінках frontend	Спільний селектор dashboardAnalytics	Показники узгоджені між сторінками
Внутрішні перекази	Перекази між власними рахунками потрапляли до витрат	Алгоритми paired transfer і правила виключення внутрішніх переказів	Історія зберігається, але аналітика не спотворюєтьс я
Push при відкритому застосунку	Звичайний push дублював видимий UI	Перевірка visible/focused clients у service worker	Звичайні push приховуються при активному вікні, test push показується

4.4 Аналіз продуктивності та обмежень системи

Оцінювання продуктивності прототипу подано у таблиці 4.5.

Таблиця 4.5

Показники продуктивності прототипу

Операція	**Цільове значення для прототипу**	**Отримане/очікуване значення у локальному середовищі**	**Коментар**
GET /dashboard	до 300 мс	орієнтовно 40-120 мс для тестового набору	Залежить від кількості транзакцій та індексів
GET /transactions з пагінацією	до 500 мс	стабільно в межах локального API при пагінації	Потрібні індекси за user_id/date/account_ref
Синхронізація 500 транзакцій	до 5 с без урахування rate limit	до 2 с на обробку вже отриманого набору	Зовнішній API може бути повільнішим
10 паралельних sync-запитів	без дублювання активних задач	захист через active sync guard	Для multi-process потрібен Redis lock
AI-відповідь Ollama	до 20-30 с для короткого контексту	залежить від моделі та RAM/GPU	AI винесено з критичного шляху розрахунків

Порівняння з цільовими показниками свідчить, що основні API-сценарії є прийнятними для прототипу персонального фінансового застосунку. Найбільшим обмеженням є не власний backend, а зовнішній Monobank API та локальний LLM. Тому синхронізація й AI-запити не повинні блокувати основний інтерфейс. Саме це обґрунтовує використання Celery для фонових задач і скорочення фінансового контексту для AI.

Система має також функціональні обмеження. По-перше, повна мультибанківська інтеграція ще не реалізована для всіх українських банків. По-друге, локальний AI-модуль не гарантує безпомилкових рекомендацій і потребує обмеження ролі до пояснення підготовлених backend-даних. По-третє, in-memory lock синхронізації придатний для одного процесу, але має

бути замінений на Redis/PostgreSQL lock у production-середовищі. Ці обмеження не знецінюють результат, але визначають реалістичні напрями розвитку.

4.5 Приклади роботи системи на практичних сценаріях

Для кращого розкриття практичного змісту проєкту доцільно розглянути кілька сценаріїв, які демонструють взаємодію реалізованих модулів. Ці сценарії показують, яку саме проблему вирішує система на рівні користувацької поведінки та backend-логіки.

Сценарій 1: користувач підключає Monobank, запускає синхронізацію і переходить на dashboard. Система отримує рахунки, формує account_ref/account_label, зберігає транзакції, визначає тип expense/income, виконує первинну категоризацію та повертає агреговані показники. На dashboard користувач бачить загальні витрати, доходи, категорії та останні операції. Якщо синхронізація запускається повторно, дублікати не повинні вплинути на суму витрат.

Сценарій 2: користувач обирає окрему картку, наприклад «Чорна картка». У цьому режимі частина рухів, які в загальному режимі можуть бути внутрішніми переказами, мають значення для аналітики конкретної картки. Тому логіка dashboardAnalytics повинна враховувати контекст selectedAccount. Це приклад ситуації, коли одна й та сама транзакція має різну аналітичну інтерпретацію залежно від рівня перегляду.

Сценарій 3: користувач з активним ФОП-режимом переглядає витрати. Backend класифікує частину операцій як business або tax на основі рахунку, опису та ключових слів. У dashboard такі транзакції можуть бути відокремлені від персональних витрат. Практична цінність цього сценарію полягає в тому, що користувач не змішує побутові витрати з підприємницькими операціями.

Сценарій 4: користувач запитує AI-помічника: «На що я витратив найбільше цього місяця?». Система не передає LLM повну базу даних. Backend формує короткий фінансовий контекст: total_expenses, total_income, balance,

top_categories, recent_transactions, goals, debts і fop_summary. Після цього Ollama формує пояснення. Якщо категорія «Транспорт» займає найбільшу частку, AI може пояснити це природною мовою, але сам відсоток і сума мають походити з backend-розрахунку.

Сценарій 5: користувач підписаний на push-сповіщення. Якщо система надсилає тестове повідомлення, воно відображається незалежно від стану вікна. Якщо це звичайне повідомлення про подію, service worker перевіряє, чи є активне вікно застосунку. Якщо сторінка вже відкрита, повідомлення не дублюється. Це підвищує зручність використання і зменшує кількість зайвих системних повідомлень.

Практичні сценарії роботи системи узагальнено у таблиці 4.6.

Таблиця 4.6

Практичні сценарії використання системи

Сценарій	Задіяні модулі	Очікуваний практичний результат
Початкова синхронізація Monobank	monobank endpoint, MonobankService, Celery, Transaction	Користувач отримує історію операцій без ручного введення
Повторна синхронізація	sync guard, account-scoped dedupe	Суми не спотворюються через дублікати
Аналіз окремої картки	dashboardAnalytics, account filters	Показники відповідають контексту вибраного рахунку
ФОП-розмежування	Transaction.scope, fop_service, filters	Особисті й бізнес-операції не змішуються в аналітиці
AI-запит	FinancialContextService, Ollama, chat UI	Користувач отримує пояснення на основі підготовлених даних
Push-подія	push endpoint, push_subscriptions, service worker	Повідомлення показується доречно, без дублювання активного UI

Зазначені сценарії також демонструють межі відповідальності модулів. Frontend відповідає за стан інтерфейсу і взаємодію з користувачем, backend -

за авторизацію, валідацію, фінансові правила і збереження даних, Celery - за тривалі операції, Ollama - за текстове пояснення, а service worker - за поведінку push-повідомлень. Такий розподіл спрощує подальше тестування та розвиток системи.

Окремої уваги потребує питання підтримки достовірності фінансових підсумків у випадку ручного редагування користувачем. Якщо користувач змінює категорію операції після синхронізації, наступний імпорт із банку не повинен безумовно перезаписувати цю зміну. У проєкті це враховується на рівні обережної логіки оновлення: повторно отримана транзакція використовується для підтвердження її наявності, але не має знищувати користувацьке уточнення без окремої підстави.

Також важливим є питання часових зон. Для українського користувача дата транзакції, час новин і момент push-повідомлення мають інтерпретуватися у локальному контексті, зокрема Europe/Kyiv. Інакше операції, виконані пізно ввечері або біля межі доби, можуть потрапляти до іншого дня або місяця, що спотворює місячну аналітику. Тому у подальшому розвитку системи доцільно уніфікувати всі перетворення часу на рівні backend utility-функцій.

З погляду подальшої підтримки продукту, найбільш ризиковими зонами залишаються банківські інтеграції та фінансові правила. Інтерфейс можна змінювати поступово, не зачіпаючи збережені дані, але зміна правил обробки транзакцій впливає на історичну аналітику. Тому такі правила повинні бути покриті тестами на контрольних наборах даних: внутрішній переказ, переказ на банку, ФОП-дохід, податковий платіж, повернення коштів, кешбек і підписка.

Додатково під час проєктування було враховано питання супроводжуваності коду. Для Finance Monitor важливо, щоб новий модуль не змінював усі частини системи одночасно. Наприклад, додавання нового банківського джерела має переважно стосуватися service-рівня інтеграції, схеми нормалізації транзакцій і UI-налаштувань рахунку, але не повинно

змінювати базову логіку бюджетів, цілей, AI-контексту чи push-підписок. Саме тому структура backend поділена на endpoints, services, schemas, db/models і tasks.

Ще одним критерієм якості є можливість діагностики помилок. У фінансовому застосунку недостатньо просто повідомити користувача, що «щось пішло не так». Потрібно розуміти джерело проблеми: помилка авторизації, недоступність банківського API, перевищення ліміту запитів, помилка формату відповіді, дубльована транзакція або внутрішня помилка бази даних. Наявність таблиці background_jobs та логів синхронізації дозволяє відокремити помилки зовнішнього сервісу від помилок власної бізнес-логіки.

Реалізована система також враховує майбутню зміну формату даних. Банківські API можуть додавати нові поля або змінювати набір метаданих транзакцій. Якщо модель буде занадто жорсткою, будь-яка зміна потребуватиме міграції бази даних. Тому основні поля транзакцій винесено в окремі колонки, а менш стабільні атрибути можуть зберігатися як додаткові метадані. Це компроміс між нормалізацією та практичною гнучкістю інтеграції.

Для користувацького інтерфейсу важливо, що система не приховує складність фінансових даних повністю. Користувач має бачити історію транзакцій, розуміти обраний рахунок, мати можливість змінити категорію та порівняти результати між dashboard і analysis. Тому UI не повинен бути лише набором красивих графіків; він має пояснювати, з яких даних сформовано підсумки. Це особливо важливо для довіри до фінансового застосунку.

Висновки до розділу 4

У четвертому розділі розглянуто методику тестування Finance Monitor та проаналізовано результати перевірки основних функціональних сценаріїв. Тестування охопило авторизацію, роботу з транзакціями, побудову дашбордів, синхронізацію з Monobank, роботу AI-модуля, push-сповіщення, негативні сценарії API та поведінку системи під час обробки типових помилок.

Результати тестування показали працездатність реалізованої системи у межах заявлених сценаріїв і водночас дозволили визначити напрями, які потребують подальшого посилення. До них належать стабілізація синхронізаційних блокувань у багатопроцесному середовищі, оптимізація роботи з великими вибірками транзакцій, розширення банківських конекторів, деталізація правил категоризації та покращення спостережуваності системи через логи й технічні метрики.

ВИСНОВКИ

У кваліфікаційній роботі досягнуто поставленої мети: спроєктовано, реалізовано та перевірено вебзастосунок Finance Monitor для моніторингу фінансових рахунків користувача, синхронізації транзакцій, аналітичного подання фінансових даних, роботи з бюджетами, push-сповіщеннями та локальним AI-модулем. Розроблений застосунок орієнтований на практичний сценарій українського користувача, який працює з кількома рахунками, потребує прозорої історії операцій і хоче отримувати аналітику без повної залежності від окремого банківського застосунку.

У ході дослідження проаналізовано підходи до побудови PFM-систем і розглянуто наявні рішення для управління особистими фінансами. Встановлено, що банківські застосунки добре працюють у межах власної інфраструктури, а міжнародні PFM-сервіси мають сильні інструменти бюджетування та аналітики, проте для локального українського контексту залишаються обмеження, пов’язані з інтеграцією банківських даних, приватністю, ФОП-сценаріями, мовною адаптацією та контролем над власною історією транзакцій.

На основі аналізу предметної області сформовано вимоги до Finance Monitor і обґрунтовано архітектуру системи. Backend реалізовано як модульний FastAPI-застосунок із сервісним шаром, моделями даних, REST API та фоновими задачами Celery/Redis. Для збереження даних використано PostgreSQL/SQLAlchemy, а frontend побудовано на Next.js 14 з адаптивними сторінками дашборду, аналізу, транзакцій, бюджетів, цілей, новин, AI-чату та інших користувацьких модулів.

Практична реалізація підтвердила важливість коректної роботи з фінансовими даними на рівні бізнес-логіки. У системі передбачено account-scoped підхід до унікалізації транзакцій, обробку повторних синхронізацій, врахування обмежень Monobank API, розмежування видимої історії операцій і аналітичних розрахунків, а також формування підготовленого фінансового

контексту для локального Ollama-модуля. Це дозволяє зменшити ризик дублювання операцій, спотворення дашбордів і неконтрольованого передавання чутливих даних зовнішнім AI-сервісам.

Проведене тестування охопило основні API-запити, користувацькі сценарії, негативні випадки, інтеграційні процеси та поведінку системи за умов зовнішніх обмежень. За результатами перевірки встановлено, що реалізовані модулі виконують свої основні функції, а виявлені проблеми мають технічно зрозумілий характер і можуть бути усунуті в межах подальшого розвитку системи. Найбільш важливими напрямами посилення є стабільні синхронізаційні блокування, оптимізація великих аналітичних вибірок, розширення банківських інтеграцій і покращення моніторингу фонових задач.

Практичне значення роботи полягає у створенні функціонального програмного продукту, який може бути використаний як основа для персонального фінансового кабінету. Finance Monitor не замінює банківський застосунок, а доповнює його окремим шаром агрегації, збереження, аналізу й пояснення фінансових даних. Подальший розвиток системи доцільно спрямувати на підключення нових джерел даних, удосконалення правил категоризації, розширення ФОП-модуля, покращення AI-пояснень і підготовку архітектури до більш стабільної експлуатації в умовах реального навантаження.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. ДСТУ 3008:2015. Інформація та документація. Звіти у сфері науки і техніки. Структура та правила оформлювання. Київ : ДП «УкрНДНЦ», 2016. 26 с.

2. ДСТУ 8302:2015. Інформація та документація. Бібліографічне посилання. Загальні положення та правила складання. Київ : ДП «УкрНДНЦ», 2016. 16 с.

3. ДСТУ ISO 5807:2016. Обробляння інформації. Символи та угоди щодо документації стосовно даних, програм та системних блок-схем, схем мережевих програм та схем системних ресурсів. Київ : ДП «УкрНДНЦ», 2016.

4. Машкіна І. В., Абрамов В. О., Носенко Т. І., Іваніченко Є. В., Яскевич В. О. Навчально-методичний посібник до виконання і захисту кваліфікаційної роботи бакалавра спеціальності 122 Комп’ютерні науки. Київ : Київський столичний університет імені Бориса Грінченка, 2024. 43 с.

5. Monobank API. Офіційна документація. URL: https://api.monobank.ua/docs/ (дата звернення: 26.05.2026).

6. Revolut Help Centre. Pockets and budgeting analytics. URL: https://help.revolut.com/ (дата звернення: 26.05.2026).

7. YNAB. You Need A Budget : офіційний вебсайт. URL: https://www.ynab.com/ (дата звернення: 26.05.2026).

8. Rocket Money. Subscription management and spending insights : офіційний вебсайт. URL: https://www.rocketmoney.com/ (дата звернення: 26.05.2026).

9. FastAPI Documentation. URL: https://fastapi.tiangolo.com/ (дата звернення: 26.05.2026).

10. Next.js Documentation. URL: https://nextjs.org/docs (дата звернення: 26.05.2026).

11. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/ (дата звернення: 26.05.2026).

12. SQLAlchemy Documentation. URL: https://docs.sqlalchemy.org/ (дата звернення: 26.05.2026).

13. Alembic Documentation. URL: https://alembic.sqlalchemy.org/ (дата звернення: 26.05.2026).

14. Redis Documentation. URL: https://redis.io/docs/ (дата звернення: 26.05.2026).

15. Celery Documentation. URL: https://docs.celeryq.dev/ (дата звернення: 26.05.2026).

16. Pydantic Documentation. URL: https://docs.pydantic.dev/ (дата звернення: 26.05.2026).

17. Python Documentation. URL: https://docs.python.org/3/ (дата звернення: 26.05.2026).

18. HTTPX Documentation. URL: https://www.python-httpx.org/ (дата звернення: 26.05.2026).

19. React Documentation. URL: https://react.dev/ (дата звернення: 26.05.2026).

20. Chart.js Documentation. URL: https://www.chartjs.org/docs/latest/ (дата звернення: 26.05.2026).

21. MDN Web Docs. Service Worker API. URL: https://developer.mozilla.org/docs/Web/API/Service_Worker_API (дата звернення: 26.05.2026).

22. MDN Web Docs. Push API. URL: https://developer.mozilla.org/docs/Web/API/Push_API (дата звернення: 26.05.2026).

23. Thomson M., Damaggio E., Raymor B. Generic Event Delivery Using HTTP Push : RFC 8030. IETF, 2016. URL: https://www.rfc-editor.org/rfc/rfc8030 (дата звернення: 26.05.2026).

24. Thomson M., Volz B., Wang V. Voluntary Application Server Identification (VAPID) for Web Push : RFC 8292. IETF, 2017. URL: https://www.rfc-editor.org/rfc/rfc8292 (дата звернення: 26.05.2026).

25. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT) : RFC 7519. IETF, 2015. URL: https://www.rfc-editor.org/rfc/rfc7519 (дата звернення: 26.05.2026).

26. OWASP API Security Top 10. URL: https://owasp.org/www-project-api-security/ (дата звернення: 26.05.2026).

27. OWASP Password Storage Cheat Sheet. URL: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html (дата звернення: 26.05.2026).

28. Ollama API Documentation. URL: https://docs.ollama.com/api (дата звернення: 26.05.2026).

29. Pytest Documentation. URL: https://docs.pytest.org/ (дата звернення: 26.05.2026).

30. Locust Documentation. URL: https://docs.locust.io/ (дата звернення: 26.05.2026).

31. Finance Monitor : програмний код інтерактивного вебзастосунку моніторингу фінансових рахунків користувача / розробник В. Верхочуб. Backend: FastAPI, SQLAlchemy, PostgreSQL, Celery, Redis; Frontend: Next.js, TypeScript, React. Особистий архів автора, 2026.

ДОДАТОК А

Приклади фрагментів реалізації

Фрагменти коду подано для ілюстрації реалізованих рішень. У основній частині роботи залишено пояснення логіки, а детальніші фрагменти винесено до додатків, щоб не перевантажувати текст.

Приклад service worker для push-повідомлень

```
// Звичайний push не показується, якщо застосунок активний
const hasVisibleAppWindow = windowClients.some((client) => {
  return client.url.startsWith(self.location.origin) &&
    (client.visibilityState === 'visible' || client.focused === true);
});
if (hasVisibleAppWindow && !testPush) return;
```

Приклад формування AI-контексту

```
transactions = db.query(Transaction)
  .filter(Transaction.user_id == user.id, Transaction.date >= month_start)
  .order_by(Transaction.date.desc())
  .limit(60)
  .all()
# Далі контекст скорочується до підсумків, категорій і останніх операцій.
```

Приклад обробки HTTP 429 від Monobank

```
if res.status_code == 429:
    retry_after = res.headers.get('Retry-After')
    wait_seconds = max(1, min(int(retry_after), 60)) if retry_after else
60
    if attempt < retries:
        time.sleep(wait_seconds)
        continue
    raise MonobankRateLimitError('Monobank тимчасово обмежив запити')
```

ДОДАТОК Б

Таблиця Б.1

Негативне тестування системи Finance Monitor

№	Сценарій	Вхідні дані	Очікуваний результат	Фактичний результат
1	Запит без авторизації	GET /transactions без JWT	401 Unauthorized	Доступ блокується
2	Невалідний токен Monobank	POST /monobank/sync з неправильним токеном	Контрольована помилка	Користувач отримує повідомлення
3	Повторний sync	Два запити поспіль	Другий не створює нову активну задачу	Застосовано active sync guard
4	Некоректний період	days > 31	422 Validation Error	Параметр обмежено MAX_SYNC_DAYS
5	Push при активному вікні	Звичайне push-повідомлення	Не дублювати видимий UI	Service worker не показує ordinary push

ДОДАТОК В

Діаграми системи

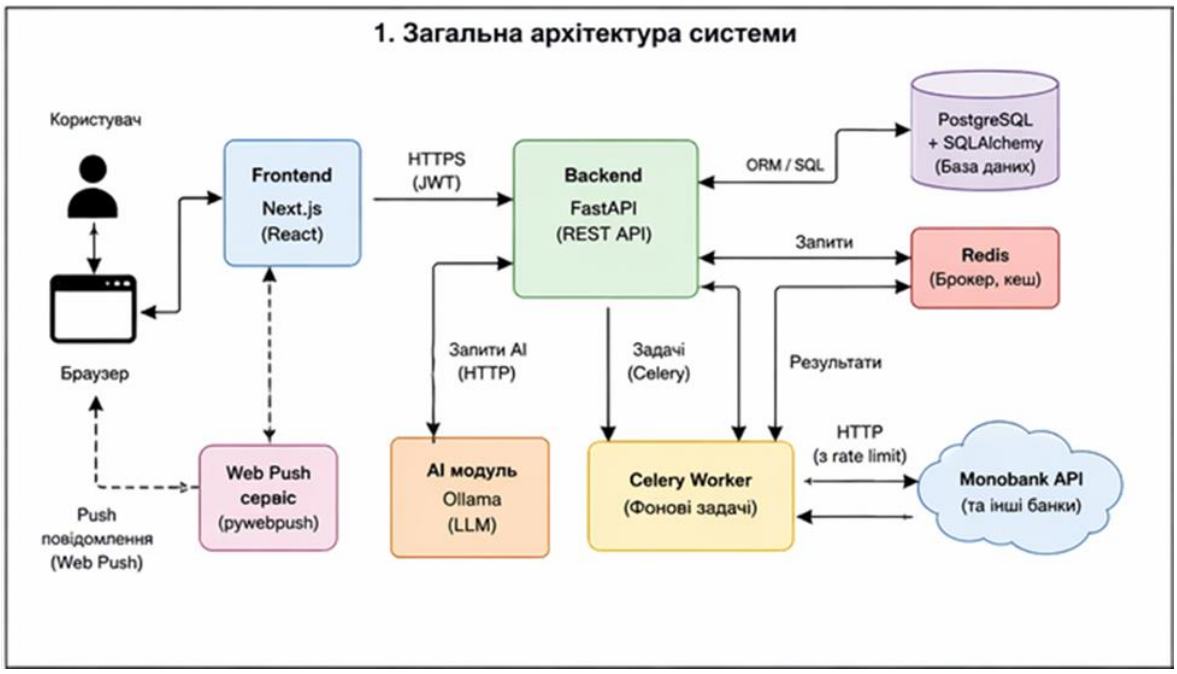

Рисунок В.1. Загальна архітектура системи Finance Monitor

Джерело: складено автором.

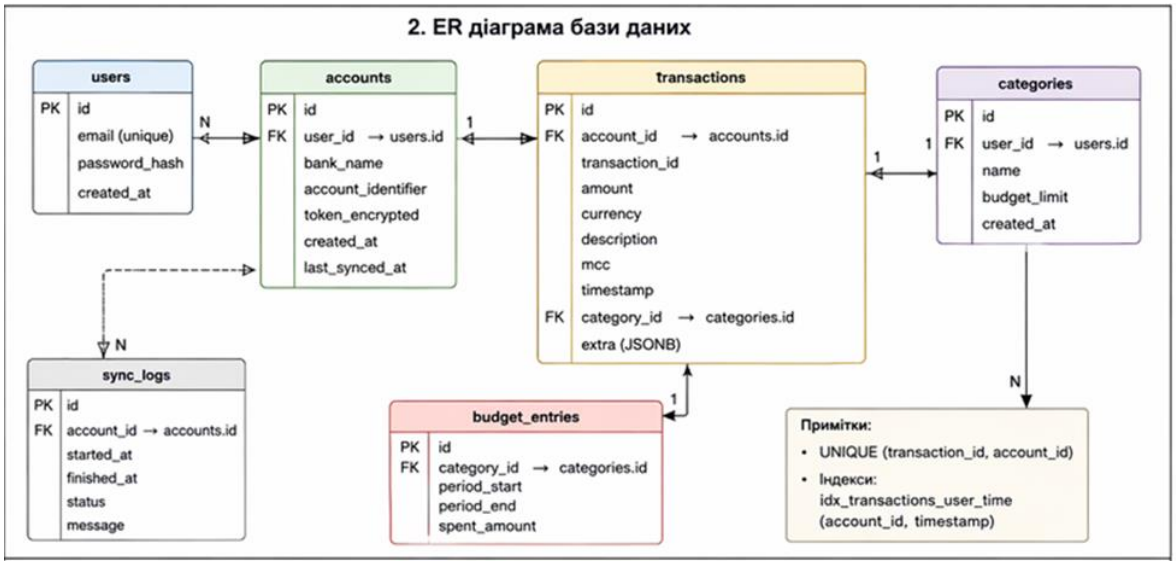

Рисунок В.2. ER-діаграма бази даних

Джерело: складено автором.

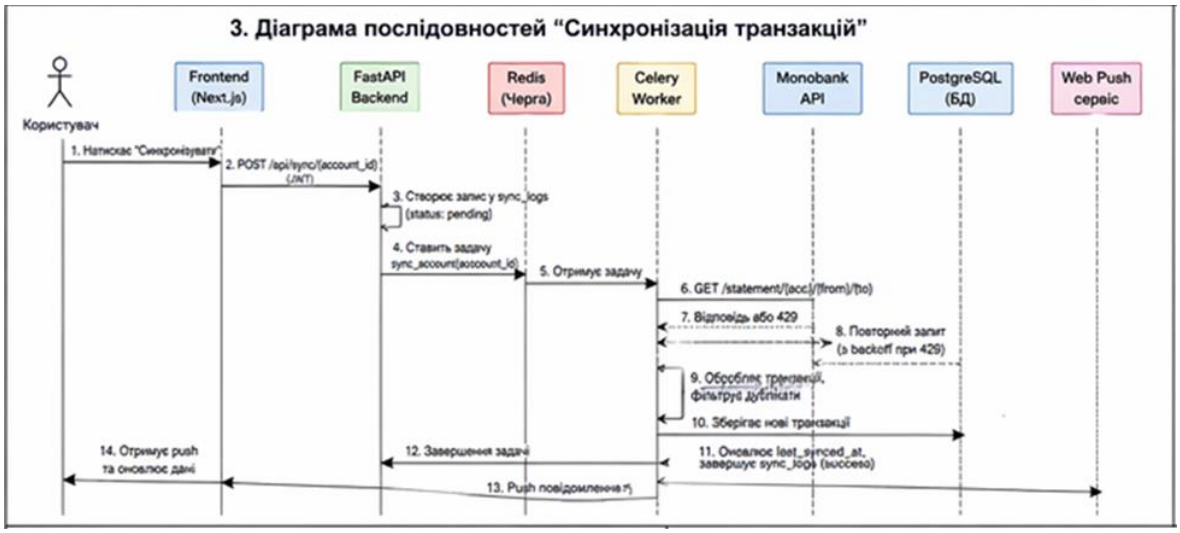

Рисунок В.3. Діаграма послідовностей «Синхронізація транзакцій»

Джерело: складено автором.

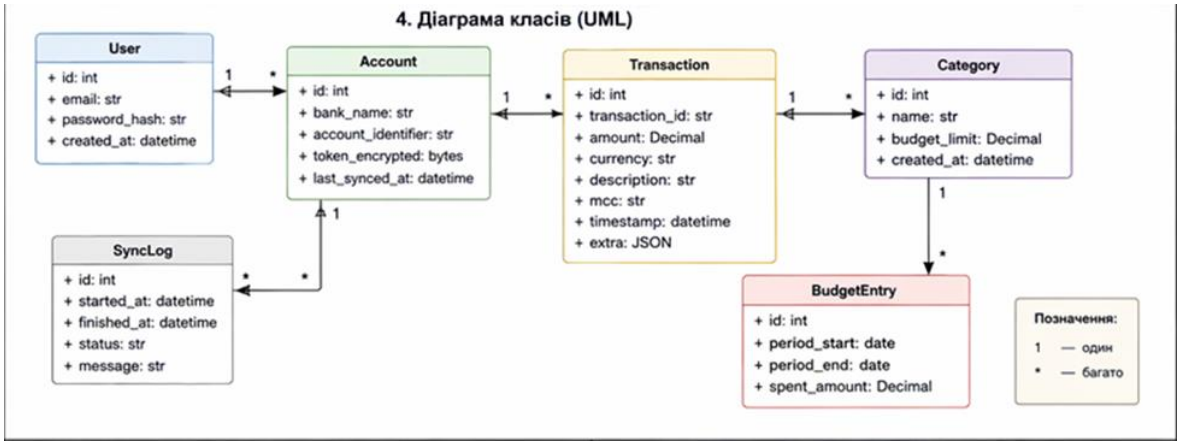

Рисунок В.4. UML-діаграма класів

Джерело: складено автором.

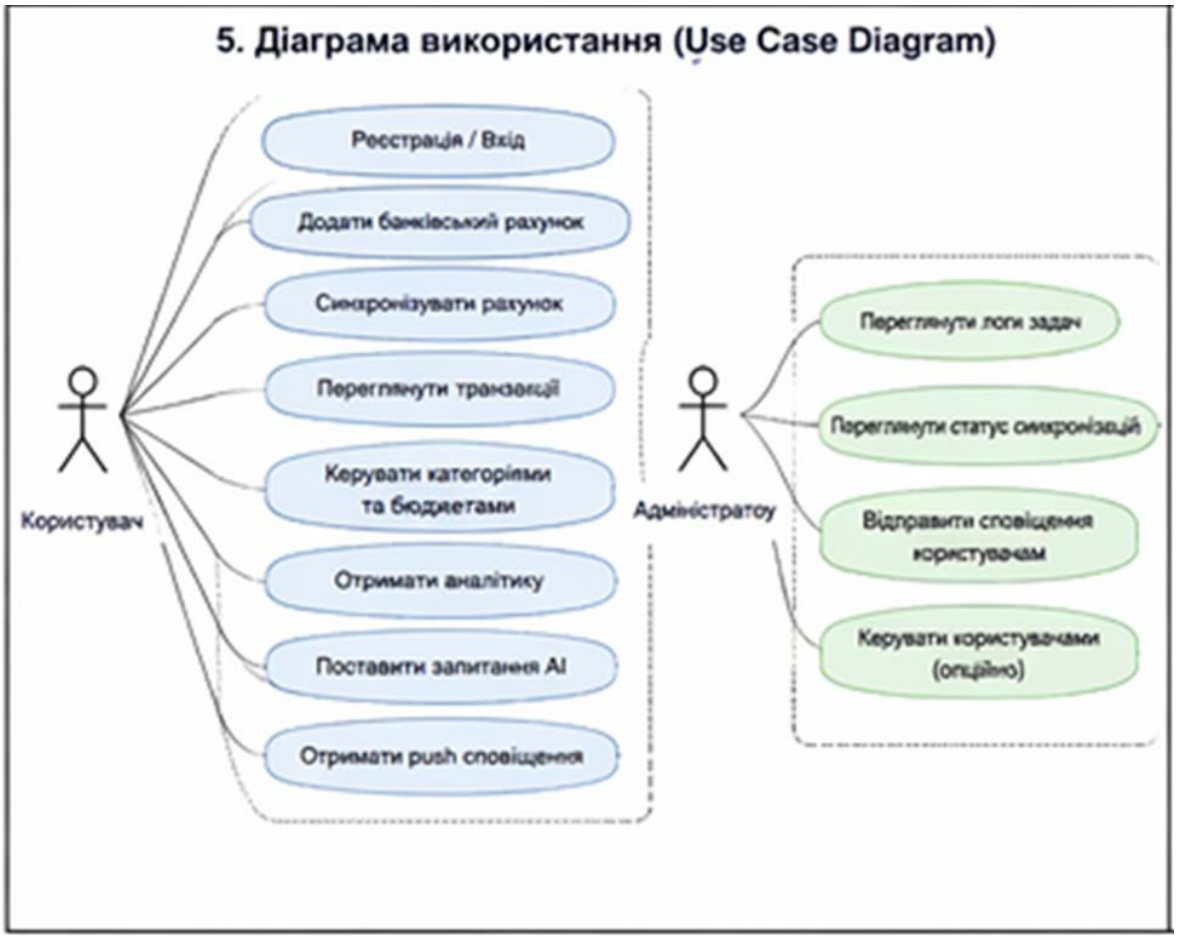

Рисунок В.5. Діаграма використання системи

Джерело: складено автором.

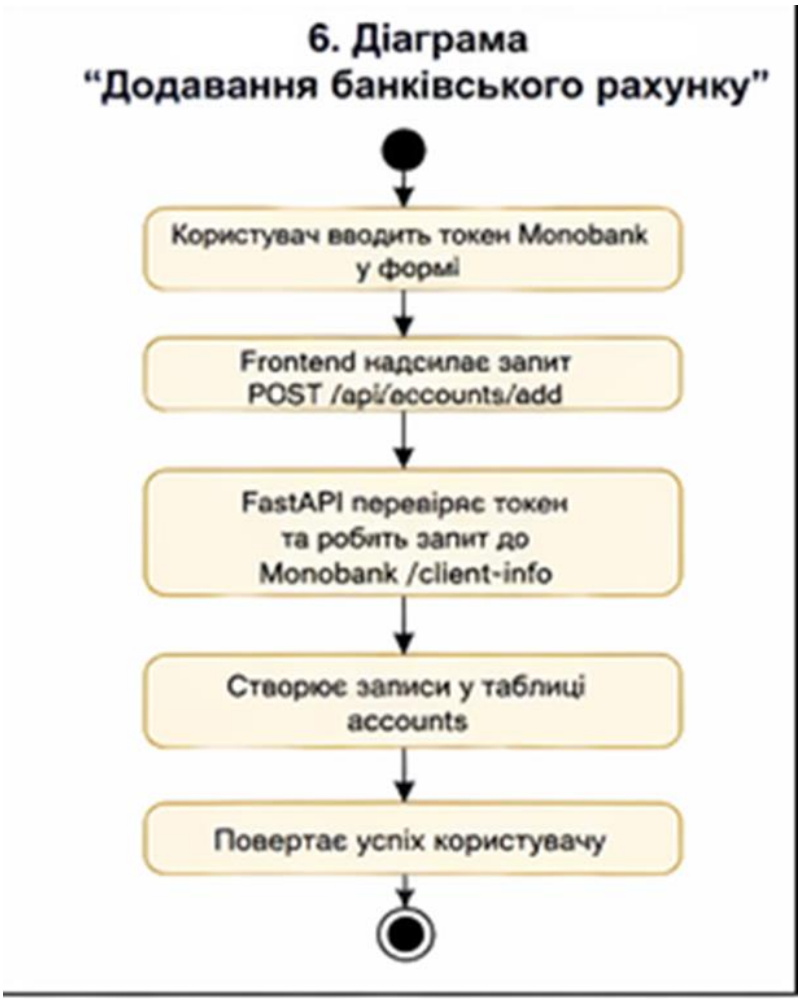

Рисунок В.6. Діаграма додавання банківського рахунку

Джерело: складено автором.

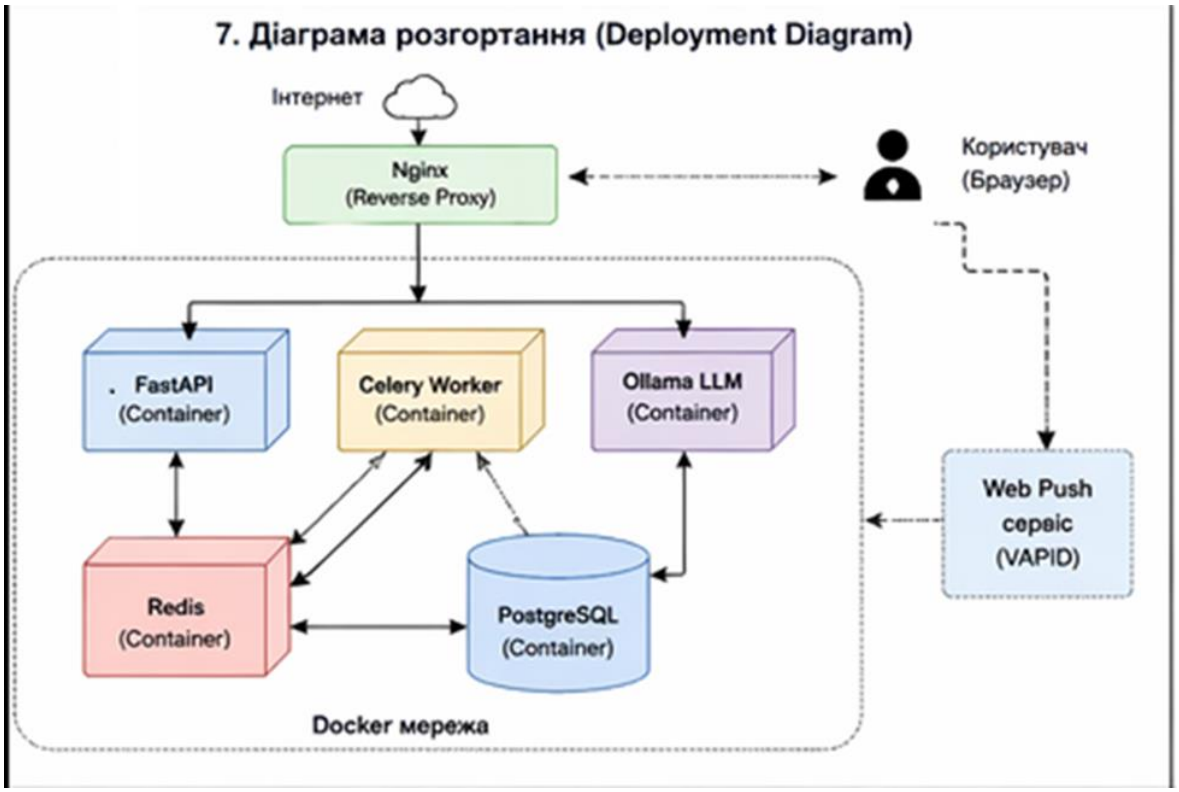

Рисунок В.7. Діаграма розгортання системи

Джерело: складено автором.

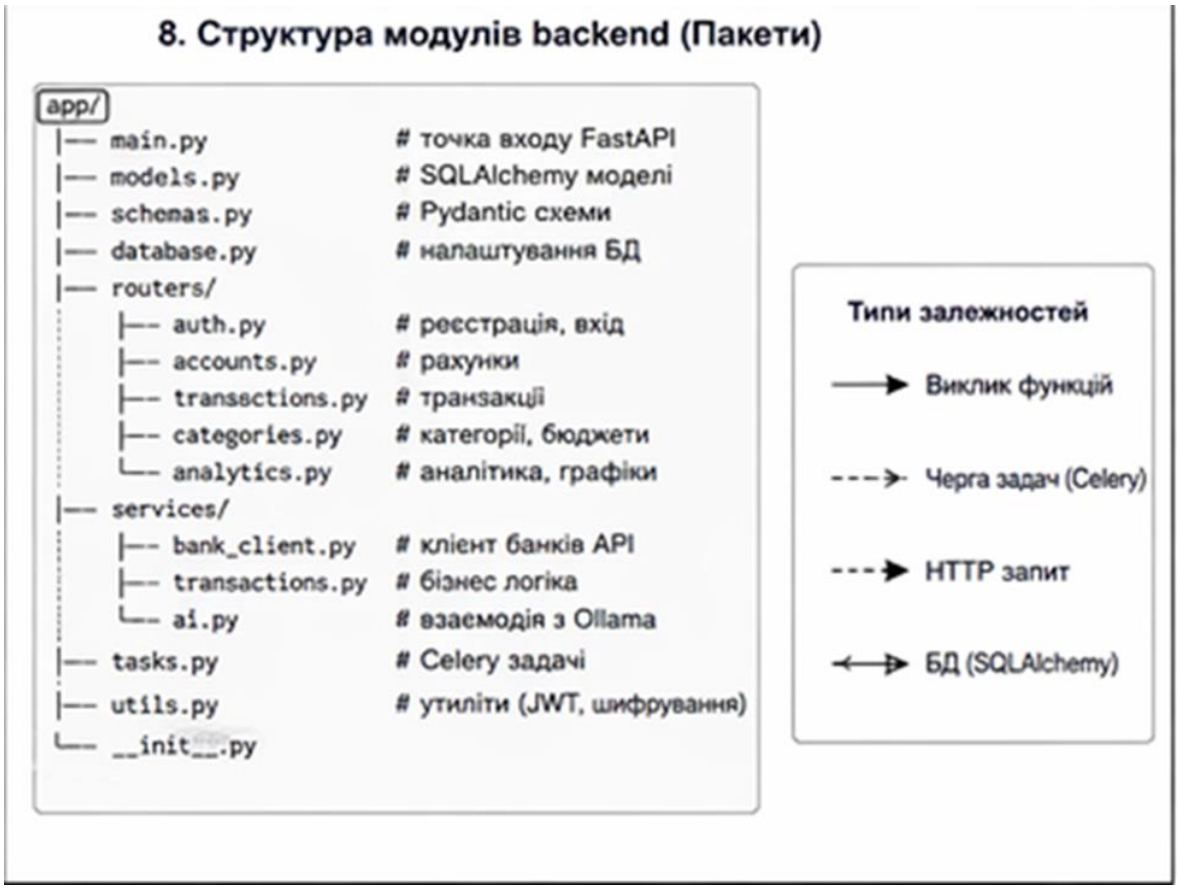

Рисунок В.8. Структура backend-модулів

Джерело: складено автором.