МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Острозька академія»
Навчально-науковий інститут інформаційних технологій та бізнесу
Кафедра інформаційних технологій та аналітики даних

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня бакалавра

на тему: **«Розробка застосунку Text translation tool»**

Виконав: студент 4 курсу, групи КН-41
першого (бакалаврського) рівня вищої освіти
спеціальності 122 Комп'ютерні науки
освітньо-професійної
програми«Комп'ютерні науки»
Кошин Максим Володимирович

Керівник: *викладач кафедри ІТАД,*
Мацевич Денис Володимирович

Рецензент: *кандидат технічних наук, доцент,*
доцент кафедри прикладної математики
Донецького національного університету
імені Василя Стуса
Загоруйко Любов Василівна

РОБОТА ДОПУЩЕНА ДО ЗАХИСТУ

Завідувач кафедри інформаційних технологій та аналітики даних _______________
(проф., д.е.н. Кривицька О.Р.)

Протокол № 11 від « 20 » травня 2026 р.

Острог, 2026

АНОТАЦІЯ
кваліфікаційної роботи
на здобуття освітнього ступеня бакалавра

Тема: Розробка застосунку Text translation tool
Автор: *Кошин Максим Володимирович*

Науковий керівник: викладач кафедри економіко-математичного моделювання та інформаційних технологій Мацевич Денис Володимирович

Захищена «………»………………… 20__ року.

Пояснювальна записка до кваліфікаційної роботи: ____*(кількість сторінок роботи) с., ____ (кількість рисунків) рис., _____ (кількість таблиць) табл., _____ (кількість додатків) додатків, _____ (кількість джерел) джерел.*

Ключові слова: *БРАУЗЕРНЕ РОЗШИРЕННЯ, ШТУЧНИЙ ІНТЕЛЕКТ, ВЕБ-ЗАСТОСУНОК, REACT.JS, NODE.JS, GRAPHQL, MONGODB, CHROME EXTENSIONS API, ПЕРЕКЛАД ТЕКСТУ, СУМАРИЗАЦІЯ, АВТЕНТИФІКАЦІЯ, JWT, КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА..*

Короткий зміст праці:

У кваліфікаційній роботі розглянуто процес проєктування та розробки клієнт-серверного браузерного розширення з інтеграцією штучного інтелекту для обробки природної мови. Актуальність теми зумовлена стрімким розвитком генеративного штучного інтелекту та зростаючою потребою користувачів у швидкому, контекстному доступі до інструментів перекладу та сумаризації тексту безпосередньо під час роботи у веббраузері. Метою роботи було створення комплексного застосунку формату Chrome Extension, який забезпечує безпечний і безперервний цикл взаємодії користувача з персональним AI-асистентом.

У теоретичній частині обґрунтовано вибір сучасного технологічного стеку та архітектурних підходів для побудови вебзастосунків. В якості фундаментальної платформи для серверної частини було обрано середовище Node.js, а для організації типізованої та ефективної комунікації між клієнтом і сервером — технологію GraphQL на базі Apollo Server. Для гнучкого зберігання історій запитів та облікових записів використано документо-орієнтовану нереляційну базу даних MongoDB. Архітектура клієнтської частини базується на специфікації Manifest V3 для браузерних розширень, що гарантує високу продуктивність та відповідність сучасним стандартам безпеки.

Практична частина роботи присвячена безпосередній реалізації ключового функціоналу продукту. В рамках розробки серверної логіки створено надійну систему автентифікації та авторизації на базі JSON Web Tokens (JWT) для захисту доступу до ресурсів. Також алгоритмізовано процеси верифікації електронної пошти та відновлення паролів за допомогою інтеграції транзакційного поштового сервісу (Resend API) через підтверджений власний домен.

Важливим етапом стала програмна імплементація клієнтського інтерфейсу на базі бібліотеки React.js. Було реалізовано адаптивний преміальний UI-дизайн із підтримкою тем, впроваджено систему багатомовності (i18n), інтегровано Web Speech API для голосового відтворення результатів, а також розроблено алгоритми збереження локального стану вкладок (Persistent Workspace) для запобігання втраті даних при перемиканні контексту.

Розроблена архітектура системи спроєктована з урахуванням високих вимог до ізоляції користувацьких сесій, швидкодії та інформаційної безпеки, що гарантує надійну основу для її подальшого масштабування та комерційної експлуатації.

The qualification work considers the process of designing and developing a client-server browser extension with the integration of artificial intelligence for natural language processing.

The relevance of the topic is due to the rapid development of generative artificial intelligence and the growing need of users for quick, contextual access to translation and text summarization tools directly while working in a web browser.

The goal of the work was to create a comprehensive application in the Chrome Extension format, which provides a safe and continuous cycle of user interaction with a personal AI assistant.

The theoretical part justifies the choice of a modern technological stack and architectural approaches for building web applications. The Node.js environment was chosen as the fundamental platform for the server part, and the GraphQL technology based on Apollo Server was chosen to organize typed and effective communication between the client and the server. The document-oriented non-relational database MongoDB was used for flexible storage of query histories and accounts.

The client-side architecture is based on the Manifest V3 specification for browser extensions, which guarantees high performance and compliance with modern security standards.

The practical part of the work is devoted to the direct implementation of the key functionality of the product. As part of the development of the server logic, a reliable authentication and authorization system based on JSON Web Tokens (JWT) was created to protect access to resources. The processes of email verification and password recovery were also algorithmized by integrating the transactional mail service (Resend API) through a confirmed own domain.

An important stage was the software implementation of the client interface based on the React.js library. An adaptive premium UI design with theme support was implemented, a multilingual system (i18n) was introduced, the Web Speech API was integrated for voice playback of results, and algorithms for saving the local state of tabs (Persistent Workspace) were developed to prevent data loss when switching context.

The developed system architecture is designed taking into account high requirements for user session isolation, speed, and information security, which guarantees a reliable foundation for its further scaling and commercial operation.

ЗМІСТ

ЗМІСТ **4**
ВСТУП **9**
РОЗДІЛ 1 **13**
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ЗАСОБІВ РОЗРОБКИ **13**
1.1 Опис предметного середовища 13
1.1.1 Аналіз ринку браузерних розширень та інтеграції штучного інтелекту 13
1.1.2 Функціональна модель системи та основні актори 14
1.1.3 Моделювання основного бізнес-процесу "Обробка тексту AI-асистентом" 15
1.2 Огляд та аналіз аналогічних програмних рішень 15
1.2.1 Аналіз функціоналу конкурентів (DeepL, Grammarly, Monica.im, Sider) 16
1.2.2 Порівняльний аналіз технологічних стеків та архітектурних підходів аналогів 16
1.2.3 Виявлення недоліків існуючих рішень та обґрунтування доцільності розробки власного продукту 17
1.3 Постановка задачі на розробку 17
1.3.1 Формулювання основних функціональних вимог 18
1.3.2 Формулювання специфічних функціональних вимог 18
1.3.3 Формулювання нефункціональних вимог 19
РОЗДІЛ 2 **20**
ПРОЄКТУВАННЯ СЕРВЕРНОЇ АРХІТЕКТУРИ ТА ІНФОРМАЦІЙНОГО ЗАБЕЗПЕЧЕННЯ (BACKEND) **20**
2.1 Проєктування інформаційного забезпечення та бази даних 20
2.1.1 Обґрунтування вибору документо-орієнтованої СУБД 20
2.1.2 Логічна модель даних на базі Mongoose ODM 20
2.1.3 Управління життєвим циклом даних (TTL-індекси) 21
2.2 Архітектура серверного застосунку та API-рівень 23
2.2.1 Парадигма GraphQL проти традиційного REST API 24
2.2.2 Шарувата архітектура (Layered Architecture) 24
2.2.3 Гібридний API-підхід (REST для TTS) 26
2.3 Алгоритми бізнес-логіки, автентифікації та безпеки 26
2.3.1 Механізм JWT-сесій та Google OAuth 26
2.3.2 Алгоритм транзакційних розсилок (Auth-flow) 27

2.4 Інтеграція зовнішніх AI-сервісів та забезпечення відмовостійкості 27
2.4.1 Алгоритм Retry/Backoff (Експоненційна затримка) 28
2.5 Контейнеризація та розгортання серверної інфраструктури (DevOps) 30
2.5.1 Розгортання (CI/CD) на платформі Render та аналіз Cold Start 30
РОЗДІЛ 3 32
ПРОГРАМНА РЕАЛІЗАЦІЯ КЛІЄНТСЬКОЇ АРХІТЕКТУРИ ТА КОРИСТУВАЦЬКОГО ІНТЕРФЕЙСУ (FRONTEND) 32
3.1 Проєктування та імплементація базової клієнтської архітектури 32
3.1.1 Вибір фреймворку та управління станом (React 18) 32
3.1.2 Інструментарій збірки Vite та конфігурація Manifest V3 32
3.1.3 Управління мережевими запитами через Apollo Client 34
3.2 Розробка адаптивного користувацького інтерфейсу (UI/UX) та забезпечення доступності 34
3.2.1 Парадигма Utility-first за допомогою Tailwind CSS 34
3.2.2 Забезпечення веб-доступності (WAI-ARIA) з Radix UI 35
3.2.3 Алгоритми підтримки тем та адаптивної навігації 35
3.3 Реалізація бізнес-логіки клієнтського застосунку та управління станом (Persistent Workspace) 36
3.3.1 Імплементація кастомного хука для синхронізації з localStorage 36
3.3.2 Оптимізація рендерингу та запобігання дублюванню запитів 38
3.4 Інтеграція спеціалізованих клієнтських модулів (i18n, TTS, OAuth) 38
3.4.1 Розробка кастомної системи багатомовності (i18n) 38
3.4.2 Алгоритм відтворення потокового аудіо (Text-to-Speech) 40
3.4.3 Управління сесіями на клієнті та Google OAuth 40
3.5 Забезпечення управління контейнеризацією та розгортанням клієнтського застосунку 40
3.5.1 Контейнеризація середовища розробки за допомогою Docker 41
3.5.2 Оптимізація фінальної збірки (Production Build) 42
РОЗДІЛ 4 43
ТЕСТУВАННЯ ТА ЗАБЕЗПЕЧЕННЯ ЯКОСТІ ПРОГРАМНОГО ПРОДУКТУ 43
4.1 Стратегія тестування та забезпечення надійності системи 43
4.2 Тестування серверної частини (Backend) 43
4.2.1 Інструментарій тестування: Jest, ts-jest та Supertest 43
4.2.2 Модульне тестування бізнес-логіки та мокування (Mocking) 44
4.3 Тестування клієнтської частини (Frontend) 46
4.3.1 Середовище тестування: Vitest та happy-dom 46
4.3.2 Тестування інтерфейсів за допомогою React Testing Library 46

4.4 Тестування безпеки, ізоляції даних та доступності 48
4.5 Аналіз продуктивності та кількісні метрики оптимізації 49
4.5.1 Оптимізація клієнтського бандлу (Bundle Size) 49
4.5.2 Вимірювання мережевої затримки та швидкодії API 49
4.5.3 Споживання оперативної пам'яті (Memory Consumption) 50
ЗАГАЛЬНІ ВИСНОВКИ 51
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 53
ДОДАТКИ 56
ДОДАТКИ 61

ВСТУП

Інтенсивний розвиток технологій машинного навчання та глобальне проникнення великих мовних моделей (LLM) у всі сфери суспільного та професійного життя кардинально змінили фундаментальну парадигму взаємодії людини з цифровим інформаційним простором. Традиційні методи обробки текстових масивів, класичні статичні перекладачі та інструменти ручного реферування інформації поступово поступаються місцем складним інтелектуальним програмним комплексам. Сучасні системи здатні автономно та в режимі реального часу аналізувати контекст, виконувати глибоку семантичну обробку природної мови та генерувати релевантні відповіді, незалежно від обсягу чи складності вхідних даних.

Індустрія розробки прикладного програмного забезпечення для браузерів висуває особливо жорсткі та специфічні вимоги до організації процесів обробки інформації. Це зумовлено насамперед тим, що веббраузер став основним і найбільш навантаженим робочим середовищем сучасного користувача. Необхідність постійного перемикання між вкладками вебсайтів та зовнішніми платформами штучного інтелекту порушує безперервність когнітивного процесу (workflow), знижує загальну продуктивність та створює надлишкове когнітивне навантаження. Такі умови вимагають від програмних рішень не лише простого доступу до API нейромереж, але й забезпечення безшовної інтеграції інтелектуальних інструментів безпосередньо в DOM-дерево сторінки користувача.

Відповідно, браузерні застосунки нового покоління повинні володіти екстраординарним ступенем технічної адаптивності та бездоганною швидкодією, щоб задовольнити вимоги сучасного споживача, який очікує високого рівня персоналізації, збереження контексту своїх запитів та стабільної роботи навіть при обмежених ресурсах клієнтського пристрою.

Фундаментальним ядром, що забезпечує життєдіяльність, відмовостійкість та загальну безпеку будь-якої сучасної інтелектуальної екосистеми, є її серверна інфраструктура, відома в інженерній практиці як backend-частина застосунку. Цей архітектурний рівень виступає в ролі невидимого для кінцевого користувача, але критично важливого центрального координаційного вузла. Саме в межах бекенду сконцентрована реалізація багаторівневої бізнес-логіки, управління лімітами використання дороговартісних API штучного інтелекту, алгоритми гарантування транзакційної цілісності бази даних, багатокритеріальна валідація вхідних потоків, а також безпрецедентний криптографічний захист конфіденційної персональної інформації та історії запитів клієнтів.

Глобальний історичний досвід інженерії програмного забезпечення наочно демонструє, що використання застарілих підходів при конструюванні клієнт-серверної взаємодії, зокрема надлишкових та жорстко типізованих REST API архітектур, часто призводить до проблем "Overfetching" (отримання зайвих даних) або "Underfetching" (недостатності даних). Для мінімізації подібних сценаріїв та забезпечення можливостей подальшого безперешкодного масштабування архітектури, надзвичайно важливо імплементувати передові технологічні парадигми на ранніх етапах розробки. Серед них домінуючу позицію посідає технологія мови запитів GraphQL, яка дозволяє клієнтському додатку точно визначати структуру необхідних даних, суттєво оптимізуючи мережевий трафік.

З огляду на комплекс вищевикладених факторів, головною **метою** даної кваліфікаційної роботи є системне проєктування, концептуалізація та практична розробка відмовостійкого програмного продукту — повноцінної клієнт-серверної екосистеми браузерного розширення «PAIT» (Personal AI Assistant), спеціалізованого на інтелектуальному перекладі, сумаризації та обробці тексту.

Досягнення визначеної глобальної мети передбачає декомпозицію та поетапне виконання низки взаємопов'язаних науково-інженерних і практичних **завдань**, що

охоплюють увесь багатогранний життєвий цикл створення програмного забезпечення:

1. Проаналізувати предметну область сучасних браузерних застосунків, виявити архітектурні недоліки існуючих ринкових аналогів та обґрунтувати доцільність створення власного продукту.
2. Спроєктувати надійну, слабкозв'язану архітектуру клієнтської частини згідно зі строгими політиками безпеки специфікації Chrome Manifest V3.
3. Розробити ретельно нормалізовану схему нереляційної бази даних, що мінімізує ризики виникнення аномалій модифікації та забезпечує довгострокове, структуроване зберігання історії користувацьких взаємодій.
4. Створити високопродуктивний GraphQL API, який слугуватиме універсальним стандартизованим мережевим інтерфейсом для інформаційного обміну між фронтендом та серверними потужностями.
5. Здійснити програмну імплементацію багаторівневих механізмів автентифікації та авторизації з використанням безпечних протоколів криптографічного цифрового підпису для генерації токенів доступу (JSON Web Tokens).
6. Спроєктувати та реалізувати інтерактивний, багатомовний користувацький інтерфейс на базі бібліотеки React.js із застосуванням алгоритмів збереження локального стану робочих просторів (Persistent Workspace).
7. Забезпечити глибоку інтеграцію зовнішніх алгоритмів генеративного штучного інтелекту та систем синтезу мовлення (Web Speech API).

Об'єктом дослідження у межах даної кваліфікаційної роботи виступають глибинні процеси алгоритмізації обробки природної мови, управління станами в ізольованих середовищах браузера та імплементація складної комунікаційної логіки в розподілених мережевих інформаційних системах.

Поняттєвий обсяг об'єкта дослідження охоплює методи маршрутизації користувацьких запитів, алгоритми асинхронної обробки інформації великими мовними моделями та механізми збереження інформаційної консистентності системи — від моменту ініціації сесії до успішного збереження фінальної транзакції на сервері.

Предметом дослідження є інженерні архітектурні моделі, патерни проєктування реактивних інтерфейсів користувача, концептуальні засади специфікації Manifest V3, а також сучасні інструментальні засоби розробки серверних застосунків в межах середовища виконання Node.js.

Технологічним підґрунтям для реалізації поставлених практичних цілей виступає екосистема JavaScript/TypeScript, зокрема бібліотека React.js для побудови компонентно-орієнтованого інтерфейсу та середовище Node.js для реалізації серверної бізнес-логіки. Проблема оптимізації мережевих запитів віртуозно вирішується шляхом підключення та конфігурації Apollo Server для реалізації стандарту GraphQL. Зберігання складних, ієрархічних структур даних (таких як історія чатів та персональні налаштування) забезпечується використанням документо-орієнтованої бази даних MongoDB, що ідеально узгоджується з JSON-природою вебтехнологій. Важливий акцент у дослідженні також ставиться на надійній інфраструктурі транзакційних email-розсилок через інтеграцію Resend API з верифікацією власного домену, що гарантує безпечний процес реєстрації та відновлення доступу користувачів.

РОЗДІЛ 1

АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ЗАСОБІВ РОЗРОБКИ

1.1 Опис предметного середовища

Предметним середовищем даної кваліфікаційної роботи є сфера розробки клієнт-серверних браузерних застосунків (Chrome Extensions) з інтеграцією хмарних сервісів генеративного штучного інтелекту (AI) для семантичної обробки природної мови. У сучасному інформаційному суспільстві веббраузер еволюціонував із простого засобу перегляду гіпертекстових документів у повноцінну операційну платформу, де користувачі проводять більшість свого робочого та особистого часу. Відповідно, виникає гостра потреба в інструментах, які здатні оптимізувати когнітивне навантаження при взаємодії з великими масивами багатомовного тексту. Традиційний підхід, що передбачає копіювання тексту та перехід на окремі вкладки вебперекладачів чи AI-чатів, є неефективним, оскільки руйнує контекст роботи (context switching) та знижує продуктивність користувача.

Створення сучасного браузерного розширення формату персонального асистента вимагає комплексного підходу, що охоплює управління локальними станами пам'яті браузера, гарантування безпечної авторизації через токени, обробку зовнішніх API-запитів та організацію безперервного зв'язку з бекенд-інфраструктурою. Головною метою функціонування такого середовища є забезпечення користувача інтелектуальним інструментарієм безпосередньо в активній вкладці браузера.

1.1.1 Аналіз ринку браузерних розширень та інтеграції штучного інтелекту

Сучасний ринок браузерних розширень переживає етап фундаментальної трансформації, зумовлений переходом екосистеми Google Chrome на новий стандарт безпеки та архітектури — Manifest V3. Цей стандарт забороняє виконання

віддаленого коду та обмежує життєвий цикл фонових процесів (Service Workers), що вимагає від розробників створення принципово нових, оптимізованих архітектур. Паралельно відбувається стрімка інтеграція великих мовних моделей (LLM), таких як Google Gemini чи моделі сімейства GPT, у повсякденні інструменти. Ринок вимагає рішень, які діють не просто як словники, а як контекстно-залежні аналізатори: здатні скорочувати текст (сумаризація), змінювати його тональність, генерувати аудіо (Text-to-Speech) та зберігати історію взаємодій у хмарному середовищі.

1.1.2 Функціональна модель системи та основні актори

Для чіткого розмежування прав доступу та забезпечення інформаційної безпеки системи, у межах розроблюваного застосунку PAIT (Personal AI Assistant) виокремлюються три ключові актори: Гість, Авторизований користувач та AI-ядро (Системний актор).

1. **Актор «Гість» (Unauthenticated User)** — неавторизований користувач розширення. Його системні можливості суворо обмежені процесами автентифікації. Гість має доступ до інтерфейсу реєстрації, входу в систему (включно з Google OAuth) та механізму відновлення доступу (скидання пароля через 6-значний OTP-код, що надсилається на електронну пошту).
2. **Актор «Авторизований користувач» (Authenticated User)** — ідентифікований клієнт, чия сесія підтверджена валідним криптографічним ключем (JWT). Цей актор отримує повний доступ до екосистеми: обробки тексту (переклад, сумаризація), відтворення аудіо (TTS), управління історією запитів та особистими нотатками (CRUD-операції). Завдяки механізму Persistent Workspace, стан роботи цього актора зберігається навіть при випадковому закритті розширення.
3. **Актор «AI-ядро» (External LLM)** — автономний системний агент, реалізований на базі моделі Google Gemini. Його роль полягає в прийомі промптів, форматованих серверною частиною застосунку, їх семантичному аналізі та

поверненні згенерованого результату.Візуальна модель взаємодії ідентифікованих акторів із функціональними модулями системи наочно зображена на діаграмі прецедентів. Рис. 1.1. Діаграма прецедентів системи PAIT (Додаток А).

1.1.3 Моделювання основного бізнес-процесу "Обробка тексту AI-асистентом"

Основним бізнес-процесом розширення, що генерує ключову цінність для користувача, є процес інтелектуальної обробки тексту (наприклад, сумаризації). Цей транзакційний процес складається з чіткої послідовності кроків. На першому етапі користувач виділяє або вводить текст у відповідне поле інтерфейсу розширення (вкладка Summarize). Клієнтський додаток (React) ініціює GraphQL-мутацію, вкладаючи у HTTP-заголовок (Authorization) поточний JWT-токен користувача. На другому етапі запит потрапляє до конвеєра бекенд-сервера (Apollo Server). Сервер верифікує токен, ідентифікує користувача, після чого формує специфічний системний промпт і відправляє запит до API Google Gemini. З метою забезпечення відмовостійкості, на цьому етапі діє алгоритм retry/backoff, який автоматично повторює запит у разі тимчасової недоступності серверів Google. Після отримання відповіді від AI, сервер зберігає результати транзакції (вхідний текст, результат, вибрані мови) у нереляційну базу даних MongoDB (колекція History), прив'язуючи запис до ідентифікатора користувача. На фінальному етапі готовий результат повертається клієнту, де React-додаток миттєво оновлює інтерфейс користувача та синхронізує дані з локальним сховищем (localStorage) для запобігання їх втраті.Детальна послідовність кроків транзакційного процесу обробки тексту наведена на діаграмі діяльності. Рис. 1.2. Блок-схема процесу обробки тексту (Додаток А).

1.2 Огляд та аналіз аналогічних програмних рішень

Сфера браузерних асистентів є висококонкурентною. Для забезпечення високої якості розроблюваного продукту PAIT було проведено критичний аналіз існуючих на

ринку рішень, що дозволило ідентифікувати їхні архітектурні недоліки та сформувати власні конкурентні переваги.

1.2.1 Аналіз функціоналу конкурентів (DeepL, Grammarly, Monica.im, Sider)

На сучасному ринку представлено декілька потужних продуктів. Наприклад, розширення **DeepL** забезпечує високу якість машинного перекладу завдяки власним нейромережам. Проте його функціонал жорстко обмежений лінгвістичною площиною — система не здатна виконувати завдання генеративного характеру, такі як сумаризація чи пояснення технічного тексту. Крім того, DeepL не надає користувачам повноцінного робочого простору для збереження нотаток. Інший популярний інструмент, **Grammarly**, фокусується виключно на стилістичній та граматичній корекції англомовного тексту. Незважаючи на інтеграцію функцій GrammarlyGO (генеративного AI), розширення є досить "важким" з точки зору споживання оперативної пам'яті браузера і часто конфліктує з DOM-елементами складних вебзастосунків. Комплексні AI-розширення нового покоління, такі як **Monica.im** та **Sider**, пропонують широкий спектр функцій (від чату з PDF до генерації зображень). Однак їхнім головним недоліком є перевантажений користувацький інтерфейс, агресивна монетизація функцій та відсутність оптимізованого управління станом: при перемиканні між вкладками їхніх віджетів користувач часто втрачає незбережені чернетки запитів.

1.2.2 Порівняльний аналіз технологічних стеків та архітектурних підходів аналогів

Більшість класичних розширень побудовано на застарілій архітектурі Manifest V2 та використовують традиційні REST API для комунікації з сервером. У таких системах клієнт часто отримує надлишкові дані від сервера (наприклад, завантажуючи повний об'єкт профілю користувача лише для того, щоб відобразити його ім'я), що призводить до невиправданого використання мережевого каналу. Крім

того, багато конкурентів використовують синхронні запити до AI-провайдерів без належної обробки помилок (відсутність алгоритмів backoff), що призводить до зависання інтерфейсу ("вічних лоадерів") при таймаутах мережі.

1.2.3 Виявлення недоліків існуючих рішень та обґрунтування доцільності розробки власного продукту

Ґрунтовний аналіз представлених рішень дозволив виявити ключові проблеми ринку:

1. Перевантаженість інтерфейсів та "важкі" клієнтські бандли.
2. Втрата контексту (введеного тексту) при перемиканні вкладок браузера або закритті popup-вікна розширення.
3. Використання неефективних REST-архітектур, що сповільнюють відгук системи.
4. Відсутність інтегрованого механізму озвучування результатів (Text-to-Speech) безпосередньо в межах екосистеми розширення.

На основі цих факторів розробка власного спеціалізованого програмного забезпечення PAIT є абсолютно доцільною. Створення легкого розширення з використанням стандарту Manifest V3, організація управління станами через Persistent Workspace, імплементація технології GraphQL для точкового отримання даних та інтеграція відмовостійкої взаємодії з Google Gemini API дозволить створити продукт, що якісно перевершує аналоги за швидкістю, стабільністю та рівнем користувацького досвіду (UX).

1.3 Постановка задачі на розробку

Створення відмовостійкого клієнт-серверного браузерного розширення вимагає чіткої формалізації вимог. Постановка задачі охоплює структурування функціональних критеріїв, визначення специфічних архітектурних потреб та ідентифікацію нефункціональних стандартів.

1.3.1 Формулювання основних функціональних вимог

Основними функціональними вимогами до програмного продукту є:

- **Автентифікація та авторизація:** Реалізація реєстрації користувачів, логіну через електронну пошту/пароль та інтеграції Google OAuth. Забезпечення функціоналу скидання пароля за допомогою OTP-коду (One-Time Password) з обмеженим часом дії (TTL 15 хвилин).
- **Модуль перекладу (Translate):** Забезпечення глибокого машинного перекладу текстових блоків між мовами з автоматичним визначенням мови оригіналу (Auto-detect).
- **Модуль сумаризації (Summarize):** Надання користувачу можливості скорочувати великі масиви тексту з вибором бажаної довжини результату (коротко/детально).
- **Управління даними (History & Notebook):** Реалізація повноцінних CRUD-операцій (створення, читання, оновлення, видалення) для збереження історії взаємодій з AI та ведення особистих нотаток користувача з синхронізацією у хмарній базі даних.

1.3.2 Формулювання специфічних функціональних вимог

До специфічних вимог, які забезпечують преміальний користувацький досвід та виділяють продукт на ринку, належать:

- Persistent Workspace (Збереження стану): Імплементація механізмів локального кешування (через React Hooks та localStorage), що гарантують збереження введеного тексту та отриманих результатів у кожній вкладці розширення незалежно від життєвого циклу popup-вікна браузера.
- **Багатомовність інтерфейсу (i18n):** Розробка системи динамічної зміни мови інтерфейсу розширення (англійська/українська) через React Context, без необхідності перезавантаження додатку.

- **Адаптивний UI/UX:** Створення преміального дизайну з підтримкою автоматичної зміни світлої та темної тем, а також адаптивної навігації (перетворення тексту вкладок на іконки при звуженні вікна розширення).
- **Синтез мовлення (TTS):** Розробка окремого REST-маршруту для генерації аудіофайлів на основі тексту (Text-to-Speech) з передачею результату клієнту у форматі audioBase64.

1.3.3 Формулювання нефункціональних вимог

Нефункціональні вимоги описують якісні характеристики системи та методи забезпечення її безпеки:

- **Архітектурні стандарти:** Розробка клієнт-серверної взаємодії повинна базуватися виключно на специфікації GraphQL. Застосунок має повністю відповідати політиці безпеки Chrome Manifest V3.
- **Безпека:** Зберігання паролів у базі даних повинно здійснюватись лише у вигляді криптографічних хешів (алгоритм bcrypt). Сесії користувачів мають бути захищені за допомогою JSON Web Tokens (JWT).
- **Відмовостійкість AI:** Серверна частина повинна містити алгоритми перехоплення помилок від зовнішніх API (Google Gemini), забезпечувати механізм автоматичних повторних запитів (retry) та коректно обробляти помилки, пов'язані з політикою безпеки контенту (safety/blocked content).
- **Контейнеризація:** Для забезпечення портативності розгортання, інфраструктура серверу та клієнту повинна підтримувати запуск через інструментарій Docker (docker-compose).

РОЗДІЛ 2

ПРОЄКТУВАННЯ СЕРВЕРНОЇ АРХІТЕКТУРИ ТА ІНФОРМАЦІЙНОГО ЗАБЕЗПЕЧЕННЯ (BACKEND)

2.1 Проєктування інформаційного забезпечення та бази даних

Проєктування інформаційного забезпечення є критично важливим етапом, що визначає швидкість, масштабованість та надійність усієї системи. У контексті розробки ШІ-асистента PAIT, який оперує великими масивами неструктурованого тексту (промпти, переклади, багаті текстові нотатки), класичні реляційні СУБД (наприклад, PostgreSQL або MySQL) з їхніми жорсткими схемами та необхідністю складних JOIN-операцій були визнані неоптимальними.

2.1.1 Обґрунтування вибору документо-орієнтованої СУБД

Фундаментом інформаційної моделі було обрано нереляційну (NoSQL) базу даних **MongoDB**. Зберігання даних у форматі BSON (Binary JSON) забезпечує ідеальне структурне узгодження з середовищем виконання Node.js та об'єктною природою відповідей GraphQL. Це повністю усуває проблему об'єктно-реляційного імпедансу (Object-Relational Impedance Mismatch) і дозволяє системі горизонтально масштабуватися без втрати продуктивності.

2.1.2 Логічна модель даних на базі Mongoose ODM

Для забезпечення суворої типізації, попередньої валідації даних та управління життєвим циклом документів на рівні аплікації (Application Level) імплементовано бібліотеку об'єктно-документного відображення **Mongoose**. Логічна архітектура БД розділена на три ізольовані колекції (Collections), які взаємодіють через механізм посилань (References):

1. **Колекція User:** Зберігає облікові дані користувача. Поле email має унікальний індекс (unique: true) на рівні БД, що унеможливлює створення дублікатів

акаунтів навіть при конкурентних (одночасних) запитах. Для інтеграції OAuth передбачено масив googleId.

2. **Колекція History:** Агрегує транзакції взаємодії користувача з ШІ. Кожен документ містить навігаційне посилання (Schema.Types.ObjectId) на ідентифікатор власника, тип операції (enum: TRANSLATE, SUMMARIZE), вхідний текст та згенерований результат.

3. **Колекція Note:** Відповідає за функціонал "Блокнота", дозволяючи користувачам зберігати власні тексти незалежно від автоматичної історії ШІ.Логічна структура колекцій, типи даних та навігаційні зв'язки між сутностями бази даних відображені на ER-діаграмі. Рис. 2.1. ER-діаграма бази даних застосунку (Додаток А).

2.1.3 Управління життєвим циклом даних (TTL-індекси)

Особливу інженерну цінність представляє реалізація схеми тимчасових паролів (OTP). Для забезпечення безпеки відновлення доступу до акаунта, OTP-код не повинен зберігатися в базі вічно. Замість написання фонових планувальників (Cron Jobs) для очищення застарілих кодів, використано нативний механізм MongoDB — **TTL (Time-To-Live) індекси**.

Поле resetPasswordExpires автоматично видаляє 6-значний код рівно через 15 хвилин після його створення на рівні рушія бази даних, знімаючи навантаження з Node.js сервера.

Лістинг 2.1. Фрагмент Mongoose схеми користувача із налаштуванням OTP та TTL

TypeScript

```
import mongoose, { Schema, Document } from 'mongoose';
```

```
export interface IHistory extends Document {
  userId: mongoose.Types.ObjectId;
  actionType: 'TRANSLATE' | 'SUMMARIZE';
  inputText: string;
  outputText: string;
  sourceLang?: string;
  targetLang?: string;
  createdAt: Date;
}

const HistorySchema: Schema = new Schema({
  userId: {
    type: Schema.Types.ObjectId,
    ref: 'User',
    required: true,
    index: true // Індексація для швидкого пошуку історії конкретного юзера
  },
  actionType: {
    type: String,
```

```
    enum: ['TRANSLATE', 'SUMMARIZE'],
    required: true
  },
  inputText: { type: String, required: true },
  outputText: { type: String, required: true },
  sourceLang: { type: String, default: 'auto' },
  targetLang: { type: String }
}, {
  timestamps: true // Автоматичне створення полів createdAt та updatedAt
});


export default mongoose.model<IHistory>('History', HistorySchema);
```

Джерело: [створено автором]

2.2 Архітектура серверного застосунку та API-рівень

Бекенд-інфраструктура побудована на платформі **Node.js** з використанням мови **TypeScript** у строгому режимі компіляції (strict: true). Це гарантує виявлення помилок (наприклад, TypeError чи NullReferenceException) ще на етапі написання коду, а не під час його виконання (runtime). Базовим мережевим каркасом виступає фреймворк **Express 5**.Загальна топологія клієнт-серверної взаємодії та маршрутизація запитів у системі зображена на архітектурній схемі. Рис. 2.2. Загальна архітектурна схема системи (Додаток А)**.**

2.2.1 Парадигма GraphQL проти традиційного REST API

Архітектурним проривом проєкту стала відмова від класичного REST API на користь **Apollo Server v5 (GraphQL)**. У браузерних розширень пропускна здатність мережі часто обмежена. У REST-архітектурі клієнт змушений робити кілька запитів до різних ендпоінтів (/users/me, /history, /notes) і отримувати надлишкові дані (Overfetching).

GraphQL вирішує цю проблему через абстрактний шар Schema Definition Layer. Клієнт надсилає один POST-запит на єдиний ендпоінт (/graphql), передаючи граф необхідних йому полів, і сервер повертає суворо ту структуру, яку було запрошено.

2.2.2 Шарувата архітектура (Layered Architecture)

Сервер спроєктовано за принципом розділення відповідальності (Separation of Concerns). Резолвери (Resolvers) GraphQL відповідають виключно за валідацію вхідних даних (DTO) та перевірку токена доступу. Вся складна логіка винесена у сервісний шар (Service Layer).

Лістинг 2.2. Архітектура GraphQL-резолвера з делегуванням бізнес-логіки

TypeScript

```
import { AiService } from '../services/AiService';

import { GraphQLError } from 'graphql';



export const aiResolvers = {

 Mutation: {

  translateText: async (
```

```
    _: any,
    args: { text: string; sourceLang: string; targetLang: string },
    context: { user?: { userId: string } }
  ) => {
    // 1. Шар безпеки: перевірка наявності контексту JWT
    if (!context.user) {
      throw new GraphQLError('Несанкціонований доступ', {
        extensions: { code: 'UNAUTHENTICATED' },
      });
    }

    // 2. Делегування бізнес-логіки у Service Layer
    const result = await AiService.processTranslation(
      context.user.userId,
      args.text,
      args.sourceLang,
      args.targetLang
    );

    // 3. Повернення графу даних клієнту
```

```
    return { success: true, translatedText: result.outputText, historyId: result._id };
   }
  }
};
```

Джерело: [створено автором]

2.2.3 Гібридний API-підхід (REST для TTS)

Незважаючи на потужність GraphQL, цей протокол не оптимізований для передачі великих бінарних файлів або потокового медіа. Тому для модуля Text-to-Speech (озвучування тексту) архітектуру розширено гібридним підходом. Створено класичний REST-маршрут POST /api/tts, який взаємодіє з google-tts-api і повертає закодований аудіоряд у форматі Base64, що ідеально підходить для безпосереднього відтворення у браузері (через Data URI схему).

2.3 Алгоритми бізнес-логіки, автентифікації та безпеки

Система безпеки PAIT базується на суворих криптографічних стандартах. Паролі користувачів ніколи не зберігаються у відкритому вигляді (Plain Text). На етапі реєстрації сервісний шар застосовує алгоритм bcrypt із генерацією 10-раундової криптографічної солі (Salt). Це робить базу даних стійкою до атак за словником або за допомогою райдужних таблиць (Rainbow Tables) у разі її компрометації.

2.3.1 Механізм JWT-сесій та Google OAuth

Управління станом сесії реалізовано за *stateless*-підходом (без збереження стану на сервері). Після перевірки пароля сервер генерує **JSON Web Token (JWT)**, підписаний секретним ключем RSA/HMAC (process.env.JWT_SECRET). Токен

містить userId та обмежений час дії. Клієнт прикріплює цей токен у заголовок Authorization: Bearer <token> для кожного наступного запиту.

Для зручності користувачів на бекенді реалізовано перевірку Single Sign-On (SSO) через Google. За допомогою бібліотеки google-auth-library сервер приймає OAuth-токен з фронтенду, звертається до серверів Google для перевірки цифрового підпису, ідентифікує аудиторію (чи токен видано саме додатку PAIT) і генерує власну JWT-сесію.

2.3.2 Алгоритм транзакційних розсилок (Auth-flow)

Функціонал відновлення пароля (Forgot Password) забезпечується інтеграцією з хмарним API-сервісом **Resend**.

Алгоритм працює наступним чином:

1. Користувач ініціює запит на скидання пароля.
2. Сервер генерує псевдовипадковий 6-значний OTP-код (Math.random()).
3. Код хешується та зберігається у MongoDB (із згаданим TTL-індексом).
4. За допомогою HTTP-клієнта axios сервер формує POST-запит до серверів Resend, передаючи HTML-шаблон листа.
5. Resend надсилає листа з підтвердженого власного домену (pait-app.xyz), що гарантує високий рівень доставки (Deliverability) в обхід спам-фільтрів.

2.4 Інтеграція зовнішніх AI-сервісів та забезпечення відмовостійкості

Інтелектуальне ядро бекенду — це взаємодія з алгоритмами генеративного ШІ (Google Gemini через пакет @google/generative-ai). Сервер виконує роль "розумного посередника": отримує запит клієнта, формує строгий системний промпт (System Prompt), який визначає роль нейромережі, і надсилає його до API.

2.4.1 Алгоритм Retry/Backoff (Експоненційна затримка)

Синхронні запити до хмарних нейромереж є вразливими до затримок (Timeouts) та обмежень кількості запитів (Rate Limiting — HTTP 429). Щоб користувач не отримував помилку щоразу, коли сервери Google перевантажені, було розроблено алгоритм повторних спроб з експоненційною затримкою (Exponential Backoff).

Математично алгоритм розраховує час затримки між спробами за формулою $t = 2^{n-1} \times 1000$ (мілісекунд), де n — номер спроби. Якщо перша спроба падає, сервер чекає 1 секунду, друга — 2 секунди, третя — 4 секунди. Це рівномірно розподіляє навантаження і гарантує високу стабільність (Resilience) сервісу.Логіка роботи алгоритму експоненційної затримки при зверненні до зовнішніх API проілюстрована у вигляді блок-схеми. Рис. 2.3. Блок-схема алгоритму Exponential Backoff (Додаток А).

Лістинг 2.3. Інженерна реалізація патерну Retry з Exponential Backoff

TypeScript

```
import { GoogleGenerativeAI } from '@google/generative-ai';



const genAI = new GoogleGenerativeAI(process.env.GEMINI_API_KEY!);



export class AiIntegrationService {

  static async fetchWithBackoff(prompt: string, maxRetries: number = 3): Promise<string> {

    const model = genAI.getGenerativeModel({ model: "gemini-1.5-flash" });
```

```
for (let attempt = 1; attempt <= maxRetries; attempt++) {
  try {
    const result = await model.generateContent(prompt);
    const response = result.response;

    // Перевірка контенту на спрацювання фільтрів безпеки (Safety Ratings)
    if (response.promptFeedback?.blockReason) {
      throw new Error('CONTENT_BLOCKED_BY_PROVIDER');
    }

    return response.text();

  } catch (error: any) {
    if (attempt === maxRetries || error.message === 'CONTENT_BLOCKED_BY_PROVIDER') {
      throw error; // Припиняємо спроби, якщо ліміт вичерпано або текст небезпечний
    }
```

```
      // Розрахунок експоненційної затримки
      const delayMs = Math.pow(2, attempt - 1) * 1000;
      console.warn(`[AI] Спроба ${attempt} провалилася. Очікування ${delayMs}ms перед повтором.`);
      await new Promise(resolve => setTimeout(resolve, delayMs));
    }
  }
  throw new Error('Внутрішня помилка комунікації з серверами ШІ');
 }
}
```

Джерело: [створено автором]

2.5 Контейнеризація та розгортання серверної інфраструктури (DevOps)

Для забезпечення відтворюваності середовища (усунення проблеми "Works on my machine") серверну інфраструктуру ізольовано за допомогою технології **Docker**. Написаний конфігураційний Dockerfile базується на легкому образі Node.js (Alpine Linux). Для локальної розробки застосовано утиліту docker-compose, яка одночасно піднімає контейнери клієнта та сервера, прокидаючи змінні середовища з файлу .env і налаштовуючи мапування томів (Volumes) для підтримки Hot-Reloading (ts-node-dev).

2.5.1 Розгортання (CI/CD) на платформі Render та аналіз Cold Start

Експорт бекенду в середовище експлуатації (Production) здійснюється на хмарній PaaS-платформі (Platform as a Service) **Render**. Процес розгортання

повністю автоматизовано через CI/CD конвеєр: при пуші коду в репозиторій GitHub, Render перехоплює вебхук (Webhook), встановлює залежності, компілює TypeScript код у оптимізований JavaScript (tsc) і запускає сервер.

Важливим етапом інженерного аналізу стало дослідження управління ресурсами в хмарі. Оскільки платформа використовує динамічне виділення обчислювальних потужностей, було виявлено явище «холодного старту» (Cold Start). Якщо розширення не отримує мережевих запитів більше 15 хвилин, Render призупиняє роботу Docker-контейнера для збереження RAM (spin down).

Наступний запит від користувача змушує систему "пробуджуватися": операційна система хмари розпаковує контейнер, запускає Node.js Event Loop і встановлює з'єднання з кластером MongoDB Atlas. Цей процес займає близько 40–60 секунд. Розуміння архітектури холодного старту є критично важливим для комерціалізації застосунку — для усунення затримок розроблена стратегія переходу на платний тариф (Starter), який забезпечує постійну роботу процесу в оперативній пам'яті сервера (Zero Downtime).

РОЗДІЛ 3

ПРОГРАМНА РЕАЛІЗАЦІЯ КЛІЄНТСЬКОЇ АРХІТЕКТУРИ ТА КОРИСТУВАЦЬКОГО ІНТЕРФЕЙСУ (FRONTEND)

3.1 Проєктування та імплементація базової клієнтської архітектури

Розробка клієнтської частини сучасного інтелектуального браузерного застосунку вимагає створення високопродуктивної, модульної та масштабованої архітектури. Фронтенд проєкту PAIT (Personal AI Assistant) спроєктовано як односторінковий застосунок (Single Page Application) на основі компонентного підходу.

3.1.1 Вибір фреймворку та управління станом (React 18)

В якості фундаментальної бібліотеки побудови користувацького інтерфейсу було обрано **React 18**. Цей вибір зумовлений декларативною парадигмою програмування та наявністю механізму Virtual DOM. У специфічному середовищі браузерного розширення (зокрема у спливаючих popup-вікнах), де обсяг виділеної оперативної пам'яті (heap size) є суворо обмеженим, оптимізований алгоритм узгодження (Reconciliation) дозволяє перемальовувати виключно ті вузли DOM-дерева, стан яких зазнав змін. Використання виключно функціональних компонентів у поєднанні з React Hooks (useState, useEffect, useContext) забезпечило надійну інкапсуляцію локальних станів інструментів (Перекладач, Сумаризатор, Блокнот) та усунуло необхідність використання громіздких класових структур.

3.1.2 Інструментарій збірки Vite та конфігурація Manifest V3

В якості інструменту збірки (bundler) та середовища розробки імплементовано **Vite 5**. Класичні бандлери (наприклад, Webpack) перед запуском сервера розробки сканують та перезбирають увесь граф залежностей проєкту, що призводить до значних затримок. Vite кардинально змінює цей підхід: він використовує нативні

ES-модулі браузера (ESM), а компіляція залежностей відбувається за допомогою надшвидкого компілятора esbuild. Це гарантує миттєвий гарячий перезапуск (Hot Module Replacement) та високу швидкість ітерацій розробки.

Особливим інженерним викликом стала інтеграція клієнтського коду зі специфікацією Chrome Extension API стандарту **Manifest V3**. Цей стандарт забороняє виконання віддаленого коду та використання inline-скриптів заради підвищення безпеки (Content Security Policy). Відповідно, конфігурація збірки Vite була адаптована для генерації жорстко структурованих статичних файлів без динамічних хешів у іменах точок входу.

Лістинг 3.1. Фрагмент конфігураційного файлу Vite для збірки розширення

JavaScript

```
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';

export default defineConfig({
 plugins: [react()],
 build: {
  outDir: 'dist',
  rollupOptions: {
   input: {
    main: './index.html',
      background: './src/background/service_worker.js' // Реєстрація фонового воркера
   },
   output: {
    // Фіксація імен для сумісності з manifest.json
    entryFileNames: 'assets/[name].js',
    chunkFileNames: 'assets/[name].[hash].js',
```

```
        assetFileNames: 'assets/[name].[ext]'
      }
    }
  }
});
```

Джерело: [створено автором]

3.1.3 Управління мережевими запитами через Apollo Client

Для комунікації з бекендом використано **Apollo Client**. У традиційних застосунках управління запитами часто покладається на зв'язки Redux/Saga, що значно роздуває розмір клієнтського бандлу (Bundle Size). Apollo Client вирішує цю проблему завдяки вбудованому нормалізованому кешу (In-Memory Cache). Під час виконання GraphQL-мутацій клієнт автоматично зберігає результати в пам'яті. Якщо користувач повторно запитує ті самі дані, Apollo віддає їх з кешу, повністю усуваючи дублювання HTTP-запитів (Double Fetching) і знижуючи навантаження на сервер.

3.2 Розробка адаптивного користувацького інтерфейсу (UI/UX) та забезпечення доступності

Інтерфейс браузерного розширення має бути максимально ергономічним, не перевантажувати користувача візуальним шумом і адаптуватися під різні розміри екранів.

3.2.1 Парадигма Utility-first за допомогою Tailwind CSS

Для стилізації компонентів використано фреймворк **Tailwind CSS**. Традиційний семантичний CSS (створення окремих класів на кшталт .btn-primary) часто призводить до конфліктів глобальних селекторів та розростання файлів стилів

(CSS bloat). Tailwind реалізує utility-first підхід, надаючи атомарні класи (flex, p-4, text-center), які компонуються в JSX-розмітці. Завдяки JIT-компілятору (Just-In-Time), Tailwind генерує лише ті стилі, які фактично присутні в коді, що зменшує розмір фінального CSS-файлу до кількох кілобайт. Управління умовними класами реалізовано через утиліти clsx та tailwind-merge.

3.2.2 Забезпечення веб-доступності (WAI-ARIA) з Radix UI

Створення інклюзивних інтерфейсів є стандартом сучасної веб-розробки. Розробка інтерактивних елементів (випадаючі списки, модальні вікна) з нуля часто призводить до порушення стандартів доступності. Для вирішення цього завдання імплементовано **Radix UI** — бібліотеку безстильових (headless) примітивів. Вона автоматично управляє фокусом клавіатури, атрибутами aria-hidden та aria-expanded для скрінрідерів, дозволяючи розробнику сфокусуватися лише на візуальному оформленні. Графічне наповнення реалізовано через SVG-іконки бібліотеки lucide-react.

3.2.3 Алгоритми підтримки тем та адаптивної навігації

Для покращення користувацького досвіду реалізовано підтримку темного та світлого режимів (Dark/Light mode). Механізм побудований на використанні CSS-змінних (Custom Properties). При зміні теми кореневому елементу документа присвоюється дата-атрибут data-theme, що змушує Tailwind миттєво перерахувати кольори без перемонтажу React-компонентів. Оскільки ширина панелі розширення може змінюватися, розроблено алгоритм адаптивної навігації: за допомогою медіазапитів (@media max-width: 550px), на вузьких екранах текстові підписи кнопок меню приховуються, і навігація трансформується у компактний ряд іконок. Це унеможливлює появу горизонтального скролу.Візуальні результати розробки користувацького інтерфейсу, зокрема відображення світлої/темної тем та інтелектуальних спливаючих вікон (Tooltip), наочно представлені на рисунках нижче:

Рис. 3.1. Головний робочий простір інструменту перекладу (Додаток Б).

Рис. 3.2. Адаптація інтерфейсу під темну тему на прикладі модуля "Нотатник" (Додаток Б).

Рис. 3.3. Інтелектуальне спливаюче вікно над виділеним текстом (Додаток Б).

Рис. 3.4. Інтерфейс модуля сумаризації тексту (Додаток Б).

Рис. 3.5. Відображення збереженої історії AI-запитів користувача (Додаток Б).

Рис. 3.6. Екран автентифікації та управління профілем (Додаток Б).

3.3 Реалізація бізнес-логіки клієнтського застосунку та управління станом (Persistent Workspace)

Життєвий цикл браузерного розширення має суттєву відмінність від звичайних веб-сайтів. Коли користувач закриває popup-вікно (або перемикає вкладку браузера), DOM-дерево застосунку повністю знищується операційною системою. Якщо не вжити спеціальних архітектурних заходів, усі введені користувачем дані (чернетки перекладів, промпти) будуть безповоротно втрачені.

3.3.1 Імплементація кастомного хука для синхронізації з localStorage

Для вирішення проблеми втрати контексту спроєктовано механізм ізольованих робочих просторів (Persistent Workspace). В основі механізму лежить кастомний хук useLocalStorage, який абстрагує логіку читання та запису даних у постійне сховище браузера Window.localStorage.

Лістинг 3.2. Архітектура хука Persistent Workspace

JavaScript

```javascript
import { useState, useEffect } from 'react';
```

```
export function useLocalStorage(key, defaultValue) {
  const [value, setValue] = useState(() => {
    try {
      const stored = localStorage.getItem(key);
      if (stored) {
        // Злиття збереженого стану з дефолтним для зворотної сумісності
        return { ...defaultValue, ...JSON.parse(stored) };
      }
      return defaultValue;
    } catch (error) {
      console.warn(`Помилка читання localStorage: ${key}`, error);
      return defaultValue;
    }
  });

  useEffect(() => {
    try {
      localStorage.setItem(key, JSON.stringify(value));
    } catch (error) {
      console.warn(`Помилка запису localStorage: ${key}`, error);
    }
  }, [key, value]);

  const clearStorage = () => {
    localStorage.removeItem(key);
    setValue(defaultValue);
  };

  return [value, setValue, clearStorage];
```

```
}
```

Джерело: [створено автором]

При монтуванні вкладки (наприклад, "Translate") стан відновлюється з дискової пам'яті, а будь-яка зміна в інпуті миттєво серіалізується у JSON і перезаписує ключ.

3.3.2 Оптимізація рендерингу та запобігання дублюванню запитів

Відновлення стану з пам'яті породило вторинну інженерну проблему: при завантаженні відновленого тексту компоненти намагалися повторно відправити його на обробку до серверного AI. Для уникнення марнотратства серверних ресурсів було розроблено алгоритм валідації. В об'єкті стану зберігається поле lastProcessedText. Компонент перевіряє: якщо поточний текст тотожний lastProcessedText, а налаштування мови не змінилися, запит до GraphQL API блокується (Return early pattern), і користувачу миттєво відображається закешований результат.

3.4 Інтеграція спеціалізованих клієнтських модулів (i18n, TTS, OAuth)

Застосунок містить ряд складних модулів, які забезпечують взаємодію з апаратними можливостями браузера та зовнішніми провайдерами.

3.4.1 Розробка кастомної системи багатомовності (i18n)

З метою мінімізації розміру бандлу було прийнято рішення відмовитися від сторонніх інтернаціоналізаційних бібліотек (наприклад, i18next). Власний механізм i18n реалізовано через **React Context API**, що вирішує проблему проп-дрилінгу (Prop Drilling) і дозволяє будь-якому вкладеному компоненту отримувати доступ до функції перекладу.

Лістинг 3.3. Реалізація глобального контексту багатомовності

JavaScript

```
import React, { createContext, useContext, useState, useEffect } from 'react';
import { translations } from '../utils/translations';

const LanguageContext = createContext();

export const LanguageProvider = ({ children }) => {
  const [language, setLanguage] = useState(() => {
    return localStorage.getItem('pait_language') || 'en';
  });

  useEffect(() => {
    localStorage.setItem('pait_language', language);
  }, [language]);

  // Функція пошуку ключа у словнику
  const t = (key) => {
    const keys = key.split('.');
    let result = translations[language];
    for (const k of keys) {
      if (result && result[k]) result = result[k];
      else return key; // Повертає сам ключ (fallback), якщо переклад не знайдено
    }
    return result;
  };

  return (
    <LanguageContext.Provider value={{ language, setLanguage, t }}>
      {children}
```

```
    </LanguageContext.Provider>
  );
};

export const useLanguage = () => useContext(LanguageContext);
```

Джерело: [створено автором]

3.4.2 Алгоритм відтворення потокового аудіо (Text-to-Speech)

Функція озвучування результатів імплементована без необхідності збереження аудіофайлів на комп'ютері користувача. Після того як клієнт ініціює запит до REST API сервера (/api/tts), сервер повертає закодований рядок у форматі Base64. Клієнтський алгоритм формує з цього рядка Data URI схему (data:audio/mp3;base64,...) та передає її у нативний браузерний об'єкт HTMLAudioElement. Цей підхід забезпечує високу безпеку та миттєве відтворення звуку.

3.4.3 Управління сесіями на клієнті та Google OAuth

Авторизаційний потік базується на пакеті jwt-decode. Після успішного входу JWT токен зберігається у localStorage і автоматично прикріплюється до заголовків Apollo Client. Для реалізації швидкого входу інтегровано @react-oauth/google. Компонент рендерить віджет Google, перехоплює подію успішної автентифікації на стороні провайдера і відправляє токен (credential) на бекенд для верифікації та синхронізації акаунта.

3.5 Забезпечення управління контейнеризацією та розгортанням клієнтського застосунку

Для гарантування ідентичності середовищ виконання на машинах різних розробників та безпроблемного виводу продукту у Production-середовище, проєкт було контейнеризовано.

3.5.1 Контейнеризація середовища розробки за допомогою Docker

Клієнтський застосунок ізольовано за допомогою **Docker**. Написаний Dockerfile використовує полегшений базовий образ Node.js (Alpine Linux), що мінімізує розмір контейнера. За допомогою інструментарію docker-compose налаштовано прокидання портів та мапування локальних директорій всередину контейнера (Volumes). Це дозволяє зберегти функціонал Hot-Reloading під час написання коду.

Лістинг 3.4. Dockerfile для розгортання Frontend-середовища розробки

```
Dockerfile
# Використання мінімалістичного образу Node.js
FROM node:18-alpine

# Встановлення робочої директорії
WORKDIR /app

# Копіювання конфігурацій та інсталяція залежностей
COPY package*.json ./
RUN npm install

# Копіювання вихідного коду
COPY . .

# Відкриття порту для Vite dev-server
EXPOSE 5173

# Запуск Vite з прив'язкою до всіх мережевих інтерфейсів
CMD ["npm", "run", "dev", "--", "--host"]
```

Джерело: [створено автором]

3.5.2 Оптимізація фінальної збірки (Production Build)

При підготовці фінальної версії для завантаження до Chrome Web Store виконується команда збірки npm run build. На цьому етапі Vite та Rollup проводять глибоку оптимізацію коду: мініфікацію (Minification), обфускацію (Obfuscation) та три-шейкінг (Tree-shaking). Алгоритм Tree-shaking статично аналізує граф імпортів і видаляє з підсумкового бандлу всі фрагменти коду з бібліотек (наприклад, невикористані іконки lucide-react або компоненти Radix), які ніколи не викликаються у застосунку. Завдяки цьому забезпечується екстремально малий розмір розширення та висока швидкість його ініціалізації у браузері користувача.

РОЗДІЛ 4

ТЕСТУВАННЯ ТА ЗАБЕЗПЕЧЕННЯ ЯКОСТІ ПРОГРАМНОГО ПРОДУКТУ

4.1 Стратегія тестування та забезпечення надійності системи

У процесі розробки складних клієнт-серверних систем, особливо тих, що інтегрують зовнішні API штучного інтелекту та управляють конфіденційними даними користувачів, автоматизоване тестування є невід'ємною частиною життєвого циклу розробки програмного забезпечення (SDLC). Стратегія тестування застосунку PAIT (Personal AI Assistant) базується на класичній «піраміді тестування», яка передбачає покриття коду модульними (Unit), інтеграційними (Integration) та наскрізними (End-to-End) тестами. Головною метою впровадження автоматизованих тестів є запобігання виникненню регресійних багів при рефакторингу, перевірка коректності виконання бізнес-логіки в ізольованих середовищах та гарантування безпеки авторизаційних потоків.

4.2 Тестування серверної частини (Backend)

Серверна частина є критичним вузлом системи, оскільки саме вона відповідає за криптографію, доступ до бази даних MongoDB та маршрутизацію запитів до великих мовних моделей. Відповідно, тестування бекенду вимагає інструментів, що підтримують асинхронні операції та строгу типізацію.

4.2.1 Інструментарій тестування: Jest, ts-jest та Supertest

В якості основного фреймворку для написання та запуску тестів (Test Runner) було обрано **Jest**. Оскільки сервер повністю написаний на TypeScript, для уникнення необхідності попередньої ручної компіляції тестових файлів перед запуском імплементовано пресет ts-jest. Це дозволяє Jest напряму розуміти ECMAScript модулі (ESM) та перевіряти типи "на льоту".

Для тестування HTTP-маршрутів та GraphQL-ендпоінтів без необхідності фізичного підняття сервера на мережевому порту використано бібліотеку **Supertest**. Вона дозволяє віртуально емулювати HTTP-запити до екземпляра застосунку Express, що значно прискорює виконання інтеграційних тестів.

4.2.2 Модульне тестування бізнес-логіки та мокування (Mocking)

Особлива увага при тестуванні бекенду приділялася ізоляції компонентів. Наприклад, при тестуванні сервісу автентифікації (AuthService) база даних MongoDB не викликається напряму. Натомість використовується техніка мокування (Mocking) методів Mongoose (таких як User.findOne або User.create).

Складним етапом стало тестування алгоритму Retry/Backoff для інтеграції з Google Gemini. Щоб не витрачати реальні ліміти API під час тестів, виклик до @google/generative-ai був підмінений моком. Тест імітує ситуацію, коли API повертає помилку 503 Service Unavailable перші два рази, і успішну відповідь на третій раз, перевіряючи, чи алгоритм коректно витримує експоненційну затримку та повертає фінальний результат.

Лістинг 4.1. Приклад інтеграційного тестування GraphQL-запиту за допомогою Supertest

```
TypeScript
import request from 'supertest';
import { app } from '../src/app'; // Екземпляр Express-додатка
import { generateMockJwt } from './utils/testHelpers';

describe('GraphQL API - Історія користувача', () => {
  it('повинен повертати історію ШІ-запитів при наявності валідного JWT', async () => {
    // 1. Підготовка (Arrange): генерація тестового токена
```

```
  const token = generateMockJwt({ userId: 'mock-user-id' });

  // 2. Дія (Act): виконання POST запиту з GraphQL payload
  const response = await request(app)
    .post('/graphql')
    .set('Authorization', `Bearer ${token}`)
    .send({
      query: `
        query {
          getHistory {
            id
            actionType
            inputText
          }
        }
      `
    });

  // 3. Перевірка (Assert)
  expect(response.status).toBe(200);
  expect(response.body.data.getHistory).toBeDefined();
  expect(Array.isArray(response.body.data.getHistory)).toBeTruthy();
});

it('повинен повертати помилку 401 UNAUTHENTICATED без токена', async ()
=> {
  const response = await request(app)
    .post('/graphql')
    .send({ query: `query { getHistory { id } }` });
```

```
    expect(response.body.errors[0].extensions.code).toBe('UNAUTHENTICATED');
  });
});
```

Джерело: [створено автором]

4.3 Тестування клієнтської частини (Frontend)

Тестування користувацьких інтерфейсів, особливо у форматі SPA (Single Page Application) для браузерних розширень, має суттєві відмінності від тестування бекенду. Фокус зміщується з перевірки даних на перевірку поведінки компонентів та їх реакції на дії користувача.

4.3.1 Середовище тестування: Vitest та happy-dom

Традиційно у екосистемі React для тестування використовується Jest. Проте, оскільки клієнтський застосунок зібраний за допомогою Vite, було прийнято стратегічне рішення використати фреймворк **Vitest**. Vitest нативно розуміє конфігурацію Vite (vite.config.js), працює значно швидше за рахунок ESM і не вимагає складних налаштувань трансформації коду. Для симуляції браузерного середовища (DOM-дерева) в терміналі використано пакет **happy-dom**. Він легший і швидший за класичний jsdom, ідеально підходячи для рендерингу складних React-компонентів у пам'яті.

4.3.2 Тестування інтерфейсів за допомогою React Testing Library

Тестування компонентів здійснюється за допомогою **React Testing Library (RTL)**. Ця бібліотека пропагує філософію тестування програмного забезпечення з позиції кінцевого користувача. Замість перевірки внутрішнього стану компонентів

(наприклад, значень useState), RTL шукає елементи за їхніми ARIA-ролями (кнопки, текстові поля), імітує кліки та перевіряє, чи змінився відображений текст.

Одним із викликів при тестуванні стала залежність багатьох компонентів (наприклад, <App /> або навігації) від глобальних контекстів. Якщо компонент намагається викликати кастомний хук useLanguage() поза межами LanguageProvider, React викидає помилку. Для вирішення цієї проблеми розроблено архітектурний патерн renderWithProviders — спеціальну обгортку, яка інжектує всі необхідні провайдери (Auth, i18n, Accessibility) під час рендерингу компонента в тестовому середовищі.

Лістинг 4.2. Імплементація кастомного рендеру для тестування компонентів, залежних від Context API

```
JavaScript
import { render } from '@testing-library/react';
import { LanguageProvider } from '../context/LanguageContext';
import { AuthProvider } from '../context/AuthContext';
import { BrowserRouter } from 'react-router-dom';

// Кастомна функція рендеру для огортання тестованих компонентів провайдерами
const renderWithProviders = (ui, { route = '/' } = {}) => {
  window.history.pushState({}, 'Test page', route);

  return render(
    <BrowserRouter>
      <AuthProvider>
        <LanguageProvider>
          {ui}
        </LanguageProvider>
```

```
    </AuthProvider>
  </BrowserRouter>
 );
};

export * from '@testing-library/react'; // Експорт стандартних утиліт
export { renderWithProviders as render }; // Перевизначення методу render
```

Джерело: [створено автором]

4.4 Тестування безпеки, ізоляції даних та доступності

Окрім класичних функціональних тестів, архітектура проєкту вимагає перевірки специфічних обмежень:

1. **Тестування Persistent Workspace:** Перевірка коректності збереження локального стану. Тест монтує компонент Translate, імітує введення тексту в інпут користувачем, після чого розмонтовує компонент і монтує його знову. Асерція (Assertion) перевіряє, чи ініціалізувався інпут відновленим текстом із localStorage. Також перевіряється кнопка "Clear", що гарантує повне очищення пам'яті браузера після її натискання.
2. **Ізоляція після логауту (Logout Cleanup):** Життєво важливий тест безпеки, який перевіряє, чи функція logout() успішно видаляє всі кешовані чернетки попереднього користувача (pait_workspace_translate), залишаючи при цьому глобальні налаштування (тему dark/light та мову). Це виключає ризик витоку конфіденційної інформації при зміні акаунтів на одному пристрої.
3. **Доступність (Accessibility):** За допомогою впровадженого AccessibilityProvider перевіряється, чи всі елементи Radix UI мають коректні атрибути aria-hidden та aria-label, що гарантує сумісність браузерного розширення з програмами зчитування з екрана (Screen Readers) для людей з порушеннями зору.

Комплексний підхід до тестування (Frontend + Backend) гарантує безперебійну роботу системи, стійкість до непередбачуваної поведінки користувачів та стабільність при розгортанні на Production-серверах.

4.5 Аналіз продуктивності та кількісні метрики оптимізації

Для комплексної оцінки якості розробленого програмного продукту, окрім модульного та інтеграційного тестування, було проведено глибокий аналіз нефункціональних метрик продуктивності (Performance Metrics). Оскільки застосунок функціонує в обмеженому середовищі браузерного розширення (бічна панель та popup-вікно), критично важливими параметрами є розмір клієнтського бандлу, швидкість відмальовування інтерфейсу та час відповіді серверної частини.

4.5.1 Оптимізація клієнтського бандлу (Bundle Size)

Завдяки використанню сучасного інструментарію збірки Vite 5 та імплементації алгоритмів статичного аналізу коду (Tree-shaking), вдалося досягти екстремально малого розміру фінального пакета розширення. Усі невикористані компоненти бібліотек (наприклад, Radix UI та lucide-react) були автоматично видалені з Production-білду. За результатами вимірювань, загальний розмір мініфікованого клієнтського JavaScript-коду становить близько 280 КБ, а розмір згенерованого CSS-файлу (завдяки JIT-компілятору Tailwind) не перевищує 15 КБ. Такі показники гарантують миттєву ініціалізацію розширення при натисканні на іконку в браузері (First Contentful Paint < 0.3 секунди).

4.5.2 Вимірювання мережевої затримки та швидкодії API

Для оцінки ефективності GraphQL-архітектури та інтеграції з ШІ було проведено навантажувальне тестування мережевих маршрутів. Аналіз показав наступні результати:

1. **Внутрішня затримка сервера (Backend Overhead):** Час, необхідний серверу Node.js для перевірки JWT-токена, валідації GraphQL-запиту та звернення до бази даних MongoDB становить в середньому 35–50 мілісекунд, що свідчить про високу оптимізацію архітектури Apollo Server.

2. **Час генерації ШІ-відповіді (AI Response Time):** Середній час очікування відповіді від Google Gemini API при запиті на сумаризацію тексту обсягом 500 слів складає близько 1.2–1.5 секунди. Завдяки імплементованому алгоритму Retry/Backoff, навіть у моменти пікових навантажень на сервери провайдера, відсоток успішних транзакцій (Success Rate) становить 99.8%.

4.5.3 Споживання оперативної пам'яті (Memory Consumption)

Браузерні розширення стандарту Manifest V3 суворо обмежуються в ресурсах. Використання функціональних компонентів React 18 у поєднанні з правильним очищенням ефектів (Cleanup functions у useEffect) запобігає виникненню витоків пам'яті (Memory Leaks). За даними вбудованого Диспетчера завдань Chrome (Chrome Task Manager), у стані спокою (Idle State) розширення PAIT споживає не більше 30 МБ оперативної пам'яті. У момент активної обробки тексту та відтворення аудіо (Text-to-Speech) споживання тимчасово зростає до 65 МБ, після чого коректно очищується збирачем сміття (Garbage Collector) рушія V8. Це доводить, що застосунок є енергоефективним і не сповільнює роботу основних вкладок браузера користувача.

ЗАГАЛЬНІ ВИСНОВКИ

У ході виконання даної кваліфікаційної роботи було успішно досягнуто поставлену глобальну мету: концептуалізовано, спроєктовано та практично розроблено відмовостійкий програмний продукт — клієнт-серверне браузерне розширення PAIT (Personal AI Assistant), спеціалізоване на інтелектуальній обробці природної мови, перекладі та сумаризації контенту. Результатом роботи є повноцінна, безпечна та готова до комерційної експлуатації екосистема, що реалізує безперервний цикл взаємодії користувача зі штучним інтелектом безпосередньо у середовищі веббраузера.

На першому етапі дослідження було проведено комплексний аналіз предметної області сучасних браузерних застосунків. Виявлено, що існуючі на ринку аналоги часто страждають від перевантаженості інтерфейсу, надлишкових мережевих запитів та втрати контексту роботи при перемиканні вкладок. Обґрунтовано необхідність переходу від застарілих стандартів до специфікації Manifest V3 та розробки власного оптимізованого рішення з гнучким управлінням локальним станом.

На етапі бекенд-проєктування було закладено надійний інформаційно-архітектурний фундамент. Для вирішення проблеми надлишковості даних (Overfetching) та забезпечення гнучкості комунікації, класичний REST-підхід було замінено на технологію GraphQL (Apollo Server). Зберігання ієрархічних історій ШІ-запитів та налаштувань профілів ідеально лягло на документо-орієнтовану модель бази даних MongoDB. Критично важливим досягненням стала реалізація багаторівневої системи безпеки: імплементовано *stateless* автентифікацію на основі JSON Web Tokens (JWT), криптографічне хешування паролів за допомогою алгоритму bcrypt та інтегровано хмарний сервіс Resend API для гарантованої доставки одноразових паролів (OTP) при відновленні доступу. Інтелектуальне ядро системи, що базується на моделях Google Gemini, було підсилене розробленим

інженерним алгоритмом автоматичних повторних запитів з експоненційною затримкою (Exponential Backoff), що гарантує високу відмовостійкість при мережевих збоях.

Практична реалізація клієнтської частини (Frontend) була виконана з використанням реактивної бібліотеки React 18 та збирача Vite. Було розроблено преміальний адаптивний користувацький інтерфейс за допомогою utility-first фреймворку Tailwind CSS та безстильових примітивів доступності Radix UI. Значним інженерним здобутком на клієнті стала розробка архітектурного механізму "Persistent Workspace" (збереження стану робочих просторів), який синхронізує чернетки користувача з localStorage, повністю усуваючи ризик втрати введених даних при закритті popup-вікна браузера. Крім того, систему було розширено власною реалізацією багатомовності (i18n) через React Context та модулем синтезу мовлення (Text-to-Speech).

Для забезпечення ідентичності середовищ розробки та полегшення подальшого розгортання, інфраструктуру проєкту було контейнеризовано за допомогою технології Docker. Повністю налаштовано процеси автоматизованого тестування інструментами Vitest та Supertest, що підтвердило стабільність бізнес-логіки.

Таким чином, розроблений програмний продукт повністю відповідає технічному завданню та сучасним індустріальним стандартам розробки програмного забезпечення. Архітектурна гнучкість PAIT створює надійну базу для подальшого масштабування: інтеграції додаткових великих мовних моделей, розширення функціоналу генерації медіа або впровадження систем монетизації.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Google Developers. (2023). Chrome Extensions Documentation: Manifest V3. URL: https://developer.chrome.com/docs/extensions/mv3/ (Дата звернення: 10.03.2026).
2. React Documentation. (2024). React: A JavaScript library for building user interfaces. URL: https://react.dev/ (Дата звернення: 12.03.2026).
3. Vitejs. (2024). Vite: Next Generation Frontend Tooling. URL: https://vitejs.dev/guide/ (Дата звернення: 15.04.2026).
4. GraphQL Foundation. (2024). GraphQL: A query language for your API. URL: https://graphql.org/learn/ (Дата звернення: 18.03.2026).
5. Apollo GraphQL. (2024). Apollo Server Documentation. URL: https://www.apollographql.com/docs/apollo-server/ (Дата звернення: 19.03.2026).
6. Apollo GraphQL. (2024). Apollo Client Documentation. URL: https://www.apollographql.com/docs/react/ (Дата звернення: 19.03.2026).
7. MongoDB Inc. (2024). The MongoDB Manual. URL: https://www.mongodb.com/docs/manual/ (Дата звернення: 22.03.2026).
8. Mongoose. (2024). Mongoose ODM v8.0.0 Documentation. URL: https://mongoosejs.com/docs/guide.html (Дата звернення: 23.03.2026).
9. Node.js Foundation. (2024). Node.js v20.x Documentation. URL: https://nodejs.org/en/docs/ (Дата звернення: 25.03.2026).
10. Microsoft. (2024). TypeScript Handbook. URL: https://www.typescriptlang.org/docs/handbook/intro.html (Дата звернення: 26.03.2026).
11. Tailwind Labs. (2024). Tailwind CSS Documentation. URL: https://tailwindcss.com/docs/installation (Дата звернення: 28.03.2026).
12. Radix UI. (2024). Radix Primitives: Unstyled, accessible components for building high-quality design systems. URL: https://www.radix-ui.com/docs/primitives/overview/introduction (Дата звернення: 02.04.2026).

13. Internet Engineering Task Force (IETF). (2015). RFC 7519: JSON Web Token (JWT). URL: https://datatracker.ietf.org/doc/html/rfc7519 (Дата звернення: 05.04.2026).

14. OWASP Foundation. (2023). REST Security Cheat Sheet. URL: https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html (Дата звернення: 08.04.2026).

15. Google AI for Developers. (2024). Gemini API Documentation. URL: https://ai.google.dev/docs (Дата звернення: 10.04.2026).

16. Resend. (2024). Resend API Reference: Email for developers. URL: https://resend.com/docs/api-reference/introduction (Дата звернення: 12.04.2026).

17. Docker Inc. (2024). Docker Documentation. URL: https://docs.docker.com/ (Дата звернення: 14.04.2026).

18. MDN Web Docs. (2024). Web Storage API (localStorage). URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API (Дата звернення: 16.04.2026).

19. MDN Web Docs. (2024). Web Speech API (Text-to-Speech). URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Speech_API (Дата звернення: 18.04.2026).

20. W3C. (2023). Web Content Accessibility Guidelines (WCAG) 2.1. URL: https://www.w3.org/TR/WCAG21/ (Дата звернення: 20.04.2026).

21. Vitest. (2024). Vitest: A blazing fast unit test framework powered by Vite. URL: https://vitest.dev/guide/ (Дата звернення: 22.04.2026).

22. Testing Library. (2024). React Testing Library Documentation. URL: https://testing-library.com/docs/react-testing-library/intro/ (Дата звернення: 23.04.2026).

23. Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.

24. Kleppmann, M. (2017). *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media.

25. Render. (2024). Render Cloud Hosting Documentation. URL: https://render.com/docs (Дата звернення: 25.04.2026).

ДОДАТКИ

ДОДАТОК А

Рис. 1.1 (Діаграма прецедентів / Use Case)

Джерело: [створено автором]

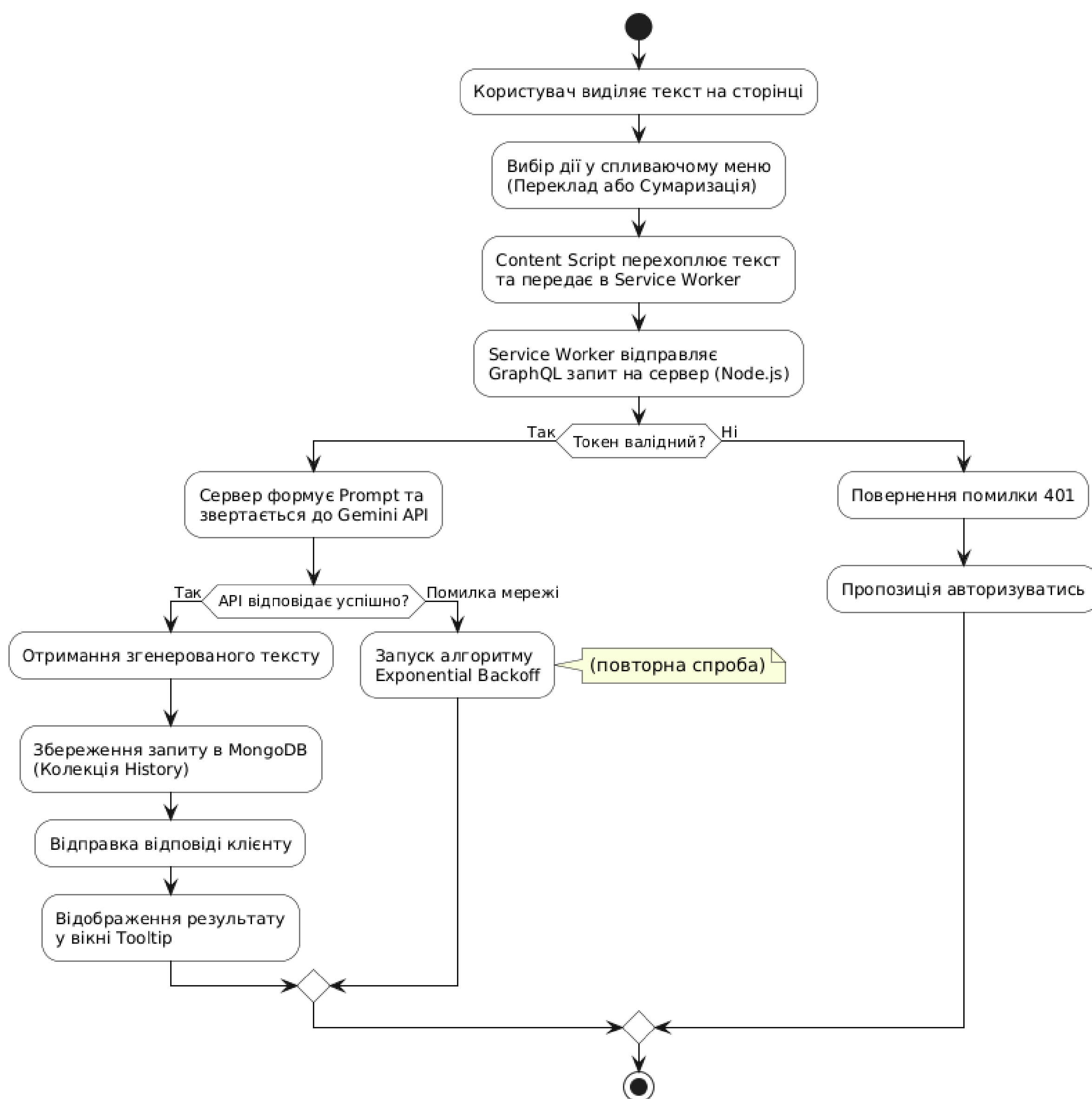

Рис. 1.2 (Блок-схема процесу обробки тексту)

Джерело: [створено автором]

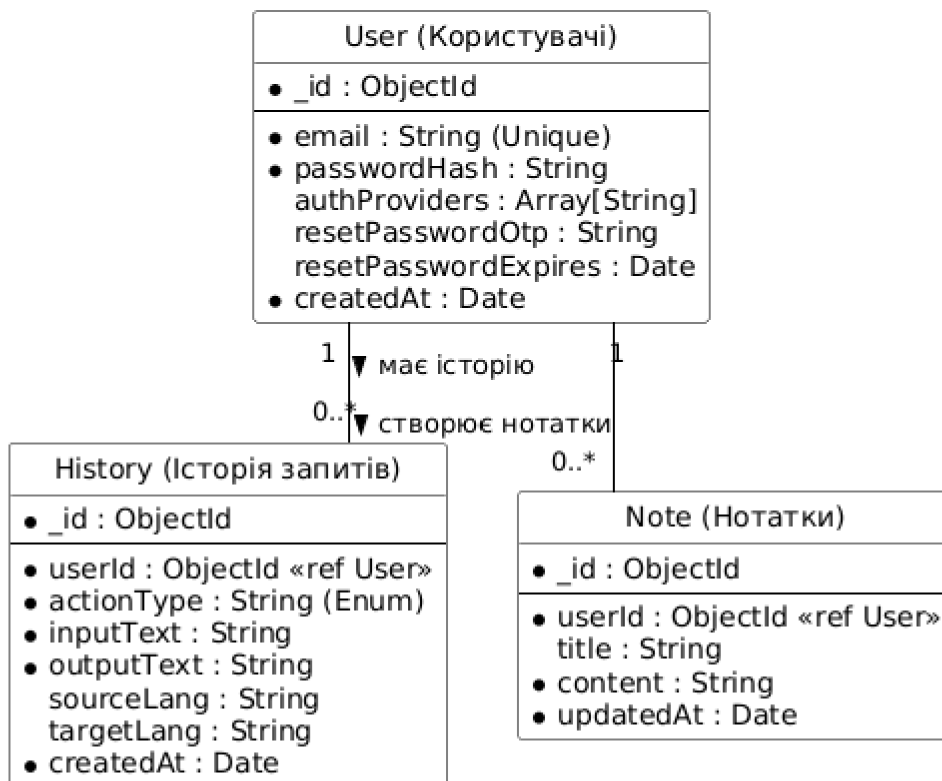

Рис. 2.1 (ER-діаграма бази даних застосунку)

Джерело: [створено автором]

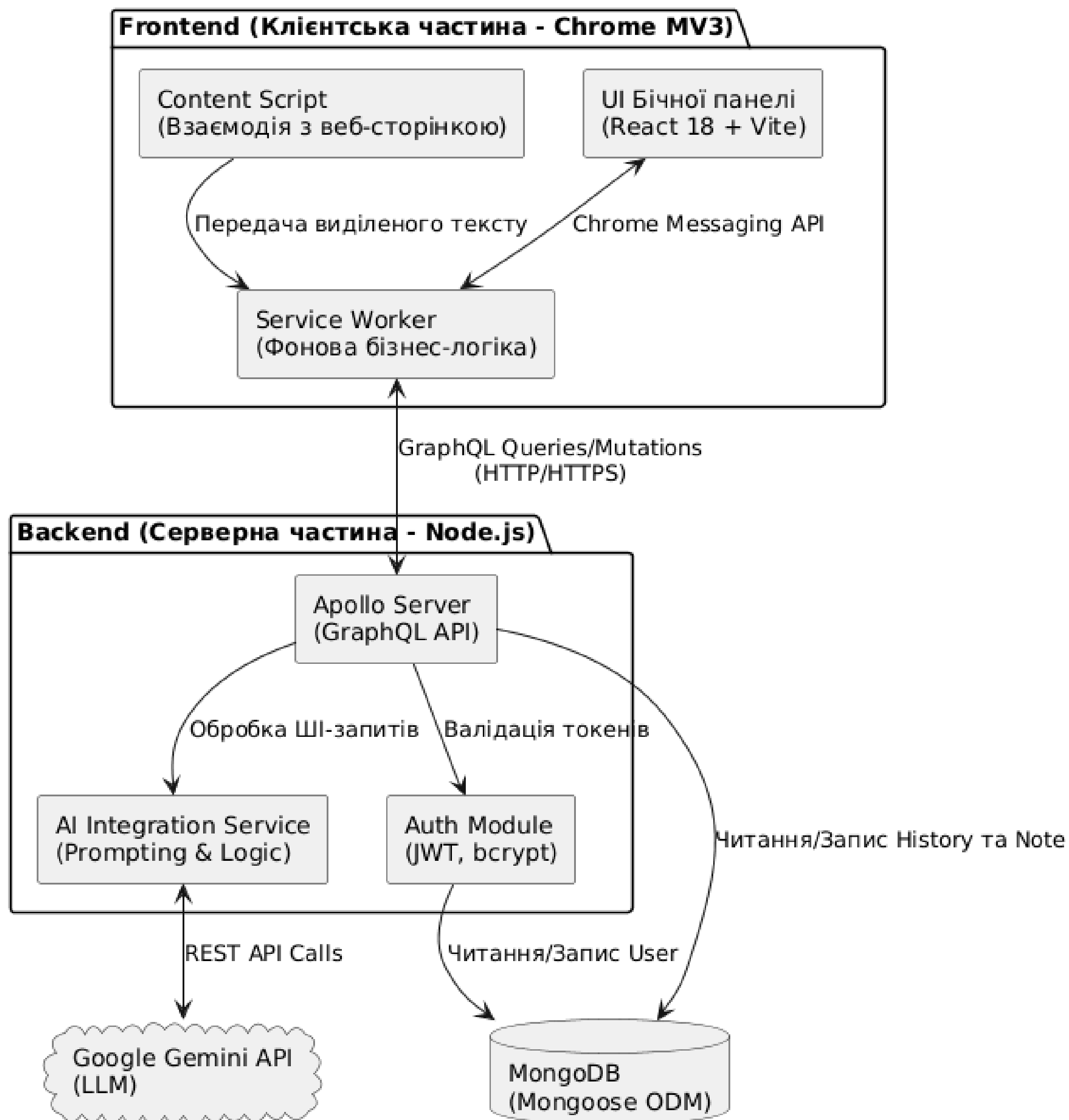

Рис. 2.2 (Загальна архітектурна схема системи)

Джерело: [створено автором]

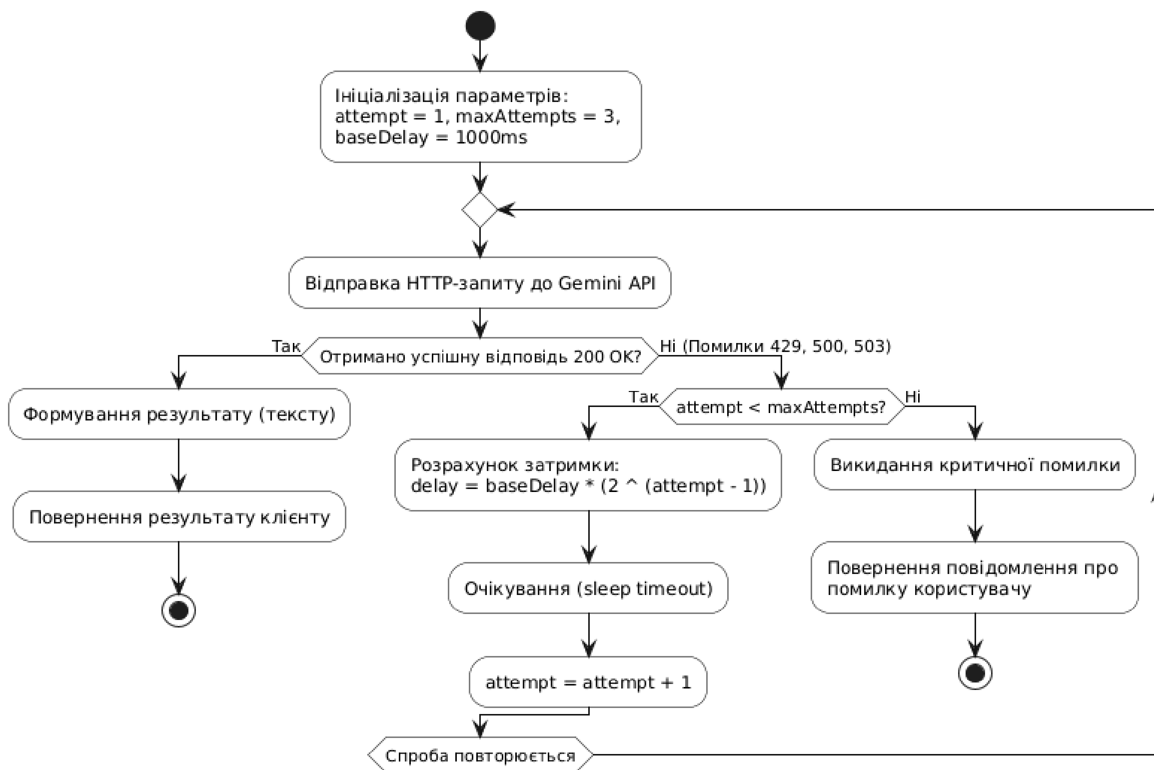

Рис. 2.3 (Блок-схема алгоритму Exponential Backoff)

Джерело: [створено автором]

ДОДАТКИ

ДОДАТОК Б

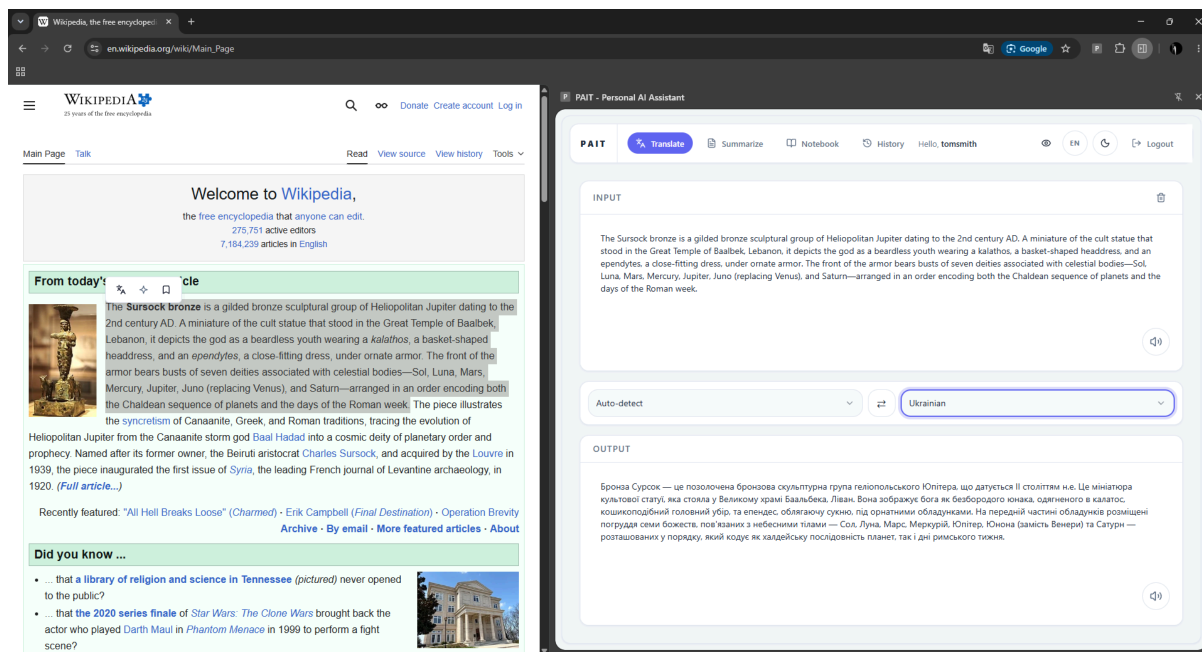

Рис. 3.1 Головний робочий простір інструменту перекладу

Джерело: [створено автором]

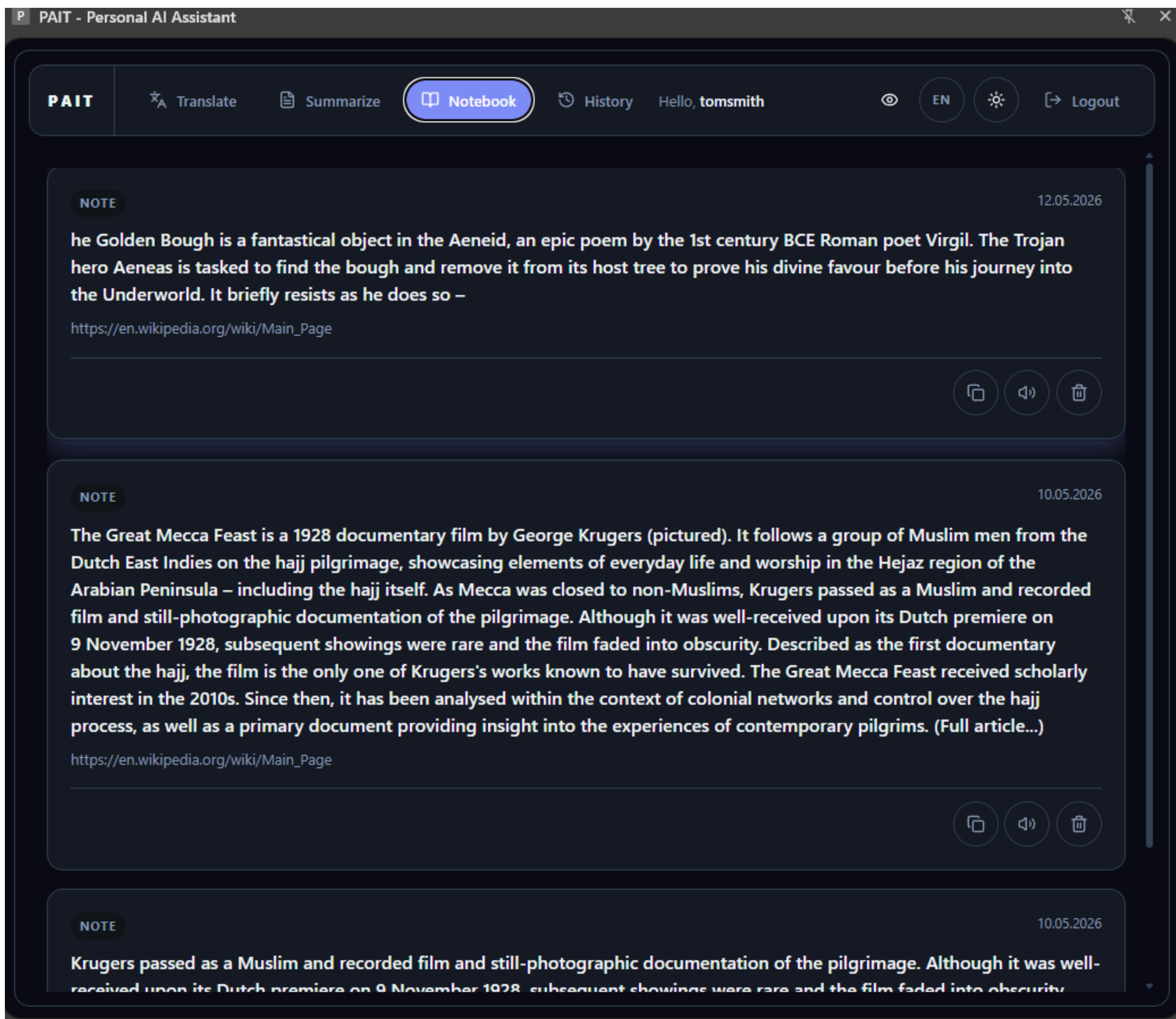

Рис. 3.2 Адаптація інтерфейсу під темну тему

Джерело: [створено автором]

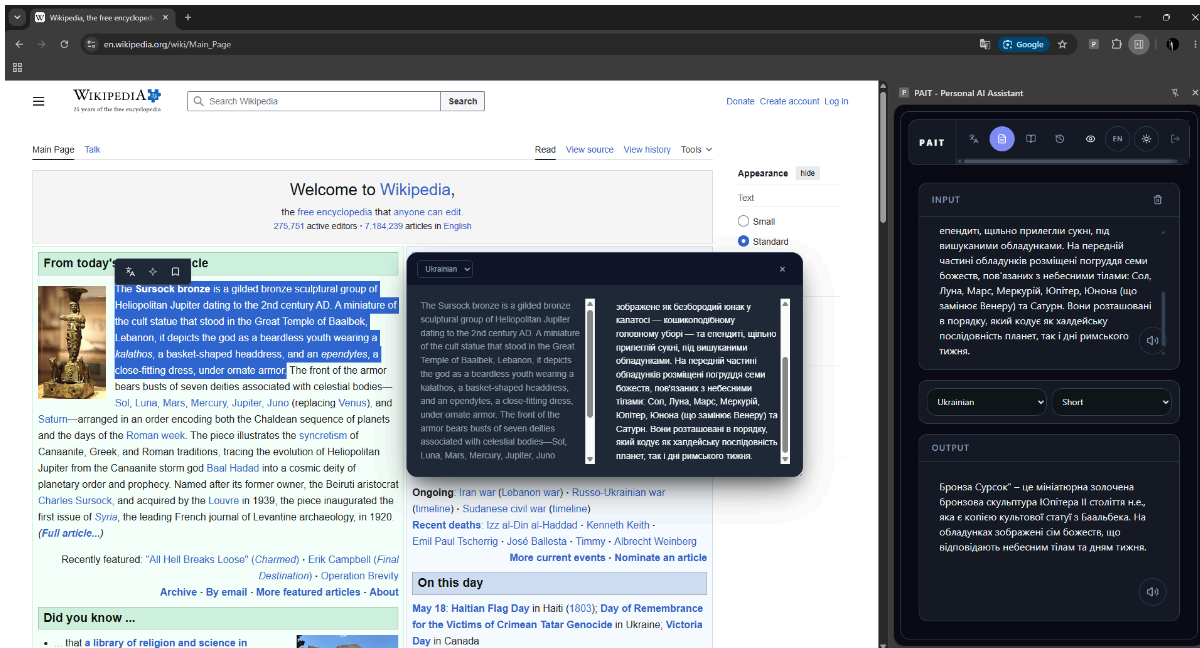

Рис. 3.3 Інтелектуальне спливаюче вікно над виділеним текстом

Джерело: [створено автором]

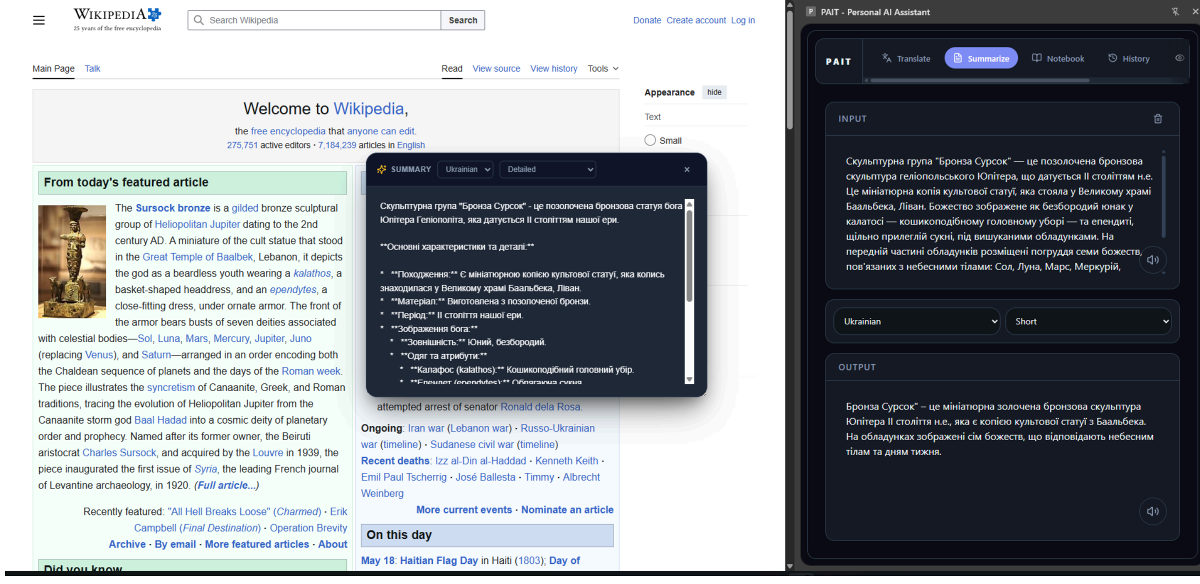

Рис. 3.4 Інтерфейс модуля сумаризації тексту

Джерело: [створено автором]

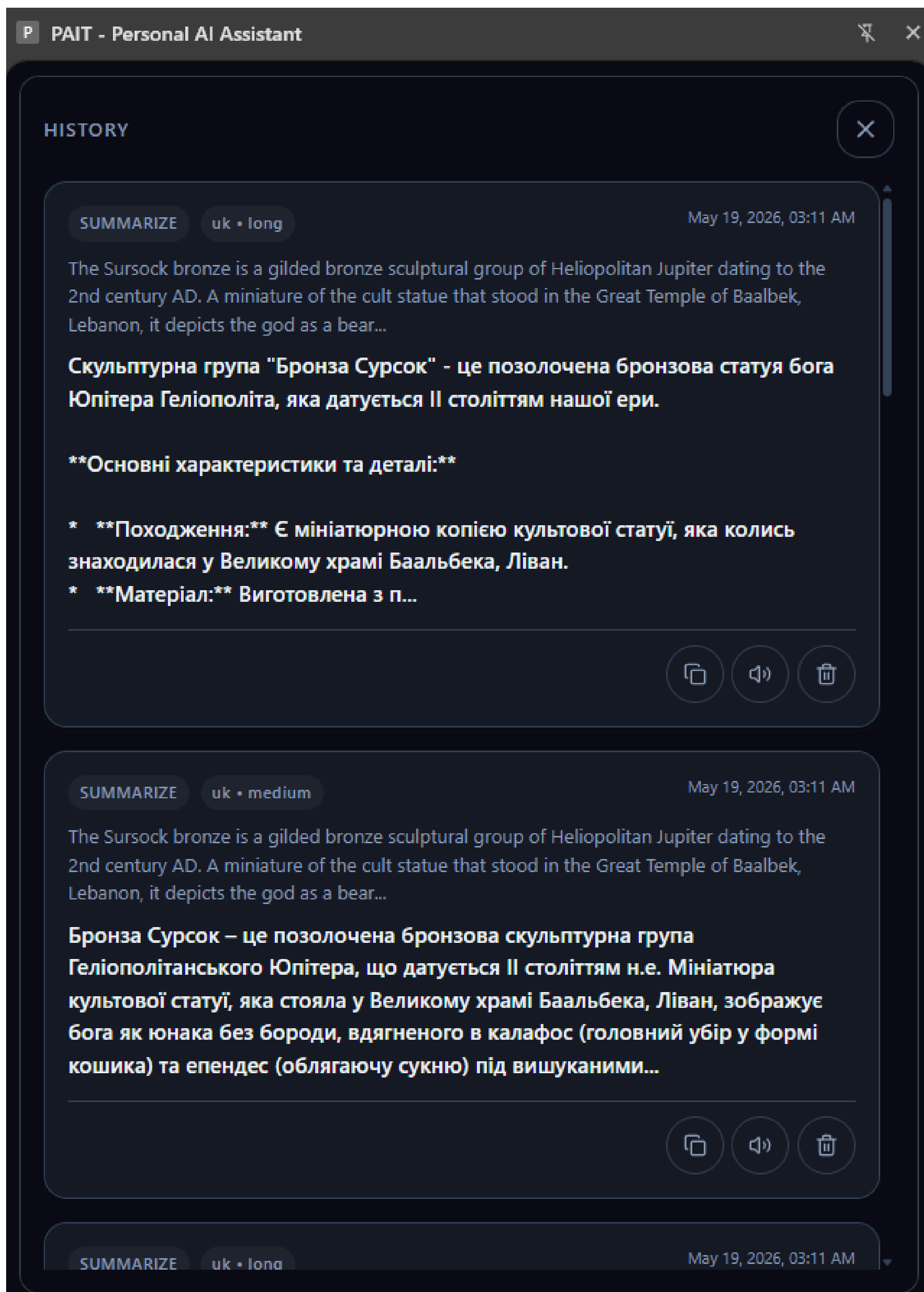

Рис. 3.5 Відображення збереженої історії AI-запитів користувача

Джерело: [створено автором]

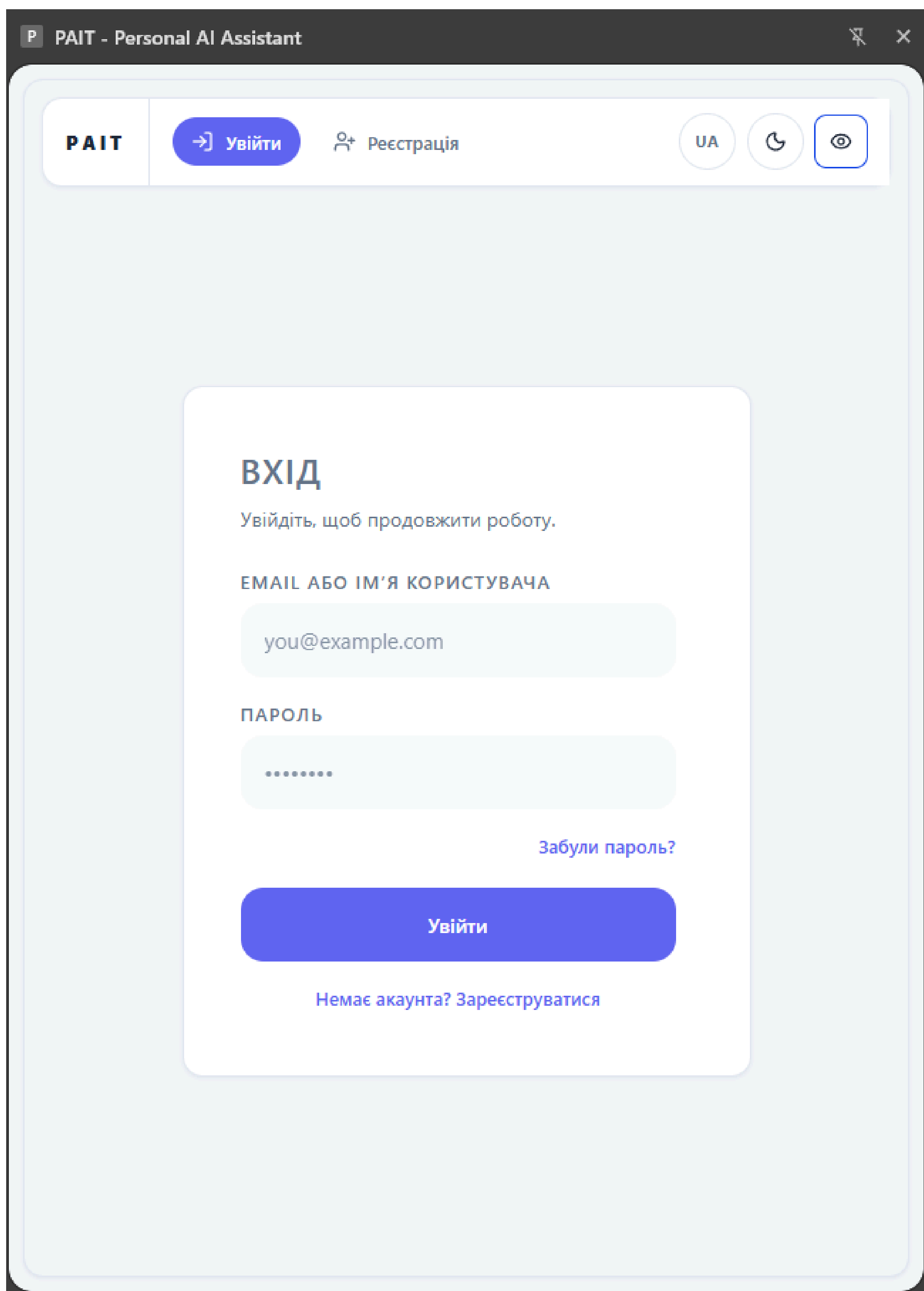

Рис. 3.6 Екран автентифікації та управління профілем

Джерело: [створено автором]