ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка клієнтського застосунку для обробки та узагальнення текстової інформації з web-ресурсів. Спец-частина: розробка Backend-частини застосунку»

на здобуття освітнього ступеня бакалавра

зі спеціальності 121 Інженерія програмного забезпечення

освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Михайло КАРМАННИЙ
(підпис)

Виконав: здобувач вищої освіти групи ПД-42
Михайло КАРМАННИЙ

Керівник: Віталій ЗАЛИВА
доктор філософії (PhD)

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ

ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Карманному Михайлу Юрійовичу

1. Тема кваліфікаційної роботи: «Розробка клієнтського застосунку для обробки та узагальнення текстової інформації з web-ресурсів. Спец-частина: розробка Backend-частини застосунку»
керівник кваліфікаційної роботи доктор філософії (PhD) Віталій ЗАЛИВА,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 року № 51.
2. Строк подання кваліфікаційної роботи «29» травня 2026 року.
3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про сучасні методи розробки високонавантажених клієнт-серверних web-застосунків, принципи автоматизованої обробки та узагальнення текстової інформації за допомогою великих мовних моделей, технічна документація середовища виконання Bun, фреймворку ElysiaJS, об'єктно-реляційного відображення MikroORM та СУБД PostgreSQL.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
   1. Аналіз предметної галузі систем автоматизованої обробки та узагальнення текстової інформації.
   2. Огляд та вибір засобів програмної реалізації серверної частини застосунку.

3. Проєктування архітектури серверної частини застосунку та структури бази даних.
4. Програмна реалізація, тестування та розгортання серверної частини застосунку.

5. Перелік графічного матеріалу: *презентація*
   1. Аналіз аналогів.
   2. Вимоги до програмного забезпечення.
   3. Програмні засоби реалізації.
   4. Діаграма варіантів використання.
   5. Діаграми авторизації та скорочення тексту
   6. Структура бази данних
   7. Архітектура додатку
   8. Продуктивність backend-частини
   9. Екранні форми.
   10. Апробація результатів дослідження.
6. Дата видачі завдання «23» лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	23.02-28.02.2026	
2	Аналіз та дослідження існуючих аналогів	01.03-07.03.2026	
3	Огляд та аналіз засобів реалізації backend частини застосунку	08.03–22.03.2026	
4	Проектування архітектури backend частини застосунку	23.03–04.04.2026	
5	Програмна реалізація та тестування backend частини застосунку	05.04–26.04.2026	
6	Оформлення роботи: вступ, висновки, реферат	27.04–03.05.2026	
7	Розробка демонстраційних матеріалів	04.05–10.05.2026	
8	Попередній захист роботи	11.05-22.05.2026	
9	Подання кваліфікаційної роботи	23.05-30.05.2026	

Здобувач вищої освіти _____ Михайло КАРМАННИЙ

Керівник
кваліфікаційної роботи _____ Віталій ЗАЛИВА

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 56 стор., 1 табл., 31 рис., 24 джерел.

Мета роботи – оптимізація процесу обробки великих обсягів тексту шляхом розробки програмного забезпечення для автоматизованого індексування та узагальнення текстів з web-ресурсів із використанням моделей штучного інтелекту.

Об'єкт дослідження – процес автоматизованої обробки, аналізу та узагальнення текстів в сучасних web-системах.

Предмет дослідження – архітектурні рішення, програмні засоби та технології розробки серверної частини застосунку для взаємодії з клієнтським інтерфейсом, інтеграції ШІ-моделей та забезпечення безпеки бізнес-логіки.

Короткий зміст роботи: В роботі проаналізовано проблему інформаційного перевантаження користувачів. Розглянуто наявні рішення на ринку такі як: Summarizer.org, Smartsummarize.i, Summarize AI, AISMRY). Розроблено багаторівневу архітектуру серверної частини застосунку для швидкої обробки тексту та взаємодії зі штучним інтелектом. Програмно реалізовано ключові модулі: безпечна автентифікація з підтримкою OAuth Google, обробка платежів за допомогою сервісу Paddle, взаємодія з ШІ та оптимізоване кешування запитів. Для забезпечення максимальної швидкодії використано середовище виконання Bun та фреймворк ElysiaJS. Особливу увагу приділено кібербезпеці: впроваджено систему моніторингу аномалій Tirreno, симетричне шифрування персональних даних та алгоритм «сліпих індексів» у базі даних PostgreSQL. Побудовано підсистему Observability для ізольованого збору логів у MongoDB за допомогою Logixlysia.

Сферою використання застосунку є автоматизація процесів швидкого аналізу та підсумування текстового контенту для широкого кола інтернет-користувачів.

КЛЮЧОВІ СЛОВА: УЗАГАЛЬНЕННЯ ТЕКСТУ, ШТУЧНИЙ ІНТЕЛЕКТ, WEB-ЗАСТОСУНОК, ELYSIAJS, BUN, POSTGRESQL, БЕЗПЕКА ДАНИХ, BACKEND-АРХІТЕКТУРА.

ЗМІСТ

ВСТУП

У сучасних умовах стрімкого розвитку цифрових технологій та безперервного зростання обсягів інформації в мережі Інтернет, користувачі все частіше стикаються з явищем «інформаційного перевантаження». Брак часу на повноцінне опрацювання довгих статей, новин чи дослідницьких матеріалів вимагає створення нових, ефективних інструментів для швидкого аналізу контенту. Актуальність дослідження зумовлена необхідністю розробки програмних рішень, що здатні інтегруватися безпосередньо у браузерне середовище та надавати користувачеві стислий зміст web-ресурсів у реальному часі за допомогою генеративного штучного інтелекту.

Проблематика, що стала основою для вибору теми дослідження, полягає в обмеженості існуючих на ринку систем сумаризації тексту. Більшість аналогів функціонують виключно як ізольовані web-сайти, що змушує користувача постійно копіювати текст, або ж штучно обмежують швидкість обробки запитів. Окрім цього, інтеграція AI-моделей вимагає побудови стійкої до навантажень та захищеної серверної інфраструктури, яка б гарантувала конфіденційність користувацьких даних та оптимізацію витрат на використання сторонніх API.

Мета роботи: оптимізація процесу опрацювання великих обсягів контенту за рахунок розробки програмного забезпечення для автоматизованого захоплення та інтелектуального узагальнення текстової інформації з web-ресурсів із використанням моделей штучного інтелекту.

Об’єкт дослідження: процес автоматизованої обробки, аналізу та узагальнення текстової інформації в сучасних інформаційних web-системах.

Предмет дослідження: архітектурні рішення, програмні засоби та технології розробки серверної частини застосунку для взаємодії з клієнтським інтерфейсом, інтеграції AI-моделей та забезпечення безпеки бізнес-логіки.

Результати цієї роботи можуть бути впроваджені як основа для розгортання масштабованих SaaS-продуктів, орієнтованих на роботу з AI-технологіями. В основі Backend-розробки покладено інноваційний технологічний стек: середовище виконання Bun та фреймворк ElysiaJS для мінімізації мережевих затримок, PostgreSQL (через MikroORM) для транзакційного збереження даних, а також інструменти екосистеми безпеки та моніторингу – Lucia Auth, Tirreno, Logixlysia та MongoDB. Інфраструктура проекту спроектована за принципом Self-hosting з використанням оркестратора Coolify та менеджера секретів Infisical. Обраний технологічний підхід дозволяє створити масштабовану, безпечну та економічно вигідну систему.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

У сучасних умовах стрімкого розвитку інформаційних технологій спостерігається експоненційне зростання обсягів цифрових даних. Щодня генерується значна кількість статей, новинних зведень, наукових публікацій, технічної документації та аналітичних звітів. Як наслідок, виникає проблема інформаційного перевантаження, оскільки когнітивні можливості людини не дозволяють ефективно опрацьовувати такі масиви інформації [1]. У цьому контексті завдання автоматизованої обробки та узагальнення текстових даних із використанням методів обробки природної мови (NLP) та великих мовних моделей (LLM) набуває виняткової актуальності.

Дане дослідження зосереджено на аналізі процесів взаємодії користувачів із великими обсягами текстової інформації у web-середовищі, а також на проектуванні backend-інфраструктури та архітектурних рішень, що забезпечують швидке й безпечне узагальнення контенту.

1.1 Особливості задачі та цільова аудиторія

Головна ідея QuickSum - дати людям можливість швидко 'схопити' суть будь-якого тексту з будь-якої web-сторінки та отримання його стислого, структурованого змісту (головних тез, ключових висновків) зберігаючи основний зміст.

Оскільки задача має загальний характер, цільова аудиторія продукту є досить широкою і включає:

Здобувачів освіти та науковців: потребують швидкого опрацювання великої кількості літературних джерел, статей та документації для написання дослідницьких робіт.

Аналітиків, маркетологів та офісних працівників: щоденно стикаються з

необхідністю аналізу об'ємних звітів, технічних завдань та новинних стрічок.

Звичайних інтернет-користувачів: бажають економити час при читанні лонгрідів або публікацій у медіа.

Користувачі досить вимогливі до таких інструментів, і ось що їм потрібно:

– мінімальна затримка: процес генерації результату має відбуватися у реальному часі. Тривале очікування відповіді від сервера псує враження від користування;

– зручність: інструмент має бути інтегрований у звичне середовище (наприклад, у вигляді браузерного розширення), щоб уникнути постійного перемикання контексту між вкладками;

– конфіденційність: гарантія того, що оброблювані тексти та персональні дані користувача не будуть скомпрометовані або використані зловмисниками.

1.2 Аналіз існуючих рішень та їх недоліків

На поточному етапі розвитку IT-ринку задача узагальнення тексту здебільшого вирішується за допомогою використання SaaS-платформ та різноманітних web-застосунків. Аналіз наукових публікацій та ринкових пропозицій дозволяє виділити загальні підходи до побудови таких систем [2]. Більшість із них працюють як проміжна ланка між користувачем та комерційними API штучного інтелекту (наприклад, OpenAI).

Незважаючи на наявність готових рішень, більшість з них мають суттєві архітектурні та функціональні недоліки:

Висока мережева затримка та неоптимізована продуктивність: Проблема в тому, що більшість існуючих рішень працюють на застарілих, повільних технологіях (наприклад, стандартному Node.js або Python/Django). Враховуючи, що сама генерація тексту LLM-моделлю займає час, додаткові затримки на рівні серверного роутингу роблять використання таких сервісів не комфортним.

Штучні обмеження та перевитрата ресурсів: Існуючі SaaS-рішення часто використовують асинхронні запити до ШІ для кожного клієнта окремо. Якщо

тисяча користувачів одночасно запросять узагальнення однієї і тієї ж популярної новини, традиційний сервер відправить тисячу платних запитів до OpenAI. Це призводить до колосальної перевитрати токенів, через що власники сервісів змушені вводити жорсткі ліміти (Rate Limits) для безкоштовних користувачів.

Низький рівень захисту персональних даних (PII): В умовах суворих міжнародних регламентів (наприклад, GDPR) зберігання електронних адрес та імен користувачів у відкритому вигляді у базах даних є неприпустимим. Більшість існуючих розширень не використовують криптографічне шифрування на рівні ORM, що робить їх вразливими до витоків даних (Data Breaches).

Відсутність механізмів захисту бізнес-логіки: Відкриті API браузерних розширень часто стають мішенню для автоматизованих скриптів та ботів, які намагаються безкоштовно використовувати платні ресурси ШІ через вразливості серверної частини.

У процесі проєктування програмного забезпечення було проведено аналіз сучасних рішень, які мають подібний функціонал. До основних аналогів можна віднести такі платформи, як Summarizer.org, Smart Summarize, Summarize AI та AISMRY. Розглянемо їх детальніше.

1.2.1 Summarizer.org

Summarizer.org[3] – це спеціалізований безкоштовний онлайн-сервіс, призначений для автоматичного створення стислих викладів (саммарі) великих обсягів тексту. Цей інструмент широко використовується студентами, дослідниками, копірайтерами та контент-мейкерами для швидкого виділення ключової інформації з довгих статей, наукових документів або есе. Він дозволяє користувачеві самостійно налаштовувати бажану довжину кінцевого результату, підтримує багатомовність (зокрема дозволяє працювати з текстами різними мовами) та використовує алгоритми штучного інтелекту для збереження початкового контексту і змісту без втрати логіки викладу. Система має зручний, мінімалістичний та інтуїтивно зрозумілий web-інтерфейс, який не вимагає обов'язкової реєстрації або встановлення додаткового програмного забезпечення

на комп'ютер користувача. Приклад інтерфейсу веб-ресурсу наведено на рисунку 1.1.

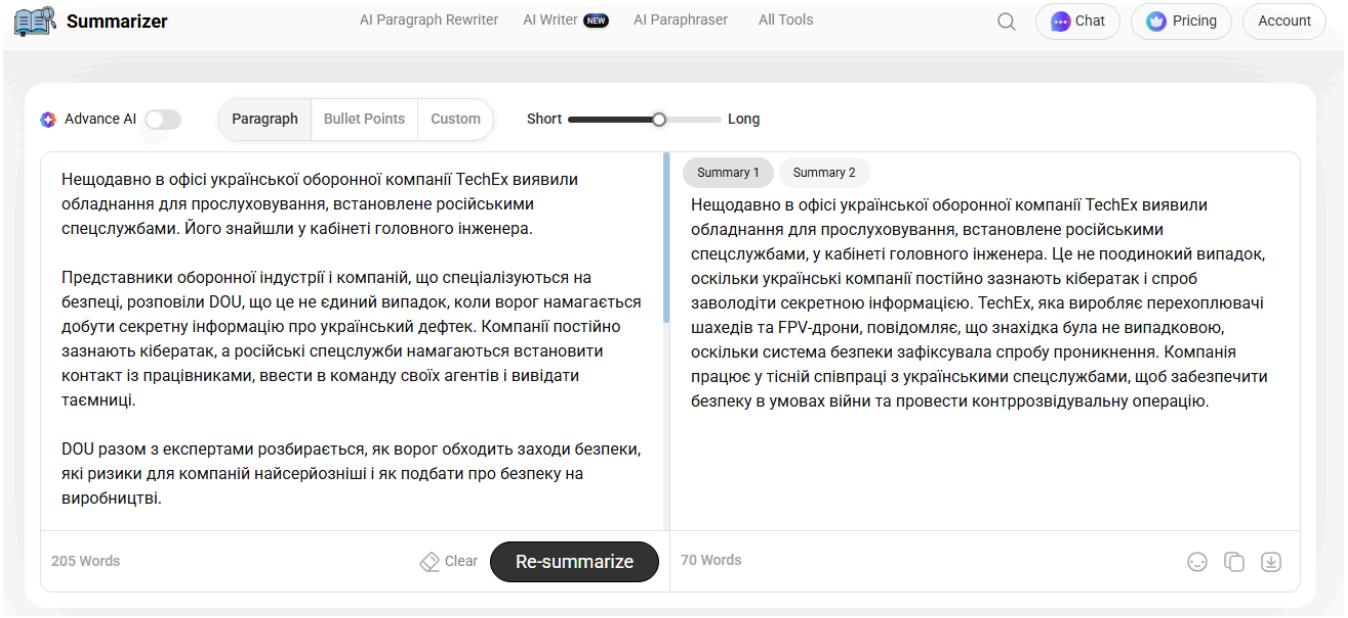


Рис. 1.1 Інтерфейс сервісу Summarizer.org

Переваги використання Summarizer.org:

– гарна оперативність обробки даних (середній час очікування відповіді становить 5 секунд);

– наявність декількох режимів узагальнення тексту (Довге, Коротке);

– відносно зрозумілий web-інтерфейс.

До недоліків сервісу можна віднести наступне:

– відсутність браузерного розширення, що змушує користувача постійно копіювати текст і переходити на іншу вкладку;

– відсутність гарантій безпеки персональних даних.

1.2.2 Smart Summarize

Smart Summarize[4] – це сучасна платформа на базі штучного інтелекту, що надає розширені можливості для глибокого аналізу та розумного скорочення текстових матеріалів. Вона орієнтована переважно на професіоналів, аналітиків та маркетологів, які щодня працюють з великими масивами текстових даних,

дозволяючи значно оптимізувати робочий час завдяки миттєвому генеруванню точних резюме. Сервіс пропонує функції не лише звичайного механічного скорочення, але й інтелектуального виділення головних тез, ключових слів, а також структурування розрізненої інформації у зручні списки. Платформа забезпечує високий рівень точності обробки природної мови (NLP) і має сучасний адаптивний web-інтерфейс, який однаково зручно використовувати як на десктопних комп'ютерах, так і на мобільних пристроях. Приклад головної сторінки платформи наведено на рисунку 1.2.

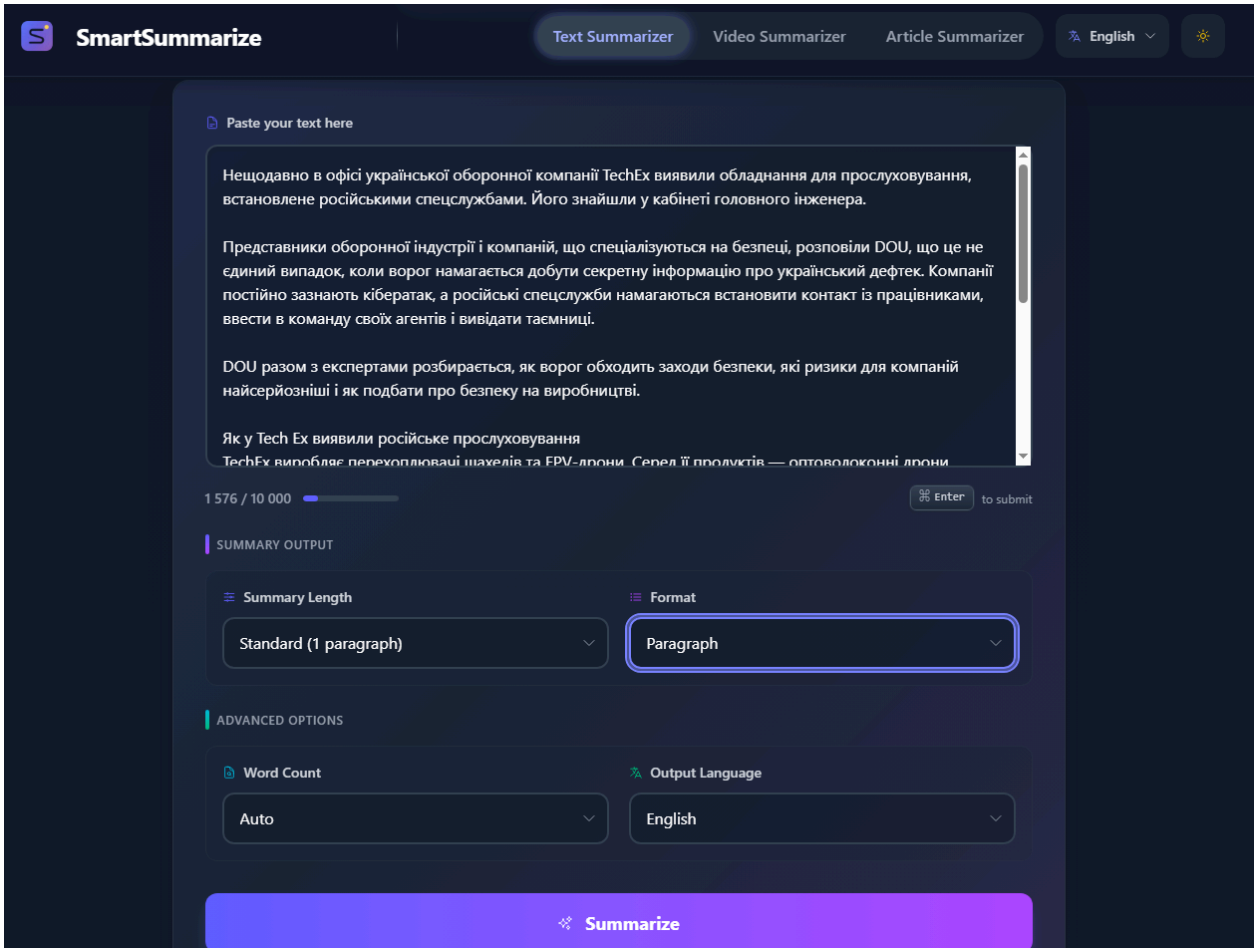


Рис. 1.2 Інтерфейс платформи Smart Summarize

Переваги сервісу Smart Summarize:

– наявність декількох режимів узагальнення тексту (Довге, Коротке);

До недоліків Smartsummarize.io відносять:

– наявність лише одного стандартизованого режиму узагальнення;

– відсутність браузерного розширення;

– робота з персональними даними не відповідає сучасним стандартам безпеки.

– критично низька оперативність обробки даних порівняно з попереднім аналогом (>40 секунд).

1.2.3 Summarize AI

Summarize AI[5] – це спеціалізоване та компактне розширення для веб-браузера Google Chrome, яке дозволяє користувачам отримувати короткий зміст веб-сторінок або конкретного виділеного тексту безпосередньо під час серфінгу в інтернеті. На відміну від окремих веб-сайтів, які вимагають копіювання та вставки тексту, цей інструмент безшовно інтегрується в браузер і активується в один клік, що значно підвищує ергономіку та швидкість роботи користувача. Він ефективно працює з довгими новинними статтями, блогами, технічними мануалами та науковими публікаціями, використовуючи потужні мовні моделі під капотом для генерації змістовних вичавок без втрати критично важливих деталей та фактів. Інтерфейс розширення реалізовано у вигляді зручного спливаючого вікна (pop-up) з інструментами швидкого копіювання результату.

Приклад інтерфейсу розширення наведено на рисунку 1.3.

Переваги додатку Summarize AI:

– наявність браузерного розширення для швидкого доступу до функціоналу.

До суттєвих недоліків Summarize AI можна віднести:

– критично низька оперативність обробки даних (середній час затримки перевищує 1хв), що свідчить про використання неоптимізованої серверної архітектури;

– наявність лише одного базового режиму узагальнення тексту;

– відсутність шифрування персональних даних користувачів.

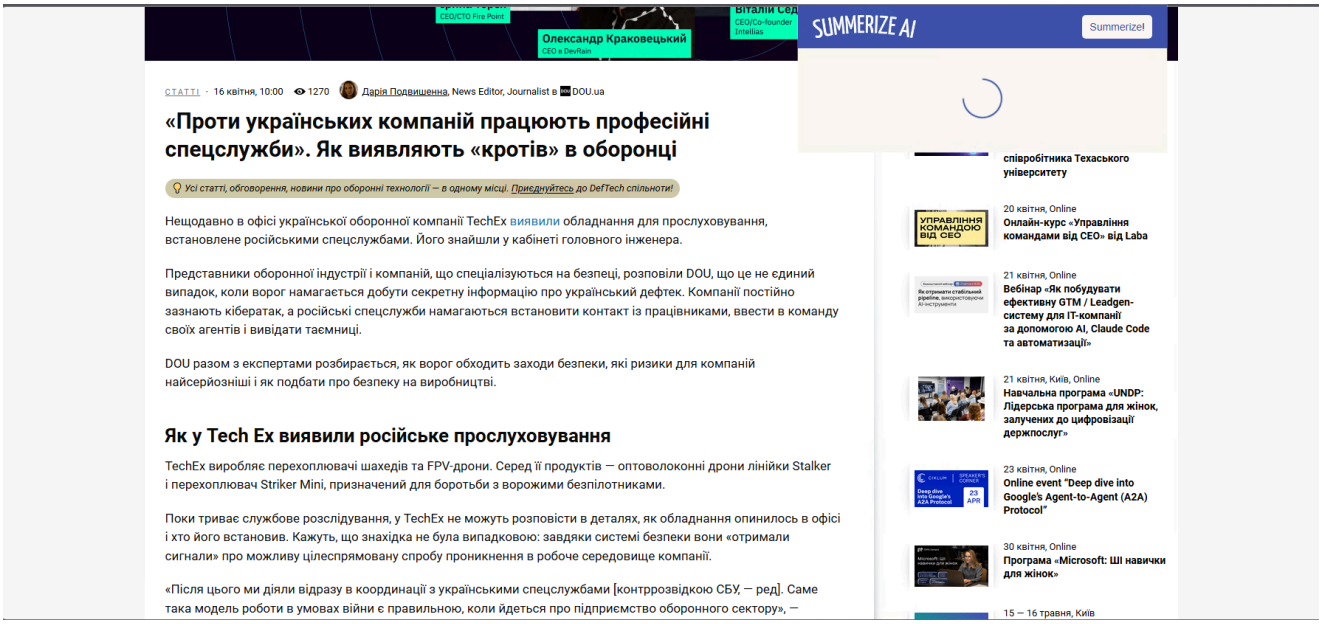


Рис. 1.3 Інтерфейс розширення Summarize AI

1.2.4 AISMRY

AISMRY[6] – це розширення для веб-браузера Google Chrome для автоматичного реферування та перефразування тексту, який використовує передові алгоритми машинного навчання для глибокого семантичного аналізу та синтезу інформації. Система розроблена для забезпечення максимальної гнучкості: вона дозволяє обробляти тексти надзвичайно великого об'єму та пропонує кілька режимів роботи (від простого екстрактивного скорочення, де виділяються існуючі речення, до абстрактивного, де нейромережа переписує текст своїми словами для кращої читабельності). Сервіс орієнтований на академічну та корпоративну сфери, де вимагається висока точність збереження смислового навантаження вихідного документа та відсутність плагіату. Додатково платформа пропонує інструменти для швидкого експорту отриманих результатів.

Приклад робочого середовища сервісу наведено на рисунку 1.4.

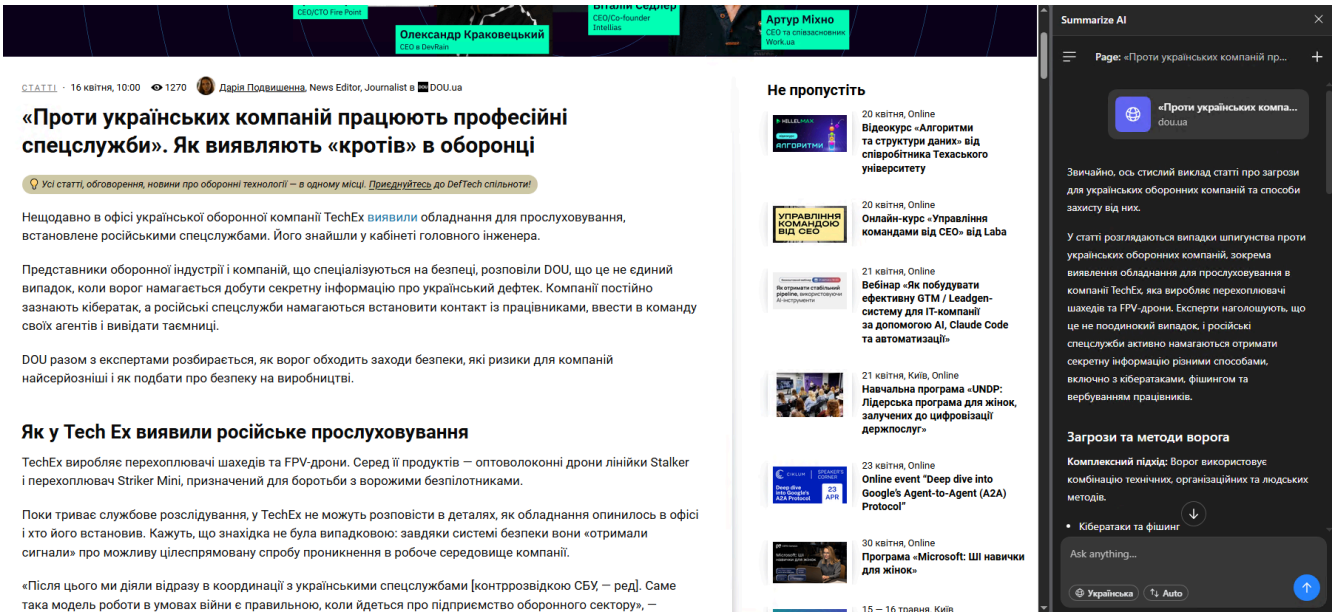


Рис. 1.4 Приклад інтерфейсу AISMRY

Переваги додатку AISMRY:

– наявність браузерного розширення для швидкого доступу до функціоналу.

Недоліками AISMRY є:

– відсутність окремого сайту з функціоналом розширення;
– тривалий час обробки запитів (±17 секунд);
– відсутність підсумувати лише обраний текст;
– відсутність захисту конфіденційних даних клієнтів.

Зведені результати аналізу характеристик розглянутих додатків наведено у таблиці 1.1.

Таблиця 1.1

Зведені результати аналізу характеристик платформ для скорочення тексту

Характеристика	Summarizer.org	Smartsummarize.io	Summarize AI	AISMRY	**QuickSum (Запропоноване рішення)**
Наявність браузерного розширення	Немає	Немає	Є	Є	**Є**
Режими узагальнення тексту	Довге, Коротке	Довге, Середнє, Коротке	Лише одне	Лише одне	**Середнє, Коротке**
Узагальнення всієї сторінки	Ні	Ні	Так	Так	**Так**
Узагальнення виділеного тексту	Так	Так	Ні	Ні	**Так**
Оперативність обробки даних	±5сек	>40сек	±1хв	±8сек	**<7сек**
Тип монетизації	Freemium	Freemium	Premium	Free	**Free/ Subscription**
Безпека персональних даних	Ні	Ні	Ні	Часткова	**Повне (за стандартом GDPR)**

1.3 Шляхи вирішення проблеми на рівні Backend-архітектури

Всі ці проблеми і стали основою для моєї дипломної роботи. Оскільки фокусом даного дослідження є розробка оптимізованої Backend-частини, вирішення задачі узагальнення тексту буде реалізовано шляхом впровадження сучасних інженерних практик:

По-перше, для подолання проблеми високої затримки (latency) класичні середовища виконання замінюються на інноваційний runtime Bun у поєднанні з високопродуктивним фреймворком ElysiaJS. Згідно з останніми дослідженнями, такий стек дозволяє обробляти HTTP-запити в декілька разів швидше за

традиційні аналоги.

По-друге, проблема перевитрати ресурсів та штучних обмежень швидкості вирішується шляхом впровадження механізму дедуплікації та кешування на рівні реляційної бази даних PostgreSQL. Перед кожним зверненням до ШІ система генеруватиме криптографічний хеш цільового тексту. У разі виявлення збігу в БД, користувач миттєво отримуватиме раніше згенеровану відповідь, що кардинально знижує навантаження на API та фінансові витрати компанії.

По-третє, для забезпечення максимального рівня кібербезпеки ми закладаємо безпеку в саму основу системи. Усі персональні дані (PII) шифруються симетричним алгоритмом безпосередньо перед записом у PostgreSQL. Для можливості ідентифікації користувачів без розшифрування бази застосовується метод «сліпих індексів» (Blind Indexes). Додатково, для захисту від фроду та зловживань бізнес-логікою, на серверний рівень інтегрується система моніторингу Tirreno.

Таким чином, розробка спеціалізованого Backend-рішення з фокусом на продуктивність, економію ресурсів (кешування) та глибокий рівень безпеки дозволить створити конкурентоспроможний продукт, який ефективно вирішує проблему інформаційного перевантаження для кінцевого користувача.

2 ЗАСОБИ ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

2.1 Середовище розробки

2.1.1 Visual Studio Code

Visual Studio Code[7] – це легке, але потужне інтегроване середовище розробки (IDE) з відкритим вихідним кодом, створене компанією Microsoft. Воно підтримує роботу з операційними системами Windows, macOS та Linux і має багату екосистему розширень, що дозволяє адаптувати редактор під будь-яку мову програмування та технологічний стек. VS Code надає розробникам такі вбудовані можливості, як підсвічування синтаксису, інтелектуальне автодоповнення коду (IntelliSense), потужні інструменти для рефакторингу, вбудований термінал та інтеграцію із системою контролю версій Git. Завдяки модульній архітектурі, продуктивність середовища залишається високою навіть при роботі з великими проєктами.

У межах даної роботи Visual Studio Code використовувався як основний редактор для написання програмного коду. Велика кількість встановлених розширень (таких як ESLint, Prettier та специфічні для обраного фреймворку плагіни) забезпечила автоматичне форматування коду та виявлення синтаксичних помилок у режимі реального часу, що значно підвищило якість кодової бази. Вбудований термінал дозволяв зручно запускати локальний сервер, виконувати скрипти збирання та керувати залежностями без необхідності перемикатися між різними вікнами. Крім того, інструменти відлагодження (дебаггінгу) безпосередньо в IDE допомогли ефективно знаходити та усувати логічні помилки під час розробки. Приклад інтерфейсу середовища розробки VS Code наведено на рисунку 2.1.

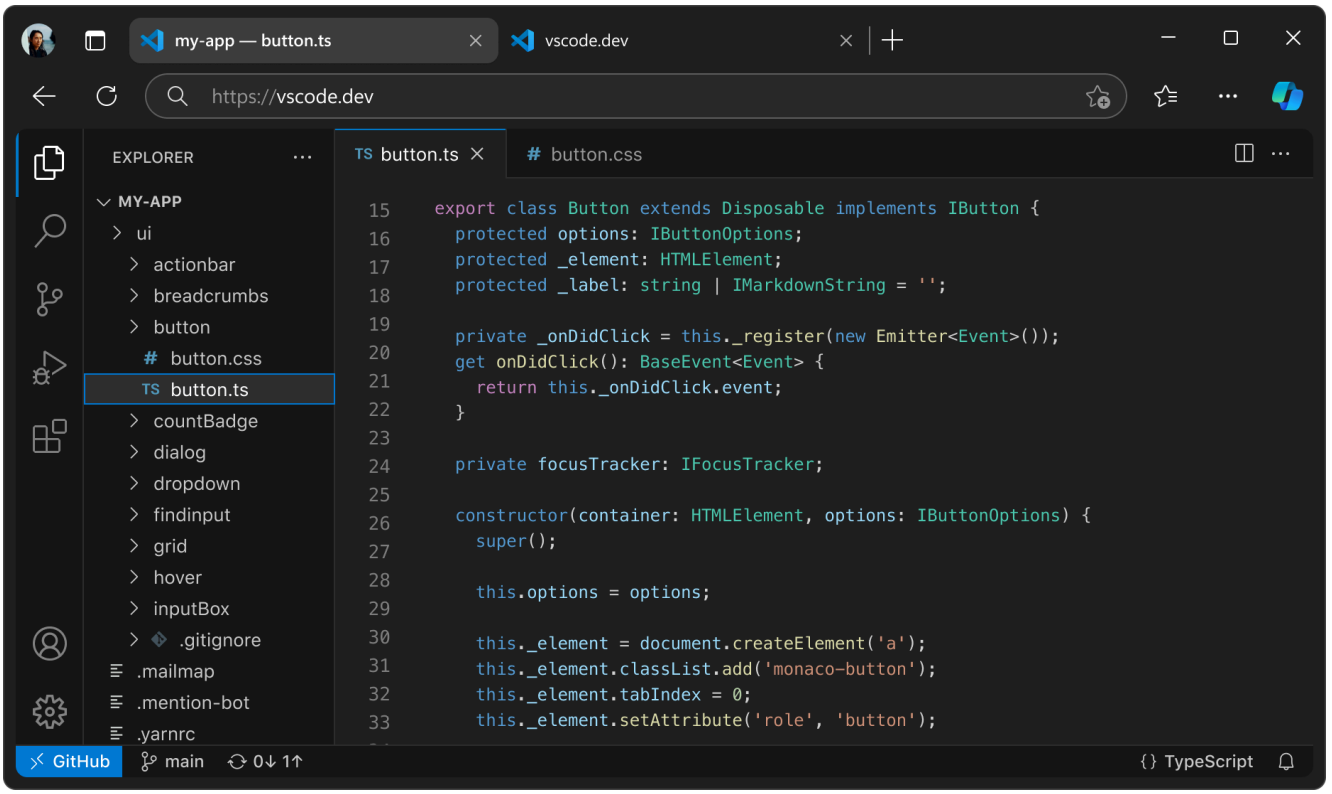


Рис. 2.1 Інтерфейс середовища розробки Visual Studio Code

2.1.2 DbGate

DbGate[8] – це сучасний кросплатформний клієнт для управління базами даних з відкритим вихідним кодом. Інструмент підтримує роботу як з реляційними (PostgreSQL, MySQL, SQLite, SQL Server), так і з NoSQL базами даних (наприклад, MongoDB). DbGate поєднує в собі потужність настільного додатка та гнучкість веб-інтерфейсу, пропонуючи зручний редактор SQL-запитів, інструменти для візуального проєктування схем, засоби імпорту та експорту даних, а також можливість порівняння структури різних баз даних. Його мінімалістичний дизайн і висока швидкість роботи роблять його ефективним інструментом для адміністрування баз даних.

У цьому проєкті DbGate відігравав роль головного інструменту для взаємодії з базою даних. З його допомогою здійснювалося проєктування архітектури даних, створення нових таблиць, налаштування зв'язків між ними та візуальний контроль збереженої інформації під час тестування функціоналу додатку. Інтегрований редактор запитів дозволяв швидко писати, тестувати та

оптимізувати складні SQL-запити перед їх перенесенням у програмний код бекенду. Можливість швидко переглядати та редагувати записи в табличному вигляді суттєво зекономила час під час наповнення бази тестовими даними. Приклад інтерфейсу клієнта баз даних DbGate наведено на рисунку 2.2.

Рис. 2.2 Інтерфейс системи управління базами даних DbGate

2.1.3 HTTPie

HTTPie[9] – це сучасний HTTP-клієнт, розроблений для тестування, налагодження та взаємодії з веб-сервісами та RESTful API. Спочатку створений як потужна альтернатива консольній утиліті cURL, сьогодні HTTPie пропонує також зручний графічний інтерфейс (Desktop App). Інструмент відрізняється інтуїтивно зрозумілим синтаксисом, автоматичним форматуванням та підсвічуванням відповідей у форматі JSON, підтримкою постійних сесій, управлінням заголовками та зручною системою авторизації. Це дозволяє розробникам максимально просто та наочно конструювати HTTP-запити будь-якої складності.

У ході виконання роботи HTTPie використовувався для тестування розробленого програмного інтерфейсу застосунку (API). Завдяки графічному

інтерфейсу програми було зручно формувати різноманітні запити (GET, POST, PUT, DELETE, PATCH), передавати тестові параметри та корисне навантаження (payload) на серверну частину додатку. Це дозволило перевіряти коректність роботи бекенду, обробку помилок та статус-коди відповідей ізольовано від клієнтської частини (фронтенду). Можливість зберігати історію запитів та створювати колекції значно спростила процес системного тестування після внесення змін у код. Приклад інтерфейсу програми HTTPie наведено
на рисунку 2.3.

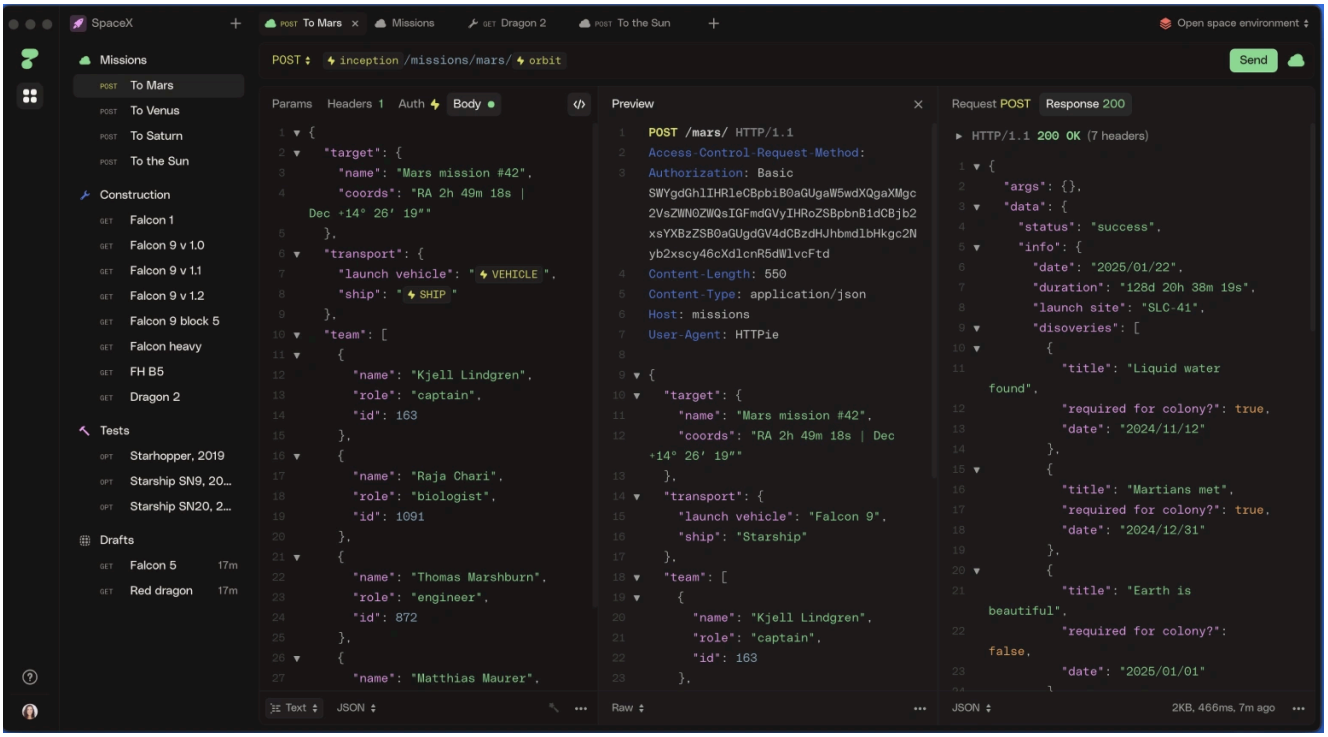


Рис. 2.3 Інтерфейс програми для тестування API HTTPie

2.1.4 OrbStack

OrbStack[10] – це швидкий, легкий і ефективний інструмент для роботи з контейнерами Docker та віртуальними машинами Linux в операційній системі macOS. Він позиціонується як високопродуктивна альтернатива традиційному Docker Desktop. Головними перевагами OrbStack є миттєвий запуск, мінімальне споживання ресурсів центрального процесора та оперативної пам'яті, безперешкодна інтеграція мережі (IPv6, локальні домени) та прямий доступ до

файлової системи контейнерів. Завдяки нативній архітектурі, інструмент не перевантажує систему розробника, забезпечуючи плавну роботу навіть при одночасному запуску великої кількості сервісів.

У межах даного проєкту OrbStack застосовувався для розгортання та керування локальним середовищем розробки за допомогою технології контейнеризації. Використання цього інструменту дозволило ізолювати всі залежності проєкту (наприклад, сервер бази даних, брокери повідомлень тощо) від основної операційної системи. Завдяки OrbStack було забезпечено повну ідентичність середовища розробки з робочим (production) середовищем, що мінімізувало ризик виникнення помилок конфігурації під час розгортання застосунку. Його низьке споживання ресурсів дозволило комфортно працювати з IDE, клієнтами БД та браузером без втрати продуктивності комп'ютера. Приклад інтерфейсу додатку OrbStack наведено на рисунку 2.4.

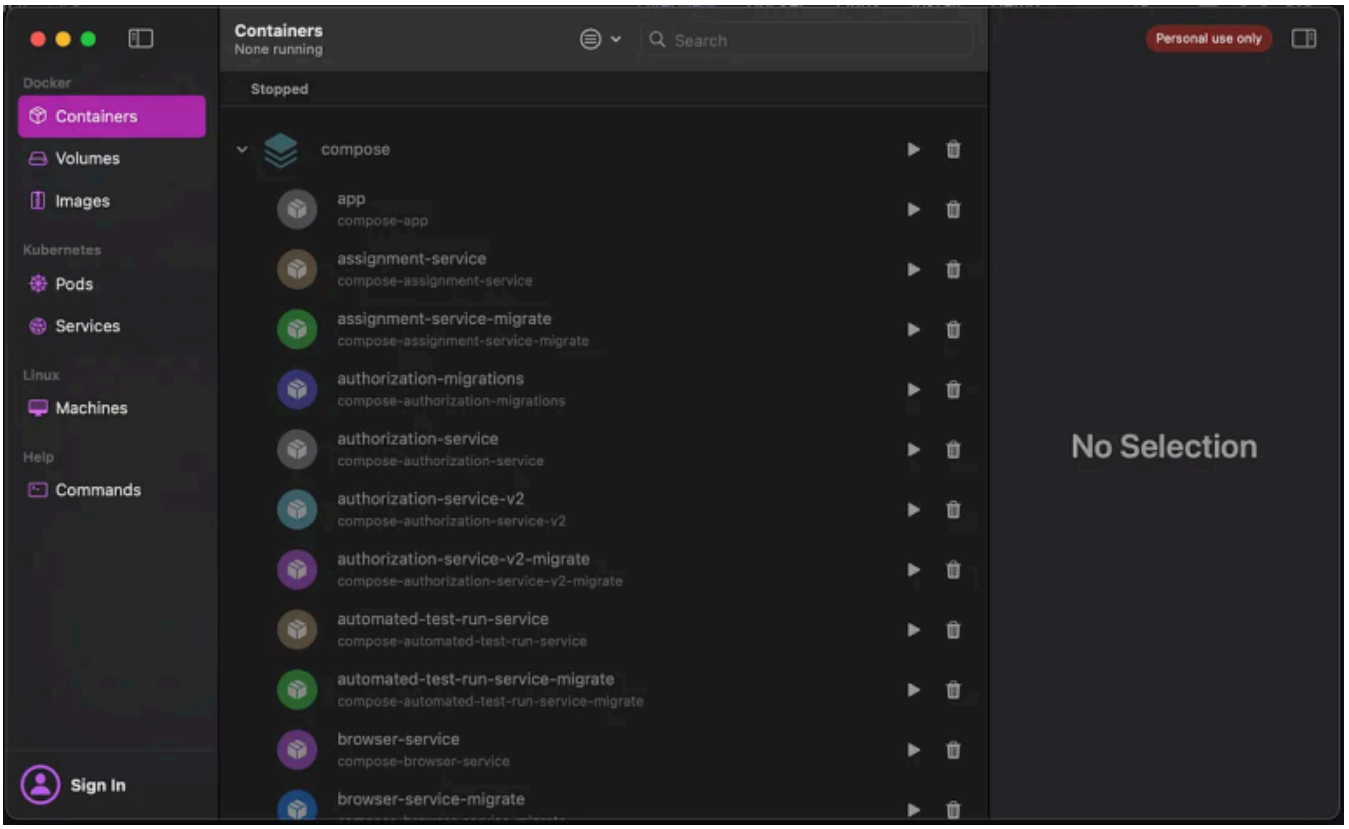


Рис. 2.4 Інтерфейс додатку для управління контейнерами OrbStack

2.2 Мови програмування та технології

2.2.1 TypeScript

TypeScript[11] – це строга синтаксична надбудова над JavaScript, розроблена та підтримувана компанією Microsoft, яка додає до мови можливість використання статичної типізації. TypeScript компілюється у чистий JavaScript, що гарантує його виконання в будь-якому сучасному веб-браузері або середовищі виконання (наприклад, Node.js). Головною перевагою TypeScript є можливість виявляти логічні та синтаксичні помилки ще на етапі написання та компіляції коду, а не під час його виконання (runtime). Це суттєво підвищує загальну надійність, стабільність та безпеку програмного продукту. Окрім цього, наявність статичних типів значно покращує роботу механізмів автодоповнення в IDE, робить архітектуру проекту прозорішою та виконує роль неявної документації для команди.

У межах даної роботи TypeScript використовувався як основна мова програмування для розробки застосунку. Завдяки його можливостям було створено чіткі типи (types) та інтерфейси (interfaces) для опису моделей бази даних, структур обміну даними через API та пропсів компонентів. Це дозволило мінімізувати кількість помилок, пов'язаних із некоректними форматами даних або відсутніми властивостями об'єктів. Використання TypeScript суттєво спростило процес рефакторингу кодової бази при масштабуванні проекту, а також забезпечило високий рівень інтеграції та сумісності з сучасними бібліотеками. Приклад фрагмента коду, написаного мовою TypeScript, наведено на рисунку 2.5.

```typescript
type filterFn = (item: string) => bool;

// The higher-order-function takes an array and a function as arguments
function filterItems(arr: string[], fn: filterFn): string[] {
    const newArray: string[] = [];
    arr.forEach(item => {
        if(fn(item)){
          newArray.push(item);
        }
    });
    return newArray;
}

function checkNameLength(name: string) {
  return name.length >= 10;
}

const doctorList = ["DoctorOne", "DoctorTwo", "DoctorThree", "DoctorFour"];

// We are passing the array and a function as arguments to filterItems method.
const output = filterItems(doctorList, checkNameLength);

console.log(output); // ["DoctorThree", "DoctorFour"]
```

Рис. 2.5 Фрагмент коду, написаного мовою TypeScript

2.2.2 Bun

Bun[12] – це сучасне, надшвидке середовище виконання (runtime) для мов JavaScript та TypeScript, розроблене з використанням мови програмування Zig та рушія JavaScriptCore. На відміну від класичного Node.js, Bun є комплексним інструментом (all-in-one), який поєднує в собі середовище виконання, високошвидкісний менеджер пакетів (сумісний з екосистемою npm), бандлер (bundler) та засіб для виконання тестів. Головною перевагою Bun є його безпрецедентна швидкість запуску та виконання скриптів, оптимізоване використання пам'яті та нативна підтримка TypeScript і JSX "з коробки", що позбавляє необхідності у складних налаштуваннях транспіляторів.

У даній роботі Bun використовувався як основне середовище виконання для серверної частини застосунку, а також як пакетний менеджер для встановлення залежностей. Його вбудована підтримка TypeScript дозволила запускати серверну

логіку безпосередньо, минаючи етап попередньої компіляції, що суттєво пришвидшило цикл розробки та перезавантаження сервера при внесенні змін (hot-reloading). Швидкість роботи пакетного менеджера Bun дозволила оптимізувати процес ініціалізації проєкту та значно скоротити час розгортання додатку в контейнеризованих середовищах. Приклад виконання скрипту за допомогою Bun наведено на рисунку 2.6.

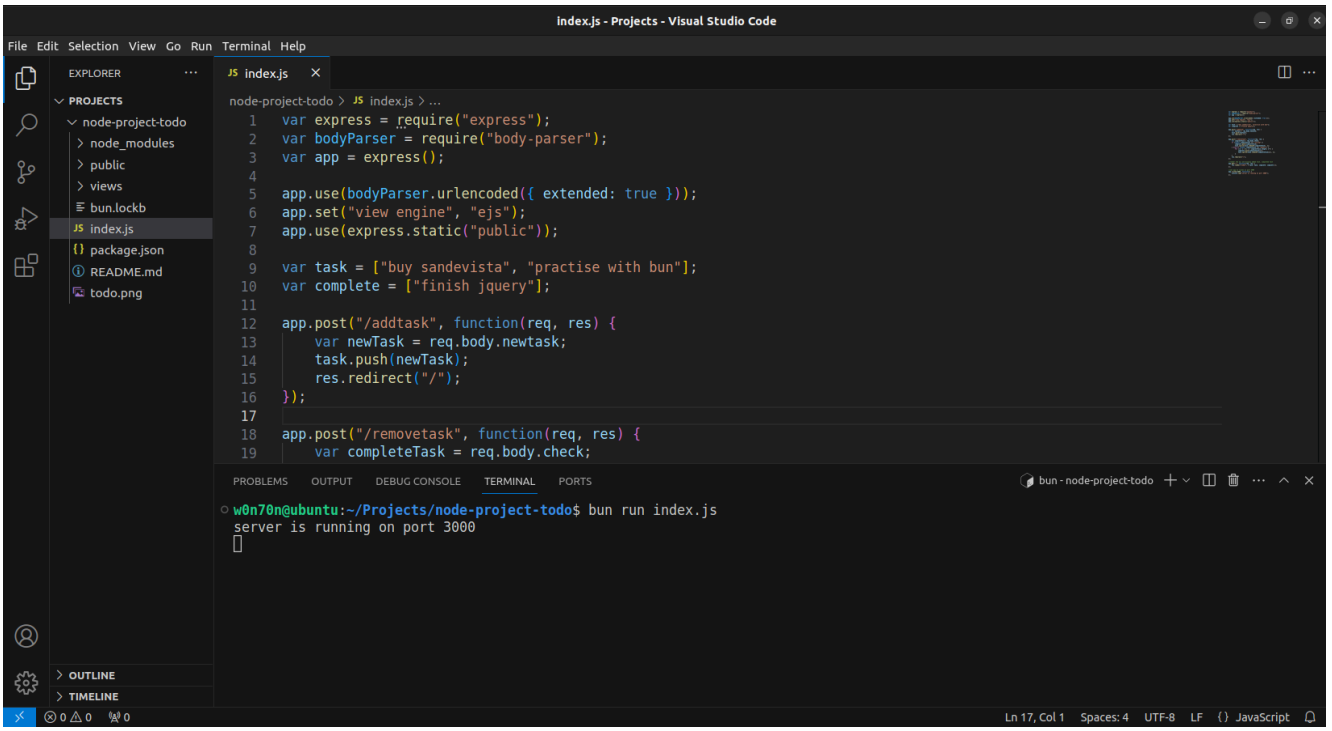


Рис. 2.6 Використання середовища виконання Bun

2.2.3 Elysia.js

Elysia.js[13] – це сучасний високопродуктивний веб-фреймворк, створений спеціально для екосистеми Bun. Він орієнтований на розробку швидких REST API та мікросервісів з максимальним акцентом на ергономіку розробника, продуктивність та сувору типізацію. Elysia має глибоку інтеграцію з TypeScript, пропонуючи концепцію наскрізної типізації (end-to-end type safety). Це дозволяє автоматично виводити типи для запитів і відповідей, що знижує ймовірність помилок. Завдяки архітектурі, оптимізованій під рушій Bun, фреймворк демонструє одні з найвищих показників пропускної здатності (requests per second) серед існуючих рішень для веб-розробки на JavaScript/TypeScript.

У межах цього проєкту Elysia.js виступив як базовий фреймворк для побудови серверної архітектури (API). З його допомогою було реалізовано маршрутизацію (роутинг) HTTP-запитів, обробку проміжних програм (middlewares), авторизацію користувачів та строгу валідацію вхідних даних. Вбудовані інструменти Elysia дозволили легко інтегрувати автоматичну генерацію Swagger-документації для розробленого API безпосередньо на основі коду та типів, що значно полегшило подальшу інтеграцію з фронтенд-частиною застосунку. Приклад конфігурації базового маршруту за допомогою Elysia.js наведено на рисунку 2.7.

```
import { Elysia } from 'elysia'

const app = new Elysia()

app.get('/id/:id', ({ params: { id } }) => id )
  .listen(3000)
```

Рис. 2.7 Фрагмент коду маршрутизації з використанням Elysia.js

2.3 Допоміжні бібліотеки та пакети

2.3.1 MicroORM

MikroORM[14] – це сучасний об'єктно-реляційний відображувач (ORM) для мови TypeScript та середовища Node.js, який базується на архітектурних патернах Data Mapper, Unit of Work та Identity Map. На відміну від багатьох інших популярних ORM, він забезпечує сувору типізацію та оптимізовану роботу із запитами до бази даних завдяки механізму відкладеного виконання та групування запитів. Це допомагає уникнути проблеми "N+1 запиту" та підвищити продуктивність застосунку.

У межах даної роботи MikroORM використовувався для взаємодії серверної

частини з базою даних. Він дозволив описати структуру таблиць у вигляді класів-сутностей (entities) мовою TypeScript, що забезпечило повноцінне автодоповнення та виявлення структурних помилок ще на етапі написання коду. Завдяки цій бібліотеці було реалізовано безпечне та зручне виконання CRUD-операцій, автоматичне управління міграціями схеми даних та підтримання цілісності бази без необхідності безпосереднього написання сирих SQL-запитів.

2.3.2 Pompelmi та Sharp

Pompelmi[15] та Sharp[16] – це спеціалізовані програмні пакети, що застосовуються для комплексної та безпечної обробки файлів, завантажених користувачами на сервер. Pompelmi – це потужний інструмент-обгортка, створений для інтеграції антивірусного сканера ClamAV у Node.js/Bun середовище. Він дозволяє автоматично сканувати будь-які вхідні файли та визначати їх статус (чистий чи шкідливий), захищаючи систему від потенційних malware-загроз. Sharp зі свого боку є популярною та високоефективною бібліотекою для обробки зображень, яка працює на базі швидкого рушія libvips.

У розробленому застосунку ці два пакети використовувалися у тандемі виключно для безпечного завантаження та обробки зображення профілю користувача. Спочатку завантажений файл аватарки проходить обов'язкову перевірку через Pompelmi: у разі виявлення загрози шкідливого коду файл негайно відхиляється. Якщо ж файл є безпечним, до нього застосовується обробка за допомогою Sharp для автоматичного конвертації у сучасний оптимізований формат WebP. Такий підхід дозволив оптимізувати збереження профілів користувачів та гарантувати захист сервера від завантаження шкідливих файлів замість зображень.

2.3.3 Arctic (OAuth)

Arctic[17] – це легка та спеціалізована бібліотека для реалізації авторизації за протоколом OAuth 2.0 та OpenID Connect у сучасних JavaScript та TypeScript додатках. Пакет розроблений з акцентом на безпеку та гнучкість, надаючи

низькорівневі, але зручні та типізовані інструменти для інтеграції з популярними провайдерами ідентифікації.

У даному проєкті пакет Arctic застосовувався для побудови безпечної системи автентифікації користувачів виключно через обліковий запис Google (Google OAuth). Використання цієї бібліотеки значно спростило процес управління усім OAuth-потоком обміну даними з серверами Google: від генерації URL-адрес для ініціалізації авторизації до валідації параметрів стану (state) та перевірки PKCE (Proof Key for Code Exchange) для захисту від потенційних CSRF-атак. Arctic забезпечив безпечний та надійний обмін кодів авторизації на токени доступу (access tokens), позбавивши необхідності розробляти складні криптографічні механізми перевірки вручну.

2.4 Бази даних

2.4.1 PostgreSQL

PostgreSQL[18] – це потужна об'єктно-реляційна система управління базами даних (СУБД) з відкритим вихідним кодом, яка розробляється понад 30 років. Вона славиться своєю надійністю, високою продуктивністю, строгим дотриманням стандартів SQL та повною підтримкою властивостей транзакцій ACID (атомарність, узгодженість, ізольованість, довговічність). PostgreSQL пропонує розширені можливості, такі як підтримка користувацьких типів даних, складних об'єднань, тригерів, а також ефективну роботу з неструктурованими даними у форматі JSONB, що робить її універсальним рішенням для проєктів будь-якого масштабу.

У межах даної роботи PostgreSQL була обрана як основна реляційна база даних для зберігання ключової інформації застосунку: облікових записів користувачів, конфігураційних налаштувань, зв'язків між сутностями та інших бізнес-даних. Сувора типізація та підтримка зовнішніх ключів (foreign keys) дозволили забезпечити цілісність даних на рівні сховища. Крім того, завдяки

інтеграції з MikroORM, взаємодія з PostgreSQL була максимально оптимізована, що забезпечило швидку обробку запитів та надійне збереження стану системи.

2.4.2 MongoDB

MongoDB[19] – це одна з найпопулярніших у світі документо-орієнтованих систем управління базами даних класу NoSQL. Замість традиційних реляційних таблиць, MongoDB зберігає дані у гнучких, подібних до JSON документах (формат BSON). Це означає, що структура записів може змінюватися без необхідності жорсткого визначення схеми (schema-less), що є ідеальним для роботи з великими обсягами гетерогенних або ієрархічних даних. СУБД відрізняється високою масштабованістю, продуктивністю при виконанні операцій читання/запису та вбудованими можливостями для горизонтального шардування.

У розроблюваному проєкті MongoDB застосовувалася виключно для ведення логів (журналювання) активності користувачів. Основною метою її використання було збереження повної історії запитів, що надходять від клієнтів до серверної частини. Завдяки гнучкій структурі документів, у базу зручно записувати деталі кожного запиту (заголовки, тіло, параметри, час виконання, статус-коди тощо) без необхідності попереднього створення жорсткої схеми таблиць для кожного типу логу. Це дозволило створити ефективний інструмент моніторингу: у разі виникнення помилок у системі розробник може швидко відфільтрувати та переглянути всі запити конкретного користувача, що значно полегшує та пришвидшує процес пошуку несправностей (дебаггінгу) та їхнього усунення.

2.5 Інфраструктура

2.5.1 Coolify

Coolify[20] – це потужна платформа з відкритим вихідним кодом, що позиціонується як альтернатива комерційним хмарним сервісам типу PaaS (Platform as a Service), таким як Heroku або Vercel, з можливістю розгортання на

власних серверах (self-hosted). Інструмент надає зручний графічний веб-інтерфейс для автоматизованого розгортання, управління та моніторингу додатків, баз даних і сервісів, використовуючи технологію контейнеризації Docker під капотом. Завдяки вбудованим механізмам Continuous Integration та Continuous Deployment (CI/CD), Coolify значно спрощує процес безперервної доставки програмного коду від репозиторію системи контролю версій до робочого (production) середовища.

У даній роботі Coolify використовувався як основний інструмент для автоматизації процесів розгортання серверної частини застосунку та керування інфраструктурою. Завдяки його можливостям було налаштовано автоматичний деплой застосунку при кожному оновленні (push) коду в основній гілці репозиторію. Крім того, платформа дозволила централізовано управляти супутніми сервісами інфраструктури, надаючи зручні інструменти для моніторингу споживання апаратних ресурсів (навантаження на процесор, використання оперативної пам'яті) та перегляду серверних логів у режимі реального часу, позбавивши необхідності ручного написання та підтримки складних DevOps-скриптів. Приклад інтерфейсу Coolify наведено на рисунку 2.8.

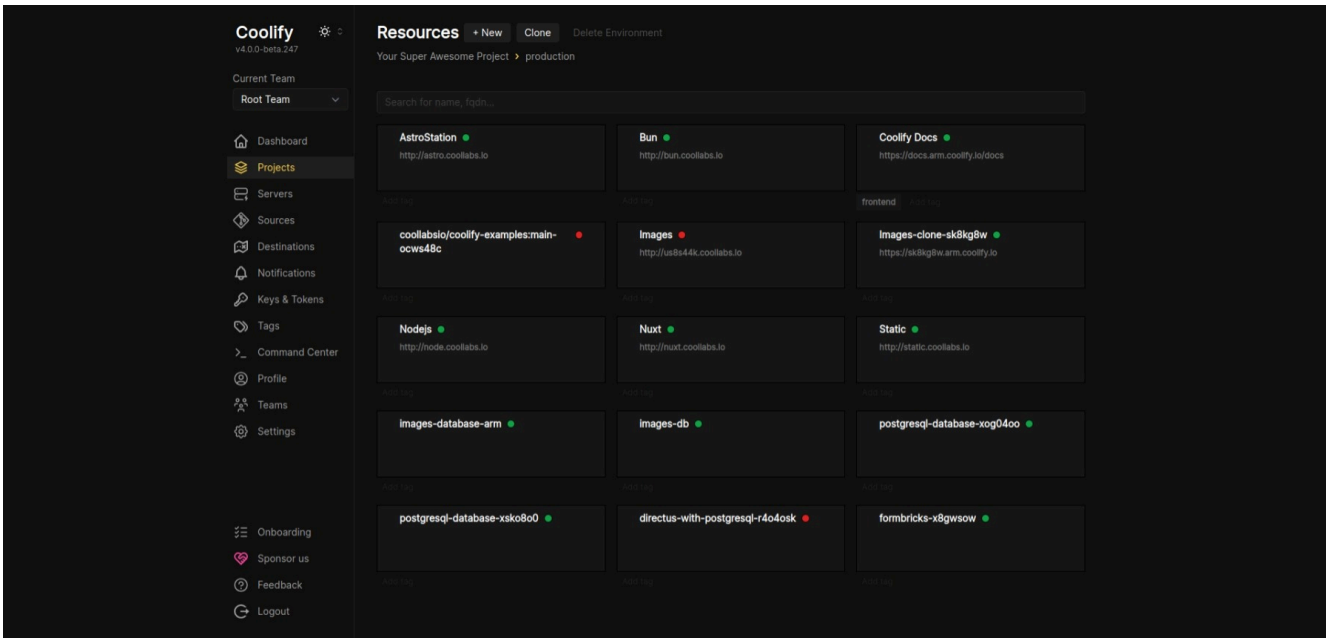


Рис. 2.8 Інтерфейс програми Coolify

2.5.2 MinIO

MinIO[21] – це високопродуктивне розподілене об'єктне сховище з відкритим вихідним кодом, яке повністю сумісне з API хмарного сервісу Amazon S3. Воно розроблене для надійного зберігання великих обсягів неструктурованих даних, таких як фотографії, відео, резервні копії та статичні ресурси веб-додатків. MinIO відрізняється простотою розгортання, мінімальним споживанням ресурсів і надзвичайно високою швидкістю операцій читання та запису (I/O), що робить його ідеальним вибором як для локального середовища розробки, так і для побудови приватних або гібридних хмарних інфраструктур.

У межах розроблюваного програмного комплексу MinIO використовувалося як ізольоване об'єктне сховище для статичного медіаконтенту, зокрема для збереження оптимізованих зображень профілів (аватарів) користувачів. Після перевірки файлу на шкідливий код та його обробки за допомогою бібліотеки Sharp, готове зображення безпечно завантажується у відповідний бакет (bucket) MinIO. Таке архітектурне рішення дозволило винести зберігання важких медіафайлів за межі основної реляційної бази даних, тим самим розвантаживши PostgreSQL та оптимізувавши швидкість виконання SQL-запитів. Крім того, повна сумісність із протоколом S3 забезпечує високу гнучкість системи: у разі масштабування проєкту локальне сховище MinIO можна буде миттєво замінити на будь-яке комерційне хмарне рішення (наприклад, AWS S3 або Cloudflare R2) без необхідності переписування програмного коду серверної частини. Приклад інтерфейсу MinIO наведено на рисунку 2.9.

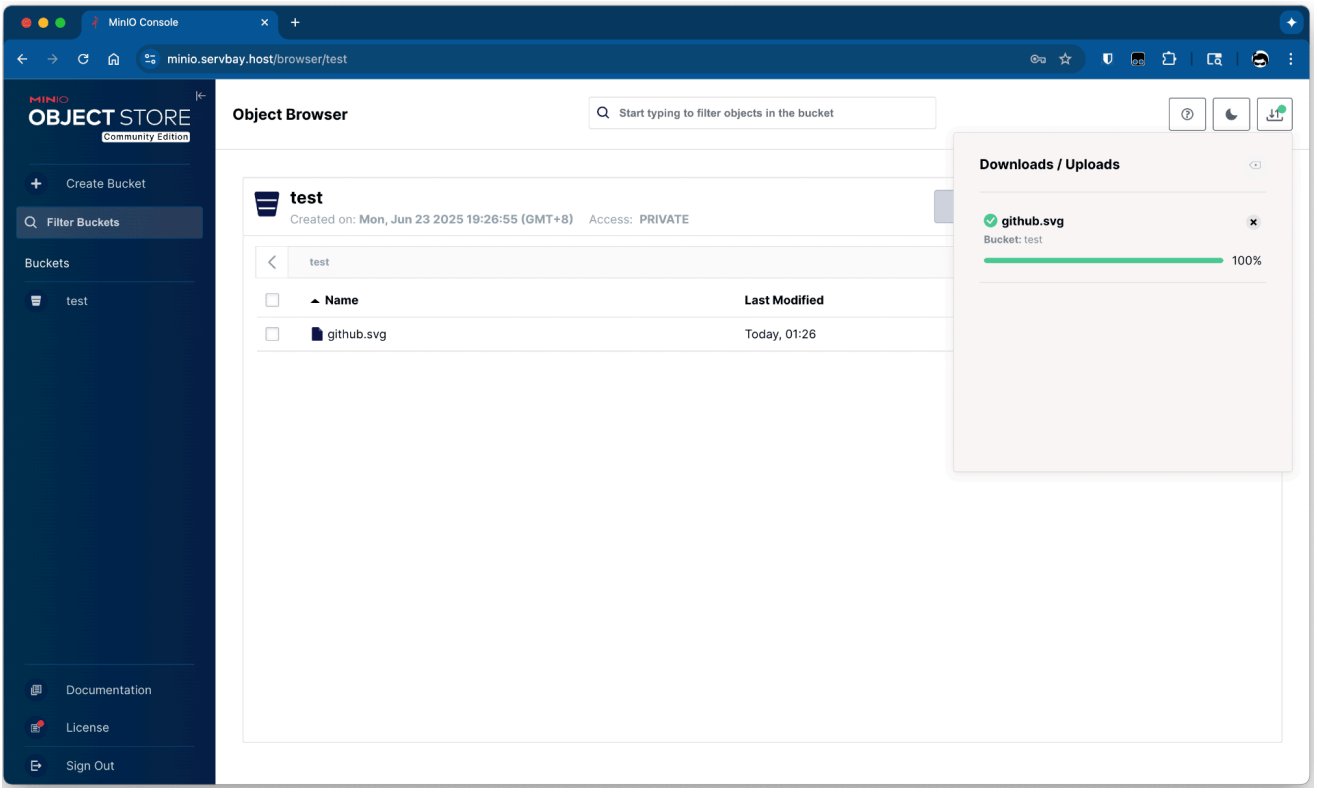


Рис. 2.9 Інтерфейс програми MinIO

2.5.3 Infisical

Infisical[22] – це сучасна платформа з відкритим вихідним кодом для централізованого управління секретами, конфігураціями та змінними середовища (environment variables). Інструмент розроблений для запобігання витоку конфіденційних даних і забезпечує наскрізне шифрування (end-to-end encryption) під час синхронізації секретів між розробниками, CI/CD-пайплайнами та серверами. На відміну від традиційного локального збереження файлів .env, Infisical автоматизує процес безпечної ін'єкції конфігурацій безпосередньо у середовище виконання проєкту.

У даній роботі Infisical відіграв критично важливу роль у забезпеченні безпеки інфраструктури застосунку. З його допомогою було організовано безпечне зберігання та керування всіма чутливими даними: ключами доступу до баз даних (PostgreSQL, MongoDB), обліковими даними для доступу до об'єктного сховища MinIO, секретними ключами для OAuth-авторизації через Google, а також токенами інтеграції для зовнішніх сервісів (OpenAI API та платіжний шлюз Paddle). Завдяки використанню Infisical було повністю усунуто ризик випадкового

потрапляння секретів до репозиторію системи контролю версій Git (hardcoding). Платформа дозволила чітко розмежувати змінні для різних середовищ (розробка, тестування, продакшн) та забезпечила автоматичне, швидке й безпечне підтягування актуальних налаштувань під час розгортання серверної частини. Приклад інтерфейсу Infisical наведено на рисунку 2.10.

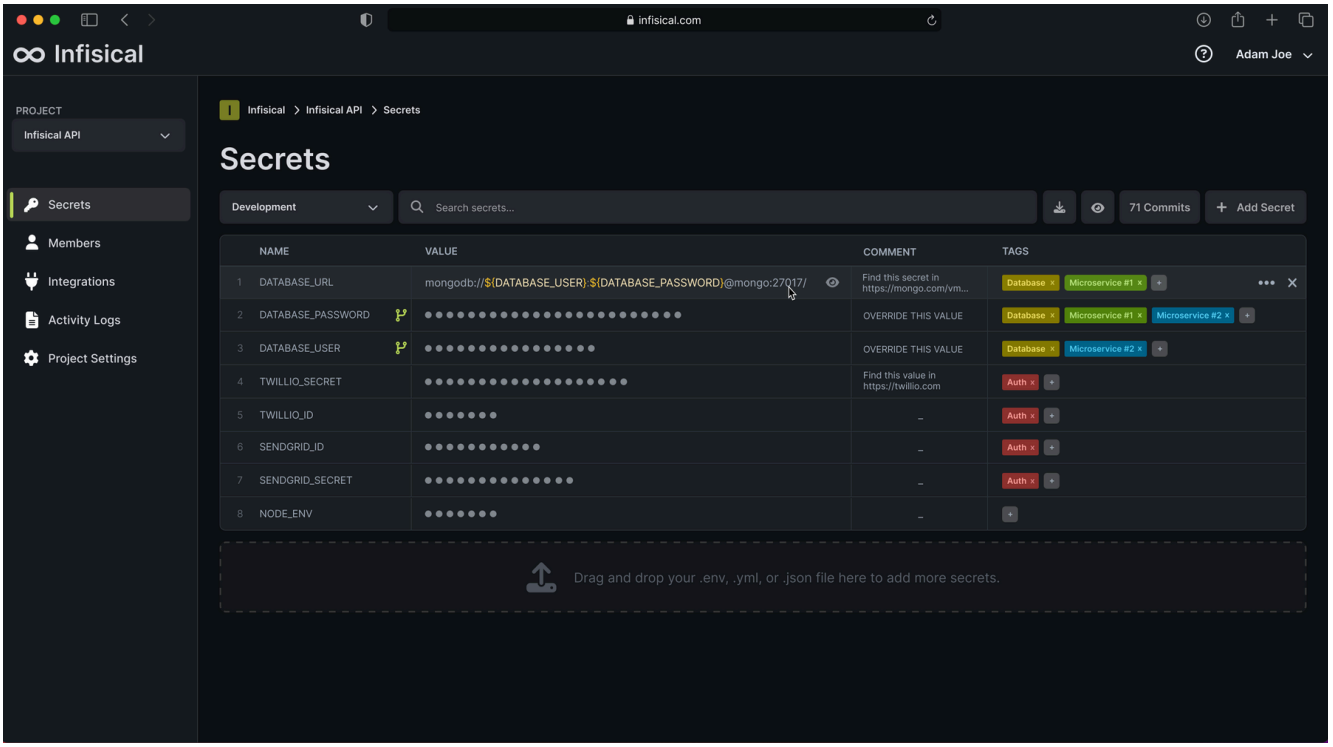


Рис. 2.10 Інтерфейс програми Infisical

2.6 Зовнішні сервіси та інтеграції

2.6.1 OpenRouter

OpenRouter[23] – це спеціалізований сервіс-агрегатор (маршрутизатор), який надає розробникам єдиний уніфікований програмний інтерфейс (API) для доступу до широкого спектру різноманітних великих мовних моделей (LLM) від різних постачальників, таких як OpenAI, Anthropic, Google, Meta, та безлічі моделей з відкритим вихідним кодом. Платформа автоматично стандартизує запити та відповіді, абстрагуючи розробника від складнощів роботи з пропрієтарними API кожного окремого провайдера. Головною перевагою OpenRouter є можливість

динамічного перемикання між різними мовними моделями в режимі реального часу, оптимізація витрат шляхом маршрутизації до найдешевших вузлів та забезпечення високої доступності сервісів (failover).

У межах даного проєкту інтеграція з OpenRouter відіграла ключову роль у реалізації базового функціоналу застосунку – генерації стислих викладів (саммарі) текстової інформації. Замість жорсткої прив'язки до одного постачальника штучного інтелекту, використання OpenRouter дозволило гнучко обирати найбільш ефективну та економічно вигідну мовну модель для обробки конкретних текстових запитів. Серверна частина застосунку (на базі Bun та Elysia.js) формує стандартизований промпт із виділеним користувачем текстом і відправляє його до OpenRouter, отримуючи натомість структурований і осмислений результат, який миттєво повертається клієнту для відображення в браузері.

2.6.2 Paddle

Paddle[24] – це комплексна платформа для управління платежами, яка працює за моделлю Merchant of Record (MoR), створена спеціально для SaaS-компаній та розробників програмного забезпечення. На відміну від традиційних платіжних шлюзів (таких як Stripe або PayPal), Paddle виступає офіційним торговим посередником, беручи на себе повну юридичну та фінансову відповідальність за обробку транзакцій, глобальний збір і сплату податків (наприклад, ПДВ у різних країнах), а також управління поверненнями коштів (refunds) та чарджбеками. Платформа пропонує потужний API та систему вебхуків (webhooks) для гнучкої інтеграції в сучасні веб-додатки.

У межах розроблюваного застосунку Paddle використовувався для реалізації модуля монетизації та управління користувацькими підписками. Інтеграція з цією платформою дозволила безпечно та зручно приймати платежі від користувачів за доступ до розширеного функціоналу системи (наприклад, збільшених лімітів використання генеративного штучного інтелекту). За допомогою обробки вебхуків (webhook-подій) на серверній частині було повністю автоматизовано процеси активації, подовження терміну дії та скасування підписок у режимі реального часу.

Використання моделі MoR значно спростило розробку бізнес-логіки бекенду, оскільки дозволило делегувати складні механізми міжнародного податкового обліку та захисту від фінансового шахрайства зовнішньому спеціалізованому сервісу. Приклад інтерфейсу Paddle наведено на рисунку 2.11.

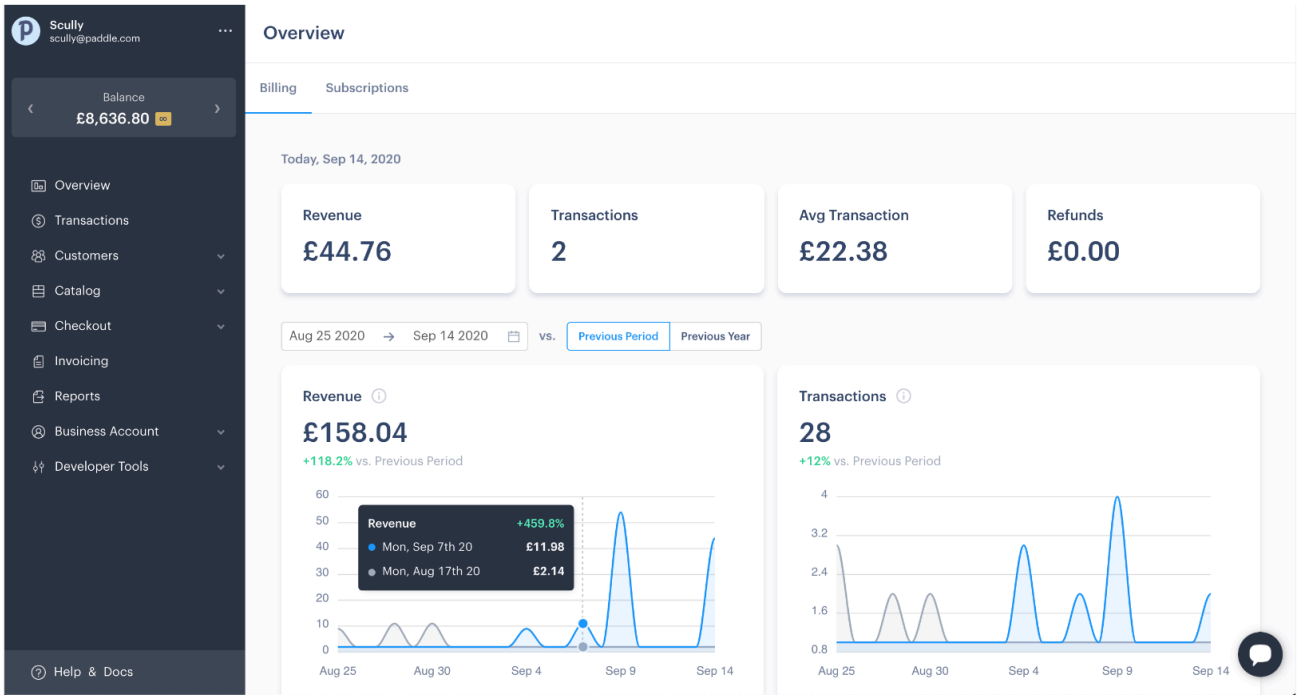


Рис. 2.11 Інтерфейс програми Paddle

3 ПРОЄКТУВАННЯ

3.1 Вимоги до програмного забезпечення

Основою для успішного проектування серверної (Backend) частини застосунку є формування чітких функціональних та нефункціональних вимог. Ці вимоги базуються на результатах аналізу предметної області, потреб цільової аудиторії та виявлених недоліках існуючих рішень (зокрема, високих мережевих затримок та відсутності надійного захисту даних).

3.1.1 Функціональні вимоги

Функціональні вимоги визначають базову поведінку системи та перелік завдань, які вона повинна виконувати для забезпечення повноцінної роботи клієнтського застосунку. Розроблюваний Backend повинен забезпечувати наступні ключові функції:

1. **Управління користувачами та сесіями:** забезпечення безпечної реєстрації, автентифікації та виходу з системи. Підтримка механізму Single Sign-On (OAuth 2.0) через акаунти Google із можливістю автоматичного об'єднання акаунтів (Account Linking).
2. **Обробка та узагальнення тексту:** прийом масивів текстових даних від клієнтського браузерного розширення та взаємодія із зовнішніми API великих мовних моделей для генерації сумаризації.
3. **Глобальне кешування запитів:** реалізація алгоритму дедуплікації. Система повинна генерувати криптографічний хеш вхідного тексту і, у разі його наявності в базі даних, миттєво повертати збережений результат без повторного звернення до платного API ШІ.

4. **Управління підписками:** інтеграція з платіжним провайдером (Paddle) для обробки вебхуків, автоматичного оновлення статусів підписок та контролю лімітів на використання AI-модуля.
5. **Криптографічний захист:** автоматичне шифрування персональних даних (PII) у базі даних та пошук користувачів за допомогою детермінованих «сліпих індексів».

3.1.2 Нефункціональні вимоги

Нефункціональні вимоги описують атрибути якості системи, такі як продуктивність, безпека та надійність:

1. **Продуктивність та низька затримка:** серверна архітектура повинна обробляти базові HTTP-запити з мінімальними затримками. Використання середовища Bun та фреймворку ElysiaJS має забезпечити середній час виконання серверної бізнес-логіки до <8 секунд (з урахуванням часу відповіді від LLM).
2. **Безпека PII:** зберігання даних має відповідати європейським стандартам конфіденційності. Паролі повинні бути захешовані алгоритмами, стійкими до перебору (Argon2), а сесії – захищені підписаними HttpOnly cookie.
3. **Спостережуваність:** система повинна вести асинхронний запис логів, HTTP-метрик та помилок у зовнішнє NoSQL-сховище (MongoDB) без деградації продуктивності основної транзакційної бази (PostgreSQL).
4. **Масштабованість:** застосунок має бути контейнеризований (Docker) для безперешкодного розгортання в оркестраторах (Coolify) та легкого горизонтального масштабування.

Діаграму варіантів використання наведено на рисунку 3.1.

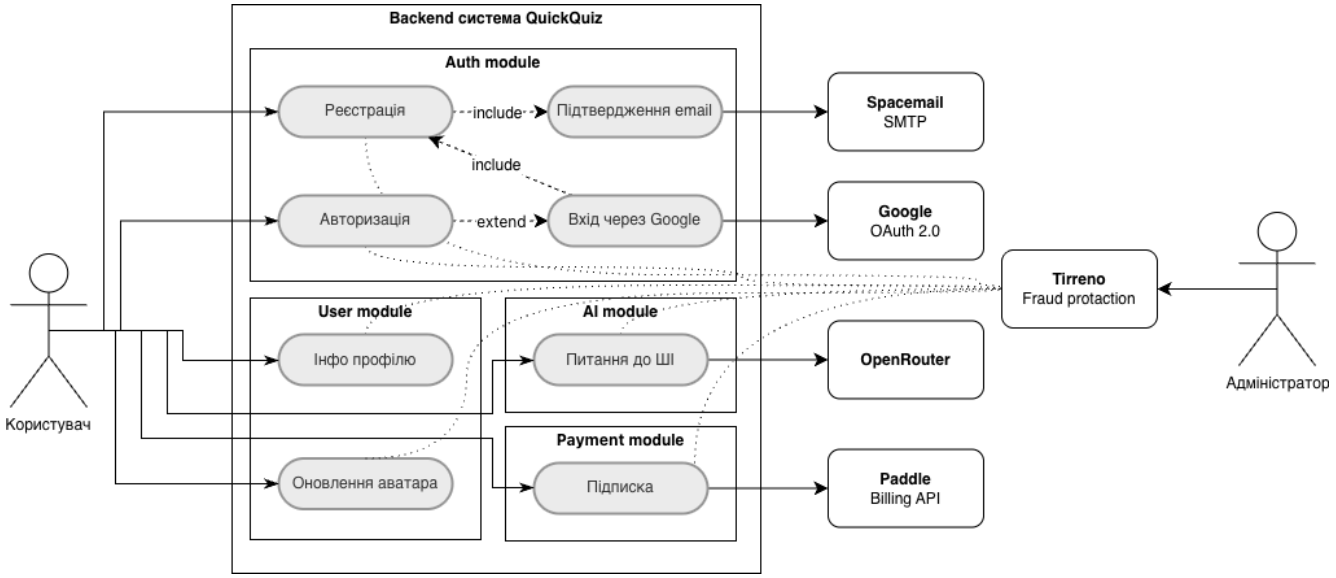


Рис. 3.1 Діаграма варіантів використання

3.2 Проєктування архітектури застосунку

Проєктування архітектури застосунку є одним із найвідповідальніших етапів створення інформаційної системи. У межах розробки системи «QuickSum» було прийнято рішення реалізувати архітектуру за класичною багаторівневою клієнт-серверною моделлю із застосуванням патерну «Модульний моноліт» на серверній стороні. Такий підхід дозволяє розділити відповідальність між клієнтською та серверною частинами, що сприяє підвищенню масштабованості, продуктивності та зручності супроводу програмного забезпечення.

Клієнтська частина реалізована у вигляді двох взаємопов'язаних компонентів: браузерного розширення на базі фреймворку Plasmo для швидкого захоплення тексту та повноцінного SPA (Single Page Application) Web-застосунку, створеного за допомогою бібліотеки React. Вона відповідає за відображення інформації, взаємодію з користувачем, виконання асинхронних запитів до серверу, обробку відповідей API та підтримку стану інтерфейсу. Інтерфейс працює динамічно без перезавантаження сторінок, що покращує користувацький досвід і знижує пряме навантаження на сервер.

Серверна частина, яка є основним предметом даного дослідження, написана з використанням мови TypeScript, інноваційного середовища виконання Bun та фреймворку ElysiaJS. Цей стек забезпечує виконання складної логіки бізнес-процесів (включаючи взаємодію з великими мовними моделями ШІ), перевірку даних, безпечну автентифікацію користувачів, обробку запитів та взаємодію з базою даних. Архітектура застосунку є RESTful, тобто використовує стандартні HTTP-методи (GET, POST, PUT, DELETE) для взаємодії з ресурсами через захищений API.

Інфраструктурна частина системи базується на технології контейнеризації Docker, що дозволяє створювати ізольовані середовища для серверного ядра, реляційної бази даних PostgreSQL, NoSQL-сховища MongoDB (для логування) та об'єктного сховища MinIO. Управління інфраструктурою (Self-hosting) здійснюється за допомогою платформи оркестрації Coolify. Це забезпечує стабільність роботи системи, її гнучке масштабування, а також спрощує адміністрування, оскільки внутрішній маршрутизатор платформи автоматично перенаправляє запити до відповідних служб та забезпечує безпечне з’єднання (SSL).

Процеси безперервної інтеграції та доставки реалізовані за допомогою вбудованих можливостей оркестратора Coolify у поєднанні з менеджером секретів Infisical. Репозиторій проєкту інтегровано таким чином, що система дозволяє автоматично збирати та розгортати застосунок при кожному оновленні коду, паралельно здійснюючи безпечну ін'єкцію конфіденційних змінних оточення (ключів API, паролів баз даних). Це скорочує час релізів, мінімізує ймовірність помилок при деплої та гарантує відсутність витоку секретів.

Завдяки такому підходу архітектура системи «QuickSum» є розділеною, стійкою до високих навантажень (low latency) та гнучкою до змін. Вона дозволяє швидко розгортати нові функції, оптимізувати витрати ресурсів завдяки продуманому кешуванню запитів та забезпечує високий рівень безпеки за рахунок ізоляції компонентів і криптографічного шифрування персональних даних. Діаграму клієнт-серверної архітектури системи наведено на рисунку 3.4.

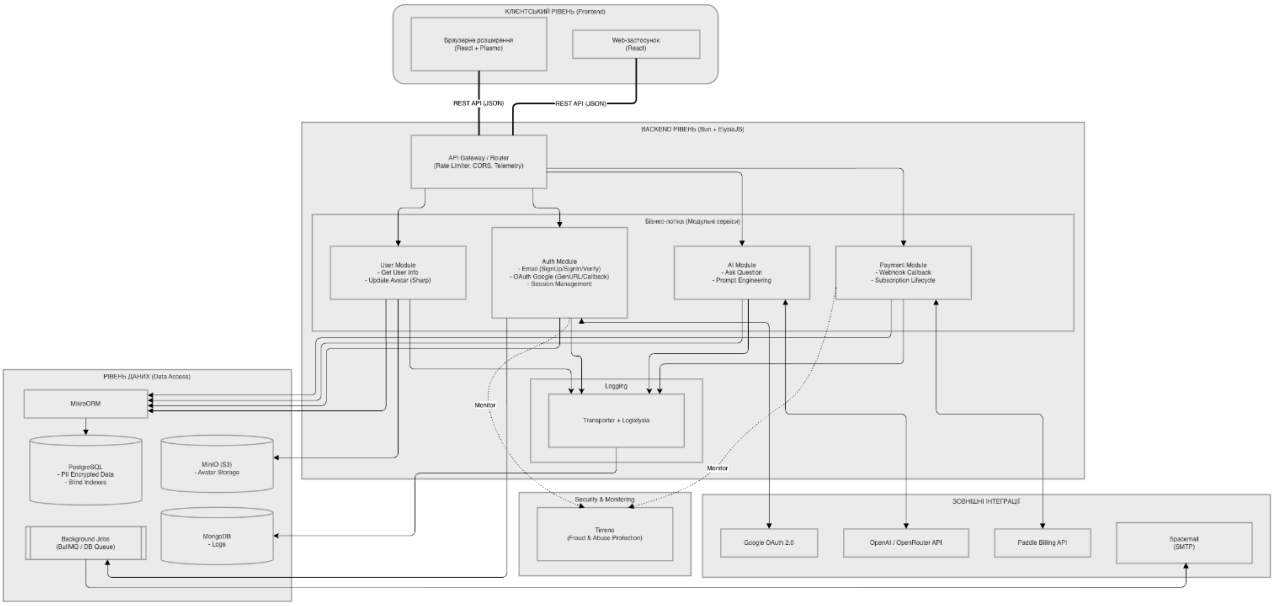


Рис. 3.2 Архітектура компонентів клієнт-серверного застосунку

3.3 Архітектура серверної частини

Основою високої продуктивності та надійності розроблюваної системи «QuickSum» є оптимізована внутрішня архітектура серверної частини (Backend). На відміну від загальної клієнт-серверної моделі, внутрішня структура бекенду побудована за принципами чистої архітектури (Clean Architecture) із використанням концепції «Контролер-Сервіс-Модель» (Controller-Service-Model) в екосистемі ElysiaJS.

3.3.1 Середовище виконання та маршрутизація

Серверна логіка виконується у середовищі Bun, яке використовує рушій JavaScriptCore. Це забезпечує нативну підтримку TypeScript без необхідності попередньої транспіляції коду, блискавичний запуск застосунку (Cold Start) та ефективне управління пам'яттю.

В якості ядра маршрутизації (Router) обрано фреймворк ElysiaJS. Його головною архітектурною перевагою є повна наскрізна типізація (End-to-End Type Safety). Завдяки інтеграції з бібліотекою TypeBox, серверна частина виконує сувору типізацію та валідацію всіх вхідних даних (тіло запиту, параметри URL, заголовки) безпосередньо на етапі маршрутизації. Якщо клієнт надсилає

невалідний JSON або відсутні обов'язкові параметри, фреймворк автоматично відхиляє запит зі статусом 400 Bad Request, не доводячи його до рівня бізнес-логіки. Це суттєво знижує ризик виникнення помилок під час виконання програми (Runtime Errors).

3.3.2 Модульна структура бізнес-логіки

Бізнес-логіка застосунку розділена на незалежні сервіси, кожен з яких відповідає за свою предметну область. Усі сервіси інкапсульовані у відповідні плагіни ElysiaJS:

1. **Модуль автентифікації:** Відповідає за безпеку доступу. Реалізує методи реєстрації, авторизації, підтвердження пошти та логіку роботи з OAuth-провайдерами. Модуль генерує унікальні токени верифікації та відповідає за створення безпечних сесій через інструментарій Lucia Auth. Для фонової відправки електронних листів модуль взаємодіє з чергою повідомлень BullMQ.
2. **Модуль штучного інтелекту:** Ключовий вузол обробки тексту. Відповідає за прийом тексту, його криптографічне хешування для пошуку в базі даних (Cache Hit/Miss), формування оптимізованих промптів та безпосередню взаємодію з OpenAI SDK. Модуль містить логіку списання спроб/токенів з балансу підписки користувача.
3. **Платіжний модуль:** Реалізує захищений ендпоінт для прийому вебхуків від системи Paddle. Архітектура модуля передбачає обов'язкову верифікацію криптографічного підпису кожного вхідного вебхука для запобігання підробці фінансових транзакцій. Відповідає за оновлення рівнів доступу (Free, Pro) у базі даних PostgreSQL.

3.3.3 Пайплайн обробки запитів (Middleware)

Архітектура ElysiaJS базується на концепції хуків (Hooks) – життєвого циклу обробки HTTP-запиту. Кожен запит до застосунку «QuickSum» проходить через строгий захищений пайплайн:

1. **onRequest (Перехоплення):** На цьому етапі спрацьовують глобальні захисні механізми (CORS, Rate Limiter) та відбувається фіксація початку запиту для метрик (OpenTelemetry).
2. **beforeHandle (Валідація та Безпека):** Тут виконується макрос авторизації (перевірка сесійного cookie через Lucia), а також запит передається на аналіз до rule-engine системи **Tirreno**. Якщо алгоритми Tirreno виявляють аномальну поведінку або фрод (наприклад, масову реєстрацію з однієї IP-адреси), запит блокується.
3. **handler (Виконання):** Безпосередня передача даних у відповідний сервіс (AuthService, AIService тощо) та взаємодія з ORM і базою даних.
4. **afterHandle (Логування):** Після формування відповіді, плагін **Logixlysia** асинхронно записує результати виконання (статус-код, час затримки, ідентифікатор користувача) у базу даних MongoDB, не затримуючи при цьому відправку відповіді клієнту.

На рисунку 3.3 наведено UML-діаграму активності, що відображає пайплайн життєвого циклу обробки HTTP-запиту всередині серверної частини.

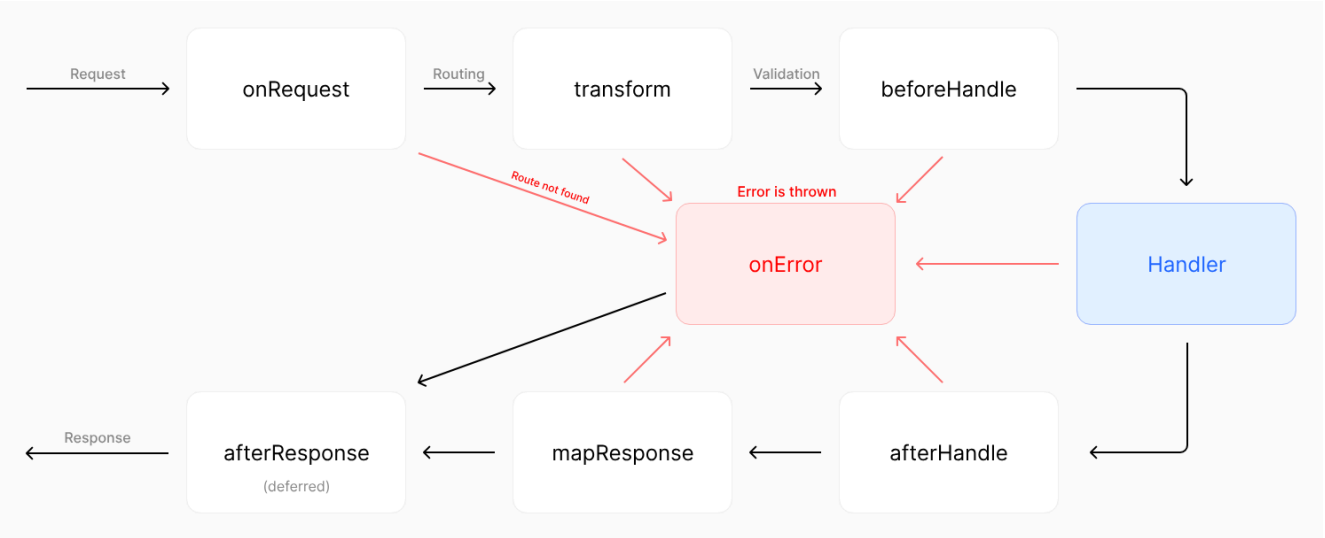


Рис. 3.3 Пайплайн життєвого циклу обробки запиту у фреймворку ElysiaJS

3.4 Архітектура баз даних

Для зберігання транзакційних даних, профілів користувачів та управління доступом у системі «QuickSum» використовується потужна об'єктно-реляційна система управління базами даних PostgreSQL. Архітектура бази даних спроектована з урахуванням суворих вимог європейського регламенту захисту персональних даних (GDPR) та базується на принципі «Secure by Design».

Вона дозволяє здійснювати безпечний облік користувачів, відстеження статусів підписок та лімітів, контроль життєвого циклу авторизаційних сесій (Lucia Auth) та керування зовнішніми ідентифікаторами (OAuth 2.0). Схема даних розроблена у вигляді набору нормалізованих взаємопов'язаних таблиць (user, session, account, subscription, emailVerify).

Приклад ER-діаграми (Entity-Relationship) розробленої бази даних наведено на рисунку 3.4.

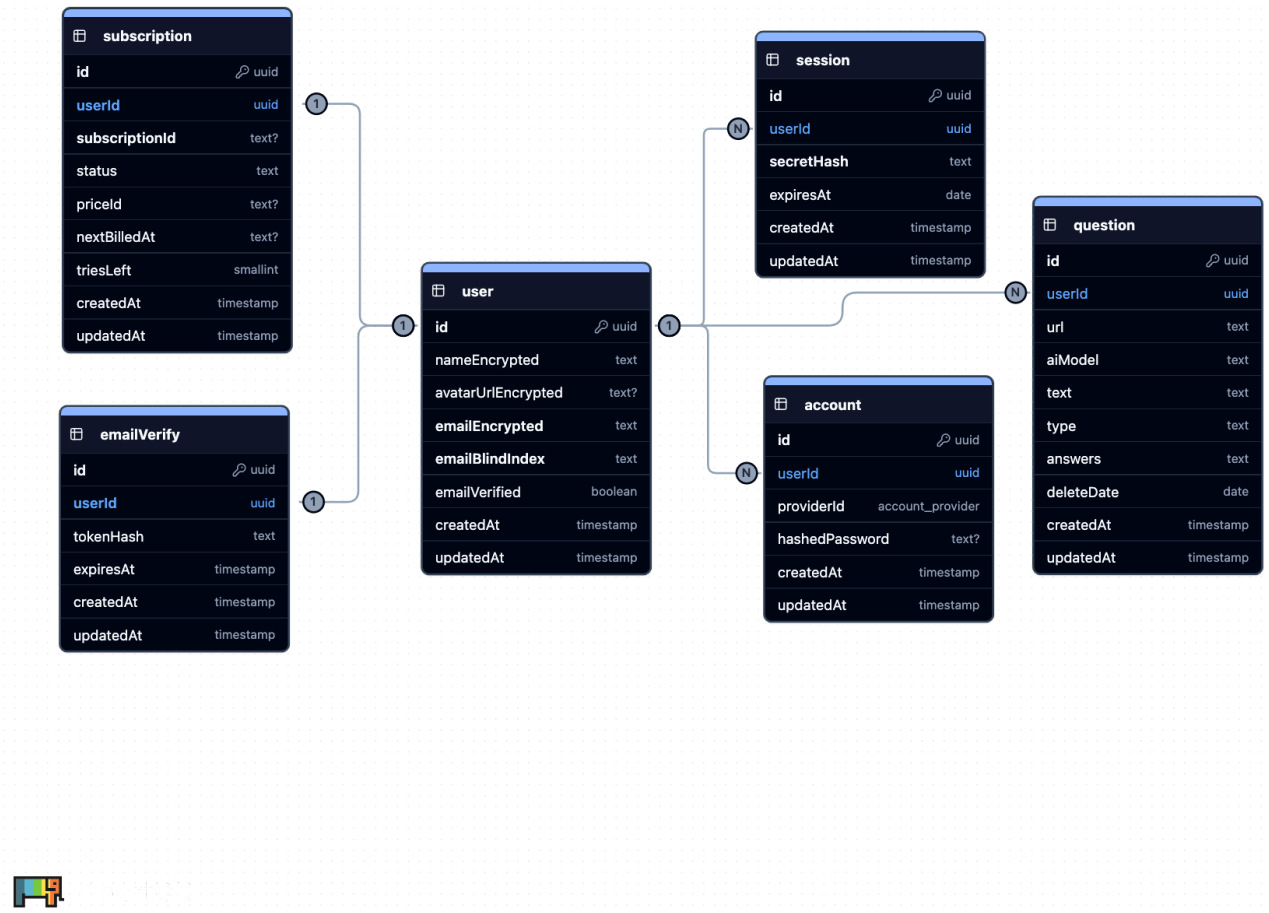


Рис. 3.4 Архітектура бази даних серверної частини

Основу бази даних складає таблиця user, яка акумулює профіль користувача, проте не зберігає жодної інформації у відкритому вигляді. Усі персональні ідентифікатори криптографічно захищені. Замість традиційних полів використовуються зашифровані колонки (nameEncrypted, emailEncrypted, avatarUrlEncrypted).

Таблиця account винесена окремо для реалізації патерну об'єднання акаунтів (Account Linking) – вона зберігає хешовані паролі або ідентифікатори сторонніх провайдерів (наприклад, Google). Таблиця session містить хеші секретів активних сесій, а subscription – фіксує ідентифікатори платіжного шлюзу Paddle, статус

поточного тарифу та залишок доступних спроб (triesLeft) на генерацію узагальнень.

Переваги розробленої архітектури бази даних: – повна конфіденційність персональних даних завдяки симетричному шифруванню критичних колонок на рівні ORM; – швидкий та безпечний пошук користувачів (наприклад, під час авторизації) без дешифрування бази завдяки використанню детермінованих «сліпих індексів» (emailBlindIndex); – гнучка підтримка мультипровайдерної авторизації завдяки відокремленню таблиці account, що дозволяє одному користувачу мати декілька способів входу; – надійна система аудиту завдяки автоматичній фіксації часових міток створення та оновлення записів (createdAt, updatedAt) у кожній таблиці; – висока продуктивність та цілісність даних завдяки строгому використанню зовнішніх ключів (Foreign Keys) з каскадними діями та унікальних ідентифікаторів формату UUID.

Окрім основної реляційної бази даних, архітектура системи реалізує патерн поліглотного збереження даних (Polyglot Persistence). Для забезпечення підсистеми спостережуваності (Observability) та неперервного моніторингу інтегровано документо-орієнтовану NoSQL базу даних MongoDB. Вона відповідає виключно за асинхронний збір системних логів, метрик продуктивності та телеметрії HTTP-запитів. Перехоплення та відправка логів до MongoDB здійснюється за допомогою плагіна Logixlysia на етапі завершення обробки запиту. Таке архітектурне розділення дозволяє уникнути перевантаження основної транзакційної бази (PostgreSQL) масивними операціями запису неструктурованих даних логування.

4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

4.1 Програмна реалізація серверного ядра та базового API

Програмна реалізація серверної частини (Backend) програмного комплексу «QuickSum» базується на використанні інноваційного середовища виконання Bun та сучасного web-фреймворку ElysiaJS. Вибір цього технологічного стеку обумовлений необхідністю забезпечення екстремально низьких мережевих затримок (low latency), оптимізації споживання системних ресурсів та впровадження надійних механізмів наскрізної типізації. На відміну від традиційного середовища Node.js, яке використовує рушій V8, Bun побудований на базі JavaScriptCore, що забезпечує значно вищу швидкість запуску (cold start) та підвищену пропускну здатність при обробці великої кількості одночасних HTTP-запитів.

4.1.1 Структурна організація коду проєкту

З метою забезпечення масштабованості, зручності підтримки та можливості паралельної розробки, проєкт має модульну ієрархічну структуру, спроєктовану за принципами чистої архітектури (Clean Architecture) та предметно-орієнтованого проектування (Domain-Driven Design). Увесь вихідний код, написаний мовою TypeScript, консолідовано в директорії src/, яка логічно розділена на наступні підсистеми:

- constants/ – директорія для зберігання глобальних констант, конфігураційних змінних та переліків, що використовуються по всьому застосунку;
- database/mikro-orm/ – конфігураційні файли об'єктно-реляційного відображення, міграції бази даних, схеми таблиць та налаштування підключення до PostgreSQL;

- modules/ – ключова директорія, де реалізовано бізнес-логіку застосунку. Вона розділена на ізольовані предметні області: ai (штучний інтелект), auth (автентифікація), paddle (обробка платежів) та user (керування профілем).
- utils/ – бібліотека загальних допоміжних функцій та інтеграцій. Тут розташовані інструменти криптографічного шифрування персональних даних (pii), управління чергами (bullmq/sending-emails), робота з поштою (nodemailer), обробка сесій (lusia), перевірка безпеки (hcaptcha, rule-engine) та робота з об'єктним сховищем (s3);
- index.ts – головний файл-точка входу (Entry Point), який виконує роль завантажувача (bootstrap). У ньому ініціалізується сервер, підключаються глобальні плагіни та встановлюються налаштування безпеки.

Варто відзначити внутрішню організацію кожної теки в modules/ (наприклад, модуля ai). Застосовано класичний патерн розділення відповідальності: файл index.ts виконує роль маршрутизатора (контролера), model.ts містить TypeBox-схеми та інтерфейси для валідації, а service.ts інкапсулює безпосередню бізнес-логіку та взаємодію з БД.

На рисунку 4.1 наведено структуру каталогів розробленого серверного застосунку, відкриту в середовищі розробки Visual Studio Code. Дана структура, разом із файлами конфігурації інфраструктури (Dockerfile, .dockerignore), дозволяє легко масштабувати застосунок у майбутньому.

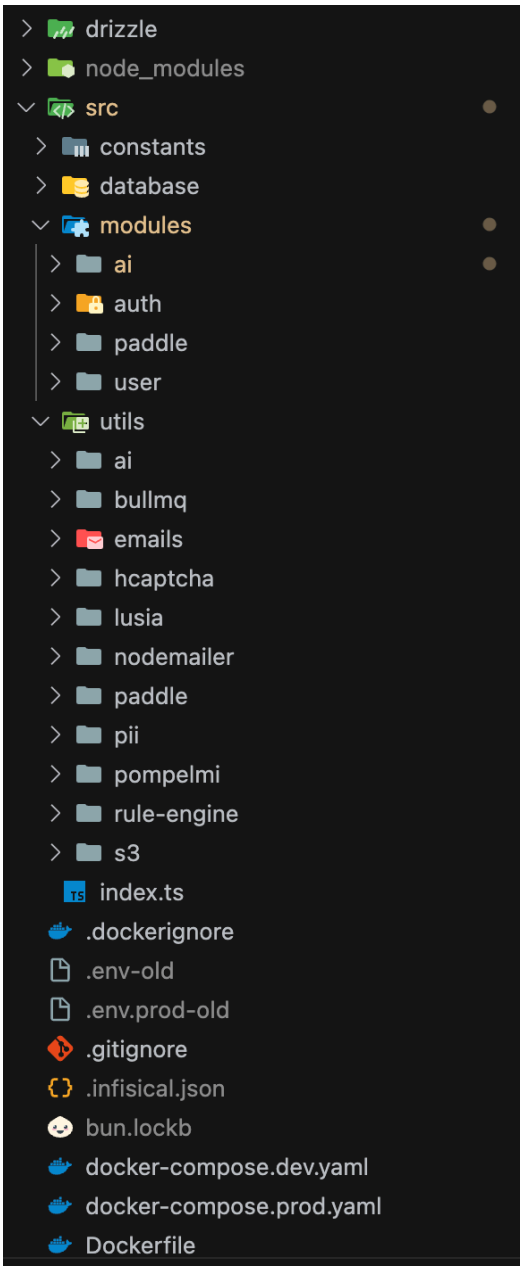


Рис. 4.1 Структура директорій серверної частини проєкту у середовищі розробки

4.1.2 Механізми наскрізної типізації та валідації вхідних даних

Однією з найвразливіших точок будь-якого web-застосунку є обробка даних, які надходять від клієнта. Традиційні серверні фреймворки вимагають ручного написання логіки перевірки наявності потрібних полів та їх форматів. Це не лише збільшує обсяг шаблонного коду, але й часто призводить до вразливостей або помилок виконання (Runtime Errors).

В архітектурі «QuickSum» цю проблему вирішено шляхом тісної інтеграції ElysiaJS із бібліотекою TypeBox. Валідація відбувається автоматично на етапі потрапляння запиту до маршрутизатора. У файлах model.ts кожного модуля декларативно описано схему очікуваних даних за стандартом JSON Schema. ElysiaJS компілює ці схеми у високопродуктивні функції перевірки. Якщо структура запиту від клієнта не відповідає схемі, система автоматично відхиляє запит зі статусом 422 Unprocessable Entity або 400 Bad Request, не доводячи невалідні дані до рівня бізнес-логіки (Service).

На рисунку 4.2 наведено приклад програмної реалізації такої суворої типізації для маршруту реєстрації.

```
// 2. Feature-specific Schemas
export const AuthSchemas = {
  registerEmailBody: t.Object({
    name: t.String({
      minLength: 3,
      maxLength: 25,
      examples: ["John Doe"],
      description: "User name",
      error: "Invalid name. Must be at least 3 characters long and at most 25 characters long.",
    }),
    email: t.String({
      format: "email",
      examples: ["[EMAIL_ADDRESS]"],
      description: "User email",
      error: "Invalid email",
    }),
    password: t.String({
      minLength: 8,
      maxLength: 32,
      examples: ["password"],
      description: "User password",
      error: "Invalid password. Must be at least 8 characters long and at most 32 characters long.",
    })
  }),
  registerSuccessResponse: t.Literal("User registered successfully. Please check your email to verify your account"),
```

Рис. 4.2 Приклад типізації за допомогою TypeBox

4.2 Реалізація серверної бази даних

Основним сховищем транзакційних даних, профілів користувачів та управління підписками в системі «QuickSum» виступає реляційна база даних PostgreSQL. Для забезпечення зручної та безпечної взаємодії з базою даних на

рівні коду інтегровано ORM-систему (Object-Relational Mapping). Вона дозволяє абстрагуватися від написання сирих SQL-запитів, забезпечуючи автоматичне маніпулювання даними через об'єктно-орієнтовані моделі та підтримуючи архітектурний патерн Unit of Work для надійного виконання транзакцій.

Підключення до бази даних ініціалізується автоматично під час запуску серверного ядра. Всі конфіденційні параметри (хост, порт, назва бази, ім'я користувача та пароль) передаються виключно через змінні оточення (Environment Variables), що унеможливлює їхній випадковий витік через репозиторій системи контролю версій.

Під час проектування бази даних було прийнято рішення відмовитися від зберігання персональної інформації у відкритому вигляді. Для візуального підтвердження дієвості розробленої архітектури було проведено інспекцію основної таблиці user за допомогою інструменту адміністрування баз даних DbGate. На рисунку 4.3 наведено стан бази даних після успішної реєстрації кількох користувачів.

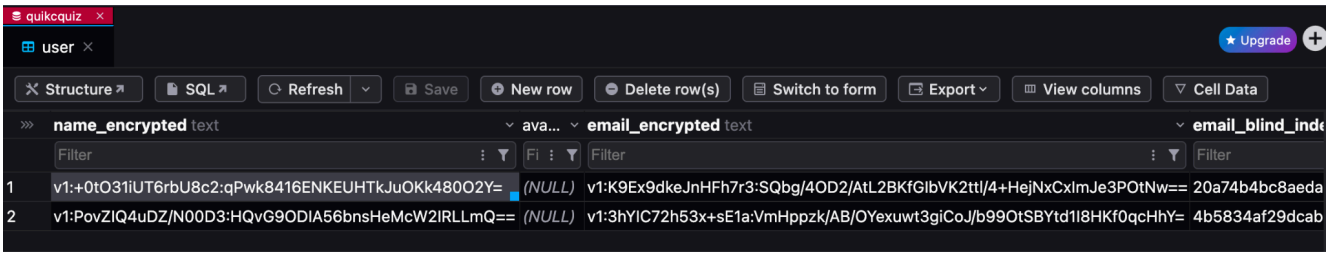


Рис. 4.3 Відображення таблиці користувачів у клієнті DbGate

Як видно з наведеного рисунка, замість звичних текстових даних, колонки nameEncrypted та emailEncrypted містять нечитабельні набори символів (вектори ініціалізації та шістнадцяткові шифротексти), а колонка emailBlindIndex містить уніфіковані хеші. Такий результат є наслідком впровадження комплексної системи криптографічного захисту, що розглядається у наступному підрозділі.

4.3 Реалізація механізмів криптографічного захисту персональних даних

Головною вимогою до розроблюваної інформаційної системи є відповідність суворим європейським стандартам захисту персональних даних (GDPR). Тому архітектура серверної частини базується на принципі «Secure by Design», що вимагає криптографічного перетворення будь-якої ідентифікованої інформації користувача (Personally Identifiable Information, PII) перед її записом на фізичні носії (базу даних).

Для досягнення цієї мети утилітарний модуль серверної частини реалізує два незалежні криптографічні механізми:

1. **Симетричне шифрування:** використовується алгоритм AES-256-GCM (Advanced Encryption Standard з режимом Galois/Counter Mode) для шифрування імен, електронних адрес та посилань на аватари.
2. **Сліпі індекси:** оскільки класичне шифрування генерує щоразу новий шифротекст через використання випадкового вектора ініціалізації (IV), прямий пошук користувача за електронною поштою стає неможливим без повного дешифрування всієї таблиці. Для вирішення цієї проблеми реалізовано алгоритм детермінованого хешування (HMAC-SHA256). Він генерує унікальний фіксований хеш для кожної пошти (Blind Index), що дозволяє базі даних миттєво знаходити записи.

На рисунку 4.4 наведено фрагмент коду утилітарного класу PII, який відповідає за реалізацію описаних криптографічних алгоритмів перед збереженням даних через ORM.

src > utils > pii > index.ts > PII

```ts
import crypto from "crypto";

const ENCRYPTION_KEY = Bun.env.ENCRYPTION_KEY!;
const BLIND_INDEX_SECRET = Bun.env.BLIND_INDEX_SECRET!;

export class PII {
  static async encrypt(text: string): Promise<string> {
    const iv = crypto.randomBytes(16);
    const cipher = crypto.createCipheriv("aes-256-gcm", Buffer.from(ENCRYPTION_KEY, "hex"), iv);

    let encrypted = cipher.update(text, "utf8", "hex");
    encrypted += cipher.final("hex");

    const authTag = cipher.getAuthTag().toString("hex");

    return `${iv.toString("hex")}:${authTag}:${encrypted}`;
  }

  static getBlindIndex(text: string): string {
    return crypto.createHmac("sha256", BLIND_INDEX_SECRET)
                 .update(text.toLowerCase().trim())
                 .digest("hex");
  }
}
```

Рис. 4.4 Відображення фрагмента коду утилітарного класу PII

Під час реєстрації, авторизації або оновлення профілю система автоматично викликає метод encrypt для полів імені та пошти, а метод getBlindIndex – виключно для пошти. Таким чином, у базу даних потрапляють лише результати криптографічних перетворень. Такий підхід повністю унеможливлює витік реальних електронних адрес та імен навіть у випадку несанкціонованого доступу зловмисників до файлової системи бази даних або крадіжки резервних копій.

4.4 Реалізація бізнес-логіки та інтеграція зовнішніх сервісів

Серцевиною застосунку «QuickSum» є модулі бізнес-логіки, що інкапсулюють процеси авторизації, обробки платежів та взаємодії зі штучним інтелектом. Надійність цих модулів досягається шляхом глибокої інтеграції зі сторонніми сервісами через офіційні Client SDK.

4.4.1 Інтеграція механізму авторизації (Lucia Auth та Google OAuth)

Для забезпечення безшовного входу користувачів реалізовано протокол OAuth 2.0. Коли клієнт обирає вхід через Google, сервер валідує параметри code та state (механізм захисту від CSRF-атак). Отримавши дані профілю, система генерує сліпий індекс електронної пошти та шукає користувача у БД. Завершальним етапом є створення криптографічно підписаної сесії за допомогою екосистеми Lucia Auth. Діаграма послідовностей OAuth-колбеку наведено нижче:

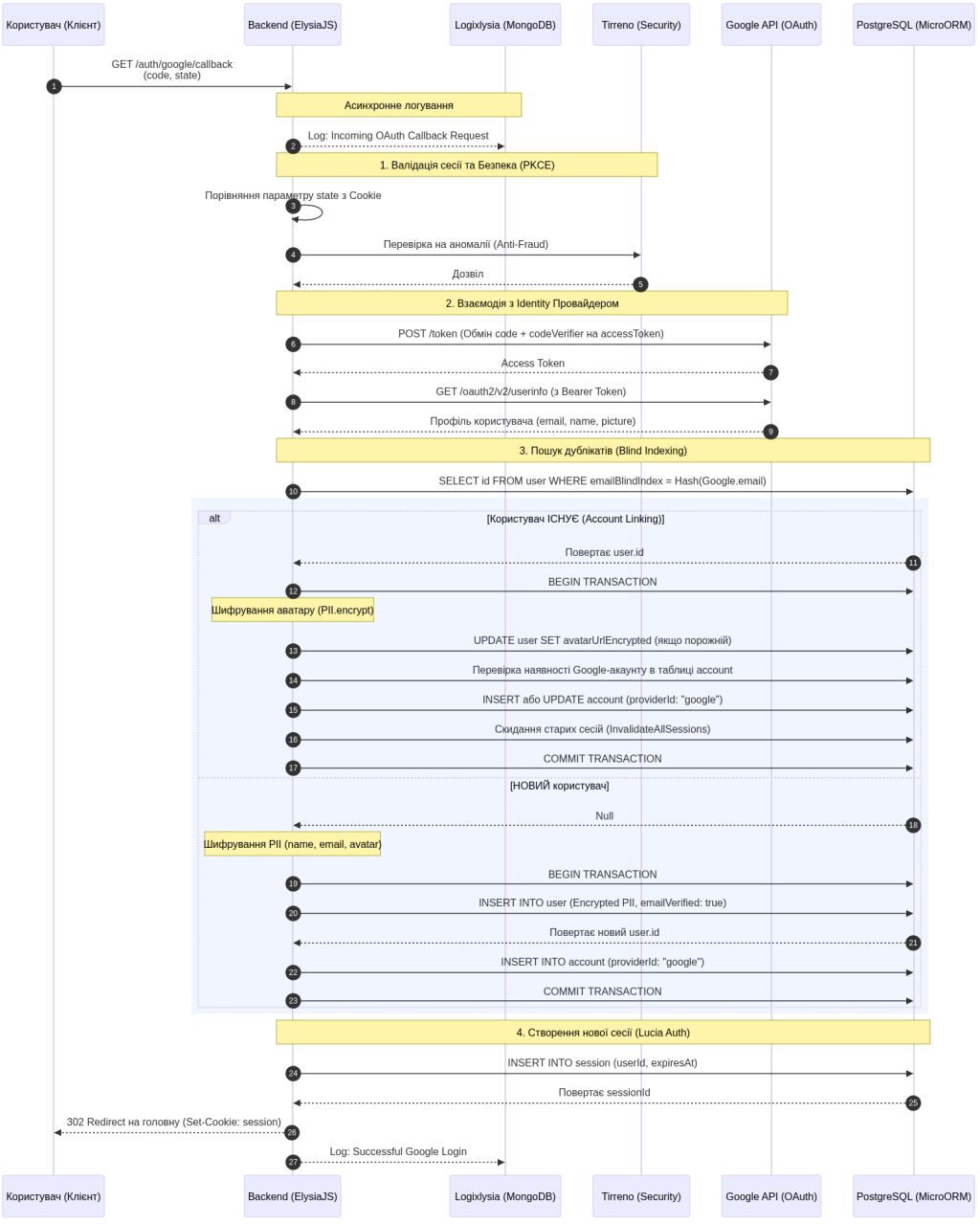


Рис. 4.5 Діаграма послідовностей OAuth-колбеку

4.4.2 Алгоритм роботи AI-модуля та глобальне кешування

Для обробки текстів сервер використовує єдиний інтерфейс маршрутизації до великих мовних моделей (LLM) – сервіс OpenRouter. Взаємодія з ним відбувається за допомогою власного SDK, конфігурованого на кастомний baseURL.

З метою мінімізації фінансових витрат на токени (API-виклики) та екстремального зменшення часу відповіді, у бізнес-логіку AI-модуля інтегровано алгоритм дедуплікації (кешування). Перед формуванням запиту до ШІ сервер генерує SHA-256 хеш вхідного тексту. Якщо такий текст вже оброблявся раніше іншим користувачем, система фіксує «Cache Hit» і миттєво повертає збережений результат з PostgreSQL. Лише у разі «Cache Miss» застосовується промпт-інжиніринг (Prompt Engineering) і текст відправляється до OpenRouter. Діаграма послідовностей запиту на скорочення тексту наведено нижче:

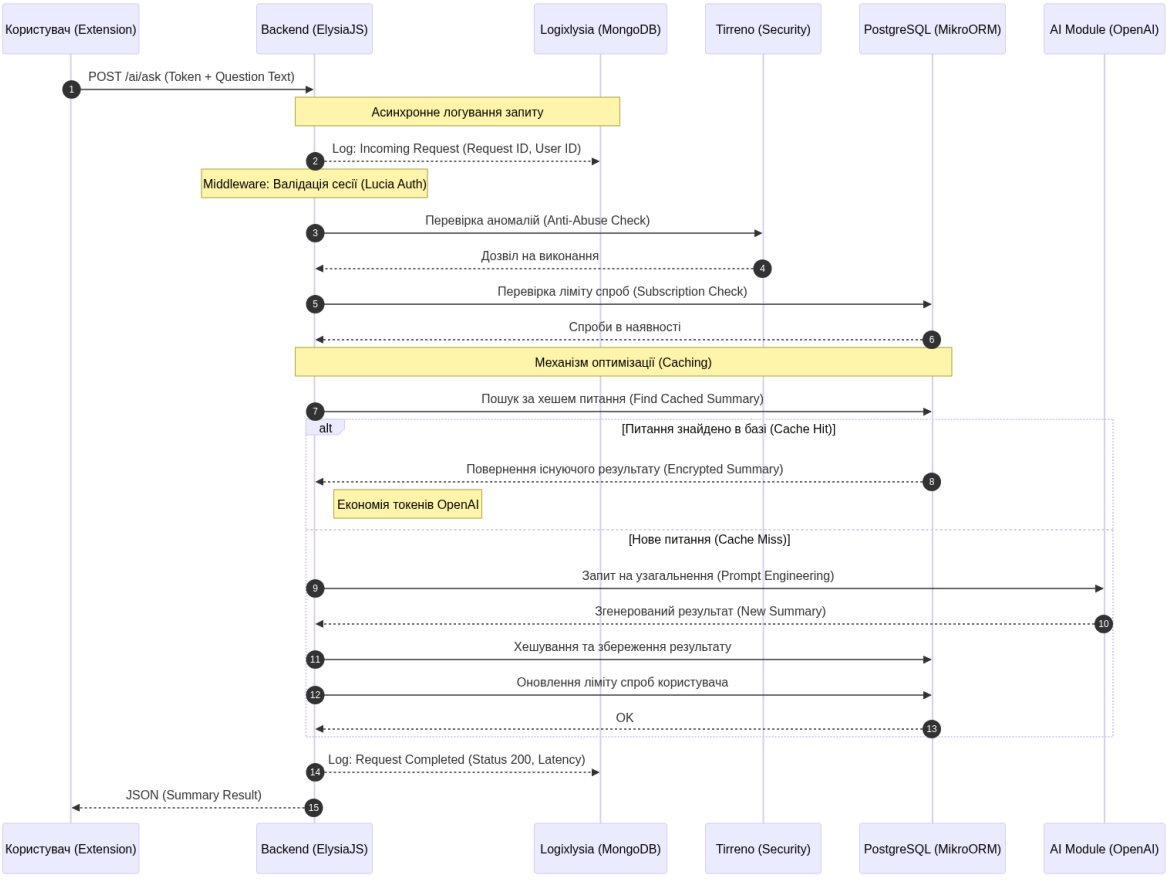


Рис. 4.6 Діаграма послідовностей запиту на скорочення тексту

4.4.3 Інтеграція платіжного шлюзу Paddle

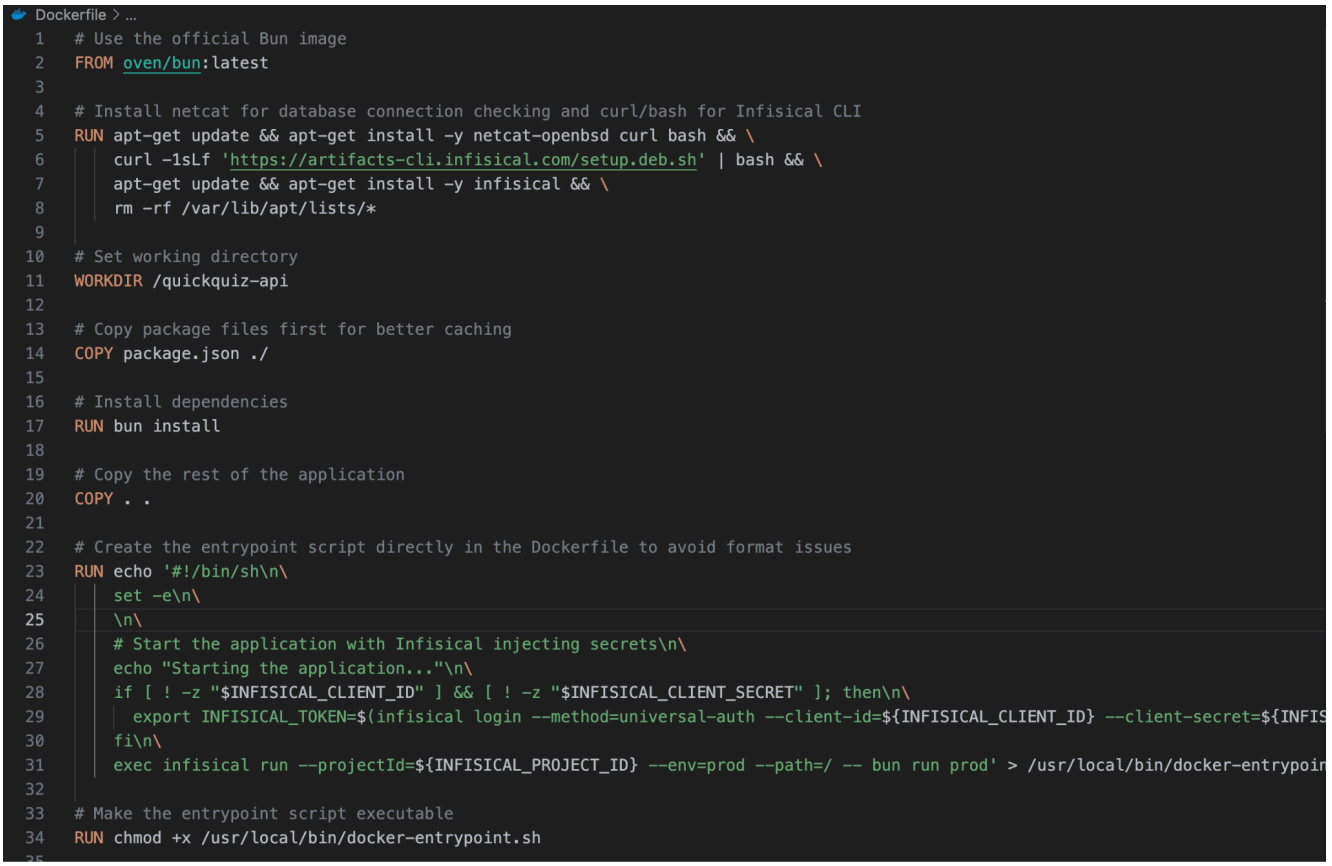
Управління лімітами на генерацію тексту та життєвим циклом підписок базується на платформі еквайрингу Paddle. Для безпечної синхронізації статусів оплат сервер обробляє вхідні вебхуки (webhooks) від сервісів Paddle. Критично важливим етапом цієї інтеграції є верифікація криптографічного підпису вебхука за допомогою Paddle SDK. Це гарантує, що запит надійшов саме від платіжної системи, а не від зловмисників. У разі успішної верифікації система оновлює баланс доступних спроб (triesLeft) та статус підписки клієнта.

4.5 Розгортання інфраструктури та підсистеми спостережуваності

Для забезпечення надійного, безпечного та безперебійного функціонування серверної частини застосунку «QuickSum» у промисловому середовищі (Production) було обрано архітектурний підхід Self-hosting. Даний підхід дозволяє отримати повний контроль над апаратно-програмною інфраструктурою, оптимізувати витрати на оренду серверних потужностей та уникнути жорсткої прив'язки (vendor lock-in) до конкретних хмарних провайдерів.

4.5.1 Контейнеризація серверного застосунку

Фундаментом розгортання системи є технологія контейнеризації Docker. Вона гарантує, що застосунок працюватиме абсолютно ідентично як на машині розробника, так і на бойовому сервері, незалежно від операційної системи хоста. Серверне ядро упаковується в ізольований контейнер на базі офіційного мінімалістичного образу середовища виконання Bun (oven/bun:latest). Процес збірки оптимізовано для зменшення кінцевого розміру образу та пришвидшення його запуску.



```
Dockerfile > ...
1   # Use the official Bun image
2   FROM oven/bun:latest
3
4   # Install netcat for database connection checking and curl/bash for Infisical CLI
5   RUN apt-get update && apt-get install -y netcat-openbsd curl bash && \
6       curl -1sLf 'https://artifacts-cli.infisical.com/setup.deb.sh' | bash && \
7       apt-get update && apt-get install -y infisical && \
8       rm -rf /var/lib/apt/lists/*
9
10  # Set working directory
11  WORKDIR /quickquiz-api
12
13  # Copy package files first for better caching
14  COPY package.json ./
15
16  # Install dependencies
17  RUN bun install
18
19  # Copy the rest of the application
20  COPY . .
21
22  # Create the entrypoint script directly in the Dockerfile to avoid format issues
23  RUN echo '#!/bin/sh\n\
24      set -e\n\
25      \n\
26      # Start the application with Infisical injecting secrets\n\
27      echo "Starting the application..."\n\
28      if [ ! -z "$INFISICAL_CLIENT_ID" ] && [ ! -z "$INFISICAL_CLIENT_SECRET" ]; then\n\
29        export INFISICAL_TOKEN=$(infisical login --method=universal-auth --client-id=${INFISICAL_CLIENT_ID} --client-secret=${INFIS
30      fi\n\
31      exec infisical run --projectId=${INFISICAL_PROJECT_ID} --env=prod --path=/ -- bun run prod' > /usr/local/bin/docker-entrypoin
32
33  # Make the entrypoint script executable
34  RUN chmod +x /usr/local/bin/docker-entrypoint.sh
```

Рис. 4.7 Файл контейнеризації Docker

4.5.2 Оркестрація та безперервне розгортання (CI/CD)

Управління життєвим циклом Docker-контейнерів здійснюється за допомогою платформи оркестрації Coolify – потужної open-source альтернативи комерційним PaaS-рішенням. Coolify автоматизує процес безперервної інтеграції та доставки (CI/CD): при злитті нового коду (Push) у головну гілку репозиторію GitHub, платформа автоматично завантажує оновлення, виконує збірку контейнера без зупинки поточних процесів (Zero-Downtime Deployment) та керує внутрішнім зворотним проксі-сервером (Traefik) для автоматичного випуску та оновлення SSL-сертифікатів (Let's Encrypt).

На рисунку 4.8 наведено скріншот панелі керування Coolify, де відображено активний статус роботи серверних контейнерів проєкту.

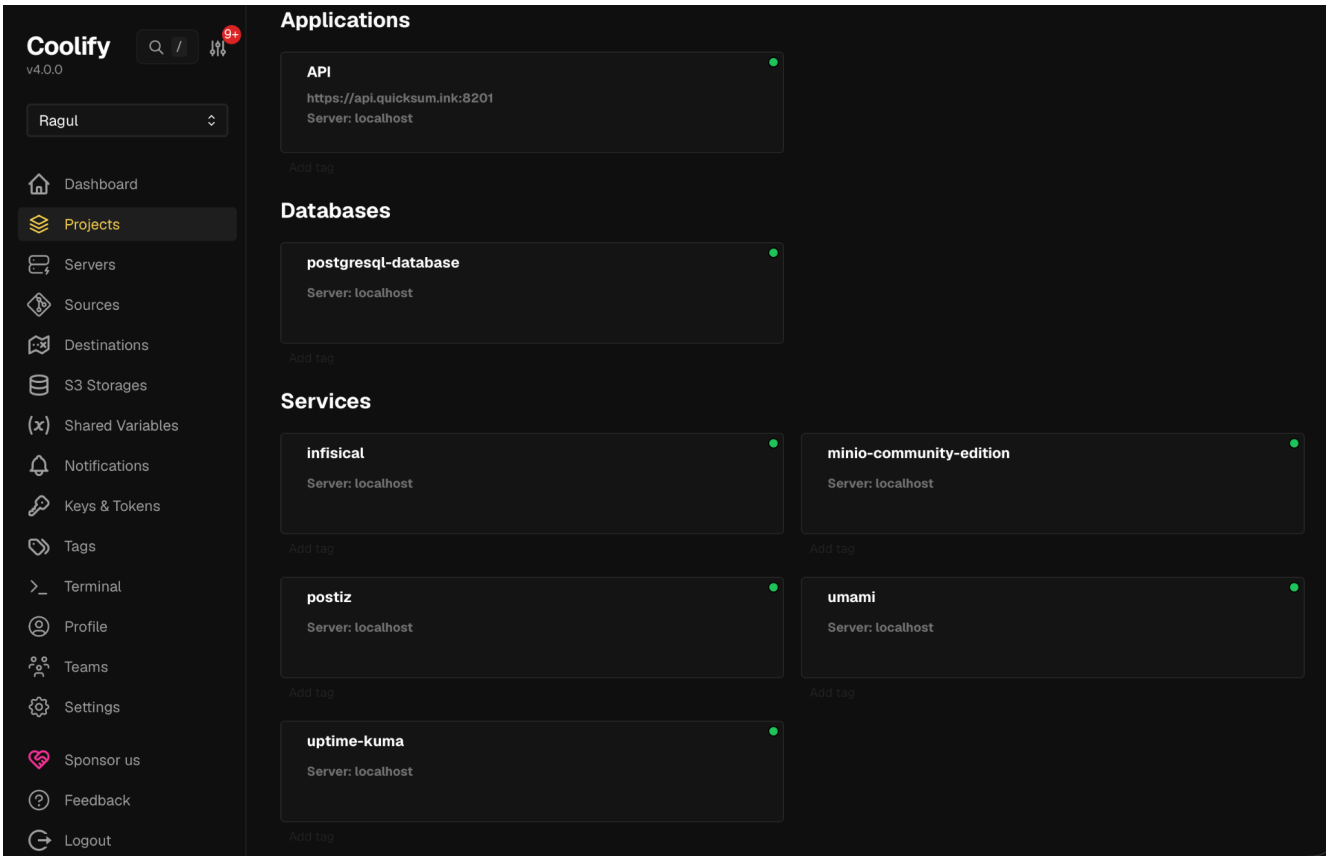


Рис. 4.8 Панель керування оркестратора Coolify зі статусом розгорнутих сервісів

4.5.3 Управління секретами та об'єктне сховище

Для забезпечення найвищого рівня кібербезпеки, конфігураційні змінні та секрети (ключі шифрування PII, токени OpenAI, Paddle, реквізити доступу до баз даних) не зберігаються у звичайних файлах .env на файловій системі сервера. Замість цього розгорнуто платформу управління секретами Infisical. Вона інтегрована безпосередньо в CI/CD пайплайн і безпечно ін'єктує (підставляє) секрети в середовище виконання контейнера виключно під час його запуску (у оперативну пам'ять).

На рисунку 4.9 наведено скріншот панелі Infisical, де відображено активний статус роботи серверних контейнерів проєкту.

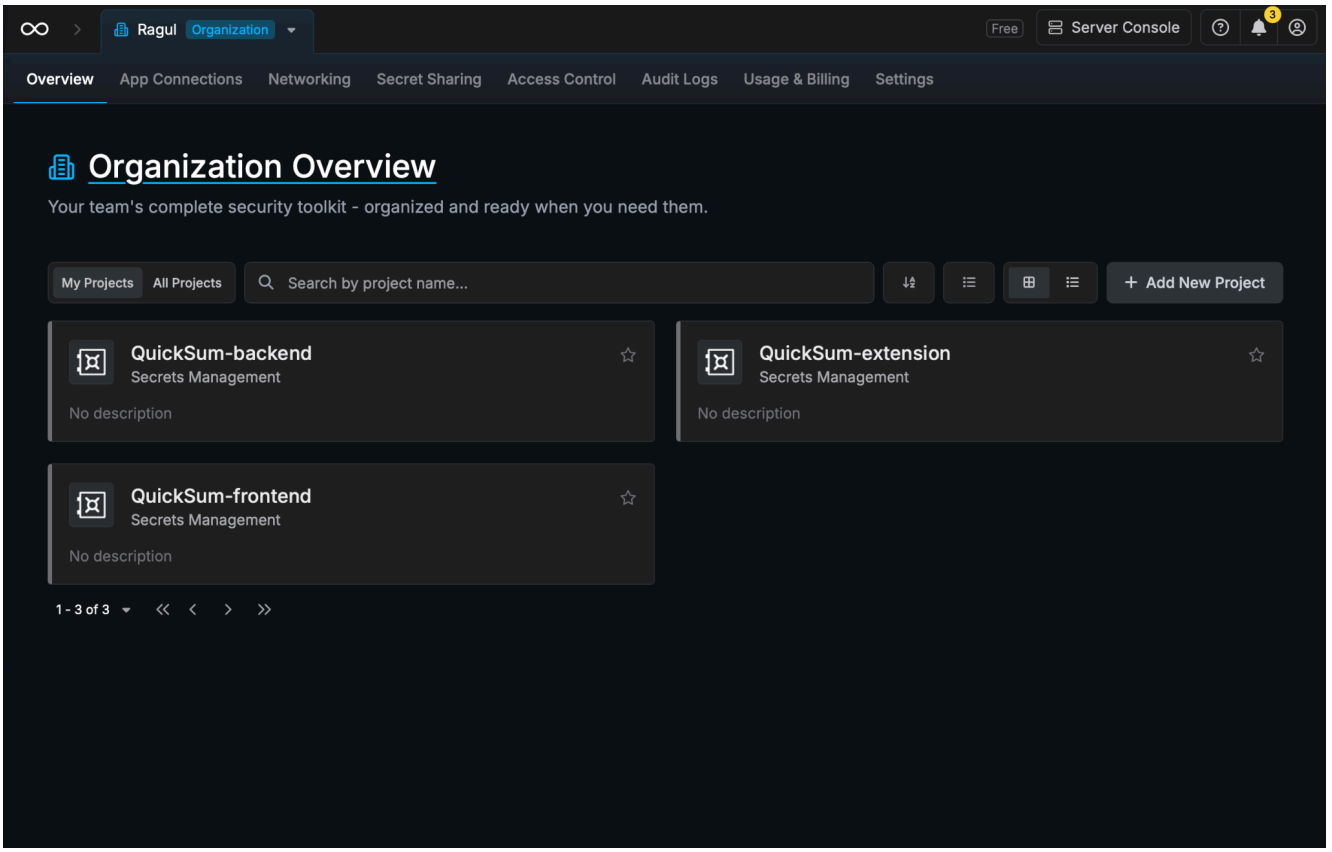


Рис. 4.9 Фрагмент коду модульних тестів

Додатково, в ізольованій віртуальній мережі (Docker Network), розгорнуто MinIO – високопродуктивне S3-сумісне об'єктне сховище. Воно використовується для надійного збереження статичних медіафайлів (зокрема, зашифрованих аватарів користувачів), знімаючи навантаження з основної реляційної бази даних PostgreSQL.

4.6 Тестування та оцінка продуктивності програмного забезпечення

Заключним та одним із найважливіших етапів життєвого циклу розробки програмного забезпечення є тестування. Метою цього етапу є перевірка відповідності розробленої серверної частини (Backend) застосунку «QuickSum» початковим функціональним та нефункціональним вимогам, а також виявлення та усунення можливих дефектів і вузьких місць у продуктивності. Для забезпечення високої надійності системи було застосовано комплексний підхід, що включає модульне, функціональне та навантажувальне тестування

4.6.1 Модульне тестування

Модульне тестування спрямоване на ізольовану перевірку окремих функцій, класів та методів бізнес-логіки. Оскільки проект базується на середовищі Bun, було прийнято рішення відмовитися від сторонніх фреймворків для тестування (таких як Jest або Mocha) на користь вбудованого інструменту bun test. Цей інструмент демонструє значно вищу швидкість виконання тестів та має нативну підтримку TypeScript.

Особлива увага під час модульного тестування приділялася утилітарним класам, що відповідають за кібербезпеку та управління лімітами. Наприклад, було створено набір тестів для класу криптографічного шифрування персональних даних (PII). Метою тестів було довести, що:

1. Алгоритм AES-256-GCM коректно шифрує рядок і успішно його дешифрує (консистентність).
2. Шифрування одного й того ж тексту двічі дає різний шифротекст (завдяки випадковому вектору ініціалізації IV), що унеможливлює частотний криптоаналіз.
3. Алгоритм «сліпих індексів» (HMAC-SHA256) є детермінованим і завжди повертає однаковий хеш для однакових email-адрес (навіть якщо вони написані в різних регістрах).

На рисунку 4.10 наведено скріншот фрагменту коду написаних модульних тестів.

src > tests > pii.test.ts > describe("Cryptography & PII Module") callback

```ts
import { expect, test, describe } from "bun:test";
import { PII } from "~/utils/pii";

describe("Cryptography & PII Module", () => {
  const sampleEmail = "User@Example.com";

  test("Blind index should be deterministic and ignore case", () => {
    const hash1 = PII.getBlindIndex(sampleEmail);
    const hash2 = PII.getBlindIndex("user@example.com");
    expect(hash1).toBe(hash2);
    expect(hash1).toBeTypeOf("string");
  });

  test("Encryption AES-256-GCM generates unique ciphertexts", async () => {
    const encrypted1 = await PII.encrypt("My Secret Name");
    const encrypted2 = await PII.encrypt("My Secret Name");
    expect(encrypted1).not.toBe(encrypted2);
  });

  test("Decryption AES-256-GCM returns original text", async () => {
    const originalText = "My Secret Name";
    const encrypted = await PII.encrypt(originalText);
    const decrypted = await PII.decrypt(encrypted);
    expect(decrypted).toBe(originalText);
  });
});
```

Рис. 4.10 Фрагмент коду модульних тестів

4.6.2 Функціональне тестування REST API

Функціональне тестування проводилося з метою перевірки коректності роботи кінцевих точок (ендпоінтів) серверного API, правильності обробки вхідних даних та генерації відповідних HTTP статус-кодів. Для виконання цього завдання використовувався сучасний консольний HTTP-клієнт HTTPie, який дозволяє зручно формувати запити, управляти заголовками та аналізувати форматовані JSON-відповіді безпосередньо в терміналі.

Під час тестування перевірялися наступні ключові сценарії:

1. **Успішна авторизація**: надсилання валідних облікових даних користувача на ендпоінт автентифікації. Перевірка повернення статусу 200 OK та коректного встановлення захищених HttpOnly файлів cookie екосистемою Lucia Auth.
2. **Відмова у доступі**: імітація запиту до захищеного ресурсу (наприклад, до профілю або ендпоінту генерації тексту) без надання валідного сесійного токена. Система успішно ідентифікувала відсутність доступу та відхиляла запити з кодом 401 Unauthorized.
3. **Валідація вхідних даних**: навмисне надсилання некоректного JSON-тіла (наприклад, занадто короткого тексту або невідомого режиму узагальнення) на ендпоінт /ai/ask. Фреймворк ElysiaJS автоматично блокував такі запити ще до етапу виконання бізнес-логіки, повертаючи детальний опис помилки з кодом 400 Bad Request.

На рисунку 4.11 наведено приклад успішного функціонального тестування ендпоінту взяття інформації користувача з використанням інструменту HTTPie.

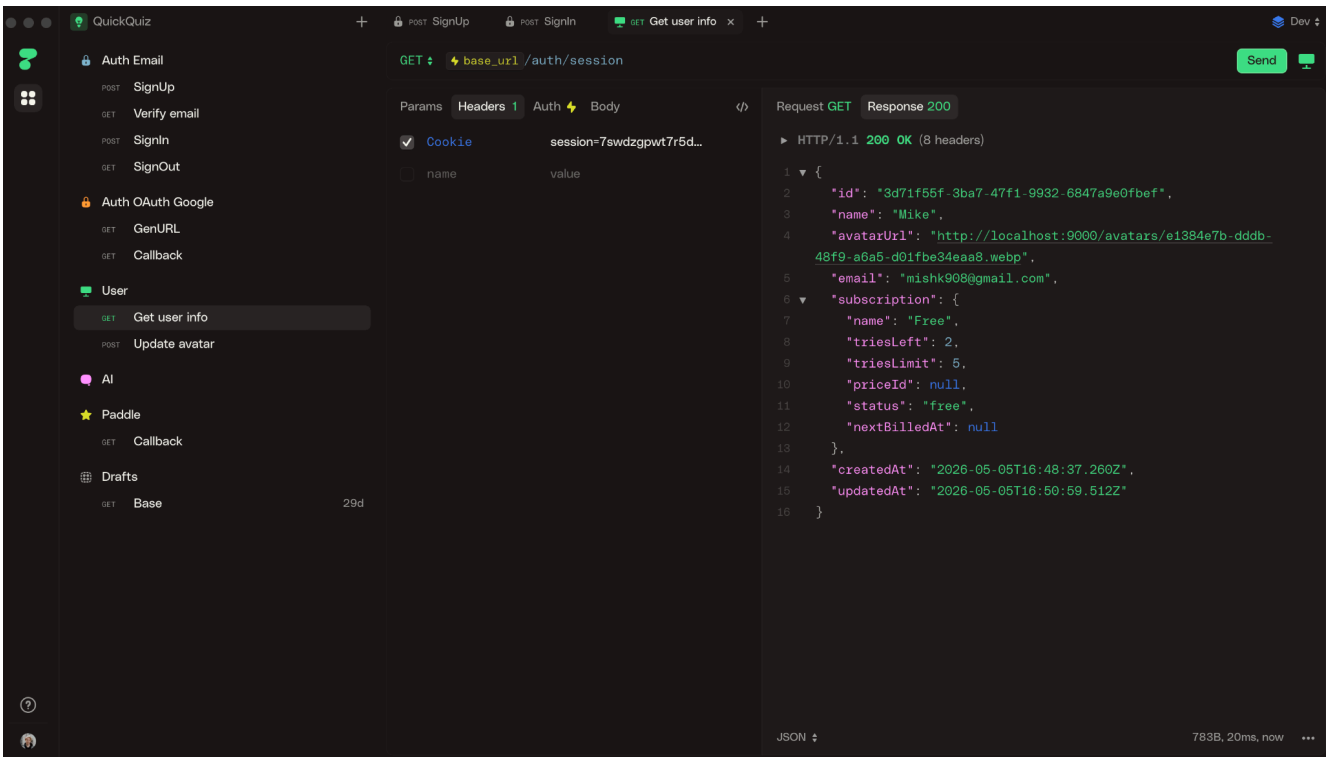


Рис. 4.11 Функціональне тестування API через консольний клієнт HTTPie

4.6.3 Навантажувальне тестування та оцінка продуктивності

Ключовою архітектурною гіпотезою при виборі технологічного стеку Bun + ElysiaJS було суттєве зниження мережевих затримок та забезпечення високої пропускної здатності системи під навантаженням. Для підтвердження цієї гіпотези та доведення ефективності обраних рішень було проведено серію навантажувальних тестувань за допомогою спеціалізованих інструментів k6.

Методика тестування передбачала імітацію конкурентного доступу великої кількості одночасних віртуальних користувачів. Було згенеровано інтенсивний потік запитів до найбільш ресурсоємного ендпоінту — обробки тексту. Основною метою було перевірити, як сервер витримує високі навантаження та як швидко обробляється алгоритм пошуку дедуплікатів у базі даних.

Результати тестування повністю підтвердили високу продуктивність системи та перевершили очікування, притаманні традиційним Node.js-рішенням:

1. **Безвідмовність**: Сервер не продемонстрував жодного відхиленого запиту внаслідок перевантаження процесора або вичерпання оперативної пам'яті.
2. **Ефективність кешування**: Завдяки впровадженню алгоритмів глобального кешування, медіанний час відповіді для текстів, які вже оброблялися раніше іншими користувачами, склав лічені мілісекунди (<50 мс).
3. **Оптимізація AI-запитів**: Навіть при первинному повноцінному зверненні до зовнішнього API штучного інтелекту, оптимізований роутинг ElysiaJS та середовище Bun дозволили утримати загальний час затримки генерації тексту в межах суворої нефункціональної вимоги — менше ніж 10 секунд.

На рисунку 4.12 наведено графік результатів навантажувального тестування.

```
HTTP
http_req_duration.......................: avg=17.11ms  min=114µs   med=377µs    max=489.61ms p(90)=31.66ms  p(95)=160.34ms
  { expected_response:true }............: avg=55.49ms  min=114µs   med=558µs    max=489.61ms p(90)=171.21ms p(95)=189.39ms
  { group:::Auth - google-callback }...: avg=414.2µs  min=168µs   med=297µs    max=6.95ms   p(90)=488.2µs  p(95)=765.8µs
  { group:::Auth - google-url }.........: avg=523.84µs min=239µs   med=405µs    max=3.74ms   p(90)=581.2µs  p(95)=884.39µs
  { group:::Auth - root }...............: avg=708.35µs min=114µs   med=453µs    max=11.4ms   p(90)=1.5ms    p(95)=1.8ms
  { group:::Auth - session }............: avg=280.16µs min=123µs   med=219µs    max=4.2ms    p(90)=339.6µs  p(95)=423.59µs
  { group:::Auth - sign-in }............: avg=1.86ms   min=836µs   med=1.59ms   max=24.66ms  p(90)=3.1ms    p(95)=3.76ms
  { group:::Auth - sign-out }...........: avg=340.68µs min=150µs   med=246µs    max=4.21ms   p(90)=392.6µs  p(95)=555.79µs
  { group:::Auth - sign-up }............: avg=165.23ms min=94.66ms med=160.37ms max=489.61ms p(90)=200.38ms p(95)=214.19ms
  { group:::Auth - verify-email }.......: avg=1.04ms   min=462µs   med=876µs    max=6.15ms   p(90)=1.54ms   p(95)=2.63ms
  { group:::Paddle - webhook }.........: avg=364.36µs min=172µs   med=284µs    max=3.68ms   p(90)=435.6µs  p(95)=518.19µs
  { group:::User - upload-avatar }......: avg=375.43µs min=171µs   med=291µs    max=5.82ms   p(90)=435µs    p(95)=539.99µs
http_req_failed.........................: 70.00% 5495 out of 7850
http_reqs...............................: 7850   65.00378/s
```

Рис. 4.12 Результати навантажувального тестування продуктивності серверної частини

ВИСНОВКИ

Досліджено сучасні підходи до автоматизованої обробки та узагальнення текстової інформації у web-середовищі. Виявлено типові проблеми існуючих SaaS-рішень, серед яких: високі мережеві затримки при взаємодії з великими мовними моделями, відсутність алгоритмів глобального кешування запитів, штучні обмеження швидкості та вразливість конфіденційної інформації користувачів.

Визначено технічні бар'єри побудови високонавантажених AI-сервісів. Сформульовано функціональні та нефункціональні вимоги до серверної частини застосунку, які охоплюють екстремальну продуктивність, масштабованість, захищеність персональних даних та оптимізацію фінансових витрат на зовнішні API.

Обґрунтовано вибір технологій: інноваційного середовища виконання Bun та фреймворку ElysiaJS для реалізації серверної логіки з наскрізною типізацією, PostgreSQL (через MikroORM) — як надійної реляційної СУБД для транзакцій, та MongoDB для підсистеми спостережуваності. Вибраний технологічний стек дозволяє створити ефективну, розширювану та стійку до зломів систему.

Спроєктовано й реалізовано структуру бази даних, яка охоплює сутності користувачів, сесій, підписок та історії генерацій. Забезпечено логіку взаємодії зі штучним інтелектом, безпечну автентифікацію через Lucia Auth та Google OAuth, алгоритми дедуплікації тексту, а також інтеграцію з платіжним провайдером Paddle. Забезпечено найвищий рівень конфіденційності завдяки впровадженню симетричного шифрування PII-даних та механізму «сліпих індексів».

Проведено тестування серверного застосунку (модульне, функціональне та навантажувальне), під час якого перевірено відповідність реалізованого функціоналу поставленим вимогам.

Здійснено оцінку стабільності: доведено, що архітектура забезпечує безвідмовну роботу під навантаженням із медіанним часом відгуку при кешуванні

менше ніж 50 мілісекунд та загальною затримкою генерації ШІ до 8 секунд.

Оцінено високий потенціал застосунку до комерційного впровадження як повноцінного SaaS-продукту. Запропоновано напрями подальшого вдосконалення, зокрема розширення пулу підтримуваних LLM-моделей, додавання векторних баз даних для пошуку по контексту та подальше масштабування серверної інфраструктури за допомогою кластеризації.

Робота пройшла апробацію. За її результатами було опубліковано наступні тези доповідей:

1. Карманний М.Ю., Залива В.В. Визначення вимог до серверної частини для клієнтському застосунку для обробки та узагальнення текстової інформації з web-ресурсів. Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». 23 квітня 2026 року, Державний університет інформаційно-комунікаційних технологій, Київ. Збірник тез. К.: ДУІКТ. С. 420-422.

2. Карманний М.Ю., Залива В.В. Механізми забезпечення конфіденційності та безпеки даних у клієнтському застосунку для обробки та узагальнення текстової інформації з web-ресурсів. Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». 23 квітня 2026 року, Державний університет інформаційно-комунікаційних технологій, Київ. Збірник тез. К.: ДУІКТ. С. 245-246.

ПЕРЕЛІК ПОСИЛАНЬ

1. Roetzel, P. G. Information overload in the information age: a review of the literature. International Review of Applied Economics. 2019. 362 p.
2. Sarmad, M. N., et al. A Survey of Text Summarization Techniques for Online Social Networks. IEEE Access. 2020. 583 p.
3. Summarizer.org. Free Text Summarization Tool. URL: https://summarizer.org/ (дата звернення 01.03.2026).
4. Smart Summarize. URL: https://smartsummarize.io/ (дата звернення 03.03.2026).
5. Summarize AI. Chrome Web Store. URL: https://chromewebstore.google.com/detail/summarize-ai/pcndedfdmllnllkgbohdhhjoeigbgmod (дата звернення 05.03.2026).
6. AISMRY. AI Text Summarizer. URL: https://aismry.com/ (дата звернення 07.03.2026).
7. Visual Studio Code. The open source code editor. URL: https://code.visualstudio.com/ (дата звернення 08.03.2026).
8. DbGate. SQL & noSQL database manager. URL: https://www.dbgate.io/ (дата звернення 10.03.2026).
9. HTTPie. API testing client that flows with you. URL: https://httpie.io/ (дата звернення 11.03.2026).
10. OrbStack. Fast, light, simple Docker & Linux. URL: https://orbstack.dev/ (дата звернення 12.03.2026).
11. TypeScript. JavaScript With Syntax For Types. URL: https://www.typescriptlang.org/ (дата звернення 15.03.2026).
12. Bun. A fast all-in-one JavaScript runtime. URL: https://bun.com/ (дата звернення 18.03.2026).

13. Elysia. Ergonomic Framework for Humans. URL: https://elysiajs.com/ (дата звернення 20.03.2026).
14. MikroORM. TypeScript ORM built on proven patterns. URL: https://mikro-orm.io/ (дата звернення 22.03.2026).
15. Pompelmi. Node.js antivirus scanning via ClamAV. URL: https://pompelmi.app/ (дата звернення 24.03.2026).
16. Sharp. High performance Node.js image processing. URL: https://sharp.pixelplumbing.com/ (дата звернення 26.03.2026).
17. Arctic. URL: https://arcticjs.dev/ (дата звернення 28.03.2026).
18. PostgreSQL. The world's most advanced open source database. URL: https://www.postgresql.org/ (дата звернення 29.03.2026).
19. MongoDB. The World’s Leading Modern Data Platform. URL: https://www.mongodb.com/ (дата звернення 30.03.2026).
20. Coolify. Deploy apps, databases & 280+ services to your server. URL: https://coolify.io/ (дата звернення 07.04.2026).
21. MinIO. S3 Compatible, Exascale Object Store. URL: https://www.min.io/ (дата звернення 10.04.2026).
22. Infisical. Platform for secrets, certificates, and privileged access management. URL: https://infisical.com/ (дата звернення 20.04.2026).
23. OpenRouter. One API for Any Model. URL: https://openrouter.ai/ (дата звернення 23.03.2026).
24. Paddle. Subscriptions, Payments & Tax for SaaS & Apps. URL: https://www.paddle.com/ (дата звернення 24.04.2026).

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка клієнтського застосунку для обробки та узагальнення текстової інформації з web-ресурсів.
Спец-частина: розробка Backend-частини застосунку

Виконав студент 4 курсу групи ПД-42
Карманний Михайло Юрійович
Керівник роботи
доцент кафедри ІПЗ, доктор філософії (PhD) Залива Віталій Вікторович

Київ – 2026

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: оптимізація процесу опрацювання великих обсягів контенту користувачами на основі автоматизованого захоплення та інтелектуального узагальнення текстової інформації з web-ресурсів із використанням моделей штучного інтелекту. Завдяки пришвидшенню цього процесу.

Об'єкт дослідження: процес автоматизованої обробки, аналізу та узагальнення текстової інформації в сучасних інформаційних web-системах.

Предмет дослідження: програмні засоби та технології розробки серверної частини застосунку для взаємодії з клієнтським інтерфейсом, інтеграції AI-моделей та забезпечення безпеки бізнес-логіки.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Провести аналіз літературних джерел з проблеми інформаційного перевантаження та автоматизованої обробки текстової інформації, визначити вплив великих мовних моделей на процеси узагальнення тексту, оглянути проблеми безпеки даних та мережевих затримок в наявних системах.

2. Здійснити огляд та аналіз наявних web-сервісів та браузерних розширень для автоматизованої сумаризації тексту, визначити їх переваги та ключові недоліки.

3. Провести огляд сучасних засобів розробки серверного програмного забезпечення, систем управління базами даних та технологій криптографічного захисту інформації.

4. Сформувати функціональні та нефункціональні вимоги до розробки серверної частини клієнт-серверного застосунку для узагальнення тексту.

5. Спроєктувати архітектуру серверної частини застосунку, включаючи структуру реляційної бази даних, механізми безпеки (GDPR), пайплайн взаємодії із зовнішніми API (OpenAI, платіжні шлюзи) та алгоритми дедуплікації запитів.

6. Розробити серверну частину застосунку для узагальнення текстової інформації з імплементацією механізмів оптимізації та криптографічного шифрування персональних даних.

7. Провести функціональне та навантажувальне тестування розробленої серверної частини, оцінити продуктивність (мінімізацію затримок) та надійність алгоритмів захисту.

3

АНАЛІЗ АНАЛОГІВ

Характеристика	Summarizer.org	Smartsummarize.io	Summarize AI	AISMRY	QuickSum
Наявність браузерного розширення	Немає	Немає	Є	Є	Є
Декілька режимів узагальнення тексту	Довге Коротке	Довге Середнє Коротке	Лише одне	Лише одне	Середнє Коротке
Узагальнення всієї сторінки	Ні	Ні	Так	Так	Так
Узагальнення виділеного тексту	Так	Так	Ні	Ні	Так
Оперативність обробки даних	±5сек	>40сек	>1хв	±8сек	<10сек
Тип монетизації	Freemium	Free	Premium	Free	Free/ Subscription
Робота з історією запитів	Ні	Так	Ні	Ні	Так

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги до системи

1. **Управління користувачами та сесіями:** забезпечення безпечної реєстрації, автентифікації та виходу з системи. Підтримка механізму Single Sign-On (OAuth 2.0) через акаунти Google із можливістю автоматичного об'єднання акаунтів (Account Linking).
2. **Обробка та узагальнення тексту:** прийом масивів текстових даних від клієнтського браузерного розширення та взаємодія із зовнішніми API великих мовних моделей для генерації сумаризації.
3. **Глобальне кешування запитів:** реалізація алгоритму дедуплікації. Система повинна генерувати криптографічний хеш вхідного тексту і, у разі його наявності в базі даних, миттєво повертати збережений результат без повторного звернення до платного API ШІ.
4. **Управління підписками:** інтеграція з платіжним провайдером (Paddle) для обробки вебхуків, автоматичного оновлення статусів підписок та контролю лімітів на використання AI-модуля.
5. **Криптографічний захист:** автоматичне шифрування персональних даних (PII) у базі даних та пошук користувачів за допомогою детермінованих «сліпих індексів».

Нефункціональні вимоги до системи

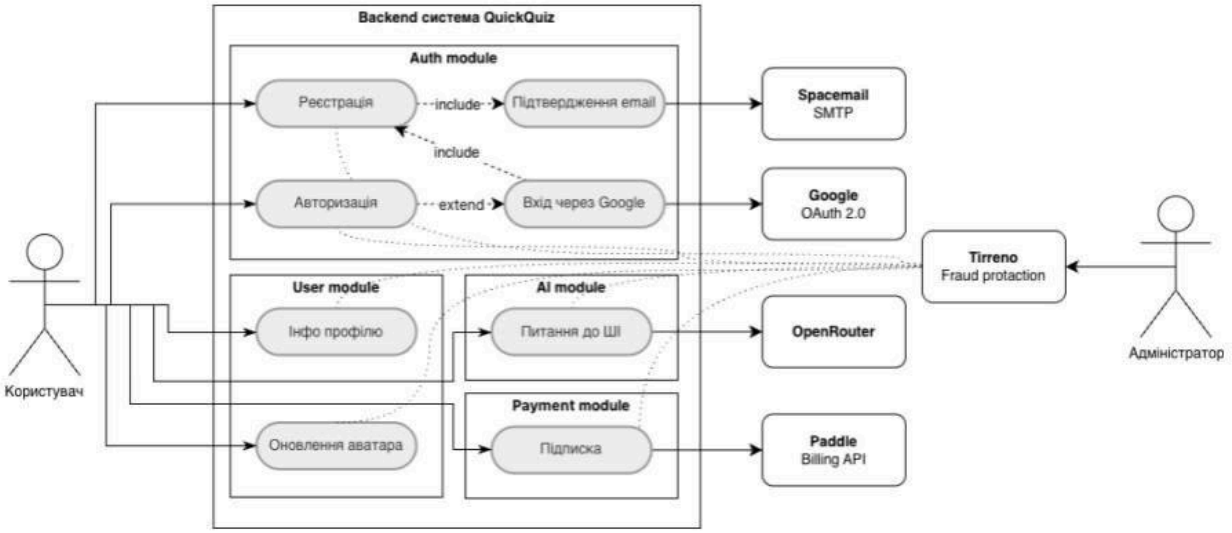
1. **Продуктивність та низька затримка:** серверна архітектура повинна обробляти базові HTTP-запити з мінімальними затримками. Використання середовища Bun та фреймворку ElysiaJS має забезпечити середній час виконання серверної бізнес-логіки до <8 секунд (з урахуванням часу відповіді від LLM).
2. **Безпека PII:** зберігання даних має відповідати європейським стандартам конфіденційності. Паролі повинні бути захешовані алгоритмами, стійкими до перебору (Argon2), а сесії – захищені підписаними HttpOnly cookie.

5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

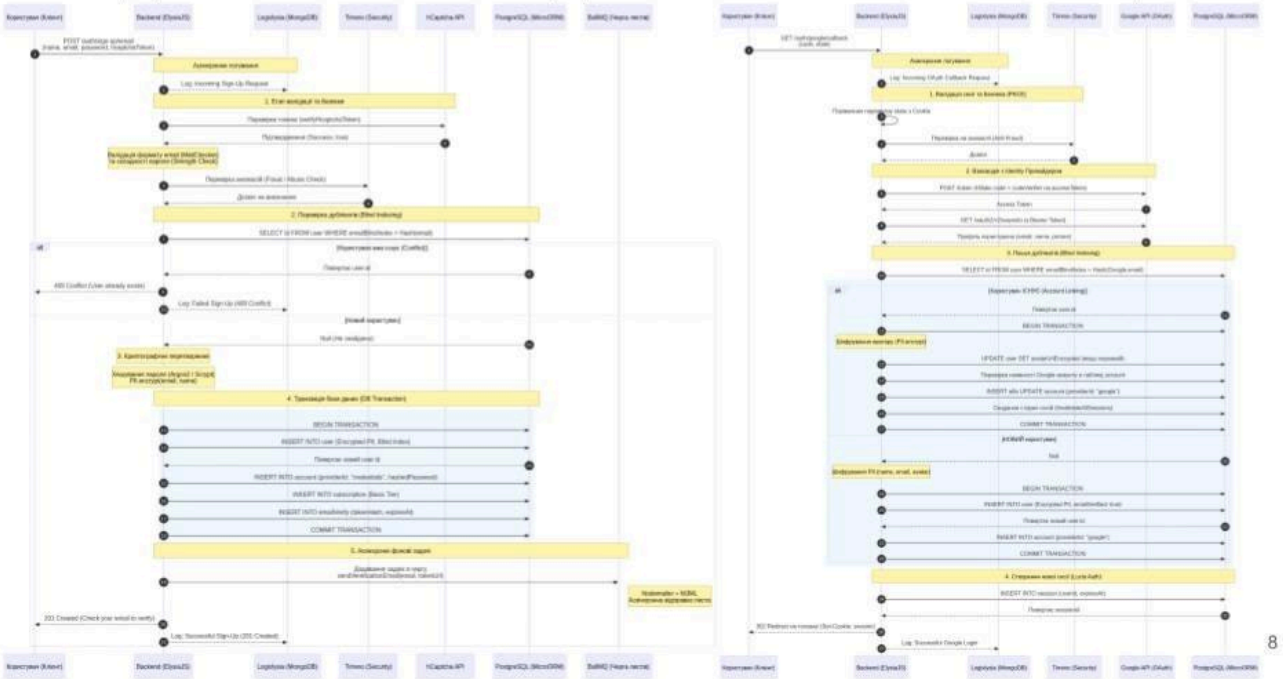
6

ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ



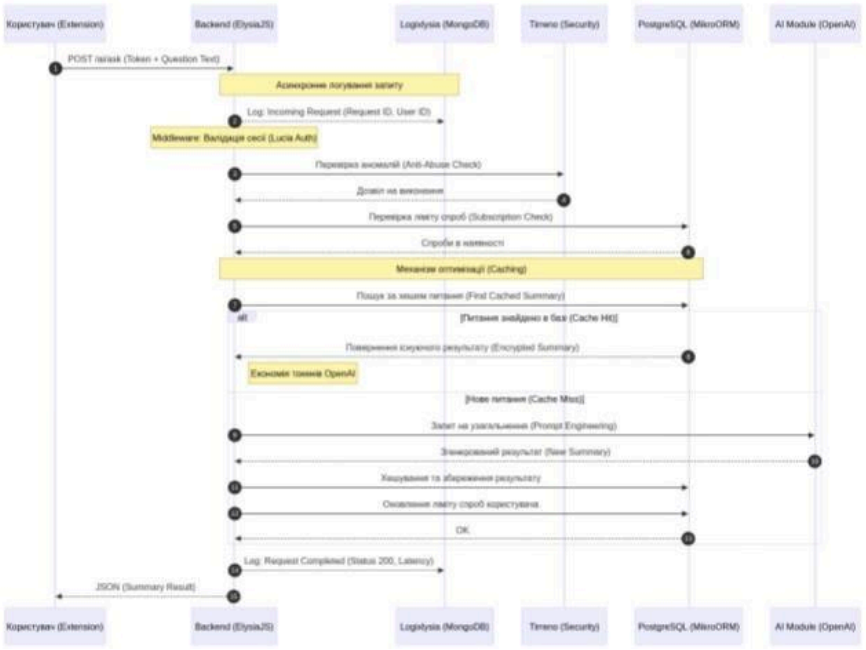
7

ДІАГРАМИ ПОСЛІДОВНОСТІ АВТОРИЗАЦІЇ



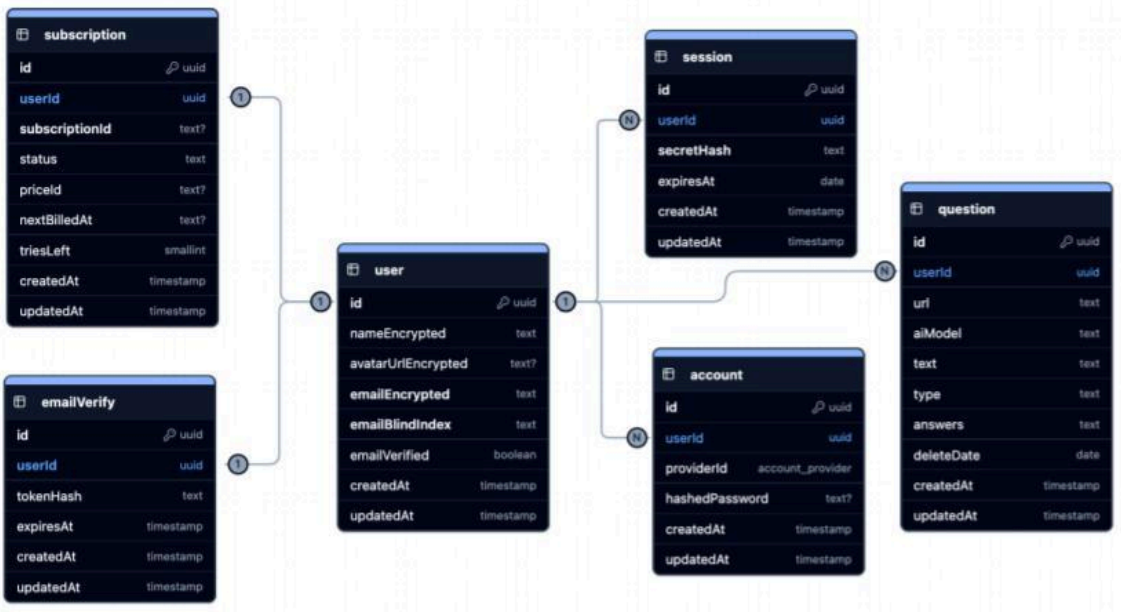
8

ДІАГРАМИ ПОСЛІДОВНОСТІ СКОРОЧЕННЯ ТЕКСТУ



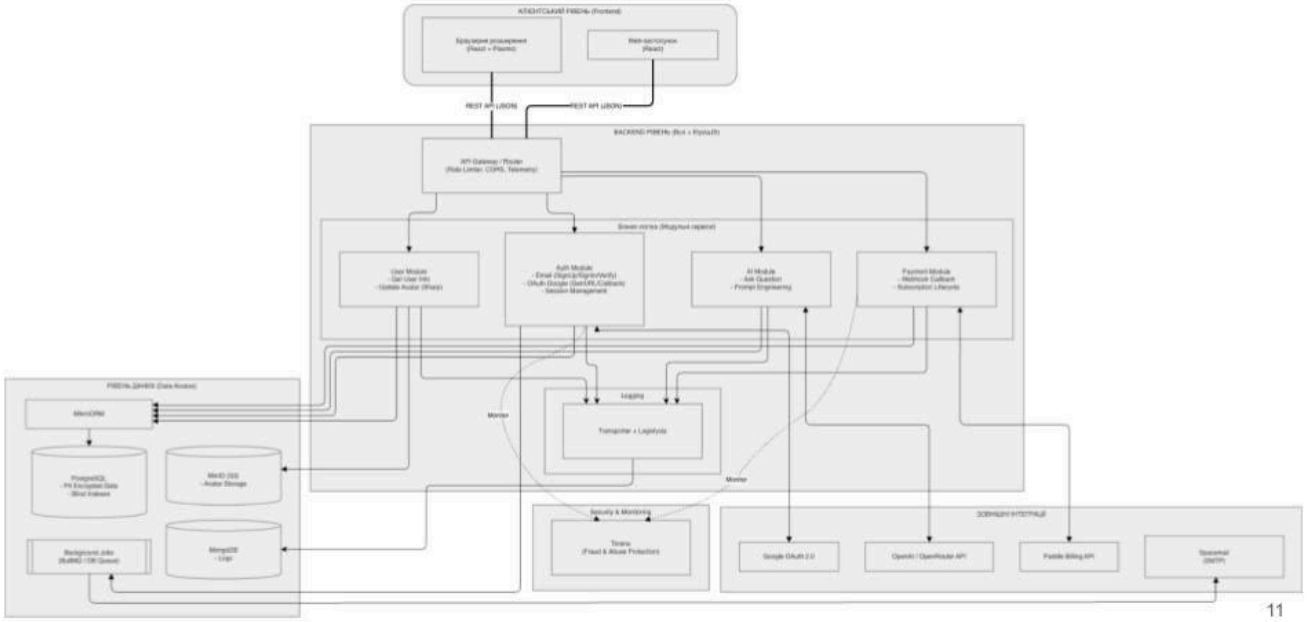
9

СТРУКТУРА БАЗИ ДАНИХ



10

АРХІТЕКТУРА ДОДАТКУ



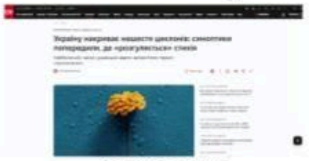
11

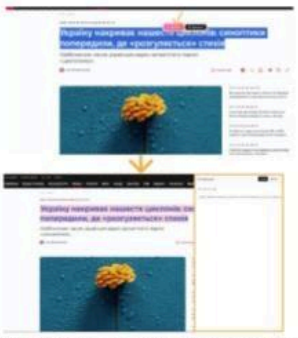
ПРОДУКТИВНІСТЬ BACKEND-ЧАСТИНИ

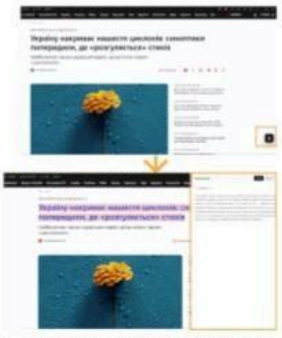
```
HTTP
http_req_duration.....................: avg=17.11ms  min=114µs   med=377µs    max=489.61ms p(90)=31.66ms  p(95)=160.34ms
  { expected_response:true }..........: avg=55.49ms  min=114µs   med=558µs    max=489.61ms p(90)=171.21ms p(95)=189.39ms
  { group:::Auth - google-callback }...: avg=414.2µs  min=168µs   med=297µs    max=6.95ms   p(90)=488.2µs  p(95)=765.8µs
  { group:::Auth - google-url }........: avg=523.84µs min=239µs   med=405µs    max=3.74ms   p(90)=581.2µs  p(95)=884.39µs
  { group:::Auth - root }..............: avg=708.35µs min=114µs   med=453µs    max=11.4ms   p(90)=1.5ms    p(95)=1.8ms
  { group:::Auth - session }...........: avg=280.16µs min=123µs   med=219µs    max=4.2ms    p(90)=339.6µs  p(95)=423.59µs
  { group:::Auth - sign-in }...........: avg=1.86ms   min=836µs   med=1.59ms   max=24.66ms  p(90)=3.1ms    p(95)=3.76ms
  { group:::Auth - sign-out }..........: avg=340.68µs min=150µs   med=246µs    max=4.21ms   p(90)=392.6µs  p(95)=555.79µs
  { group:::Auth - sign-up }...........: avg=165.23ms min=94.66ms med=160.37ms max=489.61ms p(90)=200.38ms p(95)=214.19ms
  { group:::Auth - verify-email }......: avg=1.04ms   min=462µs   med=876µs    max=6.15ms   p(90)=1.54ms   p(95)=2.63ms
  { group:::Paddle - webhook }.........: avg=364.36µs min=172µs   med=284µs    max=3.68ms   p(90)=435.6µs  p(95)=518.19µs
  { group:::User - upload-avatar }.....: avg=375.43µs min=171µs   med=291µs    max=5.82ms   p(90)=435µs    p(95)=539.99µs
http_req_failed.....................: 70.00% 5495 out of 7850
http_reqs...........................: 7850   65.00378/s

EXECUTION
iteration_duration..................: avg=1.17s    min=1.1s    med=1.16s    max=1.49s    p(90)=1.2s     p(95)=1.22s
iterations..........................: 785    6.500378/s
vus.................................: 1      min=1          max=10
vus_max.............................: 10     min=10         max=10

NETWORK
data_received.......................: 4.0 MB 34 kB/s
data_sent...........................: 2.0 MB 17 kB/s
```

12

ЕКРАННІ ФОРМИ
елементи що створюються на сторінках для аналізу


Довільна сторінка

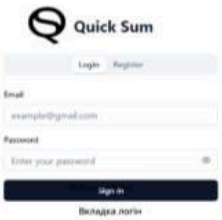

Виділення тексту і натискання режиму коротко


Натискання кнопки сумарізування всієї сторінки

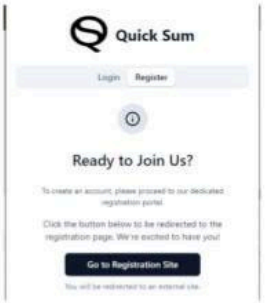
13

ЕКРАННІ ФОРМИ
сторінки в popup

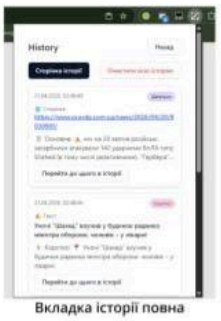

Вкладка логін в режимі розробника



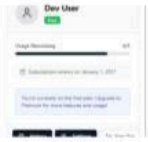
Вкладка логін



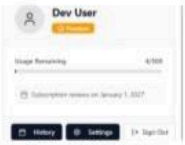
Вкладка регістрації



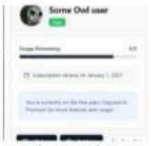
Вкладка історії повна



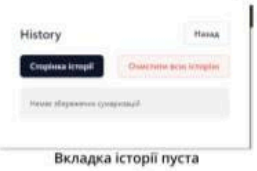
Вкладка дашбод Free User



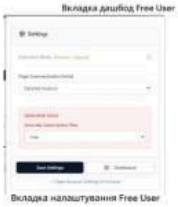
Вкладка дашбод Premium User



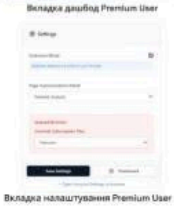
Вкладка дашбод



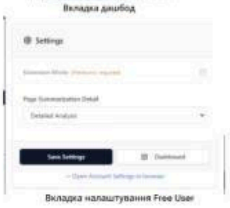
Вкладка історії пуста



Вкладка налаштування Free User



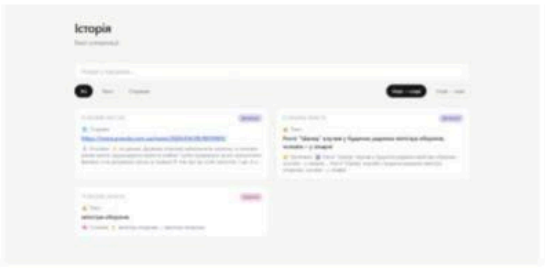
Вкладка налаштування Premium User

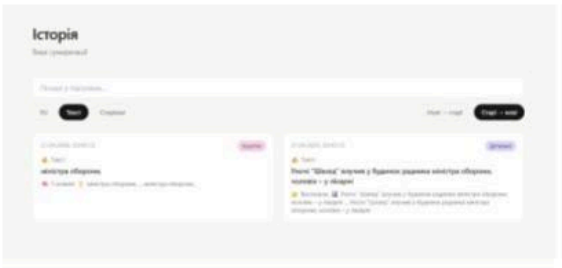


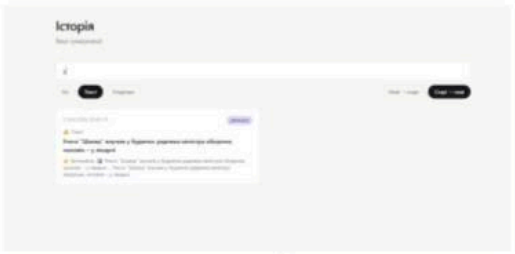
Вкладка налаштування Free User

14

ЕКРАННІ ФОРМИ
окрема сторінка історії в розширенні





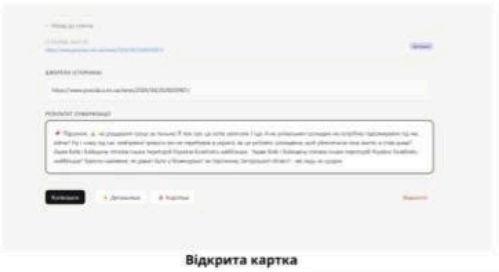


Історія якщо є данні | якщо данних немає

Сортування | Пошук

15

ЕКРАННІ ФОРМИ
окрема сторінка історії в розширенні



Відкрита картка

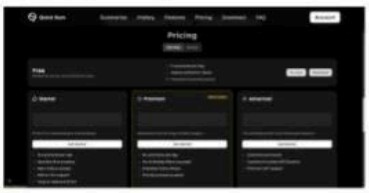
Ресумарізація

Версії після ресумарізації

16

ЕКРАННІ ФОРМИ

Сторінка веб-додатку



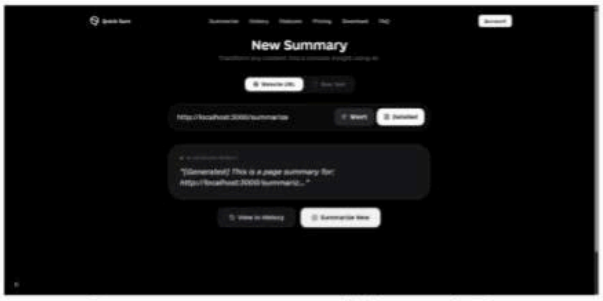
17

ЕКРАННІ ФОРМИ

сторінка сумаризації в веб-додатку

Якщо користувач авторизований до сумарізації

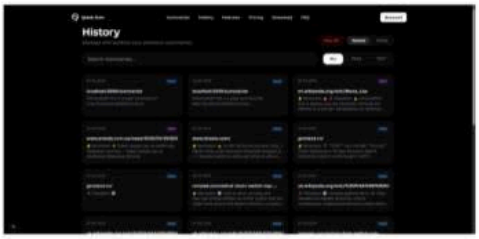
Якщо користувач не авторизований



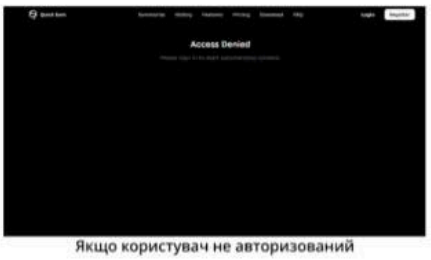
Якщо користувач авторизований після сумарізації

18

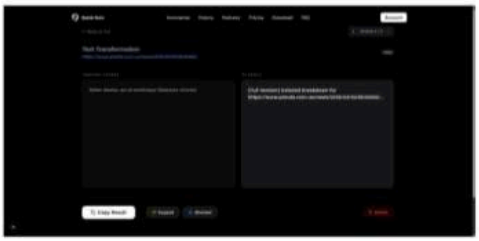
ЕКРАННІ ФОРМИ
сторінка історії в веб-додатку



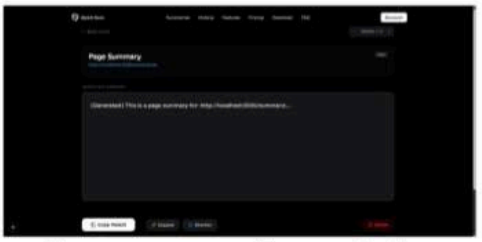
Якщо користувач авторизований



Якщо користувач не авторизований



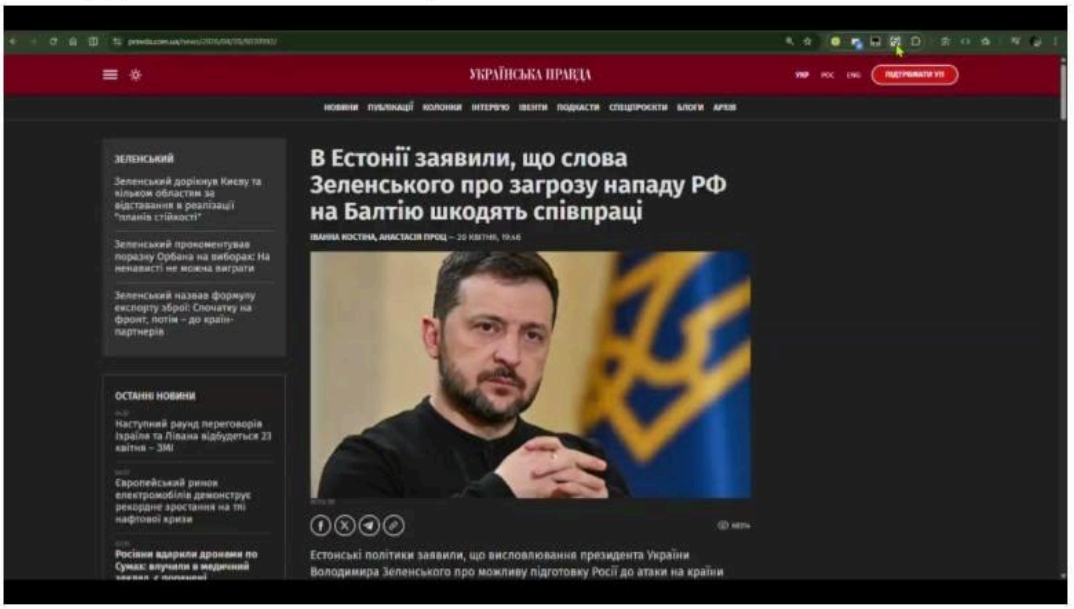
Користувач авторизований | сумаризація тексту



Користувач авторизований | сумаризація сайту

19

ДЕМОНСТРАЦІЯ РОБОТИ ПРОГРАМИ



20

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Карманний М.Ю., Залива В.В. Визначення вимог до серверної частини для клієнтському застосунку для обробки та узагальнення текстової інформації з web-ресурсів. Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». 23 квітня 2026 року, Державний університет інформаційно-комунікаційних технологій, Київ. Збірник тез. К.: ДУІКТ, 2026. С. 420-422.
2. Карманний М.Ю., Залива В.В. Механізми забезпечення конфіденційності та безпеки даних у клієнтському застосунку для обробки та узагальнення текстової інформації з web-ресурсів. Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». 23 квітня 2026 року, Державний університет інформаційно-комунікаційних технологій, Київ. Збірник тез. К.: ДУІКТ, 2026. С. 245-246.

ВИСНОВКИ

1. Досліджено сучасні AI-підходи та виявлено проблеми існуючих SaaS-рішень (мережеві затримки, кешування, безпека).
2. Сформульовано вимоги до високонавантаженого сервісу з акцентом на продуктивність, масштабованість і захист даних.
3. Обґрунтовано вибір технологічного стеку: Bun, ElysiaJS, PostgreSQL, MongoDB, а також інтеграцію Lucia Auth та Paddle.
4. Реалізовано ключові механізми: криптографічне шифрування PII-даних та алгоритм глобального кешування запитів.
5. Навантажувальне тестування підтвердило високу продуктивність: медіанний час відгуку при кешуванні менше 50 мс.
6. Оцінено високий комерційний потенціал застосунку та запропоновано шляхи подальшого вдосконалення інфраструктури.

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

```
export default abstract class AIService {
  static async askQuestion(
    user: User,
    quiz: AIModel["quizBody"],
  ): Promise<QuizResponse[]> {
    const { questions, quiz_url } = quiz;


    if (!questions.length) return [];


    if (user.subscription.triesLeft < questions.length) {
      throw status(402, "User is out of day limit");
    }


     const { aiModel, acceptableModels, worseModels } =
await this.evaluateUserModels(user);


        const questionTexts = questions.map((q) =>
q.question);
    const cachedRows = await db
      .select({
        id: question.id,
        question: question.question,
        answers: question.answers,
      })
      .from(question)
      .where(
        and(
          eq(question.quizUrl, quiz_url),
          inArray(question.question, questionTexts),
          inArray(question.aiModel, acceptableModels),
        ),
      );



                    const cachedByText = new
Map(cachedRows.map((row) => [row.question, row]));


     const cachedHits = new Map<string, { rowId: string;
response: QuizResponse }>();
    const uncachedQuestions: QuizQuestion[] = [];


    for (const q of questions) {
      const hit = cachedByText.get(q.question);

      if (hit) {
        cachedHits.set(q.id, {
          rowId: hit.id,
          response: {
            questionId: q.id,
                                answers: resolveAnswerIds(q,
hit.answers.map((text) => ({ text }))),
          },
        });
      } else {
        uncachedQuestions.push(q);
      }
    }



    const aiAnswers = uncachedQuestions.length > 0
            ? await aiHelperService.askQuestion(aiModel,
uncachedQuestions)
      : [];



    await this.syncDatabaseState({
      userId: user.id,
      quizUrl: quiz_url,
      totalQuestions: questions.length,
      triesLeft: user.subscription.triesLeft,
                                                cachedRowIds:
Array.from(cachedHits.values()).map((hit) => hit.rowId),
      uncachedQuestions,
      worseModels,
      aiModelName: aiModel.model,
      aiAnswers,
    });


    let uncachedIndex = 0;

    return questions.map((q) => {
      const cached = cachedHits.get(q.id);
      if (cached) return cached.response;


      const fresh = aiAnswers[uncachedIndex++];
      return {
        questionId: q.id,
        answers: resolveAnswerIds(q, fresh.answers),
      };
    });
  }


        static async removeOutdatedQuestions():
Promise<void> {
    try {
      await db
        .delete(question)
                              .where(lt(question.deleteDate, new
Date().toISOString()));
    } catch {
        throw status(500, "Error while removing outdated
questions");
    }
```

```
}
/**
   * Evaluates the active ruleset to define which AI models this user is
  * allowed to query, access from cache, or override.
  */
 private static async evaluateUserModels(user: User) {
                    const ruleResult = await RuleEngine.evaluate<SubscriptionTierRuleResult>(
    subscriptionTierRule,
                user as unknown as Record<string, SubscriptionTierRuleResult>,
   );


   const aiModel = ruleResult.value.model;
   const acceptableModels = [
    aiModel.model,
              ...ruleResult.value.betterModels.map((m) => m.model),
   ];
                          const worseModels = ruleResult.value.worseModels.map((m) => m.model);


   return { aiModel, acceptableModels, worseModels };
 }


 /**
  * Handles all DB operations ensuring data consistency after an AI cycle.
  */
    private static async syncDatabaseState(params: SyncDBParams): Promise<void> {
   await db.transaction(async (tx) => {
    await tx
     .update(subscription)
                      .set({ triesLeft: params.triesLeft - params.totalQuestions })
      .where(eq(subscription.userId, params.userId));


    if (params.cachedRowIds.length > 0) {
     await tx
      .update(question)
      .set({ deleteDate: getExpirationDate() })
                                  .where(inArray(question.id, params.cachedRowIds));
    }


          if (params.uncachedQuestions.length > 0 && params.worseModels.length > 0) {
                               const uncachedTexts = params.uncachedQuestions.map((q) => q.question);

     await tx
      .delete(question)
      .where(
       and(
        eq(question.quizUrl, params.quizUrl),
        inArray(question.question, uncachedTexts),
                                                inArray(question.aiModel, params.worseModels),
        ),
       );
    }


    if (params.aiAnswers.length > 0) {
                                const payloadToInsert = params.aiAnswers.map((answer, i) => ({
      quizUrl: params.quizUrl,
      userId: params.userId,
      aiModel: params.aiModelName,
      question: params.uncachedQuestions[i].question,
      answers: answer.answers.map((ans) => ans.text),
      deleteDate: getExpirationDate(),
     }));


     await tx.insert(question).values(payloadToInsert);
    }
   });
 }
}

export class AIService {


 private transformQuestions(qs: IQuestion[]): string {
  return encode(qs)
 }


 private transformImage(image: Buffer): string {
                                                                return `${image.toString("base64")}`;
 }


 // TODO: add image support
 async askQuestion(
  ai: IAI,
  quizData: IQuestion[],
  image?: Buffer,
 ): Promise<IAnswers> {


  const prompt = this.transformQuestions(quizData);


  let image_url: string | undefined;
  if (image && ai.image_support) {
   image_url = this.transformImage(image)
  }
```

```
    const response = await openRouter.chat.send({
      chatRequest: {
        model: ai.model + "@preset/ask-question",
        messages: [
          {
            role: "user",
            content: prompt,
          },
        ],
        stream: false,
      },
    });

    const content = response.choices[0].message.content;

    console.log(content);
    if (!content) {
      throw new Error("No response content received from
OpenAI chat");
    }

    try {
      return JSON.parse(content) as IAnswers;
    } catch (error) {
      console.error("askQuestion error:", error);
          throw new Error("Failed to parse OpenAI chat
response as JSON");
    }
  }
}
```