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 Frontend-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

_____ Денис ГАСИЛО
(підпис)

Виконав: здобувач вищої освіти групи ПД-42
Денис ГАСИЛО

Керівник: Віталій ЗАЛИВА
доктор філософії (PhD)

Рецензент: _____

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

Гасило Денису Віталійовичу

1. Тема кваліфікаційної роботи: «Розробка клієнтського застосунку для обробки та узагальнення текстової інформації з web-ресурсів. Спец-частина: розробка Frontend-частини застосунку»
керівник кваліфікаційної роботи доктор філософії (PhD) Віталій ЗАЛИВА
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 р. № 51

2. Строк подання кваліфікаційної роботи «29» травня 2026 р.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про сучасні методи розробки клієнтських web-застосунків та браузерних розширень, принципи автоматизованого узагальнення текстової інформації за допомогою великих мовних моделей (LLM), технічна документація фреймворку React, платформи Plasmo Framework, мови TypeScript, системи керування станом Zustand, CSS-фреймворку Tailwind CSS та засобів інтеграції з REST API.

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 фронтенд частини застосунку.
3. Проєктування архітектури фронтенд частини застосунку
4. Програмна реалізація, тестування фронтенд частини застосунку.

5. Перелік графічного матеріалу: презентація

1. Аналіз аналогів.
2. Вимоги до додатку.
3. Програмні засоби реалізації.
4. Діаграма варіантів використання.
5. Архітектура QuickSum.
6. Блок схеми.
7. Тестування
8. Екранні форми.
9. Апробація результатів дослідження.
10. Висновки.

6. Дата видачі завдання «23» лютого 2026 р.

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 існуючих рішень	08.03-10.03.2026	
4	Огляд та аналіз засобів реалізації frontend частин системи	11.03-13.03.2026	
5	Проектування архітектури frontend частин системи	14.03-06.04.2026	

6	Програмна реалізація та тестування frontend частин системи	07.04-25.04.2026	
7	Оформлення роботи: вступ, висновки, реферат	26.04-11.05.2026	
8	Розробка демонстраційних матеріалів	12.05-13.05.2026	
9	Попередній захист роботи	14.05-22.05.2026	

Здобувач вищої освіти _____ Денис ГАСИЛО

(підпис)

Керівник кваліфікаційної роботи _____ Віталій ЗАЛИВА

(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 55 стор., 1 табл., 26 рис., 25 джерел.

Мета роботи – спрощення процесу опрацювання великих обсягів контенту користувачами на основі 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 – програмні засоби та технології розробки клієнтської частини застосунку для автоматизованого захоплення та інтелектуального узагальнення текстової інформації з web-ресурсів із використанням моделей штучного інтелекту.

Короткий зміст роботи: У роботі проаналізовано проблему стрімкого зростання обсягів інформаційного контенту та складність його опрацювання користувачами. Було розглянуто наявні рішення аналоги на ринку. Розроблено архітектуру фронтенд частини застосунку для обробки тексту та взаємодії зі штучним інтелектом. Програмно реалізовано ключові модулі: вилучення повного та виділеного тексту сторінки, аналіз тексту через API серверної частини, систему автентифікації, а також режим розробника для тестування функцій без прив'язки до сервера. Запроваджено механізм збереження історії сумаризації та візуального відображення виділеного тексту на сторінці, що зберігається навіть після оновлення вкладки.

Для проектування та технічної реалізації клієнтської частини браузерного розширення було використано спеціалізований мета-фреймворк Plasmo, який забезпечує повноцінну кросбраузерність продукту, автоматизує генерацію маніфесту згідно зі стандартами Manifest V3 та дозволяє ефективно

перевикористовувати модульні React-компоненти в ізольованому середовищі вебсторінок. Водночас повнофункціональний вебзастосунок екосистеми побудовано на базі сучасного фреймворку Next.js, що дозволило оптимізувати пошукову індексацію (SEO) та радикально знизити обчислювальне навантаження на клієнтську сторону користувача завдяки механізмам серверного рендерингу (SSR) та гібридної генерації сторінок.

Комплексне тестування працездатності та стабільності інтерфейсу реалізовано за допомогою високопродуктивного середовища Vitest, призначеного для автоматизованого виконання модульних та навантажувальних інтеграційних тестів UI-компонентів. Окрему увагу в межах архітектури було приділено багаторівневому захисту інформації та безпеці користувацьких даних. Для цього на стороні вебплатформи налаштовано суворі політики контекстного доступу до приватних сторінок сайту, а всередині браузерного розширення всі критично важливі конфіденційні налаштування, сесії авторизації та рівні підписки клієнта зберігаються у локальному сховищі Plasmo Storage із застосуванням алгоритмів шифрування.

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

КЛЮЧОВІ СЛОВА: СУМАРИЗАЦІЯ ТЕКСТУ, ШТУЧНИЙ ІНТЕЛЕКТ, БРАУЗЕРНЕ РОЗШИРЕННЯ, PLASMO, NEXT.JS, REACT, БЕЗПЕКА ДАНИХ, АВТОМАТИЗАЦІЯ АНАЛІЗУ ІНФОРМАЦІЇ, СЕРВЕРНИЙ РЕНДЕРИНГ, ІНТЕГРАЦІЙНЕ ТЕСТУВАННЯ.

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 контенту вимагає створення нових інструментів, що забезпечують безшовну взаємодію з веб-ресурсами. Актуальність дослідження зумовлена необхідністю розробки рішення, яке інтегрується безпосередньо у браузерне середовище та надає користувачеві стислий зміст сторінок у реальному часі за допомогою генеративного штучного інтелекту.

Проблематика дослідження полягає в архітектурній складності побудови кросбраузерних рішень, що мають забезпечувати стабільне вилучення контенту (DOM-парсинг), збереження контексту користувача та візуалізацію результатів сумаризації без порушення цілісності вихідних web-ресурсів. Більшість існуючих аналогів мають обмежений UI/UX, вимагаючи ручного копіювання тексту, або не забезпечують збереження історії взаємодії безпосередньо в інтерфейсі браузера.

Мета роботи – спрощення процесу опрацювання великих обсягів контенту користувачами на основі 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 – програмні засоби та технології розробки клієнтської частини застосунку для автоматизованого захоплення та інтелектуального узагальнення текстової інформації з web-ресурсів із використанням моделей штучного інтелекту.

Методи дослідження – Першим кроком було проаналізовано існуючі рішення на ринку систем сумаризації тексту та інструментів для роботи з AI-контентом для

формування початкових функціональних та нефункціональних вимог до власної системи.

Далі було проведено огляд технологічного стеку, здатного забезпечити створення кросбраузерного рішення з високим рівнем продуктивності. Для розробки клієнтської частини було обрано фреймворк Plasmo (для браузерного розширення) та Next.js (для вебзастосунку), основною мовою програмування — TypeScript, а для побудови інтерфейсів — бібліотеку React. Для гарантування якості коду та стабільності інтерфейсу було впроваджено середовище тестування Vitest. Дослідження архітектури та реалізації механізмів вилучення контенту (DOM-парсинг) здійснювалося безпосередньо під час етапів проєктування та розробки.

Наукова новизна – полягає у розробці багаторівневої фронтенд-архітектури для забезпечення безшовної синхронізації між браузерним розширенням та вебзастосунком, що доповнюється реалізацією авторського алгоритму персистентності для збереження та візуалізації сумаризованого контенту навіть після оновлення сторінок. Поряд із цим, новизна визначається впровадженням механізмів захищеного зберігання конфіденційних даних за допомогою технології Plasmo Secret Storage із застосуванням клієнтського шифрування, а також розробкою автономного «режиму розробника», який дозволяє здійснювати гнучке тестування інструментів штучного інтелекту без прямої залежності від серверної інфраструктури.

Практична значущість – полягає у створенні ефективного інструменту «QuickSum» для автоматизації процесів аналізу великих обсягів текстової інформації. Розроблене програмне забезпечення дозволяє суттєво зменшити час на опрацювання контенту для широкого кола користувачів (дослідників, студентів, аналітиків).

Крім того, архітектурні підходи та обраний технологічний стек (Plasmo + Next.js) можуть бути використані як технологічна база для розгортання масштабованих SaaS-продуктів, орієнтованих на інтеграцію генеративного штучного інтелекту в браузерне середовище, також можна розширити функціонал.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

Обсяги генерованого контенту стрімко зростають, випереджаючи можливості людини щодо його аналітичного опрацювання. Це призводить до явища «інформаційного перевантаження», за якого критичне сприйняття інформації ускладнюється через надмірну кількість неструктурованих даних. У зв’язку з цим виникає актуальна науково-практична потреба у створенні систем інтелектуального аналізу та автоматизованого узагальнення контенту на основі архітектур Large Language Models (LLM)[26]. Використання таких систем дозволяє значно підвищити швидкість опрацювання інформації та продуктивність користувача[1].

Дане дослідження зосереджене на проєктуванні архітектурних рішень системи «QuickSum», яка забезпечує швидке, безпечне та зручне узагальнення контенту безпосередньо у браузерному середовищі.

1.1 Особливості задачі та цільова аудиторія

Концепція «QuickSum» базується на принципі мінімальної когнітивної дистанції: користувач повинен отримувати аналітичне резюме тексту безпосередньо в точці споживання контенту, не перериваючи основний робочий процес.

Оскільки задача має загальний характер, то основними векторами використання системи є:

Академічний та R&D сектор: інтелектуальний скринінг масивів наукових публікацій для виокремлення методологічного базису та результатів досліджень.

Аналітичний менеджмент: обробка багатосторінкової документації та галузевих звітів для підтримки прийняття управлінських рішень.

Інфо-гігієна та медіа-споживання: фільтрація інформаційного «шуму» у

новинних потоках для економії часу рядових користувачів.

Проєктування системи здійснювалося з урахуванням трьох імперативів: інфраструктурної реактивності (відгук під час обробки тексту сторінки < 500мс без урахування генерації LLM), браузерна нативність та криптографічної стійкості персональних даних.

1.2 Аналіз існуючих рішень та їх недоліків

На сучасному ринку задача сумаризації тексту вирішується переважно за допомогою SaaS-платформ, що виступають проміжною ланкою між користувачем та комерційними API (наприклад, OpenAI). Проте детальний аналіз таких систем, як Summarizer.org, Smart Summarize, Summarize AI та AISMRY, дозволив виявити критичну невідповідність сучасним вимогам до фронтенд-архітектури та користувацького досвіду (UX):

Відсутність гібридності: Існуючі аналоги демонструють жорстку прив’язку до одного типу інтерфейсу. Більшість із них функціонують виключно, як ізольовані вебсайти, що змушує користувача здійснювати надлишкові ітерації «копіювання-вставки», руйнуючи безперервність робочого процесу. Ті ж рішення, що представлені у вигляді розширень, часто не мають повноцінної вебплатформи для аналізу історії запитів. Це ігнорує варіативність користувацьких сценаріїв: необхідність миттєвого аналізу «на місці» (через розширення) та потребу в окремому робочому середовищі для роботи з архівом сумаризацій (через вебсайт).

Ефемерність інтерфейсного стану (Lack of State Persistence): Аналізовані системи здебільшого реалізують модель «one-shot» взаємодії. Це означає, що результати аналізу та візуальні акценти на сторінці втрачаються одразу після оновлення вкладки або переходу за посиланням. Відсутність механізмів збереження стану змушує користувача повторно генерувати запити, що створює зайве навантаження на API та погіршує показник утримання користувачів.

Для розробки власної системи було проаналізовано найпопулярніші сервіси, що допомагають стискати текст. Основна увага приділялася тому, як користувач

взаємодіє з програмою: чи зручно це робити через сайт, чи через розширення, і як швидко він отримує результат.

1.2.1 Summarizer.org

Першим серед аналогів було детально розглянуто сервіс Summarizer.org[2], який позиціонує себе як спеціалізований онлайн-інструмент для автоматичного створення стислих викладів тексту на основі алгоритмів обробки природної мови (NLP). Основною технологічною перевагою цього рішення є висока швидкість обробки вхідних даних (середній час генерації не перевищує 5 секунд), що досягається шляхом використання оптимізованих серверних потужностей. Сервіс пропонує користувачеві гнучкість у налаштуваннях: наявність кількох режимів узагальнення дозволяє вручну регулювати відсоткове співвідношення стислого викладу до оригіналу, що є корисним при роботі з різними типами контенту — від новинних заміток до наукових статей.

Графічний інтерфейс платформи виконаний у мінімалістичному стилі, що робить його доступним для широкого кола користувачів без необхідності попереднього навчання або реєстрації. Проте, з точки зору продуктивності професійної роботи, сервіс має суттєвий архітектурний недолік — повна відсутність інтеграції з браузерним середовищем у формі розширення. Це створює серйозний технічний бар'єр: користувач змушений виконувати велику кількість рутинних операцій, як-от перемикання між вкладками, ручне виділення та копіювання тексту, а потім повернення до сервісу для вставки. Такий робочий процес (workflow) значно уповільнює аналіз великої кількості джерел і робить систему неефективною для активного серфінгу в мережі.

Приклад інтерфейсу web-ресурсу наведено на рисунку 1.1.

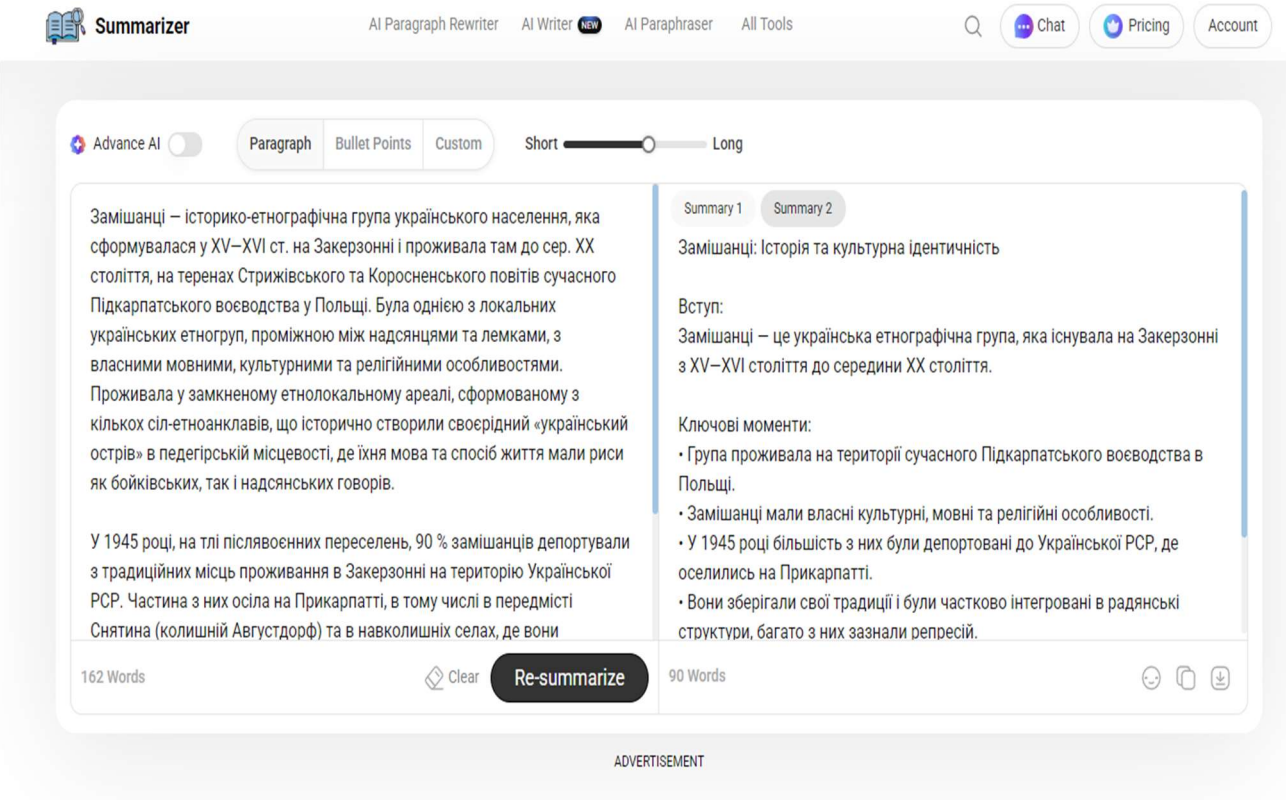


Рис. 1.1 Інтерфейс сервісу Summarizer.org

1.2.2 Smart Summarize

Платформа Smart Summarize[3] позиціонується як професійне рішення для глибокого аналізу та структурування інформації у зручні списки й тези. До її переваг можна віднести якісну візуальну подачу результату та адаптивний дизайн вебсайту, який зручно відображається на різних пристроях. Однак сервіс має критичний недолік у швидкості роботи: час очікування відповіді часто перевищує 40 секунд, що є неприпустимим для оперативного аналізу новин або статей. Окрім того, відсутність інтеграції безпосередньо в браузер обмежує мобільність користувача, змушуючи його використовувати лише веб-інтерфейс платформи. Приклад головної сторінки платформи наведено на рисунку 1.2.

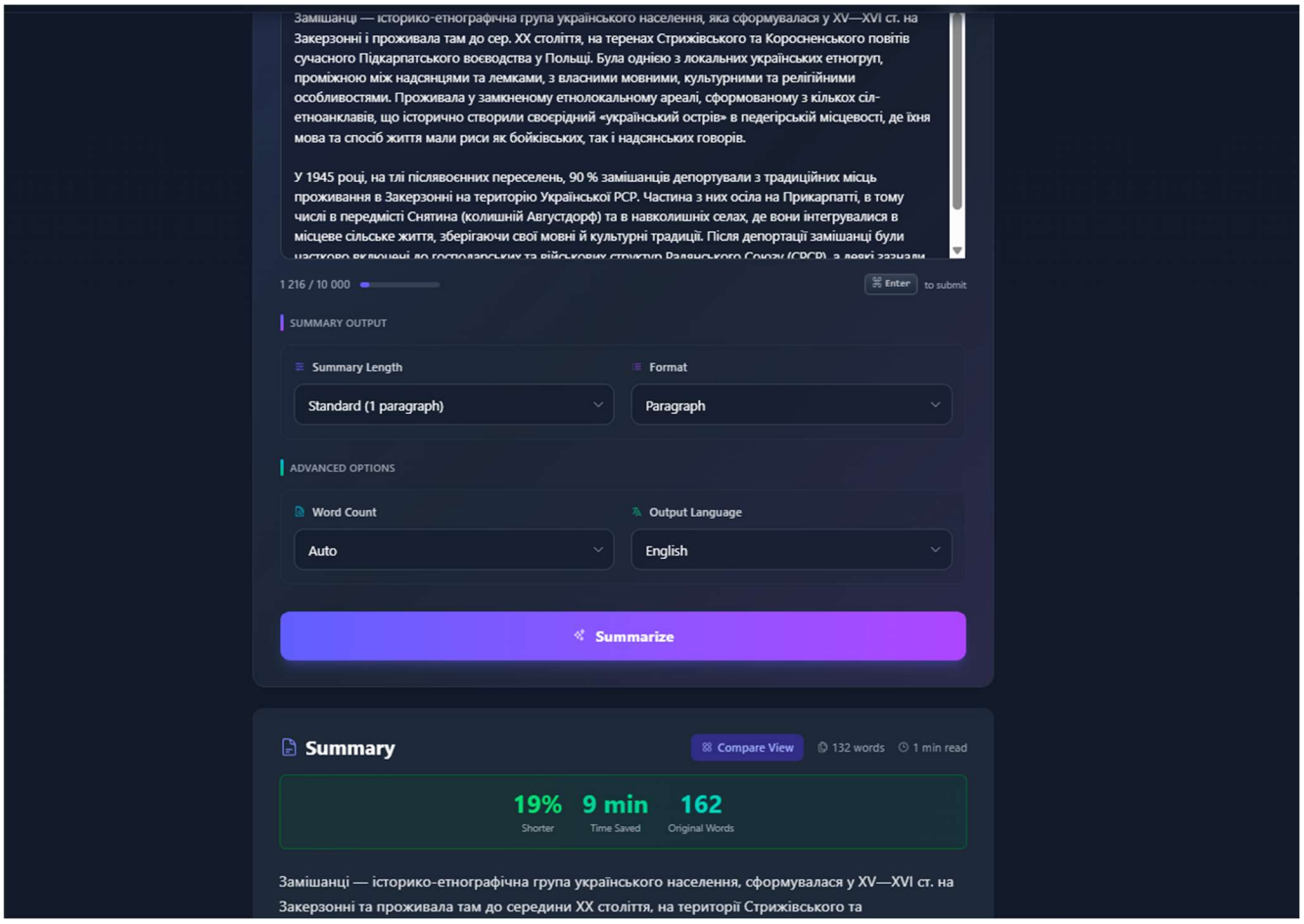
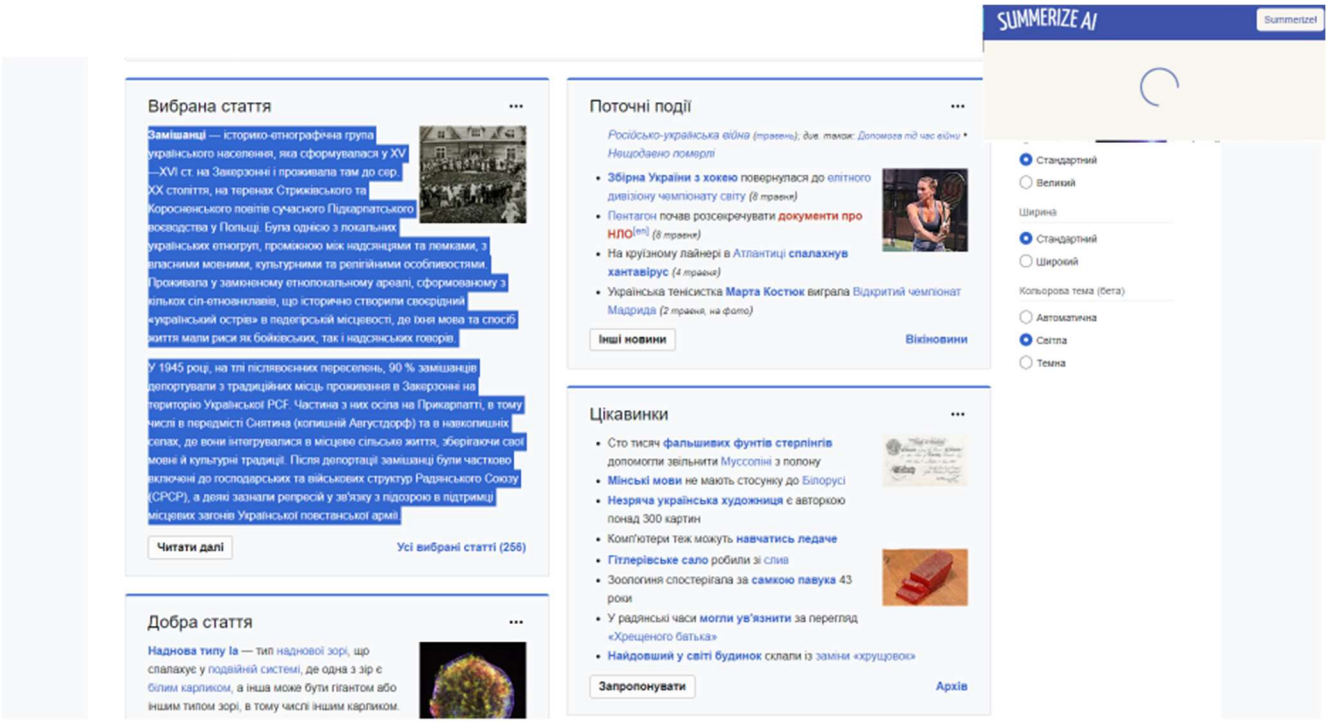


Рис. 1.2 Інтерфейс платформи Smart Summarize

1.2.3 Summarize AI

На відміну від попередніх сайтів, Summarize AI[4] реалізовано у формі розширення для Google Chrome, що дозволяє отримувати зміст сторінки в один клік безпосередньо під час перегляду. Головною перевагою тут є висока ергономічність, оскільки інструмент завжди під рукою у вигляді спливаючого вікна. Проте функціональність розширення обмежена лише одним базовим режимом роботи, а неоптимізована серверна архітектура призводить до значних затримок у генерації тексту (близько 1 хвилини). Також великим мінусом є відсутність повноцінного вебсайту, де можна було б зберігати та переглядати історію своїх запитів у зручному форматі. Приклад інтерфейсу розширення наведено на рисунку 1.3.



1.3 Інтерфейс розширення Summarize AI

1.2.4 AISMRY

Сервіс AISMRY[5] розглядається як прямий конкурент у сегменті браузерних рішень, оскільки він пропонує передові алгоритми для семантичного аналізу великих масивів тексту безпосередньо під час перегляду вебсторінок. Основною перевагою цього інструменту є підтримка багаторівневих стилів реферування: від простого екстрактивного методу (вилучення найбільш значущих речень) до глибокого абстрактного перефразування за допомогою нейромережевих моделей останнього покоління. Це дозволяє обробляти значні обсяги інформації, зберігаючи логічну структуру та ключовий зміст першоджерела.

Проте, незважаючи на потужну алгоритмічну базу, AISMRY має низку функціональних обмежень, що знижують його привабливість для досвідчених користувачів. Головним недоліком є відсутність можливості сумарізувати лише окремий виділений фрагмент тексту — система орієнтована виключно на аналіз всієї сторінки одночасно, що часто призводить до появи зайвого "шуму" в результатах (рекламні блоки, навігаційні посилання тощо). Крім того, на відміну від «QuickSum», AISMRY не пропонує окремої вебплатформи для роботи з

накопиченими даними поза межами браузера. Швидкість обробки запитів у цьому рішенні оцінюється як середня, а відсутність гібридного формату «сайт + розширення» робить систему менш універсальною та не здатною забезпечити безшовну синхронізацію між різними робочими середовищами. Користувач фактично стає "заручником" однієї вкладки, що не відповідає сучасним вимогам до кросплатформних AI-сервісів.

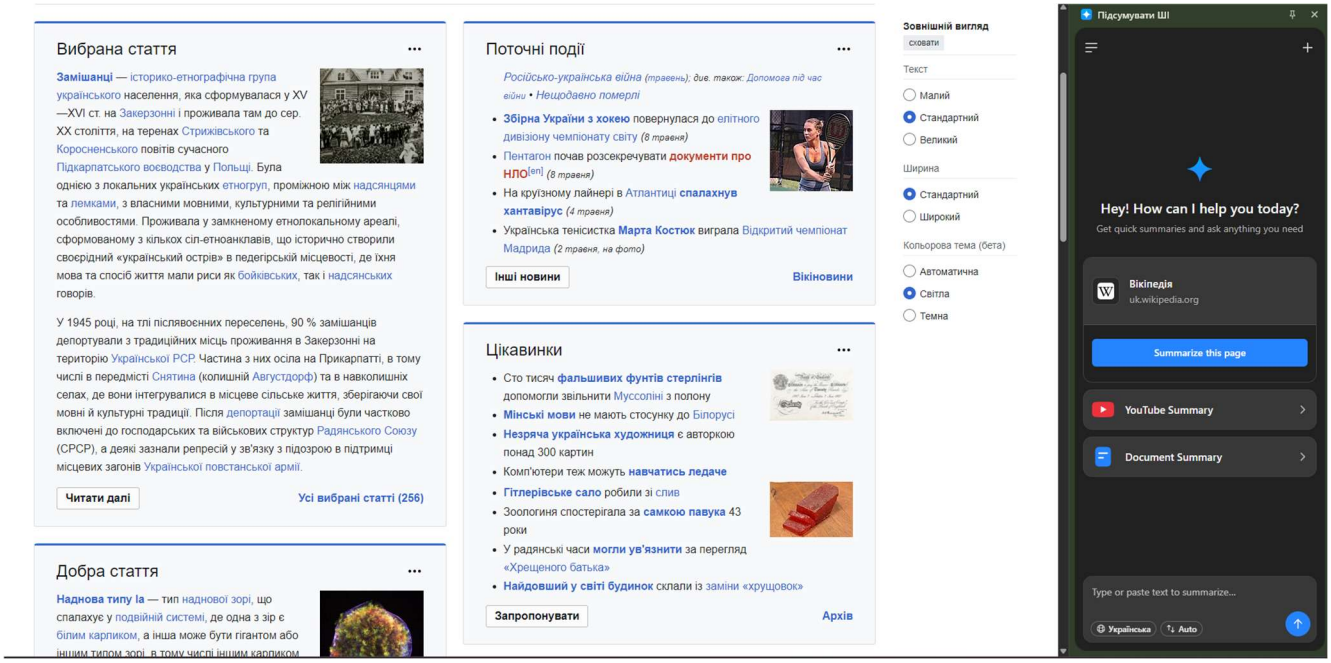


Рис. 1.4 Приклад інтерфейсу AISMRY

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**
Формат узагальнення тексту	Довгий, Короткий	Довгий, Середній, Короткий	Лише один	Лише один	**Середній, Короткий**
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)**
Робота з історією запитів	Ні	Так	Ні	Ні	**Так**

1.3 Шляхи вирішення проблеми на рівні Frontend-архітектури

Виявлені недоліки існуючих рішень стали основою для розробки клієнтської частини системи «QuickSum». Оскільки центральним завданням дослідження є створення зручного та продуктивного інтерфейсу, вирішення проблеми інформаційного перевантаження реалізується через впровадження сучасних фронтенд-практик:

По-перше, для забезпечення високої швидкості завантаження та кросбраузерності використовується гібридний підхід. Вебплатформа базується на фреймворку Next.js, що дозволяє використовувати серверний рендеринг (SSR) для миттєвого відображення контенту та покращення SEO-показників. Водночас браузерне розширення розробляється на базі фреймворку Plasmo, який оптимізує збірку під різні браузери (Chrome, Firefox, Edge) та мінімізує обсяг клієнтського коду, забезпечуючи реактивний відгук інтерфейсу.

По-друге, проблема розриву контексту та втрати даних вирішується через механізми персистентності та інтелектуального парсингу. На відміну від аналогів, система «QuickSum» впроваджує алгоритм збереження стану: виділені фрагменти тексту та результати їхнього аналізу фіксуються на сторінці за допомогою «шару візуалізації», який залишається активним навіть після оновлення вкладки. Це дозволяє користувачеві працювати з інформацією без повторних запитів до системи, що значно покращує UX (користувацький досвід) та економить ресурси.

По-третє, захист конфіденційності реалізується безпосередньо на рівні клієнта за принципом Security by Design. Для зберігання токенів доступу та персональних налаштувань використовується Plasmo Secret Storage[25]. Ця технологія дозволяє шифрувати дані на стороні браузера перед їхнім збереженням, що унеможливлює доступ до них сторонніх скриптів або зловмисників через стандартні інструменти розробника. Крім того, на рівні фронтенду впроваджено валідацію вхідних даних для захисту бізнес-логіки від некоректних запитів.

2 ЗАСОБИ ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

2.1 Інструментарій розробки та допоміжне програмне забезпечення

Для проектування фронтенд-частини, що включає в себе одночасно архітектуру автономного вебзастосунку та інтегрованого браузерного розширення, а також для їхньої безшовної взаємодії з серверною інфраструктурою було обрано комплекс сучасних програмних інструментів. Головним критерієм формування цього технологічного стеку стало забезпечення максимальної швидкості написання вихідного коду, оптимізація рутинних операцій збірки та мінімізація часу розгортання робочого середовища. Застосування компонентного підходу та суворої типізації даних на клієнтській стороні дозволило створити уніфіковану кодову базу, де логіка керування станами інтерфейсу, елементи дизайну та сервісні модулі є перевикористовуваними як у межах стандартної вебсторінки, так і всередині ізольованого контексту розширення.

2.1.1 Visual Studio Code

Visual Studio Code[6] - це основне середовище розробки, яке використовувалося для написання коду як браузерного розширення (Plasmo), так і вебплатформи (Next.js). Завдяки модульній архітектурі та підтримці TypeScript, VS Code забезпечив інтелектуальне автодоповнення та перевірку типів у реальному часі, що є критично важливим при роботі з складними об’єктами ШІ-відповідей. Використання розширень ESLint та Prettier дозволило підтримувати єдиний стиль коду, а вбудований термінал використовувався для одночасного запуску серверів розробки фронтенду та бекенду, що значно пришвидшило ітерації тестування функціоналу. Приклад інтерфейсу середовища розробки VS Code наведено на рисунку 2.1.

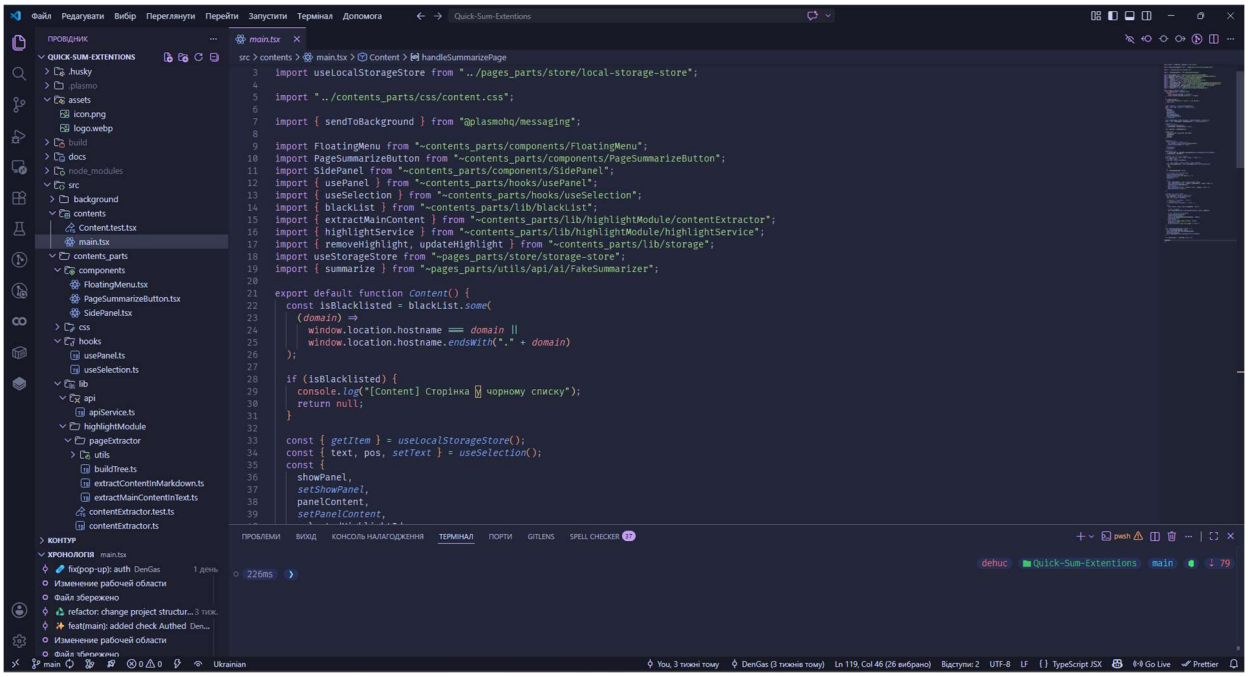


Рис. 2.1 Інтерфейс середовища розробки Visual Studio Code

2.1.2 HTTPie

HTTPie[7] використовувався як основний інструмент для налагодження взаємодії між клієнтською частиною та сервером, особливо на етапі реалізації системи авторизації. У ході розробки виникла архітектурна проблема: бібліотека Better Fetch, що використовувалася у розширенні, некоректно обробляла порожні POST-запити для отримання сесії. Оскільки запит містив лише заголовки (headers) без корисного навантаження (payload), за замовчуванням він інтерпретувався як GET-запит, що призводило до помилок на стороні сервера.

За допомогою HTTPie вдалося ізольовано протестувати API-ендпоінти та виявити цю невідповідність між очікуваним методом запиту та фактичним. Це дозволило скоригувати логіку фетч-запитів у фронтенд-частині, примусово встановивши правильні параметри передачі даних, що розв'язало проблему авторизації в браузерному розширенні.

Приклад тестування API показаний на рисунку. 2.2

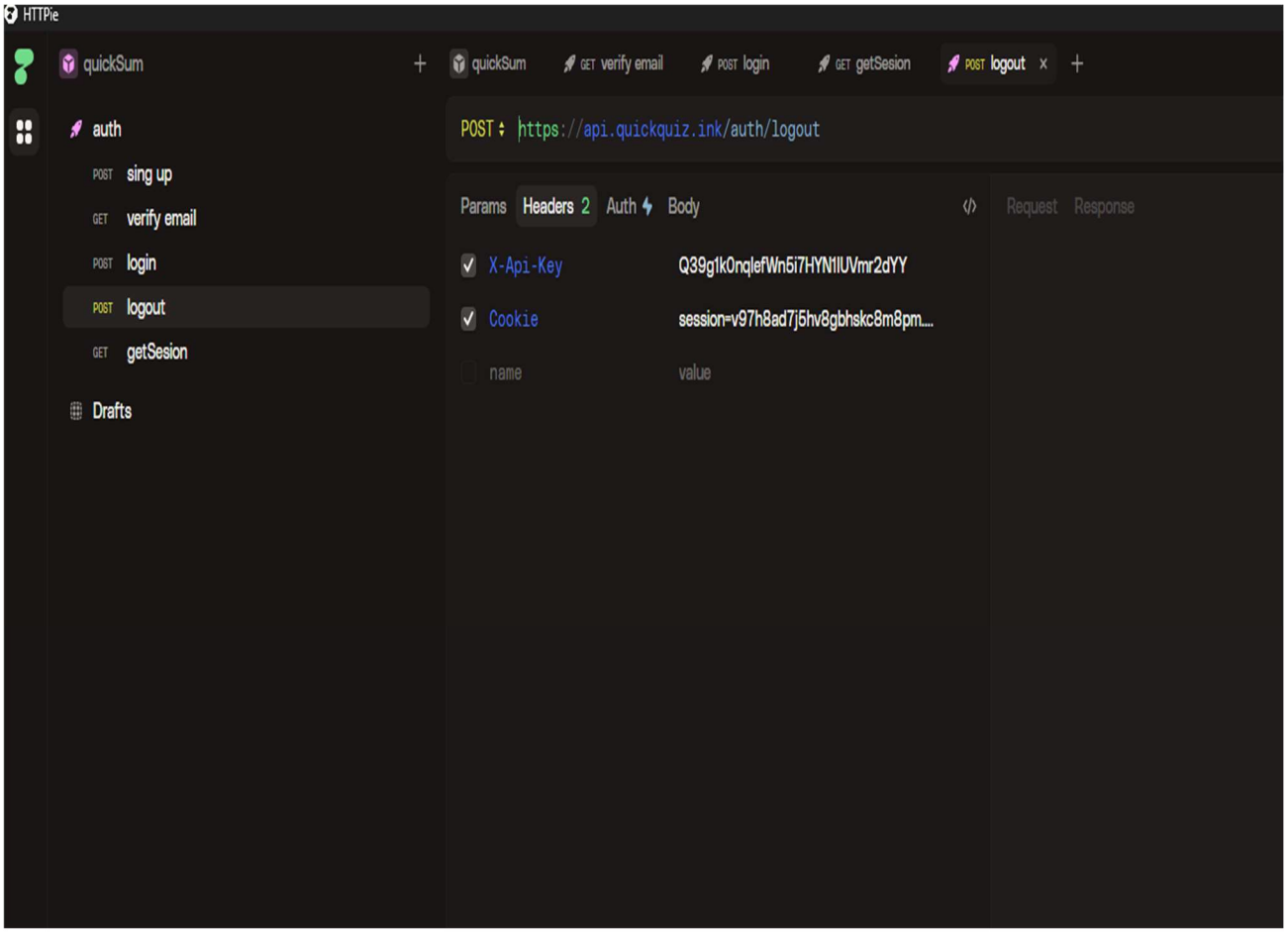


Рис. 2.2 Інтерфейс програми для тестування API HTTPie

2.1.3 Docker Desktop

Docker Desktop[8] використовувався для контейнеризації проєкту, що є стандартом для сучасної веб розробки. Основною метою використання Docker було створення стабільного середовища для розгортання сайту. Завдяки технології контейнеризації всі залежності вебплатформи Next.js були упаковані в окремий образ, що дозволило легко та швидко розгорнути сайт на віддалених серверах без ризику виникнення конфліктів у налаштуваннях середовища.

Використання Docker забезпечило повну ідентичність локальної копії проєкту та версії, що працює на сервері (production), що значно спростило процес підтримки та оновлення системи. Це дозволило зосередитися на розробці функціоналу, не витрачаючи час на ручне налаштування серверного оточення при кожному деплої. Приклад інтерфейсу додатку наведено на рисунку 2.3.

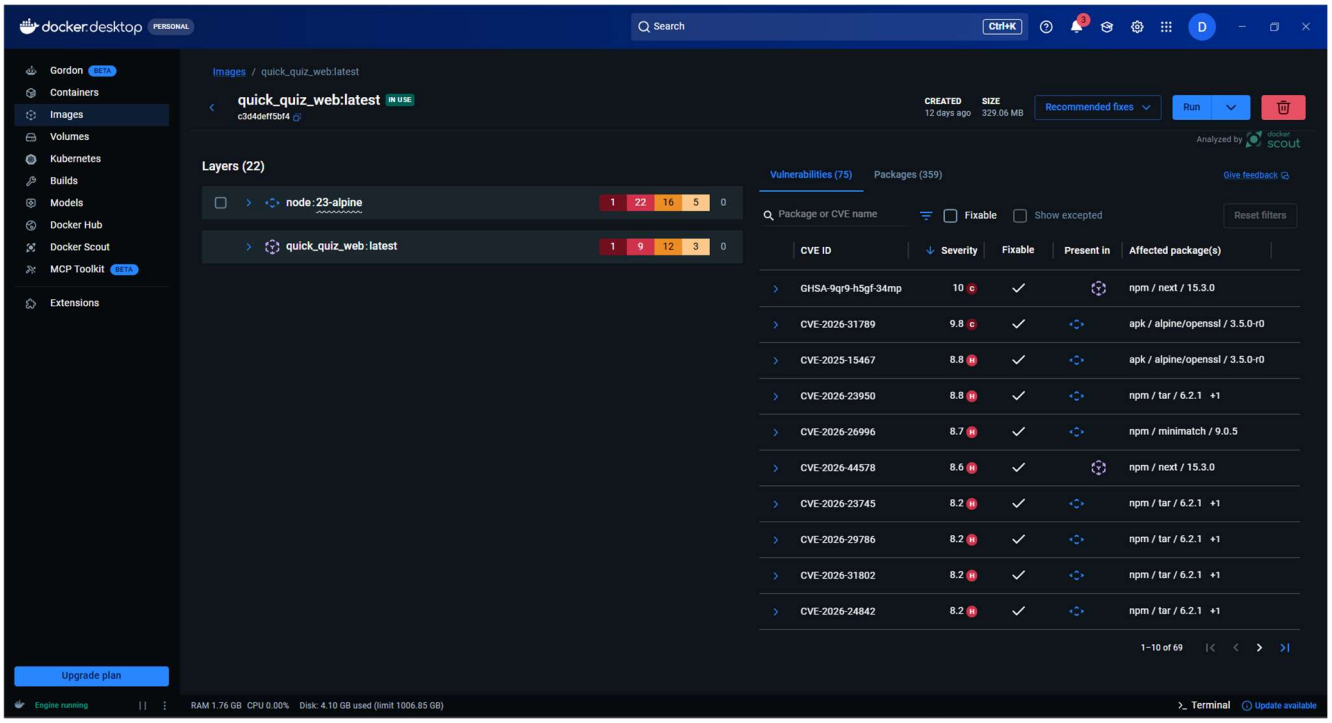


Рис. 2.3 Інтерфейс додатку для управління контейнерами Docker Desktop

2.1.4 GitHub

GitHub[9] — це хмарна платформа для хостингу IT-проєктів та спільної розробки, що базується на системі контролю версій Git. Вона виступає не лише як сховище коду (репозиторій), але й як центр управління життєвим циклом розробки програмного забезпечення.

У межах даної кваліфікаційної роботи GitHub використовувався як центральна платформа для контролю версій та управління життєвим циклом розробки проєкту «QuickSum». Завдяки системі Git було забезпечено повну історію змін коду з можливістю оперативного відкату до стабільних версій, а створення окремих репозиторіїв дозволило ізолювати розробку браузерного розширення від вебплатформи, гарантуючи їхню стабільну синхронізацію через механізм Pull Requests. Особлива увага була приділена автоматизації: інтеграція репозиторію з інструментами Husky та lint-staged дозволила автоматично перевіряти код на відповідність стандартам якості перед кожним підтвердженням змін (commit). Це забезпечило ідентичність версій залежностей між різними середовищами (Node.js/pnpm для розширення та Bun для сайту) та дозволило структурувати

процес розробки відповідно до сучасних промислових стандартів. Приклад сторінки з частинами системи в робочому просторі GitHub наведено на рисунку 2.4.

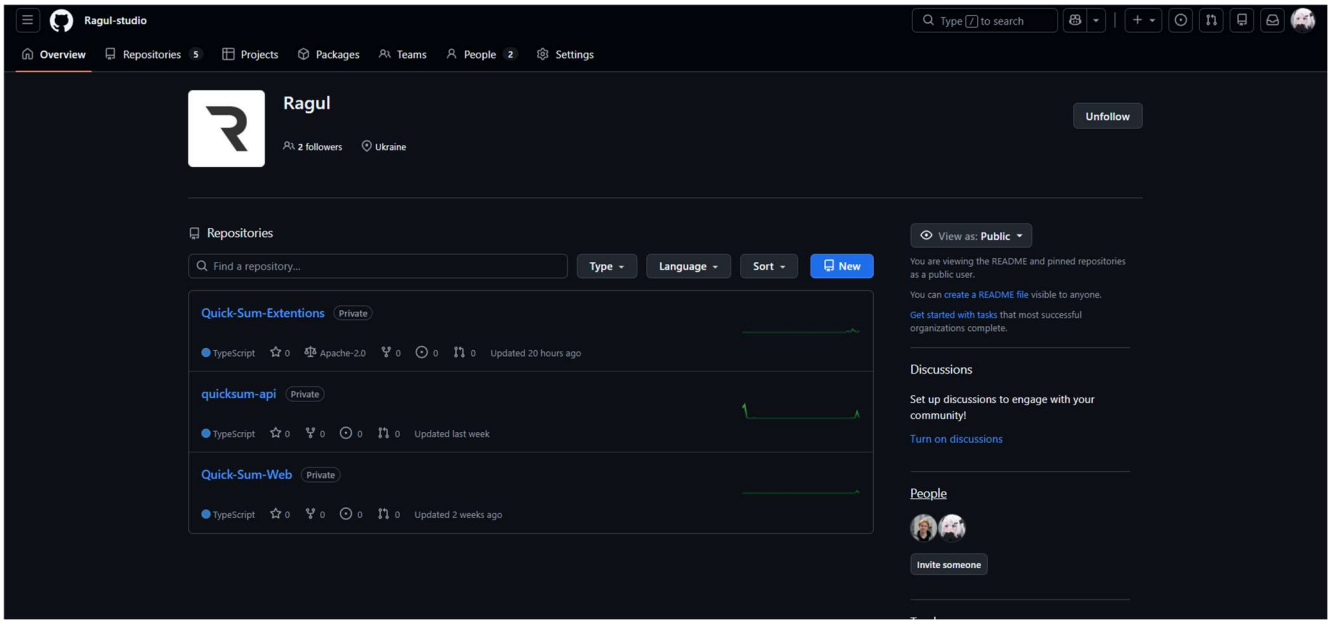


Рис. 2.4 Інтерфейс сторінки з частинами системи в робочому просторі GitHub

2.2 Мови програмування та технології

2.2.1 TypeScript та екосистема типізації даних

TypeScript[10] виступив фундаментальною технологією для забезпечення надійності всього проєкту, реалізуючи концепцію наскрізної типізації (End-to-End Type Safety). Це дозволило ще на етапі проектування описати строгі контракти даних та інтерфейси, що використовуються в обох репозиторіях клієнтської частини. Використання TypeScript дозволяє суттєво знизити когнітивне навантаження на розробника завдяки інтелектуальним підказкам IDE та автоматичній перевірці цілісності об'єктів під час маніпуляцій з DOM-деревом сторонніх ресурсів.

Особливу роль у архітектурі клієнтських застосунків відіграє синергія TypeScript із бібліотекою Zod[11]. Оскільки розширення та вебплатформа отримують дані ззовні (через API або парсинг сторінок), виникає ризик невідповідності типів у рантаймі (наприклад, зміна структури об'єкта сервером).

Zod вирішує цю проблему шляхом створення схем валідації, які гарантують відповідність структур даних очікуваним типам у режимі реального часу. У проєкті «QuickSum» кожна відповідь від сервера, особливо при використанні ендпоїнта /summarize, проходить крізь схеми Zod. Це дозволяє миттєво відловлювати помилки структури, такі як відсутність обов'язкових полів xpathChunks або range, які є критичними для коректного рендерингу хайлайтів на сторінці.

Важливим аспектом реалізації є типізація об'єкта Highlight, який є центральною сутністю системи. Він охоплює не лише контент (текст та сумаризацію), а й метадані для позиціонування: HighlightRange (індекси зміщення в дереві вузлів) та HighlightXPathChunk (масив XPath-шляхів). Це дозволяє системі відновлювати візуальні маркери навіть після перезавантаження сторінки. Завдяки використанню Zod, розширення впевнено оперує статусами генерації (loading, done, error) та режимами сумаризації (short, full), автоматично відкидаючи некоректні дані, що могли б спричинити збій у роботі браузера.

На рисунку 2.5 представлено програмну реалізацію схеми Zod для ендпоїнта /summarize, що базується на структурі інтерфейсу Highlight.

```
// Схема для позиціонування в DOM
const HighlightRangeSchema = z.object({
  startPath: z.array(z.number()),
  startOffset: z.number(),
  endPath: z.array(z.number()),
  endOffset: z.number()
});

const HighlightXPathChunkSchema = z.object({
  xpath: z.string(),
  text: z.string()
});

// Основна схема Highlight
const HighlightSchema = z.object({
  id: z.string(),
  url: z.union([z.string().url(), z.literal("From_App_Summarization")]),
  type: z.enum(["text", "page"]),
  text: z.string(),
  summary: z.string(),
  mode: z.enum(["short", "full"]),
  status: z.enum(["loading", "done", "error"]),
  range: HighlightRangeSchema.optional(),
  xpathChunks: z.array(HighlightXPathChunkSchema).optional(),
  createdAt: z.string()
});

const zodSchema = createSchema({
  "/ai/summarize": {
    input: z.object({
      origin_url: z.string().url(),      You, позавчора • fix(pop-up): auth
      highlight: HighlightSchema,
    }),
    output: z.array(HighlightSchema) // Сервер повертає оновлені об'єкти (summary + id)
  },
  "/ai/resummarize": {
```

Рис. 2.5 Програмна реалізація схеми Zod для ендпоїнта /summarize

2.2.2 Середовища виконання: Bun та Node.js (pnpm)

У проєкті реалізовано диференційований підхід до вибору середовищ виконання для кожного репозиторію. Для розробки та запуску вебплатформи на базі Next.js було обрано інноваційний рантайм Bun[12]. Його використання дозволило критично скоротити час холодного запуску серверних функцій та пришвидшити обробку HTTP-запитів за рахунок вбудованих високопродуктивних API. Bun виступає як надшвидкий менеджер пакетів, що оптимізує дерево залежностей за допомогою двійкових протоколів, що значно пришвидшує процес розгортання вебзастосунку.

Для репозиторію браузерного розширення, що базується на фреймворку Plasmo, було обрано традиційне середовище Node.js[13] із використанням менеджера пакетів pnpm. Такий вибір зумовлений необхідністю суворого дотримання стандартів безпеки браузерних маніфестів (MV3). Оскільки більшість інструментів для трансформації коду розширень оптимізовані саме під Node.js, це гарантує стабільність збірки. Використання pnpm дозволило ефективно керувати залежностями розширення, уникаючи дублювання модулів у файловій системі та забезпечуючи високу швидкість інсталяції пакетів через механізм жорстких посилань (hard links). Приклад бенчмарку швидкості завантаження залежностей одного великого проєкту різними пакетними менеджерами представлено на рисунку 2.6

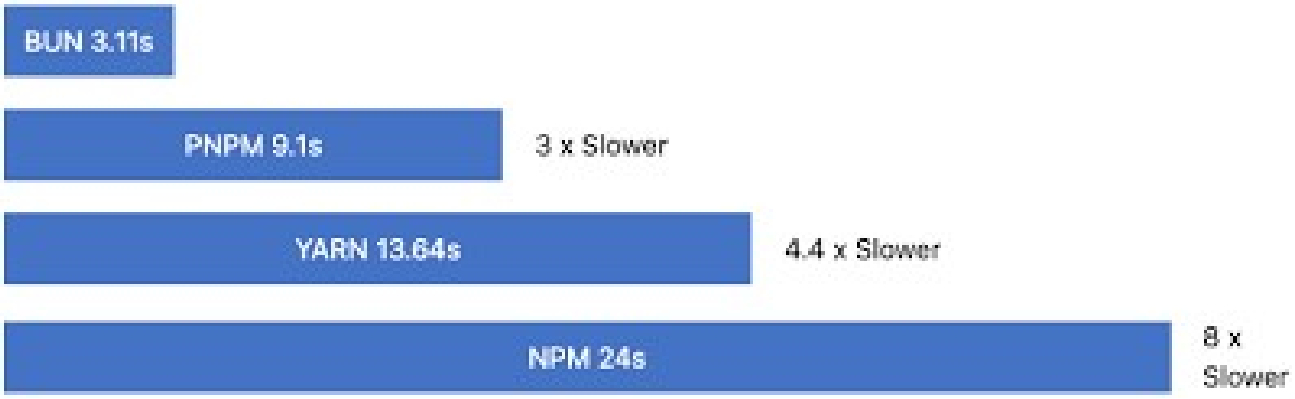


Рис. 2.6 Бенчмарк швидкості завантаження залежностей одного великого проєкту

різними пакетними менеджерами

2.3 Допоміжні бібліотеки та пакети

2.3.1 Клієнтська архітектура: React, Next.js та Plasmo

Веб-інтерфейс системи побудовано на базі сучасної версії фреймворку Next.js 15[14], що використовує архітектуру Turbopack[15] — високоефективний бандлер, написаний на мові Rust. Turbopack працює в рази швидше за Webpack[16], що дозволило досягти миттєвого відображення змін (HMR) під час розробки складного дашборду користувача. Для створення нативного досвіду взаємодії безпосередньо у браузері використано фреймворк Plasmo[17]. Він бере на себе складну логіку ізоляції контентних скриптів та фонових сервіс-воркерів, забезпечуючи автоматичну ін'єкцію React-компонентів у DOM сторонніх сайтів, що повністю виключає конфлікти стилів «QuickSum» із оригінальною верстkою сторінок.

Для побудови сучасного та адаптивного візуального стилю інтерфейсу було обрано фреймворк Tailwind CSS [18]. Його використання дозволяє відійти від традиційного написання громіздких монолітних стилей на користь гнучкої системи низькорівневих утилітарних класів, які застосовуються безпосередньо у розмітці компонентів. Такий підхід не лише прискорює процес проектування інтерфейсів, але й суттєво мінімізує обсяг фінального продуктового CSS-коду завдяки автоматичному видаленню невикористовуваних селекторів під час збирання проєкту (процес Tree Shaking).

З метою підвищення динаміки, створення відчуття глибини простору та покращення загального користувацького досвіду (UX), в архітектуру фронтенду було інтегровано сучасну бібліотеку Motion [19] (відому раніше як Framer Motion). Цей інструмент надає можливість реалізувати складні високопродуктивні анімації на основі фізичних принципів руху (пружинні механізми, інерція та затухання), що сприймається людським оком набагато природніше за стандартні лінійні CSS-переходи. Впровадження цієї бібліотеки дозволило зробити процес появи, згортання та взаємодії з бічною панеллю сумаризації контенту максимально

плавним, безшовним та ергономічним, знижуючи когнітивне навантаження на кінцевого користувача під час роботи з розширенням.

На рисунку 2.7 зображено екранну форму сторінки налаштувань в розширенні з використанням React та Tailwind.

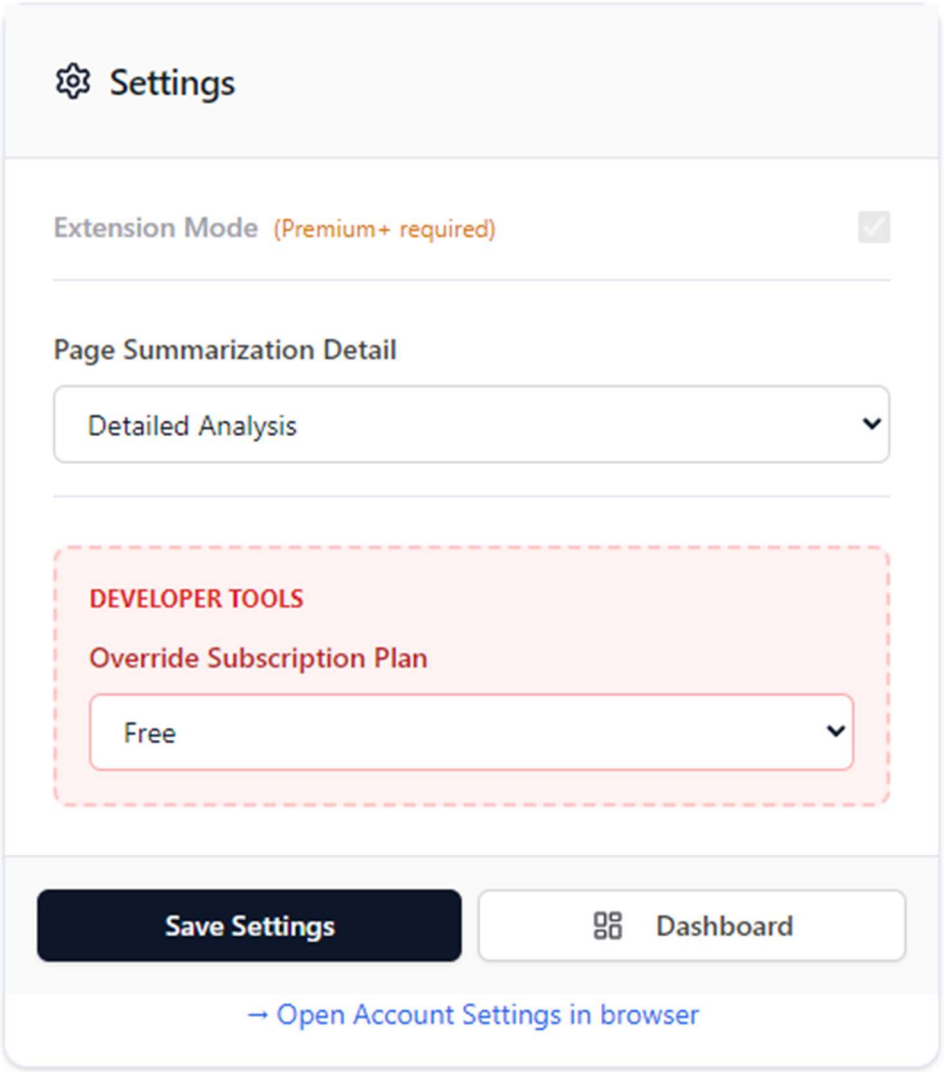


Рис. 2.7 Екранна форма сторінки налаштувань в розширенні

2.3.2 Мережева взаємодія та авторизація через Better Fetch

Взаємодія між клієнтами та серверними ендпоінтами побудована на базі бібліотеки @better-fetch/fetch[20]. Це рішення було обрано через його здатність інтегруватися зі схемами валідації (зокрема, Zod) та автоматично виводити типи

даних для відповідей сервера на основі цих схем. Під час розробки модулів системи саме цей інструмент дозволив вирішити проблему типізації даних та забезпечення стабільності мережевих запитів. Завдяки гнучкому налаштуванню клієнта, було реалізовано автоматичну валідацію вхідних і вихідних даних за допомогою zodSchema, а також механізм повторних запитів (retry) у разі тимчасових збоїв зв'язку. Це дозволяє гарантувати коректність структури даних, що передаються між ізольованим контекстом розширення та API, без необхідності впровадження додаткових громіздких шарів перевірки типу всередині кожного окремого запиту.

На рисунку 2.8 продемонстровано конфігурацію клієнта Better Fetch для виконання запитів зі схемою.

```
const $fetch = createFetch({
  baseURL: process.env.PLASMO_PUBLIC_SERVER_URL,
  schema: zodSchema,
  retry: {
    type: "linear",
    attempts: 3,
    delay: 1000
  },
  throw: true,
  plugins: [
    logger({
      enabled: process.env.NODE_ENV === "development"
    })
  ]
});

export default $fetch;
```

Рис. 2.8 Конфігурація клієнта Better Fetch

2.3.3 Стейт-менеджмент та UI-екосистема

Для ефективного керування реактивним станом обох клієнтських застосунків було обрано Zustand[21] - мінімалістичний та високопродуктивний менеджер станів, що функціонує на основі концепції кастомних React-хуків.

Базою для проектування графічного інтерфейсу користувача став набір

низькорівневих примітивів Radix UI, зокрема такі модулі як Tabs (вкладки), Dialog (модальні вікна) та Progress (індикатори завантаження). Оскільки ці компоненти є повністю безстильовими, вони надали розробнику абсолютну свободу для реалізації унікального візуального дизайну за допомогою кастомних CSS-стилів. Водночас вони гарантують повну вбудовану відповідність міжнародним стандартам доступності вебконтенту (WAI-ARIA) та коректну підтримку керування з клавіатури, що підвищує інклюзивність цифрового продукту.

Доповнює та завершує візуальну систему уніфікована бібліотека іконок Lucide React. Вона забезпечує сувору візуальну цілісність, правильний геометричний баланс та однакову товщину ліній графічних маркерів в усіх розділах програми. Окрім естетичної функції, вибір Lucide React зумовлений підтримкою технології Tree Shaking, що дозволяє імпортувати лише активовані в коді іконки, суттєво зменшуючи фінальний розмір збірки розширення та вебдодатка.

2.3.4 Тестування та автоматизація контролю якості

Для забезпечення надійності кодової бази в обох репозиторіях впроваджено тестовий стек на основі Vitest[22] та jsdom. Це дозволило виконувати юніт-тести для React-компонентів та алгоритмів обробки тексту в середовищі, що імітує браузер. Процес розробки жорстко контролюється через інструменти автоматизації Husky[23] та lint-staged. Вони інтегровані в життєвий цикл Git через Git Hooks які представляють собою вбудовані скрипти-тригери, що автоматично спрацьовують під час настання ключових подій у системі контролю версій.

У даному проєкті головним інструментом автоматизації став хук pre-commit, який перехоплює керування безпосередньо в момент виконання команди створення коміту, але ще до фіксації змін в локальній історії розробника.

Нативні хуки Git мають суттєвий недолік, оскільки вони зберігаються в ізольованій локальній директорії, яка за замовчуванням не відстежується системою контролю версій і не передається іншим розробникам при клонуванні репозиторію.

Використання Husky дозволяє повністю розв'язати цю проблему шляхом перевизначення шляху до хуків у глобальній конфігурації Git та винесення

конфігураційних файлів у публічну кореневу папку проєкту, що гарантує синхронізацію та ідентичність правил перевірки для всіх учасників розробки. Приклад популярних Git Hooks приведенно на рисунку 2.9.

Під час активації хука pre-commit утиліта Husky делегує виконання завдань інструменту lint-staged. Цей крок є критично важливим для оптимізації швидкості розробки, оскільки замість перевірки всього проєкту, lint-staged ізолює та аналізує виключно ті змінені файли, які вже було додано до індексу за допомогою команди git add. Таким чином, при кожному коміті автоматично запускаються ESLint[24] для пошуку логічних або синтаксичних помилок і потенційних багів у коді, а також Prettier для автоматичного форматування стилю коду відповідно до заданих конфігурацій. Якщо в процесі аналізу ESLint виявляє критичні помилки, або якщо підключені через цей же конвеєр модульні тести Vitest завершуються невдачею, скрипт хука повертає ненульовий код помилки. Це призводить до негайного переривання транзакції створення коміту, повністю блокуючи потрапляння неробочого коду в локальну гілку. Такий підхід гарантує, що до репозиторіїв потрапляє лише чистий, синтаксично коректний та відтестований код, що повністю відповідає стандартам якості, поставленим в проєкті.

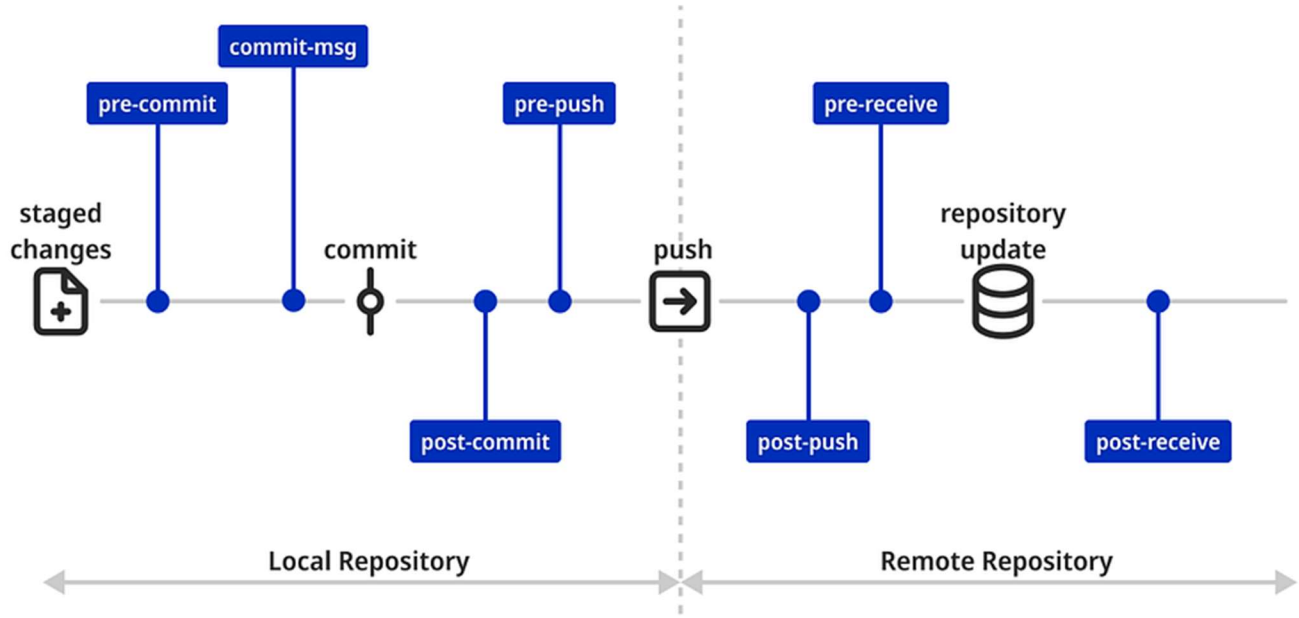


Рис. 2.9 Точки виконання поширених Git-хуків

3 ПРОЄКТУВАННЯ

3.1 Вимоги до програмного забезпечення

Основою для успішного проєктування клієнтської частини застосунку (Frontend та браузерного розширення) є формування чітких функціональних та нефункціональних вимог. Ці вимоги базуються на результатах аналізу взаємодії користувача з вебсторінками, потреб цільової аудиторії та необхідності забезпечення безшовної інтеграції між розширенням і вебзастосунком.

3.1.1 Функціональні вимоги

Функціональні вимоги визначають базову поведінку клієнтської частини системи та перелік завдань, які вона повинна виконувати. Frontend та браузерне розширення повинні забезпечувати наступні ключові функції:

1. Забезпечення зручного інтерфейсу реєстрації, авторизації та автентифікації користувачів. Можливість повноцінного управління профілем, включаючи перегляд, редагування даних та завантаження аватара.
2. Можливість переглядати історію попередніх сумаризацій (через панель розширення або вебзастосунок) та взаємодіяти з нею (видалення, створення ресумаризацій).
3. Миттєве отримання та захоплення виділеного тексту зі сторінки браузера, а також повний парсинг основного контенту сторінки. Здійснення попередньої обробки та очищення текстових даних (видалення HTML-тегів, реклами, навігаційних елементів) перед передачею на сервер.

4. Ініціація запиту на узагальнення тексту, відображення стану завантаження та фінального результату безпосередньо поверх активної сторінки (через спливаюче меню та бічну панель).
5. Взаємодія з серверною частиною для відправки запитів і отримання відповідей, з належною обробкою помилок та відображенням станів виконання операцій.
6. Підтримка спеціального тестового режиму (Dev Mode) для роботи застосунку без активного підключення до бекенду (з використанням заглушок типу FakeSummarizer).

3.1.2 Нефункціональні вимоги

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

1. Оперативність парсингу та видобування тексту зі сторінки повинна займати не більше 500 мс для забезпечення миттєвого відгуку інтерфейсу.
2. Система повинна мати відмовостійку архітектуру з ефективною обробкою помилок та мережевих збоїв, гарантуючи збереження локального стану (хайлайтів) навіть при втраті з'єднання.
3. Забезпечення безпеки клієнтської частини через захист конфіденційних даних (використання безпечних сховищ для токенів), валідацію вводу та ізоляцію контекстів виконання розширення.
4. Frontend-архітектура повинна бути модульною (Feature-Sliced Design), де кожна функціональність винесена в окремий незалежний модуль для полегшення підтримки.
5. Розширення має підтримувати роботу у сучасних браузерах (Chrome, Firefox) та браузерах на їх основі (Brave, Zen) з обов'язковою підтримкою стандарту ES2019.
6. Вебзастосунок повинен підтримувати контейнеризацію для зручного розгортання в хмарних середовищах.

Діаграму варіантів використання наведено на рисунку 3.1.

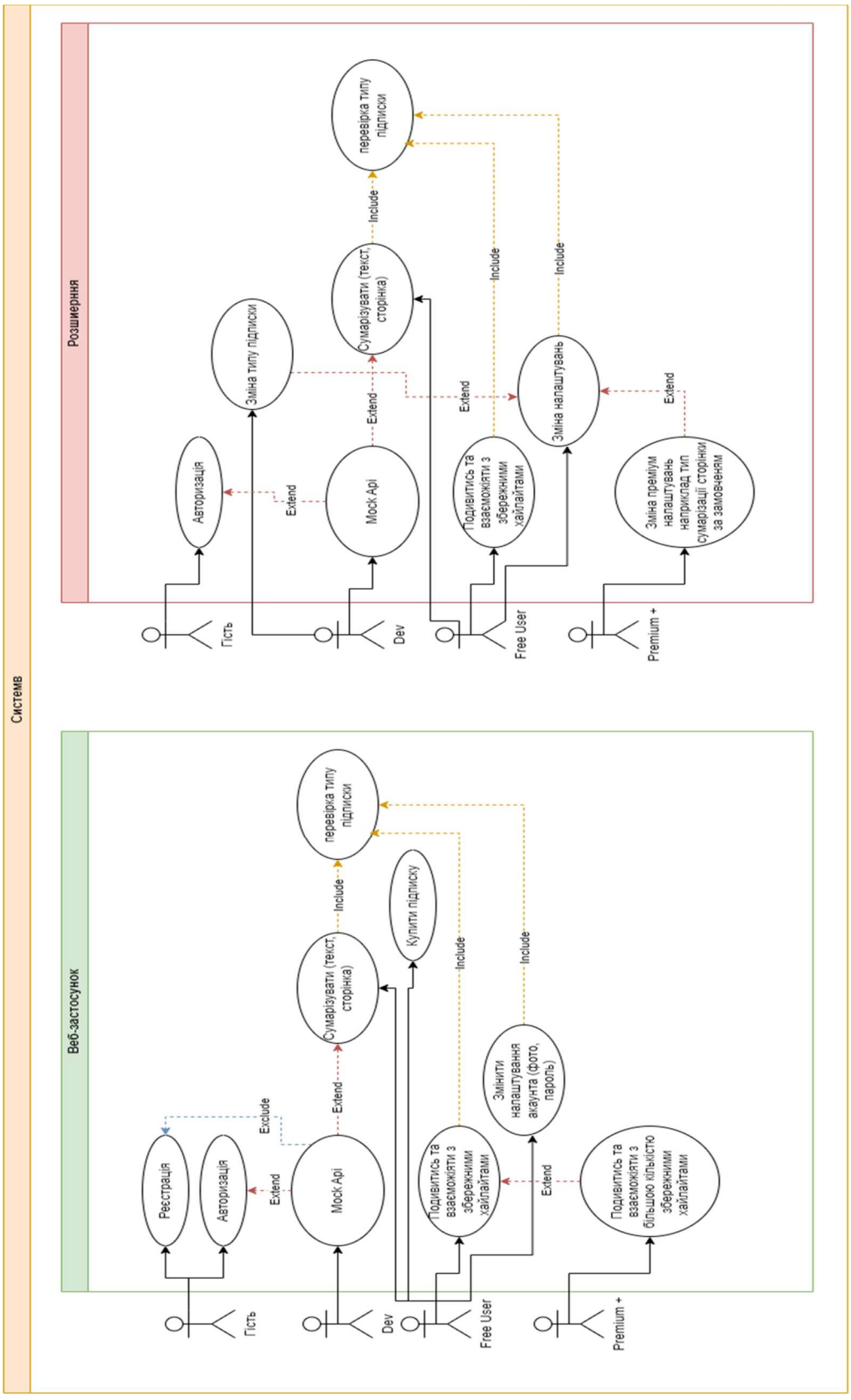


Рис. 3.1 Діаграма варіантів використання

3.2 Проєктування архітектури застосунку

Архітектура клієнтської частини системи реалізована за допомогою двох взаємопов'язаних, але відокремлених платформ: браузерного розширення для прямої взаємодії з контентом та повноцінного вебзастосунку (Web App) для управління акаунтом і підписками.

Браузерне розширення побудоване на базі сучасного фреймворку Plasmo, що дозволяє використовувати React для створення користувацьких інтерфейсів (UI), які ін'єктуються безпосередньо у вебсторінки. Архітектура розширення розділена на три основні контексти:

1. Content Scripts (Контентні скрипти): Відповідають за взаємодію з DOM-деревом активної сторінки. Вони відстежують виділення тексту користувачем, відмальовують спливаюче меню (FloatingMenu) біля курсора та відображають бічну панель (SidePanel) з результатами.
2. Background Scripts (Фонові сервіси): Працюють ізольовано як Service Workers. Виконують роль диспетчера повідомлень, обробляючи глобальні події (наприклад, відкриття сторінки історії) та координуючі роботу між різними вкладками браузера.
3. Popup / Options (Вікна розширення): Надають швидкий доступ до базових налаштувань, статусу авторизації та увімкнення/вимкнення режиму розробника.

Вебзастосунок (Сайт) реалізований на базі фреймворку Next.js з використанням App Router. Він виконує роль головного порталу системи, надаючи доступ до лендінгу, повної історії сумаризацій, налаштувань профілю та платіжного шлюзу (інтеграція з Paddle для обробки підписок). Завдяки Server-Side Rendering (SSR) та статичній генерації, сайт забезпечує швидке завантаження та високі показники SEO. Вебзастосунок контейнеризований і підготовлений до розгортання через Docker-оркестратори.

3.3 Архітектура клієнтської частини (Frontend та Розширення)

Основою стабільності та високої продуктивності розширення є строга модульна структура, що розділяє логіку роботи з DOM, управління станом та взаємодію з API.

Діаграма фронтенд архітектури наведено на рисунку 3.2.

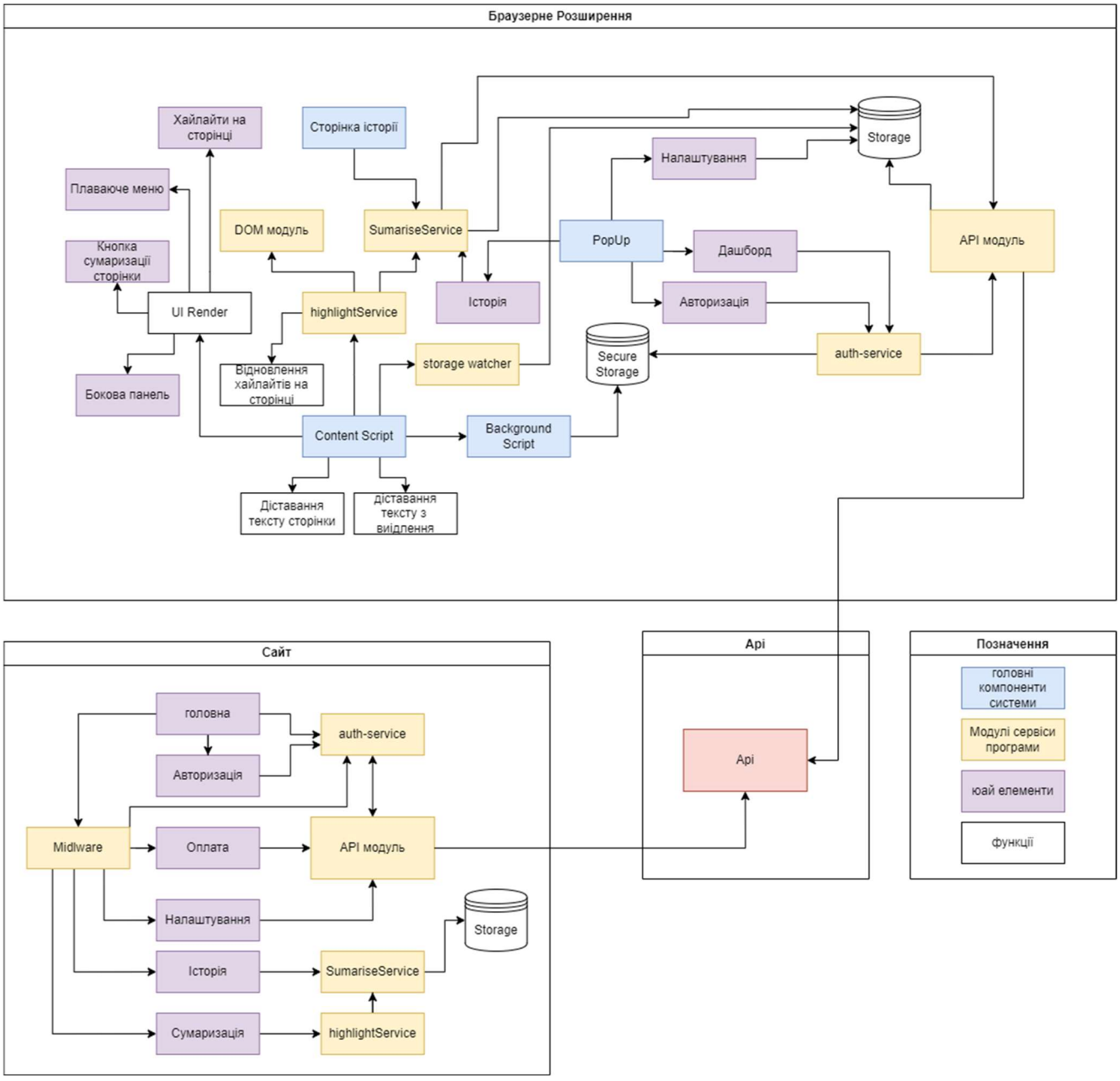


Рис. 3.2 Діаграма фронтенд архітектури

3.3.1 Модуль захоплення та обробки контенту (Highlight Service)

Найскладнішим компонентом контентного скрипту є HighlightService, який відповідає за видобування тексту та його візуальне маркування на сторінці.

При виділенні тексту користувачем, система не лише захоплює текстове значення, але й формує об'єкт Highlight, який зберігає просторові координати виділення. Для цього використовується гібридний підхід: зберігаються як стандартні об'єкти Range, так і масиви HighlightXPathChunk, що містять XPath-шляхи до конкретних текстових вузлів DOM.

Це забезпечує збереження виділень навіть після перезавантаження сторінки або динамічної зміни структури сайту. Якщо точний Range відновити неможливо, система використовує фолбек (fallback) механізм, шукаючи текстові фрагменти за збереженими XPath-координатами та огортаючи їх у спеціальні елементи. Блок схему отримання хайлайту з виділення показано на рисунку 3.3

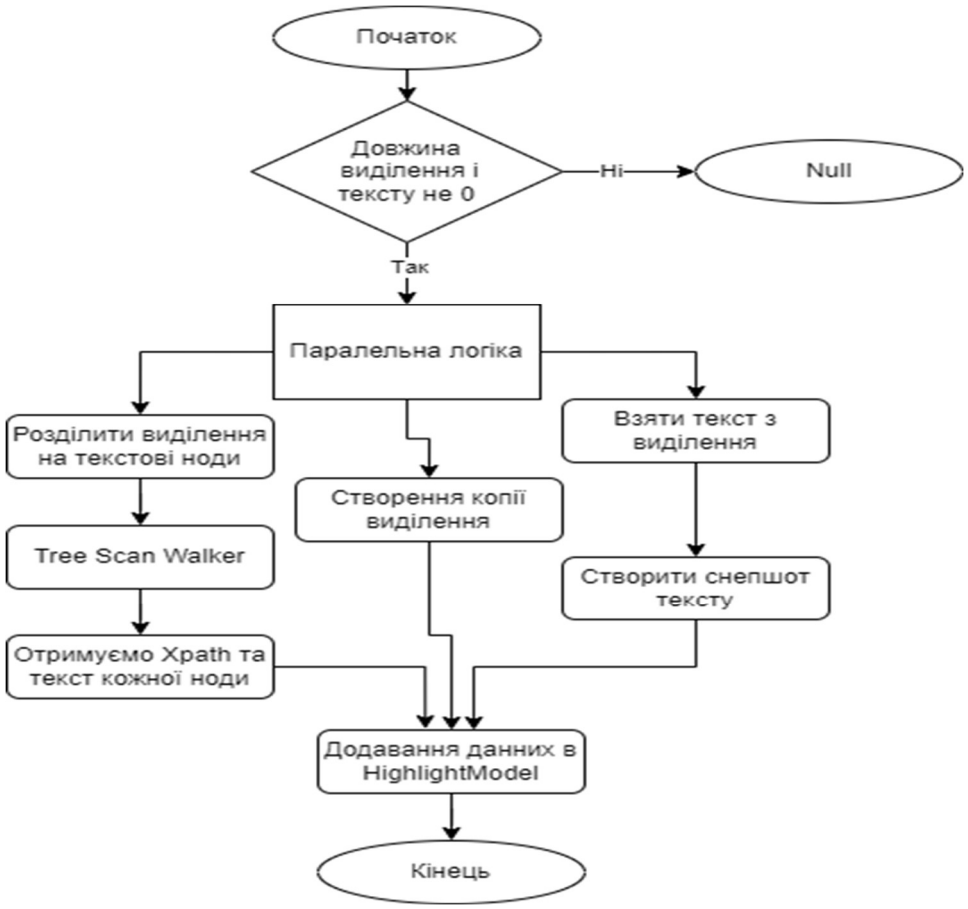


Рис. 3.3 Блок схема алгоритма отримання хайлайту з виділення

Для сумаризації всього масиву інформації на вебсторінці було спроектовано та реалізовано високоефективний алгоритм екстракції головного змісту, впроваджений у кодову базу у вигляді спеціалізованої функції extractMainContentInText.

Проблема якісного вилучення цільових даних є однією з ключових при аналізі сучасних вебдокументів, оскільки реальні сторінки інтернет-ресурсів перевантажені допоміжними елементами, які не несуть смислового навантаження для кінцевого користувача, але здатні суттєво спотворити результати подальшої обробки або сумаризації тексту штучним інтелектом.

З метою запобігання деструктивному впливу на поточний стан інтерфейсу та уникнення небажаного перерендерінгу сторінки розробником, першим етапом роботи алгоритму є створення повного ізольованого клону поточного DOM-дерева документа в оперативній пам'яті за допомогою методу cloneNode(true). Над цією скопійованою структурою виконується глибоке попереднє очищення від візуального та технічного шуму. Алгоритм використовує заздалегідь визначений масив селекторів noiseSelectors, цілеспрямовано ідентифікуючи та повністю видаляючи з дерева службові теги (script, style, noscript, iframe), елементи глобальної навігації (header, footer, nav, aside), а також інтегровані рекламні модулі, банери, віджети та блоки коментарів (.ads, .sidebar, .menu, .comments, .share).

Після завершення фази грубої фільтрації, очищене DOM-дерево передається на вхід системі евристичного оцінювання, яка за допомогою методу querySelectorAll вилучає масив потенційно контентних блоків, таких як article, main, p та заголовки h1-h3. Цей механізм виконує обхід структури та розраховує математичну вагу (релевантність) за алгоритмом Arc-style scoring. Базова вага кожного текстового контейнера визначається чистою довжиною його контенту (text.length), після чого система нараховує додаткові вагові коефіцієнти залежно від семантичної значущості тегу: текстові абзаци <p> отримують бонус у +20 балів, а комплексні контейнери <article> — +100 балів. Для відсікання інформаційного шуму на етапі скорингу застосовується первинна фільтрація, що повністю ігнорує блоки, довжина тексту в яких становить менше 40 символів.

Заключний етап функціонування конвеєра присвячений багаторівневій текстовій нормалізації та лінеаризації відібраних високорелевантних вузлів, чий сумарний показник скорингу перевищив жорсткий поріг у 60 балів. Після об'єднання елементів у суцільний текстовий масив за допомогою роздільника join("\n\n"), система виконує комплексне очищення отриманого рядка через регулярні вирази (RegExp). Зокрема, здійснюється заміна нерозривних пробілів (\u00a0) на стандартні, усуваються надлишкові горизонтальні та вертикальні пробіли, а також нормуються відступи між абзацами. Крім того, алгоритм розбиває текст на окремі рядки для проведення фінального семантичного аудиту: відсікаються занадто короткі фрагменти (менше 25 символів) та примусово видаляється залишкове інтерфейсне сміття, що містить маркери «Останні новини», «Powered by», «RSS», «комент» або «Sign in».

Вузли, які успішно пройшли всі етапи фільтрації та скорингу, визнаються текстовим ядром сторінки, після чого з них акумулюється виключно релевантний контент, повністю очищений від технічного контексту сайту. Оскільки цей алгоритм виконується безпосередньо у браузері користувача під час активного серфінгу, критично важливою вимогою до нього є забезпечення високої швидкодії.

Весь описаний комплексний конвеєр обробки від клонування та деструктуризації до фінальної нормалізації рядка регулярними виразами було оптимізовано для виконання в межах жорсткого часового ліміту у 500 мс, що робить його роботу абсолютно непомітною для користувача.

Детальна логіка взаємодії, етапи розгалуження та послідовність операцій цього процесу наочно відображені на блок-схемі алгоритму, яка показана на рисунку 3.4.

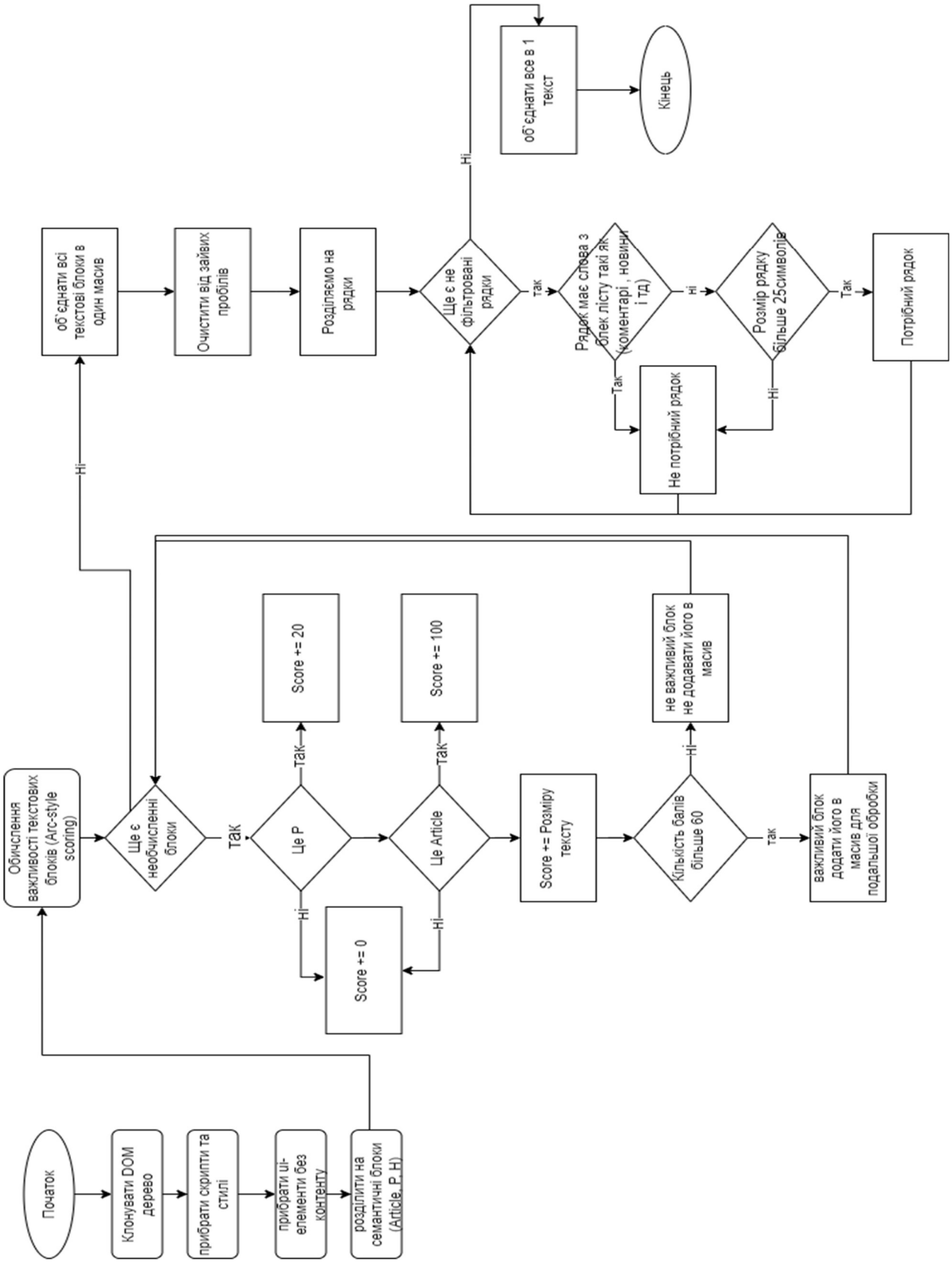


Рис. 3.4 Блок схема алгоритма отримання тексту зі сторінки

3.3.2 Життєвий цикл обробки взаємодії користувача через розширення

Взаємодія користувача з інтерфейсом розширення базується на концепції чітко структурованого, асинхронного та подієво-орієнтованого конвеєра (пайплайну) обробки даних. Керування цим процесом повністю делеговано головному контентному скрипту, який розроблено у вигляді центрального React-компонента Main.tsx. Даний скрипт функціонує як головний архітектурний контролер клієнтської сторони. Він координує взаємодію ізольованих модулів, динамічно відстежує стани інтерфейсу користувача та виконує інтеграцію функціональних елементів розширення у DOM-структуру сторонніх вебдокументів.

Перший етап роботи конвеєра активується безпосередньо під час завантаження або оновлення цільової вебсторінки в браузері.На цій станії архітектурний вузол здійснює миттєвий аудит поточного URL-посилання, зіставляючи домен із локальним статичним реєстром винятків, описаним у файлі blackList.ts. Така перевірка дозволяє превентивно блокувати виконання стороннього коду на відеоплатформах, вебдодатках зі штучним інтелектом (Gemini, ChatGPT), службових чи конфіденційних інтернет-ресурсах, де збір та аналіз текстового вмісту є небезпечним чи не має сенсу. Паралельно з аудитом адреси ініціюється процедура перевірки сесії користувача. Компонент звертається до захищеного локального сховища браузера для верифікації токенів доступу. Це забезпечує визначення рівня прав клієнта (isUserAccess або isDevModeAccess) та конфігурацію інтерфейсу ще до моменту першої безпосередньої взаємодії з функціоналом.

Для мінімізації затримок візуального рендерингу первинне зчитування налаштувань здійснюється оптимістично з локального сховища, що запобігає «миготінню» елементів керування. Одразу після цього клієнтська частина ініціює фоновий запит крос-валідації через шину повідомлень до ізольованого обробника verifySettings. Це дозволяє скоригувати реактивний стан у разі виявлення спроб маніпуляції даними з боку сторонніх вебсторінок.

Другий етап конвеєра присвячений динамічному перехопленню дій користувача та аналізу поведінкових тригерів у межах сторінки.

В архітектурі клієнтської частини реалізовано два базові сценарії ініціації запитів. Перший сценарій покладається на роботу кастомного React-хука useSelection, який у реальному часі прослуховує події виділення текстових фрагментів. Як тільки користувач завершує виділення тексту, хук розраховує просторові координати курсора на екрані, ініціюючи рендеринг контекстного меню FloatingMenu у заданій точці. Другий сценарій розрахований на комплексне опрацювання всього документа. При натисканні на інтегровану кнопку "Сумаризувати сторінку" система ігнорує виділення окремих абзаців. У цей момент автоматично активується алгоритм парсингу та екстракції всього доступного текстового масиву за допомогою спеціалізованого контент-екстрактора.

На третьому етапі конвеєр переходить у фазу активної асинхронної обробки даних та формування вихідних об'єктів. Для підвищення показника відгуку інтерфейсу (responsiveness) в архітектурі застосовано паттерн оптимістичного інтерфейсу (Optimistic UI). Програма не очікує завершення обчислень на сервері, а миттєво генерує унікальний ідентифікатор UUID для майбутнього хайлайту.

Одночасно з цим бічна панель SidePanel переводиться у стан очікування із відображенням тексту "Генеруємо підсумок...". У фоновому режимі паралельно надсилається асинхронний запит до модуля apiService.ts.

Для оптимізації витрат під час відладки архітектура підтримує інструменти розробника, де реальний виклик нейромережі прозоро підміняється імітаційним сервісом FakeSummarizer.

Фінальний, четвертий етап життєвого циклу конвеєра відповідає за персистентність даних та синхронізацію внутрішніх станів.

Після успішного виконання асинхронної операції та повернення згенерованого стислого змісту мовною моделлю, результати проходять первинну валідацію структури. Після цього контентний скрипт записує нові дані у локальне сховище (LocalStorage) браузера, що гарантує збереження історії та швидке кешування інформації.

Останнім кроком оновлюється реактивний стан (state) всього компонента. Завдяки цьому бічна панель миттєво виходить із режиму очікування та презентує користувачу фінальний структурований текст.

Загальний алгоритм запуску та життєвого циклу головного контентного скрипту наочно проілюстровано на блок-схемі, що наведена на рисунку 3.5.

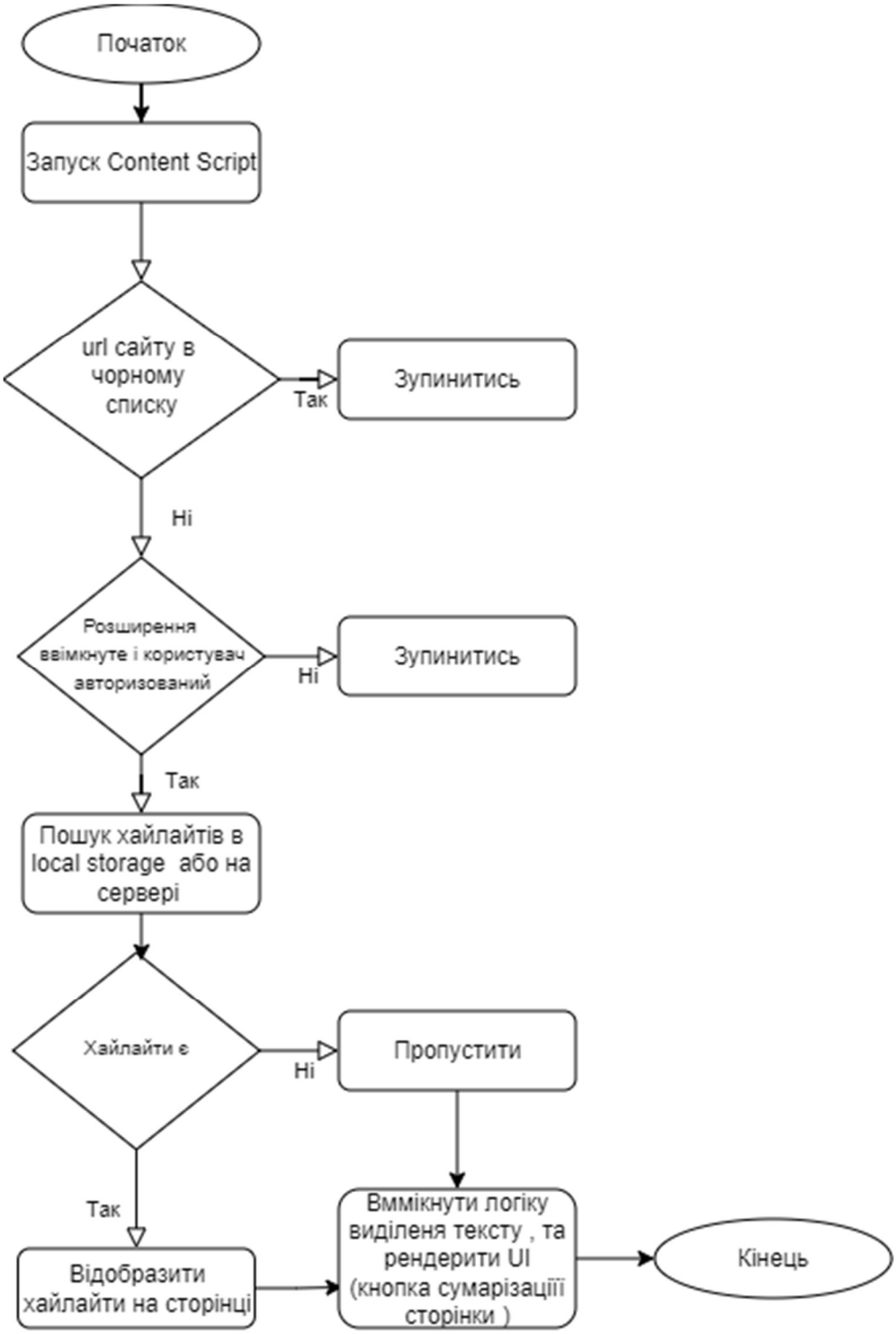


Рис. 3.5 Блок схема запуску контент скрипту на сторінці

3.3.3 Управління станом та безпека (Storage & Auth)

Уся архітектура побудована навколо надійного управління станом за допомогою кастомних хуків та сервісів зберігання:

1. Для збереження налаштувань, статусу розширення та історії використовуються адаптери local-storage-store та синхронізований storage-store.
2. Чутливі дані, такі як JWT-токени доступу, а також дублюючі копії конфігурацій користувача зберігаються виключно через механізми secure-storage-store. Це сховище ізольоване від цільових вебсторінок, що дозволяє використовувати його як еталонний сервіс для верифікації локальних налаштувань. У разі спроби користувача або шкідливого XSS-скрипту вручну модифікувати параметри розширення у LocalStorage, система виявляє невідповідність за допомогою secure-storage-store та примусово відновлює оригінальні значення, блокуючи підміну конфігурації. Модуль автентифікації (auth-service.ts) забезпечує інтеграцію між вебзастосунком та браузерним розширенням. Після виконання POST-запиту на авторизацію розширення використовує браузерне API chrome.cookies.get для перехоплення сесійної куки з назвою session із домену сайту. Отримане значення автоматично копіюється в ізольоване сховище secureStorage як authToken для подальшої ручної передачі в заголовках запитів до API.
3. Розділення потоків авторизації та реєстрації: Процеси створення нового акаунта повністю централізовані й відбуваються виключно на стороні вебсайту, куди розширення перенаправляє користувача за допомогою кнопки «Реєстрація». Натомість авторизаційні сесії працюють у взаємовиключному режимі, оскільки сервер підтримує обмеження на один активний сеанс для одного акаунта. Через відсутність повноцінного спільного використання кук між доменом та розширенням, успішний вхід або оновлення сесії на сайті призводить до автоматичного скидання та деавторизації поточного токена всередині браузерного розширення.

4. У разі використання режиму розробника (Dev Mode), авторизація відбувається за допомогою локально згенерованих mock-токенів без звернення до мережі.

3.3.4 Взаємодія користувача з вебзастосунком (Web UI)

Вебзастосунок, побудований на базі Next.js (App Router), виступає не лише панеллю управління профілем, але й повноцінним інструментом для взаємодії з ядром системи. Архітектура клієнтських сторінок ("use client") орієнтована на забезпечення високої інтерактивності та швидкого зворотного зв'язку.

Основні сценарії взаємодії користувача з сайтом поділяються на такі модулі:

1. Головна сторінка (Landing Page):
   1.1. Слугує основною точкою входу та презентації продукту. Структура сторінки побудована з незалежних компонентів для забезпечення високої швидкості рендерингу: головний екран (Hero), опис ключових можливостей (Features), порівняння з аналогами (Comparing), інформація про тарифні плани (Pricing), секція завантаження розширення (Download) та блок відповідей на часті запитання (FAQ).
2. Модуль ручної сумаризації (Summarize Page):
   2.1. Надає користувачам можливість генерувати підсумки безпосередньо на сайті, без використання розширення.
   2.2. Гнучкість вводу: Реалізовано перемикач типу контенту, що дозволяє користувачу надати як пряме посилання на вебсторінку (з вбудованою валідацією URL), так і вставити об'ємний «сирий» текст (Raw Text).
   2.3. Управління форматом: Користувач може ініціювати створення короткого базового підсумку (Short) або розгорнутого детального аналізу (Detailed).
   2.4. Захист маршрутів: Доступ до функціоналу захищено перевіркою активної сесії, з автоматичним відображенням екрану "Access Denied" для неавторизованих відвідувачів (за винятком тестового режиму isDevMode).

3. Модуль управління історією (History Page):
   3.1. Комплексний інтерфейс для роботи зі збереженими результатами, побудований з використанням асинхронного завантаження (Suspense) та синхронізації стану з URL-параметрами (useSearchParams).
   3.2. Фільтрація та пошук: Користувач може здійснювати текстовий пошук по історії, фільтрувати записи за джерелом (вебсторінка або звичайний текст) та змінювати порядок сортування (за зростанням чи спаданням дати створення).
   3.3. Детальний перегляд та версіонування: При виборі конкретного запису відкривається компонент SummaryDetailView, який дозволяє переглядати поточний результат, копіювати його в буфер обміну, а також навігувати між різними версіями сумаризації одного й того ж тексту (наприклад, між короткою та детальною версією). приклад показано на рисунку 3.6.
   3.4. Безпечне видалення: Реалізовано оптимістичний алгоритм видалення записів. Для уникнення помилок маршрутизації (зокрема, проблеми "битих посилань"), система спочатку очищує URL від ID видаленого елемента (router.replace("/history")), і лише потім видаляє дані з локального сховища та стейту, інформуючи користувача через систему спливаючих повідомлень (Toasts).
   3.5. Ресумаризація: Можливість повторно звернутися до API для генерації нового підсумку (в іншому об'ємі) на основі вже збереженого в історії вихідного тексту.

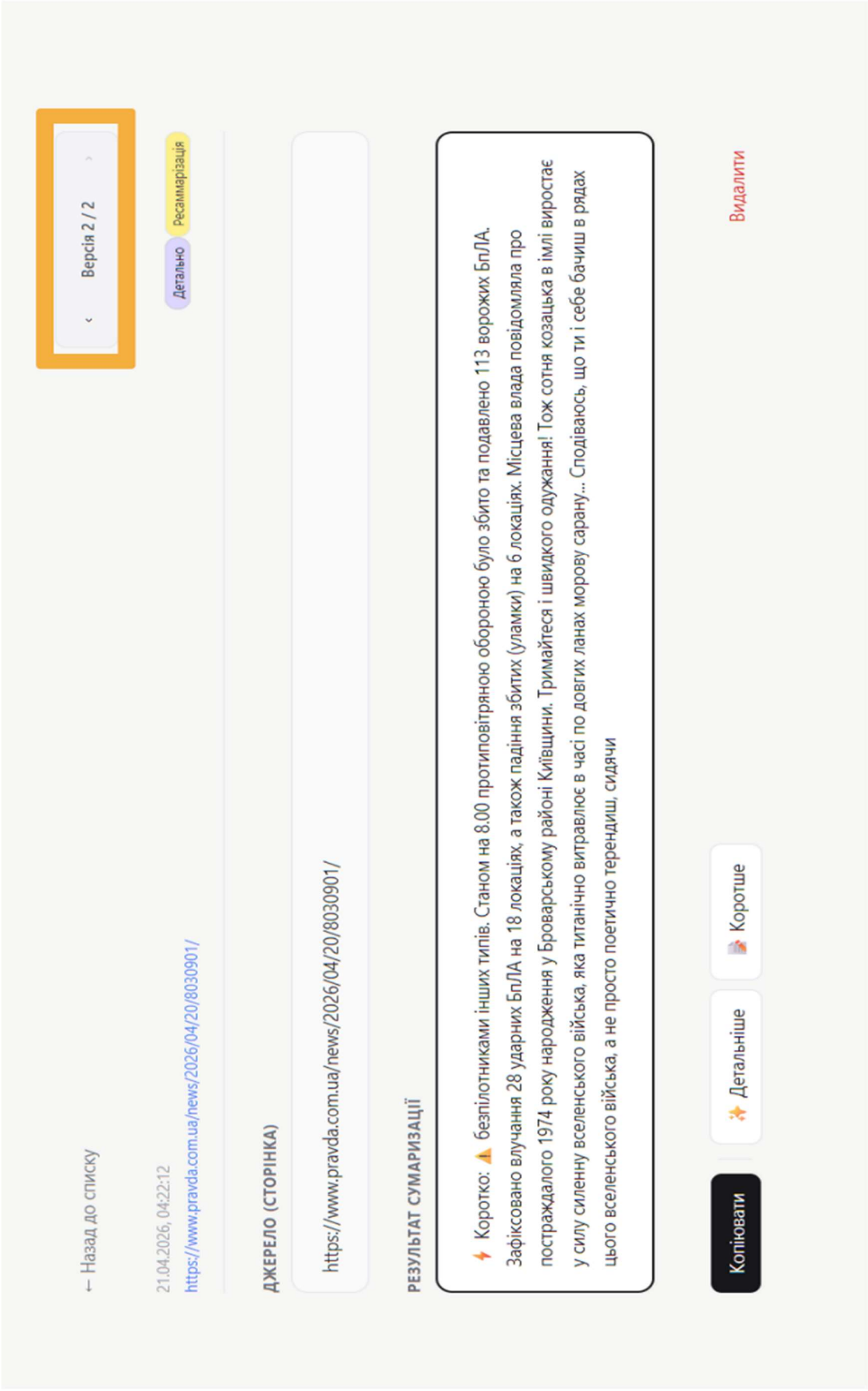


Рис. 3.6 Екранна форма компоненту SummaryDetailView

4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

4.1 Програмна реалізація клієнтської частини

Програмна реалізація клієнтської частини програмного комплексу «QuickSum» базується на комплексному використанні бібліотеки React та сучасних фреймворків Next.js і Plasmo, кожен з яких відповідає за свій сегмент загальної архітектури системи. Зокрема, вебплатформа розроблена з використанням Next.js, що надає широкі можливості для серверного рендерингу та оптимізації завантаження сторінок, тоді як клієнтська частина браузерного розширення побудована на базі спеціалізованого фреймворку Plasmo.

Вибір цього технологічного стеку був обумовлений жорсткими архітектурними вимогами проєкту, серед яких головними є забезпечення високої реактивності користувацького інтерфейсу, зручності управління глобальним та локальним станами компонентів, а також гарантування надійної ізоляції контентних скриптів (content scripts) від оригінального DOM-дерева цільових вебсторінок, що унеможливлює виникнення конфліктів стилів або безпеки.

На відміну від традиційного підходу до розробки розширень на чистому JavaScript, який часто призводить до написання спагеті-коду та ускладнює підтримку проєкту на пізніх етапах, використання Plasmo дозволяє кардинально оптимізувати процес створення продукту за рахунок автоматизації збірки під різні стандарти маніфестів, такі як Manifest V3 (MV3) для браузерів на базі Chromium (Google Chrome, Edge) та Manifest V2 (MV2) для стабільної підтримки у Firefox.

Крім того, даний фреймворк забезпечує повноцінну підтримку суворої типізації за допомогою TypeScript та інтеграцію сучасних модульних систем, що дозволяє виявляти потенційні помилки ще на етапі компіляції та значно спрощує масштабування кодової бази.

Додатковою перевагою інтеграції React у структуру браузерного розширення через Plasmo є можливість використання компонентного підходу, де кожен елемент інтерфейсу — від кнопки запуску сумаризації до складних віджетів бічної панелі — є ізольованим модулем із власною логікою та життєвим циклом. Це дозволяє розробникам повторно використовувати створені компоненти як у веб-версії сайту на Next.js, так і всередині самого розширення, забезпечуючи ідентичність візуального стилю та поведінки елементів без необхідності дублювання коду.

Завдяки вбудованим механізмам гарячого перезавантаження (Hot Module Replacement, HMR), які надає Plasmo, процес розробки та відладки інтерфейсу розширення відбувається в реальному часі без необхідності постійного ручного перезапуску плагіна в браузері, що суттєво підвищує загальну продуктивність праці та якість фінального програмного забезпечення.

4.1.1 Структурна організація коду проєкту

З метою забезпечення масштабованості та зручності підтримки, проєкт має модульну ієрархічну структуру. Увесь вихідний код логічно розділений на дві великі підсистеми: екосистему розширення та екосистему вебзастосунку.

- Основні директорії браузерного розширення:
- background/: фонові сервіс-воркери, які відповідають за комунікацію (messaging) між вкладками та обробку довготривалих подій (наприклад, відкриття сторінки історії).
- contents/ та contents_parts/: ключова директорія, де реалізовано логіку ін'єкції у вебсторінки. Містить компоненти плаваючого меню (FloatingMenu), бічної панелі (SidePanel) та сервіси захоплення DOM (HighlightService, ContentExtractor).
- pages/ та pages_parts/: повноцінні React-сторінки розширення (Dashboard, History, Settings), що відкриваються в окремих вкладках або спливаючих вікнах.
- lib/ та store/: утиліти для роботи з API, безпечним сховищем (Secure Storage) та глобальними станами (Zustand) та модулі для взаємодії зі сторінкою.

Папкову структуру розширення показано на рисунку 4.1.

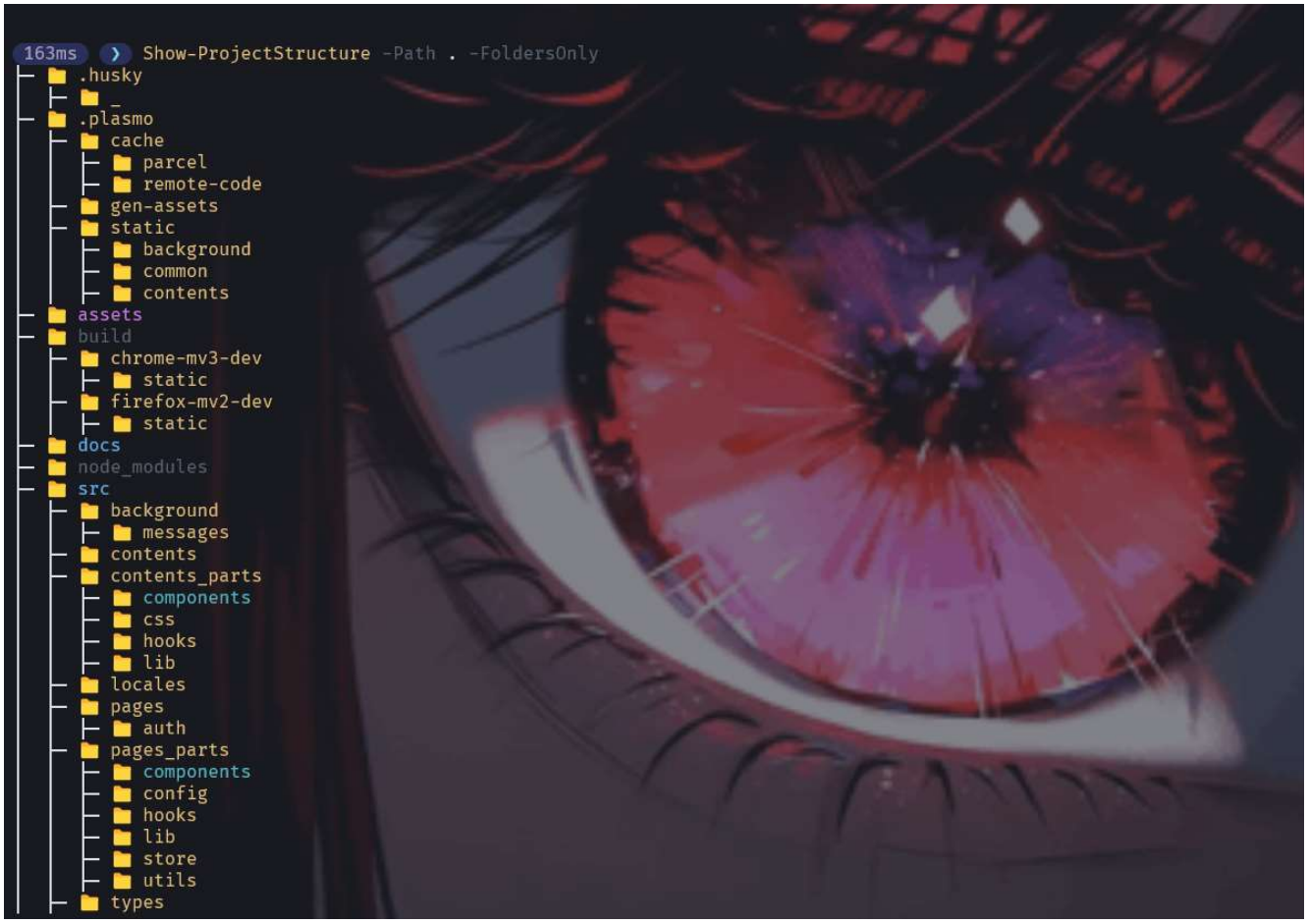


Рис. 3.4 Папкова структура розширення відображена в Windows Terminal

Основні директорії вебзастосунку (Next.js):

- app/: маршрутизатор застосунку (App Router), що містить сторінки авторизації, історії, налаштувань профілю та лендінгу.
- components/: перевикористовувані UI-компоненти (створені на базі Radix UI та Tailwind CSS), розділені за функціоналом (checkout, history, home).

Папкову структуру сайту показано на рисунку 4.2.

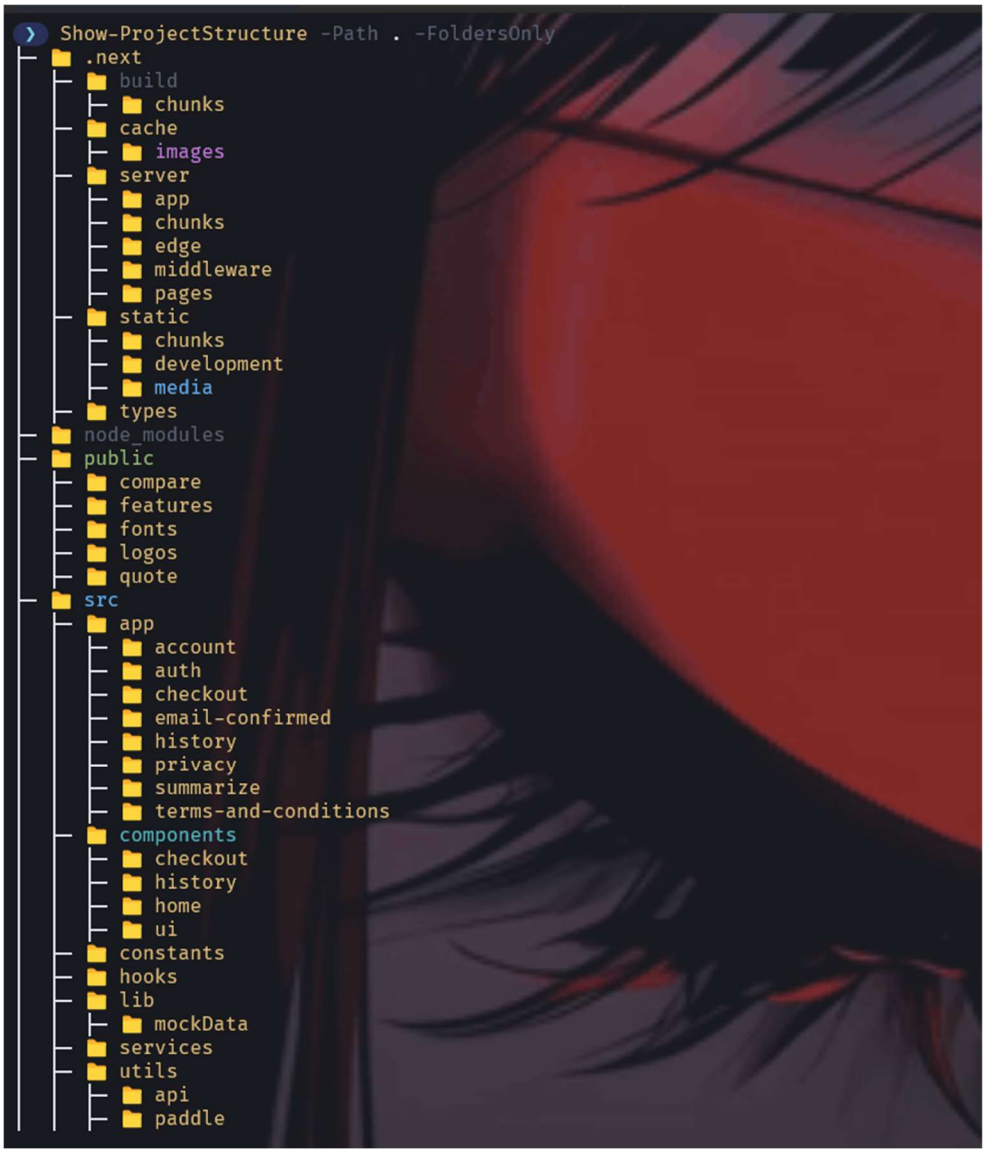


Рис. 4.2 Папкова структура сайту відображена в Windows Terminal

4.1.2 Реалізація алгоритмів парсингу та маркування DOM

Однією з найскладніших задач клієнтської частини є коректне захоплення тексту з довільних вебсторінок та його візуальне маркування без порушення верстки сайтів.

В архітектурі «QuickSum» цю проблему вирішено через модуль contentExtractor. Алгоритм попередньо створює ізольовану копію (клон) DOM-дерева, з якого видаляються всі технічні теги (script, style, noscript) та елементи навігації чи реклами (nav, footer, aside, .ads). Далі система застосовує алгоритм скорингу, оцінюючи текстові вузли за їхньою довжиною та семантичною

важливістю (наприклад, тегам article або p надається вища вага). Це гарантує, що на сумаризацію буде відправлено лише релевантний контент.

Для підсвічування тексту використовується HighlightService, який зберігає просторові координати виділення користувача (XPath та зміщення). Архітектурне проєктування цього сервісу обумовлене специфікою роботи з динамічним вебконтентом, де структура сторінки може зазнавати змін, а прямі посилання на DOM-вузли втрачаються між сесіями. Збереження точних координат у форматі XPath (шляху до конкретного вузла в ієрархії документа) разом із символьним зміщенням (offset) від початку текстового блоку дозволяє абстрагуватися від тимчасового стану пам'яті браузера та створити надійний зліпок виділеного користувачем фрагмента для його подальшого відтворення.

Проте у реальних умовах функціонування браузерних розширень розробники часто стикаються з проблемою асинхронної зміни DOM-дерева або модифікації скриптів сторонніх сайтів. Якщо точне відновлення оригінального об'єкта Range (стандартного браузерного інтерфейсу для представлення фрагментів документа) стає неможливим через структурні зсуви, перезавантаження сторінки користувачем або динамічний рендеринг контенту, алгоритм HighlightService не припиняє свою роботу, а задіює спеціально розроблений багаторівневий механізм фолбеку (резервного сценарію).

Цей фолбек-алгоритм ініціює процедуру інтелектуального пошуку текстових вузлів, спираючись на збережені раніше XPath-координати та метадані зміщення. Замість спроби сліпого відновлення об'єкта виділення, сервіс покроково аналізує ієрархічну структуру дерева елементів, перевіряє відповідність текстового наповнення у знайдених вузлах і виконує гнучку підстройку під поточну верстку сайту, мінімізуючи ризик виникнення помилок при невідповідності індексів.

Наступним етапом роботи резервного механізму є фаза безпосередньої візуалізації маркування, яка відбувається повністю в ізольованому режимі для запобігання руйнування цілісності та стилістики оригінального коду сторінки. Після того, як цільові текстові вузли успішно ідентифіковані та верифіковані в DOM-структурі, алгоритм динамічно обгортає їх у спеціальні кастомні елементи .

Цим тегам автоматично присвоюються унікальні ідентифікатори та утилітарні класи CSS (наприклад, класи Tailwind), що відповідають за колірне кодування та графічні ефекти виділення.

Завдяки такій логіці обробки, процес інтерактивного підсвічування інтегрується в документ безшовно та без виникнення витоків пам'яті чи конфліктів із нативними скриптами цільового сайту. Впровадження подібної відмовостійкої архітектури всередині HighlightService гарантує стабільне збереження та коректне відображення всіх користувацьких маркувань навіть за умов тривалого моніторингу вебсторінок зі складною, асинхронно оновлюваною структурою даних.

Приклад, як виглядає збережене підсвічування на сторінці на рисунку 4.3.

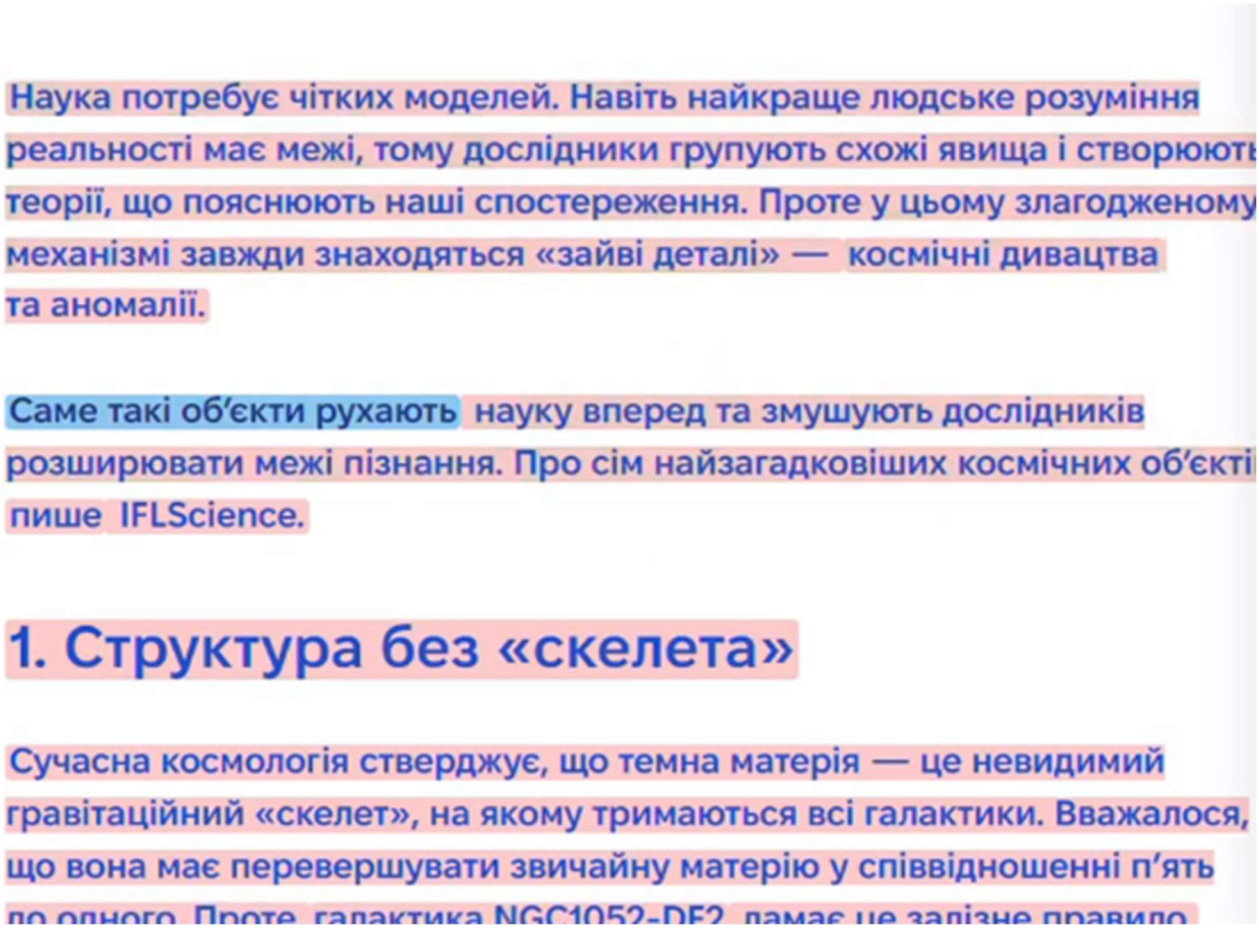


Рис. 4.3 Приклад збережених виділень на сторінці

4.2 Тестування та оцінка продуктивності програмного забезпечення

Заключним та одним із найважливіших етапів розробки фронтенд-архітектури є комплексне тестування створеного програмного забезпечення. Головною метою цього етапу є детальна перевірка стабільності роботи контентних скриптів під час їх інтеграції у DOM-структуру різноманітних сторонніх вебсайтів, верифікація коректності функціонування евристичних алгоритмів очищення тексту, а також превентивне виявлення вузьких місць у загальній продуктивності браузера.

Для забезпечення високого рівня надійності та автоматизації перевірок у кодову базу було інтегровано сучасний фреймворк Vitest, який вирізняється високою швидкістю виконання тестів завдяки вбудованій підтримці ESM-модулів та паралелізації обчислень. Тестування реактивних компонентів користувацького інтерфейсу та кастомних хуків здійснюється у поєднанні з бібліотекою React Testing Library, що дозволяє емітувати реальну поведінку клієнта без прив'язки до деталей внутрішньої реалізації компонентів.

Графічне представлення архітектури та результатів виконання тестів у середовищі Vitest наочно продемонстроване на рисунку 4.4.

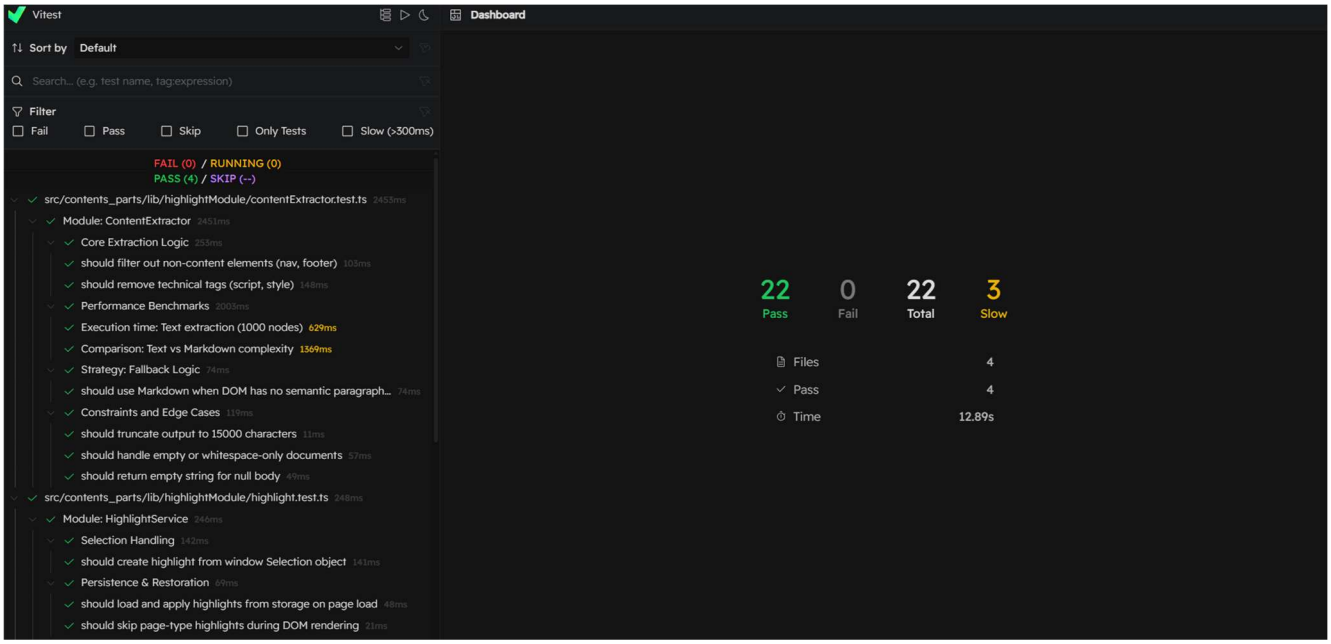


Рис. 4.4 Графічне представлення Vitest

4.2.1 Модульне тестування логіки екстракції та маркування

Модульне тестування (Unit Testing) спрямоване на ізольовану перевірку алгоритмів роботи з DOM-деревом без рендерингу графічного інтерфейсу. Під час розробки було покрито тестами ключові сервіси: HighlightService та ContentExtractor.

У процесі тестування HighlightService було реалізовано мокування (mocking) API-запитів та браузерного сховища. Успішно пройдено наступні сценарії:

1. Створення виділень (Selection Handling): Перевірка алгоритму перехоплення стандартного об'єкта window.getSelection(), його трансформації у внутрішній формат системи та виклику методу збереження у сховище.
2. Персистентність та відновлення: Доведено, що при завантаженні сторінки сервіс коректно витягує збережені хайлайти з локальної бази та ін'єктує їх у DOM у вигляді елементів з класом .plasmo-highlight. Також перевірено логіку фільтрації: система успішно ігнорує сумаризації типу "повна сторінка" (page-type), не намагаючись відмалювати їх як текстові мітки.
3. Cross-node маркування: Перевірено складний крайовий випадок (edge case), коли виділений користувачем текст розділений вкладеними HTML-тегами (наприклад, Hello nested world). Сервіс успішно знаходить та підсвічує такі фрагменти.
4. Під час тестування модуля ContentExtractor було доведено коректність роботи фільтрів: система гарантовано видаляє "шумові" селектори (навігацію, футери) та технічні теги, залишаючи лише чистий текст статті. Також перевірено механізм обмеження довжини рядка (truncation), який гарантує, що генерований текст не перевищує максимально дозволений ліміт символів.

Фрагмент коду модульних тестів “екстрактора тексту сторінки” показано на рисунку 4.5.

```
  });

  describe("Constraints and Edge Cases", () => {
    const MAX_LIMIT = maxContentLength;

    it(`should truncate output to ${MAX_LIMIT} characters`, () => {
      const longText = "A".repeat(MAX_LIMIT + 5000);
      document.body.innerHTML = `<main><p>${longText}</p></main>`;

      const result = extractMainContent();

      expect(result.length).toBe(MAX_LIMIT);
      expect(result.length).toBeLessThanOrEqual(MAX_LIMIT);
    });

    it("should handle empty or whitespace-only documents", () => {
      document.body.innerHTML = `<main><p>   </p><span> </span></main>`;

      const result = extractMainContent();

      expect(result).toBe("");
    });

    it("should return empty string for null body", () => {
      document.body.innerHTML = "";
      expect(extractMainContent()).toBe("");
    });
  });
});
```

Рис. 4.5 Фрагмент коду модульних тестів екстрактора

4.2.2 Інтеграційне тестування React-компонентів

Інтеграційне тестування розробленого програмного комплексу проводилося з метою детальної перевірки коректності взаємодії користувача з UI-елементами системи, валідації логіки обробки вхідних даних та верифікації поведінки контентних скриптів розширення, які динамічно вбудовуються безпосередньо у DOM-структуру сторонніх вебсторінок.

Головним інструментом для виконання цих завдань став пакет @testing-library/react, який дозволяє взаємодіяти з відрендереними компонентами через екологічні селектори, максимально наближені до дій реального користувача. Важливою архітектурною особливістю підготовки тестового середовища у фреймворку Vitest стало застосування механізму глибокого макетування (vi.mock). Це дозволило повністю ізолювати клієнтську логіку від зовнішніх факторів: було створено заглушки для медіаресурсів, заблоковано реальні виклики до сервісу автентифікації auth-service із примусовим поверненням успішної імітаційної сесії, а також підмінено глобальні глобальні менеджери станів storage-store та local-

storage-store, що дозволило гнучко керувати прапорцями активації режиму розробника (isDevModeON) безпосередньо всередині тест-кейсів.

Перший масштабний блок інтеграційного тестування був зосереджений на верифікації компонента форми авторизації Login.tsx. У межах цього сценарію перевірявся комплекс поведінкових реакцій інтерфейсу на дії користувача, зокрема валідація полів введення в режимі втрати фокусу (onBlur). Емуляція за допомогою методу fireEvent.change некоректного імейлу та занадто короткого пароля підтвердила, що інтерфейс миттєво реагує на порушення правил безпеки та асинхронно відмальовує повідомлення про помилки. Окремо було протестовано реактивну поведінку форми під час повторного введення символів, де очищення повідомлень про помилки (clearMessage) контролювалося через метод waitFor. Крім того, тести успішно підтвердили логіку роботи у режимі розробника, перевіряючи наявність специфічних інформаційних сповіщень та автоматичне підтягування системних облікових даних. Фінальний сценарій цього блоку повністю підтвердив інтеграційний конвеєр успішного входу: при введенні валідних даних метод fireEvent.click ініціює виклик функції loginWithEmail із точними параметрами та успішно викликає зворотний колбек onLoginSuccess, повністю блокуючи появу помилок валідації на екрані.

Другий етап тестування був присвячений аналізу життєвого циклу головного контентного компонента Content.tsx, який відповідає за впровадження функціоналу сумаризації в активні вкладки браузера. Для симуляції реального вебсерфінгу перед кожним тест-кейсом у блоці beforeEach здійснювалося примусове конфігурування об'єкта window.location та ручне наповнення глобального DOM-дерева тестовою HTML-розміткою (зокрема, табличними структурами даних). Сценарій рендерингу за замовчуванням підтвердив, що при завантаженні стандартної сторінки розширення коректно ініціалізує та відображає кнопку активації бічної панелі, орієнтуючись на локалізовані назви елементів. Емуляція кліку на цю кнопку підтвердила виклик інтегрованого сервісу штучного інтелекту FakeSummarizer.summarize, що свідчить про правильне з'єднання інтерфейсного шару з модулями обробки природної мови. Також за допомогою мокання

кастомного хука useSelection було перевірено логіку виділення тексту: при появі контенту система миттєво відмальовує плаваюче контекстне меню (FloatingMenu) із кнопками вибору режимів стиснення інформації.

Критично важливою частиною інтеграційної перевірки контентного скрипта став аудит безпеки та сумісності через механізм чорного списку сайтів (Blacklist Logic). Цей тест симулював перехід користувача на заборонений або чутливий вебдомен (наприклад, chatgpt.com), для чого об'єкт window.location.hostname динамічно перевизначався в процесі виконання. Тестування повністю підтвердило захисну логіку компонента: за допомогою методу waitFor було зафіксовано, що контентний скрипт миттєво припиняє свій життєвий цикл, повертає значення null і очищує перший дочірній вузол контейнера (container.firstChild), повністю виключаючи будь-яке втручання в DOM-структуру цільового ресурсу. Водночас паралельний тест для дозволених доменів (наприклад, wikipedia.org) продемонстрував успішний рендеринг та стабільну ініціалізацію всіх дочірніх компонентів. Фрагмент коду інтеграційних тестів показано на рисунку 4.6.

Рис. 4.6 Фрагмент коду інтеграційних тестів

4.2.3 Оцінка продуктивності (Performance Benchmarks)

Оскільки розроблене браузерне розширення функціонує безпосередньо в однопотоковому середовищі виконання клієнтського браузера, нефункціональні вимоги до програмного комплексу диктують жорсткі обмеження на час виконання синхронних операцій з DOM-деревом. Будь-яка тривала блокуюча операція здатна викликати повне зависання (freeze) активної вкладки, що критично погіршує користувацький досвід (UX). Для експериментального підтвердження та доведення високої обчислювальної ефективності створених рішень було розроблено спеціалізований бенчмарк-тест у файлі contentExtractor.test.ts. Методика тестування передбачала як ізольовану перевірку базових селекторів, так і стрес-тестування за допомогою штучної генерації великих масивів даних в оперативній пам'яті з фіксацією часу виконання через високоточний інтерфейс performance.now().

Початковий блок тестування (Core Extraction Logic) повністю підтвердив коректність роботи алгоритму відсікання структурного шуму. Шляхом маніпуляцій з об'єктом document.body.innerHTML у середовище впроваджувалися фрагменти типової верстки, що містили елементи глобальної навігації (<nav>), підвали сторінок (<footer>) та технічні теги (<script>, <style>). Тест зафіксував, що функція extractMainContent безпомилково ізолює цільові інформаційні абзаци, повністю видаляючи сторонній код та UI-елементи. Окремий етап перевірок був присвячений аналізу крайових випадків (Constraints and Edge Cases). Було верифіковано поведінку системи при обробці порожніх документів або сторінок, заповнених виключно спецсимволами та нерозривними пробілами ( ), де алгоритм коректно повертав порожній рядок. Також протестовано механізм захисту від переповнення контекстного вікна нейромережі: при передачі наддовгого тексту система стабільно обрізає вихідний масив до граничного значення, що строго відповідає константі maxContentLength.

Центральною частиною дослідження став безпосередній навантажувальний бенчмаркінг продуктивності (Performance Benchmarks). На першому етапі було реалізовано автоматизований цикл, який динамічно створював у DOM-структурі

навантажувальний контейнер із 1000 текстових абзаців. Запуск функції extractMainContentInText над цією структурою продемонстрував середній час виконання, що стабільно не перевищує 500 мілісекунд (в середньому фіксувалися показники у межах 200-300мс). Це повністю задовольняє жорсткі критерії оперативної синхронної обробки даних усередині клієнтського потоку. На другому етапі було проведено порівняльний аналіз швидкості між розробленим методом лінеаризації у звичайний текст та альтернативним алгоритмом extractContentInMarkdown, який виконує складнішу структурну конвертацію у формат Markdown для збереження посилань, таблиць та заголовків. Експеримент підтвердив очікувану закономірність: обробка у форматі Markdown є обчислювально складнішою та виконується повільніше.

Також вирішальним фактором на користь розробленого текстуального методу став аналіз економічної ефективності та щільності корисних даних. Під час проведення ручних тестів на реальній сторінці Вікіпедії обсягом понад 180 тисяч символів було виявлено, що конвертація в Markdown зберігає величезну кількість синтаксичного шуму (розмітки таблиць, посилань, списків лінків). Це призвело до того, що підсумковий обсяг токенів, які надсилалися до штучного інтелекту, зріс у 5 разів порівняно з очищеним plain text. Таке перевитрачання контекстного вікна робить Markdown-екстракцію у п'ять разів дорожчою за собівартістю запитів до LLM API, що ще раз доводить високу практичну та архітектурну ефективність розробленої функції extractMainContentInText.

Для досягнення максимальної гнучкості системи в алгоритм було впроваджено інтелектуальну стратегію відкату (Fallback Logic). Тестування цього модуля показало, що у випадках, коли на сторінці повністю відсутні стандартні семантичні текстові абзаци (наприклад, на сторінках-каталогах, де вся інформація представлена виключно у вигляді таблиць <table>, <tr>, <td>), алгоритм автоматично перемикається на альтернативну стратегію збору даних. Тест підтвердив, що при такій структурі функція extractMainContent успішно видобуває та акумулює контент із таблиць (наприклад, назви товарів та ціни), гарантуючи стабільність сумаризації для вебдокументів будь-якого типу верстки.

Повні результати навантажувального тестування та бенчмарк швидкості алгоритму скрапінгу тексту сторінки графічно відображено на рисунку 4.7.

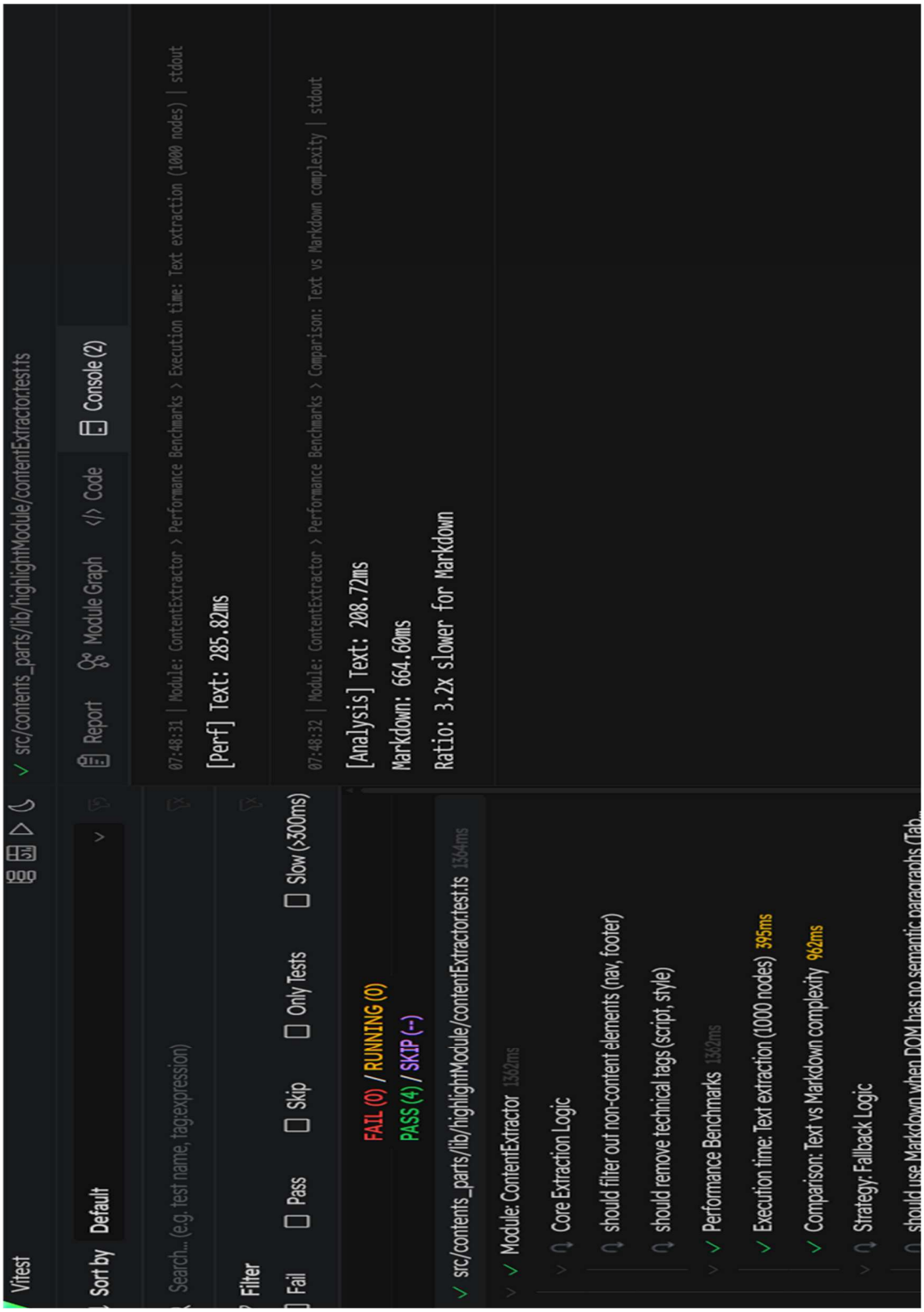


Рис. 4.7 Бенчмарк швидкості алгоритму скрапінгу тексту сторінки

ВИСНОВКИ

Досліджено сучасні підходи до побудови інтерфейсів браузерних розширень та алгоритмів інтерактивної взаємодії з вебконтентом. Виявлено типові проблеми існуючих SaaS-рішень для сумаризації, серед яких: Більшість існуючих сервісів функціонують лише як вебсайти, що змушує користувача вручну копіювати текст та постійно перемикатися між вкладками браузера. У багатьох аналогах відсутня система збереження історії запитів, через що результати попередніх обробка стають недоступними після закриття сторінки. Значна частина платформ надає лише один фіксований алгоритм стиснення, не дозволяючи користувачеві обирати між детальним переказом та короткими тезами. Популярні браузерні розширення часто орієнтовані на аналіз усієї сторінки цілком і не підтримують сумаризацію окремих виділених фрагментів тексту.

Визначено технічні бар’єри реалізації складних UI-рішень у середовищі браузера. Сформульовано функціональні та нефункціональні вимоги до клієнтської частини (Frontend) застосунку, які охоплюють миттєву реакцію на виділення тексту (low latency), коректність роботи на різних типах DOM-структур, стабільність ін'єкцій у Shadow DOM та відповідність сучасним стандартам безпеки браузерних маніфестів (Manifest V3).

Обґрунтовано вибір технологій: фреймворку Plasmo для швидкої розробки та збірки розширення, бібліотеки React для побудови реактивного інтерфейсу та мови TypeScript для забезпечення наскрізної типізації всієї клієнтської логіки. Вибраний технологічний стек дозволяє створити ізольоване, продуктивне та легко підтримуване рішення, яке мінімально впливає на продуктивність основної вкладки браузера.

Спроєктовано й реалізовано архітектуру клієнтських модулів, що включає систему автоматичного захоплення контенту та інтерактивного маркування. Забезпечено логіку вилучення релевантної інформації через модуль

contentExtractor, механізми збереження та відновлення текстових виділень за допомогою highlightService, а також реалізовано гнучкі UI-компоненти плаваючого меню та бічної панелі. Забезпечено захист від некоректної роботи на конфліктних доменах завдяки впровадженню інтелектуальної системи фільтрації («чорний список»).

Проведено тестування фронтенд-частини застосунку (модульне та інтеграційне), під час якого перевірено відповідність реалізованого функціоналу поставленим вимогам. Здійснено оцінку продуктивності екстракції даних: доведено, що розроблені алгоритми забезпечують обробку великих масивів тексту (1000+ вузлів) з часом затримки менше 500 мілісекунд, що гарантує плавність роботи інтерфейсу та відсутність блокувань основного потоку браузера.

Оцінено високий потенціал клієнтського рішення для інтеграції у повноцінний SaaS-продукт. Запропоновано напрями подальшого вдосконалення, зокрема впровадження складніших алгоритмів семантичного аналізу сторінок (Readability), розширення можливостей кастомізації візуальних тем інтерфейсу та додавання підтримки роботи з PDF-файлами безпосередньо у вікні браузера.

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

1. Гасило Д.В., Залива В.В. Захист інформації у frontend-частині клієнтського застосунку для обробки та узагальнення текстової інформації з web-ресурсів. Матеріали Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». 23 квітня 2026 року, Київ, ДУІКТ. Збірник тез. К.: ДУІКТ, 2026. С.248-249.

2. Гасило Д.В., Залива В.В. Визначення вимог до frontend частин у клієнтському застосунку для обробки та узагальнення текстової інформації з web-ресурсів. Матеріали Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». 23 квітня 2026 року, Київ, ДУІКТ. Збірник тез. К.: ДУІКТ, 2026. С.65-67.

ПЕРЕЛІК ПОСИЛАНЬ

1. Co-Intelligence: Living with AI Mollick E. Co-Intelligence: Living with AI. Boston : Portfolio, 2024. P. 6-9. (дата звернення: 14.05.2026).
2. Summarizer.org. URL: https://summarizer.org/ (дата звернення: 01.03.2026).
3. Smart Summarize. URL: https://smartsummarize.io/ (дата звернення: 01.03.2026).
4. Summarize AI Chrome Extension. URL: https://chromewebstore.google.com/ (дата звернення: 02.03.2026).
5. AISMRY. URL: https://aismry.com/ (дата звернення: 02.03.2026).
6. Visual Studio Code Documentation. URL: https://code.visualstudio.com/docs (дата звернення: 11.03.2026).
7. HTTPie Documentation. URL: https://httpie.io/docs (дата звернення: 12.04.2026).
8. Docker Desktop Documentation. URL: https://docs.docker.com/desktop/ (дата звернення: 12.03.2026).
9. GitHub Documentation. URL: https://docs.github.com/ (дата звернення: 12.03.2026).
10. TypeScript Documentation. URL: https://www.typescriptlang.org/docs/ (дата звернення: 12.03.2026).
11. Zod Documentation. URL: https://zod.dev/ (дата звернення: 13.03.2026).
12. Bun Documentation. URL: https://bun.sh/docs (дата звернення: 13.03.2026).
13. Node.js Documentation. URL: https://nodejs.org/docs/latest/api/ (дата звернення: 13.03.2026).
14. Next.js Documentation. URL: https://nextjs.org/docs (дата звернення: 13.03.2026).
15. Turbopack Documentation. URL: https://nextjs.org/docs/architecture/turbopack (дата звернення: 13.03.2026).

16. Webpack Documentation. URL: https://webpack.js.org/concepts/ (дата звернення: 13.03.2026).
17. Plasmo Documentation. URL: https://docs.plasmo.com/ (дата звернення: 13.03.2026).
18. Tailwind CSS Documentation. URL: https://tailwindcss.com/docs (дата звернення: 13.03.2026).
19. Motion (Framer Motion) Documentation. URL: https://motion.dev/docs (дата звернення: 13.03.2026).
20. Better Fetch Documentation. URL: https://better-fetch.vercel.app/ (дата звернення: 13.03.2026).
21. Zustand Documentation. URL: https://zustand.docs.pmnd.rs/ (дата звернення: 13.03.2026).
22. Vitest Documentation. URL: https://vitest.dev/guide/ (дата звернення: 07.04.2026).
23. Husky Documentation. URL: https://typicode.github.io/husky/ (дата звернення: 11.03.2026).
24. ESLint Documentation. URL: https://eslint.org/docs/latest/ (дата звернення: 11.03.2026).
25. Пірог О.В. Безпека вебдодатків: навч. посіб. Житомир: Житомирська політехніка, 2025. С. 225-237. (дата звернення: 14.03.2026).
26. IBM. What are Large Language Models (LLMs). URL: https://www.ibm.com/think/topics/large-language-models (дата звернення: 13.05.2026).

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 Frontend-частини застосунку

Виконав студент 4 курсу групи ПД-42
Гасило Денис Віталійович
Керівник роботи
доцент кафедри, доктор філософії (PhD)
Залива Віталій Вікторович

Київ – 2026

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи спрощення процесу опрацювання великих обсягів контенту користувачами на основі 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 програмні засоби та технології розробки клієнтської частини застосунку для автоматизованого захоплення та інтелектуального узагальнення текстової інформації з web-ресурсів із використанням моделей штучного інтелекту.

2

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Аналіз предметної галузі систем автоматизованої обробки та узагальнення текстової інформації з метою визначення основних принципів роботи, існуючих підходів, проблем та вимог до сучасних систем сумаризації тексту.
2. Огляд та вибір засобів програмної реалізації фронтенд частини застосунку з метою підбору оптимального стеку технологій для створення швидкого, масштабованого програмного забезпечення
3. Проєктування архітектури фронтенд частини застосунку з метою побудови структурованої, модульної та підтримуваної системи взаємодії компонентів застосунку.
4. Програмна реалізація, тестування фронтенд частини застосунку з метою створення працездатного користувацького інтерфейсу, перевірки його коректної роботи та забезпечення стабільності функціонування застосунку.

3

АНАЛІЗ АНАЛОГІВ

Характеристика	Summarizer.org	Smartsummarize.io	Summarize AI	AISMRY	**QuickSum (Запропоноване рішення)**
Наявність браузерного розширення	Немає	Немає	€	€	**€**
Формат узагальнення тексту	Довгий, Короткий	Довгий, Середній, Короткий	Лише один	Лише один	**Середній, Короткий**
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)**
Робота з історією запитів	Ні	Так	Ні	Ні	**Так**

4

ВИМОГИ ДО ДОДАТКУ

Функціональні

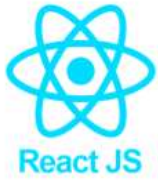
1. Реєстрація, авторизація та автентифікація користувачів із можливістю редагування профілю.
2. Перегляд історії сумаризацій через розширення або вебсайт із можливістю видалення та повторної сумаризації.
3. Захоплення виділеного тексту й парсинг основного контенту сторінки з попередньою обробкою тексту.
4. Надсилання запиту на сумаризацію з відображенням завантаження та результату поверх сторінки.
5. Взаємодія з сервером для обробки запитів, отримання відповідей і відображення помилок.
6. Підтримка тестового режиму (Dev Mode) для роботи без підключення до серверної частини.

Нефункціональні

1. Оперативність парсингу та видобування тексту зі сторінки повинна займати не більше 500 мс для забезпечення миттєвого відгуку інтерфейсу.
2. Система повинна мати архітектуру з ефективною обробкою помилок та мережевих збоїв, гарантуючи збереження локального стану (хайлайтів) навіть при втраті з'єднання.
3. Забезпечення безпеки клієнтської частини через валідацію та ізоляцію контекстів виконання розширення.
4. Frontend-архітектура повинна бути модульною
5. Розширення має підтримувати роботу у сучасних браузерах (Chrome, Firefox) та браузерах на їх основі (Brave, Zen) з обов'язковою підтримкою стандарту ES2019.

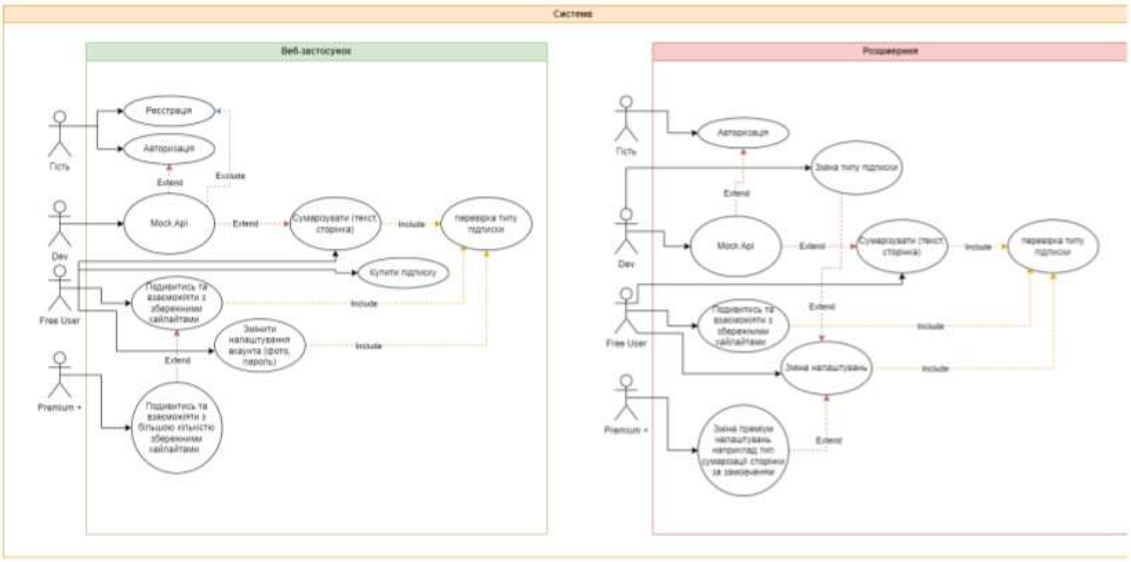
5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ



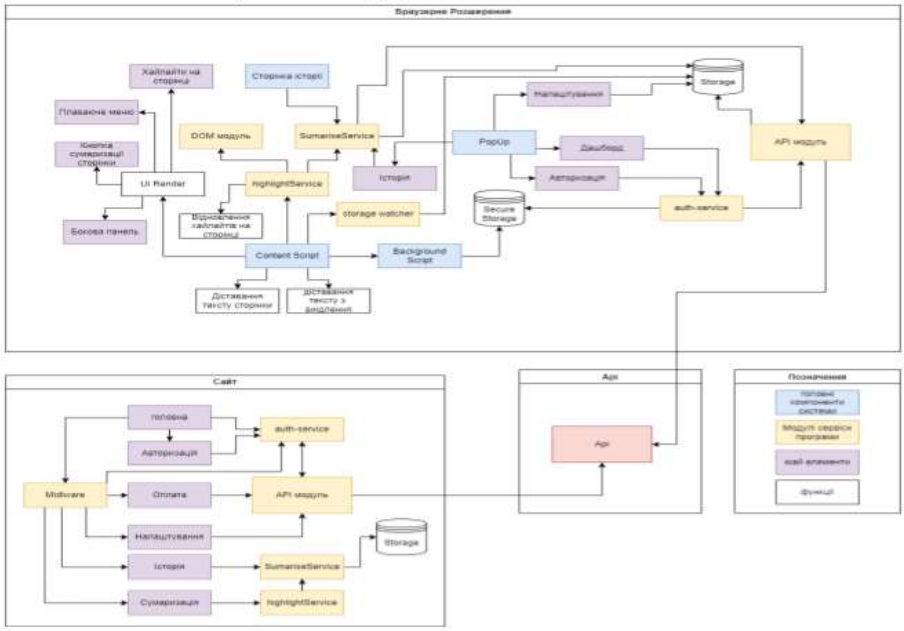
6

Діаграма варіантів використання



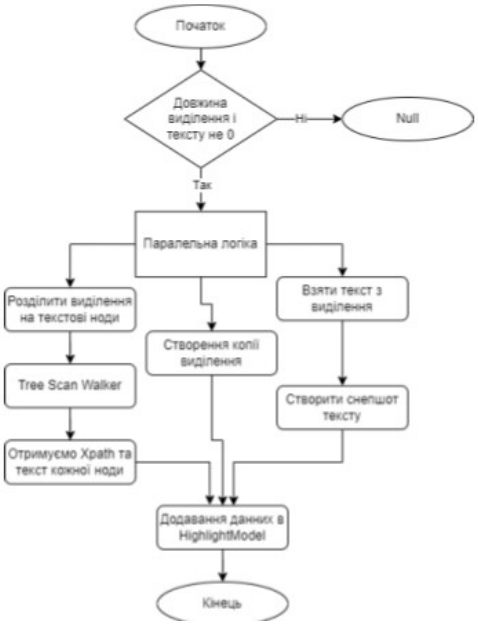
7

Архітектура QuickSum



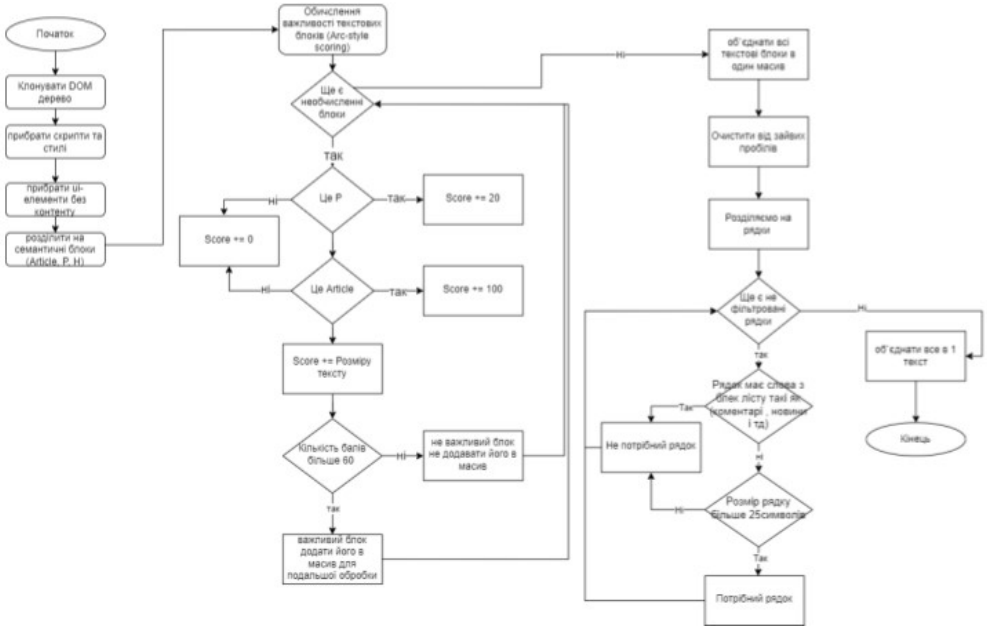
8

Блок схема алгоритма отримання хайлайту з виділення



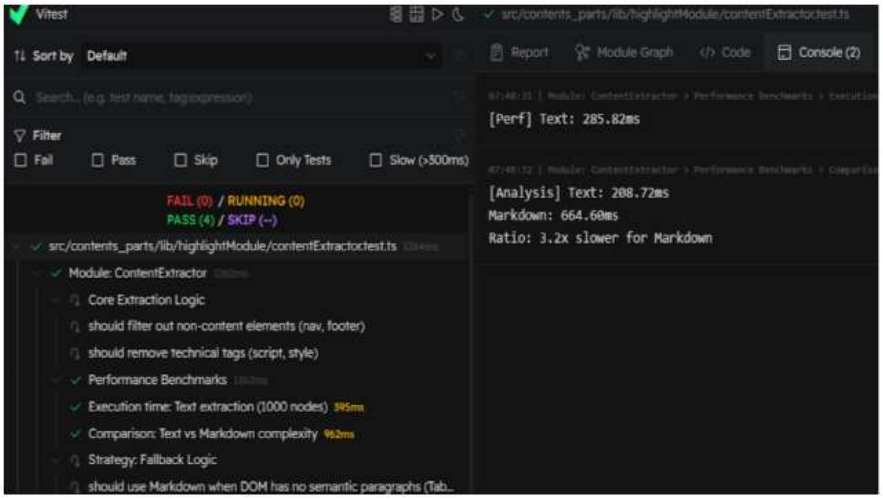
9

Блок схема алгоритма отримання тексту зі сторінки

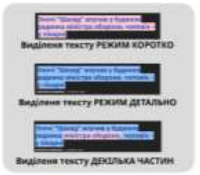


10

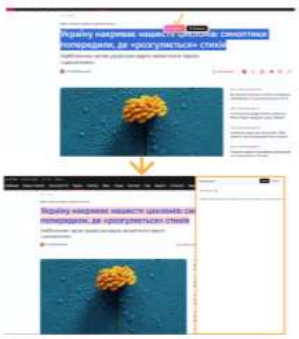
Тестування
бенчмарк швидкості скрапінгу тексту зі сторінки

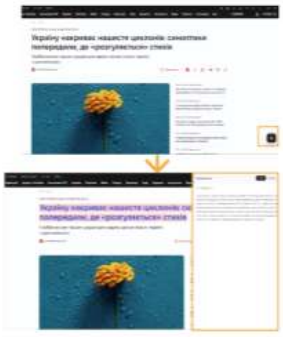


ЕКРАННІ ФОРМИ

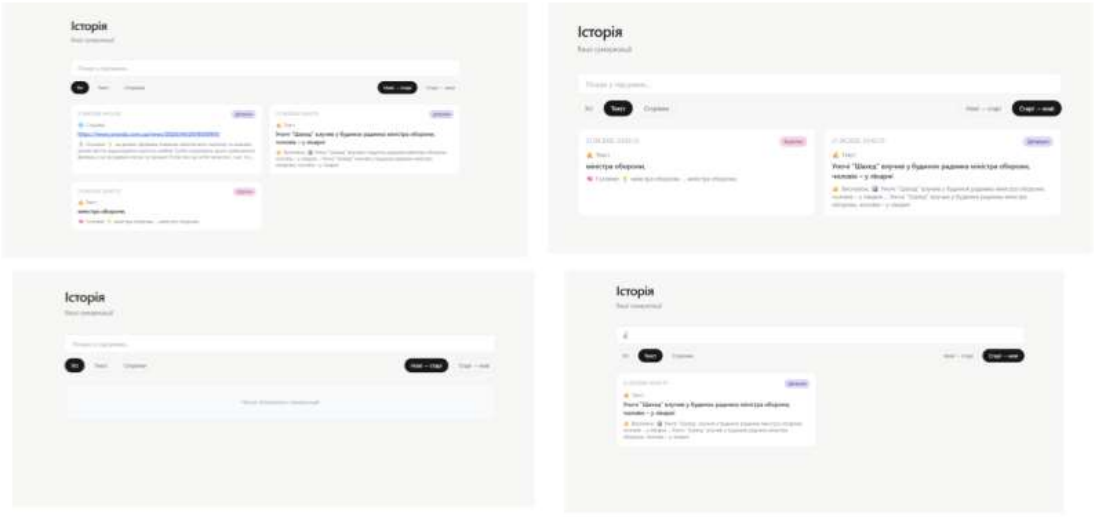



Довільна сторінка


Виділення тексту і натискання режиму коротко


Натискання кнопки сумарізування всієї сторінки

ЕКРАННІ ФОРМИ

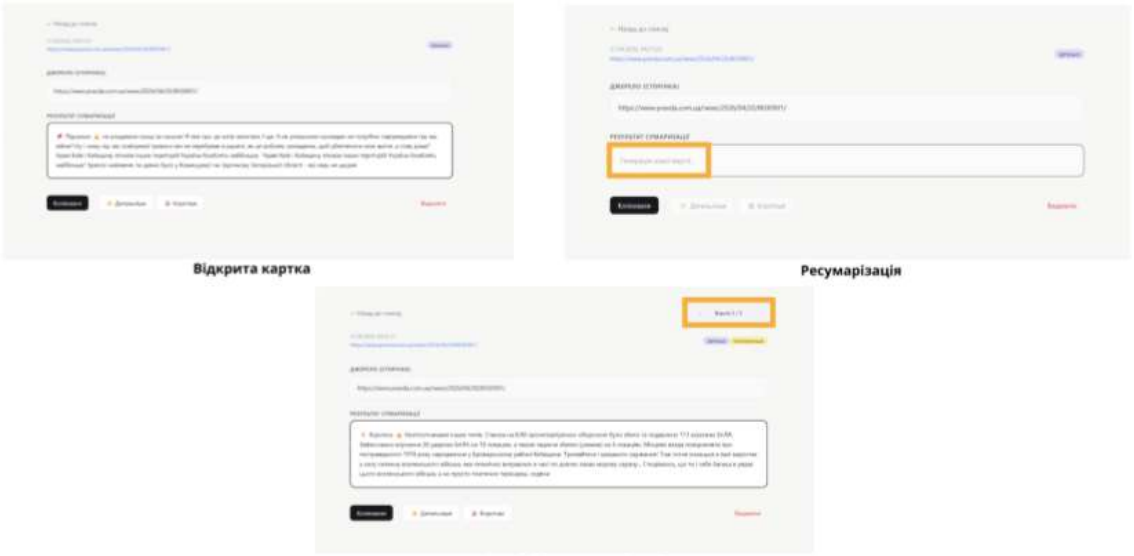


Історія якщо є данні | якщо данних немає

Сортування | Пошук

13

ЕКРАННІ ФОРМИ



Відкрита картка

Ресумарізація

Версії після ресумарізації

14

ЕКРАННІ ФОРМИ

Вкладка логін в режимі розробника

Вкладка логін

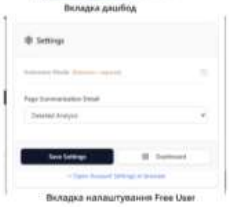
Вкладка дашбод Free User

Вкладка дашбод Premium User

Вкладка дашбод

Вкладка налаштування Free User

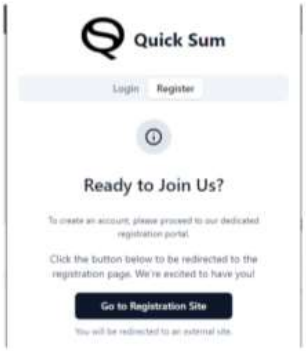
Вкладка налаштування Premium User



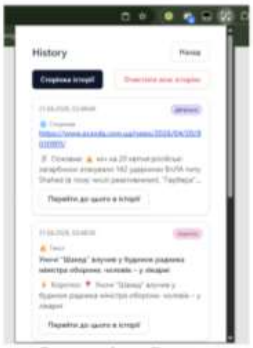
Вкладка налаштування Free User

15

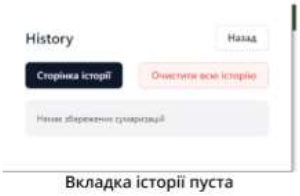
ЕКРАННІ ФОРМИ



Вкладка регістрації



Вкладка історії повна

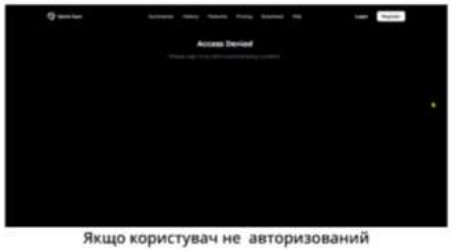


Вкладка історії пуста

16

ЕКРАННІ ФОРМИ

Якщо користувач авторизований до сумарізації

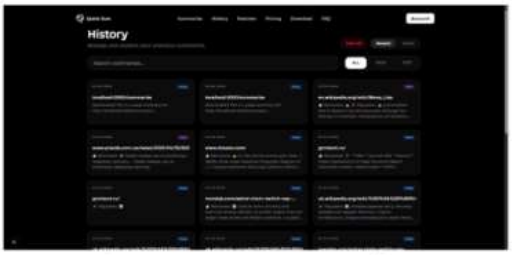


Якщо користувач не авторизований

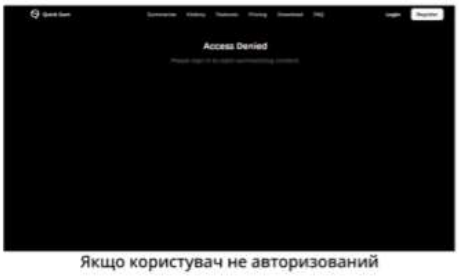
Якщо користувач авторизований після сумарізації

17

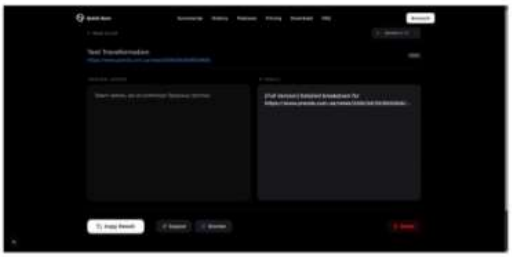
ЕКРАННІ ФОРМИ



Якщо користувач авторизований



Якщо користувач не авторизований

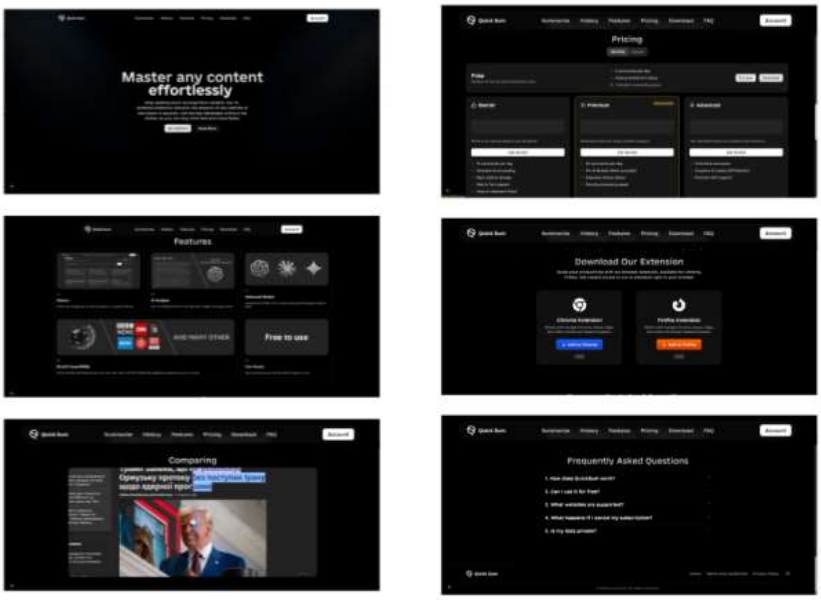


Користувач авторизований | сумаризація тексту

Користувач авторизований | сумаризація сайту

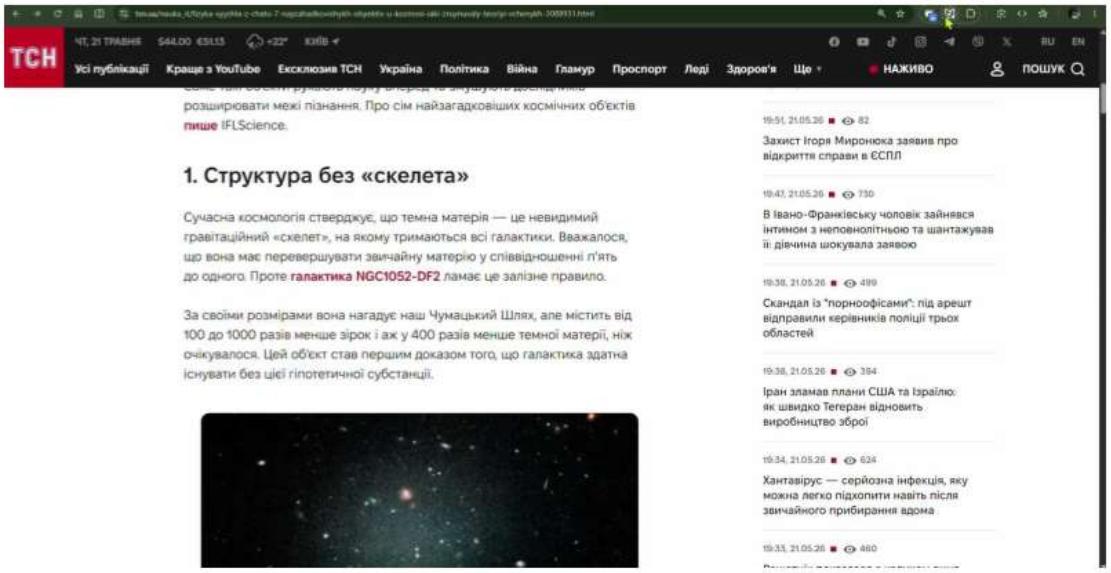
18

ЕКРАННІ ФОРМИ



19

ВІДЕО РОБОТИ ЗАСТОСУНКУ



20

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

Тези доповідей:

1. Гасило Д.В., Залива В.В. Захист інформації у frontend-частині клієнтського застосунку для обробки та узагальнення текстової інформації з web-ресурсів. Матеріали Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». 23 квітня 2026 року, Київ, ДУІКТ. Збірник тез. К.: ДУІКТ, 2026. С.248-249.

2. Гасило Д.В., Залива В.В. Визначення вимог до frontend частини у клієнтському застосунку для обробки та узагальнення текстової інформації з web-ресурсів. Матеріали Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». 23 квітня 2026 року, Київ, ДУІКТ. Збірник тез. К.: ДУІКТ, 2026. С.65-67.

21

ВИСНОВКИ

1. Досліджено сучасні підходи до побудови браузерних розширень та систем інтерактивної взаємодії з веб контентом. Виявлено основні недоліки існуючих сервісів сумаризації: відсутність інтеграції у браузер, історії запитів та підтримки роботи з окремими фрагментами тексту.

2. Сформульовано функціональні та нефункціональні вимоги до frontend-частини застосунку на основі аналізу конкурентів, зокрема низьку затримку обробки тексту, підтримку складних DOM-структур і відповідність стандарту Manifest V3.

3. Обґрунтовано вибір технологічного стеку: Plasmo, NextJs, React та TypeScript, що забезпечують продуктивність, ізольованість і підтримуваність.

4. Спроєктовано та реалізовано архітектуру клієнтських модулів, включаючи механізми екстракції контенту, збереження текстових виділень та інтерактивні UI-компоненти.

5. Проведено тестування frontend-застосунку, яке підтвердило коректність роботи та високу продуктивність до 500мс обробки великих обсягів тексту при 1000 нодів.

6. Визначено перспективи подальшого розвитку системи такі, як інтеграція алгоритмів семантичного аналізу, підтримка PDF та розширення можливостей кастомізації інтерфейсу.

22

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

Контент скрипт розширення (без імпортів)

Файл: src\contents\main.tsx

```
export default function Main() {
  const { getItem } = useLocalStorageStore();
  const { text, pos, setText } = useSelection();
  const {
    showPanel,
    setShowPanel,
    panelContent,
    setPanelContent,
    selectedHighlightId,
    setSelectedHighlightId
  } = usePanel();

  const { init, isDevModeON, isDevModeAccess, isUserAccess
} =    useStorageStore();

  const  [extensionOn,  setExtensionOn]  =  useState<boolean  |
null>(null);
  const [summaryMode, setSummaryMode] = useState<"short" |
"full">("short");

  const isAuthed = isUserAccess || isDevModeAccess;

  // Динамічний аудит поточного URL за реєстром винятків
  const isBlacklisted = blackList.some(
    (domain) =>
      window.location.hostname === domain ||
      window.location.hostname.endsWith("." + domain)
  );

  useEffect(() => {
    init().catch(console.error);
  }, [isDevModeON, isDevModeAccess, isUserAccess, init]);

  // --- ФУНКЦІЯ ВЕРИФІКАЦІЇ ЧЕРЕЗ БЕКГРАУНД ---
  const verifySettingWithSecureStorage = async (
    key: string,
    localValue: string
  ) => {
    try {
      const response = await sendToBackground({
        name: "verifySettings",
        body: { key, localValue }
      });

      if (response && !response.isValid && response.actualValue
!== null) {
        console.warn(
          `[Verification] Невідповідність для ${key}. Коригуємо
стейт.`
        );

        if (key === "IsExtensionOn") {
          if (response.actualValue === undefined) {
            console.warn(
              `[Verification] Ключ ${key} відсутній у secure.
Автовимкнення.`
            );
            setExtensionOn(false);
          } else {
            setExtensionOn(response.actualValue === "true");
          }
        }

        if (key === "PageSummaryDetailLevel") {
          if (
            response.actualValue === "short" ||
            response.actualValue === "full"
          ) {
            setSummaryMode(response.actualValue);
          }
        }
      }
    } catch (error) {
      console.error(`[Verification] Помилка верифікації ключа
${key}:`, error);
    }
  };

  // 1. Первинне завантаження налаштувань з LocalStorage
при монтуванні
```

```
 useEffect(() => {
  const fetchSettings = async () => {
   if (isBlacklisted) return;

   const status = await getItem("IsExtensionOn");
   const mode = await getItem("PageSummaryDetailLevel");

   const currentStatus = status === "true";
   setExtensionOn(currentStatus);

   if (mode === "short" || mode === "full") {
    setSummaryMode(mode);
   }

   // Фоновий запуск крос-валідації
   verifySettingWithSecureStorage("IsExtensionOn",
String(status));

verifySettingWithSecureStorage("PageSummaryDetailLevel",
String(mode));
  };

  fetchSettings();
 }, [getItem, isBlacklisted]);

 // 2. ВОТЧЕР: Реактивне відстеження та валідація змін у
реальному часі
 useEffect(() => {
  if (isBlacklisted) return;

  const unsubscribePromise = storage.watch({
   IsExtensionOn: ({ newValue }) => {
    const statusStr = String(newValue);
    setExtensionOn(newValue === "true" || newValue ===
true);
    verifySettingWithSecureStorage("IsExtensionOn",
statusStr);
   },
   PageSummaryDetailLevel: ({ newValue }) => {
    if (newValue === "short" || newValue === "full") {
     setSummaryMode(newValue);

verifySettingWithSecureStorage("PageSummaryDetailLevel",
newValue);
    }
   }
  });

  return () => {
   // unsubscribePromise
   //  .then((unsubscribe) => {
   //   if (typeof unsubscribe === "function") unsubscribe();
   //  })
   //  .catch(console.error);
  };
 }, [isBlacklisted]);

 // 3. Керування рендерингом хайлайтів у DOM-структурі
 useEffect(() => {
  if (extensionOn && isAuthed && !isBlacklisted) {
   highlightService.loadAndApplyForCurrentPage();
  } else {
   clearHighlightSpans();
  }
 }, [extensionOn, isAuthed, isBlacklisted]);

 // Сценарій 1: Обробка виділеного фрагмента тексту
 const handleAction = async (mode: "short" | "full") => {
  if (!text) return;
  const id = crypto.randomUUID();

  const pendingHighlight = await
highlightService.createFromSelection(
   id,
   mode,
   "..."
  );
  if (!pendingHighlight) return;

  setSelectedHighlightId(id);
  setPanelContent("Генеруємо підсумок...");
  setShowPanel(true);
  setText("");

  try {
   const summaryResult = await summarize(text, mode);
   await updateHighlight(id, { summary: summaryResult,
status: "done" });
   setPanelContent(summaryResult);
  } catch (error) {
   await updateHighlight(id, { status: "error", summary:
"Error" });
   setPanelContent("Помилка.");
  }
```

```
  };

  // Сценарій 2: Комплексна сумаризація всієї вебсторінки
  const handleSummarizePage = async () => {
    const pageText = extractMainContent();
    const id = crypto.randomUUID();

    try {
      const result = await summarize(pageText, summaryMode);
      await highlightService.createPageHighlight(id, result,
pageText);

      setSelectedHighlightId(id);
      setPanelContent(result);
      setShowPanel(true);
    } catch (error) {
      console.error("Page summary failed:", error);
      setShowPanel(true);
      setPanelContent("Помилка сумаризації сторінки.");
    }
  };

  const handleDeleteHighlight = async () => {
    if (!selectedHighlightId) return;
    await removeHighlight(selectedHighlightId);
    setShowPanel(false);
    await highlightService.loadAndApplyForCurrentPage();
  };

  // Перевірка умов безпеки рендерингу інтерфейсу
  if (isBlacklisted || !extensionOn || !isAuthed) return null;

  return (
    <>
      <PageSummarizeButton
onSummarize={handleSummarizePage} />

      {text && !showPanel && (
        <div
          style={{
            position: "fixed",
            top: pos.y - 30,
            left: pos.x,
            zIndex: 2147483647
          }}>
          <FloatingMenu onAction={handleAction} />
        </div>
      )}

      {showPanel && (
        <SidePanel
          content={panelContent}
          onClose={() => setShowPanel(false)}
          onOpenHistory={(id) =>
            sendToBackground({
              name: "openHistory",
              body: {
                highlightId: id === null ? null : id ||
selectedHighlightId
              }
            })
          }
          onDelete={handleDeleteHighlight}
          selectedHighlightId={selectedHighlightId}
        />
      )}
    </>
  );
}
```

AuthService Розширення

Файл: src\pages_parts\lib\auth\auth-service.ts

```
import useSecureStorageStore from "~pages_parts/store/secure-storage-store";
import useStorageStore from "~pages_parts/store/storage-store";
import $fetch from "~pages_parts/utils/api";

import { getDevToken, validateDevCredentials } from "./dev-user";

export async function loginWithEmail(email: string, password: string) {
  const { getItem, setItem } = useStorageStore.getState();
  const secureStorage = useSecureStorageStore.getState();

  const devMode = (await getItem("isDevModeON")) === "true";

  if (devMode && validateDevCredentials(email, password)) {
    const token = getDevToken();

    await secureStorage.setItem("authToken", token);

    try {
      useStorageStore.getState().setDevModeAccess(true);
    } catch {
      await setItem("isDevModeAccess", "true");
      await setItem("UserPlan", "dev");
    }

    return true;
  }

  await $fetch("/auth/sign-in/email", {
    method: "POST",
    body: {
      email,
      password
    },
    credentials: "include",
    headers: {
      "X-Api-Key": process.env.PLASMO_PUBLIC_API_KEY!
    }
  });

  const cookie = await new Promise<chrome.cookies.Cookie | null>((resolve) => {
    chrome.cookies.get(
      {
        url: process.env.PLASMO_PUBLIC_SERVER_URL!,
        name: "session"
      },
      resolve
    );
  });

  if (cookie) {
    await secureStorage.setItem("authToken", cookie.value);
  }

  return true;
}

export async function getSession() {
  const { getItem, setItem } = useStorageStore.getState();
  const secureStorage = useSecureStorageStore.getState();
  const storage = useStorageStore.getState();

  const token = await secureStorage.getItem("authToken");

  if (!token) return null;

  if (token.startsWith("dev-token")) {
    const { DEV_USER } = await import("./dev-user");

    const storedUser = await secureStorage.getItem("user");

    if (!storedUser) {
      await secureStorage.setItem("user", JSON.stringify(DEV_USER));
    } else {
      const parsed =
        typeof storedUser === "string" ? JSON.parse(storedUser) : storedUser;
      return parsed;
```

```
    }

    return DEV_USER;
  }

  try {
    const session = await $fetch("/auth/session", {
      method: "GET",
      headers: {
        "X-Api-Key":
process.env.PLASMO_PUBLIC_API_KEY!,
        Cookie: `session=${token}`
      }
    });

    try {
      useStorageStore.getState().setUserAccess(true);
    } catch {
      await setItem("isUserAccess", "true");
    }

    await                                storage.setItem("user_plan",
session.subscription?.name || "none");
    await storage.setItem("isUserAcsess", "true");
    await                                secureStorage.setItem("user",
JSON.stringify(session));

    return session;
  } catch {
    await secureStorage.removeItem("authToken");
    await storage.removeItem("user_plan");
    await storage.removeItem("isUserAcsess");
    return null;
  }
}

export async function logout() {
  const secureStorage = useSecureStorageStore.getState();

  console.log("[auth-service: logout] Logging out user...");

  const token = await secureStorage.getItem("authToken");

  if (!token) {
    console.log(
      "[auth-service: logout] No auth token found, skipping logout
request."
    );
    return;
  }

  console.log(
    "[auth-service: logout] Found auth token, proceeding with
logout.",
    token
  );

  if (token.startsWith("dev-token")) {
    await secureStorage.removeItem("authToken");

    try {
      useStorageStore.getState().setDevModeAccess(false);
    } catch {
      await
useStorageStore.getState().removeItem("isDevModeAccess");
    }
  } else {
    try {
      await $fetch("/auth/sign-out", {
        method: "GET",
        headers: {
          Cookie: `session=${token}`,
          "X-Api-Key":
process.env.PLASMO_PUBLIC_API_KEY!
        }
      });
    } catch {
      console.error(
        "[auth-service: logout] Logout request failed, but
proceeding to clear session."
      );
    } finally {
      await secureStorage.removeItem("authToken");
      await
useStorageStore.getState().removeItem("isUserAccess");
    }
  }
}
```

Екстрактор тексту зі сторінки

Файл:

src\contents_parts\lib\highlightModule\pageExtractor\extractMainContentInText.ts

```
export function extractMainContentInText(): string {
 const clone = document.body.cloneNode(true) as HTMLElement;

 // 1. прибираємо шум DOM
 const noiseSelectors = [
  "script",
  "style",
  "noscript",
  "header",
  "footer",
  "nav",
  "aside",
  "iframe",
  ".ads",
  ".sidebar",
  ".menu",
  ".comments",
  ".social",
  ".share",
  ".banner",
  ".latest-news",
  ".related",
  ".tags",
  ".widget"
 ];

 noiseSelectors.forEach((sel) => {
  clone.querySelectorAll(sel).forEach((el) => el.remove());
 });

 // 2. беремо тільки контентні блоки
 const blocks = Array.from(
  clone.querySelectorAll("article, main, p, h1, h2, h3")
 ) as HTMLElement[];

 const scored = blocks
  .map((el) => {
   const text = (el.textContent || "").replace(/\s+/g, " ").trim();

   // Arc-style scoring
   const score =
    text.length +
    (el.tagName === "P" ? 20 : 0) +
    (el.tagName === "ARTICLE" ? 100 : 0);

   return { text, score };
  })
  .filter((b) => b.text.length > 40);

 // 3. беремо тільки "сильні" блоки
 const filtered = scored.filter((b) => b.score > 60).map((b) => b.text);

 // 4. склеюємо текст
 let result = filtered.join("\n\n");

 // 5. фінальна чистка whitespace (ВАЖЛИВО)
 result = result
  .replace(/\u00a0/g, " ")
  .replace(/[ \t]+/g, " ")
  .replace(/\n{3,}/g, "\n\n")
  .split("\n")
  .map((l) => l.trim())
  .filter((l) => {
   if (!l) return false;
   if (l.length < 25) return false;

   // прибираємо UI-сміття
   if (
    l.includes("Останні новини") ||
    l.includes("Powered by") ||
    l.includes("RSS") ||
    l.includes("комент") ||
    l.includes("Sign in")
   ) {
    return false;
   }

   return true;
  })
  .join("\n\n");

 return result.trim();}
```