НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ПОГОДЖЕНО	**ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ**
Декан факультету	**Завідувач кафедри комп'ютерних наук**
_____ **Ігор БОЛБОТ**	_____ **Белла ГОЛУБ**
(підпис)	(підпис)
«___» _____ 2026 р.	«___» _____ 2026 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Інтелектуальний сервіс генерації програмного коду на основі методів машинного навчання**»**
Спеціальність «122 Комп'ютерні науки»

Освітньо-професійна програма «Комп'ютерні науки»

Гарант освітньої програми д.е.н., професор	_____ (підпис)	**Роман РУДЕНСЬКИЙ**
Керівник бакалаврської кваліфікаційної роботи	_____ (підпис)	**Максим НЕДЬОШЕВ**
Консультант бакалаврської кваліфікаційної роботи д.е.н., професор	_____ (підпис)	**Роман РУДЕНСЬКИЙ**
Виконав	_____ (підпис)	**Тимур УХІН**

Київ-2026

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри комп'ютерних наук

к.т.н., доцент _____ **Белла ГОЛУБ**

(підпис)

«___» _____ 2026 р.

ЗАВДАННЯ
ДО ВИКОНАННЯ БАКАЛАВРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ
ЗДОБУВАЧУ
Ухіну Тимуру Артемовичу

Спеціальність «122 Комп'ютерні науки»

Освітньо-професійна програма «Комп'ютерні науки»

Тема бакалаврської кваліфікаційної роботи

«Інтелектуальний сервіс генерації програмного коду на основі методів машинного навчання»

затверджена наказом від «06» січня 2026 р. № 46 «С»
Термін подання завершеної роботи на кафедру «22» травня 2026 р.

Вихідні дані до бакалаврської кваліфікаційної роботи (проєкту): база знань UI-шаблонів, стильових профілів та каталог зображень

Перелік питань, що підлягають дослідженню: аналіз предметної області; проектування бази даних; реалізація RAG-пайплайну

_____

Перелік графічних документів (за необхідності).

_____

Дата видачі завдання «22» травня 2026 р.

Керівник бакалаврської кваліфікаційної роботи _____ (підпис) **Максим НЕДЬОШЕВ**

Консультант бакалаврської кваліфікаційної роботи _____ (підпис) **Роман РУДЕНСЬКИЙ**
д.е.н., професор

Завдання взяв до виконання _____ (підпис) **Тимур УХІН**

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

1. ANN (Approximate Nearest Neighbor) — наближений пошук найближчих сусідів
2. API (Application Programming Interface) — інтерфейс програмування застосунків
3. CPU (Central Processing Unit) — центральний процесор
4. CORS (Cross-Origin Resource Sharing) — механізм спільного використання ресурсів між різними джерелами
5. CSS (Cascading Style Sheets) — каскадні таблиці стилів
6. CRUD (Create, Read, Update, Delete) — базові операції з даними
7. GPU (Graphics Processing Unit) — графічний процесор
8. HTML (HyperText Markup Language) — мова розмітки гіпертексту
9. HNSW (Hierarchical Navigable Small World) — алгоритм ієрархічного навігаційного пошуку найближчих сусідів
10. HTTP (HyperText Transfer Protocol) — протокол передачі гіпертексту
11. IDOR (Insecure Direct Object Reference) — небезпечне пряме посилання на об'єкт
12. JS — JavaScript, мова програмування для вебзастосунків
13. JSON (JavaScript Object Notation) — формат обміну даними
14. LLM (Large Language Model) — велика мовна модель
15. LPU (Language Processing Unit) — спеціалізований процесор для обробки мови
16. ORM (Object-Relational Mapping) — об'єктно-реляційне відображення
17. PEFT (Parameter-Efficient Fine-Tuning) — параметрично-ефективне донавчання
18. RAG (Retrieval-Augmented Generation) — генерація з доповненням пошуком

19. REST (Representational State Transfer) — архітектурний стиль побудови API
20. SPA (Single Page Application) — односторінковий вебзастосунок
21. SQL (Structured Query Language) — мова структурованих запитів
22. СУБД — система управління базами даних
23. UML (Unified Modeling Language) — уніфікована мова моделювання
24. UUID (Universally Unique Identifier) — універсально унікальний ідентифікатор
25. WAL (Write-Ahead Log) — журнал транзакцій із попереднім записом

ВСТУП

Сучасна розробка програмного забезпечення неможлива без якісного клієнтського інтерфейсу. Відповідно, попит на швидке та ефективне створення веб інтерфейсів сильно зростає. Разом із тим проектування якісного інтерфейсу традиційно вимагає глибокого знання мови розмітки HTML, каскадних таблиць стилів CSS, принципів адаптивної верстки та специфіки роботи браузерів — компетенцій, якими далеко не завжди володіють замовники або предметні фахівці, котрі хочуть швидко отримати прототип свого продукту.

Поява великих мовних моделей суттєво змінила підхід до автоматизації рутинних завдань у сфері програмної інженерії. Такі системи, як GPT-4 або Claude, здатні генерувати функціональний HTML-код на основі текстового опису, однак результат часто має узагальнений, шаблонний характер і не враховує специфіки предметної галузі чи брендових вимог конкретного проекту. Це пов'язано з тим, що мовні моделі під час синтезу відповіді спираються виключно на внутрішні знання, сформовані на етапі навчання, без можливості оперативно звернутися до актуальної зовнішньої інформації.

Для подолання цього обмеження в останні роки набув широкого поширення підхід Retrieval-Augmented Generation — доповнена генерація з пошуком. Ідея методу полягає в тому, що перед формуванням відповіді модель отримує з зовнішньої бази знань набір релевантних фрагментів, які включаються до контексту запиту. Такий підхід дозволяє поєднати гнучкість генеративних моделей із точністю структурованих сховищ даних, що відкриває широкі можливості для побудови предметно-орієнтованих систем генерації контенту.

Актуальність цієї роботи визначається тим, що поєднання RAG-підходу з векторними базами даних для задачі генерування вебінтерфейсів є відносно новим напрямом, практична реалізація якого дозволяє скоротити час розробки прототипів, зробити процес проектування доступним для нетехнічних учасників команди та забезпечити структурну й стилістичну узгодженість результатів.

Метою роботи є проектування та реалізація інтелектуальної системи автоматичного генерування HTML-інтерфейсів користувача на основі підходу Retrieval-Augmented Generation із застосуванням векторної бази знань і великої мовної моделі.

Для досягнення поставленої мети визначено такі завдання:

1. Провести системний аналіз предметної області, сформулювати вимоги до розроблюваної системи та виконати порівняльний огляд існуючих рішень.
2. Дослідити принципи функціонування підходу RAG, механізми формування векторних представлень тексту та технології семантичного пошуку у векторних базах даних.
3. Розробити логічну модель даних системи, що враховує вимоги як реляційного, так і векторного зберігання.
4. Обрати та обґрунтувати систему управління базами даних, визначити схему зберігання даних.
5. Спроектувати та реалізувати серверну частину застосунку із REST API-інтерфейсом на основі фреймворку Flask.
6. Реалізувати модуль семантичного пошуку на основі sentence-transformers та модуль генерації HTML засобами API Groq.
7. Розробити клієнтський інтерфейс, що підтримує сеансову роботу та відображення результатів генерації.
8. Провести тестування системи та підготувати документацію щодо її розгортання й експлуатації.

У процесі виконання роботи використовувались методи системного аналізу, об'єктно-орієнтованого проектування, семантичного пошуку на основі векторної схожості, а також методи функціонального тестування. Технологічну основу розробки склали: мова програмування Python версії 3.11, мікрофреймворк Flask, система управління базами даних PostgreSQL із розширенням pgvector, бібліотека sentence-transformers для обчислення векторних представлень, API-сервіс Groq із моделлю llama-3.3-70b-versatile для генерації коду.

Практичне значення роботи полягає у створенні програмного продукту, що автоматизує розробку прототипів веб інтерфейсів. Розроблена система може бути використана в навчальних закладах, стартапах і маркетингових командах для швидкого прототипування і візуалізації ідей без залучення фронтенд-розробника.

Пояснювальна записка складається з вступу, чотирьох розділів, висновків та списку використаних джерел. Загальний обсяг записки становить 63 сторінок.

У першому розділі виконано системний аналіз предметної області: сформульовано постановку завдання, проведено огляд і порівняльний аналіз існуючих інструментів генерації вебінтерфейсів, а також детально розглянуто теоретичні основи технології RAG та механізми векторного пошуку.

Другий розділ присвячений інформаційному забезпеченню системи. В ньому розроблено логічну модель даних, обґрунтовано вибір PostgreSQL як системи управління базами даних, описано фізичну структуру таблиць і застосування розширення pgvector для зберігання та пошуку векторних представлень.

Третій розділ розкриває питання розробки прикладного програмного забезпечення. Визначено організаційну структуру застосунку, обґрунтовано вибір інструментів і бібліотек, детально описано реалізацію серверного API, модуля роботи з базою даних, модуля RAG-генерації та клієнтської частини.

Четвертий розділ містить рекомендації щодо впровадження та експлуатації системи: описано проведене тестування, сформульовано вимоги до апаратного та програмного середовища, визначено склад інсталяційного пакету.

1 СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Постановка завдання

Інформаційні технології сьогодні охоплюють практично всі галузі діяльності людини — бізнес, освіту, медицину, публічне управління та повсякденне життя. Головним інструментом взаємодії між людиною і програмним забезпеченням є вебінтерфейс, що поєднує графічне представлення і функціональні можливості застосунку в єдиному браузерному середовищі. Від рівня опрацьованості цього інтерфейсу залежить ефективність роботи користувача, його задоволеність і, як наслідок, комерційні показники продукту. Відомо, що первинна оцінка застосунку складається протягом перших секунд взаємодії і визначається передусім візуальним враженням. Таким чином, якість інтерфейсу є не другорядним естетичним питанням, а стратегічним чинником конкурентоспроможності програмного продукту.

Попит на якісні веб інтерфейси зростає значно швидше, ніж пропозиція кваліфікованих фронтенд-розробників. Сучасний ринок праці у сфері інформаційних технологій характеризується стабільним дефіцитом спеціалістів із фронтенд-розробки: за даними щорічних звітів провідних технологічних платформ, ця спеціальність стабільно входить до переліку найбільш затребуваних упродовж останніх років. Вартість послуг таких спеціалістів зростає, а час від відкриття вакансії до прийому на роботу збільшується. Це породжує ситуацію, коли навіть добре фінансовані команди не можуть забезпечити достатню швидкість розробки інтерфейсів для всіх своїх потреб.

Особливої гостроти проблема набуває в умовах гнучкої розробки, де конкурентна перевага визначається швидкістю перевірки гіпотез і довжиною ітераційного циклу. Нетехнічні учасники команди — менеджери продукту, спеціалісти з маркетингу, бренд-стратеги, засновники стартапів — як правило, мають сформоване бачення бажаного результату, проте позбавлені технічних

засобів для його самостійного втілення. Будь-який запит на прототип чи демонстраційний макет потрапляє до загальної черги розробки, очікує звільнення ресурсу і неминуче затримується. У середовищі, де місяці можуть вирішувати долю продукту, такі затримки є критичними.

Традиційний процес розробки веб інтерфейсів вимагає від виконавця глибокого знання цілого технологічного стека. Мова розмітки HTML (HyperText Markup Language) забезпечує структуру документа і описує семантику вмісту: заголовки, параграфи, списки, форми, посилання, медіа елементи. Каскадні таблиці стилів CSS (Cascading Style Sheets) визначають візуальне оформлення цієї структури: розміри, кольори, шрифти, відступи, розташування елементів, анімації. Принципи адаптивної верстки (responsive design) забезпечують коректне відображення сторінки на пристроях різних розмірів та орієнтацій — від компактних смартфонів до широкоформатних моніторів. Семантична розмітка впливає на доступність (accessibility, a11y) і пошукову оптимізацію (SEO). Питання продуктивності рендерингу, зокрема оптимізація критичного шляху, мінімізація Layout Shift і правильне завантаження ресурсів, також потребують спеціалізованих знань. Опанування всього цього до рівня, що дозволяє стабільно створювати якісні інтерфейси, займає роки практики і є практично недосяжним для більшості не технічних фахівців.

Поява великих мовних моделей на початку 2020-х років окреслила принципово нові горизонти автоматизації фронтенд-розробки. Сучасні LLM здатні формувати синтаксично і семантично коректний HTML із CSS на основі природномовного опису, що дозволяє отримувати функціональні прототипи без написання коду вручну. Це значно розширило уявлення про можливості автоматизованої генерації програмних артефактів.

Проте масштабне практичне застосування цих можливостей наштовхується на системне обмеження: мовні моделі, навчені на загальних відкритих даних, генерують узагальнені, типові рішення, що не враховують специфіку конкретного бренду, предметної галузі або дизайн-системи. Кожна організація має власну ідентичність — кольорову палітру, типографічні

стандарти, тон і характер комунікації з аудиторією, типові структури сторінок для свого домену, каталог власних фотографій і зображень. Ця ідентичність, як правило, повністю відсутня у результатах генерації без додаткового контексту, оскільки специфічні корпоративні знання просто не потрапили до загальнодоступних навчальних даних моделей.

Саме ця невирішена проблема створює потребу в реалізації інтелектуальної системи, яка буде поєднувати як можливості генерації великих мовних моделей так і залучення структурованої інформації щодо конкретних дизайн-систем, патернів інтерфейсної взаємодії та наявних графічних ресурсів. Реалізація такого підходу забезпечується методологією Retrieval-Augmented Generation, детальний аналіз якої наведено у підрозділі 1.3 даного розділу.

Об'єктом розробки в даній роботі є процес автоматизованого генерування HTML-сторінок за текстовим описом із врахуванням структурованого контексту з бази знань. Предметом дослідження є методи та інструменти, що забезпечують підвищення якості, структурної узгодженості та контекстної відповідності згенерованих інтерфейсів порівняно зі стандартними підходами до генерації коду засобами мовних моделей.

Мета розроблюваної системи полягає в наданні користувачеві можливості ввести текстовий опис бажаного веб інтерфейсу й отримати у відповідь повноцінну HTML-сторінку зі вбудованими CSS-стилями, що враховує відповідні шаблони структури, стилістичні характеристики обраної предметної галузі і придатні візуальні матеріали з бази знань системи.

Функціональні вимоги до системи охоплюють такі аспекти. Система повинна забезпечувати управління сеансами у форматі чатів, де кожен чат є окремою тематичною сесією роботи з генерацією інтерфейсів для конкретного проекту або задачі. У межах сеансу підтримується збереження всіх запитів і результатів з можливістю перегляду попередніх генерацій в будь-який момент. Функціонал генерації передбачає формування HTML-файлу, його збереження на сервері та надання URL для попереднього перегляду у браузері через вбудований iframe. Система підтримує семантичний підбір тематично відповідних зображень

із семантичного сховища й надає користувачеві можливість вибрати конкретне зображення до запуску генерації. Передбачена підтримка перейменування і видалення чатів з автоматичним видаленням всіх пов'язаних даних.

Нефункціональні вимоги включають такі положення. Персистентність усіх даних між сесіями роботи гарантує, що результати генерацій не втрачаються після закриття браузера. Ізоляція даних різних користувачів через унікальний ідентифікатор у форматі UUID, що передається в HTTP-заголовку, гарантує конфіденційність. Час відповіді на запит генерації не повинен перевищувати 30 секунд за нормальних умов навантаження. Наявність відкритого REST API з документованими ендпоінтами забезпечує можливість інтеграції з іншими системами.

З погляду класифікації інформаційних систем розроблювана система належить одночасно до кількох класів. За характером використання інформації вона є системою підтримки прийняття рішень у галузі дизайну інтерфейсів: пропонує готове рішення, проте остаточний вибір, оцінка і редагування результату залишаються за фахівцем. За масштабом це малогрупова вебсистема, що не потребує складної рольової моделі або корпоративного управління доступом. За рівнем автоматизації система функціонує у напівавтоматичному режимі. За технологічним підходом — до класу інтелектуальних інформаційних систем, що використовують методи машинного навчання та штучного інтелекту для обробки природної мови і семантичного пошуку.

1.2 Моделювання предметної області

Для формалізації вимог до системи і визначення сценаріїв взаємодії застосовано два типи UML-діаграм: діаграму варіантів використання і діаграму діяльності.

Діаграму варіантів використання системи генерації HTML-інтерфейсів зображено на рисунку 1.1.

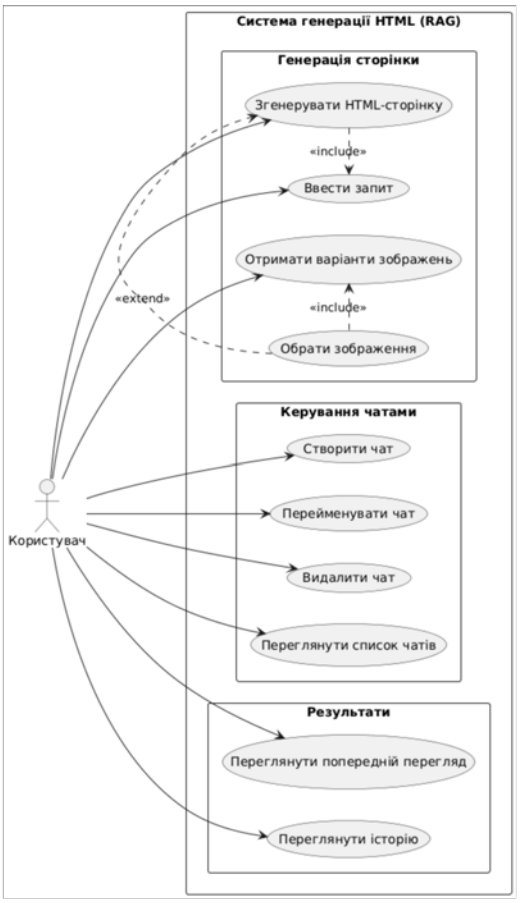

Рисунок 1.1 — Діаграма варіантів використання системи

З рисунку 1.1 видно, що система організована у три функціональні підсистеми з єдиним актором — Користувачем. Підсистема «Генерація сторінки» охоплює варіанти «Згенерувати HTML-сторінку» (include «Ввести запит»), «Отримати варіанти зображень» і «Обрати зображення» (extend). Підсистема «Керування чатами» включає «Створити чат», «Перейменувати чат», «Видалити чат» і «Переглянути список чатів». Підсистема «Результати» охоплює «Переглянути попередній перегляд» і «Переглянути історію».

Алгоритм роботи основного сценарію генерації зображено у вигляді діаграми діяльності на рисунку 1.2.

Рисунок 1.2 — Діаграма діяльності алгоритму генерації HTML-інтерфейсу

З рисунку 1.2 видно послідовність восьми кроків пайплайну. Перший крок — валідація вхідного промпту: порожній запит повертає помилку 400 без запуску ресурсомісткого пошуку. При валідному промпті виконується перетворення на ембедінг, пошук у PostgreSQL і отримання релевантних даних. Розгалуження визначає спосіб вибору зображення — явний вибір користувача або автоматичний підбір за семантикою. Після формування RAG-запиту виконується генерація HTML через LLM і збереження результату.

1.3 Огляд існуючих рішень та джерел інформації

Перед проектуванням нової системи необхідно проаналізувати наявні рішення у галузі автоматизованого створення вебінтерфейсів. Такий аналіз дозволяє визначити сильні і слабкі сторони існуючих підходів, виявити незаповнені ніші і обґрунтувати доцільність розробки нового інструменту. Для порівняльного аналізу обрано представників чотирьох класів рішень: конструктори вебсайтів, інструменти візуального програмування, мовні моделі загального призначення і спеціалізовані AI-генератори коду. Кожен клас розглядається з погляду функціональних можливостей, технічних обмежень і придатності для задачі предметно-орієнтованої генерації HTML-інтерфейсів.

1.3.1 Еволюція інструментів автоматизації вебінтерфейсів

Еволюція інструментів автоматизованого створення вебінтерфейсів пройшла кілька послідовних етапів, кожен із яких розширював доступність розробки для нетехнічних користувачів, однак жоден не вирішував фундаментальну проблему контекстуальної відповідності результатів.

На початку 2010-х років домінували конструктори вебсайтів типу Wix, Squarespace і Weebly. Ці платформи дозволяли нетехнічним користувачам збирати сайти із готових блоків через drag-and-drop інтерфейс без написання жодного рядка коду. Разом із тим вони пропонували вкрай обмежену гнучкість.

Приблизно з 2018 року набули популярності інструменти візуального програмування, орієнтовані на розробників, — Webflow і Framer. Вони поєднували гнучкість ручного кодування з візуальним редагуванням і виробляли семантично коректний HTML. Однак ці інструменти, незважаючи на візуальний інтерфейс, потребували значного занурення в деталі верстки і залишались більш орієнтованими на технічних фахівців, ніж на нетехнічних учасників команди.

З 2022–2023 років, після появи GPT-4 і моделей відповідного рівня, сформувався клас AI-орієнтованих інструментів для генерації коду і вебінтерфейсів. Саме цей клас є найрелевантнішим для порівняння з розроблюваною системою.

1.3.2 Мовні моделі загального призначення

До першої і найчисленнішої категорії сучасних інструментів генерації вебінтерфейсів належать універсальні чат-системи на основі великих мовних моделей. Представники цього класу — ChatGPT компанії OpenAI та модельна лінійка Claude від Anthropic — приймають текстові описи довільного рівня деталізації і формують HTML-розмітку з каскадними стилями. Завдяки навчанню на масштабних текстових корпусах такі системи демонструють впевнене розуміння загальних принципів вебдизайну і стабільно виробляють синтаксично грамотний код.

Разом із тим практичне застосування цих інструментів виявляє низку принципових обмежень.

Перше стосується відсутності предметного контексту. Мовні моделі оперують виключно параметричними знаннями, набутими в процесі навчання, і не мають доступу до специфічної інформації про конкретну організацію — її фірмову кольорову систему, корпоративну типографіку, характерні шаблони сторінок чи власний фотобанк. Наслідком є відтворення стандартних архітектурних рішень, однотипних незалежно від галузевого контексту запиту.

Друге обмеження пов'язане з варіативністю виводу. Через імовірнісну природу генерації ідентичні запити при повторних зверненнях дають помітно відмінні результати. Така непередбачуваність унеможливлює використання цих систем у робочих процесах, де ключовою вимогою є відтворюваність і консистентність результатів.

Третє обмеження полягає у відсутності механізмів управління згенерованими матеріалами. Чат-інтерфейс не передбачає структурованого збереження попередніх результатів, їхнього порівняння між собою або систематизації в рамках спільного проекту, що суттєво знижує практичну цінність інструменту для командної роботи.

1.3.3 Спеціалізовані AI-інструменти дизайну

Другий клас становлять спеціалізовані інструменти дизайну з інтегрованими AI-компонентами. Figma AI є частиною платформи Figma — найпоширенішого середовища векторного дизайну серед UX/UI-спеціалістів. Система надає функції автоматичного формування макетів, підбору компонентів із власних бібліотек і генерації варіантів оформлення на основі текстових підказок. Figma AI добре вбудована в типовий дизайн-процес і суттєво прискорює роботу спеціалістів. Проте цей інструмент орієнтований на виробництво проектних файлів у форматі .fig, а не готового HTML-коду, придатного до розгортання. Перехід від дизайн-макету до виробничого HTML потребує або ручного кодування, або підключення додаткових інструментів, що є окремим кроком, не автоматизованим у рамках самої платформи.

1.3.4 Спеціалізовані AI-генератори коду

Третій клас утворюють спеціалізовані AI-генератори коду з фокусом на компонентній розробці. Платформа v0 від Vercel генерує React-компоненти на основі текстових або візуальних підказок і надає можливість редагування результатів у браузері. Якість генерованих компонентів помітно вища, ніж у загальних мовних моделей, завдяки спеціалізованому навчанню. Проте v0 орієнтований виключно на React-екосистему і не підтримує генерацію чистого

HTML без фреймворку. Він також не підтримує інтеграцію з власними базами знань про дизайн.

Locofy спеціалізується на перетворенні макетів Figma на HTML і React-код, вирішуючи задачу автоматизації переходу від дизайну до коду. Однак цей інструмент потребує наявності вихідного дизайн-файлу Figma, що обмежує його застосовність для ранніх стадій прототипування без готового дизайну.

1.3.5 Порівняльний аналіз і обґрунтування нового підходу

Проведений аналіз дозволяє виявити системний недолік усіх розглянутих рішень: відсутність механізму семантичного пошуку в структурованій предметно-специфічній базі знань із подальшим включенням знайденого контексту в процес генерації. Ні загальні мовні моделі, ні спеціалізовані AI-інструменти не реалізують цього механізму.

Ще одним важливим аспектом є управління результатами. Жодна з розглянутих систем не пропонує вбудованого механізму управління сеансами і персистентного збереження всіх генерацій у структурованому форматі для подальшого аналізу і порівняння. Система, що проектується, передбачає збереження кожного запиту і результату у реляційній базі даних разом із збагаченим промптом, що забезпечить можливість відстежувати еволюцію результатів і аналізувати ефективність RAG-пайплайну.

Таблиця 1.1

Порівняльна характеристика існуючих рішень

Критерій	**ChatGPT/Claude**	**Figma AI**	**Locofy**
Генерація HTML/CSS	Так	Ні	Так
Семантичний пошук у базі знань	Ні	Ні	Ні
Врахування шаблонів дизайну	Ні	Частково	Ні
Підбір зображень за контекстом	Ні	Ні	Ні
Відкритий REST API	Так	Ні	Ні

Порівняльну характеристику розглянутих рішень за ключовими критеріями наведено у таблиці 1.1 очевидно, що розроблювана, яка проектується є єдиним рішенням, що одночасно забезпечує генерацію HTML, семантичний пошук у базі знань, підбір зображень і персистентне збереження результатів через відкритий API.

1.4 Технологія Retrieval-Augmented Generation

Технологія Retrieval-Augmented Generation є ключовою теоретичною основою розробленої системи. У цьому підрозділі розглянуто передумови її виникнення, принцип роботи, механізм формування збагаченого промпту і роль векторного пошуку як основи ретривера. Окрему увагу приділено порівнянню RAG із альтернативним підходом — донавчанням мовних моделей — та обґрунтуванню вибору RAG для задачі генерації вебінтерфейсів.

1.4.1 Передумови виникнення і концептуальна ідея

Великі мовні моделі навчаються на масивних текстових корпусах і зберігають знання у вигляді параметрів нейронної мережі. Після завершення навчання ці параметри фіксуються — модель не отримує нових знань без повторного навчання [1]. На практиці це породжує три проблеми, що обмежують застосовність LLM у реальних проектах. Перша — застарілість знань. Модель не має відомостей про те, що відбулося після дати зрізу навчальних даних. Друга — галюцинації. Якщо тема недостатньо представлена у навчальному корпусі, модель може впевнено видати правдоподібний, але хибний результат. Третя — відсутність специфічних знань. Внутрішні стандарти, власні дизайн-системи, корпоративні бази — нічого цього в загальнодоступних навчальних даних немає і бути не може [1].

Для вирішення цих проблем дослідники Meta AI запропонували підхід Retrieval-Augmented Generation — RAG [1]. Ідея проста: перед тим як генерувати відповідь, система спочатку знаходить у зовнішній базі знань усе релевантне і передає це до LLM разом із запитом. Модель отримує не просто «що хоче користувач», а повний контекст: що хоче плюс що про це відомо з бази. Результат виходить значно точнішим і передбачуванішим.

1.4.2 Опис принципу роботи Retrieval-Augmented Generation

Загалом процес можна розбити на чотири кроки [1].

Робота RAG-системи здійснюється у чотири послідовні етапи [1]:

1. Формування запиту. Користувач вводить текстовий опис бажаного результату — від короткого формулювання до розгорнутої специфікації.
2. Семантичний пошук. Система перетворює запит на векторне представлення і здійснює пошук у базі знань за схожістю змісту, а не збігом ключових слів. Детальніше про векторний пошук — у підрозділі 1.3.4.
3. Формування збагаченого промпту. Знайдені дані об'єднуються з оригінальним запитом і передаються до мовної моделі. Цей крок докладніше розглянуто у підрозділі 1.3.3.
4. Генерація результату. Модель поєднує власні параметричні знання з отриманим контекстом і формує відповідь, що враховує специфічну інформацію з бази знань [1].

1.4.3 Формування збагаченого промпту

Якість результату RAG-системи визначається не лише тим, що знайдено у базі знань, а й тим, як ці дані сформульовані у промпті для LLM. Це окрема дисципліна — context engineering, — яка займається тим, як структурувати контекст, що передається моделі [2].

У розробленій системі збагачений промпт складається з трьох частин.

1. Системний промпт. Він задає LLM роль і правила: що модель повинна зробити, у якому форматі повернути результат, яких обмежень дотримуватися. Без чіткого системного промпту навіть найкращий контекст не гарантує структурованого результату [2].

2. Контекстні дані з бази знань. Це конкретна інформація, витягнута семантичним пошуком: опис типової структури сторінки з переліком блоків і компонентів, стильові характеристики відповідної предметної галузі (кольорова палітра, типографіка, характер комунікації) і метадані підібраного зображення. Користувач також може явно обрати зображення до запуску генерації — тоді воно включається до промпту замість автоматично підібраного.
3. Запит користувача. Оригінальний текст у незміненому вигляді, що інтерпретується LLM з урахуванням двох попередніх частин.

Дослідження в галузі context engineering показують, що структурована подача контексту з чіткими розділами і заголовками дає значно кращі результати, ніж суцільний текст [2]. Саме тому у системі кожна з трьох частин промпту має явні текстові мітки.

1.4.4 Векторний пошук як основа ретривера

Семантичний пошук, що є серцем RAG-ретривера, реалізується через векторні представлення і пошук найближчих сусідів. Замість того, щоб порівнювати текст посимвольно, система перетворює і запит, і записи бази знань на числові вектори — ембедінги — і знаходить ті, що геометрично близькі у багатовимірному просторі [3].

Ця властивість дозволяє знаходити семантично споріднені документи навіть коли спільних слів між ними немає. Наприклад, запит «vet clinic login page» і опис шаблону «сторінка входу для медичного вебзастосунку з формою автентифікації» не мають жодного спільного слова — але їх ембедінги будуть поряд у векторному просторі [3].

Технічно пошук здійснюється через наближений пошук найближчих сусідів (ANN — Approximate Nearest Neighbor). Серед поширених алгоритмів

виділяють графові методи (HNSW — Hierarchical Navigable Small World), кластерні (IVFFLAT) і хеш-методи (LSH) [3].

Для роботи з векторами у рамках традиційної реляційної СУБД використовується розширення pgvector, яке додає до PostgreSQL тип VECTOR(n) і оператори пошуку за схожістю. Це дозволяє не вводити окрему векторну базу даних, а тримати реляційні і векторні дані в одній системі [3].

1.4.5 Порівняння RAG і донавчання

Альтернативою RAG є донавчання (fine-tuning) — повторне навчання попередньо навченої моделі на специфічному наборі даних. Модель коригує власні параметри для кращого відповідання конкретному домену [1].

Порівняльну характеристику двох підходів наведено у таблиці 1.2.

Існують два варіанти донавчання: повне, коли оновлюються всі параметри, і параметрично-ефективне (PEFT), коли оновлюється лише частина. Навіть PEFT потребує значних обчислювальних ресурсів — кількох потужних GPU і тривалого часу навчання, що є неприйнятним для більшості прикладних проєктів [1].

Щодо контролю якості: у RAG якість результатів безпосередньо залежить від якості записів у базі знань, яку адміністратор може перевіряти, редагувати і покращувати в будь-який момент. При донавчанні знання «розчинені» у вагах мережі і не піддаються прямій перевірці чи виправленню без повторного навчання [1].

Щодо складності впровадження: RAG вимагає налаштування ретривера, підготовки бази знань і інтеграції з LLM API, що є задачею середньої складності. Донавчання потребує підготовки навчального датасету, запуску процесу навчання на спеціалізованому обладнанні, валідації результатів і подальшого розгортання нової версії моделі — що є значно складнішим процесом [1].

Таблиця 1.2

Порівняння підходів RAG і fine-tuning

Критерій	**RAG**	**Fine-tuning**
Актуальність знань	Динамічна — БД оновлюється без перенавчання	Статична — потрібне нове навчання
Обчислювальні витрати	Низькі — тільки пошук і inference	Високі — GPU, час, пам'ять
Прозорість	Висока — джерело кожного результату відоме	Низька — знання «розчинені» у вагах
Вартість оновлення	Мінімальна — додати записи до БД	Значна — повторне навчання
Контроль якості	Є — через наповнення і перевірку БД	Обмежений — залежить від навчальних даних
Складність впровадження	Середня	Висока

На відміну від RAG, донавчання формує статичну модель зі знаннями, зафіксованими на момент навчання. При зміні або оновленні предметної бази знань потрібне нове повне або часткове перенавчання, тоді як у RAG достатньо оновити відповідні записи у базі даних [1].

Проте на відміну від RAG, яке забезпечує доступ до актуальних динамічних даних, дообучення створює статичну модель, що потребує повторного навчання при появі нової інформації [1].

1.4.6 Обґрунтування вибору RAG для розробленої системи

Для задачі генерації веб-інтерфейсів на основі конкретних дизайн-систем RAG є оптимальним вибором з таких причин:

- Динамічність бази знань. Дизайн-системи регулярно оновлюються: додаються нові шаблони, змінюються стилістичні вимоги. RAG дозволяє оновлювати базу знань без перенавчання моделі [1].
- Обчислювальна доступність. Донавчання моделі розміром 70 млрд параметрів потребує спеціалізованого GPU-обладнання і є неприйнятним для академічного проекту [1].
- Прозорість результатів. RAG дозволяє точно встановити, які записи бази знань вплинули на конкретний результат генерації — що є важливим для верифікації і покращення системи [2].
- Семантична гнучкість пошуку. Векторні ембедінги дозволяють знаходити релевантні шаблони навіть тоді, коли запит сформульовано інакше, ніж записи у базі знань [3].

Висновки до розділу 1

У першому розділі проведено системний аналіз предметної галузі і сформовано теоретичну базу для розробки системи.

Показано, що зростаючий попит на якісні вебінтерфейси за умов дефіциту фронтенд-розробників створює реальний запит на інструменти автоматизованої генерації, доступні нетехнічним учасникам команди. Огляд існуючих рішень — від no-code конструкторів до AI-генераторів коду — виявив спільний системний недолік: жоден із них не поєднує семантичний пошук у структурованій базі знань із генерацією HTML і збереженням результатів. Порівняльна таблиця 1.1 наочно підтверджує, що розроблювана система займає власну нішу серед існуючих рішень.

Розглянуто підхід RAG як технологічну основу системи. Встановлено, що він вирішує три ключові проблеми великих мовних моделей — застарілість знань, галюцинації і відсутність специфічного контексту. Описано чотири етапи роботи RAG-пайплайну і детально розглянуто формування збагаченого промпту з трьох частин: системного промпту, контекстних даних із бази знань і запиту користувача. Показано, що структурована подача контексту з чіткими мітками дає кращий результат, ніж суцільний текст [2].

Розглянуто механізм семантичного векторного пошуку як основу ретривера: перетворення тексту на ембедінги дозволяє знаходити релевантні записи навіть за відсутності спільних слів між запитом і базою знань [3]. Розширення pgvector дозволяє реалізувати цей механізм безпосередньо в PostgreSQL без введення окремої векторної СУБД.

Порівняння RAG і fine-tuning у таблиці 1.2 підтвердило обґрунтованість вибору: RAG є динамічним, прозорим і не потребує ресурсомісткого перенавчання при оновленні бази знань — що є критично важливим для системи, де дизайн-шаблони можуть регулярно поповнюватися [1].

2 ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ

2.1 Логічна модель даних

Логічна модель даних є основою інформаційного забезпечення системи і визначає склад сутностей, їх атрибути та зв'язки між ними незалежно від конкретної СУБД. У цьому підрозділі описано структуру даних системи RAG Page Studio: виявлено два принципово різні типи даних, детально розглянуто кожну сутність із обґрунтуванням прийнятих проектних рішень, а також описано зв'язки між сутностями і доведено відповідність моделі вимогам третьої нормальної форми.

2.1.1 Два типи даних в одній системі

Під час проектування бази даних було виявлено дві ключові задачі. Перша — визначити природу даних, з якими оперує система, та встановити відмінності між їх категоріями. Друга — обрати архітектурне рішення, що задовольняє вимоги обох категорій в єдиній системі зберігання.

Аналіз показав, що система оперує двома принципово різними категоріями даних, і підхід до кожної з них має бути відмінним.

Перша категорія — оперативні транзакційні дані. До них відносяться чати, запити користувачів і результати генерацій. Ці дані з'являються і зникають у процесі звичайної роботи: користувач створює чат, відправляє кілька запитів, потім видаляє все й починає заново. Для цих даних критично важливі цілісність і транзакційність: якщо видалення чату пройде наполовину — система опиниться в неузгодженому стані. Тому тут потрібна повноцінна реляційна модель із зовнішніми ключами, каскадами і транзакційними гарантіями.

Друга категорія — умовно постійні дані, або база знань. Сюди входять шаблони структур сторінок, стильові профілі для різних предметних галузей і каталог зображень із семантичними описами. Ці дані практично не змінюються в процесі звичайної роботи системи — вони заповнюються один раз при запуску або коли адміністратор вирішує розширити бібліотеку шаблонів. Для звичайного користувача ці таблиці доступні лише для читання. По суті, це довідкова інформація — як таблиця тарифів або каталог продуктів у традиційних інформаційних системах.

Ключова технічна особливість умовно постійних даних: кожен запис зберігає ще й векторне представлення свого вмісту — числовий масив, що описує семантичний «зміст» запису у векторному просторі. Саме він дозволяє знаходити семантично близькі шаблони і стилі при пошуку. Як обчислюється цей вектор — описано в розділі 3; тут важливо лише зрозуміти, що це числовий масив розмірності 384, який зберігається в полі типу VECTOR(384). Розмірність 384 визначається архітектурою моделі вбудовування all-MiniLM-L6-v2, що застосовується у системі: її вихідний шар формує вектор саме такої довжини. Детально про цю модель йдеться у розділі 3.

Такий поділ на дві категорії — не просто архітектурне рішення: він безпосередньо впливає на проєктування бази даних. Оперативні таблиці з'єднані жорсткими реляційними зв'язками. Таблиці бази знань не мають прямих зовнішніх ключів; їх зв'язок із оперативними таблицями встановлюється динамічно через векторну схожість у момент кожного запиту. Це дозволяє оновлювати базу знань незалежно від оперативних даних.

2.1.2 Детальний опис оперативних сутностей

Сутність CHATS є контейнером для пов'язаних генерацій одного користувача в рамках спільної тематики або проекту. Наприклад, всі варіанти

лендінгу для ветеринарної клініки природно зберігати в одному чаті — щоб бачити еволюцію результатів і легко порівнювати різні підходи.

Атрибут id реалізований як SERIAL — автоматично інкрементований цілочисельний ідентифікатор.

Атрибут title зберігає текстову назву із обмеженням NOT NULL. Логіка обробки при створенні: рядок обрізається від пробілів, і якщо після обрізки він порожній — підставляється значення за замовчуванням «New Chat». Це обробляє всі крайні випадки: відсутнє поле, None, рядок із пробілів.

Атрибут user_id зберігає ідентифікатор користувача у форматі UUID у вигляді текстового рядка. Зберігання як TEXT, а не як нативний PostgreSQL-тип UUID, спрощує роботу з psycopg2 і виключає необхідність явної конвертації типів при підстановці параметрів у запити. Поле допускає NULL — це забезпечує зворотну сумісність із записами, що з'явились до введення механізму ідентифікації.

Атрибут created_at типу TIMESTAMP зі значенням за замовчуванням CURRENT_TIMESTAMP використовується для сортування списку чатів у порядку спадання — останні чати відображаються першими.

Сутність MESSAGES є найбільш інформаційно насиченою таблицею системи. Кожен запис відповідає одному завершеному раунду взаємодії і є повністю самодостатнім — із нього можна відновити весь контекст конкретної генерації без звернення до інших записів.

Атрибут chat_id є зовнішнім ключем на CHATS із опцією ON DELETE CASCADE. Каскадне видалення означає: видалив чат — автоматично видалились всі його повідомлення на рівні СУБД без жодного додаткового коду в застосунку. Це важливо і для цілісності даних, і для простоти логіки.

Атрибут user_prompt зберігає оригінальний текст запиту у незміненому вигляді.

Атрибут rag_prompt — це те, що реально надсилалось до мовної моделі: збагачений промпт із чотирьох секцій (системний промпт, контекстні дані з бази знань, метадані зображення, запит користувача). Зберігати обидві версії промпту

— оригінальну і збагачену — це усвідомлене рішення. Воно дозволяє в будь-який момент перевірити, які конкретно дані вплинули на результат, і оцінити якість роботи ретривера.

Атрибут generated_html містить повний текст згенерованої HTML-сторінки. Тип TEXT у PostgreSQL не має обмежень на довжину рядка, що дозволяє зберігати документи розміром від кількох до десятків кілобайт, що є необхідністю для масового аналізу створеного контенту

Атрибут preview_path зберігає відносний URL файлу на файловій системі сервера — наприклад.

Атрибут selected_image_url фіксує URL зображення, включеного до конкретної генерації. Допускає NULL у випадках, коли таблиця зображень порожня або жодне зображення не підійшло за контекстом.

2.1.3 Детальний опис сутностей бази знань

Сутність UI_PAGE_STRUCTURE – ця таблиця є реєстром структурних цілостності вебсторінок. Кожен запис описує не конкретну реалізацію сторінки, а загальну функціональну модель: що зазвичай присутнє на сторінці цього типу, як розташовані елементи, яке призначення має кожен блок.

Атрибут page_type містить коротку категоріальну назву: landing_page, dashboard, auth, pricing, about, blog_post, checkout, profile, contact. Цей перелік відкритий і може розширюватися шляхом додавання нових записів.

Атрибут description містить розгорнутий текстовий опис типу сторінки. Саме цей текст разом із page_type стає основою для обчислення векторного ембедінгу. Якість опису прямо впливає на точність семантичного пошуку: чим конкретніше і детальніше описана сторінка, тим точніше система підбере відповідний шаблон для запиту.

Атрибут ui_elements зберігає структурований опис типових компонентів у форматі JSON. Наприклад, для лендінгу це може описувати hero-секцію з

великим заголовком і CTA-кнопкою, блок із трьома колонками переваг, секцію відгуків і форму підписки. Використання JSON тут обумовлено природою даних: структура компонентів є вкладеною і може довільно відрізнятися для різних типів сторінок. Реляційна схема для такого опису була б надмірно складною і жорсткою. PostgreSQL підтримує зберігання JSON через тип TEXT або JSONB; для цього поля обрано TEXT, оскільки SQL-запити всередині JSON-структури не потрібні — поле цілком передається у промпт як рядок [7].

Атрибут embedding типу VECTOR(384) містить числовий вектор, обчислений із конкатенації полів page_type і description. Цей вектор є технічною основою семантичного пошуку.

Сутність UI_BRAND_STYLE Зберігає стильові профілі для різних предметних галузей і типів бізнесу.

Атрибут domain ідентифікує галузь: fintech, healthcare, e-commerce, education, saas, creative, legal. Атрибут tone описує характер комунікації: professional, friendly, minimalist, bold, playful. Атрибут style_description описує детальний текстовий опис кольорової палітри, типографіки і загального візуального напряму.

Наприклад, для healthcare із тоном friendly це може бути: «ніжні відтінки зеленого і синього, гуманістичні sans-serif шрифти, заокруглені кнопки, зображення з теплим освітленням, без агресивних або тривожних кольорів». Саме такий рівень деталізації дозволяє мовній моделі адаптувати стилістику результату під конкретний домен.

Атрибут embedding обчислюється з конкатенації domain, tone і style_description.

Сутність UI_IMAGES – каталог зображень із семантичними описами для включення у згенеровані сторінки.

Атрибути image_url, description, width і height дозволяють системі не тільки підібрати тематично відповідне зображення, а й передати мовній моделі точні розміри для коректного формування тегу без цього модель могла б поставити довільні розміри або взагалі їх не вказати.

Атрибут description є текстовим описом вмісту і настрою зображення: «Золотистий ретривер дивиться у камеру на тлі ветеринарної клініки, тепле освітлення, дружній настрій». Цей текст є основою ембедінгу і визначає результат семантичного пошуку.

2.1.4 Зв'язки між сутностями і нормалізація

Відношення між CHATS і MESSAGES є класичним «один до багатьох»: один чат може містити довільну кількість повідомлень, кожне повідомлення належить рівно одному чату. Реалізується зовнішнім ключем chat_id з ON DELETE CASCADE.

Принципово важливим є рішення щодо відсутності реляційних зв'язків між MESSAGES і таблицями бази знань. Їхній зв'язок є динамічним і семантичним — встановлюється в реальному часі через векторний пошук при кожному запиті. Збережений rag_prompt є опосередкованою фіксацією цього зв'язку: аналізуючи його вміст, завжди можна відновити, які конкретно записи бази знань вплинули на результат.

Така архітектура дає важливу практичну перевагу: нові шаблони і стилі можна додавати до бази знань у будь-який момент без жодних змін у реляційній структурі.

Логічна модель відповідає вимогам третьої нормальної форми [8]. Перша нормальна форма дотримана: всі атрибути є атомарними, немає множинних значень в одному полі. Друга нормальна форма виконана автоматично, оскільки в усіх таблицях єдиний первинний ключ. Третя нормальна форма забезпечена відсутністю транзитивних залежностей: у кожній таблиці всі неключові атрибути залежать виключно від первинного ключа, а не один від одного.

ER-діаграму бази даних наведено на рисунку 2.1.

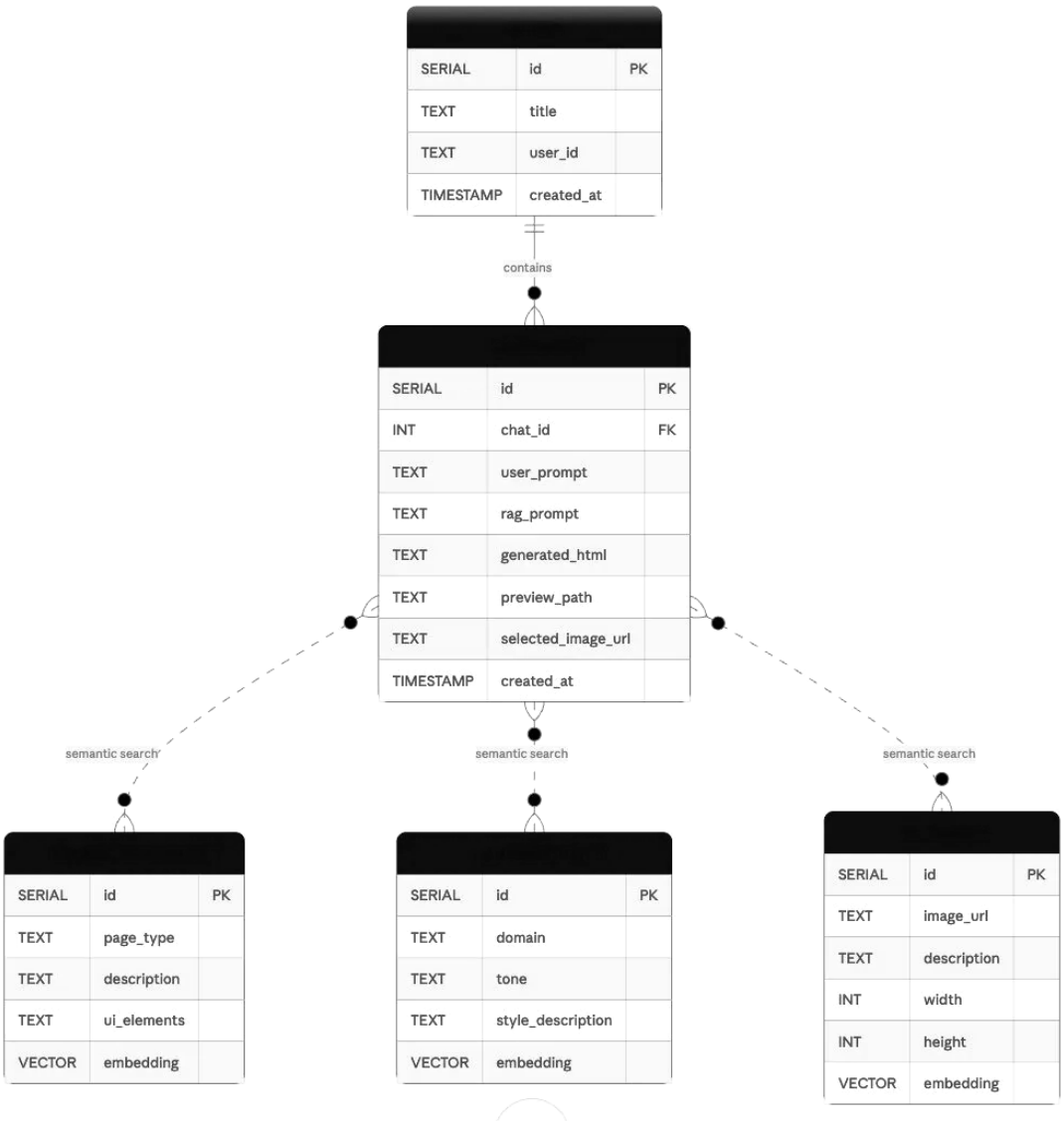

Рисунок 2.1 — ER-діаграма бази даних системи генерації веб-інтерфейсів

З рисунку 2.1 видно поділ на дві функціональні зони. Верхня зона — реляційна частина: CHATS і MESSAGES з'єднані явним зв'язком «contains» з кардинальністю один-до-багатьох. Нижня зона — база знань: три таблиці з полем VECTOR у кожній, що з'єднані з MESSAGES пунктирними лініями «semantic search» замість жорстких зовнішніх ключів. Кожна з таблиць бази знань розвивається незалежно.

2.2 Вибір системи управління базою даних

Вибір СУБД є одним із ключових рішень при проектуванні інформаційного забезпечення системи. Для розробленого застосунку це рішення ускладнюється специфічною технічною вимогою: система одночасно потребує реляційного зберігання оперативних даних із підтримкою транзакцій і зовнішніх ключів та векторного зберігання ембедінгів із підтримкою пошуку за схожістю. Більшість поширених СУБД задовольняють лише одну з цих вимог, що робить вибір нетривіальним. У цьому підрозділі сформульовано критерії вибору, розглянуто альтернативи і обґрунтовано вибір PostgreSQL із розширенням pgvector як єдиного рішення, що відповідає всім вимогам проекту.

2.2.1 Критерії вибору

До СУБД для цього проекту було сформовано такі вимоги:

1. Підтримка реляційної моделі та векторних операцій в єдиній системі — це основна і принципова вимога, що суттєво звужує перелік кандидатів.
2. Підтримка транзакцій ACID і зовнішніх ключів із каскадними операціями.
3. Наявність зрілого і стабільного Python-драйвера для інтеграції з серверним застосунком.
4. Відкрита ліцензія без комерційних обмежень для вільного розгортання.
5. Якість документації і активна спільнота як гарантія довгострокової підтримки.

2.2.2 Аналіз альтернатив

MySQL 8.0 і MariaDB: зрілі і поширені реляційні СУБД із відмінною продуктивністю для стандартних CRUD-операцій. MySQL добре знайома більшості розробників і широко підтримується хостинг-провайдерами. Але нативної підтримки векторних типів немає. Для семантичного пошуку довелося б підключати окрему систему — Qdrant або Milvus — і підтримувати дві бази даних паралельно. Це подвоює складність інфраструктури, резервного копіювання і моніторингу без жодного реального виграшу для проекту такого масштабу.

MongoDB: документо орієнтована СУБД, добре підходить для гнучко-схемних даних. Починаючи з версії 6.0 MongoDB підтримує Atlas Vector Search, але тільки у хмарному сервісі Atlas — не в локальній community-версії. Для проекту, де важлива можливість локального розгортання, це критичне обмеження. Крім того, підтримка ACID-транзакцій і реляційних зв'язків у MongoDB значно слабша, ніж у PostgreSQL.

Qdrant, Milvus, Weaviate: спеціалізовані векторні бази даних, оптимізовані для зберігання мільярдів ембедінгів і ANN-пошуку. Для задач, де основне навантаження — пошук серед великих векторних колекцій, вони перевершують pgvector за продуктивністю. Але вони є вузькоспеціалізованими: реляційних транзакцій, зовнішніх ключів і стандартного SQL вони не підтримують. Тобто поряд із ними все одно потрібна реляційна СУБД для оперативних даних. Дві СУБД паралельно — це непотрібна складність для проекту, де векторна колекція налічує кілька сотень записів.

SQLite: відмінний вибір для локальної розробки і тестування. Проста у налаштуванні, не потребує окремого сервера. Але не підтримує pgvector, погано підходить для паралельних записів і не масштабується до виробничого навантаження.

2.2.3 PostgreSQL і властивості ACID

PostgreSQL — об'єктно-реляційна СУБД з відкритим кодом, що активно розвивається з 1996 року. Для оперативних даних системи критично важливою є підтримка ACID-властивостей [5].

Atomicity (Атомарність). Транзакція виконується повністю або відкочується цілком — проміжних станів не існує. Коли система зберігає повідомлення, виконується кілька операцій: INSERT до messages і запис HTML-файлу. Якщо щось пішло не так — незавершений запис не залишається в базі. Без атомарності можливі ситуації, де файл є, а запису в базі немає, або навпаки [5].

Consistency (Узгодженість). База завжди перебуває у валідному стані. Обмеження цілісності — NOT NULL, зовнішні ключі, перевірки типів — не дозволяють записати некоректні дані. Зовнішній ключ chat_id просто не пропустить повідомлення для неіснуючого чату [5].

Isolation (Ізоляція). Паралельні транзакції не бачать незавершені зміни одна одної. Це важливо при одночасній роботі кількох користувачів: поки один зберігає результат генерації, інший отримує свій список чатів без «фантомних» записів [5].

Durability (Довговічність). Після підтвердження транзакції дані збережені назавжди — навіть якщо сервер одразу після цього впав. PostgreSQL реалізує це через журнал транзакцій WAL (Write-Ahead Log), що записує зміни до їх застосування до основних файлів даних [5].

2.2.4 Розширення pgvector і векторні операції

pgvector — розширення PostgreSQL з відкритим кодом, що реалізує підтримку векторних операцій безпосередньо в СУБД [4]. Встановлюється однією командою:

CREATE EXTENSION IF NOT EXISTS vector;

Після цього PostgreSQL отримує кілька нових можливостей [4].

Тип даних VECTOR(n) дозволяє зберігати числові вектори фіксованої розмірності в стовпцях таблиць. У цьому проекті використовується розмірність 384, що відповідає вихідній розмірності моделі вбудовування (детальніше в розділі 3).

Оператор “<->” обчислює евклідову (L2) відстань між двома векторами. Саме цей оператор використовується для ранжування результатів пошуку: ORDER BY embedding <-> '[0.1, 0.2, ...]'::vector. Чим менша відстань — тим ближчий за змістом запис [6].

Оператор “<=>” обчислює косинусну відстань — метрику, нечутливу до довжини векторів. Для нормалізованих векторів евклідова і косинусна відстані дають еквівалентне ранжування, тому вибір між ними є практичним питанням. Оскільки модель all-MiniLM-L6-v2 виробляє ненормалізовані вектори, у системі застосовується евклідова відстань як стабільніша метрика [6].

Щодо індексів: pgvector підтримує два типи. Індекс IVFFLAT розбиває векторний простір на кластери методом k-means і при пошуку переглядає лише найближчі кластери. Індекс HNSW будує ієрархічний граф і забезпечує суб-логарифмічну складність пошуку при вищій точності, але більших вимогах до пам'яті [6].

Для бази знань із кількома десятками записів повний перебір без індексу є достатнім і навіть швидшим, ніж пошук через індекс (накладні витрати на обхід графу перевищують час прямого перебору). При масштабуванні до тисяч і більше записів рекомендується HNSW [6]:

```
CREATE INDEX ON ui_page_structure USING hnsw (embedding vector_l2_ops);
```

Практичний приклад семантичного SQL-запиту у цій системі:

```
SELECT page_type, description, ui_elements,
    embedding <-> %s::vector AS distance
FROM ui_page_structure
ORDER BY embedding <-> %s::vector
```

```
LIMIT 1;
```

Цей запит знаходить найближчий за семантикою шаблон сторінки до вектора запиту користувача. Аналогічні запити виконуються по таблицях стилів і зображень [4].

2.3 Фізична структура бази даних

Фізична структура бази даних є реалізацією логічної моделі засобами конкретної СУБД — PostgreSQL із розширенням pgvector. На відміну від логічної моделі, що описує концептуальну схему незалежно від технологій, фізична структура відображає реальні типи даних, обмеження цілісності і специфічні конструкції PostgreSQL. У цьому підрозділі наведено SQL-схеми всіх п'яти таблиць, описано підхід до ініціалізації схеми при запуску застосунку і процес первинного наповнення умовно постійних даних.

2.3.1 SQL-схеми таблиць

Фізична реалізація логічної моделі в PostgreSQL охоплює п'ять таблиць.
Таблиця чатів:

```
CREATE TABLE IF NOT EXISTS chats (
   id         SERIAL PRIMARY KEY,
   title      TEXT NOT NULL,
   user_id    TEXT,
   created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Таблиця повідомлень із каскадним видаленням:

```
CREATE TABLE IF NOT EXISTS messages (
   id                SERIAL PRIMARY KEY,
```

```
    chat_id          INT REFERENCES chats(id) ON DELETE CASCADE,
    user_prompt      TEXT NOT NULL,
    rag_prompt       TEXT NOT NULL,
    generated_html   TEXT NOT NULL,
    preview_path     TEXT NOT NULL,
    selected_image_url TEXT,
    created_at       TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Таблиця шаблонів сторінок із JSON-полем і векторним ембедінгом:

```
CREATE TABLE IF NOT EXISTS ui_page_structure (
    id          SERIAL PRIMARY KEY,
    page_type   TEXT NOT NULL,
    description TEXT NOT NULL,
    ui_elements TEXT,
    embedding   VECTOR(384)
);
```

Таблиця стильових профілів:

```
CREATE TABLE IF NOT EXISTS ui_brand_style (
    id              SERIAL PRIMARY KEY,
    domain          TEXT NOT NULL,
    tone            TEXT NOT NULL,
    style_description TEXT NOT NULL,
    embedding       VECTOR(384)
);
```

Таблиця каталогу зображень:

```
CREATE TABLE IF NOT EXISTS ui_images (
    id          SERIAL PRIMARY KEY,
    image_url   TEXT NOT NULL,
    description TEXT NOT NULL,
    width       INT,
```

```
    height      INT,
    embedding   VECTOR(384)
);
```

Наявність поля VECTOR(384) у всіх трьох таблицях бази знань є технічним втіленням архітектурного рішення про семантичний пошук [4].

2.3.2 Фізична реалізація у середовищі PostgreSQL

На відміну від ER-діаграми, що показує концептуальну модель незалежно від СУБД, фізична реалізація відображає реальну структуру таблиць у PostgreSQL із урахуванням специфічних типів даних. Фізичну структуру таблиць у середовищі PostgreSQL представлено на рисунку 2.2.

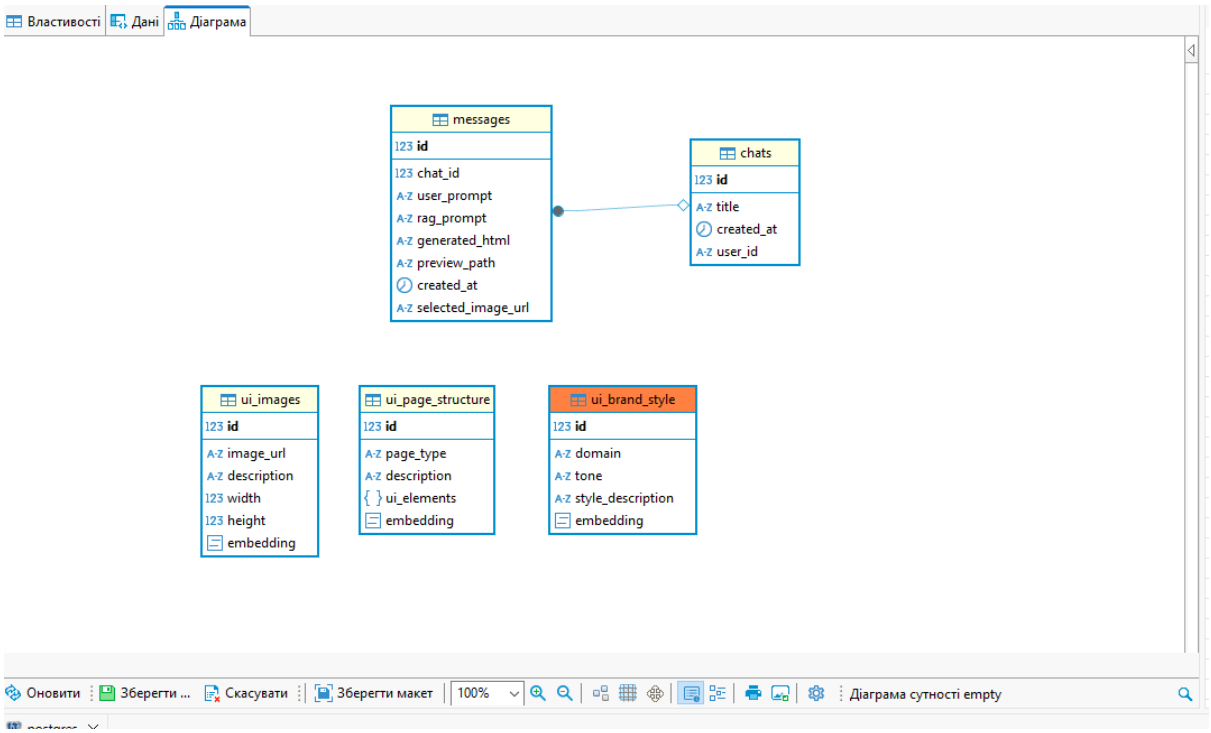

Рисунок 2.2 — Фізична структура таблиць бази даних у середовищі PostgreSQL

З рисунку 2.2 видно реальну реалізацію всіх п'яти таблиць. Тип VECTOR(384), доданий розширенням pgvector, є ключовою відмінністю від

стандартних реляційних схем і підтверджує вибір PostgreSQL як єдиної СУБД, здатної поєднати реляційне і векторне зберігання. Тип SERIAL для id гарантує автоматичну унікальність первинних ключів без ручного управління лічильниками.

2.3.3 Стратегія ініціалізації і міграцій схеми

У зрілих виробничих системах зміни схеми бази даних управляються спеціалізованими інструментами міграцій — Alembic для Python-проектів або Flyway для Java. Ці інструменти зберігають версійність схеми і дозволяють відкочувати зміни.

У цьому проекті застосовано спрощений підхід із ідемпотентними директивами. Функція init_tables викликається при кожному старті застосунку і використовує конструкції, що не ламаються при повторному виконанні:

CREATE TABLE IF NOT EXISTS chats (...);

ALTER TABLE chats ADD COLUMN IF NOT EXISTS user_id TEXT;

ALTER TABLE messages ADD COLUMN IF NOT EXISTS selected_image_url TEXT;

Завдяки цьому при першому запуску всі таблиці і стовпці створюються. При наступних запусках функція переконується, що все є, і нічого не чіпає. Поля user_id і selected_image_url були додані на пізніших етапах розробки саме через ALTER TABLE IF NOT EXISTS — без втрати наявних даних і без потреби пересоздавати таблиці.

2.3.4 Наповнення умовно постійних даних

Таблиці бази знань наповнюються скриптом seed_knowledge_base.py, що виконується окремо від основного застосунку — один раз при розгортанні або при оновленні бази шаблонів. Повний лістинг скрипту наведено у Додатку В.

Процес наповнення для кожного запису включає такі кроки: формування текстового рядка для векторизації шляхом конкатенації ключових полів; обчислення векторного ембедінгу через модель вбудовування; форматування вектора у рядок pgvector-формату [x1,x2,...,xn]; виконання SQL INSERT із збереженням усіх полів і ембедінгу.

Критична умова: всі ембедінги в таблицях бази знань і всі ембедінги запитів користувачів повинні бути обчислені однією і тією ж моделлю. Вектори різних моделей живуть у різних векторних просторах і не є порівнянними. Зміна моделі вбудовування вимагає повного перерахунку всіх збережених ембедінгів.

Для наочності прикладу запис для сторінки входу ветеринарної клініки виглядає так: page_type = "auth", description = "Сторінка входу для ветеринарного вебзастосунку. Форма з email і паролем, логотип клініки, зображення тварини, дружній тон". Відповідний стильовий профіль: domain = "healthcare", tone = "friendly", style_description = "Ніжні відтінки зеленого і синього, sans-serif шрифти, заокруглені кнопки, тепле освітлення".

Висновки до розділу 2

У другому розділі розроблено повне інформаційне забезпечення системи.

Введено поділ даних на оперативні транзакційні (CHATS, MESSAGES) і умовно постійні (UI_PAGE_STRUCTURE, UI_BRAND_STYLE, UI_IMAGES). Детально описано всі атрибути кожної сутності з обґрунтуванням прийнятих проектних рішень: вибір SERIAL замість UUID для id, зберігання user_id як TEXT, використання JSON для ui_elements, каскадне видалення через ON

DELETE CASCADE. Модель нормалізована до 3НФ [8]. Відсутність прямих реляційних зв'язків між таблицями бази знань і MESSAGES є усвідомленим рішенням, що забезпечує незалежність розвитку двох частин системи.

Обґрунтовано вибір PostgreSQL з pgvector як єдиного рішення, що задовольняє всі вимоги: ACID-транзакції [5], реляційні зв'язки і векторний пошук в одній СУБД [4]. Розглянуто чотири оператори pgvector і два типи індексів (IVFFLAT і HNSW) із рекомендаціями щодо їх застосування [6].

Описано SQL-схеми п'яти таблиць, фізичну реалізацію в PostgreSQL, стратегію ідемпотентної ініціалізації через IF NOT EXISTS і процес наповнення умовно постійних даних скриптом seed_knowledge_base.py.

3 ПРИКЛАДНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Організаційна структура програмного забезпечення

Розроблений застосунок RAG Page Studio побудований за дворівневою клієнт-серверною архітектурою. Серверна частина реалізує всю бізнес-логіку: HTTP-маршрутизацію, семантичний пошук, виклик LLM API і збереження результатів. Клієнтська частина є SPA-застосунком, що взаємодіє із сервером виключно через REST API без прямого доступу до бази даних або зовнішніх сервісів.

Серверна частина складається з чотирьох модулів із суворо розмежованою відповідальністю. Модуль app.py є точкою входу і HTTP-шаром. Модуль config.py завантажує конфігурацію з середовища. Модуль db.py реалізує патерн Repository і є єдиним місцем виконання SQL-запитів. Модуль rag.py реалізує повний RAG-пайплайн. Структурну організацію пакетів і компонентів детально описано у підрозділах 3.3 і 3.4.

3.2 Моделювання структури та взаємодії об'єктів

Після визначення організаційної структури програмного забезпечення необхідно формально описати динамічну поведінку системи — тобто яким чином компоненти взаємодіють між собою в процесі виконання конкретного сценарію. Для цих цілей у нотації UML використовують діаграми взаємодії, серед яких найпоширенішою є діаграма послідовності (Sequence Diagram).

Діаграма послідовності відображає учасників взаємодії у вигляді вертикальних ліній життя і показує повідомлення між ними у хронологічному порядку зверху вниз. На відміну від діаграми компонентів, що показує статичну

структуру, діаграма послідовності розкриває часовий порядок викликів, умовні розгалуження і повернення результатів. Це робить її особливо корисною для розуміння складних пайплайнів із великою кількістю учасників.

В розробленій системі основним і найскладнішим сценарієм є генерація HTML-інтерфейсу за текстовим запитом. Цей сценарій залучає шість учасників — Користувача, Веб-інтерфейс, app.py, rag.py, db.py, PostgreSQL і Groq API — і включає умовне розгалуження щодо вибору зображення. Для формального опису цього сценарію застосовано діаграму послідовності. Діаграму зображено на рисунку 3.1.

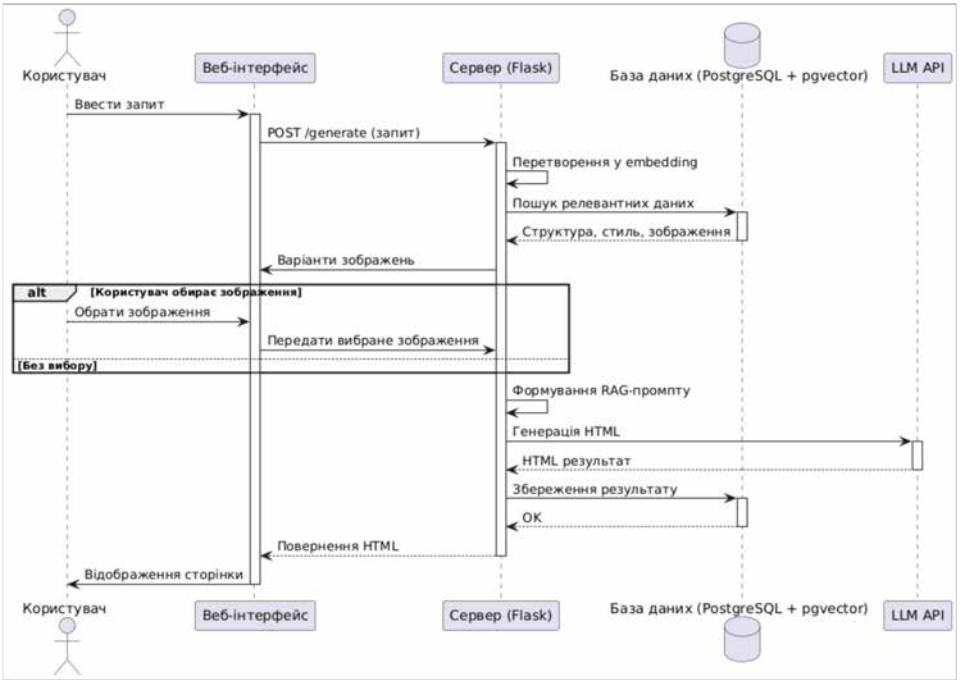

Рисунок 3.1 — Діаграма послідовності процесу генерації HTML-інтерфейсу

З рисунку 3.1 видно повну послідовність взаємодій між шістьма учасниками: Користувачем, Веб-інтерфейсом, app.py, rag.py, db.py, PostgreSQL і Groq API.

Взаємодія починається з того, що Веб-інтерфейс надсилає POST /api/chats/{id}/generate до app.py. Першим кроком виконується валідація

заголовка X-User-Id через get_request_user_id(). Далі перевіряється приналежність чату через chat_exists() із запитом до PostgreSQL: при невдачі повертається 404. Після успішної перевірки app.py передає керування до rag.py через generate_html().

У rag.py виконується encode() для обчислення ембедінгу, потім fetch_rag_context() з трьома векторними запитами до PostgreSQL по таблицях структур, стилів і зображень. Блок alt показує два сценарії вибору зображення: якщо користувач явно вказав URL — виконується fetch_image_by_url(); якщо ні — обирається перший кандидат за семантикою. Після побудови rag_prompt виконується виклик Groq API і clean_html(). Результат записується у файл static/preview/uuid.html і зберігається через save_message(). Клієнт отримує JSON із preview_path і завантажує iframe.

3.3 Організаційна структура програмного забезпечення

Для відображення організаційної структури серверної частини застосовано діаграму пакетів. Вона показує логічне групування компонентів системи та залежності між групами. Діаграму пакетів зображено на рисунку 3.2.

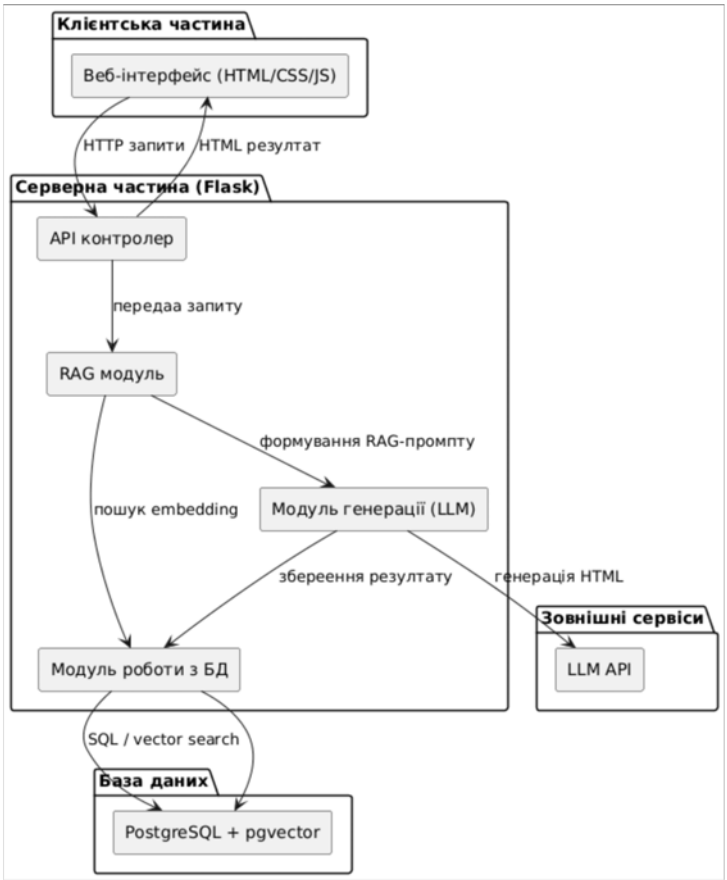

Рисунок 3.2 — Діаграма пакетів програмного забезпечення системи

З рисунку 3.2 видно шість пакетів із чітко визначеними залежностями. Пакет Frontend містить компоненти UI Pages, Chat Interface і Preview Viewer та взаємодіє із системою виключно через API Layer. Пакет API Layer включає Flask Routes і Controllers і є посередником між клієнтом і серверною логікою. Пакет RAG Core реалізує ядро системи: Embedding Service, Context Retriever, Image Ranker і Prompt Builder. Пакет Chat Management містить Chat Service і History Manager. Пакет Generation реалізує фінальний крок пайплайну через LLM Client і HTML Generator. Пакет Storage є рівнем зберігання і включає PostgreSQL і File Storage. Однопрямований потік залежностей Frontend → API Layer → Chat Management → Generation → Storage дозволяє замінювати компоненти всередині пакету без зміни інтерфейсів між пакетами.

3.4 Компонентна структура системи

Для відображення внутрішньої структури компонентів і фізичних зв'язків між ними застосовано діаграму компонентів. На відміну від діаграми пакетів, що показує логічну організацію, діаграма компонентів відображає реальні модулі системи і напрями потоків даних між ними. Діаграму компонентів зображено на рисунку 3.3.

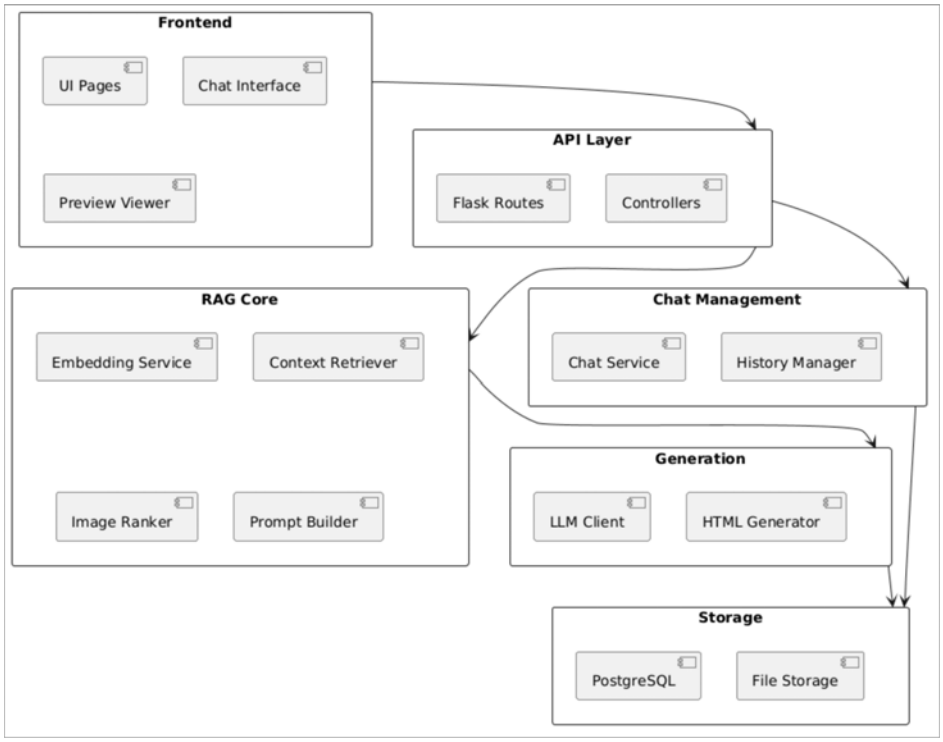

Рисунок 3.3 — Діаграма компонентів системи RAG Page Studio

З рисунку 3.3 видно чотири рівні системи. Клієнтська частина — Веб-інтерфейс (HTML/CSS/JS) — взаємодіє із серверною частиною через двосторонній HTTP-канал. Серверна частина (Flask) включає API контролер, RAG модуль, Модуль генерації (LLM) і Модуль роботи з БД. API контролер приймає запити і виконує первинну валідацію, після чого передає їх до RAG модуля. RAG модуль виконує два паралельні процеси: пошук embedding через Модуль роботи з БД і формування RAG-промпту для Модуля генерації. Модуль генерації надсилає промпт до зовнішнього LLM API і отримує HTML. Модуль

роботи з БД є єдиною точкою доступу до бази даних і виконує SQL та векторний пошук до PostgreSQL + pgvector. Зовнішні сервіси представлені LLM API, що виконує безпосередню генерацію HTML-коду. Такий розподіл забезпечує чітке розмежування відповідальності: жоден інший компонент серверної частини не взаємодіє з базою даних безпосередньо.

3.5 Вибір інструментарію для створення ППЗ

Вибір технологічного стека є одним із ключових архітектурних рішень, що безпосередньо впливає на якість, продуктивність і підтримуваність системи. У цьому підрозділі обґрунтовано вибір мови програмування, вебфреймворку, моделі вбудовування, LLM-сервісу і технологій клієнтської частини. Для кожного компонента розглянуто технічні вимоги і пояснено чому обраний інструмент є оптимальним для задач цього проекту.

3.5.1 Мова програмування Python 3.11

Для реалізації серверної частини обрано мову програмування Python версії 3.11 [9]. Вибір обумовлений такими чинниками:

- Python є стандартом де-факто для застосунків у галузі машинного навчання і обробки природної мови. Бібліотека sentence-transformers, без якої неможлива реалізація семантичного пошуку, розроблена виключно для Python [12].
- Лаконічний синтаксис мінімізує обсяг шаблонного коду: dataclass для конфігурації, contextmanager для управління з'єднаннями, анотації типів для документування.
- Екосистема PyPI пропонує готові рішення для всіх потреб: Flask для HTTP, psycopg2 для PostgreSQL, groq для LLM API, python-dotenv для конфігурації.

3.5.2 Фреймворк Flask

Для реалізації HTTP-шару обрано мікрофреймворк Flask [10]. Вибір базується на попередньому досвіді роботи з цим інструментом, а також на його технічних перевагах для цього проекту.

Flask не нав'язує структуру проекту і не містить зайвих залежностей. Синхронна природа фреймворку ідеально узгоджується із синхронними бібліотеками psycopg2 і sentence-transformers. Можливість писати чистий SQL без ORM є ключовою для використання pgvector-специфічних операторів [11]. Глобальний обробник помилок через @app.errorhandler забезпечує централізовану обробку виключень — будь-який ValueError з будь-якого модуля автоматично перетворюється на JSON-відповідь зі статусом 400.

3.5.3 Модель вбудовування all-MiniLM-L6-v2

Для обчислення векторних представлень тексту застосовано бібліотеку sentence-transformers і модель all-MiniLM-L6-v2 [12, 13].

Модель all-MiniLM-L6-v2 є результатом застосування технології дистиляції знань до архітектури MiniLM. Дистиляція — це процес навчання меншої моделі-«учня» відтворювати поведінку більшої моделі-«вчителя». У випадку MiniLM учень відтворює матриці відносин само-уваги останнього шару вчителя, що дозволяє зберегти якість семантичного розуміння при суттєвому скороченні розміру [13].

Вихідна точка — модель BERT-base з 12 трансформерними шарами і 110 мільйонами параметрів. all-MiniLM-L6-v2 має лише 6 шарів і 22 мільйони параметрів — вп'ятеро менше. Технічні характеристики моделі наведено у таблиці 3.1.

Таблиця 3.1

Технічні характеристики моделі all-MiniLM-L6-v2

Параметр	**Значення**
Кількість трансформерних шарів	6
Кількість параметрів	~22 мільйони
Розмірність вихідного ембедінгу	384
Розмір файлів моделі	~90 МБ
Час обчислення ембедінгу (CPU)	5–15 мс
Ліцензія	Apache 2.0
Навчальний корпус	>1 млрд пар речень

Розмірність 384 визначається архітектурою моделі і безпосередньо відповідає типу VECTOR(384) у таблицях бази знань PostgreSQL. Модель виробляє ненормалізовані вектори, тому у системі застосовується евклідова відстань (оператор <-> у pgvector) як стабільніша метрика [13].

У застосунку модель ініціалізується один раз при старті через декоратор @lru_cache(maxsize=1), що виключає повторне завантаження при кожному запиті:

```
@lru_cache(maxsize=1)
def get_embedding_model() -> SentenceTransformer:
    return SentenceTransformer(settings.embedding_model_name)
```

3.5.4 Groq API і модель llama-3.3-70b-versatile

Для генерації HTML застосовується модель llama-3.3-70b-versatile через API платформи Groq [14]. Модель Llama 3.3 є відкритою розробкою Meta AI [15] із такими характеристиками: 70 мільярдів параметрів; контекстне вікно 128 000 токенів; оптимізація для instruction following; відкрита ліцензія.

Платформа Groq застосовує архітектуру LPU (Language Processing Unit) — спеціалізований процесор для послідовної генерації токенів [14]. Швидкість інференсу становить 500–1500 токенів на секунду, що для HTML-сторінки розміром 2000–4000 токенів дає час відповіді 3–8 секунд. Конфігурація повністю параметризована через змінні оточення LLM_API_KEY, LLM_MODEL, LLM_BASE_URL, що дозволяє без змін у коді перемикатися між Groq, OpenAI і локальним Ollama.

3.5.5 Клієнтська частина: нативний JavaScript

Клієнтська частина реалізована на нативному JavaScript [16]. Вибір обумовлено трьома аргументами: функціональна складність клієнта є помірною і не потребує SPA-фреймворку; відсутність build-step спрощує розгортання; HTML-шаблони через тег template забезпечують повторне використання розмітки без бібліотечних абстракцій.

3.6 Алгоритмізація та програмування програмних модулів

У цьому підрозділі описано реалізацію трьох основних програмних модулів системи: модуля REST API з алгоритмом взаємодії із базою даних, модуля RAG-генерації і клієнтського інтерфейсу. Для кожного модуля наведено ключові алгоритмічні рішення, описано логіку обробки запитів і проілюстровано результати роботи.

3.6.1 Модуль REST API і взаємодія з базою даних

Серверний API містить дев'ять ендпоінтів, що реалізують варіанти використання. Перелік ендпоінтів наведено у таблиці Д.1 Додатку Д.

Валідація вхідних даних реалізована на кількох рівнях. Функція get_request_user_id перевіряє наявність заголовка X-User-Id і коректність його формату через uuid.UUID(). Конвертер int:chat_id у маршрутах автоматично відхиляє запити з нечисловим ідентифікатором на рівні маршрутизації Flask. Функція get_json(silent=True) повертає порожній словник при відсутності або некоректному JSON-тілі. Порожній промпт перевіряється явно і повертає статус 400 до запуску RAG-пайплайну. Глобальний обробник @app.errorhandler (ValueError) централізовано перетворює всі помилки валідації на JSON-відповідь зі статусом 400.

Алгоритм взаємодії із базою даних зображено на рисунку 3.4.

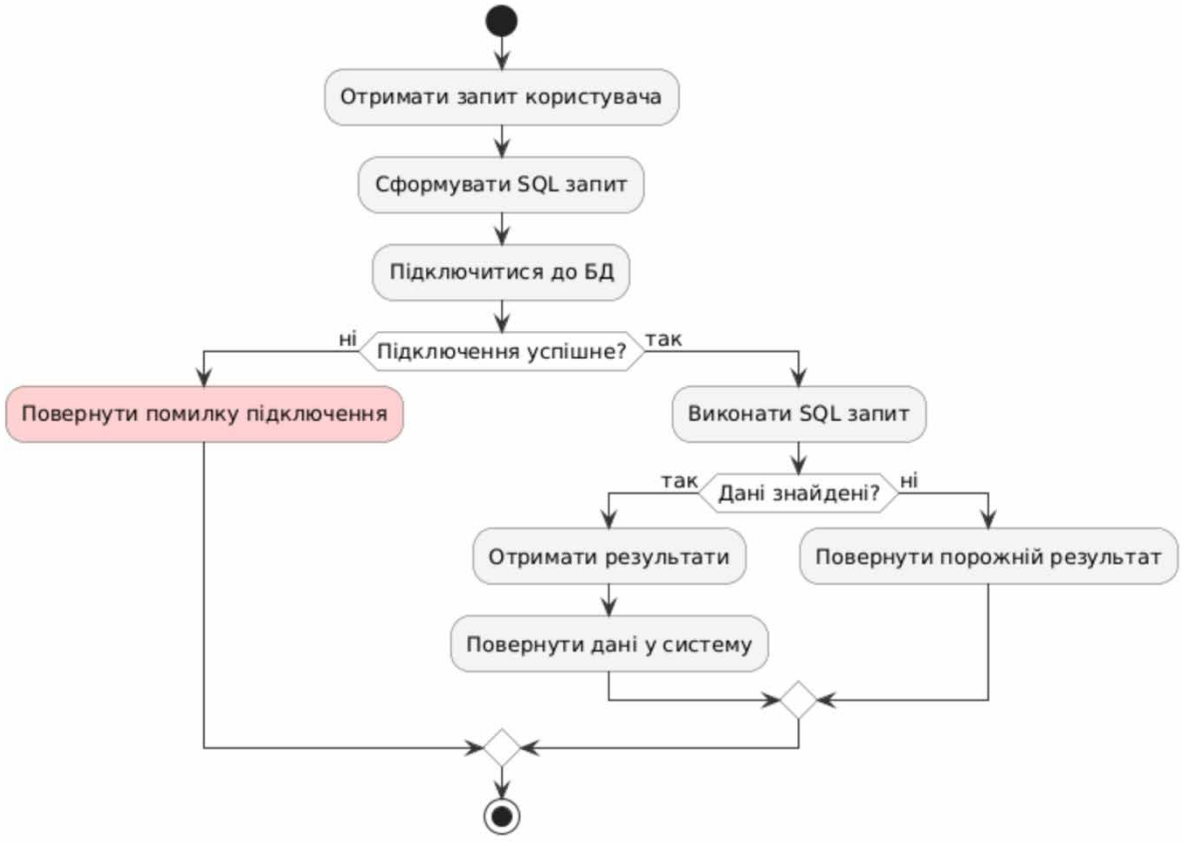

Рисунок 3.4 — Блок-схема алгоритму взаємодії із базою даних

З рисунку 3.4 видно, що алгоритм містить два вузли прийняття рішення у вигляді ромбів. Перший перевіряє успішність підключення до PostgreSQL: при невдачі виконання повністю зупиняється і повертається помилка підключення. При успішному підключенні виконується сформований SQL-запит. Другий ромб перевіряє наявність результату: якщо дані не знайдено — повертається порожній результат як нормальний сценарій без генерації виключення; якщо дані є — вони серіалізуються і повертаються у систему. В обох випадках з'єднання закривається через блок finally контекстного менеджера get_conn.

3.6.2 RAG-модуль

RAG-модуль реалізує повний пайплайн від текстового запиту до згенерованого HTML. Загальний алгоритм роботи модуля описано у вигляді діаграми діяльності у підрозділі 1.2 (рисунок 1.2). Детальну послідовність взаємодій між компонентами при виконанні генерації описано у підрозділі 3.2 (рисунок 3.1).

Функція generate_html виконує такі кроки. Перший — обчислення ембедінгу запиту: model.encode([user_prompt])[0].tolist() повертає вектор із 384 дійсних чисел. Другий — семантичний пошук: fetch_rag_context(embedding) виконує три векторних SQL-запити. При порожній таблиці функція генерує RuntimeError, що перехоплюється обробником ендпоінта. Третій — вибір зображення: за трьома гілками — явний URL, перший за семантикою або None. Четвертий — побудова rag_prompt із чотирьох секцій із явними заголовками. П'ятий — виклик Groq API. Шостий — clean_html() для видалення markdown-огорток. Сьомий — повернення словника з результатами.

3.6.3 Клієнтський інтерфейс

Клієнтська частина реалізована як SPA із двопанельною компоновкою. Головний екран застосунку зображено на рисунку 3.5.

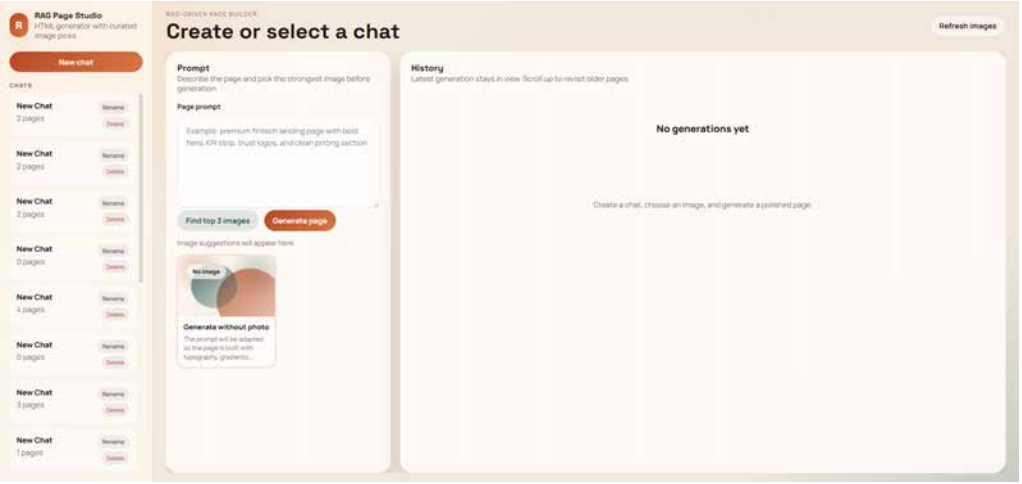

Рисунок 3.5 — Головний екран застосунку RAG Page Studio

Ліва панель містить брендовий блок, кнопку «New chat» і список сеансів із кнопками «Rename» і «Delete». Компоновку з активним сеансом зображено на рисунку 3.6.

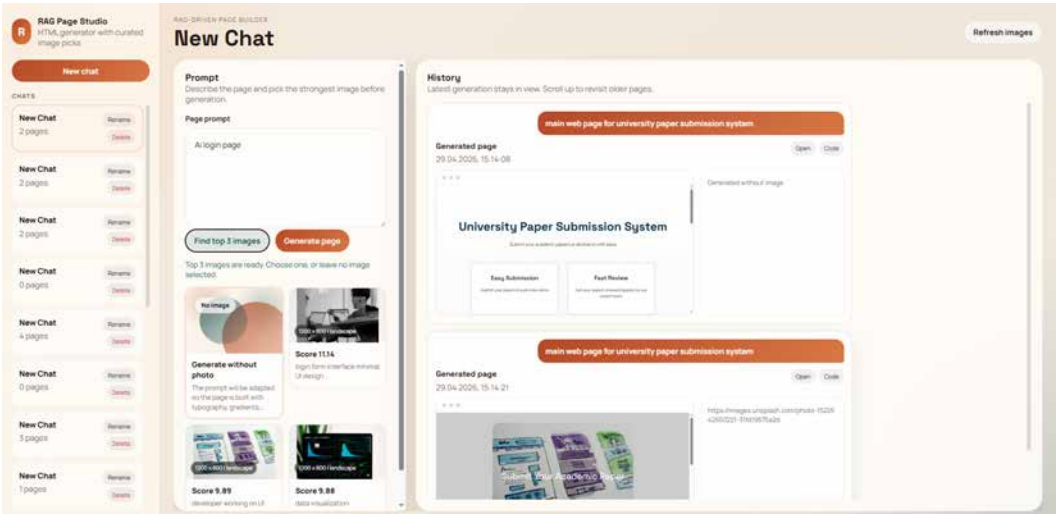

Рисунок 3.6 — Двопанельна компоновка інтерфейсу застосунку

Панель генерації містить текстове поле для запиту, кнопку «Find top 3 images» для семантичного підбору зображень через POST /api/image-options, сітку карток-кандидатів, кнопку «Generate page» і анімований лоадер під час очікування відповіді.

Результат генерації відображається у вигляді картки з iframe-переглядом. Приклад картки зображено на рисунку 3.7.

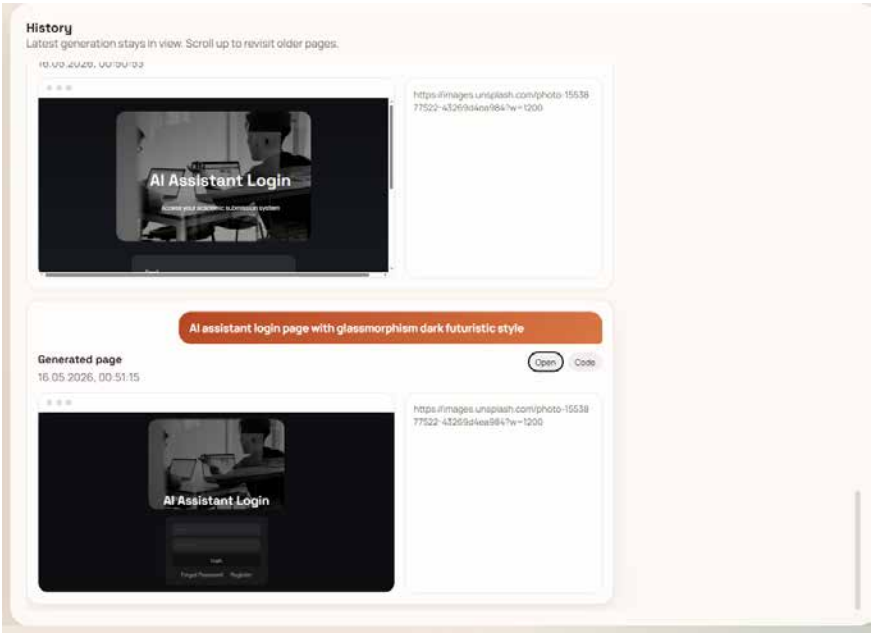

Рисунок 3.7 — Картка результату генерації із попереднім переглядом

При завантаженні JavaScript перевіряє localStorage.getItem('userId'). При відсутності — генерується UUID через crypto.randomUUID() і зберігається. Цей UUID підставляється у заголовок X-User-Id усіх наступних запитів.

3.6.4 Блок-схема алгоритму функції generate_html

Алгоритм функції generate_html є реалізацією повного RAG-пайплайну від вхідного запиту до готового HTML-документа. Блок-схему алгоритму зображено на рисунку 3.8.

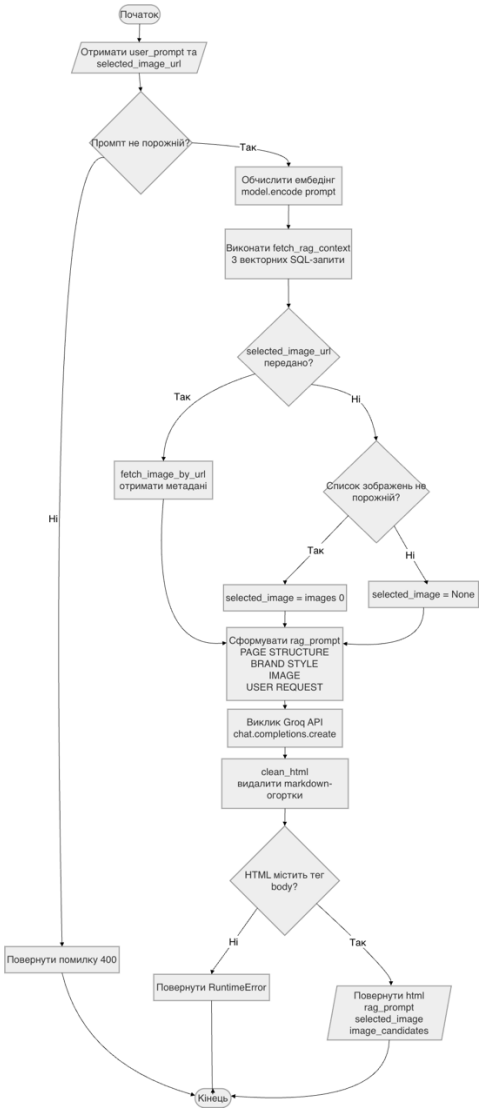

Рисунок 3.8 — Блок-схема алгоритму функції generate_html

З рисунку 3.8 видно, що алгоритм містить три вузли прийняття рішення. Перший перевіряє порожність промпту — при порожньому значенні виконання зупиняється і повертається відповідь із кодом 400 без запуску ресурсомісткого пошуку. Другий визначає спосіб вибору зображення: при явно вказаному URL виконується пошук по базі даних, інакше перевіряється наявність кандидатів і обирається перший або встановлюється None. Третій перевіряє валідність отриманого HTML — відсутність тегу body свідчить про некоректну відповідь моделі і генерує RuntimeError. При успішному проходженні всіх перевірок

функція повертає словник із HTML, збагаченим промптом і метаданими зображення.

3.7 Дослідження точності семантичного ретривера

Розробка системи не обмежується лише реалізацією функціональних модулів — важливим етапом є дослідження якості ключового компонента, що визначає результативність усього підходу. У контексті RAG-системи таким компонентом є ретривер: саме від точності семантичного пошуку залежить релевантність контексту, що передається мовній моделі, і, відповідно, якість згенерованого HTML-інтерфейсу. У цьому підрозділі проведено кількісну оцінку точності ретривера на реальних тестових запитах різних предметних галузей.

3.7.1 Методологія дослідження

Центральним технічним компонентом RAG-системи є ретривер — модуль, що за текстовим запитом знаходить найближчі записи у базі знань через векторний пошук. Від точності ретривера залежить якість усього пайплайну: навіть досконала мовна модель не компенсує неправильно підібраний контекст.

Метою дослідження є кількісна оцінка точності семантичного ретривера при одночасному визначенні типу сторінки (page_type) і стилістичного профілю (brand style) для запитів різних предметних галузей.

Для тестування сформовано 6 комбінованих запитів, що одночасно вказують на тип сторінки і бажаний стиль. Кожен запит навмисно сформульовано інакше, ніж відповідний опис у базі знань — для перевірки семантичного, а не ключового характеру пошуку [17].

3.7.2 Результати тестування

Тестування проводилось на шести комбінованих запитах, кожен із яких одночасно вказував на тип сторінки і стилістичний профіль. Запити навмисно формулювались інакше, ніж відповідні описи у базі знань — для перевірки саме семантичного характеру пошуку. Повні результати тестування із зазначенням знайдених значень і евклідових відстаней наведено у таблиці Г.1 Додатку Г.

За результатами шести тестових запитів система коректно визначила тип сторінки у п'яти випадках із шести, що відповідає точності 83%. Аналогічний показник отримано для визначення стилістичного профілю — 5/6 (83%). Середній час виконання одного векторного запиту становить 4 мс. Значення евклідової відстані для всіх запитів знаходяться у стабільному діапазоні 0.89–1.23 без аномальних викидів.

Середній час виконання: 4 мс. Точність page_type: 5/6 (83%). Точність brand style: 5/6 (83%).

3.7.3 Аналіз результатів

Запити 1, 2, 3, 5 і 6 демонструють коректну роботу семантичного пошуку в умовах, коли запит сформульовано принципово інакше, ніж опис у базі знань. Показовим є запит 3: «travel app forgot password page with bright colors» знайшов запис «Reset password request page with email input and instructions» (відстань 0.916) без жодного спільного слова.

Частковий збіг у запиті 4 пояснюється домінуванням слова «analytics» у запиті: система обрала технічно близький шаблон за відсутності точного відповідника — це демонструє граціозну деградацію. Усі відстані стабільні в діапазоні 0.89–1.23 без аномальних викидів.

3.7.4 Висновки дослідження

З результатів тестування можна сформулювати чотири висновки.

1. Семантичний ретривер системи демонструє точність 83% при одночасному пошуку типу сторінки і стилістичного профілю. Запити навмисно сформульовано інакше, ніж описи у базі знань, що підтверджує семантичну природу пошуку.
2. Єдиний випадок часткового збігу є семантично виправданим. Система обрала технічно близький шаблон і стиль за відсутності точного відповідника — це демонструє граціозну деградацію: прийнятний результат навіть без ідеального збігу.
3. Час виконання 4 мс є достатнім для інтерактивного застосунку і залишає значний запас для масштабування бази знань.
4. Точність ретривера безпосередньо залежить від якості описів у базі знань. Специфічні описи (gaming, forgot_password) дають кращі відстані, ніж загальні (dashboard). Це підкреслює важливість якісного наповнення бази знань.

Висновки до розділу 3

У третьому розділі розроблено прикладне програмне забезпечення системи.

Описано організаційну структуру через три UML-діаграми. Діаграма послідовності (рисунок 3.1) відображає взаємодію між шістьма учасниками при генерації, включаючи блок вибору зображення. Діаграма пакетів (рисунок 3.2) показує шість пакетів із односпрямованим потоком залежностей. Діаграма компонентів (рисунок 3.3) деталізує чотири рівні серверної частини.

Обґрунтовано вибір технологічного стека. Flask обрано на основі досвіду роботи і технічної сумісності із синхронними бібліотеками psycopg2 і sentence-

transformers [10]. Модель all-MiniLM-L6-v2 забезпечує 384-вимірні ембедінги за 5–15 мс на CPU завдяки дистиляції знань [13]. Groq API з llama-3.3-70b забезпечує генерацію HTML за 3–8 секунд завдяки LPU-архітектурі [14, 15].

Описано алгоритми програмних модулів: таблиця 3.2 із дев'ятьма ендпоінтами API, блок-схема взаємодії із БД (рисунок 3.4) із двома ромбами прийняття рішення, опис RAG-пайплайну із посиланнями на рисунки 1.2 і 3.1, клієнтський інтерфейс (рисунки 3.5–3.7).

Проведено дослідження точності семантичного ретривера на шести тестових запитах різних предметних галузей. Точність 83%, середній час 4 мс, стабільний діапазон відстаней 0.89–1.23 підтверджують придатність підходу для реального часу.

4 РЕКОМЕНДАЦІЇ ЩОДО ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЇ СИСТЕМИ

4.1 Тестування системи

Тестування є обов'язковим етапом розробки програмного забезпечення, що дозволяє підтвердити відповідність реалізованої системи сформульованим вимогам і виявити потенційні проблеми до введення в експлуатацію. Для системи RAG Page Studio тестування охоплює як стандартні функціональні перевірки серверного API, так і специфічне для RAG-систем дослідження якості семантичного пошуку — компонента, що не піддається класичному модульному тестуванню і потребує окремої методології оцінки.

4.1.1 Стратегія тестування

Тестування розробленої системи RAG Page Studio проводилось комплексно і охоплювало три напрями: функціональне тестування ендпоінтів REST API; дослідження точності семантичного ретривера; порівняльне тестування якості генерації з RAG-контекстом і без нього.

Функціональне тестування API виконувалось засобами Postman і curl. Для кожного ендпоінта перевірялись нормальні сценарії і граничні випадки з некоректними вхідними даними. Результати фіксувались у вигляді протоколу із порівнянням очікуваного і фактичного результату.

Дослідження ретривера виконувалось шляхом надсилання тестових запитів і аналізу знайдених записів бази знань. Для оцінки точності кожен запит навмисно формулювався інакше, ніж відповідний опис у базі знань — для перевірки саме семантичного, а не ключового пошуку.

Порівняльне тестування RAG versus без контексту виконувалось шляхом тимчасового відключення fetch_rag_context і порівняння результатів при однаковому запиті.

4.1.2 Функціональне тестування серверної частини

Функціональному тестуванню підлягали всі дев'ять ендпоінтів API у нормальних і аномальних сценаріях. Результати тестування зведено у таблицю 4.1.

Таблиця 4.1

Протокол функціонального тестування серверної частини

Тест-кейс	**Вхідні дані**	**Очікуваний результат**	**Фактичний результат**	**Статус**
Створення чату без назви	POST /api/chats, тіло {}	title = "New Chat", 200	Відповідає	✓
Відсутній заголовок X-User-Id	GET /api/chats без заголовка	400, error	Відповідає	✓
Некоректний UUID	X-User-Id: "not-uuid"	400, error	Відповідає	✓
Доступ до чужого чату	GET повідомлень чужого chat_id	404	Відповідає	✓
Генерація без промпту	POST /generate, prompt: ""	400, error	Відповідає	✓
Успішна генерація HTML	Валідний промпт	200, HTML-документ	Відповідає	✓

Таблиця 4.1 (продовження)

Файл попереднього перегляду	GET /preview/uuid.html	200, HTML	Відповідає	✓
Перейменування чату	PATCH /api/chats/{id}	Нова назва, 200	Відповідає	✓

Усі вісім перевірок пройдено успішно. Особливо важливим є перевірка захисту від IDOR (Insecure Direct Object Reference): спроба отримати повідомлення чужого чату повертає 404, а не дані — завдяки умові AND user_id = %s у SQL-запиті. Каскадне видалення підтверджено: після DELETE чату всі його повідомлення автоматично видалені на рівні СУБД через ON DELETE CASCADE.

4.1.3 Порівняльне тестування: RAG versus без контексту

Ключовим якісним тестом є порівняння результатів генерації з RAG-контекстом і без нього для однакового запиту. Для тесту обрано запит «vet clinic login page» як короткий, але тематично специфічний.

Режим без контексту. fetch_rag_context тимчасово відключено, RAG-промпт замінено на оригінальний запит. Результат: мінімалістична форма входу з білим фоном, полями Email і Password і синьою кнопкою Login. Жодних ознак ветеринарної тематики — типовий генеричний результат.

Режим з RAG. База знань містила: шаблон auth зі специфікою ветеринарного застосунку; профіль healthcare з кольорами ніжного зеленого і синього і тоном friendly; зображення золотистого ретривера.

Результат з RAG: двоколонковий layout — ліва колонка із зображенням тварини, назвою клініки і слоганом; права колонка з формою входу. Кольорова

схема відповідає healthcare-профілю. Placeholder-текст: «Your email address», «Your password». Усі ці деталі є прямим наслідком RAG-контексту.

Порівняльну характеристику наведено у таблиці 4.3.

Таблиця 4.3

Порівняння результатів генерації RAG проти без контексту

Характеристика	**Без RAG**	**З RAG**
Структура сторінки	Одна колонка, проста форма	Дві колонки: зображення + форма
Кольорова схема	Стандартний синій	Медичний зелений + синій
Зображення тварини	Відсутнє	Золотистий ретривер (з БЗ)
Типографіка	Arial/Helvetica	Lato (healthcare-профіль)
Тон тексту	«Email», «Password»	«Your email address»
Vet-тематика	Відсутня	Слоган і назва клініки
Стабільність	Варіюється між запитами	Стабільна (контекст фіксований)

Результати підтверджують ключову гіпотезу роботи: RAG-підхід із предметно специфічною базою знань суттєво підвищує тематичну узгодженість і стилістичну коректність згенерованих веб інтерфейсів.

4.2 Вимоги до апаратного та програмного забезпечення

Коректне розгортання системи потребує відповідного апаратного і програмного середовища. У цьому підрозділі визначено мінімальні і рекомендовані вимоги до обладнання, описано необхідне програмне забезпечення і наведено топологію розгортання вузлів системи. Врахування цих вимог є необхідною умовою для стабільної роботи застосунку в реальному середовищі.

4.2.1 Топологія розгортання

Система розгортається відповідно до архітектури,ображеної на рисунку 4.1. У мінімальній конфігурації Application Server і Database Server розміщуються на одному хості. Для середнього навантаження рекомендується розмістити PostgreSQL на окремому сервері і налаштувати Nginx як зворотній проксі перед Flask/Gunicorn.

Зовнішні сервіси — Groq API і Hugging Face Hub для завантаження моделі — доступні через HTTPS і не потребують власного розгортання.
Топологію розгортання системи зображено на рисунку 4.1.

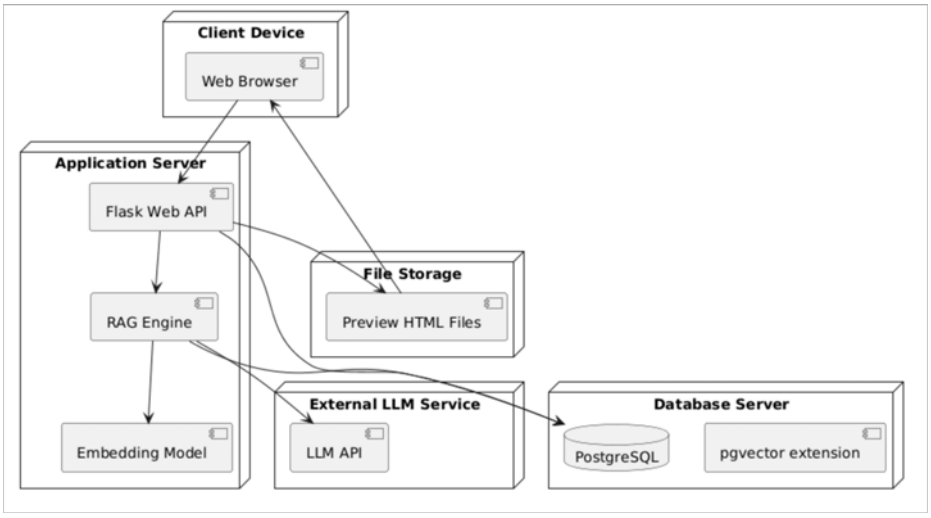

Рисунок 4.1 — Діаграма розгортання системи RAG Page Studio

З рисунку 4.1 видно п'ять вузлів системи. Client Device містить Web Browser. Application Server містить Flask Web API, RAG Engine і Embedding Model. File Storage зберігає Preview HTML Files. External LLM Service надає LLM API через Groq. Database Server містить PostgreSQL і pgvector.

4.2.2 Вимоги до апаратного забезпечення

Для мінімальної конфігурації (розробка, прототипування):

- CPU: від 2 ГГц, 2 ядра
- RAM: від 4 ГБ (2 ГБ для Python-процесу з моделлю, 1 ГБ для PostgreSQL)
- Накопичувач: від 10 ГБ

Для конфігурації середнього рівня (кілька одночасних користувачів):

- CPU: від 4 ядер
- RAM: від 8 ГБ
- SSD: від 50 ГБ

GPU не є обов'язковим: обчислення ембедінгів виконується на CPU, генерація HTML делегується Groq.

4.2.3 Вимоги до програмного забезпечення

Серверне оточення: Ubuntu Server 22.04 LTS або новіша версія; Python 3.10 або вище; PostgreSQL 14 або вище з pgvector 0.5.0+; Nginx (для виробничого розгортання).

Python-залежності визначено у файлі requirements.txt:

flask>=3.0

flask-cors>=4.0

psycopg2-binary>=2.9

sentence-transformers>=2.7

groq>=0.9

python-dotenv>=1.0

4.3 Склад інсталяційного пакету та розгортання

Підготовка системи до розгортання потребує чіткого розуміння складу інсталяційного пакету і послідовності кроків налаштування. У цьому підрозділі описано структуру файлів проекту, покрокову процедуру розгортання і механізм конфігурації через змінні оточення. Дотримання наведеної послідовності гарантує коректну ініціалізацію всіх компонентів системи — від схеми бази даних до наповнення бази знань.

4.3.1 Структура файлів проекту

Інсталяційний пакет містить такі компоненти. Серверна логіка: app.py, config.py, db.py, rag.py. Конфігурація: requirements.txt, .env.example. Клієнтська частина: templates/index.html, static/css/app.css, static/js/app.js. Наповнення бази знань: scripts/seed_knowledge_base.py (лістинг у Додатку В). Директорія static/preview/ створюється автоматично при першому запуску.

4.3.2 Послідовність розгортання

Розгортання системи виконується у сім послідовних кроків.

1. Підготовка середовища. Встановити Python 3.10+, PostgreSQL 14+. Підключити pgvector у psql: CREATE EXTENSION IF NOT EXISTS vector;
2. Розпакування проекту. Клонувати репозиторій або розпакувати архів.
3. Встановлення залежностей.pip install -r requirements.txt

4. Конфігурація. Скопіювати .env.example до .env і заповнити параметри: PG_HOST, PG_PORT, PG_DATABASE, PG_USER, PG_PASSWORD, LLM_API_KEY.
5. Ініціалізація схеми. Запустити застосунок — функція init_tables виконається автоматично і створить усі п'ять таблиць:python app.py
6. Наповнення бази знань.python scripts/seed_knowledge_base.py
7. Запуск. Для розробки: python app.py Для виробництва: gunicorn -w 4 -b 0.0.0.0:5000 app:app

4.3.3 Конфігурація через змінні оточення

Система конфігурується виключно через файл .env без змін у коді. Перелік змінних: APP_HOST і APP_PORT — адреса і порт Flask-сервера; PG_HOST, PG_PORT, PG_DATABASE, PG_USER, PG_PASSWORD — параметри PostgreSQL; LLM_API_KEY і LLM_MODEL — ключ і назва моделі (за замовчуванням llama-3.3-70b-versatile); EMBEDDING_MODEL — модель ембедінгів (за замовчуванням all-MiniLM-L6-v2); PREVIEW_DIR — директорія для HTML-файлів.

Такий підхід відповідає принципам дванадцятифакторних застосунків: конфігурація зберігається в середовищі, а не в коді, що дозволяє розгортати ту саму кодову базу у різних середовищах без змін.

Висновки до розділу 4

У четвертому розділі проведено комплексне тестування системи і підготовлено рекомендації щодо впровадження.

Функціональне тестування дев'яти ендпоінтів REST API підтвердило коректність обробки всіх нормальних і аномальних сценаріїв, включаючи захист від IDOR і каскадне видалення даних.

Дослідження точності семантичного ретривера на шести тестових запитах показало точність 83% при одночасному визначенні типу сторінки і стилістичного профілю. Єдиний частковий збіг є семантично виправданим. Середній час виконання запиту становить 4 мс, що є прийнятним для реального часу.

Порівняльне тестування RAG versus без контексту підтвердило ключову гіпотезу роботи: система з RAG-контекстом генерує тематично узгоджений, структурно і стилістично відповідний інтерфейс, тоді як без контексту результат є генеричним.

Визначено мінімальні та рекомендовані вимоги до апаратного і програмного середовища. Описано склад інсталяційного пакету і покрокову процедуру розгортання за сімома кроками.

ВИСНОВКИ

У бакалаврській кваліфікаційній роботі розроблено інтелектуальну систему автоматичного генерування HTML-інтерфейсів користувача на основі підходу Retrieval-Augmented Generation. Усі поставлені завдання вирішено в повному обсязі.

За результатами системного аналізу предметної галузі встановлено, що зростаючий попит на вебінтерфейси в умовах дефіциту фронтенд-розробників формує стійкий запит на інструменти автоматизованої генерації. Проведений огляд існуючих рішень — від no-code конструкторів до AI-генераторів коду — виявив спільний системний недолік: жодне з них не поєднує семантичний пошук у структурованій базі знань із генерацією HTML і персистентним збереженням результатів. Обґрунтовано вибір підходу RAG як оптимального для даної задачі: він не потребує перенавчання моделі, забезпечує динамічне оновлення бази знань і зберігає прозорість роботи системи.

Розроблено логічну модель даних із п'яти сутностей, нормалізовану до третьої нормальної форми. Модель поєднує реляційні таблиці оперативних даних (CHATS, MESSAGES) і три таблиці бази знань із полями VECTOR(384). Обґрунтовано вибір PostgreSQL з pgvector як єдиної СУБД, що задовольняє вимоги реляційного і векторного зберігання в одній системі. Описано SQL-схеми таблиць і механізм ідемпотентної ініціалізації.

Реалізовано серверну частину з дев'ятьма REST-ендпоінтами на Flask. Обґрунтовано вибір Flask на основі досвіду роботи і технічної сумісності із синхронними бібліотеками psycopg2 і sentence-transformers. Реалізовано повний RAG-пайплайн: обчислення ембедінгу через all-MiniLM-L6-v2 (384-вимірний вектор, 5–15 мс на CPU), семантичний пошук через pgvector, побудова збагаченого промпту з чотирьох секцій, генерація HTML через llama-3.3-70b-versatile (Groq API, 3–8 секунд). Розроблено клієнтський SPA-інтерфейс RAG

Page Studio на нативному JavaScript із двопанельною компоновкою і підтримкою двоетапного воркфлоу підбору зображень.

Проведено комплексне тестування системи. Функціональне тестування дев'яти ендпоінтів API підтвердило коректність обробки нормальних і аномальних сценаріїв. Дослідження точності семантичного ретривера на шести тестових запитах різних предметних галузей показало точність 83% при часі виконання 4 мс. Єдиний частковий збіг є семантично виправданим: система обрала технічно близький шаблон за відсутності точного відповідника в базі знань, що демонструє граціозну деградацію. Порівняльне тестування RAG versus без контексту підтвердило ключову гіпотезу роботи: система з RAG-контекстом генерує тематично узгоджений і стилістично відповідний інтерфейс, тоді як без контексту результат є не стабільним.

Практична цінність роботи полягає в тому, що розроблений застосунок RAG Page Studio дозволяє не технічним фахівцям генерувати прототипи вебінтерфейсів за текстовим описом з урахуванням предметної специфіки. Система адаптується до нових доменів шляхом поповнення бази знань без перенавчання моделі.

Напрямами подальшого розвитку системи можуть бути: впровадження JWT-автентифікації для виробничого середовища; реалізація пулу з'єднань для масштабування; побудова HNSW-індексу при зростанні бази знань; ітеративне редагування результату через діалог; автоматизоване наповнення бази знань.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. What is retrieval-augmented generation? [Електронний ресурс]. IBM, 2024. URL: https://www.ibm.com/think/topics/retrieval-augmented-generation
2. Context Engineering: Best Practices for an Emerging Discipline [Електронний ресурс]. Redis Labs, 2024. URL: https://redis.io/blog/ context-engineering-best-practices-for-an-emerging-discipline (дата звернення: 10.05.2026).
3. Vector Search Explained: How It Works and Why It Matters [Електронний ресурс]. Shaped.ai, 2024. URL: https://www.shaped.ai/blog/vector-search-explained (дата звернення: 10.05.2026).
4. pgvector: Open-source vector similarity search for Postgres [Електронний ресурс]. GitHub, 2024. URL: https://github.com/pgvector/pgvector (дата звернення: 10.05.2026).
5. ACID Properties in DBMS [Електронний ресурс]. Technology Coder, 2024. URL: https://www.technologycoder.info/uk/acid-properties-in-dbms (дата звернення: 10.05.2026).
6. Vector Similarity Search with PostgreSQL's pgvector: A Deep Dive [Електронний ресурс]. Severalnines Blog, 2024. URL: https://severalnines.com/blog/vector-similarity-search-with-postgresqls-pgvector-a-deep-dive (дата звернення: 10.05.2026).
7. JSON Types. PostgreSQL 16 Documentation [Електронний ресурс]. The PostgreSQL Global Development Group, 2024. URL: https://www.postgresql.org/docs/current/datatype-json.html (дата звернення: 10.05.2026).
8. Database Normalization: 1NF, 2NF, 3NF, BCNF with Examples [Електронний ресурс]. Guru99, 2024. URL: https://www.guru99.com/uk/database-normalization.html (дата звернення: 10.05.2026).

9. Python 3.11 Documentation [Електронний ресурс]. Python Software Foundation, 2024. URL: https://docs.python.org/3.11/ (дата звернення: 10.05.2026).
10. Flask Documentation 3.0.x [Електронний ресурс]. Pallets Projects, 2024. URL: https://flask.palletsprojects.com/en/3.0.x/ (дата звернення: 10.05.2026).
11. psycopg2: PostgreSQL adapter for Python [Електронний ресурс]. URL: https://www.psycopg.org/docs/ (дата звернення: 10.05.2026).
12. Sentence-Transformers: Multilingual Sentence and Image Embeddings [Електронний ресурс]. SBERT, 2024. URL: https://www.sbert.net/ (дата звернення: 10.05.2026).
13. all-MiniLM-L6-v2 Model Card [Електронний ресурс]. Hugging Face, 2024. URL: https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2 (дата звернення: 10.05.2026).
14. Groq API Documentation [Електронний ресурс]. Groq Inc., 2024. URL: https://console.groq.com/docs/openai (дата звернення: 10.05.2026).
15. Llama 3.3: The Most Capable Openly Available LLM [Електронний ресурс]. Meta AI, 2024. URL: https://ai.meta.com/blog/llama-3/ (дата звернення: 10.05.2026).
16. JavaScript. MDN Web Docs [Електронний ресурс]. Mozilla Foundation, 2024. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript (дата звернення: 10.05.2026).
17. Prompt Engineering Guide [Електронний ресурс]. DAIR.AI, 2024. URL: https://www.promptingguide.ai/ (дата звернення: 10.05.2026).

Додаток А

Лістинг модуля app.py

```
from __future__ import annotations

import os
import uuid

from flask import Flask, jsonify, render_template, request, send_from_directory
from flask_cors import CORS

from config import settings
from db import (
    chat_exists,
    create_chat,
    delete_chat,
    get_chats,
    get_messages,
    init_tables,
    rename_chat,
    save_message,
)
from rag import generate_html, suggest_image_candidates

app = Flask(__name__)
CORS(app)

os.makedirs(settings.preview_dir, exist_ok=True)
init_tables()


def get_request_user_id() -> str:
    user_id = (request.headers.get("X-User-Id") or "").strip()
    if not user_id:
        raise ValueError("Missing X-User-Id header.")
    try:
        uuid.UUID(user_id)
    except ValueError as exc:
        raise ValueError("Invalid X-User-Id header.") from exc
    return user_id


@app.errorhandler(ValueError)
def handle_value_error(exc: ValueError):
    return jsonify({"error": str(exc)}), 400


@app.route("/")
def index():
    return render_template("index.html")


@app.route("/preview/<path:filename>")
def serve_preview(filename: str):
    return send_from_directory(settings.preview_dir, filename)


@app.route("/api/chats", methods=["GET"])
def api_get_chats():
    user_id = get_request_user_id()
```

```
    return jsonify(get_chats(user_id))


@app.route("/api/chats", methods=["POST"])
def api_create_chat():
    user_id = get_request_user_id()
    data = request.get_json(silent=True) or {}
    title = (data.get("title") or "New Chat").strip() or "New Chat"
    return jsonify(create_chat(title, user_id))


@app.route("/api/chats/<int:chat_id>", methods=["DELETE"])
def api_delete_chat(chat_id: int):
    user_id = get_request_user_id()
    if not delete_chat(chat_id, user_id):
        return jsonify({"error": "Chat not found."}), 404
    return jsonify({"ok": True})


@app.route("/api/chats/<int:chat_id>", methods=["PATCH"])
def api_rename_chat(chat_id: int):
    user_id = get_request_user_id()
    data = request.get_json(silent=True) or {}
    title = (data.get("title") or "Chat").strip() or "Chat"
    if not rename_chat(chat_id, title, user_id):
        return jsonify({"error": "Chat not found."}), 404
    return jsonify({"ok": True})


@app.route("/api/chats/<int:chat_id>/messages", methods=["GET"])
def api_get_messages(chat_id: int):
    user_id = get_request_user_id()
    if not chat_exists(chat_id, user_id):
        return jsonify({"error": "Chat not found."}), 404
    return jsonify(get_messages(chat_id, user_id))


@app.route("/api/image-options", methods=["POST"])
def api_image_options():
    get_request_user_id()
    data = request.get_json(silent=True) or {}
    prompt = (data.get("prompt") or "").strip()
    if not prompt:
        return jsonify({"error": "Prompt is required."}), 400

    try:
        result = suggest_image_candidates(prompt)
    except Exception as exc:
        return jsonify({"error": str(exc)}), 500

    return jsonify(result)


@app.route("/api/chats/<int:chat_id>/generate", methods=["POST"])
def api_generate(chat_id: int):
    user_id = get_request_user_id()
    if not chat_exists(chat_id, user_id):
        return jsonify({"error": "Chat not found."}), 404

    data = request.get_json(silent=True) or {}
    user_prompt = (data.get("prompt") or "").strip()
    selected_image_url = (data.get("selected_image_url") or "").strip() or None
```

```
    if not user_prompt:
        return jsonify({"error": "Prompt is required."}), 400

    try:
        result = generate_html(user_prompt=user_prompt, selected_image_url=selected_image_url)
    except Exception as exc:
        return jsonify({"error": str(exc)}), 500

    filename = f"{uuid.uuid4().hex}.html"
    file_path = os.path.join(settings.preview_dir, filename)
    with open(file_path, "w", encoding="utf-8") as file_obj:
        file_obj.write(result["html"])

    preview_url = f"/preview/{filename}"
    saved_message = save_message(
        chat_id=chat_id,
        user_prompt=user_prompt,
        rag_prompt=result["rag_prompt"],
        generated_html=result["html"],
        preview_path=preview_url,
        selected_image_url=result["selected_image"]["image_url"] if result["selected_image"] else None,
    )

    saved_message["image_candidates"] = result["image_candidates"]
    saved_message["selected_image"] = result["selected_image"]
    return jsonify(saved_message)


if __name__ == "__main__":
    app.run(host=settings.app_host, port=settings.app_port, debug=settings.debug
```

Додаток Б

Лістинг модуля db.py

```
from __future__ import annotations

from contextlib import contextmanager
from datetime import date, datetime
from typing import Any, Dict, Iterator, List, Optional

import psycopg2
import psycopg2.extras

from config import settings


def _serialize_value(value: Any) -> Any:
    if isinstance(value, (datetime, date)):
        return value.isoformat()
    return value


def _serialize_row(row: Dict[str, Any]) -> Dict[str, Any]:
    return {key: _serialize_value(value) for key, value in row.items()}


def _to_vector_literal(values: List[float]) -> str:
    return "[" + ",".join(f"{float(value):.12f}" for value in values) + "]"


@contextmanager
def get_conn() -> Iterator[psycopg2.extensions.connection]:
    conn = psycopg2.connect(
        host=settings.pg_host,
        port=settings.pg_port,
        dbname=settings.pg_database,
        user=settings.pg_user,
        password=settings.pg_password,
    )
    conn.set_client_encoding("UTF8")
    try:
        yield conn
    finally:
        conn.close()


def init_tables() -> None:
    with get_conn() as conn:
        with conn.cursor() as cur:
            cur.execute(
                """
                CREATE TABLE IF NOT EXISTS chats (
                    id SERIAL PRIMARY KEY,
                    title TEXT NOT NULL,
                    user_id TEXT,
                    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
                );
                """
            )
            cur.execute(
                """
```

```
            CREATE TABLE IF NOT EXISTS messages (
                id SERIAL PRIMARY KEY,
                chat_id INT REFERENCES chats(id) ON DELETE CASCADE,
                user_prompt TEXT NOT NULL,
                rag_prompt TEXT NOT NULL,
                generated_html TEXT NOT NULL,
                preview_path TEXT NOT NULL,
                created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
            );
            """
        )
        cur.execute(
            """
            ALTER TABLE chats
            ADD COLUMN IF NOT EXISTS user_id TEXT;
            """
        )
        cur.execute(
            """
            ALTER TABLE messages
            ADD COLUMN IF NOT EXISTS selected_image_url TEXT;
            """
        )
        conn.commit()


def create_chat(title: str, user_id: str) -> Dict[str, Any]:
    with get_conn() as conn:
        with conn.cursor(cursor_factory=psycopg2.extras.RealDictCursor) as cur:
            cur.execute(
                """
                INSERT INTO chats (title, user_id)
                VALUES (%s, %s)
                RETURNING id, title, user_id, created_at;
                """,
                (title, user_id),
            )
            row = cur.fetchone()
            conn.commit()
            return _serialize_row(dict(row))


def get_chats(user_id: str) -> List[Dict[str, Any]]:
    with get_conn() as conn:
        with conn.cursor(cursor_factory=psycopg2.extras.RealDictCursor) as cur:
            cur.execute(
                """
                SELECT
                    c.id,
                    c.title,
                    c.user_id,
                    c.created_at,
                    COUNT(m.id) AS msg_count
                FROM chats c
                LEFT JOIN messages m ON m.chat_id = c.id
                WHERE c.user_id = %s
                GROUP BY c.id
                ORDER BY c.created_at DESC;
                """,
                (user_id,),
            )
            return [_serialize_row(dict(row)) for row in cur.fetchall()]
```

```
def chat_exists(chat_id: int, user_id: str) -> bool:
    with get_conn() as conn:
        with conn.cursor() as cur:
            cur.execute(
                """
                SELECT 1
                FROM chats
                WHERE id = %s AND user_id = %s;
                """,
                (chat_id, user_id),
            )
            return cur.fetchone() is not None


def get_messages(chat_id: int, user_id: str) -> List[Dict[str, Any]]:
    with get_conn() as conn:
        with conn.cursor(cursor_factory=psycopg2.extras.RealDictCursor) as cur:
            cur.execute(
                """
                SELECT
                    m.id,
                    m.user_prompt,
                    m.rag_prompt,
                    m.generated_html,
                    m.preview_path,
                    m.created_at,
                    m.selected_image_url
                FROM messages m
                JOIN chats c ON c.id = m.chat_id
                WHERE m.chat_id = %s AND c.user_id = %s
                ORDER BY m.created_at ASC;
                """,
                (chat_id, user_id),
            )
            return [_serialize_row(dict(row)) for row in cur.fetchall()]


def save_message(
    chat_id: int,
    user_prompt: str,
    rag_prompt: str,
    generated_html: str,
    preview_path: str,
    selected_image_url: Optional[str],
) -> Dict[str, Any]:
    with get_conn() as conn:
        with conn.cursor(cursor_factory=psycopg2.extras.RealDictCursor) as cur:
            cur.execute(
                """
                INSERT INTO messages (
                    chat_id,
                    user_prompt,
                    rag_prompt,
                    generated_html,
                    preview_path,
                    selected_image_url
                )
                VALUES (%s, %s, %s, %s, %s, %s)
                RETURNING
                    id,
                    user_prompt,
                    rag_prompt,
```

```
                generated_html,
                preview_path,
                created_at,
                selected_image_url;
            """,
            (
                chat_id,
                user_prompt,
                rag_prompt,
                generated_html,
                preview_path,
                selected_image_url,
            ),
        )
        row = cur.fetchone()
        conn.commit()
        return _serialize_row(dict(row))


def delete_chat(chat_id: int, user_id: str) -> bool:
    with get_conn() as conn:
        with conn.cursor() as cur:
            cur.execute(
                "DELETE FROM chats WHERE id = %s AND user_id = %s;",
                (chat_id, user_id),
            )
            deleted = cur.rowcount > 0
            conn.commit()
            return deleted


def rename_chat(chat_id: int, title: str, user_id: str) -> bool:
    with get_conn() as conn:
        with conn.cursor() as cur:
            cur.execute(
                "UPDATE chats SET title = %s WHERE id = %s AND user_id = %s;",
                (title, chat_id, user_id),
            )
            updated = cur.rowcount > 0
            conn.commit()
            return updated


def fetch_image_by_url(image_url: str) -> Optional[Dict[str, Any]]:
    with get_conn() as conn:
        with conn.cursor(cursor_factory=psycopg2.extras.RealDictCursor) as cur:
            cur.execute(
                """
                SELECT
                    id,
                    image_url,
                    description,
                    width,
                    height
                FROM ui_images
                WHERE image_url = %s
                LIMIT 1;
                """,
                (image_url,),
            )
            row = cur.fetchone()
            return dict(row) if row else None
```

```
def fetch_rag_context(query_embedding: List[float], limit_images: int = 3) -> Dict[str, Any]:
    vector_literal = _to_vector_literal(query_embedding)
    with get_conn() as conn:
        with conn.cursor(cursor_factory=psycopg2.extras.RealDictCursor) as cur:
            cur.execute(
                """
                SELECT
                    page_type,
                    description,
                    ui_elements,
                    embedding <-> %s::vector AS distance
                FROM ui_page_structure
                ORDER BY embedding <-> %s::vector
                LIMIT 1;
                """,
                (vector_literal, vector_literal),
            )
            page_row = cur.fetchone()
            if not page_row:
                raise RuntimeError("Table ui_page_structure is empty.")
            page = dict(page_row)

            cur.execute(
                """
                SELECT
                    domain,
                    tone,
                    style_description,
                    embedding <-> %s::vector AS distance
                FROM ui_brand_style
                ORDER BY embedding <-> %s::vector
                LIMIT 1;
                """,
                (vector_literal, vector_literal),
            )
            brand_row = cur.fetchone()
            if not brand_row:
                raise RuntimeError("Table ui_brand_style is empty.")
            brand = dict(brand_row)

            cur.execute(
                """
                SELECT
                    id,
                    image_url,
                    description,
                    width,
                    height,
                    embedding <-> %s::vector AS distance
                FROM ui_images
                WHERE image_url IS NOT NULL
                ORDER BY embedding <-> %s::vector
                LIMIT %s;
                """,
                (vector_literal, vector_literal, limit_images),
            )
            images = [dict(row) for row in cur.fetchall()]

    return {"page": page, "brand": brand, "images": images}
```

Додаток В

Лістинг скрипту seed_knowledge_base.py

```
from __future__ import annotations

import json
import psycopg2
from sentence_transformers import SentenceTransformer

from config import settings


def get_conn():
    return psycopg2.connect(
        host=settings.pg_host,
        port=settings.pg_port,
        dbname=settings.pg_database,
        user=settings.pg_user,
        password=settings.pg_password,
    )


def to_vector(model: SentenceTransformer, text: str) -> str:
    vec = model.encode(text).tolist()
    return "[" + ",".join(f"{v:.12f}" for v in vec) + "]"


PAGE_STRUCTURES = [
    {
        "page_type": "login",
        "description": (
            "Authentication page for user login. Contains email and password "
            "inputs, remember me checkbox, social login options and link to "
            "registration. Clean centered layout with brand logo at the top."
        ),
        "ui_elements": json.dumps({
            "inputs": ["email", "password"],
            "buttons": ["login", "google_login"],
            "links": ["forgot_password", "register"],
            "extras": ["remember_me_checkbox", "brand_logo"],
        }),
    },
    {
        "page_type": "register",
        "description": (
            "User registration sign-up form with full name, email, password "
            "and confirm password fields. Includes password strength indicator "
            "and terms acceptance checkbox."
        ),
        "ui_elements": json.dumps({
            "inputs": ["name", "email", "password", "confirm_password"],
            "buttons": ["submit"],
            "extras": ["password_strength_bar", "terms_checkbox"],
        }),
    },
    {
        "page_type": "forgot_password",
        "description": (
```

```
            "Reset password request page. User enters email to receive reset "
            "link. Minimal layout with instruction text and back to login link."
        ),
        "ui_elements": json.dumps({
            "inputs": ["email"],
            "buttons": ["submit"],
            "links": ["back_to_login"],
        }),
    },
    {
        "page_type": "dashboard",
        "description": (
            "Main user dashboard with analytics summary cards, recent activity "
            "feed, quick action buttons and navigation sidebar. Data-focused "
            "layout with charts and key metrics."
        ),
        "ui_elements": json.dumps({
            "sections": ["stats_cards", "recent_activity", "quick_actions"],
            "charts": ["line_chart", "bar_chart"],
            "navigation": ["sidebar", "top_navbar"],
        }),
    },
]


BRAND_STYLES = [
    {
        "domain": "ai",
        "tone": "minimal futuristic",
        "style_description": (
            "Soft gradients from deep purple to indigo, glassmorphism cards "
            "with frosted glass effect. Dark background #0a0a1a with accent "
            "colors #6366f1 and #8b5cf6. Inter or Space Grotesk font."
        ),
    },
    {
        "domain": "gaming",
        "tone": "immersive",
        "style_description": (
            "Dark UI with deep blacks #0d0d0d. Neon glowing accents in cyan "
            "#00d4ff and magenta #ff00ff. Orbitron font for headings."
        ),
    },
]


IMAGES = [
    {
        "image_url": "https://images.unsplash.com/photo-1587300003388-59208cc962cb?w=1200",
        "description": "Golden retriever in veterinary clinic, warm lighting",
        "width": 1200,
        "height": 800,
    },
]


def seed_table(conn, model, table, data, text_fn, insert_sql):
    with conn.cursor() as cur:
        cur.execute(f"SELECT COUNT(*) FROM {table};")
        if cur.fetchone()[0] > 0:
            print(f"{table} already seeded, skipping.")
            return
```

```
        for item in data:
            vec = to_vector(model, text_fn(item))
            cur.execute(insert_sql, vec)

        conn.commit()
        print(f"Seeded {len(data)} rows into {table}.")


def main():
    print("Loading embedding model...")
    model = SentenceTransformer(settings.embedding_model_name)

    conn = get_conn()

    try:
        # UI page structures
        with conn.cursor() as cur:
            cur.execute("SELECT COUNT(*) FROM ui_page_structure;")
            if cur.fetchone()[0] == 0:
                for item in PAGE_STRUCTURES:
                    text = f"{item['page_type']} {item['description']}"
                    vec = to_vector(model, text)

                    cur.execute(
                        """
                        INSERT INTO ui_page_structure
                        (page_type, description, ui_elements, embedding)
                        VALUES (%s, %s, %s, %s::vector);
                        """,
                        (
                            item["page_type"],
                            item["description"],
                            item["ui_elements"],
                            vec,
                        ),
                    )
                conn.commit()
                print(f"Seeded {len(PAGE_STRUCTURES)} page structures.")

        # Brand styles
        with conn.cursor() as cur:
            cur.execute("SELECT COUNT(*) FROM ui_brand_style;")
            if cur.fetchone()[0] == 0:
                for item in BRAND_STYLES:
                    text = f"{item['domain']} {item['tone']} {item['style_description']}"
                    vec = to_vector(model, text)

                    cur.execute(
                        """
                        INSERT INTO ui_brand_style
                        (domain, tone, style_description, embedding)
                        VALUES (%s, %s, %s, %s::vector);
                        """,
                        (
                            item["domain"],
                            item["tone"],
                            item["style_description"],
                            vec,
                        ),
                    )
                conn.commit()
                print(f"Seeded {len(BRAND_STYLES)} brand styles.")
```

```
        # Images
        with conn.cursor() as cur:
            cur.execute("SELECT COUNT(*) FROM ui_images;")
            if cur.fetchone()[0] == 0:
                for item in IMAGES:
                    vec = to_vector(model, item["description"])

                    cur.execute(
                        """
                        INSERT INTO ui_images
                        (image_url, description, width, height, embedding)
                        VALUES (%s, %s, %s, %s, %s::vector);
                        """,
                        (
                            item["image_url"],
                            item["description"],
                            item["width"],
                            item["height"],
                            vec,
                        ),
                    )
                conn.commit()
                print(f"Seeded {len(IMAGES)} images.")

        print("Done. Knowledge base is ready.")

    finally:
        conn.close()


if __name__ == "__main__":
    main()
```

Додаток Г

Таблиця Г.1 — Ендпоінти REST API системи

Метод	**URL**	**Опис**
GET	/api/chats	Список чатів користувача
POST	/api/chats	Створити чат
PATCH	/api/chats/<id>	Перейменувати чат
DELETE	/api/chats/<id>	Видалити чат
GET	/api/chats/<id>/messages	Повідомлення чату
POST	/api/chats/<id>/generate	Генерація HTML
POST	/api/image-options	Кандидати зображень
GET	/preview/<filename>	Файл попереднього перегляду
GET	/	Клієнтський SPA

Додаток Д

Таблиця Д.1 — Результати тестування семантичного ретривера

№	Запит	Очік. page_type	Знайдено	Відстань	Очік. домен	Знайдено	Відстань	Точність
1	AI assistant login glassmorphism futuristic	login	login	1.015	ai	ai	0.984	✓
2	gaming registration dark neon glowing UI	register	register	1.231	gaming	gaming	0.894	✓
3	travel app forgot password bright colors	forgot_password	forgot_password	0.916	travel	travel	1.127	✓
4	CRM dashboard sales analytics sidebar	dashboard	analytics	0.999	crm	finance	1.072	~
5	enterprise admin panel user management	admin_panel	admin_panel	0.994	enterprise	enterprise	1.016	✓
6	university order confirmation formal clean	order_success	order_success	1.132	university	university	0.952	✓

Додаток Е

Лістинг модуля rag.py (фрагмент основних функцій)

```
from __future__ import annotations

import json
import re
from functools import lru_cache
from typing import Any, Dict, List, Optional

from openai import OpenAI
from sentence_transformers import SentenceTransformer

from config import settings
from db import fetch_image_by_url, fetch_rag_context


FONT_PAIRS = {
    "saas": ("Sora", "Inter"),
    "finance": ("Plus Jakarta Sans", "Inter"),
    "ecommerce": ("Space Grotesk", "Manrope"),
    "portfolio": ("Syne", "Manrope"),
    "agency": ("Space Grotesk", "Inter"),
    "healthcare": ("Outfit", "Inter"),
    "default": ("Space Grotesk", "Manrope"),
}


def _extract_html_document(text: str) -> str:
    cleaned = text.strip()
    if cleaned.startswith("```"):
        lines = cleaned.splitlines()
        lines = [line for line in lines if not line.strip().startswith("```")]
        cleaned = "\n".join(lines).strip()

    match = re.search(r"<!DOCTYPE html>.*</html>", cleaned, flags=re.IGNORECASE | re.DOTALL)
    if match:
        return match.group(0).strip()

    match = re.search(r"<html.*</html>", cleaned, flags=re.IGNORECASE | re.DOTALL)
    if match:
        return "<!DOCTYPE html>\n" + match.group(0).strip()

    return cleaned


def _ensure_google_fonts(html: str, heading_font: str, body_font: str) -> str:
    if "fonts.googleapis.com" in html:
        return html

    font_query = (
        heading_font.replace(" ", "+")
        + ":wght@400;500;600;700;800&family="
        + body_font.replace(" ", "+")
        + ":wght@400;500;600;700;800"
    )
    font_links = (
        '<link rel="preconnect" href="https://fonts.googleapis.com">\n'
        '<link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>\n'
        f'<link href="https://fonts.googleapis.com/css2?family={font_query}&display=swap" rel="stylesheet">'
```

```
    )

    if "</head>" in html:
        return html.replace("</head>", font_links + "\n</head>", 1)
    return html


def _ensure_viewport(html: str) -> str:
    if 'name="viewport"' in html.lower():
        return html
    if "</head>" in html:
        return html.replace(
            "</head>",
            '<meta name="viewport" content="width=device-width, initial-scale=1.0">\n</head>',
            1,
        )
    return html


def _ensure_font_css(html: str, heading_font: str, body_font: str) -> str:
    css = (
        ":root{"
        f"--font-heading:'{heading_font}',sans-serif;"
        f"--font-body:'{body_font}',sans-serif;"
        "}"
        "body{font-family:var(--font-body);}"
        "img{max-width:100%;display:block;}"
        "h1,h2,h3,h4,h5,h6,.display-title,.hero-title{font-family:var(--font-heading);}"
    )

    if "<style>" in html:
        return html.replace("<style>", "<style>\n" + css + "\n", 1)
    if "</head>" in html:
        return html.replace("</head>", f"<style>{css}</style>\n</head>", 1)
    return html


def _normalize_generated_html(html: str, heading_font: str, body_font: str) -> str:
    html = _extract_html_document(html)
    html = _ensure_viewport(html)
    html = _ensure_google_fonts(html, heading_font, body_font)
    html = _ensure_font_css(html, heading_font, body_font)

    if "<!DOCTYPE html>" not in html[:40]:
        html = "<!DOCTYPE html>\n" + html
    return html.strip()


def _guess_image_orientation(width: Optional[int], height: Optional[int]) -> str:
    if not width or not height:
        return "unknown"
    if width > height:
        return "landscape"
    if height > width:
        return "portrait"
    return "square"


def _extract_keywords(prompt: str) -> List[str]:
    tokens = re.findall(r"\b[\w-]{4,}\b", prompt.lower(), flags=re.UNICODE)
    deduped: List[str] = []
    for token in tokens:
        if token not in deduped:
```

```
            deduped.append(token)
    return deduped[:8]


def _score_image_for_prompt(image: Dict[str, Any], prompt: str) -> float:
    prompt_lower = prompt.lower()
    description = (image.get("description") or "").lower()
    width = image.get("width") or 0
    height = image.get("height") or 0
    distance = float(image.get("distance") or 0)

    score = max(0.0, 10.0 - distance)
    keywords = _extract_keywords(prompt)
    for keyword in keywords:
        if keyword in description:
            score += 1.2

    if any(word in prompt_lower for word in ("hero", "landing", "header", "banner", "main")) and width > height:
        score += 2.0
    if any(word in prompt_lower for word in ("mobile", "phone", "portrait", "story", "vertical")) and height > width:
        score += 2.0
    if width >= 1200:
        score += 0.6
    if height >= 800:
        score += 0.4

    return round(score, 3)


def _enrich_image_metadata(image: Dict[str, Any], prompt: str) -> Dict[str, Any]:
    item = dict(image)
    item["orientation"] = _guess_image_orientation(item.get("width"), item.get("height"))
    item["fit_score"] = item.get("fit_score", _score_image_for_prompt(item, prompt))
    return item


def _prepare_image_candidates(images: List[Dict[str, Any]], prompt: str) -> List[Dict[str, Any]]:
    scored = [_enrich_image_metadata(image, prompt) for image in images]
    scored.sort(key=lambda item: (-float(item["fit_score"]), float(item.get("distance") or 9999)))
    return scored[:3]


def _merge_selected_image(
    candidates: List[Dict[str, Any]],
    selected_image_url: Optional[str],
    user_prompt: str,
) -> List[Dict[str, Any]]:
    if not selected_image_url:
        return candidates

    for item in candidates:
        if item["image_url"] == selected_image_url:
            return candidates

    selected_row = fetch_image_by_url(selected_image_url)
    if not selected_row:
        return candidates

    selected_item = _enrich_image_metadata(selected_row, user_prompt)
    merged = [selected_item]
    for candidate in candidates:
        if candidate["image_url"] != selected_item["image_url"]:
            merged.append(candidate)
```

```
    return merged[:3]


def _choose_font_pair(brand: Dict[str, Any], user_prompt: str) -> Dict[str, str]:
    source = f"{brand.get('domain', '')} {brand.get('tone', '')} {user_prompt}".lower()
    for key, fonts in FONT_PAIRS.items():
        if key != "default" and key in source:
            return {"heading": fonts[0], "body": fonts[1]}
    default_fonts = FONT_PAIRS["default"]
    return {"heading": default_fonts[0], "body": default_fonts[1]}


def _build_layout_guidance(selected_image: Optional[Dict[str, Any]]) -> str:
    if not selected_image:
        return (
            "No image was selected. Build a visually strong page without relying on photography. "
            "Use typography, gradients, cards, stats, icons made with pure CSS or simple shapes, layered panels, and refined
spacing. "
            "Do not leave an empty hero hole where an image should have been."
        )

    orientation = selected_image.get("orientation")
    width = selected_image.get("width")
    height = selected_image.get("height")
    ratio = round((width / height), 3) if width and height else None

    if orientation == "portrait":
        return (
            f"The image is portrait with ratio about {ratio}. Build a split hero layout where text and CTA live in one column
"
            "and the image lives inside a framed vertical card. Do not crop the subject aggressively. Prefer object-fit:
contain "
            "or a card ratio close to the source. Never stretch a portrait image across the full screen width."
        )
    if orientation == "square":
        return (
            f"The image is square with ratio about {ratio}. Use a balanced composition with either a centered media block
or a two-column "
            "hero where the image sits in a square card. Preserve the square feeling and avoid turning it into a giant
background."
        )
    return (
        f"The image is landscape with ratio about {ratio}. Use a cinematic but restrained hero media panel. Let the image
breathe without clipping "
        "important details. Do not make it absurdly tall. Use overlays or separate text panels for readability."
    )


@lru_cache(maxsize=1)
def get_embedding_model() -> SentenceTransformer:
    return SentenceTransformer(settings.embedding_model_name)


def get_llm_client() -> OpenAI:
    client_kwargs: Dict[str, Any] = {"api_key": settings.llm_api_key}
    if settings.llm_base_url:
        client_kwargs["base_url"] = settings.llm_base_url
    elif settings.llm_provider.lower() == "groq":
        client_kwargs["base_url"] = "https://api.groq.com/openai/v1"

    return OpenAI(**client_kwargs)
```

```
def build_rag_prompt(
    user_prompt: str,
    page: Dict[str, Any],
    brand: Dict[str, Any],
    selected_image: Optional[Dict[str, Any]],
    alternatives: List[Dict[str, Any]],
) -> str:
    alternatives_payload = [
        {
            "image_url": image["image_url"],
            "description": image.get("description"),
            "width": image.get("width"),
            "height": image.get("height"),
            "orientation": image.get("orientation"),
            "fit_score": image.get("fit_score"),
        }
        for image in alternatives
    ]
    fonts = _choose_font_pair(brand, user_prompt)
    layout_guidance = _build_layout_guidance(selected_image)
    ratio_width = selected_image.get("width") if selected_image else 16
    ratio_height = selected_image.get("height") if selected_image else 9

    image_section = f"""
SELECTED IMAGE TO USE:
URL: {selected_image["image_url"]}
Description: {selected_image.get("description")}
Width: {selected_image.get("width")}
Height: {selected_image.get("height")}
Orientation: {selected_image.get("orientation")}
""".strip() if selected_image else """
SELECTED IMAGE TO USE:
No image was selected for this generation.
""".strip()

    image_rules = f"""
Image rules:
- Respect the real width and height of the selected image.
- Never distort the image.
- Use CSS that preserves the image ratio in a container shaped for that ratio.
- Use `aspect-ratio: {ratio_width} / {ratio_height}` on the main image wrapper when appropriate.
- Default to a contained card, framed panel, or restrained hero media block instead of making the image cover the entire screen.
- Prefer `object-fit: contain` for portrait and square imagery unless a matching card ratio makes `cover` safe.
- If using the image as a background, cap the section height and use overlays instead of stretching the media.
- If text overlays the image, ensure strong readability with scrims, gradients, or separate text panels.
- The selected image must actually appear in the HTML.
- Keep image sections elegant and bounded, not oversized and not stretched.
""".strip() if selected_image else """
Image rules:
- No image should be used in this page.
- Build the hero and sections without `<img>` tags or photo backgrounds.
- Replace image impact with typography, gradients, abstract shapes, glass cards, stat blocks, and layout rhythm.
- Keep the result rich and premium even without photography.
""".strip()

    implementation_hint = (
        f"- A good pattern is `.hero-media {{ aspect-ratio: {ratio_width} / {ratio_height}; overflow: hidden; border-radius: 28px; max-height: min(72vh, 760px); }}` with a nested `img`."
        if selected_image
        else "- A good pattern is a hero with layered gradient background, one strong heading, one support paragraph, stat cards, and a polished CTA cluster."
    )
```

```
    return f"""
You are an elite web designer and senior frontend engineer creating premium marketing-quality pages.

Your job is to produce a polished single-file HTML page that looks intentional, modern, and art directed.
The result must feel like a real designer made it, not a generic AI wireframe.

USER REQUEST:
{user_prompt}

RAG PAGE STRUCTURE:
Page type: {page["page_type"]}
Description: {page["description"]}
UI elements: {json.dumps(page["ui_elements"], ensure_ascii=False)}

RAG BRAND STYLE:
Domain: {brand["domain"]}
Tone: {brand["tone"]}
Style direction: {brand["style_description"]}

{image_section}

IMAGE USAGE GUIDANCE:
{layout_guidance}

TOP IMAGE OPTIONS FOR CONTEXT:
{json.dumps(alternatives_payload, ensure_ascii=False, indent=2)}

TYPOGRAPHY DIRECTION:
- Heading font: {fonts["heading"]}
- Body font: {fonts["body"]}
- Load fonts from Google Fonts in the HTML head.

Quality bar:
- The page must look visually premium and production-like.
- Do not make text and background same color.
- Avoid ugly gradients, random neon, and generic bootstrap-looking sections.
- Use a strong spacing system, restrained color palette, clear hierarchy, and one coherent visual concept.
- Design for desktop first, then make it adapt beautifully to mobile.
- The hero must be impressive and not cramped.
- Every section should have a reason to exist.

{image_rules}

Implementation rules:
- Return only valid complete HTML.
- Include <!DOCTYPE html>, <html>, <head>, <body>.
- Put all CSS inside a single <style> tag.
- Do not use external CSS libraries.
- You may use Google Fonts links.
- No markdown fences.
- No explanations before or after the HTML.
- Use semantic HTML tags.
- Make sure the page feels finished, not like a mockup.

Suggested structure:
- Hero
- Supporting proof or metrics
- One or two feature/content sections
- CTA/footer

Implementation hint:
{implementation_hint}
```

```
Output only the final HTML document.
""".strip()


def suggest_image_candidates(user_prompt: str) -> Dict[str, Any]:
    embedding = get_embedding_model().encode(user_prompt).tolist()
    context = fetch_rag_context(embedding, limit_images=9)
    candidates = _prepare_image_candidates(context["images"], user_prompt)

    return {
        "page": context["page"],
        "brand": context["brand"],
        "images": candidates,
    }


def generate_html(user_prompt: str, selected_image_url: Optional[str] = None) -> Dict[str, Any]:
    if not settings.llm_api_key:
        raise RuntimeError("LLM_API_KEY is not set. Add it to your .env file.")

    retrieval = suggest_image_candidates(user_prompt)
    image_candidates = _merge_selected_image(retrieval["images"], selected_image_url, user_prompt)

    selected_image = None
    if selected_image_url:
        selected_image = next(
            (image for image in image_candidates if image["image_url"] == selected_image_url),
            None,
        )

    rag_prompt = build_rag_prompt(
        user_prompt=user_prompt,
        page=retrieval["page"],
        brand=retrieval["brand"],
        selected_image=selected_image,
        alternatives=image_candidates,
    )

    system_prompt = """
You are a world-class frontend engineer and digital art director.
Produce one complete HTML file with embedded CSS.
Make the result elegant, responsive, and visually strong.
Do not output markdown or commentary.
""".strip()

    response = get_llm_client().chat.completions.create(
        model=settings.llm_model,
        temperature=0.35,
        messages=[
            {"role": "system", "content": system_prompt},
            {"role": "user", "content": rag_prompt},
        ],
    )
    raw_html = response.choices[0].message.content or ""
    fonts = _choose_font_pair(retrieval["brand"], user_prompt)
    generated_html = _normalize_generated_html(raw_html, fonts["heading"], fonts["body"])
    if not generated_html or "<body" not in generated_html.lower():
        raise RuntimeError("Model returned invalid HTML.")

    return {
        "html": generated_html,
        "rag_prompt": rag_prompt,
```

```
        "page": retrieval["page"],
        "brand": retrieval["brand"],
        "selected_image": selected_image,
        "image_candidates": image_candidates,
    }
```

```
def _clean_html(text: str) -> str: text = text.strip() if text.startswith(""): lines = [l for l in text.splitlines() if not l.strip().startswith("")] text = "\n".join(lines).strip() match = re.search( r"<!DOCTYPE html>.*</html>", text, flags=re.IGNORECASE | re.DOTALL) return match.group(0).strip() if match else text
```

Додаток Є

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

Декларація про академічну доброчесність та використання ШІ
ДЕКЛАРАЦІЯ ЗДОБУВАЧА

Я, Ухін Тимур Артемович, здобувач(ка) НУБіП України, усвідомлюючи відповідальність за порушення академічної доброчесності, цією заявою підтверджую, що:

1. Моя бакалаврська кваліфікаційна робота на тему «ІНТЕЛЕКТУАЛЬНИЙ СЕРВІС ГЕНЕРАЦІЇ ПРОГРАМНОГО КОДУ НА ОСНОВІ МЕТОДІВ МАШИННОГО НАВЧАННЯ» є результатом моєї особистої праці.
2. Усі запозичення з текстів інших авторів мають відповідні посилання згідно з чинними стандартами.
3. Щодо використання інструментів ШІ зазначаю, що:
    - [✓] інструменти ШІ ChatGPT, Claude, DeepL були використані для:
        - [✓] перекладу іншомовних джерел;
        - [✓] стилістичного редагування та перевірки граматики;
        - [✓] допомоги у написанні/відлагодженні програмного коду;

Я розумію, що надання недостовірної інформації може призвести до скасування результатів захисту та відрахування з числа здобувачів НУБіП України.

«___» _____ 2026 р. _____ **Тимур Ухін**
(підпис)