МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ ДЛЯ ПОШУКУ СХОЖИХ ВЕБСАЙТІВ ЗАСОБАМИ REACT ТА NODE.JS»**

Виконав: студент 2 курсу, групи 8.1214-іпз-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми інженерія програмного забезпечення
(назва освітньої програми)
М.С. Литвинов
(ініціали та прізвище)

Керівник доцент кафедри програмної інженерії, к.ф.-м.н. Кривохата А.Г.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент в.о. завідувача кафедри комп'ютерних наук, доцент, к.пед.н. Пшенична О.С.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Литвинову Микиті Сергійовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка інформаційної системи для пошуку схожих вебсайтів засобами React та Node.js

керівник роботи Кривохата Анастасія Григорівна, к.ф.-м.н.
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 28 » лютого 2025 року № 326-с

2. Строк подання студентом роботи 04.12.2025 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Аналіз предметної області.
2. Проєктування системи.
3. Розробка системи.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 07.03.2025 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	21.03.2025	
2.	Збір вихідних даних.	25.04.2025	
3.	Обробка методичних та теоретичних	20.06.2025	
	джерел.		
4.	Розробка першого та другого розділу.	01.09.2025	
5.	Розробка третього розділу.	24.11.2025	
6.	Оформлення та нормоконтроль	01.12.2025	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	18.12.2025	

Студент _____ М.С. Литвинов
(підпис) (ініціали та прізвище)

Керівник роботи _____ А.Г. Кривохата
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка інформаційної системи для пошуку схожих вебсайтів засобами React та Node.js»: 73 с., 27 рис., 11 табл., 20 джерел, 1 додаток.

ВЕБСКРЕЙПІНГ, СИСТЕМА ПОШУКУ, BEDROCK, JAVASCRIPT, LAMBDA, LLM, NODE.JS, REACT.

Об’єкт дослідження – процеси та методи пошуку схожих вебсайтів на основі аналізу їхніх характеристик (структури, контенту, технологій, метаданих тощо) в інформаційних системах.

Предмет дослідження – методи побудови та реалізації вебсистеми пошуку схожих сайтів із використанням JavaScript-стеку (React, Node.js), алгоритмів зіставлення та обробки вебконтенту.

Мета роботи: вивчення можливих підходів для класифікації вебсайтів за допомогою засобів штучного інтелекту, а також розробка ефективної системи на основі отриманих результатів.

Метод дослідження – аналіз і узагальнення літературних джерел з тематики веброзробки, пошукових алгоритмів, класифікації сайтів, порівняння вебресурсів за контентними, структурними та іншими характеристиками, методи моделювання, проєктування, конструювання та тестування системи.

Було розроблено систему, що дає можливість користувачам знаходити сайти, надавши лише характеристики бажаного вебресурсу. Цю систему можна використовувати як у повсякденному житті, так і у комерційній діяльності.

SUMMARY

Master’s qualifying paper “Development of an Information System for Similar Websites Searching Using React and Node.js”: 73 pages, 27 figures, 11 tables, 20 references, 1 supplement.

WEBSCRAPPING, SEARCH SYSTEM, BEDROCK, JAVASCRIPT, LAMBDA, LLM, NODE.JS, REACT.JS.

The object of the study is the processes and methods of searching for similar websites based on the analysis of their characteristics (structure, content, technologies, metadata, etc.) in information systems.

The subject of research is methods for building and implementing a web system for searching for similar sites using the JavaScript stack (React, Node.js) and algorithms for comparing and processing web content.

The aim of the study is to research possible approaches to classifying websites using artificial intelligence, as well as to develop an effective system based on the results obtained.

The methods of research are analysis and generalization of literary sources on the subject of web development, search algorithms, site classification, comparison of web resources by content, structural and other characteristics, methods of modeling, design, construction and testing of the system.

The system was developed that allows users to find sites by providing only the characteristics of the desired site. This system can be used both in everyday life and in commercial activities.

ЗМІСТ

Завдання на кваліфікаційну роботу……..2
Реферат 4
Summary 5
Вступ..... 9
1 Аналіз предметної області..... 11
1.1 Огляд проблеми знаходження схожих сайтів 11
1.2 Аналіз існуючих систем 13
1.3 Порівняння методів пошуку схожих сайтів 14
1.4 Формування вимог до проєкту 17
1.4.1 Загальна характеристика проєкту 17
1.4.2 Основні функції системи..... 17
1.4.3 Функціональні вимоги..... 18
1.4.4 Нефункціональні вимоги..... 19
1.4.5 Вимоги до технологій..... 19
1.4.6 Вимоги до інтерфейсу 19
1.5 Аналіз інструментів 20
1.5.1 Екосистема JavaScript..... 20
1.5.2 Node.js Runtime Environment 21
1.5.3 Бібліотека React..... 21
1.5.4 TypeScript..... 22
1.5.5 Tailwind CSS..... 23
1.5.6 PostgreSQL СУБД 23
1.5.7 Сервіси AWS 24
1.5.8 Додаткові інструменти 24
1.6 Висновки до розділу 1 24
2 Проєктування системи..... 26
2.1 Огляд архітектури системи 26

2.2 Проєктування бази даних 27
2.2.1 Ідентифікація сутностей та атрибутів 27
2.2.2 Нормалізація та обґрунтування зв’язків 30
2.2.3 Графічне зображення бази даних 32
2.2.4 Стратегія індексації 33
2.3 Дизайн бекенду 33
2.3.1 Архітектура API 33
2.3.2 Організація lambda функцій 34
2.4 Дизайн фронтенду 36
2.4.1 Архітектура компонентів 36
2.4.2 Дизайн менеджменту станів 37
2.5 Проєктування сайту 37
2.6 UI/UX дизайн 39
2.6.1 Система проєктування 39
2.6.2 Типографія 39
2.6.3 Component Specifications 40
2.6.4 Точки розриву 41
2.7 Проєктування інфраструктури AWS 41
2.7.1 Обґрунтування вибору сервісів 42
2.8 Висновки до розділу 2 44
3 Розробка системи 45
3.1 Структура проєкту 45
3.2 Розробка бекенду 46
3.2.1 Реалізація сервісу бази даних 46
3.2.2 Реалізація обробника сайтів 47
3.2.3 Реалізація пошуку по тегам 50
3.2.4 Проміжне програмне забезпечення автентифікації 51
3.2.5 Утиліти для відповідей 53
3.3 Розробка фронтенду 55
3.3.1 Точка входу в додаток 55

3.3.2 Реалізація сторінки пошуку 56
3.3.3 Реалізації візуальних компонентів 59
3.4 Скріншоти інтерфейсу користувача 60
3.5 Висновки до розділу 3 63
Висновки 64
Перелік посилань 66
Додаток А Лістинг коду 68

ВСТУП

З ростом впливу інтернету на повсякденне життя людини, зростає кількість доступних сайтів, що у свою чергу зменшує відкритість окремо взятих сайтів до нових користувачів та засмічує їх увагу. Також, пошукові системи, такі як Google, Bing, Yahoo, все більше змінюють вектор роботи власних пошукових алгоритмів з надання найбільш точного контенту, на рекламування сайтів на базі специфічних запитів. Тож розробка нових методів пошуку сайтів, що базуватиметься не на обробці запиту користувача, а на тому, що користувач хоче отримати від сайту, може змінити те, як люди відносяться до пошуку контенту в мережі, зробить пошук більш персоналізованим. Отже така розробка є дійсно актуальною.

Враховуючи розмірність роботи, доцільно буде мінімізувати стек розробки, і використати найбільш відомі інструменти з найбільшим ком'юніті розробників. Саме тому було вирішено обмежити стек мовою JavaScript, а саме фреймворком React для розробки фронтенду, та Node – для бекенду.

Метою даної роботи є створення ефективної системи для пошуку подібних сайтів, базуючись на характеристиках сайтів, системи класифікації сайтів, використовуючи фреймворк React та Node, та провести аналіз можливості реалізації такої системи. Для досягнення цієї мети потрібно:

- проаналізувати вимоги, предметну область, проєкти конкурентів;
- спроєктувати структуру даних для зберігання (схему бази даних) та архітектуру системи;
- розробити систему;
- провести тестування системи, разом з тестовим розгортанням системи.

Об'єктами дослідження є процеси та методи пошуку схожих вебсайтів на основі аналізу їхніх характеристик та аналізу вебсторінок, а також процес розробки системи на основі цих методів.

Предметами дослідження є методи побудови та реалізації системи пошуку схожих сайтів із використанням JavaScript, алгоритмів зіставлення та обробки вебконтенту, взаємодія між JavaScript та базами даних.

Під час розробки було створено хмарну систему, що виконує усі поставлені задачі.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Огляд проблеми знаходження схожих сайтів

Пошук схожих вебсайтів – це завдання ідентифікації вебресурсів, що за змістом або призначенням наближаються до заданого. Історично подібність сайтів інтерпретувалася досить вузько, наприклад у ранні роки інтернету використовувалися вручну складені каталоги, як DMOZ чи Yahoo Directory, де ресурси групували за тематичними категоріями. Поява вебпошукачів (Google, Yahoo) та SEO-інструментів розширила підхід: тепер на схожість зважали як текстові характеристики, так і поведінкові метрики (як приклад, подібність аудиторій). Наприклад, Google пропонував оператор «related:» – запит «related:vogue.com» повертав сайти з близькою тематикою – проте відомо, що цей метод спрацьовує лише для дуже популярних або вузькопрофільних сайтів, і часто не знаходить результатів для загальних сайтів [1]. Через свою ненадійність, цей оператор був вилучений у 2023 році [2].

У сучасних дослідженнях проблема знаходження схожих сайтів сприймається комплексно. З одного боку, схожість може визначатися семантикою контенту – тобто спільністю тематики, ключових понять і використаних термінів. Проте навіть сайт з однаковою тематикою може використовувати різну лексику або структуру викладу, наприклад: два туристичні блоги можуть описувати подібні теми, але перший описує «готель» і «подорож», а інший – «хостел» та «тур». Методи по типу TF-IDF (term frequency–inverse document frequency) мають такі обмеження, оскільки вони шукають точні збіги слів. Аналіз текстів показує, що TF-IDF дає змогу знайти повні або часткові лексичні збіги, але не справляється з синонімами чи з контекстом, який потребує розуміння семантики, бо кожне слово оцінюється окремо, не враховуючи їх контекст [3]. Альтернативою є семантичний пошук, який використовує векторні представлення тексту (word embeddings). Мовні

моделі зберігають документи у вигляді векторів, які фіксують та представляють семантику, а також різні зв’язки між словами. Наприклад, сучасні алгоритми, по типу BERT перетворюють всю сторінку на семантичний вектор, придатний до порівняння через косинус подібності (Cosine Similarity) [3]. Семантичний підхід дозволяє виявляти схожі за змістом тексти навіть при різному формулюванні, але вимагає значних ресурсів та складних моделей. Однак, цей метод також має свої обмеження. У цьому випадку результати, виражені цими мовними моделями, є складними та не відповідають людському мисленню. Це ускладнює їх інтерпретацію. Крім того, семантичний пошук вимагає більше часу та обчислювальної потужності.

З іншого боку, схожість сайтів може визначатися їх структурою. Наприклад, дві торгівельні платформи можуть відрізнятися текстом і стилем, але мати майже однакову навігацію, розташування товарів і інформаційну архітектуру. У контексті протидії клонуванню сайтів (наприклад, у фішингових атаках), кібершахраї часто використовують однакові шаблони HTML або дуже подібний макет, навіть якщо трохи змінюють текст, щоб уникнути виявлення. Так, автори дослідження [4] відзначають, що навіть якщо вебсайти виглядають по-різному, вони часто мають спільні риси в тексті сторінок або структурі HTML. Це означає, що при пошуку клонів або суміжних сервісів слід аналізувати не тільки зміст, але й технічні ознаки (структуру HTML, меню, зображення). Загалом, проблема полягає у поєднанні всіх цих вимірів – семантичного і структурного. Методи пошуку схожості мають бути адаптивними та комплексно обробляти багато ознак ресурсу.

Зокрема, у SEO та маркетингу існує уявлення про «схожі сайти» як про конкурентів на ринку. Наприклад, сервіс Similarweb у своєму модулі «Similar Sites» формулює схожість як сукупність контенту, перетину аудиторії, використаних ключових слів і каналів трафіку. Такий підхід виходить за рамки чисто текстової подібності – він звертає увагу, наскільки аудиторії двох сайтів перетинаються та хто з ким обмінюється відвідувачами [5]. Отже, у загальному вигляді проблема пошуку схожих сайтів охоплює історичне переміщення від

каталогів і простих ключових пошуків до сучасних алгоритмів з глибоким аналізом тексту, структурних характеристик та аналітики трафіку.

1.2 Аналіз існуючих систем

Сьогодні існує низка комерційних та безкоштовних сервісів для пошуку схожих сайтів. Наприклад, сайт і Chrome-додаток SimilarSites позиціонує себе як «система спорідненності» вебсайтів: воно відображає список сайтів, пов’язаних із поточним, на основі колективної поведінки користувачів [7]. Творці стверджують, що технологія, що лежить в основі SimilarSites розроблена, щоб показати, що «люди, які відвідують цей сайт, також люблять відвідувати ці інші пов’язані сайти». Тобто SimilarSites збирає знеособлені дані про перегляди з «спільноти» користувачів розширення та використовує їх для рекомендацій. Переваги такого підходу – висока точність при великих обсягах даних і здатність виявляти неочевидні зв’язки. Проте це обмеженою анонімністю, залежністю від активності користувачів і ризиком зміщення рекомендацій, а також обмеженим набором сайтів, які є у базі даних.

Іншим прикладом є Google запит «related:». Хоч він і був вилучений, його вклад теж варто обговорити. Цей інструмент дозволяв обмежити пошук сайтами на кшталт заданого. Запит «related:medium.com» видавав вебресурси, пов’язані з сайтом Medium. Наслідком є те, що «related:» діє лише для дуже популярних або специфічних сайтів. Для сайтів з дуже широкою вибіркою тем, навіть відомих, як time.com, він часто не повертав нічого [1]. Перевага подібного підходу – відсутність потреби у додаткових інструментах, але мінусом є ненадійність і неявність критеріїв схожості.

Серед SEO-інструментів можна згадати Ahrefs та SpyFu. Ahrefs не має окремої функції «схожі сайти», але у розділі «competing domains» надається перелік ресурсів, що конкурують за ті ж ключові слова чи за відвідувачами. Наприклад, Ahrefs дозволяє побачити топ-конкурентів за органічними

ключовими словами та реферальними доменами. Цей підхід показує, які сайти мають перетин у ключових запитах, отже є тематично близькими конкурентами. Плюсом є наявність детальної SEO-інформації, але мінусом – обмеженість тільки сферою органічного пошуку (ігноруються інші параметри подібності) і висока вартість передплати.

Серед невеликих онлайн-сервісів варто згадати такі інструменти, як PR-CY Similar Website Finder або NicheProwler. Наприклад, PR-CY пропонує безкоштовний пошук схожих сайтів на основі аналізу тематики і контенту ресурсу. NicheProwler обіцяє надати «комплексний список релевантних сайтів» з оцінкою релевантності та метриками типу оцінки трафіку і авторитетності домену. Їхні алгоритми недокументовані і зазвичай базуються на поєднанні факторів (текстових, SEO-даних та ін.), що ускладнює оцінку їх точності. Застосування таких інструментів доцільне для швидкого орієнтування у конкурентному середовищі та попереднього збору списку кандидатів, але для серйозного використання краще покладатися на більш прозорі та надійні рішення. Також такі сервіси погано підходять для користувачів з «не популярних» країн, так як вони не мають інформації про локальні сайти.

Перевагами сучасних систем є інтеграція різних джерел даних: контенту сторінки, індексу пошукових систем, поведінкових метрик; і можливість масштабного аналізу – наприклад, одночасно для десятків сайтів, як у Ahrefs чи Similarweb. Недоліками часто є закритість алгоритмів (користувач не знає точно, які ознаки враховані), залежність від зовнішніх даних (якість збору інформації про трафік), а також потреба у великих обчислювальних ресурсах і оплаті.

1.3 Порівняння методів пошуку схожих сайтів

Методи пошуку схожих сайтів можна розділити на кілька категорій: пошук за ключовими словами (зважування термінів), семантичний пошук

(векторизація документів) і глибинні (з використанням засобів великих мовних моделей).

TF-IDF (Term Frequency-Inverse Document Frequency) – це широко використовуваний статистичний метод в обробці природної мови та пошуку інформації. Він вимірює важливість терміна в документі відносно колекції документів [7]. Перевага TF-IDF – простота реалізації та хороша ефективність для порівняння текстів із явною лексичною подібністю. Його часто використовують у методах інформаційного пошуку і рекомендацій. Однак недоліки суттєві: TF-IDF не враховує порядок слів і контексту, а також не оперує семантикою чи синонімами. Тому TF-IDF підходи добре працюють, коли очікується часткове лексичне перекриття, але видають низьку точність при перефразованих описах. При цьому TF-IDF легко інтегрувати в Node.js (є пакети natural, wink-nlp тощо) та виконувати обчислення швидко.

LDA (Latent Dirichlet Allocation) – тематичне моделювання, яке розподіляє документи по темах. Цей підхід дозволяє перейти від конкретних слів до прихованих тем і порівнювати сайти за улюбленими тематиками користувачів. Якщо два сайти мають схоже тематичне профілювання, їх можна вважати схожими. LDA – це статистичний підхід, що не потребує позначених вручну даних і може зменшувати розмірність уявлення (замість тисяч слів – кілька десятків тем) [8]. Проте він має недоліки: теми можуть виявлятися нечіткими або непов’язаними між собою, особливо якщо дані засмічені. Також вимагає вгадування числа тем і тренування на великому корпусі.

Векторні представлення (Doc2Vec, word2vec, sentence transformers). Сучасні методи використовують щільні вектори, які кодують семантичну інформацію тексту. Наприклад, бібліотека Sentence-BERT створює вставки для всього документу. Відповідно, два документи, що близькі за змістом, матимуть близькі вектори. Переваги: враховується контекст і семантика. Подібність вбудовування оцінюється головним чином за косинусною подібністю. Вона обчислюється як косинус кута між двома векторами. Це

особливо корисно, коли тексти не однакової довжини [9]. Недоліком є необхідність потужного апаратного забезпечення та великих моделей (сам BERT має сотні мільйонів параметрів). В Node.js з’явилася підтримка таких моделей через бібліотеки типу Transformers.js або LLaMa-node. Однак зазвичай їх застосовують через окремі сервіси або API (наприклад, Hugging Face Inference, чи Amazon Bedrock [10]), оскільки запуск BERT у браузері чи на сервері без GPU досить повільний.

Використання великих мовних моделей (LLaMa, GPT тощо). Це найсучасніший підхід, де великі мовні моделі можуть оцінювати схожість на дуже високому рівні. Наприклад, можна надіслати LLM завдання «порівняй два сайти і скажи, наскільки вони схожі за темою» або використовувати вбудовані семантичні вставки моделей. На відміну від традиційних методів, LLM можуть врахувати навіть прихований сенс, сарказм чи тон тексту. Однак їх використання вимагає значних обчислювальних ресурсів: сучасні LLM мають мільярди параметрів. У середовищі Node.js можна інтегрувати LLaMa за допомогою бібліотек на кшталт LLaMa-node, або працювати через REST API (наприклад, OpenAI GPT чи через LLaMa Stack), чи SDK від AWS. Це дозволяє розв’язувати нестандартні задачі – наприклад, LLM можуть врахувати контекст попередніх сторінок, створити узагальнення контенту сайтів або відповісти на запит «які сайти схожі за змістом». Переваги LLM – глибоке розуміння мови і гнучкість, недоліки – потреба в потужному апаратному забезпеченні, витрати та інтерпретованість результатів (бо LLM працює як чорний ящик). З практичної точки зору, вбудувати LLM безпосередньо в проєкт на Node.js досить складно без використання додаткових модулів: однак приклад бібліотеки LLaMa-node показує, що це можливо.

Нарешті, треба відзначити, що вибір методу залежить від конкретних вимог і ресурсів. Пошук за ключовими термінами є простим та швидким, підходить для початкового фільтрування сайтів. Тематичне моделювання забезпечує виявлення домінантних тем, але є складнішим, вимагає більше

ресурсів. Трансформери (BERT) дають кращу точність у семантичному порівнянні, але вимагають складної інтеграції. LLM-методи (LLaMa, GPT) – найбільш гнучкі, особливо якщо потрібно аналізувати велике та різнорідне семантичне поле, але коштовні в обчисленнях.

1.4 Формування вимог до проєкту

1.4.1 Загальна характеристика проєкту

Метою створення інформаційної системи є надання користувачеві можливості здійснювати пошук схожих вебсайтів на основі аналізу вмісту наданих сторінок, чи пошук за вже заданими характеристиками. Система повинна використовувати сучасні методи аналізу тексту, зокрема велику мовну модель (LLM, типу LLaMa чи DeepSeek R1), для генерації тегів та коротких описів вебсайтів. Надалі ці метадані використовуються для знаходження сайтів із подібною семантикою.

1.4.2 Основні функції системи

Додавання нового сайту:

– користувач вводить URL сайту;
– система автоматично витягує вміст сторінки;
– на основі отриманого вмісту формується опис і теги за допомогою LLM;
– інформація зберігається в базі даних.

Пошук схожих сайтів:

– за введеним URL або тегами система пропонує список сайтів із найбільшою схожістю;

– подібність оцінюється за кількістю спільних тегів та/або векторною схожістю описів;
– підтримка операторів включення (нічого), виключення (-) [11] та опціональних (~) операторів;
– результати сортувати за релевантністю.

Управління базою даних (для адміністратора):

– можливість редагувати або видаляти записи;
– перевірка якості описів та тегів.

Історія запитів (для зареєстрованих користувачів):

– збереження запитів користувачів;
– можливість перегляду раніше знайдених схожих сайтів.

1.4.3 Функціональні вимоги

До функціональних вимог розроблюваної системи належать:

– автоматичне витягування тексту з сайту: застосування puppeteer/playwright для рендерингу html;
– генерація опису та тегів: інтеграція з LLaMa або зовнішнім API для отримання результату;
– пошук за схожістю: алгоритм порівняння тегів + семантичне порівняння описів;
– збереження результатів: інформація зберігається в реляційній базі даних (sqlite/postgresql);
– інтерфейс користувача: форма для введення url, відображення результатів, історія;
– авторизація користувача: мінімальна система реєстрації/логіну;
– роль адміністратора: доступ до системи редагування даних.

1.4.4 Нефункціональні вимоги

До функціональних вимог відносяться:

– продуктивність: обробка запиту (від введення посилань до отримання результатів) не повинна перевищувати 10 секунд;
– надійність: система повинна бути стійкою до обробки непрацюючих посилань;
– безпека: хешування паролів користувачів, захист від SQL-ін’єкцій;
– масштабованість: можливість розширення функціоналу пошуку та індексації без повного перепроєктування системи.

1.4.5 Вимоги до технологій

Приблизний список технологій для використання у розробці:

– frontend: React, TypeScript/JavaScript, TailwindCSS, Axios;
– backend: Node.js (Express), LLM API або інтеграція з LLaMa через LLaMa-node, Puppeteer або Playwright (для отримання тексту з DOM);
– база даних: SQLite (локальна) або PostgreSQL (версія на випуск);
– середовище розгортання: локально (для тестування), Vercel/Render (для деплою клієнтської частини), Railway/Render/Heroku (для бекенду), або AWS для більшої доступності та надійності системи в хмарі.

1.4.6 Вимоги до інтерфейсу

Вимоги до інтерфейсу фронеңду:

– проста структура, інтуїтивно зрозумілий інтерфейс;
– адаптивна верстка для підтримки мобільних пристроїв;
– відображення тегів у вигляді кнопок;

– пошук за посиланням або тегами;
– пропозиції тегів під час пошуку в реальному часі.

1.5 Аналіз інструментів

Розробка системи для пошуку схожих сайтів вимагає використання надійного, масштабованого і сучасного технологічного стеку. Вибір інструментів ґрунтувався на таких критеріях: відповідність архітектурі вебдодатку, підтримка асинхронних запитів, активне ком'юніті, наявність готових бібліотек для обробки тексту, інтеграції з LLM і роботи з базою даних.

1.5.1 Екосистема JavaScript

JavaScript еволюціонував від мови скриптів браузера до комплексної екосистеми, що підтримує full-стек розробку [12]. Створена у 1995 році Бренданом Айхом у Netscape, мова пройшла значну стандартизацію через процес ECMAScript.

Ключовими характеристиками є:

– динамічне типізування: JavaScript використовує динамічне типізування з типами, визначеними під час виконання. TypeScript вирішує це шляхом додавання статичної перевірки типів [13];
– програмування, кероване подіями: цикл подій JavaScript дозволяє неблокуюче, асинхронне програмування, добре підходяще для застосунків з інтенсивною кількістю I/O;
– функції першого класу: функції є першокласними громадянами, що дозволяє створювати функціональні програмні шаблони.

Пакетний менеджер. Реєстр NPM містить понад 2 мільйони пакетів, утворюючи найбільший у світі реєстр програмного забезпечення [14]. Ця екосистема пропонує рішення практично для будь-яких програмних завдань.

Сучасні особливості. Функції ES6+, які використовуються в SiteAlike, включають:

– модулі ES для імпорту/експорту;
– async/wait для асинхронного програмування;
– оператори деструктурування та розповсюдження;
– оператор опціональної послідовності.

Обґрунтування вибору. Згідно з опитуванням Stack Overflow Developer Survey [15] та State of JavaScript [16], JavaScript залишається найпоширенішою мовою програмування з відмінною підтримкою екосистеми.

1.5.2 Node.js Runtime Environment

Node.js – це JavaScript-runtime, побудований на рушії V8, створений Раяном Далом у 2009 році.

Архітектура:

– рушій V8: компіляція JavaScript у native код;
– libuv: цикл подій та асинхронний I/O;
– основні модулі: http, fs, path, crypto, stream.

Node.js був обраний за:

– відмінна продуктивність введення/виведення;
– кодшеринг з фронтендом React;
– сильна безсерверна сумісність;
– активна підтримка громади.

1.5.3 Бібліотека React

React – це бібліотека JavaScript для створення користувацьких інтерфейсів, розроблена Facebook. Випущена у 2013 році, вона стала найпопулярнішою фронтенд-бібліотекою згідно з опитуваннями галузі.

Основні концепції бібліотеки. Архітектура на основі компонентів: багаторазові компоненти, що інкапсулюють логіку та репрезентацію. Віртуальний DOM: ефективний механізм оновлення, що обчислює мінімальні зміни DOM. Декларативний синтаксис: розробники описують бажаний стан інтерфейсу; React обробляє переходи. JSX: XML-подібне синтаксичне розширення, скомпільоване на JavaScript.

Сучасний React використовує функціональні компоненти з хуками:

– useState: керування станом;
– useEffect: керування ефектами;
– useCallback/useMemo: оптимізація продуктивності.

React було обрано на підставі:

– повторного використання компонентів;
– ефективності віртуального DOM;
– великої екосистема;
– статусу галузевого стандарту.

1.5.4 TypeScript

TypeScript – це статично типізований суперсет JavaScript, розроблений компанією Microsoft. Він компілюється у звичайний JavaScript, одночасно забезпечуючи перевірку типів.

Ключові особливості:

– визначення статичних типів;
– інтерфейси та псевдоніми типів;
– дженерики;
– типи сполучень і перехрестя.

Обґрунтування вибору. TypeScript надає:

– раннє виявлення помилок;
– покращену підтримку IDE;

– самодокументування коду;
– безпечний рефакторинг.

1.5.5 Tailwind CSS

Tailwind CSS – це CSS-фреймворк, орієнтований на корисність. Замість готових компонентів він надає класи утиліти, що відображаються з властивостями CSS.

Переваги:

– швидка розробка;
– узгодженість системи проєктування;
– невеликий виробничий пакет через PurgeCSS;
– нативна підтримка чутливого та темного режиму.

1.5.6 PostgreSQL СУБД

PostgreSQL – це передова реляційна база даних з відкритим кодом, відома своєю надійністю та стійкістю.

Ключові особливості:

– відповідність ACID: повна підтримка транзакцій для забезпечення цілісності даних;
– розширені типи даних: JSON/JSONB, ARRAY, UUID, повнотекстові типи пошуку;
– розширення: pg_trgm для нечіткого співпадіння, uuid-ossp для генерації UUID;
– повнотекстовий пошук: вбудовані можливості з ранжуванням.

PostgreSQL був обраний через: доведену надійність, розширені можливості запитів, повнотекстовий пошук та ліцензування з відкритим кодом.

1.5.7 Сервіси AWS

Amazon Web Services надає хмарну інфраструктуру для системи.

Використані сервіси:

– **Cognito**: автентифікація користувача та токени JWT;
– **Lambda**: виконання безсерверних функції;
– **API Gateway**: управління HTTP точками;
– **S3**: зберігання скріншотів;
– **CloudFront**: CDN для статичних образів;
– **RDS**: керований PostgreSQL.

AWS було обрано через те, що керовані послуги зменшують операційне навантаження, наявна можливість автоматичного масштабування, модель «Плати за використання» та комплексну інтеграцію послуг.

1.5.8 Додаткові інструменти

Vite: сучасний інструмент для збірки з нативними модулями ES і швидким HMR.

Zustand: легке управління станами з мінімальним API.

Puppeteer: Headless Chrome для зйомки скріншотів.

Jest: фреймворк тестування з вбудованими припущеннями.

ESLint: статичний аналіз якості коду.

Docker: контейнеризація для узгоджених середовищ.

1.6 Висновки до розділу 1

У першому розділі було детально аналізовано предметну область, сформовано розгорнуті вимоги до проєкту та проведено порівняння існуючих

систем та методів. Був обраний інструментарій для розробки майбутньої системи.

2 ПРОЄКТУВАННЯ СИСТЕМИ

2.1 Огляд архітектури системи

Система SiteAlike дотримується сучасної трирівневої архітектури, розмежуючи повноваження між шарами презентації, бізнес-логіки та збереження даних. Такий архітектурний підхід надає кілька переваг, зокрема покращення обслуговуваності, масштабованості та можливості незалежно розвивати кожен шар [18].

Система реалізує клієнт-серверну архітектуру з наступними характеристиками (див. рис. 2.1):

– інтерфейс односторінкового додатка (SPA), який працює повністю в браузері;
– RESTful API бекенд надає безстатусні сервіси;
– реляційна база даних для постійного зберігання даних;
– хмарно-нативне розгортання з використанням керованих сервісів.

Ця архітектура відповідає сучасним найкращим практикам розробки вебдодатків і дозволяє системі масштабуватися горизонтально у міру зростання попиту.

Потік взаємодії компонентів:

а) користувач взаємодіє з React SPA у своєму браузері;
б) запити на автентифікацію обробляються AWS Cognito, який видає JWT токени;
в) запити API надсилаються до шлюзу API AWS з заголовками авторизації JWT;
г) API Gateway перевіряє токени та направляє запити до відповідних Lambda функцій;
д) Lambda функції виконують бізнес-логіку та звертають запит до бази даних PostgreSQL;

е) скріншоти зберігаються та надаються з AWS S3;

ж) відповіді повертаються тим самим шляхом до клієнта.

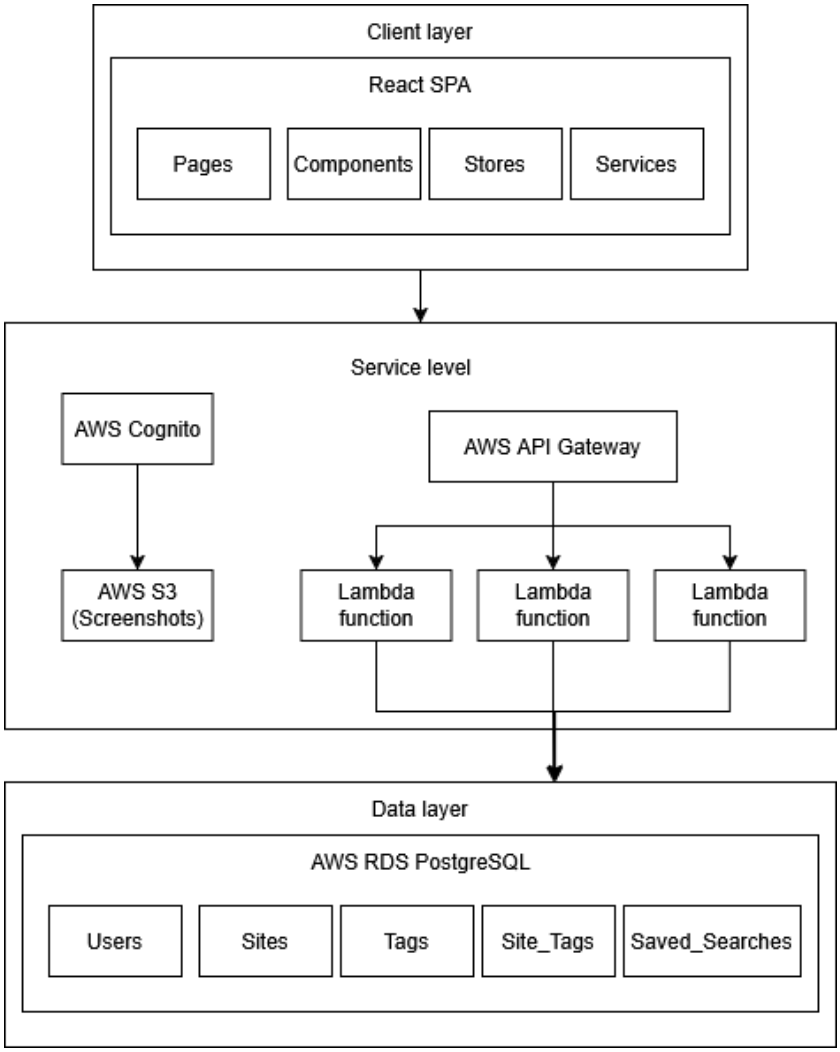


Рисунок 2.1 – Діаграма високорівневої архітектури

2.2 Проєктування бази даних

2.2.1 Ідентифікація сутностей та атрибутів

На основі аналізу вимог для бази даних SiteAlike визначено такі сутності:

– **Users** (див. табл. 2.1);

– **Sites**: представляє вебсайти, каталогізовані в системі (див. табл. 2.2);

- **Tags**: представляє описові мітки, які можна застосовувати до вебсайтів (див. табл. 2.3);
- **Site_Tags**: представляє взаємозв'язок між сайтами та тегами (див. табл. 2.4);
- **User_Favorites**: відстежує, які сайти користувачі вказали у вибране (див. табл. 2.5);
- **User_Blacklists**: відстежує теги, які користувачі хочуть виключити зі своїх результатів (див. табл. 2.6);
- **Saved_Searches**: зберігає пошукові запити користувачів для повторного використання (див. табл. 2.7);
- **User_Preferences**: зберігає налаштування користувача (див. табл. 2.8).

Таблиця 2.1 – Сутність користувачів

Атрибут	**Тип**	**Обмеження**	**Опис**
id	UUID	PRIMARY KEY	Унікальний ідентифікатор
cognito_id	VARCHAR(255)	UNIQUE, NOT NULL	Ідентифікатор AWS Cognito
username	VARCHAR(50)	UNIQUE, NOT NULL	Ім'я користувача
email	VARCHAR(255)	UNIQUE, NOT NULL	Пошта користувача
avatar_url	TEXT	NULLABLE	URL зображення профілю
role	VARCHAR(20)	CHECK constraint	user/moderator/admin
created_at	TIMESTAMP	DEFAULT NOW()	Час реєстрації
updated_at	TIMESTAMP	DEFAULT NOW()	Час останнього оновлення

Таблиця 2.2 – Сутність вебсайтів

Атрибут	**Тип**	**Обмеження**	**Опис**
id	UUID	PRIMARY KEY	Унікальний ідентифікатор
url	TEXT	UNIQUE, NOT NULL	Посилання на сайт
title	VARCHAR(255)	NOT NULL	Назва сайту
description	TEXT	NULLABLE	Опис сайту
screenshot_full_key	TEXT	NULLABLE	Ключ S3 для скріншотів
screenshot_thumb_key	TEXT	NULLABLE	Ключ S3 для мініатюр
screenshot_captured_at	TIMESTAMP	NULLABLE	Час створення скріншотів
status	VARCHAR(20)	CHECK constraint	pending/active/flagged/deleted

Продовження табл. 2.2

Атрибут	Тип	Обмеження	Опис
rating	VARCHAR(15)	CHECK constraint	safe/questionable/explicit
view_count	INTEGER	DEFAULT 0	Лічильник переглядів
favorite_count	INTEGER	DEFAULT 0	Лічильний вподобань
submitted_by	UUID	FOREIGN KEY	Додано користувачем
approved_by	UUID	FOREIGN KEY	Затверджуючий модератор
created_at	TIMESTAMP	DEFAULT NOW()	Час створення
updated_at	TIMESTAMP	DEFAULT NOW()	Час останнього оновлення

Таблиця 2.3 – Сутність тегів

Атрибут	Тип	Обмеження	Опис
id	UUID	PRIMARY KEY	Унікальний ідентифікатор
name	VARCHAR(100)	UNIQUE, NOT NULL	Назва тегу на показ
slug	VARCHAR(100)	UNIQUE, NOT NULL	URL-дружній ідентифікатор
category	ENUM	NOT NULL	general/technology/industry/style/ meta …
description	TEXT	NULLABLE	Опис
aliases	TEXT[]	DEFAULT '{}'	Альтернативні імена
usage_count	INTEGER	DEFAULT 0	Лічильник використань
created_by	UUID	FOREIGN KEY	Створено користувачем
created_at	TIMESTAMP	DEFAULT NOW()	Час створення

Таблиця 2.4 – Сутність взаємозв’язку між сайтами і тегами

Атрибут	Тип	Обмеження	Опис
site_id	UUID	FOREIGN KEY, PK	Посилання на сайт
tag_id	UUID	FOREIGN KEY, PK	Посилання на тег
added_by	UUID	FOREIGN KEY	Користувач, що додав
created_at	TIMESTAMP	DEFAULT NOW()	Час створення

Таблиця 2.5 – Сутність вподобань користувачів

Атрибут	Тип	Обмеження	Опис
user_id	UUID	FOREIGN KEY, PK	Посилання на користувача
site_id	UUID	FOREIGN KEY, PK	Посилання на тег
created_at	TIMESTAMP	DEFAULT NOW()	Час створення

Таблиця 2.6 – Сутність небажаних тегів

Атрибут	**Тип**	**Обмеження**	**Опис**
user_id	UUID	FOREIGN KEY, PK	Посилання на користувача
tag_id	UUID	FOREIGN KEY, PK	Посилання на тег
created_at	TIMESTAMP	DEFAULT NOW()	Час створення

Таблиця 2.7 – Сутність збережених пошуків

Атрибут	**Тип**	**Обмеження**	**Опис**
id	UUID	PRIMARY KEY	Унікальний ідентифікатор
user_id	UUID	FOREIGN KEY	Посилання на користувача
name	VARCHAR(100)	NOT NULL	Ім’я пошуку
query	TEXT	NOT NULL	Рядок пошукового запиту
notify_on_new	BOOLEAN	DEFAULT FALSE	Побажання сповіщення
created_at	TIMESTAMP	DEFAULT NOW()	Час створення
last_used_at	TIMESTAMP	NULLABLE	Останнє використання

Таблиця 2.8 – Сутність користувачів

Атрибут	**Тип**	**Обмеження**	**Опис**
user_id	UUID	PRIMARY KEY, FK	Посилання на користувача
default_rating	VARCHAR(15)	CHECK constraint	Фільтр рейтингу за замовчуванням
results_per_page	INTEGER	CHECK 10-100	побажання пагінації
created_at	TIMESTAMP	DEFAULT NOW()	Час створення
updated_at	TIMESTAMP	DEFAULT NOW()	Час останнього оновлення

2.2.2 Нормалізація та обґрунтування зв’язків

Схема бази даних була нормалізована до Третьої нормальної форми (3NF), щоб усунути надлишковість даних, зберігаючи при цьому продуктивність запитів [19].

Усі таблиці відповідають вимогам першої нормальної форми (**1NF**):

– кожна таблиця має первинний ключ;

– усі атрибути містять атомарні значення;

– повторюваних груп не існує.

Поле псевдонімів у таблиці тегів використовує рідний тип масиву PostgreSQL, який вважається атомарним у системі типів PostgreSQL.

Усі таблиці задовольняють вимоги другої нормальної форми (**2NF**):

- усі неключові атрибути повністю функціонально залежать від усього первинного ключа;
- таблиці з'єднання (Site_Tags, User_Favorites, User_Blacklists) мають складені первинні ключі, усі атрибути яких залежать від повного ключа.

Завершуємо тим, що таблиці також відповідають вимогам третьої нормальної форми (**3NF**):

- транзитивних залежностей не існує;
- неключові атрибути залежать лише від первинного ключа.

Рішення щодо денормалізації. Певні навмисні денормалізації були реалізовані для оптимізації продуктивності:

- view_count та favorite_count на сайтах: ці лічильники денормалізовані, щоб уникнути дорогих запитів COUNT у кожному списку сайту – вони оновлюються через тригери або логіку додатку при додаванні/видаленні вподобань.
- usage_count у тегах: аналогічно, кількість використання тегів денормалізована для ефективного сортування за популярністю.

Ці денормалізації слідують усталеним шаблонам для застосувань з великим навантаженням на читання.

Далі наведено **обгрунтування відносин** між сутностями та їх реалізація.

Користувачі та сайти (One-to-Many): користувач може додавати кілька сайтів, але на кожному сайті є один додавач. Реалізовано через зовнішній ключ submitted_by.

Користувачі обрані сайти (Many-to-Many): користувачі можуть ставити в улюблене кілька сайтів; сайти можуть бути обраними кількома користувачами. Реалізовано через таблицю з'єднань User_Favorites.

Користувачі до тегів через чорний список (Many-to-Many): користувачі можуть додавати до чорного списку багато тегів; теги можуть бути внесені до

чорного списку кількома користувачами. Реалізовано через таблицю з’єднань User_Blacklists.

Сайти до тегів (Many-to-Many): сайти мають кілька тегів; теги застосовуються до кількох сайтів. Реалізовано через таблицю з’єднань Site_Tags.

2.2.3 Графічне зображення бази даних

Враховуючи все вищезазначене, можна побудувати ER-діаграму де кожна таблиця відображається як клас із переліком властивостей (див. рис. 2.2).

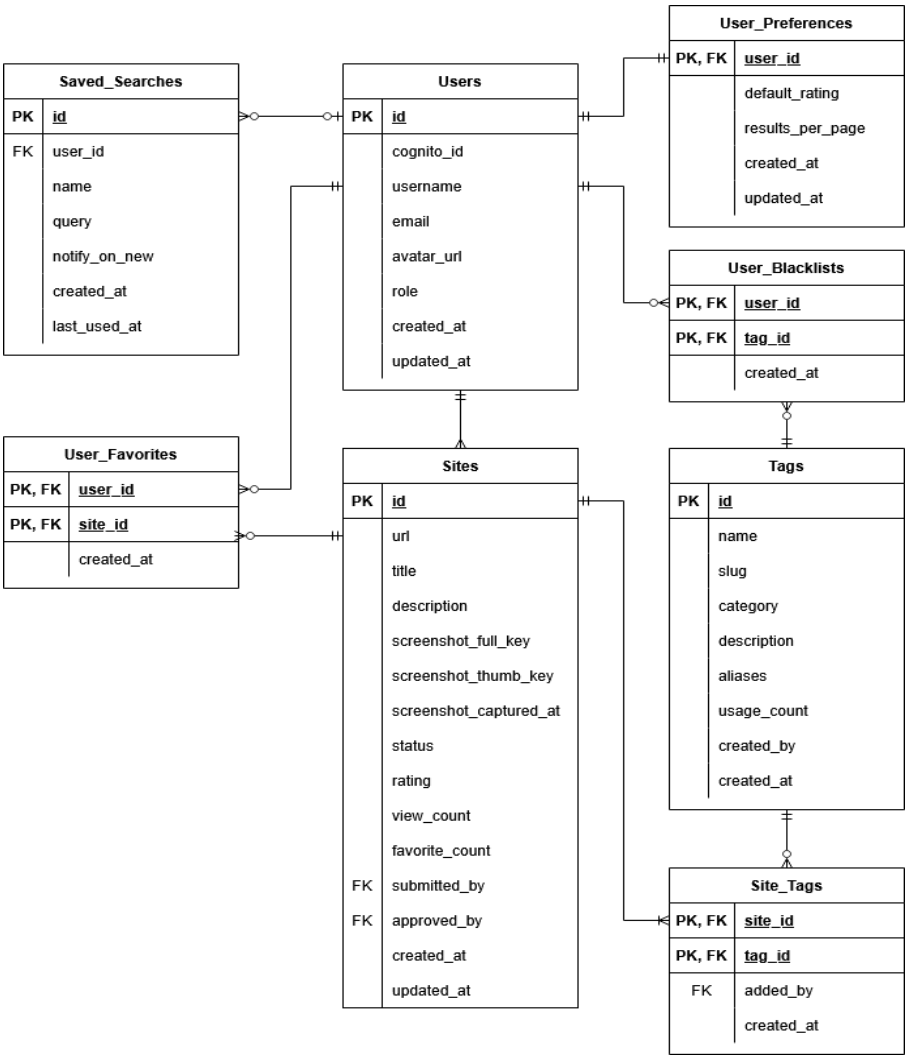


Рисунок 2.2 – ER-діаграма бази даних

2.2.4 Стратегія індексації

Відповідні індекси розроблені для оптимізації продуктивності запитів [19] (див. рис. 2.3 – 2.5).

Усі таблиці мають індекси B-дерева у своїх стовпцях первинних ключів.

```
CREATE INDEX idx_site_tags_tag ON site_tags (tag_id);
CREATE INDEX idx_site_tags_site ON site_tags (site_id);
CREATE INDEX idx_saved_searches_user ON saved_searches (user_id);
```

Рисунок 2.3 – Створення індексів зовнішніх ключів

```
-- Trigram index for fuzzy tag autocomplete
CREATE INDEX idx_tags_name_trgm ON tags USING gin (name gin_trgm_ops);

-- Full-text search index for site content
CREATE INDEX idx_sites_search ON sites USING gin (
   to_tsvector('english', title || ' ' || COALESCE(description, ''))
);
```

Рисунок 2.4 – Створення індексів оптимізації пошуку

```
CREATE INDEX idx_sites_status ON sites (status);
CREATE INDEX idx_sites_rating ON sites (rating);
CREATE INDEX idx_sites_created ON sites (created_at DESC);
CREATE INDEX idx_tags_usage ON tags (usage_count DESC);
CREATE INDEX idx_tags_category ON tags (category);
```

Рисунок 2.5 – Створення індексів фільтрації та сортування

2.3 Дизайн бекенду

2.3.1 Архітектура API

Бекенд дотримується принципів дизайну RESTful API [20], реалізуючи ресурсоорієнтовану архітектуру зі стандартними HTTP-методами. Структура кінцевих точок API зазначена у таблиці 2.9

Таблиця 2.9 – Структура кінцевих точок API

Метод	**Кінцева точка**	**Опис**	**Потрібна автентифікація**
GET	/sites	Список сайтів із пагінацією	ні
GET	/sites/:id	Отримати інформацію про сайт	ні
POST	/sites	Додати новий сайт	так
PUT	/sites/:id	Оновити сайт	так (власник\адмін)
DELETE	/sites/:id	Видалити сайт	так (власник\адмін)
GET	/tags	Перелічити теги	ні
GET	/tags/search	Шукати теги (автозаповнення)	ні
POST	/tags	Створити тег	так
GET	/users/profile	Отримати профіль користувача	так
PUT	/users/profile	Оновити профіль	так
GET	/users/blacklist	Отримати заблоковані теги	так
PUT	/users/blacklist	Оновити список заблокованих тегів	так
GET	/search/saved	Отримати збережений пошук	так
POST	/search/saved	Зберегти пошук	так
GET	/admin/stats	Отримати статистику системи	так (адмін)
GET	/admin/users	Список користувачів	так (адмін)

2.3.2 Організація lambda функцій

Бекенд організований як окремі lambda-функції, дотримуючись принципу єдиної відповідальності. Структуру можна побачити на рисунку 2.6. Також на рисунку 2.7 продемонстровано діаграму потоку функціоналу.

```
backend/
├── src/
│   ├── handlers/
│   │   ├── sites/
│   │   │   ├── getSites.ts        # List/search sites
│   │   │   ├── getSiteById.ts     # Get single site
│   │   │   ├── createSite.ts      # Submit new site
│   │   │   ├── updateSite.ts      # Update site
│   │   │   └── deleteSite.ts      # Delete site
│   │   ├── tags/
│   │   │   ├── getTags.ts         # List tags
│   │   │   ├── searchTags.ts      # Autocomplete
│   │   │   └── createTag.ts       # Create tag
```

Рисунок 2.6 – Графічне представлення структури Lambda функцій (аркуш 1)

```
│   │   ├── users/
│   │   │   ├── getProfile.ts      # Get profile
│   │   │   ├── updateProfile.ts   # Update profile
│   │   │   ├── getBlacklist.ts    # Get blacklist
│   │   │   └── updateBlacklist.ts # Update blacklist
│   │   ├── search/
│   │   │   ├── searchSites.ts     # Advanced search
│   │   │   └── savedSearches.ts   # Saved search CRUD
│   │   ├── admin/
│   │   │   ├── getStats.ts         # System statistics
│   │   │   ├── getUsers.ts        # User management
│   │   │   └── updateUser.ts      # Update user role
│   │   ├── screenshots/
│   │   │   └── captureScreenshot.ts # Screenshot capture
│   │   └── health.ts              # Health check
│   ├── middleware/
│   │   └── auth.ts                # JWT validation
│   ├── services/
│   │   ├── database.ts            # PostgreSQL connection
│   │   ├── s3Service.ts           # S3 operations
│   │   └── sqsService.ts          # Queue operations
│   └── utils/
│       ├── response.ts            # Response helpers
│       └── searchParser.ts        # Query parsing
```

Рисунок 2.6 – Графічне представлення структури Lambda функцій (аркуш 2)

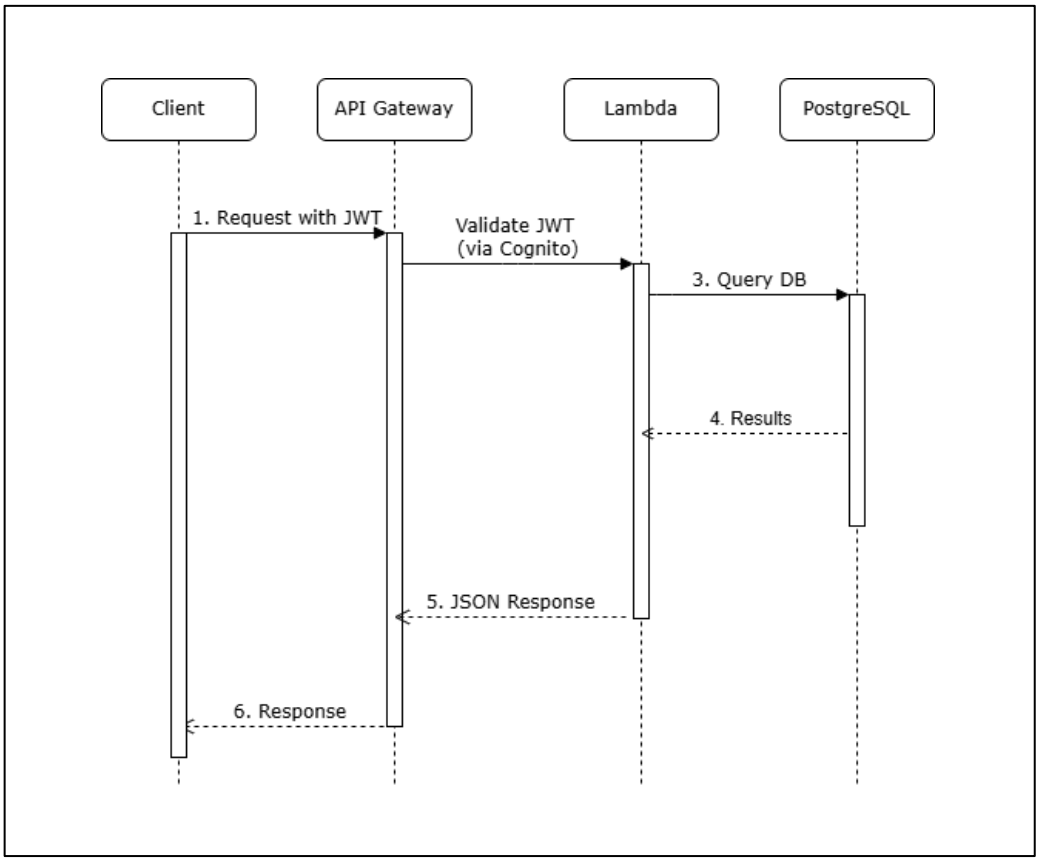


Рисунок 2.7 – Графічне представлення структури Lambda функцій

2.4 Дизайн фронтенду

2.4.1 Архітектура компонентів

Фронтенд дотримується компонентної архітектури React, організовуючи компоненти за функціями та відповідальністю (див. рис. 2.8).

```
backend/
├── src/
│   ├── handlers/
│   │   ├── sites/
│   │   │   ├── getSites.ts        # List/search sites
│   │   │   ├── getSiteById.ts     # Get single site
│   │   │   ├── createSite.ts      # Submit new site
│   │   │   ├── updateSite.ts      # Update site
│   │   │   └── deleteSite.ts      # Delete site
│   │   ├── tags/
│   │   │   ├── getTags.ts         # List tags
│   │   │   ├── searchTags.ts      # Autocomplete
│   │   │   └── createTag.ts       # Create tag
│   │   ├── users/
│   │   │   ├── getProfile.ts      # Get profile
│   │   │   ├── updateProfile.ts   # Update profile
│   │   │   ├── getBlacklist.ts    # Get blacklist
│   │   │   └── updateBlacklist.ts # Update blacklist
│   │   ├── search/
│   │   │   ├── searchSites.ts     # Advanced search
│   │   │   └── savedSearches.ts   # Saved search CRUD
│   │   ├── admin/
│   │   │   ├── getStats.ts         # System statistics
│   │   │   ├── getUsers.ts        # User management
│   │   │   └── updateUser.ts      # Update user role
│   │   ├── screenshots/
│   │   │   └── captureScreenshot.ts # Screenshot capture
│   │   └── health.ts              # Health check
│   ├── middleware/
│   │   └── auth.ts                # JWT validation
│   ├── services/
│   │   ├── database.ts            # PostgreSQL connection
│   │   ├── s3Service.ts           # S3 operations
│   │   └── sqsService.ts          # Queue operations
│   └── utils/
│       ├── response.ts            # Response helpers
│       └── searchParser.ts        # Query parsing
```

Рисунок 2.8 – Графічне представлення структури архітектури компонентів

2.4.2 Дизайн менеджменту станів

Додаток використовує Zustand для управління станами, що пропонує простішу альтернативу Redux (див. рис. 2.9).

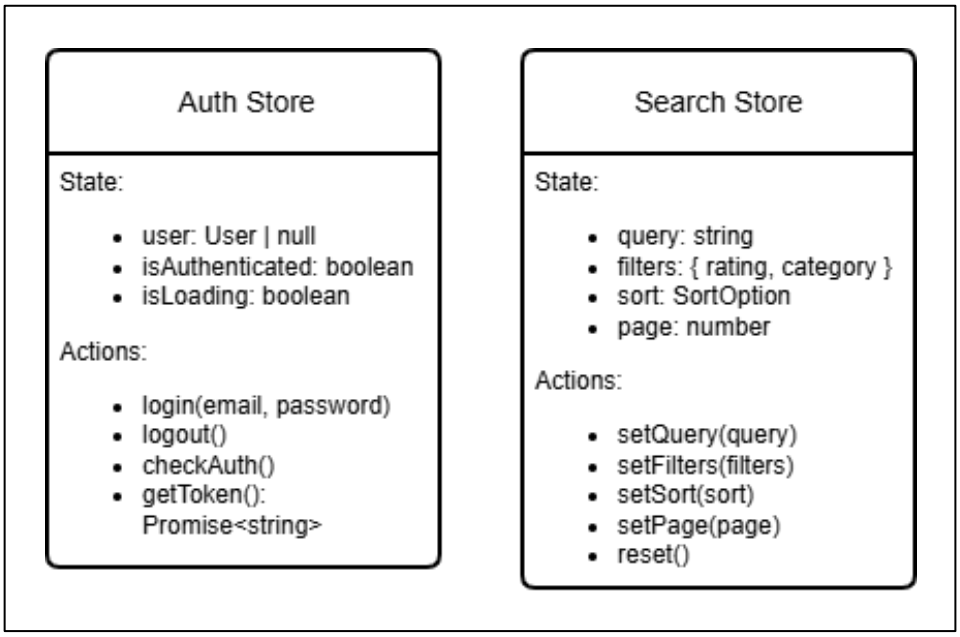


Рисунок 2.9 – Представлення станів

2.5 Проєктування сайту

Для простоти навігації, структура посилань буде пласкою (див. рис. 2.10). Білим позначені сторінки, відкриті для всіх, жовтим – для зареєстрованих користувачів, до червоних сторінок доступ є лише для адміністрації.

Базова структура сторінок складається із верхньої частини, де зліва розміщено логотип сайту, праворуч знаходиться типове відображення користувача. Центр повністю зайнятий рядком пошуку. Нижче знаходиться блок основного контенту, що відрізняється за наповненням відповідно до призначення сторінки. До низу прикріплений стандартний футер з додатковою інформацією (див. рис. 2.11).

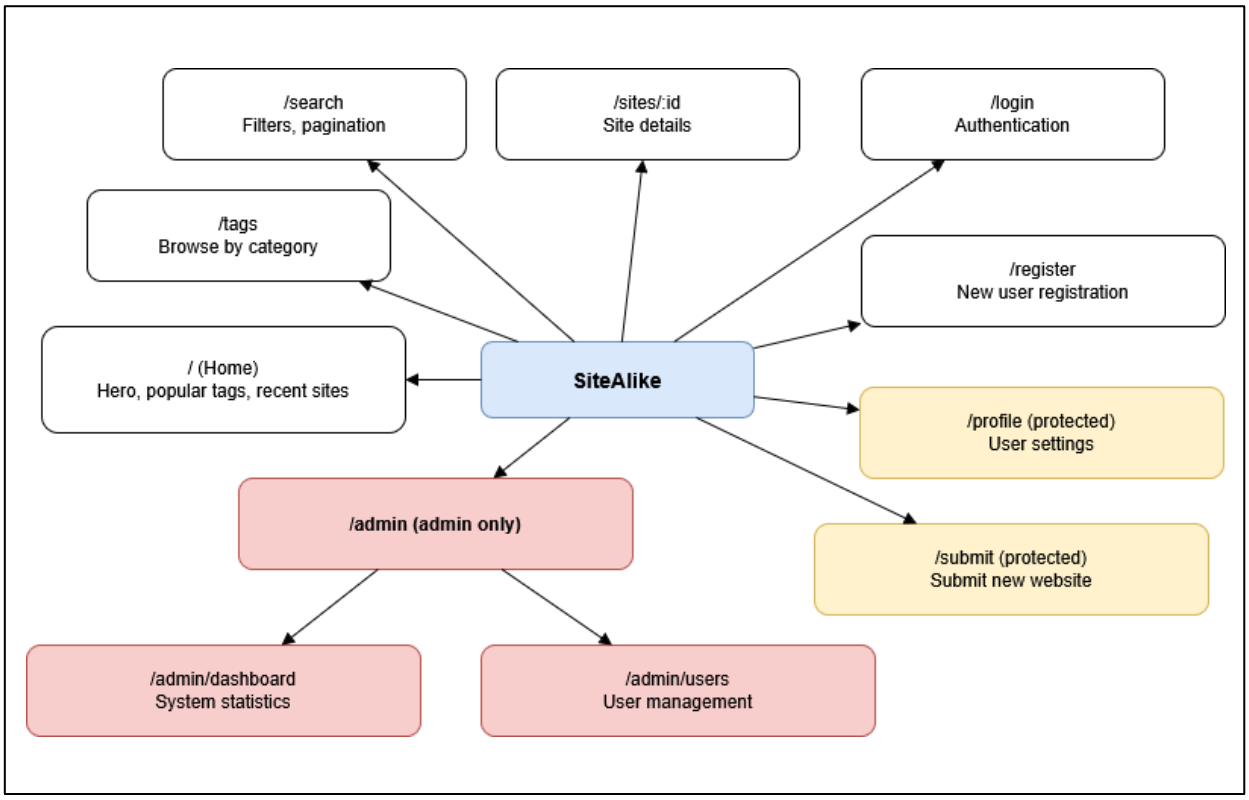


Рисунок 2.10 – Представлення структури сайту

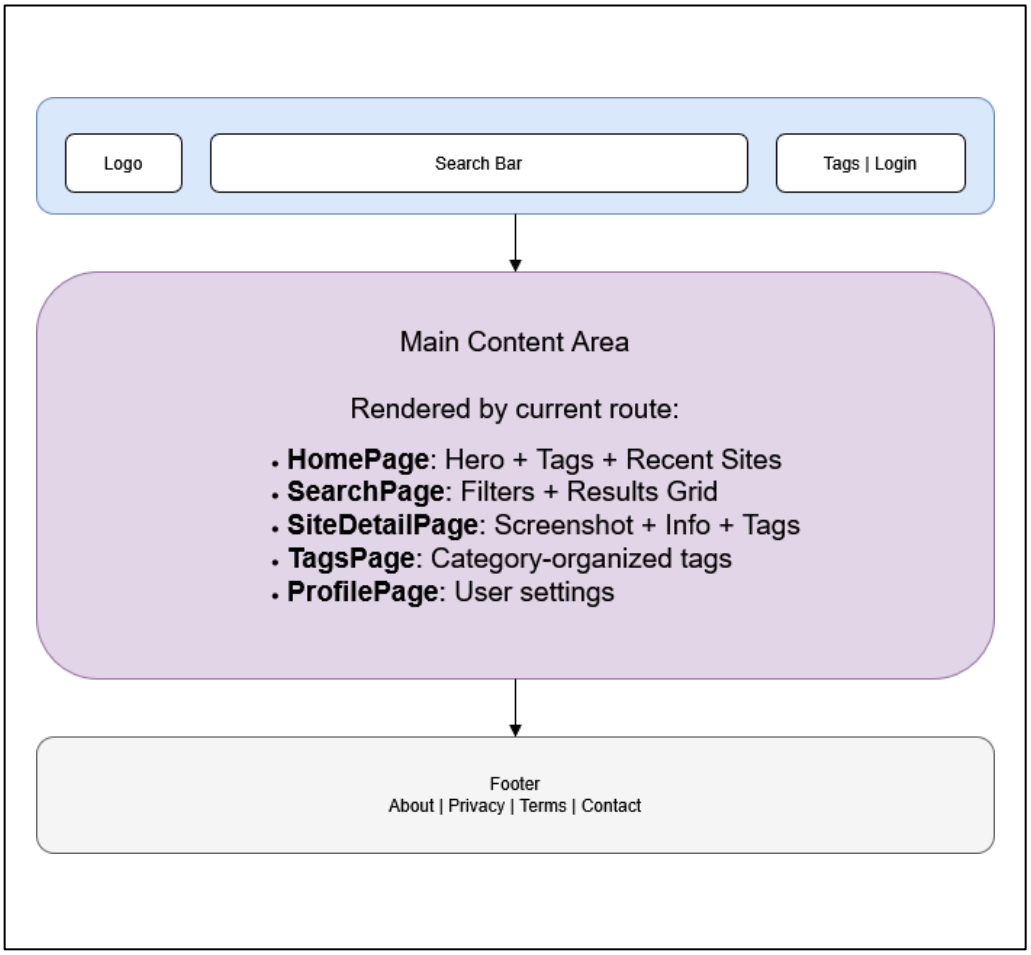


Рисунок 2.11 – Базовий шаблон сайту

2.6 UI/UX дизайн

2.6.1 Система проєктування

Палітра кольорів: система використовує темну тему, оптимізовану для подовженого перегляду та візуального акценту (див. табл. 2.10).

Таблиця 2.10 – Палітра базових кольорів

Роль кольору	**Значення**	**Використання**
Background (surface-900)	#0f172a	Фон сторінки
Card Background (surface-800)	#1e293b	Картки, панелі
Border (surface-700)	#334155	Розділювачі, кордони
Text Primary (surface-100)	#f1f5f9	Заголовки, основний текст
Text Secondary (surface-400)	#94a3b8	Вторинний текст
Primary (blue-500)	#3b82f6	Дії, посилання
Success (green-500)	#22c55e	Стани успіху
Warning (yellow-500)	#eab308	Попередження
Error (red-500)	#ef4444	Помилки

Кожна категорія тегів має унікальний колір для візуальної ідентифікації (див. табл. 11).

Таблиця 2.11 – Палітра кольорів тегів

Категорія	**Колір**	**Значення (HEX)**
Technology	Зелений	#35C64A
Industry	Пурпуровий	#C835C8
Style	Помаранчевий	#F8A800
General	Синій	#0075F8
Meta	Червоний	#FF0000

2.6.2 Типографія

Система використовує сімейство шрифтів sans і mono, як-от: Inter, system-ui, -apple-system, sans-serif та JetBrains Mono, Consolas, monospace.

Дані шрифти роблять вигляд сайту більш сучасним. Масштаб кеглю:

– Heading 1: 2.25rem (36px), bold;
– Heading 2: 1.5rem (24px), semibold;
– Heading 3: 1.25rem (20px), semibold;
– Body: 1rem (16px), normal;
– Small: 0.875rem (14px), normal;
– Caption: 0.75rem (12px), normal.

2.6.3 Component Specifications

Через компоненту орієнтованість React, оптимальною є завчасна розробка структури основних компонентів (див. рис. 2.12 – 2.14).

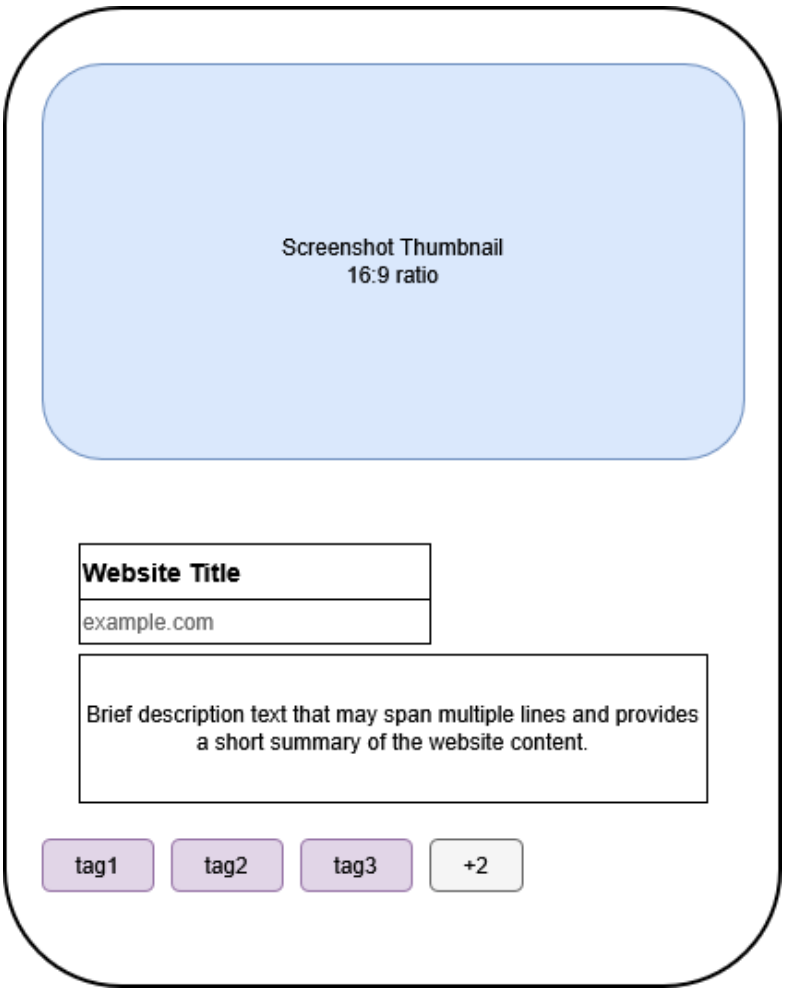


Рисунок 2.12 – Картка сайту

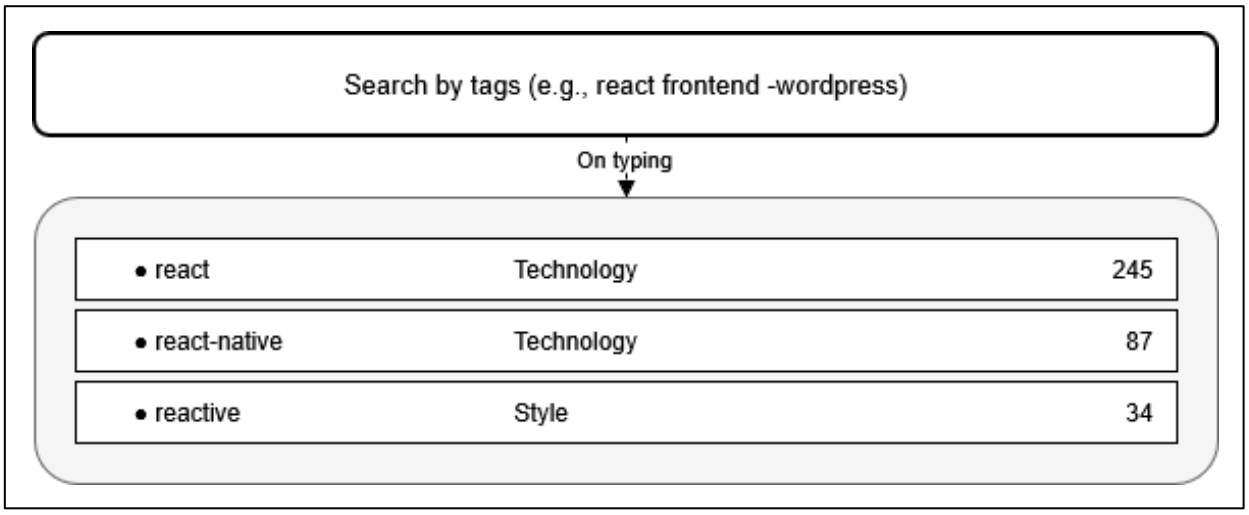


Рисунок 2.13 – Пошук з автопідказкою

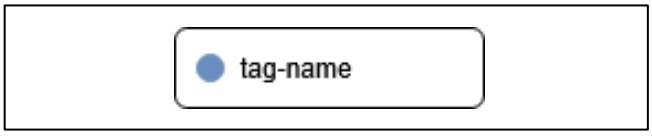


Рисунок 2.14 – Фішка тегу

2.6.4 Точки розриву

Вид плану розділяється на 3 частини: Mobile, Tablet та Desktop, які у свою чергу мають точки розриву < 640px, 640px - 1024px та > 1024px. У першому випадку основний контент подається як єдина колона, а меню сконденсоване у меню гамбургер. У другому випадку основний контент розділяється на дві колонки, а навігація розпадається на декілька все ще конденсованих частин. В останньому випадку контент розділяється на три колони, а навігація повністю розпадається.

2.7 Проєктування інфраструктури AWS

Через те, що було вирішено розгортати систему в хмарі, потрібно провести моделювання інфраструктури (див. рис. 2.15).

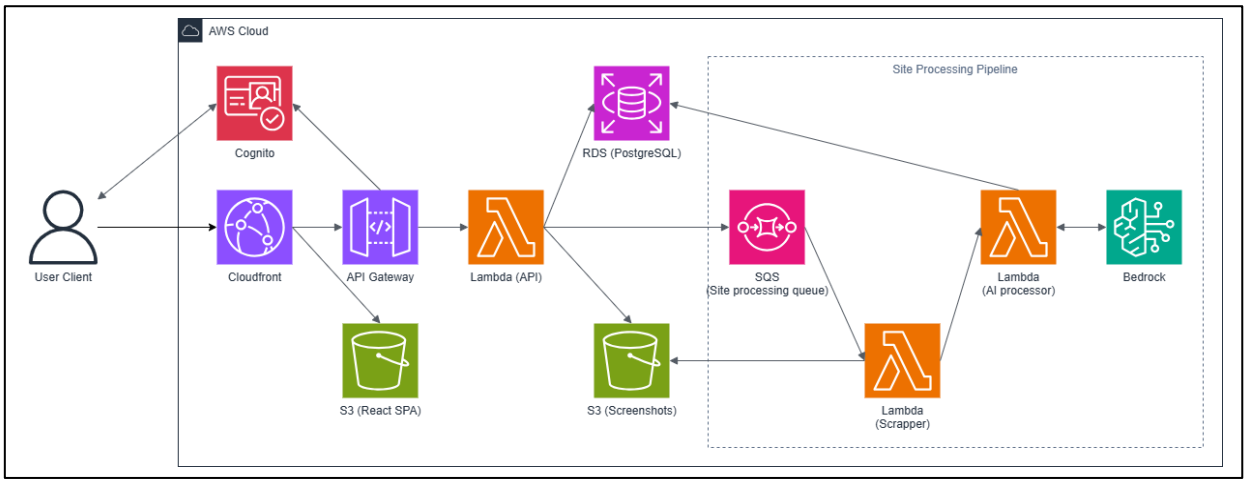


Рисунок 2.15 – Діаграма інфраструктури хмарної системи

З діаграми видно, що система має окремий пайплайн для обробки нових сайтів, а сама система тримається на serverless обчисленнях, а якщо точніше, то на AWS Lambda функціях.

2.7.1 Обґрунтування вибору сервісів

Як мережа доставки контенту (CDN) як для статичних ресурсів, так і для API-запитів використовується Amazon CloudFront. Серед основних чинників зазначимо:

– глобальні периферійні локації зменшують затримку для користувачів по всьому світу;
– автоматичний HTTPS з керованими сертифікатами;
– кешування зменшує навантаження на вихідні сервери;
– захист від DDoS через інтеграцію з AWS Shield.

Сервіс Amazon S3 обрано як об’єктне сховище для додатку React та скріншотів сайтів. Це рішення базується на наступних перевагах:

– сервіс гарантує збереження об’єктів навіть у разі масштабних збоїв;
– економічно вигідно для статичного контенту;
– нативна інтеграція з CloudFront;
– політики життєвого циклу для оптимізації зберігання.

Amazon API Gateway обрано для керування кінцевими точками REST API, обробляючи маршрутизацію, обмеження та авторизацію. Серед переваг такого рішення варто виділити:

– нативна інтеграція з Lambda та Cognito;
– вбудована валідація запитів;
– плани використання та ключі API для обмеження швидкості;
– підтримка специфікації OpenAPI.

AWS Lambda забезпечує безсерверні обчислення для обробників API. Вибір даного рішення зумовлений такими перевагами:

– керування сервером не потрібне;
– автоматичне масштабування для відповідності попиту;
– модель «Плати за виклики», тобто оплата здійснюється виключно за фактичну кількість викликів та час виконання коду;
– нативна підтримка TypeScript через Node.js runtime.

Amazon Cognito здійснює автентифікацію та авторизацію користувачів. Основні чинники такого вибору:

– керований каталог користувачів;
– генерація та валідація токена JWT;
– підтримка постачальників соціальної ідентичності;
– вбудовані функції безпеки (MFA, політики паролів).

Amazon RDS надає керовану базу даних PostgreSQL. Обґрунтування:

– автоматичне резервне копіювання та відновлення;
– розгортання в багатьох зонах для високої доступності;
– автоматичне програмне патчування;
– знайомі функції PostgreSQL.

Amazon SQS забезпечує чергу повідомлень для асинхронної обробки сайтів. Переваги такого рішення:

– відокремлення захоплення скріншотів із запитами API;
– допомога з трафіком;
– аварійна черга для невдалих обробок;
– гарантування доставки принаймні один раз.

2.8 Висновки до розділу 2

В другому розділі представлено комплексний дизайн інформаційної системи SiteAlike, що охоплює всі основні архітектурні та дизайнерські аспекти. Було обрано трирівневу архітектуру, що розділяє презентацію, бізнес-логіку та шари даних. Ця архітектура забезпечує модульність, масштабованість і підтримку, що відповідають сучасним практикам програмної інженерії. Спроєктовано схему бази даних, що включає 8 сутностей, нормалізованих до 3NF, з навмисною денормалізацією для критичних лічильників продуктивності. ER діаграма ілюструє взаємозв'язки між користувачами, сайтами, тегами та пов'язаними об'єктами. Як систему управління базами даних обрано PostgreSQL через її розширені функції, включно з повнотекстовим пошуком триграмами.

Бекенд дотримується принципів RESTful API з Lambda-функціями, організованими за доменами. Безсерверна архітектура забезпечує автоматичне масштабування та економічну ефективність. Автентифікація на основі JWT забезпечує безпечний контроль доступу.

Фронтенд React використовує компонентну архітектуру з Zustand для управління станом. Дизайн розмежування повноважень між сторінками, компонентами, сервісами та сховищами, полегшуючи обслуговування та тестування.

Хмарна інфраструктура використовує керовані сервіси AWS для мінімізації операційних накладних витрат, забезпечуючи масштабованість і надійність. Безпека реалізується рівнево – від захисту входу до шифрування даних.

Цей дизайн забезпечує міцну основу для реалізації, описаної в розділі 3.

3 РОЗРОБКА СИСТЕМИ

3.1 Структура проєкту

Проєкт SiteAlike організований як монорепозиторій з використанням робочих просторів npm, що дозволяє створювати спільний код і координувати розробку між кількома пакетами (див. рис. 3.1).

```
SiteAlike/
├── package.json              # Root package with workspace config
├── tsconfig.base.json        # Shared TypeScript configuration
├── docker-compose.yml        # Local development services
├── backend/
│   ├── package.json
│   ├── serverless.yml        # AWS deployment config
│   ├── tsconfig.json
│   └── src/
│       ├── handlers/         # Lambda function handlers
│       ├── middleware/       # Auth middleware
│       ├── services/         # Database, S3, SQS services
│       └── utils/            # Helper utilities
├── frontend/
│   ├── package.json
│   ├── vite.config.ts        # Vite build configuration
│   ├── tailwind.config.js    # Tailwind CSS config
│   ├── tsconfig.json
│   └── src/
│       ├── components/       # React components
│       ├── pages/            # Page components
│       ├── services/         # API services
│       ├── store/            # Zustand stores
│       └── hooks/            # Custom hooks
├── shared/
│   ├── package.json
│   └── src/
│       ├── site.ts           # Site type definitions
│       ├── tag.ts            # Tag type definitions
│       ├── user.ts           # User type definitions
│       └── API.ts            # API type definitions
├── scripts/
│   ├── init-db.sql           # Database schema
│   ├── seed.ts               # Sample data seeding
│   └── captureScreenshots.ts # Screenshot utility
└── tests/
    ├── package.json
    └── API.test.ts           # API integration tests
```

Рисунок 3.1 – Графічне представлення структури проєкту

3.2 Розробка бекенду

3.2.1 Реалізація сервісу бази даних

Сервіс бази даних надає пул з’єднань і помічник з запитів для доступу до PostgreSQL (див. рис. 3.2).

```
// backend/src/services/database.ts
import { Pool, PoolClient, QueryResult } from 'pg';

// Connection pool configuration
const pool = new Pool({
  host: process.env.DB_HOST,
  port: parseInt(process.env.DB_PORT || '5432'),
  database: process.env.DB_NAME,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
  max: 10,                    // Maximum connections in pool
  idleTimeoutMillis: 30000,   // Close idle connections after 30s
  connectionTimeoutMillis: 2000,
});

// Generic query function with type safety
export async function query<T>(
  text: string,
  params?: unknown[]
): Promise<T[]> {
  const client = await pool.connect();
  try {
    const result: QueryResult<T> = await client.query(text, params);
    return result.rows;
  } finally {
    client.release();
  }
}

// Transaction support
export async function withTransaction<T>(
  callback: (client: PoolClient) => Promise<T>
): Promise<T> {
  const client = await pool.connect();
  try {
    await client.query('BEGIN');
    const result = await callback(client);
    await client.query('COMMIT');
    return result;
  } catch (error) {
```

Рисунок 3.2 – Приклад сервісу бази даних (аркуш 1)

```
    await client.query('ROLLBACK');
    throw error;
  } finally {
    client.release();
  }
}

// Health check
export async function checkConnection(): Promise<boolean> {
  try {
    await query('SELECT 1');
    return true;
  } catch {
    return false;
  }
}
```

Рисунок 3.2 – Приклад сервісу бази даних (аркуш 2)

3.2.2 Реалізація обробника сайтів

Обробник getSites демонструє реалізацію нумерованого списку сайтів із фільтрацією тегів (див. рис. 3.3).

```
// backend/src/handlers/sites/getSites.ts
import { APIGatewayProxyEvent, APIGatewayProxyResult } from 'aws-lambda';
import { query } from '../../services/database';
import { success, serverError } from '../../utils/response';

interface SiteRow {
  id: string;
  url: string;
  title: string;
  description: string | null;
  screenshot_full_key: string | null;
  screenshot_thumb_key: string | null;
  status: string;
  rating: string;
  view_count: number;
  favorite_count: number;
  created_at: string;
  updated_at: string;
  submitted_by_id: string | null;
  submitted_by_username: string | null;
}
```

Рисунок 3.3 – Приклад обробника запитів (аркуш 1)

```
interface TagRow {
  id: string;
  name: string;
  slug: string;
  category: string;
}

export async function handler(
  event: APIGatewayProxyEvent
): Promise<APIGatewayProxyResult> {
  try {
    // Parse query parameters with defaults
    const page = parseInt(event.queryStringParameters?.page || '1');
    const limit = Math.min(
      parseInt(event.queryStringParameters?.limit || '20'),
      100  // Maximum limit cap
    );
    const sort = event.queryStringParameters?.sort || 'newest';
    const offset = (page - 1) * limit;

    // Build ORDER BY clause based on sort parameter
    let orderBy = 's.created_at DESC';
    switch (sort) {
      case 'oldest':
        orderBy = 's.created_at ASC';
        break;
      case 'popular':
        orderBy = 's.favorite_count DESC, s.created_at DESC';
        break;
      case 'views':
        orderBy = 's.view_count DESC, s.created_at DESC';
        break;
      case 'random':
        orderBy = 'RANDOM()';
        break;
    }

    // Get total count for pagination metadata
    const countResult = await query<{ count: string }>(
      `SELECT COUNT(*) as count FROM sites s WHERE s.status = 'active'`
    );
    const total = parseInt(countResult[0].count);

    // Get sites with submitter information
    const sites = await query<SiteRow>(
      `SELECT
        s.id, s.url, s.title, s.description,
        s.screenshot_full_key, s.screenshot_thumb_key,
        s.status, s.rating, s.view_count, s.favorite_count,
        s.created_at, s.updated_at,
        u.id as submitted_by_id, u.username as submitted_by_username
```

Рисунок 3.3 – Приклад обробника запитів (аркуш 2)

```
      FROM sites s
      LEFT JOIN users u ON s.submitted_by = u.id
      WHERE s.status = 'active'
      ORDER BY ${orderBy}
      LIMIT $1 OFFSET $2`,
      [limit, offset]
    );

    // Get tags for each site using a single query
    const siteIds = sites.map(s => s.id);
    let siteTags: Record<string, TagRow[]> = {};

    if (siteIds.length > 0) {
      const tags = await query<TagRow & { site_id: string }>(
        `SELECT t.id, t.name, t.slug, t.category, st.site_id
         FROM tags t
         JOIN site_tags st ON t.id = st.tag_id
         WHERE st.site_id = ANY($1)`,
        [siteIds]
      );

      // Group tags by site_id
      siteTags = tags.reduce((acc, tag) => {
        const { site_id, ...tagData } = tag;
        if (!acc[site_id]) acc[site_id] = [];
        acc[site_id].push(tagData);
        return acc;
      }, {} as Record<string, TagRow[]>);
    }

    // Transform to response format
    const responseData = sites.map(site => ({
      id: site.id,
      url: site.url,
      title: site.title,
      description: site.description,
      thumbnailUrl: site.screenshot_thumb_key,
      screenshotUrl: site.screenshot_full_key,
      status: site.status,
      rating: site.rating,
      tags: siteTags[site.id] || [],
      stats: {
        views: site.view_count,
        favorites: site.favorite_count,
      },
      submittedBy: site.submitted_by_id ? {
        id: site.submitted_by_id,
        username: site.submitted_by_username,
      } : undefined,
      createdAt: site.created_at,
      updatedAt: site.updated_at,
    }));
```

Рисунок 3.3 – Приклад обробника запитів (аркуш 3)

```
    return success({
      data: responseData,
      meta: {
        page,
        limit,
        total,
        totalPages: Math.ceil(total / limit),
      },
    });
  } catch (error) {
    console.error('Error fetching sites:', error);
    return serverError('Failed to fetch sites');
  }
}
```

Рисунок 3.3 – Приклад обробника запитів (аркуш 4)

3.2.3 Реалізація пошуку по тегам

Обробник пошуку по тегам реалізує нечітке співставлення для функції автозаповнення (див. рис. 3.4).

```
// backend/src/handlers/tags/searchTags.ts
import { APIGatewayProxyEvent, APIGatewayProxyResult } from 'aws-lambda';
import { query } from '../../services/database';
import { success, badRequest, serverError } from '../../utils/response';

interface TagResult {
  id: string;
  name: string;
  slug: string;
  category: string;
  usage_count: number;
  similarity: number;
}

export async function handler(
  event: APIGatewayProxyEvent
): Promise<APIGatewayProxyResult> {
  try {
    const searchQuery = event.queryStringParameters?.q;
    const limit = Math.min(
      parseInt(event.queryStringParameters?.limit || '10'),
      50
```

Рисунок 3.4 – Приклад реалізації пошуку по тегах (аркуш 1)

```
    );

    if (!searchQuery || searchQuery.length < 1) {
      return badRequest('Search query is required');
    }

    // Use PostgreSQL trigram similarity for fuzzy matching
    // This requires the pg_trgm extension
    const tags = await query<TagResult>(
      `SELECT
        id, name, slug, category, usage_count,
        similarity(name, $1) as similarity
      FROM tags
      WHERE name % $1  -- Trigram similarity operator
         OR name ILIKE $2  -- Also match prefix
      ORDER BY
        similarity DESC,
        usage_count DESC
      LIMIT $3`,
      [searchQuery, `${searchQuery}%`, limit]
    );

    return success({
      data: tags.map(tag => ({
        id: tag.id,
        name: tag.name,
        slug: tag.slug,
        category: tag.category,
        usageCount: tag.usage_count,
      })),
    });
  } catch (error) {
    console.error('Error searching tags:', error);
    return serverError('Failed to search tags');
  }
}
```

Рисунок 3.4 – Приклад реалізації пошуку по тегах (аркуш 2)

3.2.4 Проміжне програмне забезпечення автентифікації

Проміжне програмне забезпечення автентифікації перевіряє токени JWT з AWS Cognito (див. рис. 3.5).

```
// backend/src/middleware/auth.ts
import { APIGatewayProxyEvent } from 'aws-lambda';
import { CognitoJwtVerifier } from 'aws-jwt-verify';
```

Рисунок 3.5 – Приклад реалізації middleware для автентифікації (аркуш 1)

```
// Configure JWT verifier for Cognito
const verifier = CognitoJwtVerifier.create({
  userPoolId: process.env.COGNITO_USER_POOL_ID!,
  tokenUse: 'access',
  clientId: process.env.COGNITO_CLIENT_ID!,
});

export interface AuthenticatedUser {
  id: string;
  email: string;
  username: string;
  role: 'user' | 'moderator' | 'admin';
}

export async function authenticateRequest(
  event: APIGatewayProxyEvent
): Promise<AuthenticatedUser | null> {
  const authHeader = event.headers.Authorization
    || event.headers.authorization;

  if (!authHeader || !authHeader.startsWith('Bearer ')) {
    return null;
  }

  const token = authHeader.substring(7);

  try {
    // Verify the JWT token
    const payload = await verifier.verify(token);

    return {
      id: payload.sub,
      email: payload.email as string,
      username: payload['cognito:username'] as string,
      role: (payload['custom:role'] as string) || 'user',
    };
  } catch (error) {
    console.error('Token verification failed:', error);
    return null;
  }
}

// Helper to require authentication
export async function requireAuth(
  event: APIGatewayProxyEvent
): Promise<AuthenticatedUser> {
  const user = await authenticateRequest(event);
  if (!user) {
    throw new Error('Authentication required');
  }
  return user;
}

// Helper to require admin role
```

Рисунок 3.5 – Приклад реалізації middleware для автентифікації (аркуш 2)

```
export async function requireAdmin(
  event: APIGatewayProxyEvent
): Promise<AuthenticatedUser> {
  const user = await requireAuth(event);
  if (user.role !== 'admin') {
    throw new Error('Admin access required');
  }
  return user;
}
```

Рисунок 3.5 – Приклад реалізації middleware для автентифікації (аркуш 3)

3.2.5 Утиліти для відповідей

Стандартизовані помічники для відповідей забезпечують узгодження відповідей API (див. рис. 3.6).

```
// backend/src/utils/response.ts
import { APIGatewayProxyResult } from 'aws-lambda';

const headers = {
  'Content-Type': 'application/json',
  'Access-Control-Allow-Origin': '*',
  'Access-Control-Allow-Headers': 'Content-Type, Authorization',
  'Access-Control-Allow-Methods': 'GET, POST, PUT, DELETE, OPTIONS',
};

export function success(
  data: unknown,
  statusCode = 200
): APIGatewayProxyResult {
  return {
    statusCode,
    headers,
    body: JSON.stringify({
      success: true,
      ...data,
    }),
  };
}

export function created(data: unknown): APIGatewayProxyResult {
  return success(data, 201);
}

export function badRequest(message: string): APIGatewayProxyResult {
```

Рисунок 3.6 – Приклад реалізації уніфікованих відповідей (аркуш 1)

```
  return {
    statusCode: 400,
    headers,
    body: JSON.stringify({
      success: false,
      error: {
        code: 'BAD_REQUEST',
        message,
      },
    }),
  };
}

export function unauthorized(
  message = 'Authentication required'
): APIGatewayProxyResult {
  return {
    statusCode: 401,
    headers,
    body: JSON.stringify({
      success: false,
      error: {
        code: 'UNAUTHORIZED',
        message,
      },
    }),
  };
}

export function forbidden(
  message = 'Access denied'
): APIGatewayProxyResult {
  return {
    statusCode: 403,
    headers,
    body: JSON.stringify({
      success: false,
      error: {
        code: 'FORBIDDEN',
        message,
      },
    }),
  };
}

export function notFound(
  message = 'Resource not found'
): APIGatewayProxyResult {
  return {
    statusCode: 404,
    headers,
    body: JSON.stringify({
```

Рисунок 3.6 – Приклад реалізації уніфікованих відповідей (аркуш 2)

```
      success: false,
      error: {
        code: 'NOT_FOUND',
        message,
      },
    }),
  };
}

export function serverError(
  message = 'Internal server error'
): APIGatewayProxyResult {
  return {
    statusCode: 500,
    headers,
    body: JSON.stringify({
      success: false,
      error: {
        code: 'SERVER_ERROR',
        message,
      },
    }),
  };
}
```

Рисунок 3.6 – Приклад реалізації уніфікованих відповідей (аркуш 3)

3.3 Розробка фронтенду

3.3.1 Точка входу в додаток

Основний файл додатку налаштовує маршрутизацію та провайдерів (див. рис. 3.7).

```
import { Routes, Route } from 'react-router-dom';
import { Layout } from './components/common/Layout';
import { HomePage } from './pages/HomePage';
import { SearchPage } from './pages/SearchPage';
import { SiteDetailPage } from './pages/SiteDetailPage';
import { TagsPage } from './pages/TagsPage';
import { LoginPage } from './pages/LoginPage';
import { RegisterPage } from './pages/RegisterPage';
import { ProfilePage } from './pages/ProfilePage';
import { SubmitSitePage } from './pages/SubmitSitePage';
```

Рисунок 3.7 – Вхід у додаток (аркуш 1)

```
import { AdminLayout } from './components/admin/AdminLayout';
import { AdminDashboard } from './pages/admin/AdminDashboard';
import { AdminSites } from './pages/admin/AdminSites';
import { AdminTags } from './pages/admin/AdminTags';
import { AdminUsers } from './pages/admin/AdminUsers';
import { ProtectedRoute } from './components/auth/ProtectedRoute';
import { AdminRoute } from './components/auth/AdminRoute';

function App() {
  return (
    <Routes>
      {/* Public routes */}
      <Route path="/" element={<Layout />}>
        <Route index element={<HomePage />} />
        <Route path="search" element={<SearchPage />} />
        <Route path="sites/:id" element={<SiteDetailPage />} />
        <Route path="tags" element={<TagsPage />} />
        <Route path="login" element={<LoginPage />} />
        <Route path="register" element={<RegisterPage />} />

        {/* Protected routes - require authentication */}
        <Route element={<ProtectedRoute />}>
          <Route path="profile" element={<ProfilePage />} />
          <Route path="submit" element={<SubmitSitePage />} />
        </Route>
      </Route>

      {/* Admin routes */}
      <Route element={<AdminRoute />}>
        <Route path="/admin" element={<AdminLayout />}>
          <Route index element={<AdminDashboard />} />
          <Route path="sites" element={<AdminSites />} />
          <Route path="tags" element={<AdminTags />} />
          <Route path="users" element={<AdminUsers />} />
        </Route>
      </Route>
    </Routes>
  );
}

export default App;
```

Рисунок 3.7 – Вхід у додаток (аркуш 2)

3.3.2 Реалізація сторінки пошуку

Компонент SearchPage демонструє керування станами та інтеграцію API (див. рис. 3.8).

```
import React, { useEffect, useCallback } from 'react';
import { useSearchParams } from 'react-router-dom';
import { SiteGrid } from '../components/sites/SiteGrid';
import { SearchBar } from '../components/search/SearchBar';
import { SearchFilters } from '../components/search/SearchFilters';
import { LoadingSpinner } from '../components/common/LoadingSpinner';
import { EmptyState } from '../components/common/EmptyState';
import { Pagination } from '../components/common/Pagination';
import { useSearchStore, SortOption } from '../store/searchStore';
import { siteService } from '../services/siteService';
import type { SiteListItem } from '@sitealike/shared';

export const SearchPage: React.FC = () => {
  const [searchParams, setSearchParams] = useSearchParams();
  const {
    query,
    filters,
    sort,
    page,
    setQuery,
    setFilters,
    setSort,
    setPage,
  } = useSearchStore();

  const [sites, setSites] = React.useState<SiteListItem[]>([]);
  const [total, setTotal] = React.useState(0);
  const [loading, setLoading] = React.useState(true);
  const limit = 24;

  // Sync URL params to store on mount
  useEffect(() => {
    const q = searchParams.get('q') || '';
    const rating = searchParams.get('rating') || '';
    const category = searchParams.get('category') || '';
    const sortParam = searchParams.get('sort') || 'relevance';
    const pageParam = parseInt(searchParams.get('page') || '1');

    setQuery(q);
    setFilters({
      rating: rating as 'safe' | 'questionable' | 'explicit' | '',
      category,
    });
    setSort(sortParam as SortOption);
    setPage(pageParam);
  }, []);

  // Update URL when store changes
  useEffect(() => {
    const params = new URLSearchParams();
    if (query) params.set('q', query);
    if (filters.rating) params.set('rating', filters.rating);
    if (filters.category) params.set('category', filters.category);
    if (sort !== 'relevance') params.set('sort', sort);
    if (page > 1) params.set('page', page.toString());
```

Рисунок 3.8 – Вхід компонент SearchPage (аркуш 1)

```
    setSearchParams(params, { replace: true });
  }, [query, filters, sort, page, setSearchParams]);

  // Fetch sites
  const fetchSites = useCallback(async () => {
    setLoading(true);
    try {
      const response = await siteService.getSites({
        q: query,
        rating: filters.rating || undefined,
        category: filters.category || undefined,
        sort,
        page,
        limit,
      });
      setSites(response.data);
      setTotal(response.meta?.total || 0);
    } catch (error) {
      console.error('Error searching sites:', error);
      setSites([]);
      setTotal(0);
    } finally {
      setLoading(false);
    }
  }, [query, filters, sort, page]);

  useEffect(() => {
    fetchSites();
  }, [fetchSites]);

  const handleSearch = (newQuery: string) => {
    setQuery(newQuery);
    setPage(1);
  };

  const totalPages = Math.ceil(total / limit);

  return (
    <div className="space-y-6">
      {/* Search Header */}
      <div className="bg-gray-800 rounded-lg p-6 border border-gray-700">
        <SearchBar
          onSearch={handleSearch}
          initialValue={query}
          placeholder="Search with tags... (e.g., react frontend -deprecated)"
        />
        <SearchFilters
          rating={filters.rating as 'safe' | 'questionable' | 'all' || 'all'}
          sort={sort}
          onRatingChange={(r) => { setFilters({ ...filters, rating: r === 'all' ?
'' : r }); setPage(1); }}
          onSortChange={(s) => { setSort(s); setPage(1); }}
        />
      </div>
```

Рисунок 3.8 – Вхід компонент SearchPage (аркуш 2)

```
      {/* Results Info */}
      <div className="flex items-center justify-between text-gray-400 text-sm">
        <span>
          {loading ? 'Searching...' : `${total} result${total !== 1 ? 's' : ''}
found`}
        </span>
        {query && (
          <button
            onClick={() => { setQuery(''); setFilters({ rating: '', category: ''
}); }}
            className="text-blue-400 hover:text-blue-300"
          >
            Clear search
          </button>
        )}
      </div>

      {/* Results */}
      {loading ? (
        <LoadingSpinner />
      ) : sites.length === 0 ? (
        <EmptyState
          title="No sites found"
          description={query ? 'Try adjusting your search terms or filters.' : 'No
sites have been added yet.'}
        />
      ) : (
        <>
          <SiteGrid sites={sites} />
          {totalPages > 1 && (
            <Pagination
              currentPage={page}
              totalPages={totalPages}
              onPageChange={setPage}
            />
          )}
        </>
      )}
    </div>
  );
};
```

Рисунок 3.8 – Вхід компонент SearchPage (аркуш 3)

3.3.3 Реалізації візуальних компонентів

Компонент SearchBar реалізує автозаповнення тегів із навігацією клавіатурою. Реалізацію коду цього компонента можно побачити у додатку А.1.

Компонент SiteCard відображає попередній перегляд сайту та трохи інформації про цей сайт в сітці результатів пошуку. Реалізація продемонстрована в додатку А.2.

3.4 Скріншоти інтерфейсу користувача

Далі наведено скріншоти сторінок додатку та деяких компонентів (див. рис. 3.9 – 3.12).

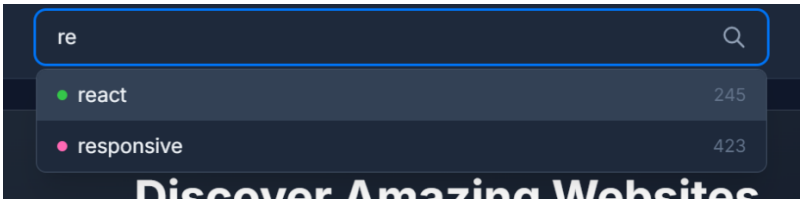


Рисунок 3.9 – Вигляд компоненту пошуку з активним автозаповненням

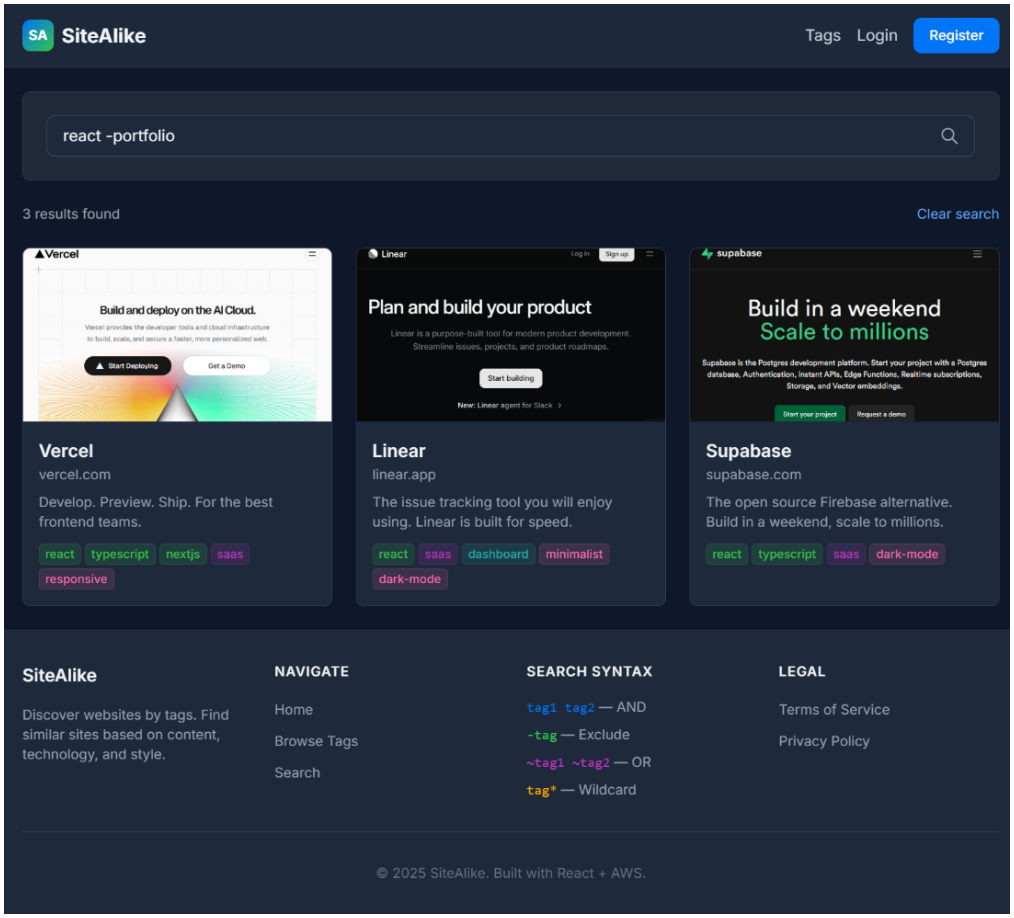


Рисунок 3.10 – Вивід результатів пошуку

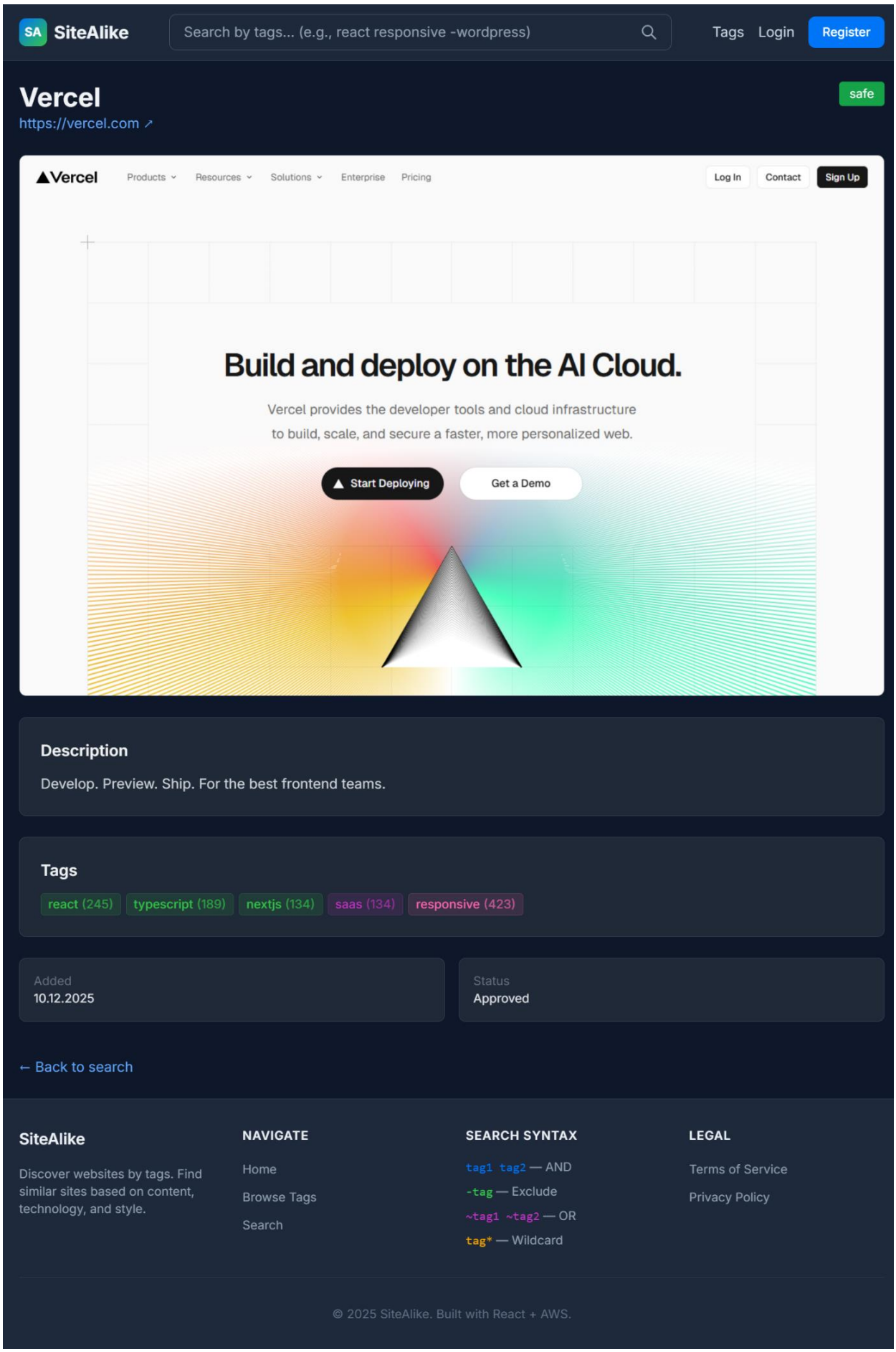


Рисунок 3.11 – Сторінка окремого сайту

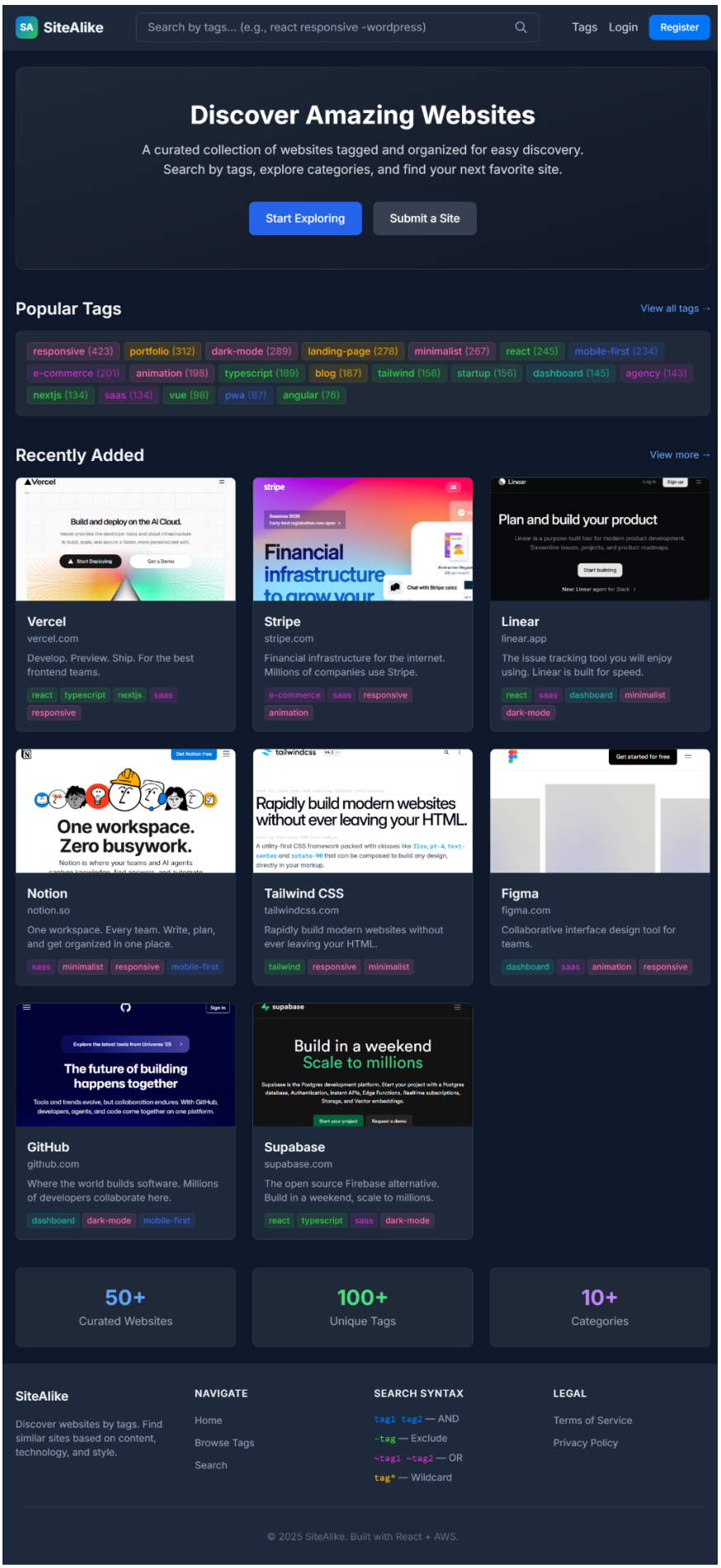


Рисунок 3.12 – Зображення домашньої сторінки

3.5 Висновки до розділу 3

Третій розділ задокументував впровадження інформаційної системи SiteAlike, охоплюючи середовище розробки, бекенд-сервіси та фронтенд-компоненти.

Монорепозиторна структура з робочими просторами npm забезпечує ефективну організацію коду та спільні залежності. Конфігурація TypeScript забезпечує безпеку типів у всіх пакетах. Безсерверна архітектура бекенду реалізує кінцеві точки RESTful API за допомогою функцій AWS Lambda. Ключові реалізації включають: сервіс баз даних із пулінгом з'єднань і підтримкою транзакцій; список сайтів із пагінацією, сортуванням і агрегацією тегів; пошук тегів за допомогою PostgreSQL trigram fuzzy matching; проміжне програмне забезпечення для JWT автентифікації із використанням AWS Cognito. React фронтенд реалізує: архітектуру на основі компонентів з багаторазовими елементами інтерфейсу; функцію пошуку з автозаповненням тегів у реальному часі; керування станом за допомогою сховищ Zustand; адаптивний дизайн з Tailwind CSS.

Програмна реалізація виконана відповідно до специфікацій наведених у Розділі 2, забезпечуючи готову до випуску систему для пошуку подібних вебсайтів.

ВИСНОВКИ

В кваліфікаційній роботі розроблено інформаційну систему для пошуку подібних вебсайтів, засобами React та Node.js.

В першому розділі наведено комплексне дослідження галузі проблеми. Аналіз показав, що пошук схожих вебсайтів – це багатовимірний виклик, що охоплює застосування конкурентної аналітики, маркетингових досліджень, SEO та розшуку контенту. Аналіз існуючих систем показав, що поточні рішення переважно орієнтовані на SEO-метрики та націлені на корпоративних користувачів за допомогою преміальних цін. Порівняльний аналіз методів виявлення схожості виправдовував вибір підходу на основі тегів з огляду на його інтерпретованість, масштабованість і потенціал внеску спільноти. Були сформульовані детальні вимоги відповідно до усталених практик розробки програмного забезпечення, а технологічний стек був обраний відповідно до галузевих тенденцій та потреб проєкту.

В другому розділі представлено архітектурну основу системи. Розроблено трирівневу архітектуру, що розділяє презентацію, бізнес-логіку та шари даних. Архітектура фронтенд-компонентів відповідає практикам React, тоді як бекенд реалізує принципи RESTful API із використанням безсерверних Lambda функцій. Дизайн інфраструктури AWS використовує керовані сервіси для масштабованості та операційної ефективності.

В третьому розділі наведено фрагменти програмної реалізації системи. Під час розробки бекенду були створені обробники Lambda функцій для всіх API з належною автентифікацією та обробкою помилок. Функція пошуку на основі тегів була реалізована з підтримкою операторів включення, виключення та нечіткого співпадіння.

Розроблена архітектура здатна підтримувати зростання від прототипу до виробничих масштабів без фундаментальних змін. Пошук на основі тегів із підтримкою операторів (+, -, ~) забезпечує виразні можливості запитів,

залишаючись при цьому інтуїтивно зрозумілим для користувачів. Процес розробки продемонстрував ефективність обраного технологічного стека для створення сучасних вебдодатків. React забезпечує продуктивне середовище для розробки складних інтерфейсів користувача, Node.js забезпечує ефективні бекенд-сервіси, а AWS пропонує комплексну платформу для масштабованого розгортання.

Розроблена система SiteAlike має практичне застосування для пошуку вебсайтів на основі конкретних характеристик, підтримуючи конкурентний аналіз та маркетингові дослідження, а інтуїтивно зрозумілий інтерфейс робить пошук сайтів доступним для користувачів без технічної експертизи.

Хоча система відповідає заявленим цілям, варто виділити кілька напрямків для майбутнього вдосконалення:

- інтеграція автоматизованого аналізу контенту за допомогою NLP-технік доповнить систему тегування на основі LLM, дозволяючи виявляти схожість навіть без попереднього тегування;
- відкриті профілі користувачів, підписки та спільне тегування підвищать залученість спільноти;
- нативні мобільні додатки розширять охоплення для користувачів, які орієнтуються на мобільні пристрої;
- підтримка багатомовності розширить потенційну базу користувачів у всьому світі.

Таким чином, інформаційна система SiteAlike успішно задовольняє виявлену потребу у доступному, точному та зручному інструменті для пошуку подібних вебсайтів.

ПЕРЕЛІК ПОСИЛАНЬ

1. A Guide to Using Google Search Operators for SEO. URL: https://www.orbitmedia.com/blog/google-search-operators-for-seo/ (дата звернення: 12.05.2025).

2. Google confirms related search operator is going away. URL: https://searchengineland.com/google-confirms-related-search-operator-is-going-away-429575 (дата звернення: 14.05.2025).

3. Detecting similar content. URL: https://www.contentside.com/en/la-detection-de-contenus-similaires/ (дата звернення: 17.05.2025).

4. Drew J.M., Moore T. Optimized combined-clustering methods for finding replicated criminal websites. *EURASIP Journal on Information Security*. 2014. Vol. 14. DOI: https://doi.org/10.1186/s13635-014-0014-4 (дата звернення: 19.05.2025).

5. Similarweb Discover Similar Sites. URL: https://support.similarweb.com/hc/en-us/articles/360019569257-Discover-Similar-Sites (дата звернення: 23.05.2025).

6. Similar Sites – Discover Related Websites. URL: https://chromewebstore.google.com/detail/similar-sites-discover-re/necpbmbhhdiplmfhmjicabdeighkndkn?hl=en (дата звернення: 24.05.2025).

7. TF-IDF – Term Frequency-Inverse Document Frequency. URL: https://www.learndatasci.com/glossary/tf-idf-term-frequency-inverse-document-frequency/ (дата звернення: 27.05.2025).

8. What is Latent Dirichlet allocation? URL: https://www.ibm.com/think/topics/latent-dirichlet-allocation (дата звернення: 04.06.2025).

9. What is Sentence Similarity? URL: https://huggingface.co/tasks/sentence-similarity (дата звернення: 09.06.2025).

10. Amazon Bedrock. URL: https://aws.amazon.com/bedrock/ (дата звернення: 04.08.2025).

11. How To Exclude Words From Google Search Results. URL: https://www.searchlogistics.com/learn/seo/exclude-words-from-google-search/#:~:text=Excluding%20Words,with%20no%20space%20in%20between (дата звернення: 10.08.2025).

12. Flanagan D. JavaScript: The Definitive Guide : 7th Edition. O'Reilly Media, 2020. URL: https://www.oreilly.com/library/view/javascript-the-definitive/9781491952016/ (дата звернення: 12.08.2025).

13. TypeScript Documentation. URL: https://www.typescriptlang.org/docs/ (дата звернення: 15.08.2025).

14. NPM.js. URL: https://docs.npmjs.com/about-npm (дата звернення: 15.08.2025).

15. Stack Overflow. Developer Survey 2025. URL: https://survey.stackoverflow.co/2025/ (дата звернення: 17.08.2025).

16. State of JavaScript. URL: https://stateofjs.com/ (дата звернення: 17.08.2025).

17. React Documentation. URL: https://react.dev/ (дата звернення: 18.08.2025).

18. What is three-tier architecture? URL: https://www.ibm.com/think/topics/three-tier-architecture (дата звернення: 23.08.2025).

19. Obe R., Hsu L. PostgreSQL: Up and Running : 3rd Edition. O'Reilly Media, 2021. URL: https://www.oreilly.com/library/view/postgresql-up-and/9781491963401/ (дата звернення: 27.08.2025).

20. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation. UC Irvine. 2000. URL: https://roy.gbiv.com/pubs/dissertation/fielding_dissertation.pdf (дата звернення: 05.11.2025).

ДОДАТОК А

Лістинг коду

А.1 Реалізація коду компонента SearchBar

```
// frontend/src/components/search/SearchBar.tsx
import { useState, useRef, useEffect } from 'react';
import { Tag, TAG_CATEGORY_COLORS } from '@sitealike/shared';
import { useDebounce } from '@/hooks/useDebounce';
import { tagService } from '@/services/tagService';

interface SearchBarProps {
  onSearch: (query: string) => void;
  initialValue?: string;
  placeholder?: string;
}

export function SearchBar({
  onSearch,
  initialValue = '',
  placeholder = 'Search by tags... (e.g., react responsive -wordpress)',
}: SearchBarProps) {
  const [query, setQuery] = useState(initialValue);
  const [suggestions, setSuggestions] = useState<Tag[]>([]);
  const [showSuggestions, setShowSuggestions] = useState(false);
  const [highlightedIndex, setHighlightedIndex] = useState(0);

  const inputRef = useRef<HTMLInputElement>(null);
  const containerRef = useRef<HTMLDivElement>(null);

  // Sync with external initialValue changes
  useEffect(() => {
    setQuery(initialValue);
  }, [initialValue]);

  // Extract the current word being typed for autocomplete
  const getCurrentWord = () => {
    const cursorPos = inputRef.current?.selectionStart || query.length;
    const beforeCursor = query.slice(0, cursorPos);
    const words = beforeCursor.split(/\s+/);
    const currentWord = words[words.length - 1] || '';
    // Remove prefix characters (-, ~) for search
    return currentWord.replace(/^[-~]/, '');
  };

  const debouncedWord = useDebounce(getCurrentWord(), 200);

  // Fetch suggestions when debounced word changes
```

```
useEffect(() => {
  if (debouncedWord.length < 1) {
    setSuggestions([]);
    return;
  }

  const fetchSuggestions = async () => {
    try {
      const results = await tagService.searchTags(debouncedWord, 6);
      setSuggestions(results);
      setHighlightedIndex(0);
    } catch {
      setSuggestions([]);
    }
  };

  fetchSuggestions();
}, [debouncedWord]);

// Close suggestions when clicking outside
useEffect(() => {
  const handleClickOutside = (event: MouseEvent) => {
    if (
      containerRef.current &&
      !containerRef.current.contains(event.target as Node)
    ) {
      setShowSuggestions(false);
    }
  };

  document.addEventListener('mousedown', handleClickOutside);
  return () => document.removeEventListener('mousedown', handleClickOutside);
}, []);

// Insert selected tag into query
const insertTag = (tagName: string) => {
  const cursorPos = inputRef.current?.selectionStart || query.length;
  const beforeCursor = query.slice(0, cursorPos);
  const afterCursor = query.slice(cursorPos);

  // Find the start of the current word
  const words = beforeCursor.split(/\s+/);
  const currentWord = words[words.length - 1] || '';
  const prefix = currentWord.match(/^[-~]/)?.[0] || '';
  const wordStart = beforeCursor.length - currentWord.length;

  const newQuery =
    query.slice(0, wordStart) +
    prefix +
    tagName +
    ' ' +
    afterCursor.trimStart();

  setQuery(newQuery);
```

```
  setSuggestions([]);
  setShowSuggestions(false);
  inputRef.current?.focus();
};

// Handle keyboard navigation
const handleKeyDown = (e: React.KeyboardEvent) => {
  if (showSuggestions && suggestions.length > 0) {
    if (e.key === 'ArrowDown') {
      e.preventDefault();
      setHighlightedIndex((prev) =>
        Math.min(prev + 1, suggestions.length - 1)
      );
    } else if (e.key === 'ArrowUp') {
      e.preventDefault();
      setHighlightedIndex((prev) => Math.max(prev - 1, 0));
    } else if (
      e.key === 'Tab' ||
      (e.key === 'Enter' && suggestions[highlightedIndex])
    ) {
      e.preventDefault();
      insertTag(
        suggestions[highlightedIndex]?.slug ||
          suggestions[highlightedIndex]?.name ||
          ''
      );
    } else if (e.key === 'Escape') {
      setShowSuggestions(false);
    }
  } else if (e.key === 'Enter') {
    e.preventDefault();
    onSearch(query);
  }
};

const handleSubmit = (e: React.FormEvent) => {
  e.preventDefault();
  onSearch(query);
  setShowSuggestions(false);
};

return (
  <div ref={containerRef} className="relative">
    <form onSubmit={handleSubmit} className="relative">
      <input
        ref={inputRef}
        type="text"
        value={query}
        onChange={(e) => {
          setQuery(e.target.value);
          setShowSuggestions(true);
        }}
        onFocus={() => setShowSuggestions(true)}
        onKeyDown={handleKeyDown}
```

```
          placeholder={placeholder}
          className="input pr-12"
        />
        <button
          type="submit"
          className="absolute right-2 top-1/2 -translate-y-1/2 p-2
                     text-surface-400 hover:text-surface-100 transition-colors"
        >
                  <svg   className="w-5   h-5"   fill="none"   viewBox="0   0   24   24"
stroke="currentColor">
            <path
              strokeLinecap="round"
              strokeLinejoin="round"
              strokeWidth={2}
              d="M21 21l-6-6m2-5a7 7 0 11-14 0 7 7 0 0114 0z"
            />
          </svg>
        </button>
      </form>

      {/* Suggestions dropdown */}
      {showSuggestions && suggestions.length > 0 && (
        <div className="absolute z-50 w-full mt-1 bg-surface-800
                        border  border-surface-600  rounded-lg  shadow-xl  overflow-
hidden">
          {suggestions.map((tag, index) => (
            <button
              key={tag.id}
              type="button"
              className={`w-full flex items-center justify-between px-4 py-2
                          text-left hover:bg-surface-700 ${
                            index === highlightedIndex ? 'bg-surface-700' : ''
                          }`}
              onClick={() => insertTag(tag.slug || tag.name)}
            >
              <span className="flex items-center gap-2">
                <span
                  className="w-2 h-2 rounded-full"
                  style={{ backgroundColor: TAG_CATEGORY_COLORS[tag.category] }}
                />
                <span className="text-surface-100">{tag.name}</span>
              </span>
              <span className="text-surface-500 text-sm">{tag.usageCount}</span>
            </button>
          ))}
        </div>
      )}
    </div>
  );
}
```

А.2 Реалізація компонента SiteCard

```
// frontend/src/components/sites/SiteCard.tsx
import { Link } from 'react-router-dom';
import { Site } from '@sitealike/shared';
import { TagChip } from '../tags/TagChip';

interface SiteCardProps {
  site: Site;
  onTagClick?: (tagSlug: string) => void;
}

export function SiteCard({ site, onTagClick }: SiteCardProps) {
  const thumbnailUrl =
    site.screenshots?.thumbnail || site.thumbnailUrl || '/placeholder-site.png';

  return (
    <article className="site-card group">
      {/* Thumbnail */}
      <Link
        to={`/sites/${site.id}`}
        className="block relative aspect-video overflow-hidden"
      >
        <img
          src={thumbnailUrl}
          alt={`Screenshot of ${site.title}`}
          className="w-full h-full object-cover transition-transform
                     group-hover:scale-105"
          loading="lazy"
        />
        {/* Overlay on hover */}
        <div
          className="absolute inset-0 bg-gradient-to-t from-black/60
                     via-transparent to-transparent opacity-0
                     group-hover:opacity-100 transition-opacity"
        />
      </Link>

      {/* Content */}
      <div className="p-4">
        {/* Title and URL */}
        <div className="mb-2">
          <Link to={`/sites/${site.id}`} className="block">
            <h3 className="text-lg font-semibold text-surface-100 truncate
                           group-hover:text-primary-400 transition-colors">
              {site.title}
            </h3>
          </Link>
          <a
            href={site.url}
            target="_blank"
            rel="noopener noreferrer"
```

```
            className="text-sm text-surface-400 hover:text-primary-400
                       truncate block"
          >
            {new URL(site.url).hostname}
          </a>
        </div>

        {/* Description */}
        {site.description && (
          <p className="text-sm text-surface-400 line-clamp-2 mb-3">
            {site.description}
          </p>
        )}

        {/* Tags */}
        <div className="flex flex-wrap gap-1">
          {site.tags.slice(0, 5).map((tag) => (
            <TagChip
              key={tag.id}
              name={tag.name}
              category={tag.category}
              size="sm"
              onClick={() => tag.slug && onTagClick?.(tag.slug)}
            />
          ))}
          {site.tags.length > 5 && (
            <span className="text-xs text-surface-500 self-center">
              +{site.tags.length - 5} more
            </span>
          )}
        </div>
      </div>
    </article>
  );
}
```