Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра програмної інженерії
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

магістр
(назва освітнього ступеня)

на тему: Проєктування та розробка атоматизованої системи генерації UI компонентів для веб застосунків на основі великих мовних моделей

Виконав(ла): студент(ка) 6 курсу, групи СПм-61
спеціальності 123 Інженерія програмного забезпечення
(шифр і назва спеціальності)

_____ Глух О.М
(підпис) (прізвище та ініціали)

Керівник _____ Мудрик І.Я
(підпис) (прізвище та ініціали)

Нормоконтроль _____ Стоянов Ю. М.
(підпис) (прізвище та ініціали)

Завідувач кафедри _____ Петрик М.Р
(підпис) (прізвище та ініціали)

Рецензент _____ Луцик Н.С
(підпис) (прізвище та ініціали)

Тернопіль
20___

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет Компютерно-іформаційних систем і програмної інженерія
(повна назва факультету)

Кафедра Програмної інженерія
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Петрик М. Р
(підпис) (прізвище та ініціали)
« » 20___ р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня магістр
(назва освітнього ступеня)

за спеціальністю 121 «Інженерія програмного забезпечення»
(шифр і назва спеціальності)

студенту Глуху Олегу Миколайовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Проєктування та розробка атоматизованої системи генерації UI компонентів для веб застосунків на основі велеких мовних моделей

Керівник роботи Мудрик Іван Ярославович докток філософії, доцент кафедри ПІ
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «___» _____ 20___ року № _____

2. Термін подання студентом завершеної роботи

3. Вихідні дані до роботи Вимоги до генерації UI-компонентів для вебзастосунків, опис великих мовних моделей та параметри їх використання, вхідні текстові запити користувачів, формати вихідних UI-компонентів; середовище розробки.

4. Зміст роботи (перелік питань, які потрібно розробити)
Аналіз предметної області та існуючих рішень; постановка задачі та формування вимог до системи; проєктування архітектури та структури даних; реалізація модулів генерації UI-компонентів; інтеграція мовних моделей; тестування та аналіз результатів роботи системи

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)

6. Консультанти розділів роботи

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

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Збір і аналіз наукових та нормативних джерел,	28.11.2025	**Виконано**
	уточнення теми, мети, задач і предмета дослідження.		
2	Аналіз предметної області, огляд існуючих рішень та	30.11.2025	**Виконано**
	формування вимог до системи.		
3	Проєктування системи: розроблення архітектури та	02.12.2025	**Виконано**
	структури даних.		
4	Розробка програмного забезпечення автоматизованої	04.12.2025	**Виконано**
	системи генерації UI-компонентів		
5	Налаштування та впровадження системи, інтеграція	06.12.2025	**Виконано**
	великих мовних моделей.		
6	Тестування системи за основними сценаріями	08.12.2025	**Виконано**
	використання, аналіз результатів.		
7	Оцінка ефективності розробленої системи, формування	10.12.2025	**Виконано**
	висновків і рекомендацій.		
8	Підготовка розділу охорони праці та безпеки в	11.12.2025	**Виконано**
	надзвичайних ситуаціях.		
9	Оформлення пояснювальної записки, списку	12.12.2025	**Виконано**
	використаних джерел та графічних матеріалів.		
10	Захист кваліфікаційної роботи.	22.12.2025	

Студент _____ Глух О.М
(підпис) (прізвище та ініціали)

Керівник роботи _____ Мудрик І.Я.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Глух Олег Миколайович. Розробка системи автоматичної генерації UI-компонентів для вебзастосунків на основі великих мовних моделей. Кваліфікаційна робота освітнього ступеня «магістр». Тернопільський національний технічний університет імені Івана Пулюя, кафедра програмної інженерії, група СПм-61, спеціальність 121 «Інженерія програмного забезпечення». Тернопіль, 2025. Сторінок 81, таблиць 2, рисунків 26, додатків 4, бібліографічних посилань .

Метою роботи є розробка веборієнтованої системи автоматичної генерації користувацьких інтерфейсних компонентів для вебзастосунків на основі текстового опису вимог із використанням великих мовних моделей.

Об’єктом дослідження є процес проєктування та створення UI-компонентів у сучасній веброзробці. Предметом дослідження є методи та програмні засоби автоматизації генерації інтерфейсних компонентів із застосуванням великих мовних моделей, вебтехнологій та інструментів фронтенд- і бекенд-розробки.

У роботі виконано аналіз предметної області та існуючих рішень у сфері AI-асистованої веброзробки, сформульовано функціональні та нефункціональні вимоги до системи, спроєктовано архітектуру та модель даних. Реалізовано прототип системи, що забезпечує обробку текстових запитів, взаємодію з мовними моделями, генерацію структурованого набору файлів UI-компонента та їх подальший перегляд у вебінтерфейсі. Описано процес впровадження та тестування розробленої системи, а також оцінено її практичну придатність. Окремо розглянуто питання безпеки та особливості роботи користувачів із вебзастосунком.

Ключові слова: веброзробка, UI-компоненти, великі мовні моделі, штучний інтелект, автоматична генерація коду, вебзастосунок, програмна інженерія, інтерфейс користувача, генерація компонентів, AI-асистована розробка, програмні системи, вебтехнології.

ABSTRACT

Hlukh Oleh Mykolaiovych. Development of an Automated UI Component Generation System for Web Applications Based on Large Language Models. Master's qualification work. Ternopil Ivan Puluj National Technical University, Department of Software Engineering, group SPm-61, specialty 121 "Software Engineering". Ternopil, 2025. Pages 81, tables 2, figures 26, appendices 4, references 39.

The aim of the work is to develop a web-oriented system for automatic generation of user interface components for web applications based on textual requirement descriptions using large language models.

The object of the study is the process of designing and developing UI components in modern web development. The subject of the study is methods and software tools for automating UI component generation using large language models, web technologies, and frontend and backend development frameworks.

The paper analyzes the subject area and existing solutions in the field of AI-assisted web development, formulates functional and non-functional system requirements, and designs the system architecture and data model. A prototype system has been implemented that provides processing of textual prompts, interaction with large language models, generation of a structured set of UI component files, and their subsequent preview in a web interface. The process of system deployment and testing is described, and the practical applicability of the proposed solution is evaluated. Particular attention is paid to security aspects and user interaction with the web application.

Keywords: web development, UI components, large language models, artificial intelligence, automatic code generation, web application, software engineering, user interface, component generation, AI-assisted development, software systems, web technologies.

ЗМІСТ

ВСТУП

Високий ріст веб-технологій поступово змінює спосіб, яким ми проектуємо інтерфейси, і користувачі очікують, що вони не лише добре працюватимуть, але й будуть стабільними, зручними та масштабованими. Веб-додаток може мати багато компонентів, і ці компоненти повинні природно співіснувати в системі дизайну, бути адаптивними та універсальними для повторного використання. Підтримка такої комбінації речей у порядку не завжди є легкою: це може зайняти час і вимагати постійних зусиль від розробників.

Процес створення частин компонентів зазвичай є дуже ручним – від написання коду та макету до продуманого стилю та створення коротких прикладів використання. Це також додає додаткове навантаження на команду, крім вже існуючого навантаження, а також схильність до надмірної зайнятості та ризику незначних розбіжностей між реалізаціями. Інша тенденція, яка зростає в останні роки, але також існує схожа тенденція агресивне застосування великих мовних моделей за останні кілька років. І ці системи навчилися аналізувати контекст – вони можуть створювати код, пропонувати нові варіанти та адаптувати його залежно від правил. У сфері створення компонентів інтерфейсу вони дозволяють зекономити години роботи на регулярних завданнях і покращити узгодженість інтерфейсу. Це особливо помітно у великих проектах, де вимоги до стилю повинні бути повністю задоволені, і будь-яке відхилення від основного дизайну не залишається непоміченим.

Крім того, сучасні команди працюють дуже швидко: ітерація та випуск, адаптація до змін з гнучкістю. У таких випадках автоматичне створення компонентів може допомогти зменшити навантаження на розробників і дозволити їм присвятити більше часу логіці додатків замість своїх завдань. Інша можливість – це закриття розриву між дизайном та реалізацією. Коли процеси налагоджені, самі артефакти дизайну можуть бути майже автоматично переведені в код.

Метою цієї кваліфікаційної роботи є встановлення та формулювання рамок для створення системи автоматичного створення компонентів інтерфейсу для веб-додатків з використанням великих мовних моделей. Система повинна вміти моделювати текстові описи, конфігурації та матеріали дизайну і перетворювати їх у типізовані компоненти, які можуть бути безпосередньо інтегровані в сучасні фронтенд-фреймворки.Для досягнення зазначеної мети у роботі поставлено такі завдання:

- Проаналізувати стан предметної області, сучасні підходи до проєктування UI та інструменти автоматизованої генерації інтерфейсів;визначити вимоги до системи та 2. Сформувати архітектуру, що враховує специфіку використання LLM у процесі генерації коду.
- Розробити модель трансформації специфікацій інтерфейсу у програмний код компонентів з урахуванням типізації, структурних залежностей та стилістичних вимог.
- Реалізувати серверну та клієнтську частину системи, включно з модулем взаємодії з LLM та інтерфейсом для формування запитів і перегляду результатів;
- Здійснити тестування розробленої системи, оцінити якість, структурну коректність та відповідність згенерованих компонентів вимогам дизайн-системи;
- Визначити напрями подальшого розвитку системи та можливості її інтеграції у процес промислової веб-розробки.

Об'єктом дослідження є автоматична генерація користувацьких інтерфейсів у веб-додатках. Дослідження зосереджено на методах і моделях застосування великих мовних моделей для синтезу типізованих компонентів інтерфейсу на основі текстових і структурованих специфікацій.

Наукова новизна цієї роботи полягає в розробці систематичного підходу до автоматизованої генерації компонентів інтерфейсу, який інтегрує принципи програмної інженерії, систем дизайну та можливості великих мовних моделей.

Запропонована архітектура враховує контекст дизайну, а також типізований код (TypeScript), і також пропонує можливість розширення генерації шляхом додавання нових шаблонів або компонентів без зміни фундаментальної логіки. Важливою характеристикою є гнучкість генерації: можливо адаптувати модель для створення різних варіантів одного компонента у відповідь на бізнес-потреби, параметри інтерфейсу та інтерпретацію артефактів дизайну.

Практична цінність цієї роботи полягає в тому, що вона дозволить застосовувати розроблену систему в розробці веб-продуктів для автоматизації рутинних операцій, прискорення створення компонентів, забезпечення узгодженості компонентів та підтримки стандартів системи дизайну. Рішення може бути використане в командах різного масштабу, інтегроване в CI/CD або використане для створення прототипів, документації та навчальних матеріалів.

1 АНАЛІЗ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

У першому розділі здійснено аналіз сучасного стану проблеми автоматизації розроблення інтерфейсних компонентів у вебзастосунках із використанням великих мовних моделей. Розглянуто основні підходи, інструменти та обмеження існуючих рішень, а також визначено передумови застосування штучного інтелекту в процесах UI-розробки. За результатами аналізу сформульовано вимоги до програмної системи та обґрунтовано актуальність обраної теми дослідження.

1.1. Аналіз предметної області автоматизованої генерації UI-компонентів

Інтерфейси користувача відіграють важливу роль у сучасній веб-розробці. Якість зовнішнього вигляду та ефективність веб-додатку, що значно впливає на його конкурентоспроможність, значною мірою залежить від того, наскільки швидко та легко можна отримати доступ до інтерфейсу [26, 27]. Сучасні веб-системи зазвичай розробляються з використанням компонентного підходу [1, 3]. Можна класифікувати на елементи з точки зору структури, поведінки, набору властивостей та стилів, що сприяє їх стабільності та повторному використанню в різних областях додатку.

Незважаючи на популярність компонентног підходу та наявність безлічі фреймворків (React, Vue, Angular), створення компонентів інтерфейсу все ще є трудомістким процесом. Це вимагатиме від розробника написання структури компонента, визначення інтерфейсів даних, реалізації логіки, забезпечення стилізації відповідно до системи дизайну та створення документації для подальшої підтримких. Багато з цих завдань є загальними та монотонними, що збільшує кількість помилок, складність підтримки коду, а також продуктивність команди [1, 2].

Забезпечення узгодженості інтерфейсів у великих проєктах також є проблемою. [3, 4]. Розробники зазвичай мають десятки та сотні компонентів, які потрібно підтримувати в одному стилі та з узгодженою робочою поведінкою. Будь-які зміни в

системі дизайну вимагають значних інвестицій у їх поширення на всі компоненти, що призводить до уповільнення розробки продукту. Існує додатковий рівень складності, коли робота між дизайнерами та розробниками координується, а макет кодується вручну. У цьому відношенні потрібні технології для автоматизації дизайну елементів інтерфейсу на основі формальних описів, специфікацій дизайну або текстових вимог.

Використовуючи моделі, що генерують високоякісний програмний код на основі природної мови або структурованих запитів, однією з найперспективніших відповідей є нове застосування великих мовних моделей [6, 7]. Моделі, такі як GPT, Claude та інші сучасні LLM, демонструють високий рівень розуміння контексту, відповідність компонентній архітектурі, здатність відтворювати стилістичні особливості та підтримку набору мов програмування (TypeScript). Це також може значно скоротити час розробки та підвищити якість коду за рахунок автоматизації генерації компонентів інтерфейсу з великими мовними моделями. На відміну від традиційних методів, LLM можуть виконувати більшу частину роботи, отримуючи специфікацію з дружнім до користувача поясненням, а не покладаючись на розробника, щоб він самостійно виконав це. Відповідно, мовна модель служить зв'язком між артефактами дизайну, специфікаціями інтерфейсу та кінцевою версією реалізації. І, звичайно, ця автоматизація є найбільш корисною для проектів, що включають багато схожих будівельних блоків, складних варіацій стану або дуже високих потреб уніфікації, де великий рівень автоматизації є перевагою. Такі системи є невід'ємною частиною сучасних практик життєвого циклу програмного забезпечення, що є ключовим фокусом предметної області.

Розробка автоматично згенерованих компонентів інтерфейсу, звичайно, повинна відповідати якості коду, що приймається в команді [4, 5], і повинна підтримувати інтеграцію з інструментами CI/CD, бути здатною взаємодіяти з бібліотеками тестування або бути здатною змінюватися будь-яким чином людиною. Особливо важливо в цьому відношенні зберігати код організованим та читабельним, оскільки компоненти повинні легко перепрофільовуватися та додаватися таким же чином.

Область автоматичної генерації інтерфейсу також схожа на систему дизайну, концепцію токенів дизайну та уніфікацію стилів у великих підприємствах. Генеративні системи повинні враховувати правила стилю та типографіку; вибір кольорів та палітри; розміри відступів та шаблони поведінки, а написання коду з чітких текстових описів є складнішим, ніж генерація коду вручну. Автоматизована система не може просто розуміти синтаксис, але й інтерпретувати семантику дизайнерських рішень. Додатковим компонентом предметної області є якість коду, що генерується автоматично. Лінтери, аналізатори, компоненти документації, типізація та тести є одними з найважливіших факторів для стабільного та надійного завершення проекту.

Автоматизована генерація повинна виробляти не тільки код, але й повинна генеруватися відповідно до прийнятих стандартів, які можуть бути протестовані та перевірені автоматично. Крім того, необхідно розуміти ризики та обмеження, пов'язані з генеративними технологіями штучного інтелекту. Мовні моделі можуть створювати неправильні структури компонентів [11], конфліктуючі залежності, дублюючу поведінку або невідповідності з системою дизайну. Тому автоматизована система повинна мати засоби для перевірки вхідних даних та постобробки результату коду, щоб зменшити кількість помилок та підвищити довіру до результату генерації.

Таким чином, предметна область автоматизованої генерації компонентів інтерфейсу охоплює широкий спектр методів та інструментів для підвищення ефективності створення інтерфейсів, їх стандартизації та оптимізації виробництва. Її розвиток нерозривно пов'язаний із сучасними тенденціями у веб-розробці, дизайні системи дизайну та застосуванням інтелектуальних технологій разом у процесі інженерії програмного забезпечення. Отже, буде визначено актуальність дослідження та впровадження систем, здатних автоматично генерувати високоякісні компоненти інтерфейсу, які відповідають технічним та функціональним вимогам сучасних веб-додатків.

У контексті автоматизованої генерації компонентів інтерфейсу залишається з'ясувати, яку роль вивчена система відіграє на етапі розробки та її взаємодію з іншими

сервісами та інструментами. На рисунку 1.1 показана контекстна діаграма для ілюстрації акторів та комунікації між основними гравцями в предметній області. Розробник подає запит на створення компонента, вказує специфікації для нього та отримує згенерований код для подальшої інтеграції з веб-додатком. Дизайнер надає специфікації інтерфейсу або описи компонентів, які можуть слугувати структурованим вхідним даними для системи. Автоматизована система обробляє запити, формує підказки для великої мовної моделі та, нарешті, отримує згенерований код елемента інтерфейсу. Модель штучного інтелекту повертає результат генерації, який потім обробляється, перевіряється та може бути завантажений у репозиторій коду. Система також приймає дані токенів дизайну або елементи системи дизайну для забезпечення стилістичної узгодженості з інтерфейсом додатку.

Наведена діаграма демонструє загальний контекст функціонування системи та дозволяє формально визначити основні зовнішні залежності, що є важливою частиною аналізу предметної.

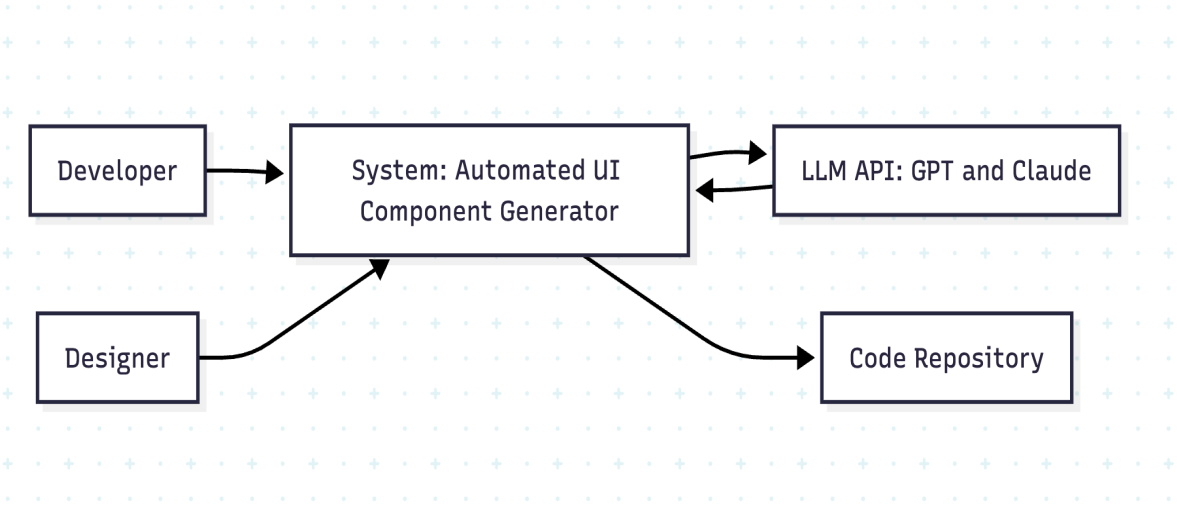


Рисунок 1.1 – Контекстна діаграма системи автоматизованої генерації UI-компонентів

1.2 Огляд сучасних рішень у сфері автоматизованої генерації інтерфейсів

Автоматизація процесів, що стоять за створенням користувацького інтерфейсу, є дуже динамічною галуззю, яка швидко розвивається в останні роки, завдяки великим мовним моделям та інструментам машинного навчання. На сучасному ринку доступні кілька рішень для часткової або повної автоматизації генерації компонентів інтерфейсу користувача, проте жодне з них не надає універсального та гнучкого підходу, що підходить для промислових веб-проєктів. Давайте розглянемо найпоширеніші технології та інструменти, які визначають поточний стан розвитку в цій галузі.

Builder.io – це платформа, у якій поєднано візуальний конструктор та елементи машинного навчання, що використовуються для формування коду інтерфейсів. Робота з системою відбувається через створення блоків, які потім можуть бути перетворені у код для різних фреймворків, зокрема React чи Vue. Хоча такий підхід справді спрощує процес побудови інтерфейсу, можливості Builder.io все ж обмежені: значна частина логіки генерації залишається «всередині» системи, і розробник не завжди має повний контроль над тим, яким буде кінцевий код. Для проєктів, де важливо дотримуватися усталених стандартів або внутрішніх вимог до коду, така залежність може стати перешкодою.

Схожим за напрямом є інструмент Mitosis – проєкт, що дозволяє формувати компоненти інтерфейсу у вигляді абстрактного опису й далі трансформувати його у код для різних фреймворків. Mitosis пропонує цікаву можливість забезпечити певну сумісність між платформами, що може бути корисним у багатоцільових системах. Однак він не працює з текстовими вимогами чи семантичними описами для автоматичної генерації. Базова структура компонента здебільшого створюється самою людиною, а Mitosis уже відповідає за її перетворення. Тому інструмент розв’язує лише окремий фрагмент ширшої задачі автоматизованої генерації UI-компонентів.

У 2023-2024 роках з’явилися генератори UI на основі великих мовних моделей, такі як Galileo AI, v0.dev та Bolt.new. Galileo AI дозволяє конвертувати дизайн-макети у програмний код компонентів, однак результати генерації залежать від якості розпізнавання макетів та можуть потребувати значного ручного доопрацювання. Платформа v0.dev від компанії Vercel позиціонує себе як універсальний AI-генератор веб-інтерфейсів. Вона створює React-компоненти на основі текстового запиту, проте кінцевий код часто не відповідає специфічним вимогам конкретного проєкту, а система не враховує дизайн-токени, структуру проєкту або індивідуальні правила типізації. Bolt.new пропонує швидке прототипування інтерфейсів за допомогою інтерактивного чат-інтерфейсу, але генерований код є фрагментарним і призначений переважно для демонстраційних проєктів.

Окремої уваги заслуговують інструменти на основі reverse-engineering макетів, серед яких Figma-to-Code генератори (Anima, Locofy, TeleportHQ). Вони намагаються автоматизувати перехід від дизайнерських макетів до коду UI-компонентів, однак такі системи часто залежать від якості розмітки у Figma, не завжди коректно визначають вкладені структури та стилі, а також генерують надлишковий або дубльований код. Генератори цього класу рідко враховують логічні правила компонентної архітектури, тому без суттєвого доопрацювання вони непридатні для використання у професійних продуктових командах.

Універсальні великі мовні моделі (GPT-4/5, Claude 3, Gemini) демонструють високу здатність до генерації структурованого коду та розуміння вимог до UI, проте їх пряме використання без допоміжних механізмів призводить до низки проблем. Модель може генерувати компоненти, які не відповідають стилістичним правилам, не узгоджуються з дизайн-системою або не дотримуються визначеної структури властивостей. Відсутність інваріантності до контексту проєкту та непослідовність у форматуванні коду є суттєвими перешкодами у промисловому застосуванні LLM. Саме тому існує потреба у системах, які не лише надсилають запит до моделі, але й

контролюють структуру, синтаксис, форматування та семантику згенерованого компонента.

Проведений огляд дозволяє виділити кілька ключових недоліків існуючих рішень, які визначають необхідність створення окремої спеціалізованої системи. По-перше, більшість AI-генераторів не забезпечують повної інтеграції з дизайн-системами та їх токенами, що є критичним аспектом для уніфікації інтерфейсів. По-друге, генератори не враховують архітектуру конкретного веб-проєкту, включно зі структурою директорій, підходами до типізації або правилами написання коду. По-третє, наявні рішення не виконують валідацію і не гарантують коректність отриманих компонентів з точки зору відповідності внутрішнім стандартам команди розробників. Нарешті, багато інструментів є або надто загальними, або жорстко прив’язаними до конкретних платформ, що обмежує їх використання у складних проєктах.

Таким чином, існуючі тенденції демонструють високу зацікавленість у застосуванні генеративних підходів для створення UI-компонентів, але доступні інструменти не забезпечують комплексного вирішення поставленої задачі. Вони або не адаптовані до індивідуальних вимог, або не гарантують стабільної якості згенерованого коду, або не підтримують повний спектр функціональних потреб сучасних веб-проєктів. Це підкреслює актуальність розроблення спеціалізованої автоматизованої системи, яка враховує контекст проєкту, підтримує стандартизацію та забезпечує стабільну якість UI-компонентів завдяки поєднанню архітектурних рішень із можливостями великих мовних моделей [11, 41].

1.3 Вимоги до системи автоматизованої генерації UI-компонентів

Системи автоматизованого генерування UI-компонентів повинні визначати будівельні блоки своїх операцій, тип взаємодії з користувачем та вимоги, а потім визначати специфіку інтеграції із зовнішніми системами (такими як великі мовні моделі та системи управління стилями). На відміну від традиційних рішень, які

генерують лише часткове створення інтерфейсу, запропонований продукт повинен пропонувати повний життєвий цикл створення контенту: від отримання описів та параметрів до повного програмного артефакту, який може бути інтегрований у веб-додаток

Підтримка авторизації користувачів є однією з вимог. Це необхідно для персоналізації історії генерації, захисту внутрішніх даних та обмеження доступу до функцій системи. Авторизований користувач повинен мати можливість працювати зі своїми попередніми результатами, переглядати історію генерації, повторно відкривати або змінювати створені компоненти та завантажувати згенеровані файли.

Основною функціональною можливістю системи є генерація UI-компонентів на основі текстового опису або структурованої конфігурації. Користувач задає вимоги до компонента, після чого система формує відповідний запит до великої мовної моделі. Отриманий код проходить етап структурування та валідації, що забезпечує стабільність та відповідність внутрішнім стандартам проєкту. Оскільки результати генерації мають бути інтегровані у реальні робочі процеси, система повинна забезпечувати збереження компонента у файлову структуру, включно з основним кодом, типами, стилями та супровідною документацією. Створені файли можуть експортуватися у форматі ZIP для швидкого перенесення у робоче середовище.

Узагальнений перелік основних функціональних характеристик системи наведено у табл. 1.1, де відображено вимоги, що визначають логіку її роботи, включаючи авторизацію, процес генерації, взаємодію з великою мовною моделлю та можливість збереження результатів у файли.

Нефункціональні вимоги описують якісні характеристики системи: її продуктивність, зручність використання, безпеку, масштабованість та надійність. Вони визначають міри, які не впливають безпосередньо на алгоритм генерації, але визначають стабільність та ефективність роботи системи в реальному середовищі. Такі вимоги систематично описані в Таблиці 1.2. Особливо важливою є необхідність підтримки читабельності та підтримуваності згенерованого коду [43], оскільки

компоненти інтерфейсу користувача зазвичай належать до великих веб-проєктів і можуть бути вручну змінені після генерації [5].

Таблиця 1.1 – Основні функціональні вимоги.

№	Функціональна вимога	Опис
1	Генерація UI-компонентів	Система повинна генерувати код UI-компонентів на основі текстового опису, параметрів або структурованої специфікації.
2	Формування пронтів для LLM	Система повинна автоматично формувати оптимізовані запити до мовної моделі на основі введених користувачем даних.
3	Отримання відповіді від LLM	Система повинна отримувати результат генерації коду від великої мовної моделі та виконувати його синтаксичну та структурну валідацію.
4	Перегляд та редагування результату	Користувач повинен мати можливість переглянути, відредагувати або повторно згенерувати компонент.
5	Інтеграція з дизайн-токенами	Система повинна враховувати стилістичні правила дизайн-системи, використовуючи токени шрифтів, кольорів та відступів
6	Експорт та збереження коду	Генерований код повинен підтримувати експорт у файлову структуру проєкту або репозиторій.
7	Повторна генерація	Система повинна дозволяти запуск повторної генерації з урахуванням попередніх результатів та виправлень.
8	Генерація супровідної документації	Система має забезпечувати формування опису компонентів, включно з прикладами використання та пропсами.

Нефункціональні вимоги описують характеристики системи, пов’язані з її ефективністю, якістю, безпекою та масштабованістю.

Таблиця 1.2 – Основні нефункціональні вимоги.

№	Категорія	Вимога
1	Продуктивність	Час генерації компонента не повинен перевищувати 2–5 секунд при стабільному з’єднанні з LLM API.
	Надійність	Система повинна забезпечувати обробку некоректних промптів і повернення повідомлень про помилки.
	Масштабованість	Система повинна підтримувати паралельні запити користувачів та адаптуватися до зростання навантаження.
	Якість коду	Згенеровані компоненти повинні відповідати вимогам TypeScript, обраним правилам лінтингу та стилю проєкту.
	Зручність використання	Інтерфейс повинен бути зрозумілим користувачеві та вимагати мінімум дій для отримання результату.
	Безпека	Ключі доступу до LLM API повинні бути захищені; дані не повинні зберігатися у незашифрованому вигляд
	Адаптивність	Система повинна підтримувати різні архітектурні шаблони компонентів залежно від проєкту.

Для узагальнення функціональних можливостей системи доцільно подати їх у вигляді діаграми варіантів використання. На рис. 1.2 наведено основні сценарії взаємодії користувачів із системою, включно з авторизацією, генерацією компонентів, переглядом та редагуванням результатів, експортом сформованих файлів та роботою з

історією генерацій. Дизайнер виступає джерелом специфікацій, що використовуються системою для формування промптів до великої мовної моделі.

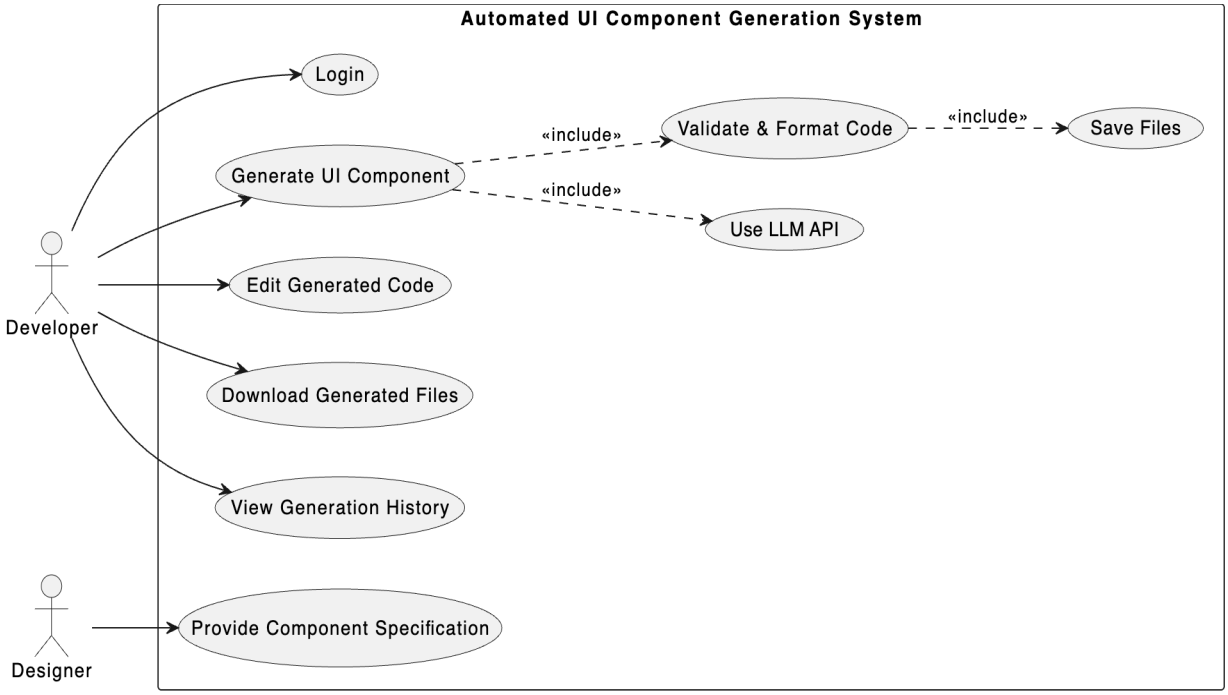


Рисунок 1.2 – Діаграма варіантів використання системи автоматизованої генерації UI-компонентів

Модель «вхід–вихід» системи подана на рис. 1.3. Вона відображає узагальнену послідовність перетворення даних під час генерації компонента: від введення користувачем опису або специфікації до формування завершеної файлової структури UI-елемента. Модель ілюструє основні етапи роботи системи, включно з попередньою обробкою даних, формуванням запиту до великої мовної моделі, отриманням та валідацією результату, а також збереженням згенерованих файлів.

Рисунок 1.3 – Модель “вхід–вихід” системи автоматизованої генерації UI-компонентів.

Логіку функціонування системи доповнює структурно-логічна схема, наведена на рис. 1.4. Схема демонструє основні процеси, що відбуваються під час роботи: авторизацію користувача, подання опису компонента, формування запиту до мовної моделі, обробку отриманого коду, створення файлової структури та збереження результатів. Такий підхід дозволяє узагальнити внутрішню організацію роботи системи та створює основу для подальшого проєктування її архітектури.

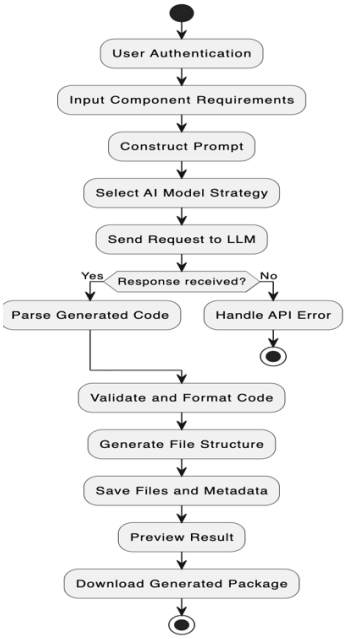


Рисунок 1.4 – Структурно-логічна схема роботи системи

Критерії прийнятності визначають вимоги до якості результатів генерації. Згенерований компонент має коректно компілюватися, відповідати параметрам, зазначеним у специфікації, містити повну файлову структуру та дотримуватися прийнятих стандартів типізації й стилю коду. Виконання цих критеріїв забезпечує можливість подальшої інтеграції компонентів у веб-проєкти.

1.4 Концептуальні моделі системи

Розуміння передбачуваної системи має бути отримане шляхом побудови різних концептуальних моделей того, як система повинна працювати, взаємодії з користувачем та внутрішньої обробки даних для отримання остаточної моделі дизайну. Ці моделі є корисним внеском на рівні дизайну, оскільки вони допомагають пояснити логічну структуру системи, де розташовані її основні елементи, і обґрунтувати вибір архітектури, яку слід враховувати в наступній частині роботи [3].

Корисним інструментом для характеристики функціонування програмних систем є діаграма випадків використання. Ця діаграма на Рисунку 1.2 показує, як зображені основні сценарії взаємодії користувача з системою автоматизованого генерування компонентів інтерфейсу користувача. Основними учасниками є розробник і дизайнер. Розробник аутентифікується, вводить опис компонента, починає процес його створення, переглядає та редагує згенерований результат, зберігає згенеровані файли або завантажує їх як архів. Альтернативно, дизайнер подає специфікацію або опис елемента інтерфейсу, який потім є частиною вхідних даних для системи. Представлена діаграма допомагає визначити обов'язки самої системи, демонструє функціональні можливості системи та взаємозв'язок користувачів з основними сценаріями використання [1, 2].

Одним із важливих аспектів функціонування системи є перетворення вхідних даних у готовий набір файлів інтерфейсного компонента. Цей процес ілюструє модель

«вхід–вихід», подана на рис. 1.3. У ній наведено узагальнену послідовність обробки даних: починаючи з моменту, коли користувач вводить опис або конфігурацію компонента, і завершуючи формуванням завершеної структури файлів. Модель включає такі етапи, як попередня обробка даних, формування запиту (промпта) для мовної моделі, отримання результату, його валідація, форматування та збереження у відповідному форматі. Подібне представлення дозволяє узагальнити інформаційні потоки, визначити ключові точки взаємодії із зовнішніми сервісами (зокрема LLM API), а також встановити логічні зв’язки між процесами. Такий підхід є особливо корисним для подальшої декомпозиції системи та визначення її програмних модулів.

Структурно-логічна схема, для повної картини, представлена (Рисунок 1.4), пояснює внутрішню логіку системи. На відміну від моделі вводу-виводу, яка лише демонструє основні етапи перетворення даних, ця схема відображає послідовність операцій, що забезпечують загальне функціонування системи. Вона потім вводить авторизацію користувача, початкове введення даних, побудову запиту, відправку запиту до великої мовної моделі, обробку отриманого результату, створення файлової структури, збереження матеріалів та запис результатів в історію генерації. Схема демонструє загальну логіку процесів, взаємозв'язки між ними та їх послідовності – необхідну передумову для розробки архітектури системи, описаної в Розділі 2.

Комплексне використання різних типів концептуальних моделей дозволяє сформувати цілісне та структуроване розуміння програмної системи ще до її введення в експлуатацію. Діаграма випадків використання розписує функціональні можливості, модель вводу-виводу дає глобальне уявлення про спосіб перетворення даних, і нарешті, структурно-логічна схема описує послідовність внутрішніх процесів. Ці моделі спільно пропонують основу для переходу від формулювання вимог через побудову архітектури, вибір технологічних інструментів та програмних компонентів.

1.5 Вигляд та призначення кінцевого програмного продукту

Кінцевий програмний продукт являє собою веб-застосунок, у якому реалізовано механізм автоматичного створення інтерфейсних компонентів на основі моделей штучного інтелекту. Система охоплює увесь процес побудови компонента: від моменту, коли користувач описує його словами, і до формування готового набору файлів, які можна переглянути та використати у власному проєкті. Основна ідея полягає у спрощенні розроблення типових UI-елементів і зменшенні кількості ручних кроків, що традиційно забирають значну частину часу.

З погляду користувача система представлена як багатосторінковий веб-застосунок із зрозумілою структурою та інструментами взаємодії. На головній сторінці розміщено форму для введення текстового опису майбутнього компонента, вибору потрібної мовної моделі та бібліотеки стилів. Після відправлення запиту користувач переходить на сторінку попереднього перегляду. Тут можна побачити отриманий код, структуру згенерованих файлів і відразу переглянути первинний рендер компонента у браузері, що дає змогу оцінити його вигляд та коректність роботи.

У застосунку передбачено каталог створених компонентів — своєрідну історію генерацій. У ньому можна відкрити будь-який раніше сформований компонент, переглянути його вміст, повторити генерацію або завантажити все як архів для подальшого використання поза межами системи. Крім того, є окрема сторінка з публічними компонентами, що формує загальнодоступну колекцію готових рішень і може слугувати основою для розширення бібліотеки.

Функціональність продукту включає підтримку кількох мовних моделей, автоматичне створення й форматування файлів, механізми валідації результатів та автентифікацію для роботи з персональними даними. Усі ці можливості разом утворюють інструмент, який допомагає суттєво спростити процес розроблення інтерфейсних компонентів і підвищити продуктивність веб-розробників.

2. ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ СИСТЕМИ

Тема розділу – проектування та розробка програмної системи для автоматизованої генерації компонентів інтерфейсу користувача. На цьому етапі здійснюється перехід від аналізу вимог з першого розділу (розділ 1) до проектування архітектури системи, моделювання її внутрішніх процесів та реалізації основних функцій. На цьому рівні проектування є важливою частиною життєвого циклу програмного забезпечення, оскільки правильність архітектурних рішень впливає на майбутню якість продукту, масштабованість, гнучкість та підтримуваність згенерованої системи [1, 3, 4].

Однією з функцій, включених у цю систему, є включення великих мовних моделей, які здійснюють синтез вихідного коду компонентів інтерфейсу на основі текстового опису або структурованих специфікацій. Це вимагає розробки архітектури, відповідальної за надійний обмін даними з різними моделями, реагування на їхні вхідні дані, структурування файлової системи, що складається з компонентів, та забезпечення правильного виходу [7, 10].

Також однією з ключових функцій є наявність кількох мовних моделей та можливість вибору найкращої для конкретного завдання, яке потрібно реалізувати з адаптивним механізмом інтеграції. Тут визначається загальна архітектура системи, визначаються логічні рівні, згадуються модулі та принципи їхньої взаємодії. Підсистеми, відповідальні за генерацію підказок, взаємодію мовних моделей та генерацію шляхом обробки результатів генерації, а також створення файлової структури та зберігання, аналізуються значною мірою [3, 4, 29].

На концептуальному та прикладному рівні це можна описати через моделі даних та діаграми взаємодії, що надаються. Разом із проектуванням розділ охоплює аспекти процесу реалізації програмної системи, а саме обґрунтування вибору технологій та моделей, підходи до інтеграції з API мовних моделей, реалізацію програмних модулів та обробку даних, серед іншого. Тому в розділі 2 ми представляємо архітектурні

рішення з їхньою фактичною роботою та закладаємо основу для подальшого тестування, оцінки та реалізації системи.

2.1 Загальна архітектура системи

Архітектура системи автоматизованої генерації UI-компонентів визначає її структурну організацію, логіку роботи функціональних модулів та механізми взаємодії між складовими частинами. На етапі проєктування було враховано вимоги, сформульовані у першому розділі, а також специфіку використання великих мовних моделей, що виконують основну інтелектуальну функцію системи. Архітектура побудована відповідно до принципів модульності, розширюваності, слабкої зв’язаності та інверсії залежностей, що забезпечує гнучкість та адаптивність системи у процесі розвитку.

Логічна структура системи реалізована за шаровою моделлю та включає презентаційний рівень, рівень прикладної логіки, рівень інтеграції з мовними моделями та рівень збереження даних. Презентаційний рівень відповідає за взаємодію з користувачем, включаючи введення опису компонента, вибір моделі генерації, перегляд історії та виконання завантаження результатів. Використання компонентного підходу дозволяє забезпечити незалежність між елементами інтерфейсу та спрощує розширення функціональності.

Рівень прикладної логіки виконує основні функції системи: формування промптів, маршрутизацію запитів, виклики відповідних стратегій мовних моделей, обробку отриманих результатів та генерацію структурованих файлів з вихідним кодом UI-компонентів. У межах цього рівня реалізовано модулі валідації коду, форматування, створення файлової структури та запису метаданих у сховище. Для спрощення взаємодії між модулями генерації файлової структури використано шаблон проєктування фасад, який ізолює складні послідовності операцій за єдиним інтерфейсом.

Особливе місце в архітектурі займає механізм вибору мовної моделі. Оскільки різні моделі (GPT, Claude, Gemini) мають відмінну поведінку, контекстні обмеження та параметри генерації коду, у системі застосовано патерн стратегія. Кожна модель представлена окремою реалізацією інтерфейсу стратегії, що інкапсулює специфічну логіку взаємодії з відповідним API. Вибір стратегії здійснюється динамічно залежно від параметрів, вказаних користувачем. Такий підхід відповідає принципу відкритості/закритості та дозволяє розширювати систему новими моделями без змін у прикладній логіці.

Аналогічний принцип застосовано для реалізації багатомодульної системи авторизації. Оскільки користувачі можуть виконувати вхід за допомогою різних механізмів (Google OAuth, GitHub OAuth або стандартна автентифікація за логіном і паролем), логіка автентифікації також реалізована через патерн стратегія. Кожна стратегія містить власні алгоритми перевірки користувача та взаємодії з відповідним зовнішнім сервісом. Це забезпечує незалежність від конкретних провайдерів авторизації та можливість додавання нових методів без порушення структури системи.

Для взаємодії прикладної логіки з мовними моделями використано патерн адаптер. Кожний адаптер інкапсулює специфічні виклики API, забезпечуючи уніфікований спосіб обробки запитів. Таким чином, адаптер виступає проміжною ланкою між стратегіями мовних моделей та зовнішніми API, ізолюючи прикладний код від особливостей протоколів доступу.

Рівень збереження даних відповідає за запис історії генерацій, користувацьких метаданих та тимчасових файлів. Структура даних побудована таким чином, що дозволяє відновлювати попередні версії компонентів, повторно використовувати їх або аналізувати якість генерації для різних моделей. Збереження здійснюється у форматах, що спрощують подальшу інтеграцію та експорт результатів. Додатково передбачено механізми версіонування та маркування станів компонентів, що забезпечує простежуваність змін протягом усього життєвого циклу генерації. Окрему увагу приділено оптимізації зберігання великих обсягів даних шляхом розмежування

постійних та тимчасових артефактів. Це підвищує продуктивність системи та зменшує витрати ресурсів під час масштабування сервісу.

Узагальнена архітектура системи представлена на рисунку 2.1 та відображає рівні, модулі та напрямки взаємодії між компонентами.

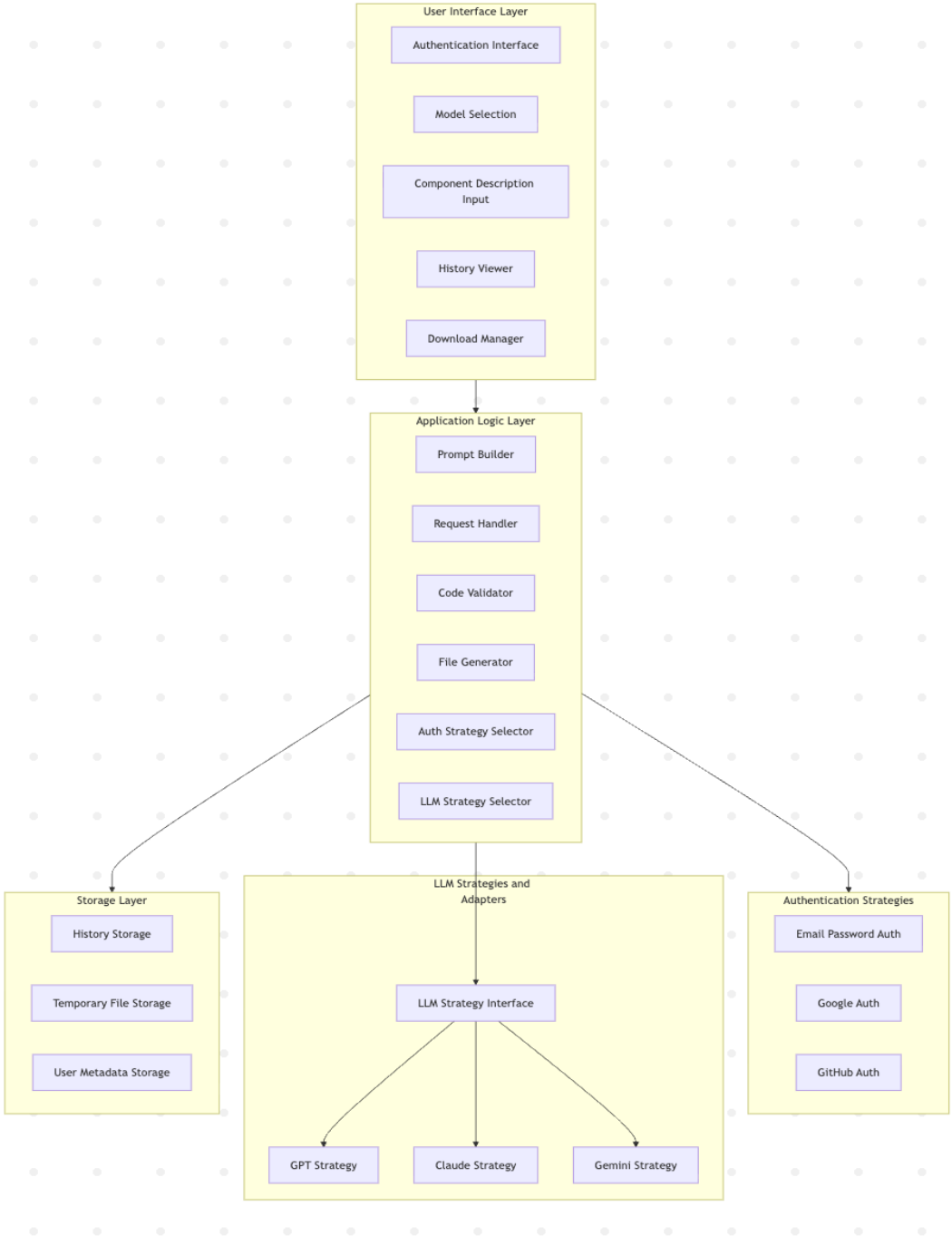


Рисунок 2.1 – Повна архітектура системи

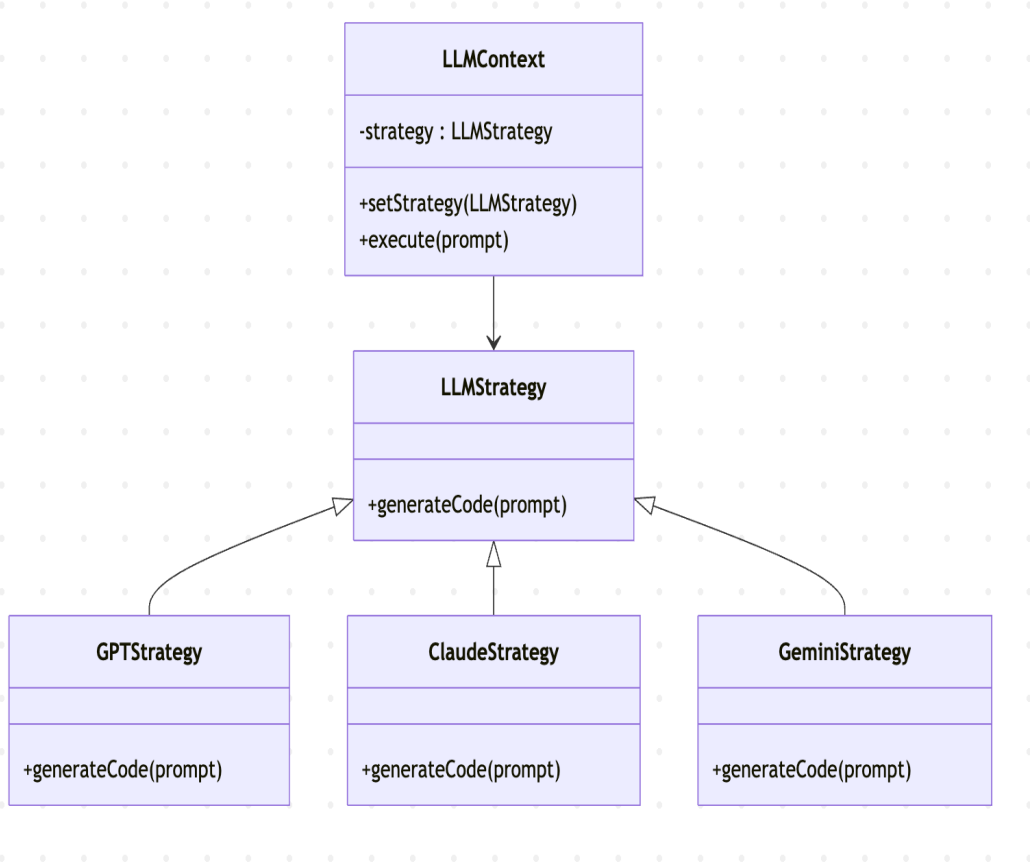


Рис. 2.2 – Патерн «Стратегія» для вибору мовної моделі

2.2 Обґрунтування вибору технологій та моделей

Вибір технологічного стеку та мовних моделей є ключовим етапом у проєктуванні програмної системи автоматизованої генерації UI-компонентів. Оскільки система працює у взаємодії з великими мовними моделями та виконує формування повноцінної файлової структури компонентів, вибір відповідних інструментів має забезпечити високу продуктивність, гнучкість, масштабованість і можливість подальшого розвитку. Технології та моделі добиралися з огляду на вимоги, визначені у розділі 1, та із урахуванням особливостей сучасних підходів до побудови веб-застосунків.

Основою клієнтської частини є фреймворк React, який реалізує компонентний підхід до побудови інтерфейсів та відзначається широкою підтримкою інструментів і бібліотек. Застосування React є доцільним, оскільки система генерує саме UI-компоненти, і використання відповідного середовища дозволяє легко інтегрувати результати генерації в реальні проєкти. Доповнення у вигляді TypeScript забезпечує статичну типізацію, яка суттєво підвищує якість та надійність згенерованого коду, а також полегшує його подальший супровід.

Для серверної логіки обрано середовище Node.js у поєднанні з фреймворком Next.js, що дає можливість реалізовувати серверні обробники запитів, забезпечувати інтеграцію з API мовних моделей та виконувати формування файлової структури на серверному рівні. Вибір Next[15, 16].js також зумовлений підтримкою сучасних механізмів маршрутизації, можливістю гібридної рендеризації та доступністю middleware-інтерфейсів. Це дозволяє оптимізувати обробку запитів та забезпечити захист приватних API-маршрутів.

Система використовує кілька способів автентифікації, зокрема стандартну авторизацію за електронною поштою, а також вхід через Google і GitHub. Реалізація кількох способів автентифікації стала можливою завдяки використанню патерну стратегії, де кожен метод автентифікації представлений окремою реалізацією спільного інтерфейсу. Такий підхід забезпечує можливість розширення системи новими способами автентифікації без зміни її основної логіки, що відповідає принципам гнучкості та відкритості до модифікацій[4].

Вибір мовних моделей є одним з ключових аспектів системи, оскільки від них залежить якість та структурованість згенерованих UI-компонентів. Система підтримує кілька моделей, серед яких GPT-4.1/5, Claude 3 та Gemini. Ці моделі було обрано з огляду на їх здатність ефективно працювати з інструкціями, генерувати структурований програмний код, підтримувати великі контекстні вікна та забезпечувати стабільність результатів. Для користувача реалізовано механізм вибору

моделі, що дозволяє адаптувати генерацію під конкретні потреби – наприклад, високу якість, швидкість або економію вартості запитів.

Оскільки взаємодія з різними мовними моделями передбачає суттєві відмінності у форматах запитів та відповідей, у системі використано поєднання патернів стратегія та адаптер. Стратегія визначає поведінковий аспект – вибір моделі та алгоритм генерації, а адаптер інкапсулює специфіку взаємодії з API конкретного постачальника. Це забезпечує незалежність прикладної логіки від зовнішніх сервісів і дозволяє додавати нові моделі без переробки архітектури.

У ролі ORM-рішення обрано Prisma[17, 18], яка забезпечує типізований доступ до даних, автоматичне формування міграцій та зручний механізм роботи зі схемою бази даних. Для збереження історії генерацій, проміжних результатів та метаданих Prisma є оптимальним вибором завдяки своїй інтеграції з TypeScript і підтримці складних зв’язків між сутностями. Система також використовує сучасні інструменти форматування вихідного коду (Prettier, ESLint), що дозволяє забезпечити високу якість та узгодженість згенерованих файлів.

Таким чином, обрані технології та підхід до реалізації механізму вибору мовних моделей забезпечують відповідність функціональним і нефункціональним вимогам системи, а також створюють передумови для її подальшого масштабування. Завдяки застосуванню патернів стратегія та адаптер система може підтримувати різні алгоритми генерації та авторизації, залишаючись при цьому гнучкою та розширюваною. Усі ці рішення визначають архітектурний фундамент, на якому базується подальша реалізація програмних модулів, що розглядаються у наступному підрозділі.

2.3 Архітектура програмних модулів системи

Архітектура програмних модулів визначає внутрішню структуру програмної системи, логіку розподілу функцій між окремими компонентами та характер їх

взаємодії. На відміну від загальних принципів і рівневої архітектури, розглянутих у підрозділі 2.1, цей підрозділ описує деталізовану організацію модулів, що реалізують функціональність системи автоматизованої генерації UI-компонентів[3].

Побудова архітектури базується на принципах модульності, низької зв’язаності та чіткого розмежування відповідальностей. Кожен модуль виконує окрему функцію й взаємодіє з іншими компонентами через визначені інтерфейси, що забезпечує можливість незалежної розробки та тестування модулів, а також полегшує модернізацію системи у майбутньому [3, 4].

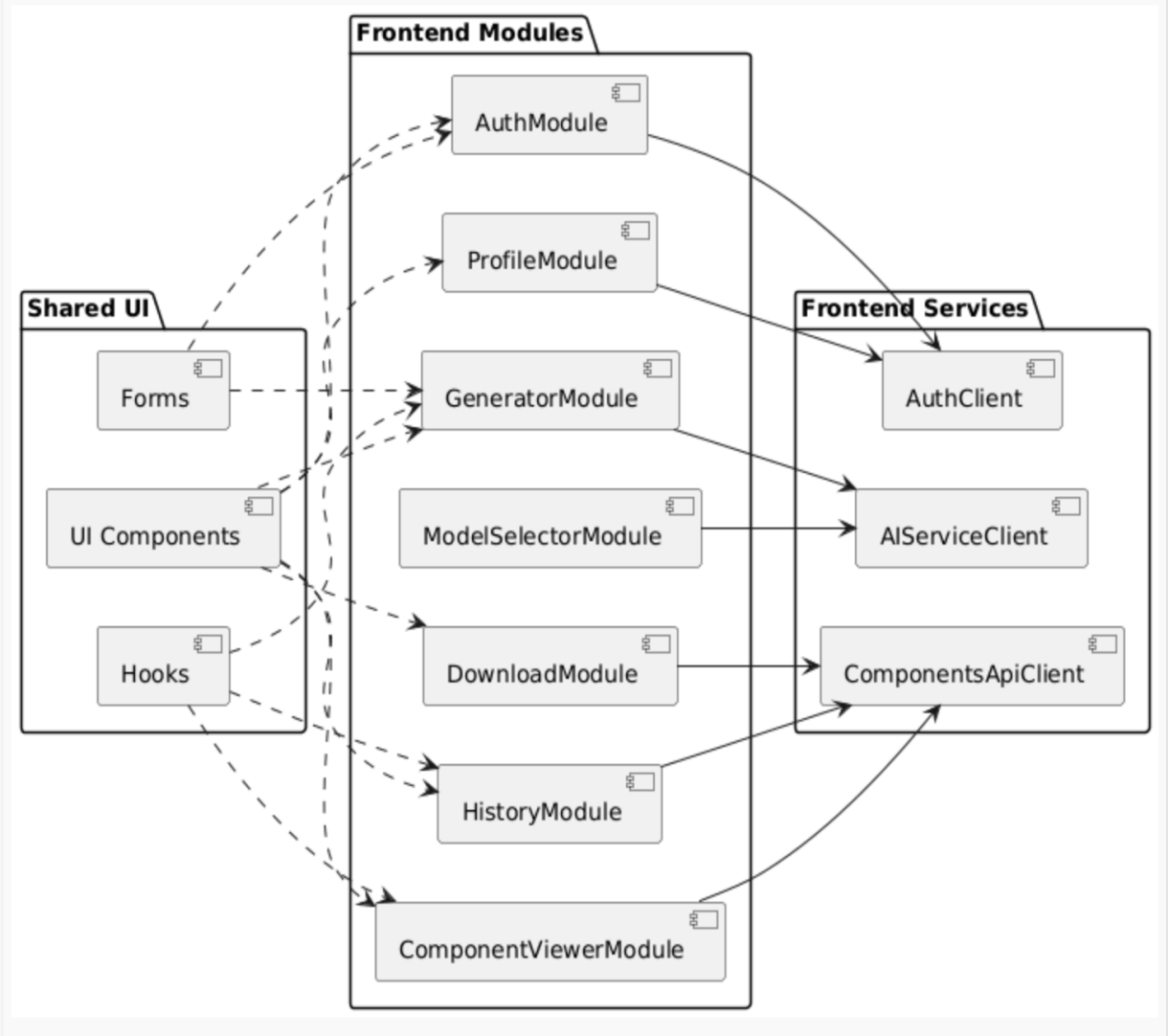


Рисунок 2.3 – Структура клієнтських модулів системи

Клієнтська частина програми охоплює модулі, які забезпечують взаємодію користувача із системою: введення опису майбутнього компонента, вибір мовної моделі, автентифікацію, перегляд історії створених компонентів та завантаження згенерованих файлів. Додатковий модуль керування станом забезпечує узгодженість даних між екранами інтерфейсу, а модуль HTTP-комунікації виконує роль проміжного шару між клієнтом і сервером. Загальну структуру клієнтських модулів наведено на рисунку 2.3.

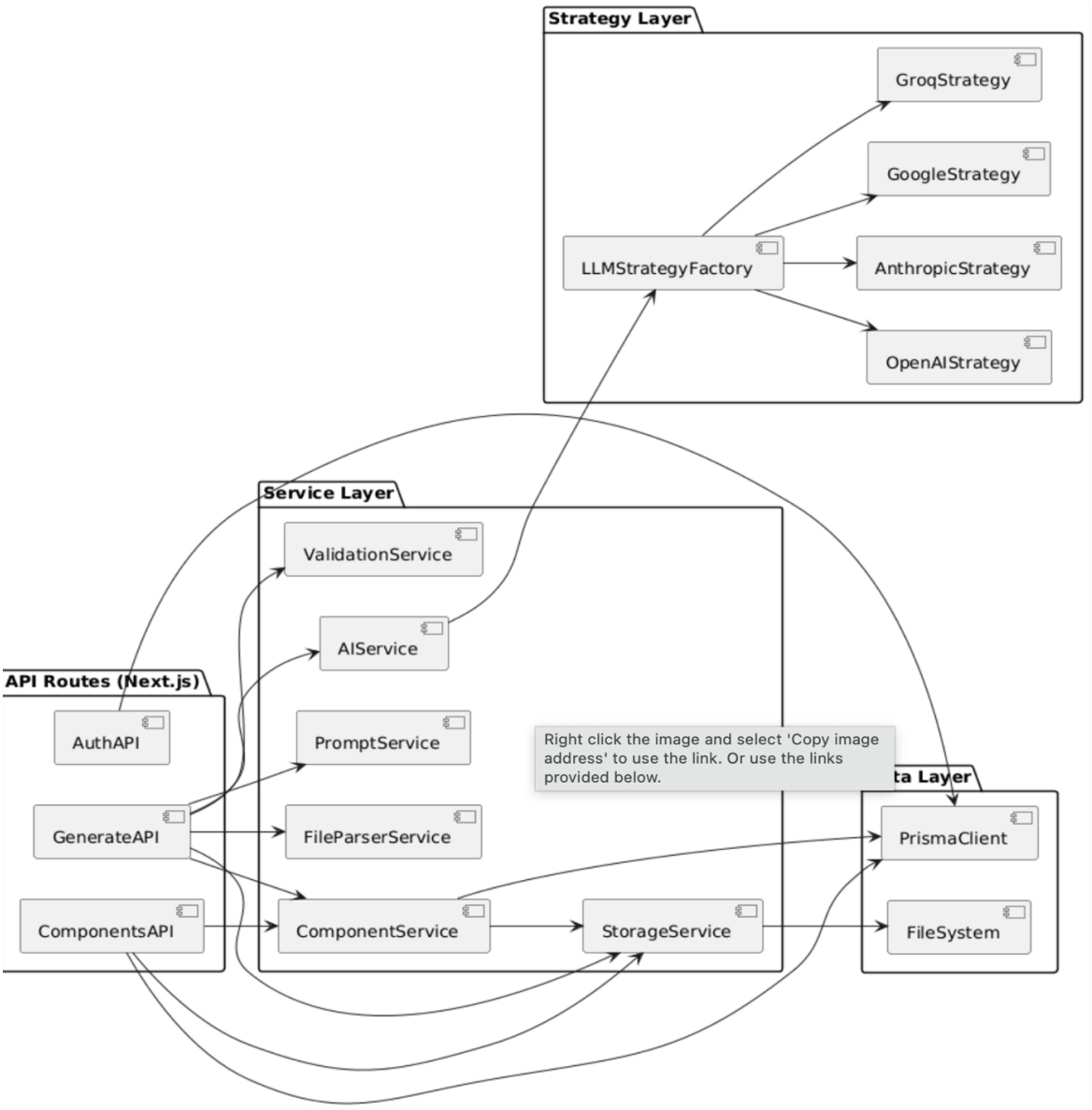


Рисунок 2.4 – Внутрішня архітектура серверних модулів і сервісного шару

Серверна частина реалізує основну прикладну логіку системи. Центральний маршрутизатор отримує запити клієнта, виконує первинну перевірку параметрів і спрямовує їх до сервісних модулів. На рівні сервісів зосереджено формування промптів, керування процесом генерації коду мовною моделлю, перевірку коректності згенерованого результату, формування файлової структури та запис інформації про створені компоненти. Сервіси забезпечують впорядкованість бізнес-логіки та зменшує дублювання коду. Структуру основних серверних модулів подано на рисунку 2.4.

Важливим елементом архітектури є модуль вибору мовної моделі. Оскільки система підтримує декілька моделей, які мають різні API та поведінку, застосовано шаблон проєктування «стратегія»[4]. Узагальнений інтерфейс описує єдиний спосіб отримання результату генерації, тоді як конкретні стратегії для GPT, Claude чи Gemini реалізують власні алгоритми взаємодії з відповідними сервісами. Специфічні відмінності зовнішніх API інкапсульовані у відповідних адаптерах, що застосовуються разом зі стратегіями. Такий підхід забезпечує можливість динамічного перемикання між моделями без змін у прикладній логіці. Узагальнену схему роботи модуля подано на рисунку 2.5.

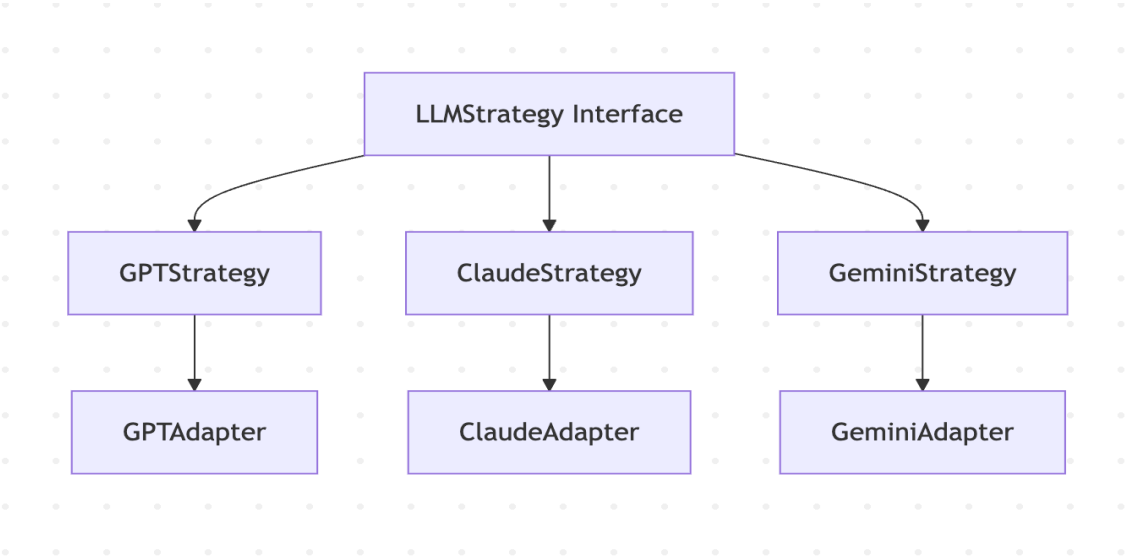


Рисунок 2.5 – Архітектура модуля взаємодії з мовними моделями на основі шаблонів «стратегія» та «адаптер»

Аналогічний принцип використано у підсистемі автентифікації. Оскільки користувач може здійснювати вхід за електронною поштою, через Google або GitHub, логіка автентифікації також реалізована у вигляді набору стратегій. Кожна стратегія інкапсулює окремий механізм перевірки користувача, а селектор стратегій визначає, яку реалізацію слід застосувати в конкретному випадку. Завдяки цьому структура системи залишається сталою, а підтримка нових способів автентифікації може бути додана без змін у сервісній логіці. Узагальнену структуру цього модуля наведено на рисунку 2.6.

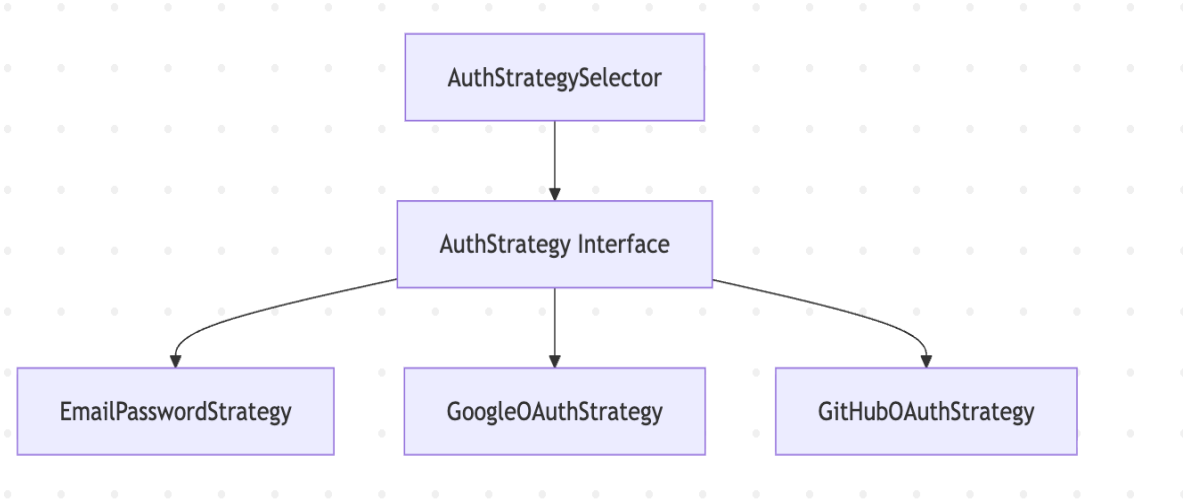


Рисунок 2.6 – Архітектура модуля автентифікації користувачів на основі шаблону «стратегія»

Окремим структурним елементом системи є модуль генерації файлової структури, який відповідає за перетворення результату роботи мовної моделі у фізичні файли компонента. Він виконує створення директорій, запис файлів, застосування форматування та формування архіву для завантаження. Для зручності використання модуль реалізовано за допомогою шаблону «фасад»[4], що дозволяє приховати складну внутрішню структуру за єдиним інтерфейсом. Схему організації цього модуля наведено на рисунку 2.7.

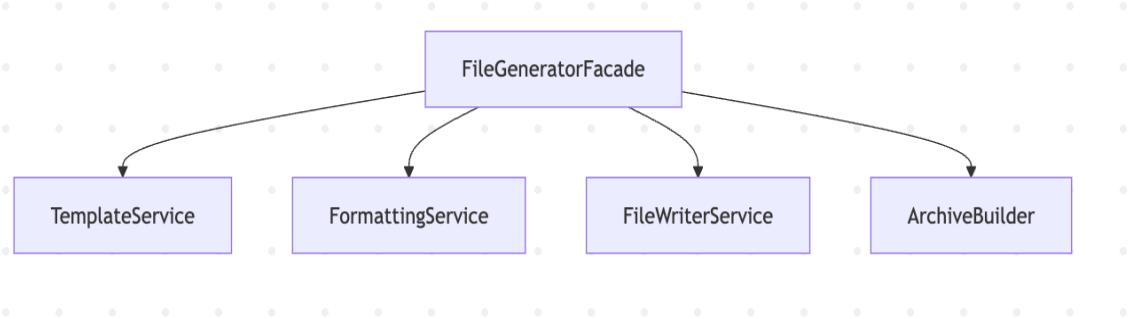


Рисунок 2.7 – Архітектура модуля генерації файлової структури на основі шаблону «фасад»

Підсистема доступу до даних реалізована за допомогою ORM Prisma, яка поєднує механізми опису моделі даних, генерації типів для TypeScript та виконання запитів до бази. Доступ до таблиць бази здійснюється через автоматично згенеровані моделі Prisma Client, що дозволяє уникнути дублювання логіки та забезпечує типобезпечність взаємодії з даними. Операції, пов’язані з історією генерацій, збереженням файлів і даними користувачів, організовані у вигляді логічних груп викликів Prisma, що використовуються відповідними сервісами. Такий підхід спрощує архітектуру й усуває потребу у створенні додаткового шару репозиторіїв.

Таким чином, архітектура програмних модулів системи поєднує шаровий підхід із застосуванням низки шаблонів проєктування, що забезпечує гнучкість, розширюваність і стійкість до змін. Деталізований опис модулів створює підґрунтя для подальшої реалізації системи та дозволяє узгодити логіку роботи окремих її компонентів[3, 4].

2.4 Модель даних та структура зберігання

Модель даних системи відображає логічну структуру інформації [3], яка формується, використовується та зберігається під час роботи програмного комплексу. Особливістю системи автоматизованої генерації UI-компонентів є необхідність зберігати як метадані користувачів та їхніх запитів, так і структуру створених компонентів, включно з окремими файлами, що формують готовий UI-модуль. Побудова моделі даних здійснювалася на основі функціональних вимог системи, визначених у першому розділі, з урахуванням необхідності забезпечення типобезпечного доступу, підтримки історичності та можливості подальшого розширення.

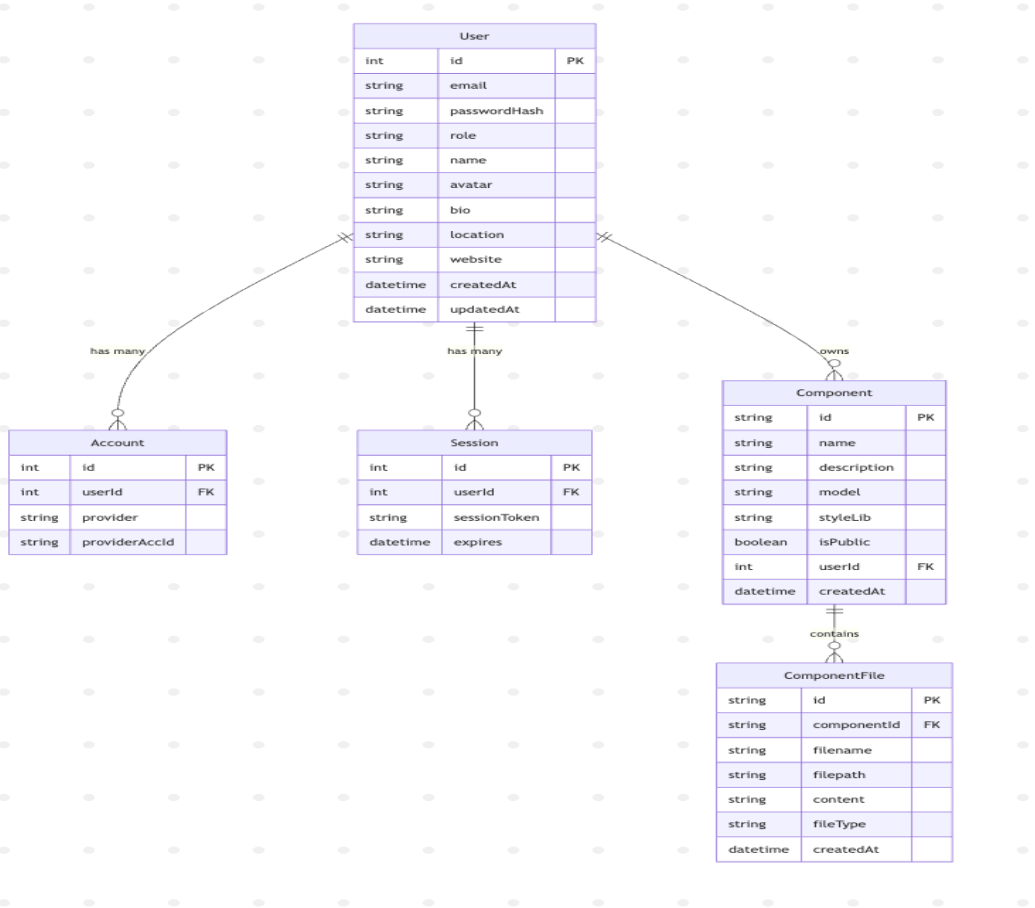


Рисунок 2.8 – Логічна модель даних системи (ER-діаграма)

У структурі застосунку основними сутностями є користувач, інформація про OAuth-акаунти, сесії, компоненти, створені за допомогою мовних моделей, та файли, що належать цим компонентам. Така модель дозволяє забезпечити повний цикл роботи системи: автентифікацію користувача, збереження його промптів і результатів генерації, повторний перегляд створених компонентів та керування їх доступністю. Узагальнене логічне представлення цих сутностей наведено на рисунку 2.8, який відображає структуру даних на концептуальному рівні.

Наведена модель демонструє багаторівневу організацію даних. Користувач пов’язаний із зовнішніми обліковими записами, що забезпечують авторизацію через сторонніх провайдерів. Крім того, користувач має одну або декілька активних сесій, залежно від способу автентифікації та налаштувань платформи. Кожен користувач може створювати довільну кількість компонентів, які у свою чергу містять набори файлів, що формують структуру UI-модуля. Такий підхід дозволяє зберігати результати роботи мовної моделі у повному вигляді, включаючи окремі частини коду.

Фізична модель даних реалізована з використанням ORM Prisma, що забезпечує автоматичне генерування типів, валідацію структури даних та виконання SQL-операцій відповідно до описаної схеми.

Важливим елементом моделі даних є зберігання структури згенерованих UI-компонентів. Оскільки результат роботи мовної моделі складається з кількох файлів, між сутністю компонента та сутністю файлу встановлено зв’язок «один до багатьох». Кожен файл містить власний шлях, тип та вміст, що дозволяє системі відображати структуру проєкту й надавати користувачеві можливість завантаження архіву з усіма сформованими файлами.

На рисунку 2.9 подано узагальнену схему потоку даних у процесі збереження результатів генерації. Діаграма демонструє, яким чином вхідний опис користувача перетворюється у компоненти, які надалі зберігаються у базі даних разом зі структурою їх файлів.

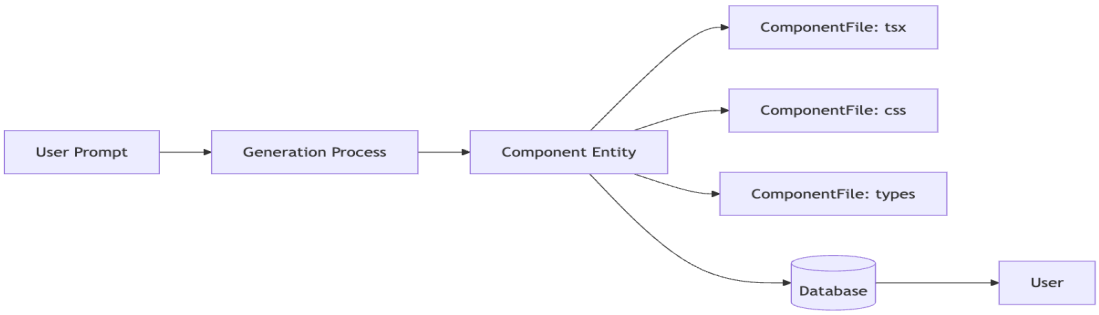


Рисунок 2.9 – Потік даних під час збереження результатів генерації

Наведені діаграми та опис моделі даних відображають структурну цілісність бази даних, її логічні зв’язки та принципи організації. Така модель забезпечує повноцінну підтримку історичності та відтворюваності результатів генерації, гарантує типобезпечність даних та створює фундамент для подальшого розширення системи, включно з можливістю додавання нових типів компонентів або метаданих без зміни базових структур.

2.5 Моделювання процесів та діаграми послідовності

Моделювання процесів дає можливість формально описати динамічну поведінку системи та порядок виконання операцій між окремими модулями. На відміну від статичних моделей, представлених у попередніх підрозділах, діаграми послідовності демонструють взаємодію між компонентами у часі та дозволяють простежити, яким чином запити користувача проходять через систему, формуються, обробляються та зберігаються. Такий підхід забезпечує глибше розуміння логіки роботи програмного комплексу та підтверджує узгодженість архітектури з вимогами [1, 3].

Основним процесом у системі є генерація UI-компонента на основі введеного користувачем опису. Взаємодія між клієнтським інтерфейсом, серверною логікою, вибраною мовною моделлю та модулем збереження результатів показує, як саме система перетворює текстовий запит на структурований набір файлів. Узагальнена діаграма послідовності цього процесу наведена на рисунку 2.10.

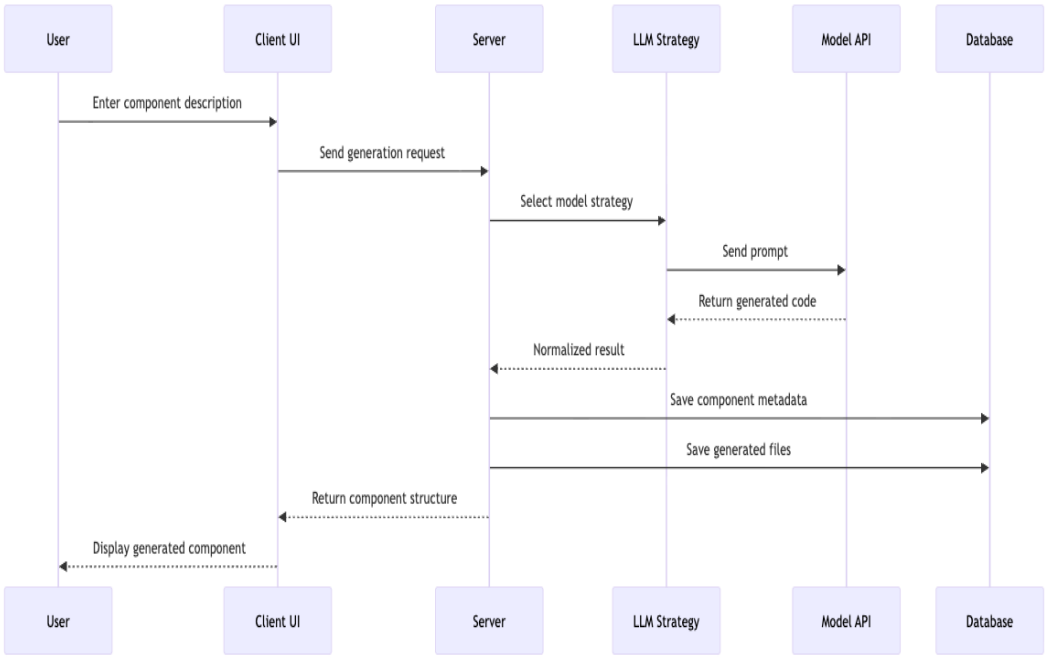


Рисунок 2.10 – Послідовність операцій у процесі генерації UI-компонента

Процес починається з введення даних користувачем у клієнтському інтерфейсі, після чого запит надсилається на сервер. Сервер вибирає відповідну стратегію мовної моделі, формує промпт і передає його до зовнішнього API. Повернутий результат нормалізується, перевіряється та зберігається у базі даних як структура компонентів. Користувач отримує готовий набір файлів, які може переглянути або завантажити.

Окремим процесом системи є автентифікація, що забезпечує доступ до історії генерацій та персоналізованого робочого середовища. Процедура авторизації може відрізнятися залежно від способу входу, але загальна послідовність взаємодій залишається сталою. На рисунку 2.11 представлено узагальнену діаграму послідовності для процесу входу користувача.

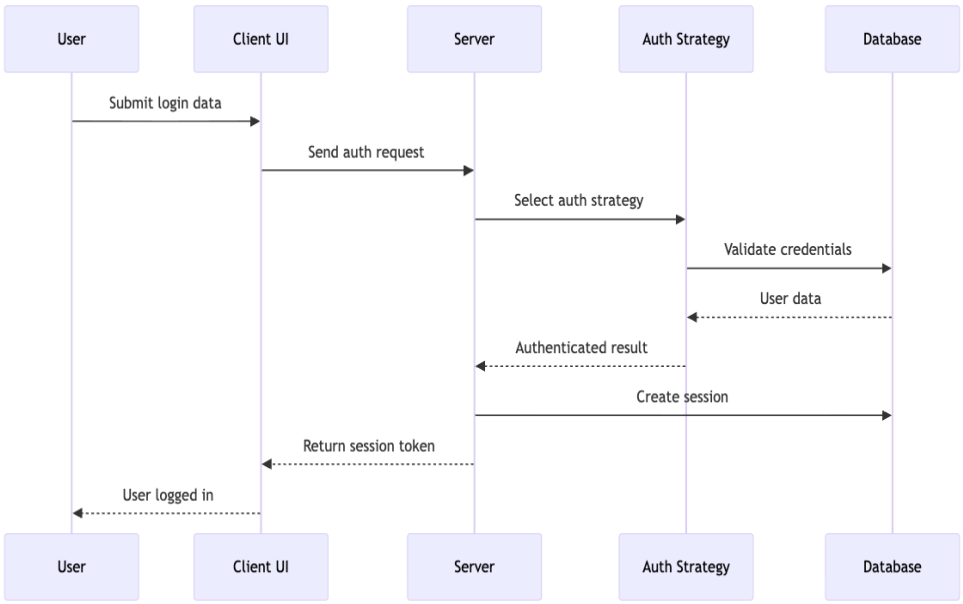


Рисунок 2.11 – Діаграма послідовності процесу автентифікації

Процес передбачає отримання даних від користувача, вибір відповідної стратегії автентифікації, перевірку даних та створення сесії. Використання стратегій дозволяє адаптувати алгоритм авторизації до різних провайдерів, зберігаючи однаковий формат взаємодії з клієнтом.

Завершальним елементом моделювання є відтворення процесу збереження результатів роботи мовної моделі. Цей процес включає перевірку повернених даних,

формування окремих файлових структур та запис результатів у базу даних. На рисунку 2.12 наведено діаграму послідовності цього етапу.

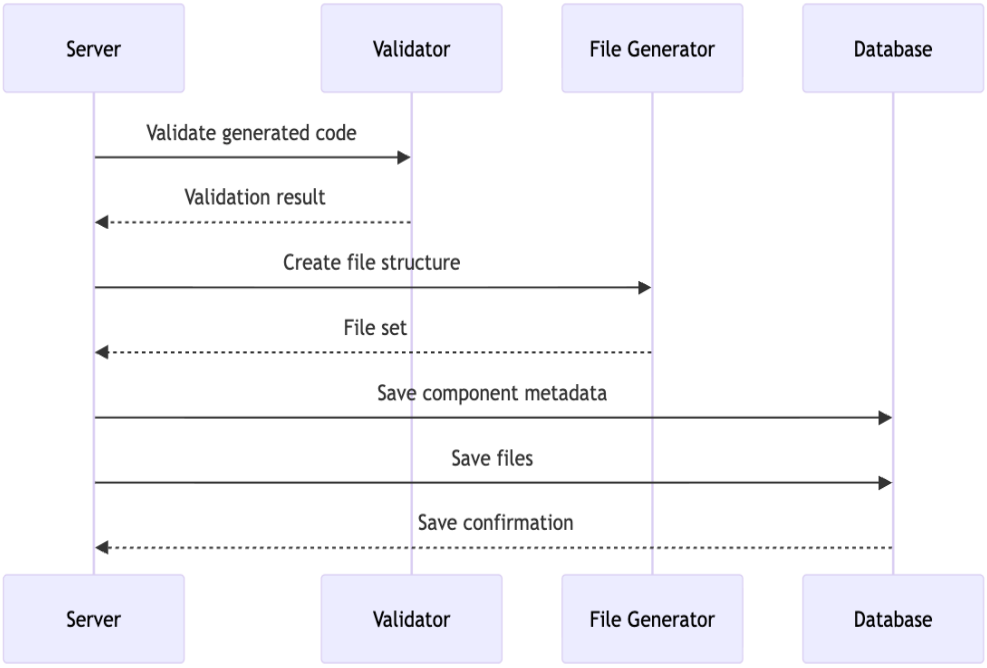


Рисунок 2.12 – Діаграма послідовності процесу збереження результатів

Після отримання результатів генерації система виконує їх перевірку, створює необхідну структуру файлів і зберігає їх у базі даних як частину компонента. Така організація процесу дозволяє забезпечити цілісність результатів і можливість подальшого доступу до них з боку користувача.

Описані діаграми ілюструють взаємодію основних модулів системи та підтверджують узгодженість програмної архітектури з функціональними вимогами. Моделювання процесів дає змогу відстежити логіку виконання ключових операцій та становить важливу основу для подальшої реалізації та тестування системи.

2.6 Реалізація серверної частини

Серверна частина системи реалізована на основі механізму App Router, який забезпечує обробку HTTP-запитів та взаємодію з базою даних і мовними моделями. Усі маршрути згруповано за функціональними доменами: автентифікація, робота з профілем користувача, управління компонентами та генерація UI-структур за допомогою штучного інтелекту.

Архітектура серверної частини побудована таким чином, щоб API залишалося мінімалістичним, а бізнес-логіка переносилася у відповідні сервісні модулі. Це забезпечує розширюваність, повторне використання та незалежність окремих частин системи.

Набір маршрутів, реалізованих у серверній частині, включає такі групи: автентифікація (реєстрація, вхід, вихід), профіль користувача, операції над компонентами та маршрути для генерації UI-компонентів. На рисунку 2.13 наведено загальну схему обробки серверних запитів відповідно до логічних груп API.

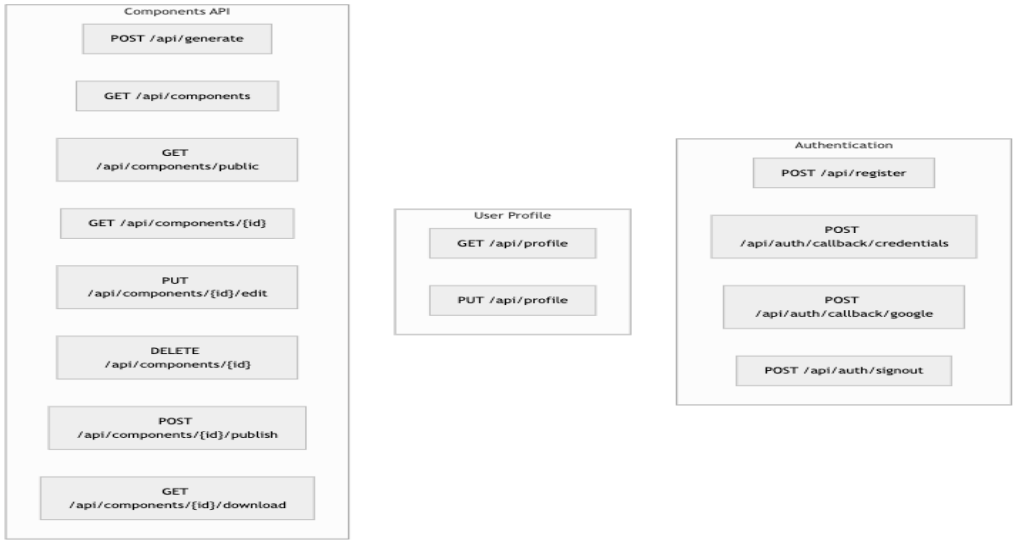


Рисунок 2.13 – Загальна структура серверних маршрутів системи

Серверні маршрути реалізовані у вигляді обробників запитів, кожен з яких відповідає за певну операцію [3, 4]. Наприклад, маршрут `/api/generate` приймає структурований опис компоненту, передає його до сервісу генерації, викликає відповідну мовну модель і зберігає отримані результати. Фрагмент реалізації обробника генерації компонента наведено в лістингу 2.1 Усі маршрути працюють у режимі server-only, що гарантує безпечне використання ключів API та взаємодію з базою даних.

Лістинг 2.1 Фрагмент реалізації обробника генерації компонента:

```
export async function POST(request: NextRequest) {
  return withErrorHandler(log, async () => {
    const session = await auth();
    const userId = getUserIdFromSession(session);

    const body: GenerateComponentRequest = await request.json();
    const { prompt, model, styleLibrary } = body;

    if (!prompt || !model || !styleLibrary) {
      throw new ValidationError("Missing required fields: prompt,
model, styleLibrary");
    }

    const result = await componentGenerationFacade.generate({
      prompt,
      model,
      styleLibrary,
      userId,
    });

    if (!result.success) {
      const statusCode = result.type === "VALIDATION_ERROR" ? 400
: 500;
      throw new ApiError(result.error, statusCode, result.type);
    }

    return NextResponse.json({ success: true, component:
result.component });
  });
}
```

Маршрут `/api/components` забезпечує отримання списку компонентів користувача. Дані повертаються у форматі, який містить метадані компонентів, але без завантаження вмісту файлів. Це дозволяє ефективно відображати історію створених компонентів без додаткових витрат на передачу великих обсягів даних.

Маршрут `/api/components/[id]` повертає структурований опис компонента разом з усіма файлами, що входять до його складу. Застосування Prisma дозволяє виконувати вкладені запити та повертати пов’язані записи у вигляді єдиної структури. Приклад реалізації цього маршруту подано в лістингу 2.2

Лістинг 2.2 Фрагмент обробника отримання одного компонента:

```
const log = logger.child("API:Components");

export async function GET() {
  return withErrorHandler(log, async () => {
    const session = await auth();
    const userId = getUserIdFromSession(session);
    const components = await getUserComponents(userId);

    return NextResponse.json({ success: true, components });
  });
}
```

Серверна частина містить також маршрути для авторизації та керування профілем. Зокрема, /api/register створює нового користувача на основі переданих даних, /api/auth/callback/credentials відповідає за вхід за електронною поштою та паролем, а /api/profile забезпечує отримання та оновлення інформації про користувача. Фрагмент реалізації обробника реєстрації користувача наведено в лістингу 2.3

Лістинг 2.3 Фрагмент обробника реєсрації

```
const registerSchema = z.object({
  email: z.string().email("Invalid email address"),
  password: z.string().min(8, "Password must be at least 8
characters"),
```

```
  role: z.enum(["user", "admin"]).optional().default("user"),
});

export async function POST(request: NextRequest) {
  return withErrorHandler(log, async () => {
    const body = await request.json();

    const validationResult = registerSchema.safeParse(body);
    if (!validationResult.success) {
      throw new
ValidationError(validationResult.error.errors[0].message);
    }

    const { email, password, role } = validationResult.data;

    const existingUser = await prisma.user.findUnique({
      where: { email },
    });

    if (existingUser) {
      throw new ValidationError("User with this email already
exists");
    }
```

Продовження лістингу 2.3 Фрагмент обробника реєсрації

```
    const passwordHash = await bcrypt.hash(password, 10);

    const user = await prisma.user.create({
      data: {
        email,
        passwordHash,
        role,
      },
      select: {
        id: true,
        email: true,
        role: true,
        createdAt: true,
      },
    });

    return NextResponse.json({ message: "User created
successfully", user }, { status: 201 });
  });
}
```

Серверна частина системи побудована з урахуванням принципів модульності та розширюваності, що дозволяє легко додавати нові функції, інтегрувати додаткові

мовні моделі та розширювати логіку опрацювання компонентів. Структурований підхід до організації API, сервісних модулів та взаємодії з базою даних забезпечує стабільність та передбачуваність роботи всієї системи.

2.7 Реалізація підсистеми роботи з базою даних

Підсистема роботи з даними є центральним елементом серверної частини застосунку, оскільки забезпечує зберігання користувацької інформації, параметрів генерації та структури згенерованих UI-компонентів. У системі використовується реляційна база даних PostgreSQL [18, 19] у поєднанні з ORM Prisma, що дозволяє підтримувати строгий контроль типів, автоматизувати процес формування запитів та гарантувати цілісність даних.

Архітектура роботи з базою даних базується на використанні Prisma Client [17], який генерується на основі описаної схеми та забезпечує типобезпечну взаємодію з таблицями. Усі моделі, що беруть участь у роботі системи, визначено у файлі schema.prisma, і вони повністю відповідають логічній структурі, поданій у підрозділі 2.5. У реалізації використовуються моделі користувачів, OAuth-акаунтів, сесій, компонентів та файлів компонентів.

У системі передбачено кілька ключових груп операцій над даними. Перша з них пов’язана з автентифікацією користувачів, що включає створення нового облікового запису, пошук користувача за електронною поштою, отримання OAuth-акаунтів та керування сесіями. Друга група операцій забезпечує повний цикл взаємодії з компонентами – їх створення, перегляд, редагування, публікацію та видалення. Третя група охоплює збереження створених AI-генератором файлів разом зі структурою відповідних компонентів.

У процесі створення нового компонента система здійснює збереження метаданих генерації, таких як назва, модель мовного інтелекту, використана бібліотека стилів та первинний текстовий опис. Структура файлів компонента формується

окремо, але пов’язується із записом компонента на рівні транзакції, що гарантує узгодженість даних. Такий підхід унеможливлює ситуації, коли компонент створено без файлів або навпаки.

Взаємодія між серверними маршрутами та базою даних здійснюється через сервісний шар, у якому зосереджено всю бізнес-логіку [3, 4]. У цьому шарі реалізовано CRUD-операції над компонентами та файлами, що забезпечує чітке розділення відповідальностей між API-маршрутами та логікою доступу до даних. У межах підсистеми Prisma використовується каскадне видалення, що полегшує підтримку зв’язків між таблицями та запобігає появі "висячих" записів. Фрагмент реалізації створення запису компонента в базі даних наведено в лістингу 2.4.

Лістинг 2.4 – Фрагмент взаємодії з базою даних при створенні компонента

```
const component = await prisma.component.create({
      data: {
        name: componentName,
        description: prompt,
        model: model,
        styleLib: styleLibrary,
        userId: userId,
      },
    });
```

Для роботи з окремими файлами компонента застосовано вкладені операції створення записів. Це дозволяє зберігати повну структуру згенерованого UI-компонента одним викликом Prisma Client. У разі помилки під час створення файла або некоректної структури запит скасовується і жоден із часткових записів не додається до бази даних. Приклад реалізації збереження файлів компонента подано в лістингу 2.5.

Лістинг 2.5 – Фрагмент створення файлів компонента

```
const fileRecords = await Promise.all(
    parsed.files.map((file) =>
      prisma.componentFile.create({
        data: {
```

```
              componentId: component.id,
              filename: file.filename,
              filepath: `generated-
components/${component.id}/${file.filename}`,
              fileType: file.fileType,
            },
          })
        )
      );
```

Отримання даних також реалізовано з використанням можливостей Prisma щодо включення пов’язаних моделей. Це дає змогу повертати структуру компонента разом із повним набором файлів у межах одного запиту, що суттєво спрощує побудову інтерфейсу та формування попереднього перегляду згенерованого UI-компонента. Фрагмент запиту для отримання компонента разом із пов’язаними файлами наведено в лістингу 2.6. Додатково сервісний шар виконує валідацію вхідних даних і контроль цілісності бізнес-правил перед виконанням операцій з базою даних

Лістинг 2.6 – Отримання компонента з файлами

```
    const components = await prisma.component.findMany({
        where,
        include: {
          files: true,
          user: {
            select: {
              id: true,
              name: true,
              email: true,
              avatar: true,
            },
          },
        },
        orderBy: sortBy === "name" ? { name: "asc" } : { createdAt:
"desc" },
        skip,
        take: limit,
      });
```

Видалення компонентів реалізоване на основі встановлених у моделі зв’язків onDelete: Cascade, що дає змогу автоматично видаляти пов’язані файли під час видалення компонента. Це спрощує підтримку бази даних і забезпечує відсутність дублювання або неповних записів.

Реалізована підсистема роботи з даними характеризується гнучкістю, розширюваністю та відповідністю принципам структурованого підходу до зберігання інформації. Завдяки використанню Prisma ORM вона забезпечує високу надійність, контроль типів та стабільну взаємодію між прикладною логікою та реляційною базою даних, що є критично важливим для системи автоматизованої генерації програмного коду.

2.8 Реалізація автентифікації та авторизації

Підсистема автентифікації виконує ключову роль у забезпеченні доступу користувача до персональних даних і згенерованих компонентів. Реалізація механізму автентифікації в системі побудована на поєднанні власної реєстрації та входу через облікові записи зовнішніх провайдерів. Такий підхід забезпечує гнучкість і дає можливість користувачам взаємодіяти із системою у відповідності до їхніх уподобань [4, 27] .

Основою механізму автентифікації є взаємодія між серверними маршрутами та моделями бази даних Prisma. Реєстрація нового користувача передбачає створення запису у таблиці User із хешованим паролем, що гарантує безпечне зберігання облікових даних. Для цього використовується окрема серверна функція, яка виконує валідацію введених даних, перевіряє унікальність електронної пошти та здійснює запис у базу [27].

Автентифікація за допомогою credentials реалізується через маршрут `/api/auth/callback/credentials`. Під час входу система отримує електронну пошту та пароль користувача, після чого порівнює хеш пароля з тим, що зберігається в базі

даних. Фрагмент реалізації входу користувача за допомогою credentials наведено в лістингу 2.7. У разі успіху створюється нова сесія та повертається відповідний токен.

Лістинг 2.7 – Вхід за допомогою credentials

```
const loginSchema = z.object({
  email: z.string().email("Invalid email address"),
  password: z.string().min(1, "Password is required"),
});

export async function POST(request: NextRequest) {
  return withErrorHandler(log, async () => {
    const body = await request.json();

    const validationResult = loginSchema.safeParse(body);
    const { email, password } = validationResult.data;

    const user = await prisma.user.findUnique({
      where: { email },
    });

    if (!user || !user.passwordHash) {
      throw new AuthenticationError("Invalid email or password");
    }

    const isPasswordValid = await bcrypt.compare(password,
user.passwordHash);

if (!isPasswordValid) {
      throw new AuthenticationError("Invalid email or password");
    }

    return NextResponse.json(
      {
        message: "Login successful",
        user: {
          id: user.id,
          email: user.email,
          role: user.role,
        },
      },
      { status: 200 }
    );
  });
}
```

Окрім базової автентифікації, система підтримує вхід через Google OAuth. У цьому випадку користувач спрямовується на сторінку авторизації Google, після чого сервер отримує перевірені дані користувача. Якщо користувач входить уперше, створюється запис у таблиці Account, який пов’язує його з відповідним провайдером [27].

Усі користувацькі дані та дані сесій зберігаються у відповідних таблицях Prisma. Структура таблиць Account та Session забезпечує можливість підтримки множинних способів входу для одного користувача. Це також дозволяє тримати роздільні токени для кожного облікового запису, що підвищує рівень безпеки.

Користувач може переглядати та оновлювати власний профіль через маршрут `/api/profile`. Оновлення профілю обмежується полями, які не впливають на механізм авторизації, що запобігає можливості порушення цілісності облікових даних.

Механізм виходу реалізований маршрутом `/api/auth/signout`, який видаляє або деактивує відповідну сесію. Це гарантує, що користувач втрачає доступ до ресурсів до наступного успішного входу.

Узагальнена діаграма взаємодії між клієнтом, сервером і механізмом автентифікації наведена на рисунку 2.14.

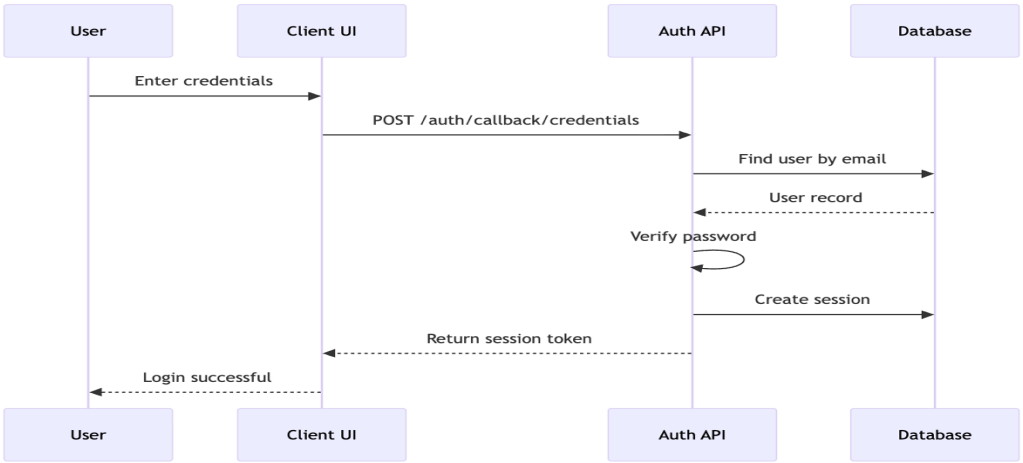


Рисунок 2.14 – Послідовність взаємодії під час входу користувача

2.9 Реалізація інтеграції з мовними моделями на основі шаблону «стратегія»

Підсистема інтеграції з великими мовними моделями забезпечує основну функціональність застосунку, оскільки саме вона відповідає за створення текстового подання UI-компонента на основі вхідного опису користувача. Реалізація побудована з використанням шаблонів проєктування «стратегія» та «адаптер», що дозволяє інкапсулювати відмінності між зовнішніми API різних провайдерів та забезпечує можливість їх динамічної заміни.

Базовий інтерфейс стратегії визначає уніфікований метод генерації компонента, який приймає підготовлену структуру промпта та повертає відповідь у стандартизованому форматі. Конкретні реалізації стратегії відповідають за виклик відповідного API, формування параметрів запиту, обробку помилок та нормалізацію результату. Такий підхід дозволяє уникнути жорсткого зв’язування серверної логіки з певним провайдером мовної моделі.

Лістинг 2.8 – Базовий інтерфейс стратегії

```
export interface GenerateParams {
  prompt: string;
  temperature?: number;
}

export interface GenerateResult {
  success: boolean;
  code?: string;
  error?: string;
  usage?: {
    promptTokens: number;
    completionTokens: number;
    totalTokens: number;
  };
}

/**
 * Базовий інтерфейс стратегії для AI генерації
 */
```

```
export interface AIStrategy {
  /**
   * Генерує код компонента
   */
  generate(params: GenerateParams): Promise<GenerateResult>;

  /**
   * Перевіряє чи налаштований провайдер (API ключі і т.д.)
   */
  isConfigured(): boolean;

  /**
   * Повертає назву провайдера
   */
  getProviderName(): string;
}
```

У системі реалізовано декілька стратегій, що відповідають різним мовним моделям (наприклад, OpenAI GPT або Google Gemini). Кожна стратегія містить власний адаптер, який виконує низькорівневу обробку запиту відповідно до вимог API конкретного провайдера. Це дозволяє ізолювати специфічні формати запитів, обмеження та параметри моделей у межах одного модуля, не впливаючи на загальну логіку роботи застосунку.

Перед виконанням генерації система формує структурований промпт, який містить опис компоненту, вибрані параметри стилізації, специфікацію очікуваного вихідного формату та допоміжні інструкції для мовної моделі. Формування промпта здійснюється в окремому сервісному модулі, що забезпечує єдину структуру вхідних даних для всіх можливих стратегій. Такий підхід дозволяє розширювати можливості системи та адаптувати її до різних моделей без змін у загальній архітектурі.

Лістинг 2.9 – Формування промпта для мовної моделі

```
## Task
Create a React component based on the following user request:
"${userPrompt}"

## Requirements

### Styling
```

```
${styleInstructions}
${CODE_QUALITY_REQUIREMENTS}
${DESIGN_REQUIREMENTS}
${APP_WRAPPER_INSTRUCTIONS}
${OUTPUT_FORMAT}

Generate the component now:`;
}
```

Після отримання відповіді від мовної моделі система виконує нормалізацію даних, що включає перевірку структури, відокремлення файлів та їх класифікацію відповідно до типів (`tsx`, `css`, `types`). Нормалізований результат передається у модуль генерації файлової структури, де завершується процес побудови компонента.

Реалізація підсистеми інтеграції з LLM забезпечує модульність, розширюваність і стійкість до змін. Завдяки використанню шаблонів «стратегія» та «адаптер» система може підтримувати нові мовні моделі, не змінюючи загальну архітектуру та не порушуючи роботу наявних функцій. Це робить модуль інтеграції з LLM одним із ключових елементів програмної платформи.

2.10 Реалізація клієнтської частини

Клієнтська частина системи забезпечує інтерактивну взаємодію користувача з основними можливостями застосунку, включно з автентифікацією, переглядом і керуванням компонентами, а також запуском процесу їх генерації. Фронтенд реалізовано засобами Next.js із використанням App Router, що дозволяє поєднувати серверний рендеринг із клієнтськими компонентами, оптимізуючи час завантаження та скорочуючи кількість запитів до API [15, 16].

Архітектура користувацького інтерфейсу побудована на основі компонентів React. Більшість сторінок є серверними компонентами, які отримують дані безпосередньо з API або бази даних під час рендерингу. Такий підхід дає змогу уникати глобального менеджера стану та мінімізувати обсяг клієнтської логіки. У

випадках, коли потрібна інтеракція (наприклад, введення параметрів генерації), застосовуються локальні стани компонентів React [21].

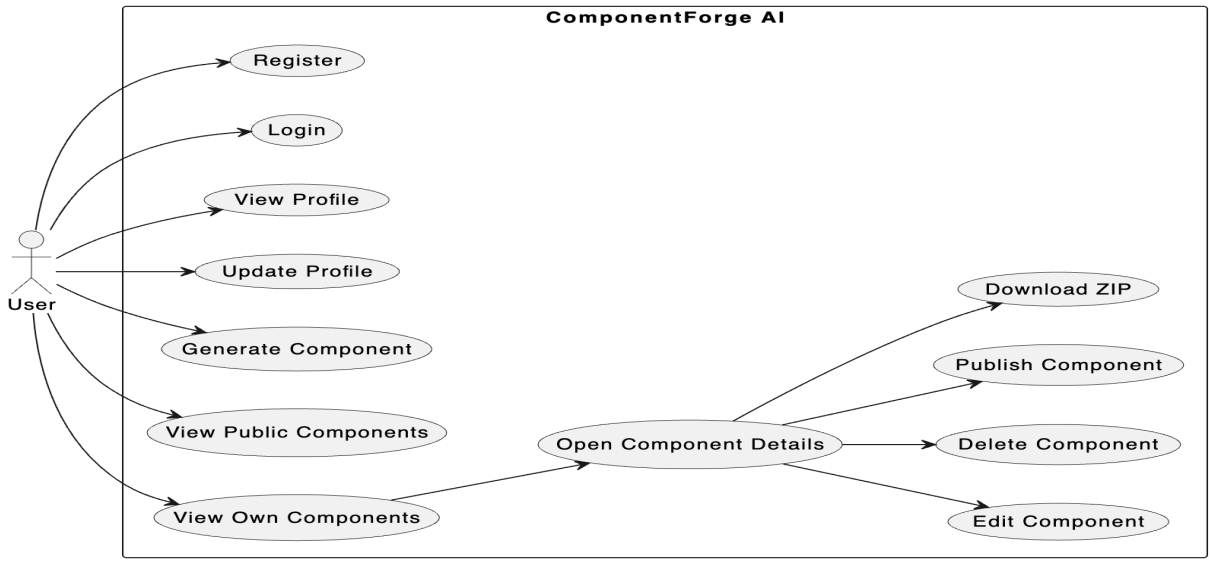


Рисунок 2.15 – Діаграма варіантів використання клієнтської частини

На рисунку 2.15 подано діаграму варіантів використання, що відображає ключові сценарії роботи користувача із клієнтським інтерфейсом.

Сторінки автентифікації забезпечують вхід та реєстрацію користувача через відповідні серверні маршрути. Після успішної автентифікації клієнтська частина перенаправляє користувача до робочого інтерфейсу. Профіль користувача реалізовано як серверну сторінку, що отримує актуальні дані з `/api/profile`, а редагування виконано за допомогою клієнтської форми.

Функція генерації UI-компонента надає користувачу інтерфейс введення опису, вибору мовної моделі та додаткових параметрів. Локальна клієнтська логіка використовується лише для опрацювання введених даних та надсилання структурованого запиту до маршруту `/api/generate`. Після отримання відповіді від сервера користувач може переглянути створений компонент або відкрити його сторінку. Приклад клієнтської логіки обробки запиту генерації та перенаправлення користувача наведено в лістингу 2.10.

Лістинг 2.10 –Код логіки виклику генерації

```
const handleGenerate = useCallback(
    async (params: SearchSubmitParams) => {
    // Check if user is authenticated
      if (isAuthenticated === false) {
        // Save params in URL for after login
        const loginParams = new URLSearchParams({
          returnUrl: "/",
          prompt: params.prompt,
          model: params.model,
          styleLibrary: params.styleLibrary,
        });
        router.push(`/login?${loginParams.toString()}`);
        return;
      }

      // Redirect to generation page
      const queryParams = new URLSearchParams({
        prompt: params.prompt,
        model: params.model,
        styleLibrary: params.styleLibrary,
      });

      router.push(`/components/new?${queryParams.toString()}`);
    },
    [isAuthenticated, router]
  );
```

Список компонентів формує серверна сторінка, яка виконує запит до `/api/components` під час рендерингу. Це дозволяє уникнути зайвих клієнтських запитів і забезпечує швидке відображення даних. Фрагмент клієнтської логіки отримання списку компонентів подано в лістингу 2.11.

Лістинг 2.11 –Код отримання списку компонентів

```
const fetchComponents = useCallback(async () => {
    try {
      setIsLoading(true);
      const response = await fetch("/api/components");
      const data = await response.json();

      if (data.success) {
        setComponents(data.components);
      } else {
```

```
          setError(data.error || "Failed to load components");
        }
      } catch {
        setError("Failed to load components");
      } finally {
        setIsLoading(false);
      }
    }, []);
```

Детальна сторінка компонента отримує структуру файлів із маршруту `/api/components/{id}` та відображає їх у вигляді вкладок, списків або попереднього перегляду. Оскільки дані завантажуються на сервері, інтерфейс працює швидко та стабільно.

Клієнтська частина системи розроблена з акцентом на простоту, швидкодію та мінімізацію клієнтської логіки за рахунок активного використання серверних компонентів. Такий підхід дозволяє підтримувати високу продуктивність та масштабованість, а також суттєво спрощує подальший розвиток інтерфейсу.

2.11 Результати розробки програмної системи

У результаті реалізації побудовано повнофункціональну програмну систему, призначену для автоматизованої генерації UI-компонентів на основі великих мовних моделей. Розроблене програмне забезпечення включає серверну та клієнтську частини, інтеграцію із зовнішніми AI-провайдерами, підсистему збереження даних, механізми автентифікації та інтерфейс для взаємодії користувача з основними можливостями платформи.

Система забезпечує реєстрацію і вхід користувачів, включно з підтримкою автентифікації через зовнішні провайдери. Основні функції, пов’язані зі створенням UI-компонентів, реалізовані через окремий інтерфейс, який дозволяє вводити текстовий опис, обирати модель генерації та виконувати запит до мовної моделі. На рисунку 2.16 наведено вигляд інтерфейсу генерації компонентів.

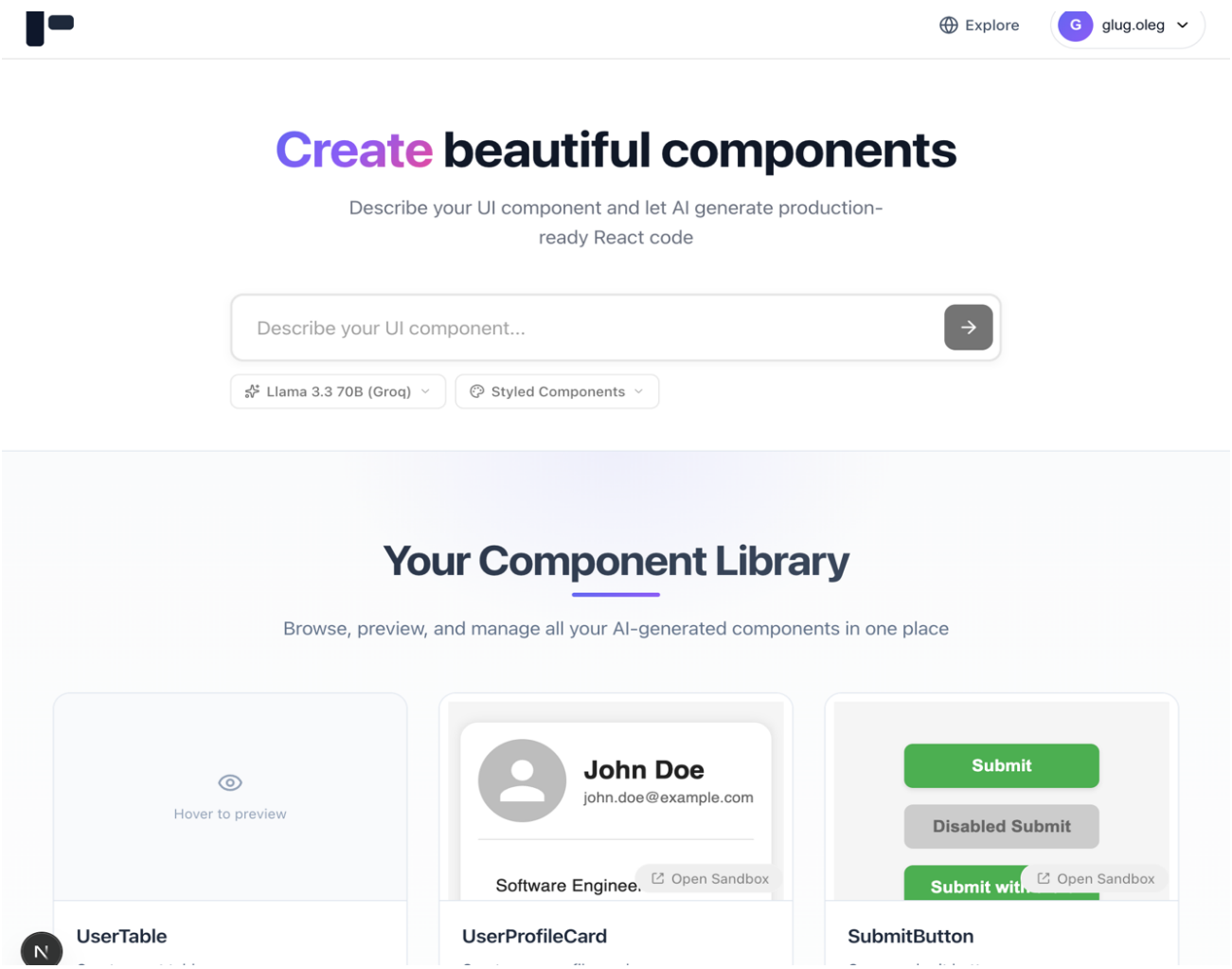


Рисунок 2.16 – Інтерфейс створення нового UI-компонента

Після успішної генерації користувач не повертається до форми створення, а автоматично перенаправляється на сторінку перегляду згенерованого компонента, де може ознайомитися зі структурою файлів, переглянути вихідний код і виконати додаткові операції. На рисунку 2.17 зображено приклад інтерфейсу перегляду даних компонента.

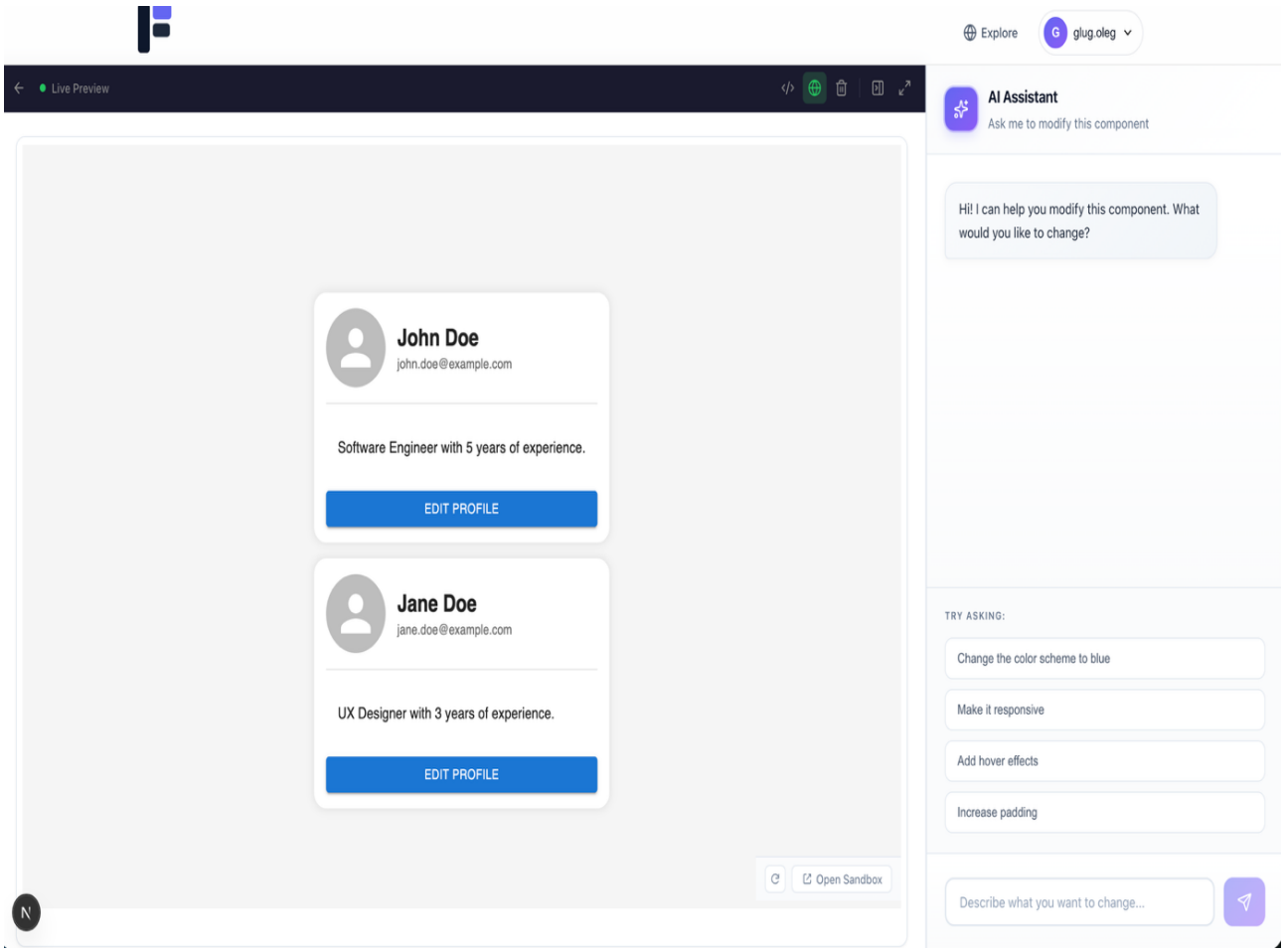


Рисунок 2.17 – Сторінка перегляду згенерованого компонента

Сторінка деталізації надає доступ до вмісту кожного файлу, що був створений мовною моделлю. Користувач може переглядати різні частини компонента, зокрема логіку, стилі та типи, а також виконувати такі операції, як редагування метаданих, публікація або видалення компонента. Зміни зберігаються через відповідні API-маршрути, а інтерфейс оновлюється без перезавантаження сторінки. Код компоненту показаний на рисунку 2.18.

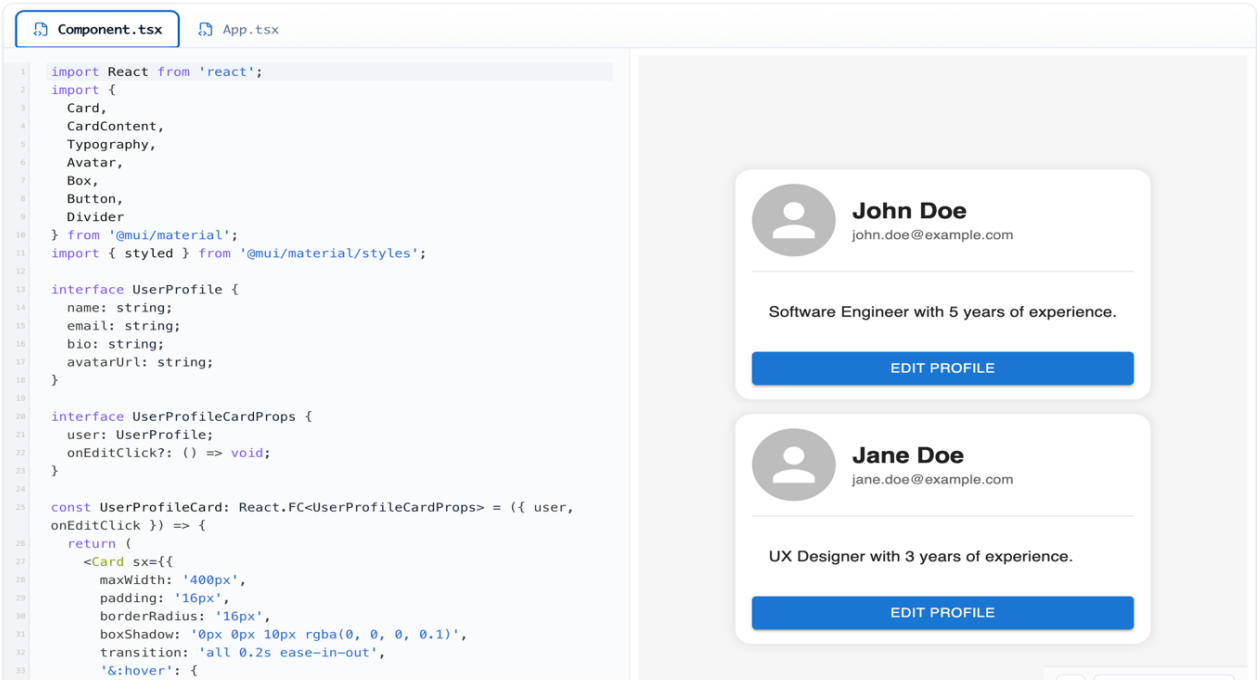


Рисунок 2.18 – Перегляд коду згенерованого компонента

Запроваджено механізм експорту, який дозволяє завантажити компонент у форматі ZIP-архіву. Архів містить повну структуру згенерованих файлів та готовий до інтеграції у проєкт користувача. Приклад завантаження наведено на рисунку 2.19.

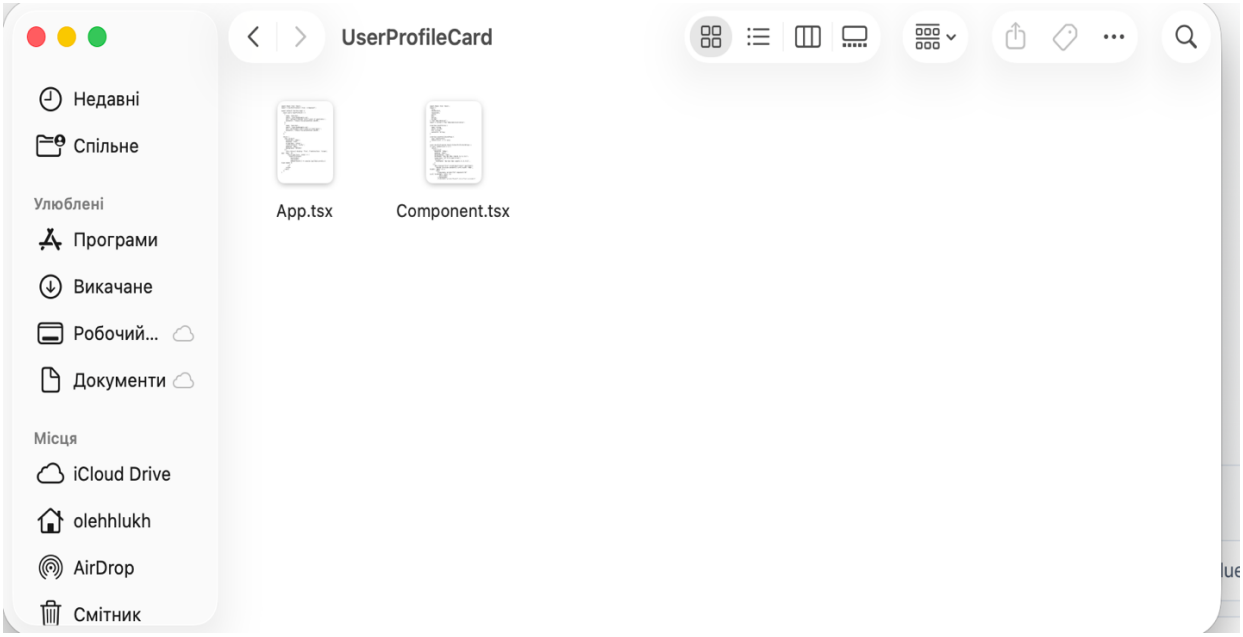


Рисунок 2.19 – Перегляд архіву завантаженого компоненту.

Система підтримує механізм публікації компонентів, що дозволяє користувачам зробити результати власної генерації доступними для інших. Опубліковані компоненти автоматично потрапляють до глобального каталогу Explore Components, де вони представлені у вигляді карточок із коротким описом, датою створення, інформацією про автора та можливістю перегляду повної структури.

Кожен елемент каталогу може бути відкритий у режимі деталізації. Користувач отримує доступ до повного набору файлів, що були створені мовною моделлю, включно з кодом, стилями та службовими типами. Таким чином Explore Components виконує функцію колекції готових UI-рішень, придатних до повторного використання у власних проєктах.

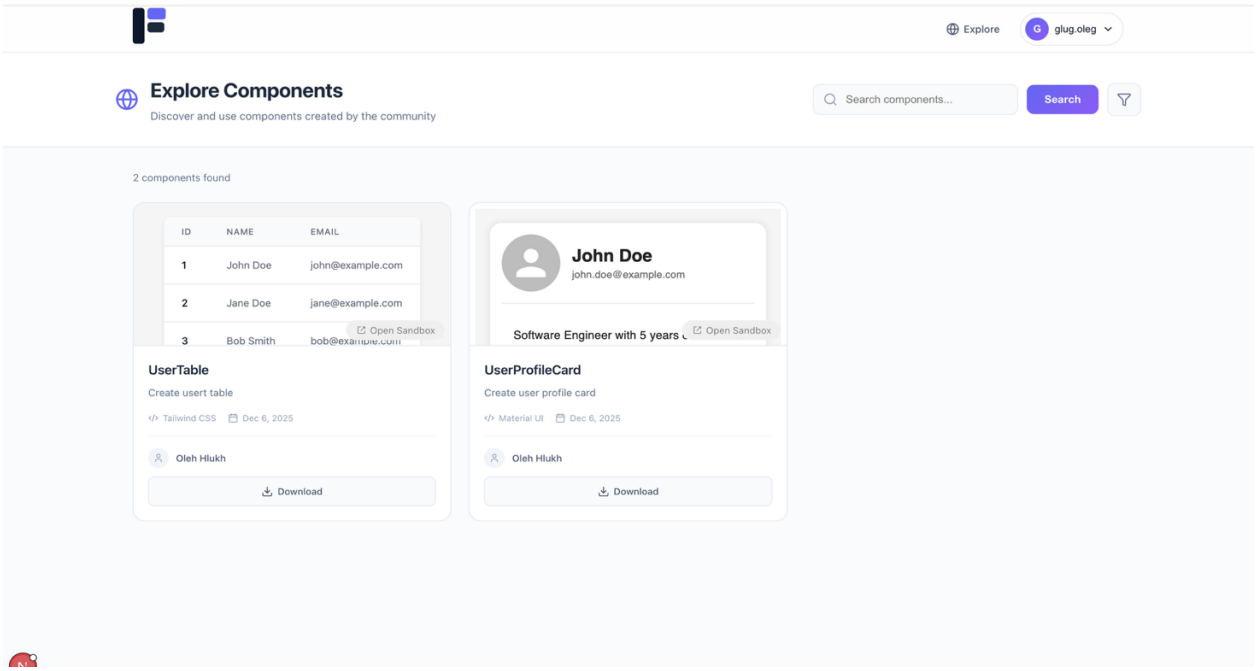


Рисунок 2.20 – Інтерфейс перегляду публічних компонентів (Explore Components)

Реалізована система відповідає встановленим вимогам, забезпечує повний цикл роботи з компонентами та демонструє стабільність під час взаємодії з мовними моделями. Інтерфейс дозволяє швидко переходити між основними функціями, а

структура застосунку забезпечує можливість подальшого розширення та додавання нових можливостей без суттєвих змін архітектури.

2.12 Висновки до розділу

У другому розділі здійснено комплексне проєктування та розробку основних підсистем програмного засобу, призначеного для автоматизованої генерації UI-компонентів на основі великих мовних моделей. Проведено аналіз предметної області та сформовано вимоги до функціонування системи, на підставі яких побудовано її логічну та структурну архітектуру. Описано взаємодію між підсистемами, визначено ключові етапи опрацювання даних від моменту введення користувачем опису компонента до отримання сформованої структури результатів.

Розроблено архітектуру серверної частини, що включає механізми обробки запитів, інтеграцію з мовними моделями та управління збереженням даних. Реалізовано підсистему доступу до бази даних на основі Prisma, яка забезпечує цілісність, узгодженість та типобезпечність операцій зі збереження компонентів, їхніх файлів, а також облікових записів користувачів. Описано модуль автентифікації, який підтримує реєстрацію, вхід за обліковими даними та зовнішніми провайдерами, а також роботу з профілем користувача.

Інтеграцію з великими мовними моделями реалізовано з використанням шаблону «стратегія», що дало змогу відокремити логіку вибору моделі та виконання запитів від загального механізму генерації. Це забезпечує розширюваність системи та можливість додавання нових провайдерів без зміни існуючої архітектури. Окремо описано підхід до побудови файлової структури згенерованих компонентів, який включає формування вихідних файлів, стилів та супровідних даних.

Клієнтську частину системи реалізовано на основі Next.js з використанням серверних та клієнтських компонентів. Забезпечено інтерфейс для введення параметрів генерації, перегляду та управління компонентами, а також взаємодії з API.

Представлено діаграму варіантів використання, що відображає основні дії користувача у системі та узгоджується із функціональністю клієнтського інтерфейсу.

У результаті виконано повний цикл проєктування програмного забезпечення: визначено вимоги, розроблено архітектуру, описано алгоритми взаємодії між підсистемами та представлено ключові рішення, що забезпечують функціональність, масштабованість і розширюваність системи. Проведена розробка створює основу для практичної реалізації програмного продукту та подальшого переходу до експериментального дослідження його ефективності.

3 ТЕСТУВАННЯ, ВПРОВАДЖЕННЯ ТА ПІДТРИМКА ПРОГРАМНОЇ СИСТЕМИ

У цьому розділі представлено процедури перевірки коректності функціонування розробленої програмної системи, порядок її впровадження та особливості подальшої підтримки. Тестування здійснюється з метою підтвердження відповідності реалізованого функціоналу вимогам, визначеним під час проєктування, а також забезпечення стабільності, надійності та передбачуваності роботи застосунку в умовах реального використання. Окрему увагу приділено перевірці основних сценаріїв взаємодії користувача із системою, зокрема операцій автентифікації, генерації UI-компонентів за допомогою великих мовних моделей, управління створеними компонентами та завантаження результатів. У межах розділу також наведено відомості щодо підготовки програмного середовища до експлуатації, розгортання серверної й клієнтської частин, а також особливостей супроводу системи після впровадження [1, 28].

3.1 Методика та підходи до тестування

Проведення тестування ґрунтується на принципах верифікації функціональних вимог, які були сформульовані на етапі проєктування системи. Основною метою методики є перевірка повноти реалізації передбачених можливостей та визначення відхилень у поведінці системи під час типових і граничних сценаріїв роботи. Тестування охоплює два рівні: перевірку окремих модулів і перевірку інтегрованої системи як цілісного програмного продукту [1, 2].

Модульне тестування передбачає оцінювання коректності роботи ізольованих функціональних блоків, зокрема модулів генерації промптів, обробників API-маршрутів, компонентів роботи з базою даних і підсистеми взаємодії з великими мовними моделями. Для кожного модуля визначаються очікувані входи та виходи,

після чого результати порівнюються з розрахунковими. Такий підхід дозволяє локалізувати можливі помилки та забезпечити передбачувану роботу системи на рівні низькорівневих операцій [28].

Інтеграційне тестування спрямоване на перевірку узгодженості взаємодії між компонентами системи, оскільки коректність кожного окремого модуля не гарантує правильного функціонування повної архітектури. Основну увагу приділено взаємодії API-маршрутів із сервісними методами, обробці відповідей мовної моделі, формуванню файлової структури та доступу до даних у базі PostgreSQL. Окремо перевіряється процес авторизації, включно з підтримкою OAuth-провайдерів і сесійним зберіганням, оскільки ці функції забезпечують доступ до захищених частин системи [2, 29].

Під час тестування користувацьких сценаріїв послідовно перевіряється робота застосунку з погляду кінцевого користувача. Такий підхід дає змогу оцінити якість взаємодії з інтерфейсом і реакцію системи на дії, які найчастіше виконуються під час реального використання: реєстрацію, вхід у систему, створення компонента, перегляд і редагування його структури, а також завантаження результатів у вигляді архіву. Для кожного сценарію визначено очікувану поведінку та результат, що забезпечує можливість подальшої автоматизації тестування [25].

Методика також включає перевірку коректності обробки некоректних даних. У межах цього етапу оцінюється здатність системи забезпечувати контроль помилок, дотримання типобезпеки, обробку неправильних параметрів запиту, відсутність доступу без автентифікації та адекватну реакцію на нестандартні відповіді великих мовних моделей. Особливу увагу приділено стабільності системи за умов некоректної поведінки зовнішнього AI-провайдера, оскільки це є типовим ризиком у застосунках, що інтегрують сторонні API [27].

Застосована методика тестування забезпечує системну перевірку функціональності, взаємодії модулів і поведінки користувацьких сценаріїв, що дає

можливість підтвердити готовність програмної системи до впровадження в умовах практичного використання.

3.2 Тестування функціональних модулів

Тестування функціональних модулів спрямоване на перевірку коректності роботи кожного окремого компоненту системи відповідно до його призначення. Основна увага приділяється контролю входів та виходів модулів, оцінюванню внутрішньої логіки та здатності обробляти як коректні, так і некоректні дані. Такий підхід дозволяє виявити помилки на ранніх етапах та забезпечує стабільність роботи системи під час інтеграції [1, 28].

Перевірка модулів серверної частини охоплює тестування API-маршрутів, сервісних функцій і допоміжних механізмів обробки запитів. У межах тестування маршрутів авторизації оцінюється коректність обробки реєстраційних даних, створення користувача у базі даних, виконання валідації введеної інформації та формування сесійних записів. Додатково перевіряється робота механізмів входу через зовнішніх провайдерів, здатність обробляти помилки автентифікації та забезпечувати необхідні обмеження доступу до захищених частин системи [2, 27].

Тестування модулів, відповідальних за взаємодію з базою даних, виконується шляхом симуляції CRUD-операцій над основними сутностями. Під час перевірки роботи моделі користувача оцінюється створення облікових записів, оновлення інформації профілю, видалення пов’язаних записів та збереження цілісності таблиць згідно з установленими правилами каскадування. Для компонентів перевіряється здатність створювати записи із вкладеними файлами, повертати повні структури у відповідь на запити та забезпечувати узгодженість даних під час операцій редагування й видалення [1, 18, 19].

Окремим етапом є тестування модуля інтеграції з мовними моделями. Його коректність перевіряється шляхом подання різних форматів вхідних описів та

оцінювання відповідності результатів очікуваній структурі. Додатково контролюється здатність системи обробляти нестандартні відповіді або помилки API, що є важливою умовою роботи із зовнішніми LLM-провайдерами. Перевіряється формування промптів, робота вибору стратегії та нормалізація отриманих даних, оскільки ці операції безпосередньо впливають на якість та повноту згенерованих компонентів [7, 10, 11].

У межах тестування клієнтської частини перевіряється правильність взаємодії користувача з інтерфейсом, адекватна реакція системи на введені дані, коректність відображення отриманих результатів і стабільність роботи сторінок за умов різних типів навантажень. Додаткові перевірки проводяться для сценаріїв, пов’язаних із динамічним оновленням інтерфейсу, зокрема завантаженням списку компонентів, відкриттям деталей та виконанням операцій над ними. Важливим аспектом є перевірка поведінки інтерфейсу у випадку відсутності даних, помилок сервера або некоректних запитів, оскільки ці ситуації є типовими під час реальної експлуатації [21, 25].

Тестування функціональних модулів продемонструвало здатність системи до стабільної роботи за умов різноманітних сценаріїв і підтвердило відповідність реалізованих компонентів формалізованим вимогам. Отримані результати створюють основу для проведення інтеграційного та системного тестування, описаного у наступних підрозділах.

3.3 Тестування користувацьких сценаріїв

Тестування користувацьких сценаріїв спрямоване на перевірку поведінки системи в умовах реальної взаємодії з користувачем. Основна увага приділяється перевірці типових дій, що виконуються в межах основних процедур роботи застосунку: автентифікації, створення компонентів за допомогою мовної моделі, управління компонентами та завантаження створених файлів. Методика передбачає послідовне виконання кожного сценарію з фіксацією фактичного результату, а також

порівняння його з очікуваною поведінкою відповідно до вимог, сформованих під час проєктування системи.

Першим сценарієм було протестовано процес автентифікації, який включає реєстрацію нового користувача, вхід за електронною поштою та можливість автентифікації через стороннього провайдера. У межах тесту перевірено обробку коректних і некоректних даних, а також реакцію інтерфейсу у разі помилки введення. На рисунку 3.1 наведено інтерфейс форми входу, який використовувався під час проведення тестування.

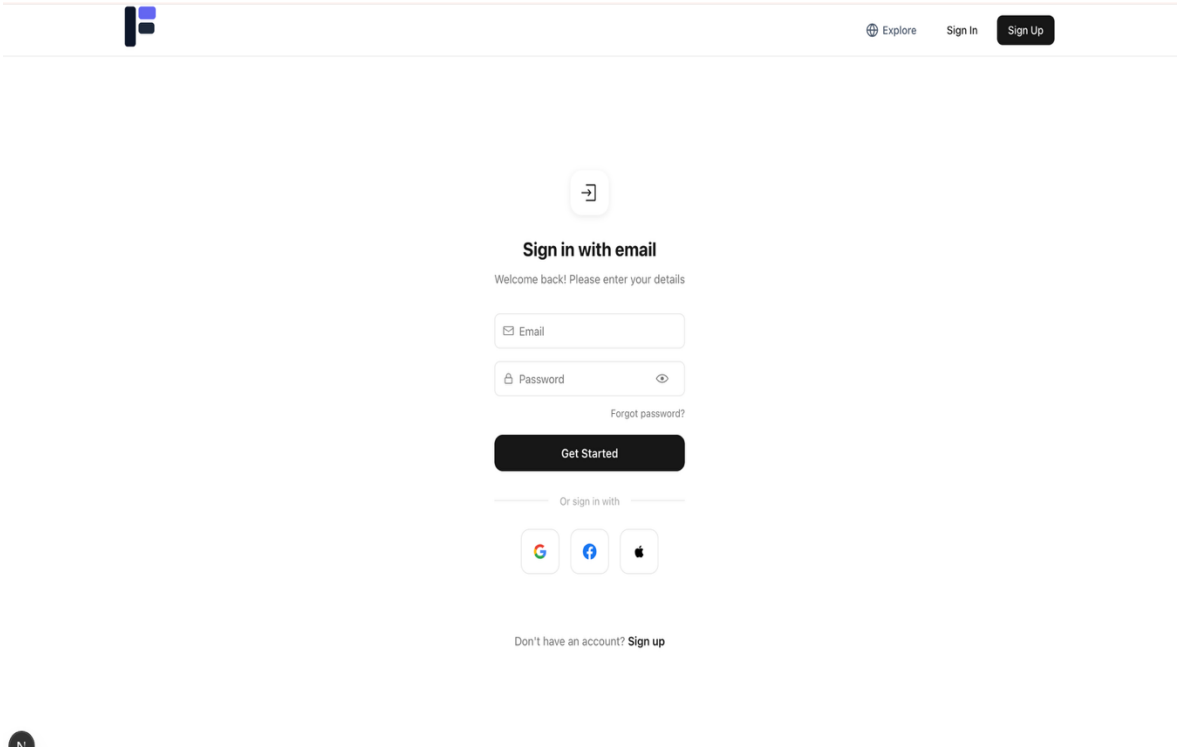


Рисунок 3.1 – Інтерфейс форми входу до системи

Подальше тестування охоплювало сценарій створення нового UI-компонента. Користувач вводить текстовий опис, обирає мовну модель і запускає процес генерації. Перевірено коректність передачі даних на сервер, стабільність роботи під час взаємодії з мовною моделлю та відображення результатів у разі успішної генерації. На

рисунку 3.2 наведено інтерфейс створення компонента під час виконання відповідного сценарію.

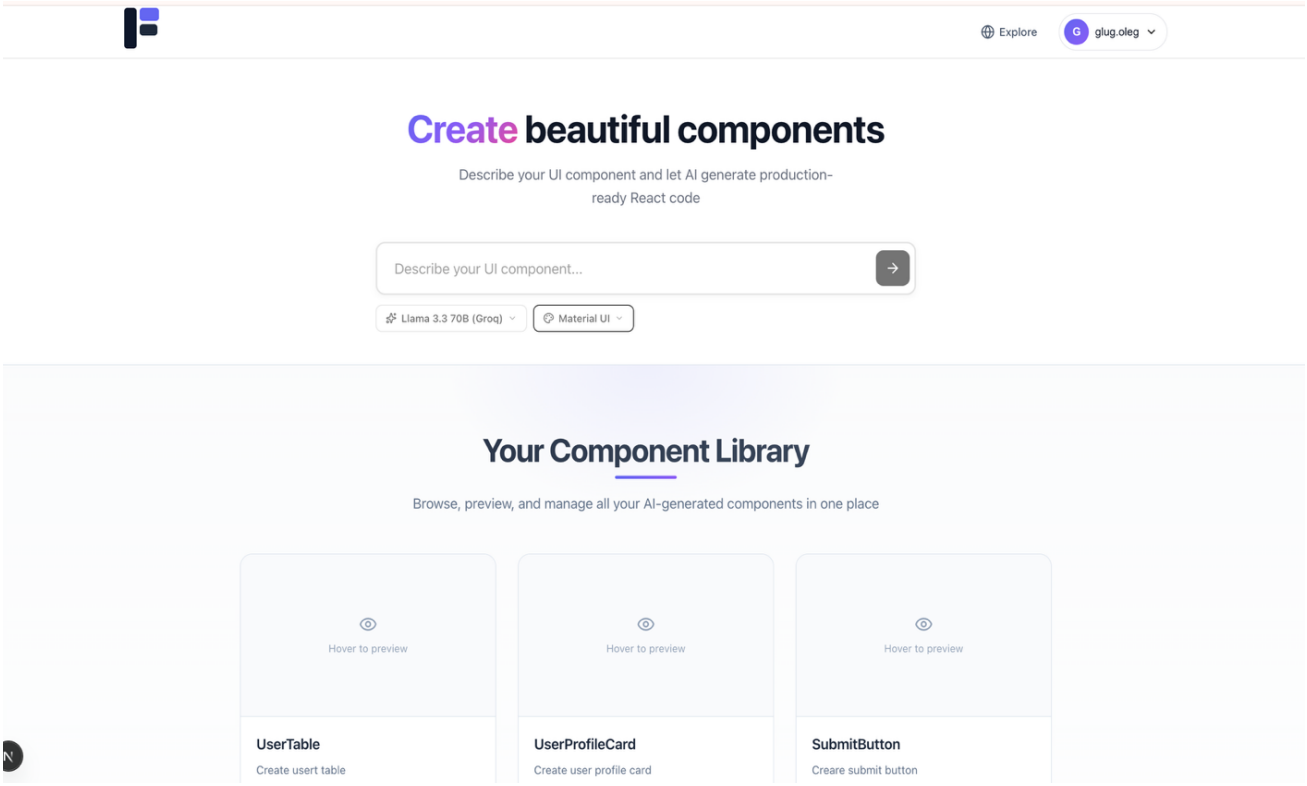


Рисунок 3.2 – Сторінка генерації UI-компонента

Окремо протестовано сценарій перегляду списку створених компонентів. Перевірялося корректне завантаження даних із сервера, відсутність дублювання записів та реакція системи у разі порожнього списку. На рисунку 3.3 зображено інтерфейс перегляду компонентів.

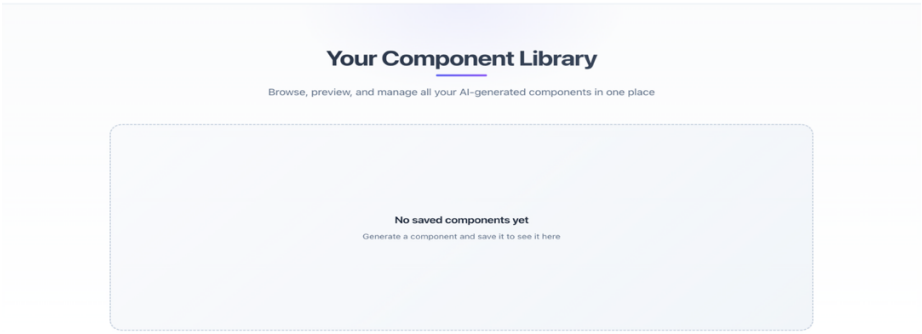


Рисунок 3.3 – Порожній список компонентів.

Сценарій редагування компонента включав зміну метаданих, оновлення опису та збереження змін. Оцінено відповідність реакції інтерфейсу очікуваним результатам, включно з відображенням повідомлення про успішне оновлення. Під час тесту також перевірено можливість помилкового введення даних, що призводить до повернення відповідного повідомлення. Інтерфейс редагування наведено на рисунку 3.4.

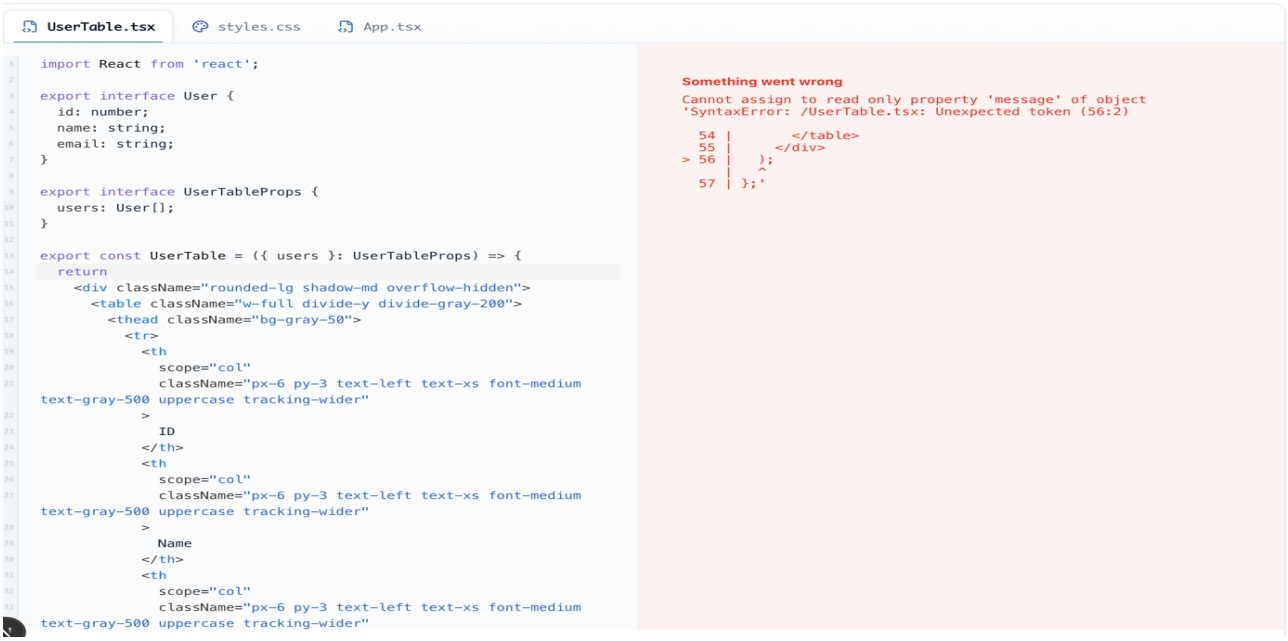


Рисунок 3.4 – Екран редагування UI-компонента

Тестування видалення компонента передбачало перевірку підтвердження користувачем дії, коректність виконання операції на сервері та оновлення списку компонентів після її завершення. У результаті підтверджено, що видалений компонент не з’являється у списку після повторного завантаження сторінки. Вікно з підтвердження видалення показано на рисунку 3.5.

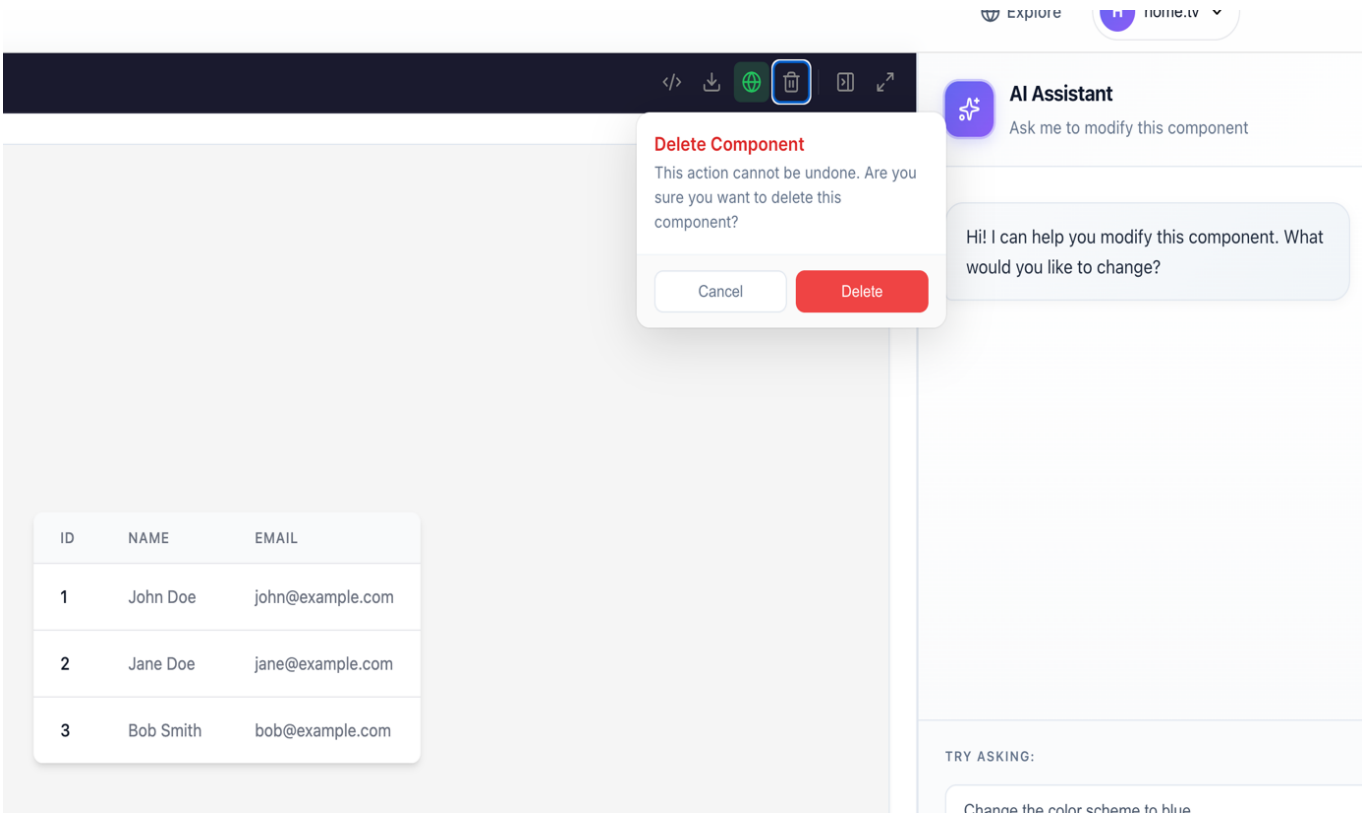


Рисунок 3.5 – Вікно підтвердження видалення

Додатково досліджено поведінку інтерфейсу у випадку помилок, зокрема помилок авторизації, збоїв під час генерації компонента, недоступності сервера або некоректних параметрів запиту. У результаті тестування встановлено, що система забезпечує належну реакцію на відхилення від штатної логіки та коректно інформує користувача про причини помилки.

Користувацькі сценарії продемонстрували стабільність роботи системи, відповідність реалізованих механізмів функціональним вимогам та коректну взаємодію клієнтської й серверної частин. Це підтверджує готовність застосунку до подальшого впровадження.

3.4 Тестування інтеграції та взаємодії підсистем

Інтеграційне тестування спрямоване на перевірку узгодженості роботи окремих модулів системи під час їх взаємодії та забезпечення стабільності функціонування програмного комплексу в умовах реального використання. Основна увага приділяється аналізу коректності обміну даними між клієнтською та серверною частинами, взаємодії сервера з базою даних і сервісами зовнішніх мовних моделей, а також перевірці роботи допоміжних механізмів, таких як логування, обмеження частоти запитів і перегляд компонентів у режимі Sandbox [2, 25].

Під час тестування взаємодії клієнтської частини з сервером проаналізовано коректність формування й обробки HTTP-запитів, відповідність структури переданих даних типам і реакцію клієнта на помилки сервера. У межах цього етапу підтверджено, що інтерфейс коректно ініціює основні операції, включно зі створенням нового компонента, оновленням метаданих, видаленням записів і завантаженням структурованих файлів. Особливу увагу приділено поведінці системи після завершення генерації, коли сервер повертає ідентифікатор створеного компонента, а клієнтська частина автоматично виконує перенаправлення на сторінку його деталізації. Результати перевірки показали повну відповідність описаній у проєкті логіці роботи [27].

Тестування зв’язку сервера з базою даних охоплювало аналіз транзакційної цілісності операцій і здатності моделі зберігати вкладені файли компонента. Перевірено коректність роботи каскадних операцій під час видалення записів і обробку виняткових ситуацій, пов’язаних із відсутністю даних або неправильно сформованими параметрами запиту. Усі перевірки підтвердили узгоджену роботу API та бази даних [7, 10].

Особливу увагу приділено інтеграції з мовною моделлю. Перевірено механізм формування промптів, які надсилаються до зовнішнього LLM-провайдера, обробку нестандартних відповідей і реакцію системи у випадку помилок мережевої взаємодії.

Система правильно опрацьовує неповні чи некоректні структури, забезпечує повернення користувачеві інформативних повідомлень і фіксує всі виняткові випадки у журналі подій.

Перевірка механізму обмеження частоти запитів показала, що система коректно реагує на надмірну активність користувача та повертає відповідні повідомлення. Тестування режиму Sandbox підтвердило ізольованість середовища та правильність відображення структури згенерованих файлів.

Узагальнену модель процесу тестування інтеграції подано на рисунку 3.6. На ній наведено основні етапи виконання тестів, включно з підготовкою середовища, перевіркою критичних точок взаємодії та аналізом отриманих результатів.

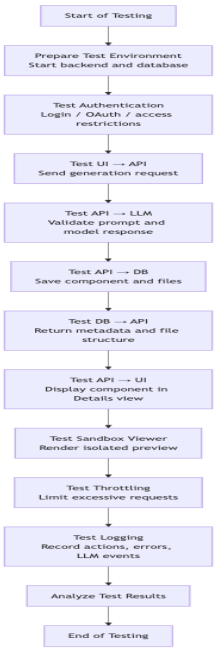


Рисунок 3.6 – Діаграма процесу інтеграційного тестування системи

Результати інтеграційного тестування підтвердили узгоджену взаємодію всіх підсистем та відповідність роботи програмної системи вимогам, визначеним на етапі проєктування.

3.5 Впровадження системи

Впровадження програмної системи передбачає комплекс заходів, спрямованих на її розгортання у робочому середовищі та забезпечення можливості подальшої експлуатації. На цьому етапі визначаються вимоги до інфраструктури, налаштовується серверне оточення, виконується підготовка бази даних, а також забезпечується коректна робота клієнтської частини в середовищі вебсерверів. Загальна мета впровадження полягає у створенні стабільного та доступного середовища, у якому система може функціонувати відповідно до вимог та навантаження, що очікується під час її використання [1, 29].

Першим етапом впровадження стало розгортання серверної частини, реалізованої на основі Node.js із використанням фреймворку Next.js у режимі серверних обчислень. Для розміщення застосунку використано контейнеризоване середовище, що забезпечує повторюваність конфігурацій і спрощує масштабування. Серверна частина потребує встановлення змінних середовища, зокрема секретних ключів для використання мовної моделі, параметрів підключення до бази даних і конфігурацій OAuth-провайдерів для автентифікації. Після розгортання серверної частини виконано первинну ініціалізацію бази даних, створення міграцій та перевірку коректності структури таблиць [15, 18, 19].

Клієнтська частина застосунку впроваджується разом із серверною, оскільки у межах обраної архітектури Next.js інтерфейс і серверні маршрути функціонують як єдина система. Під час впровадження перевірено, що статичні файли коректно збираються, формуються маршрути сторінок і забезпечується відтворення інтерфейсу на стороні сервера. Окрему увагу приділено налаштуванню кешування та попередньому рендерингу для зменшення затримок і підвищення ефективності роботи застосунку [15, 16].

Окрім інфраструктурної частини, важливою складовою впровадження стали налаштування системи журналювання та моніторингу. Логування дозволяє

відстежувати ключові події, зокрема помилки, запити до мовної моделі та спроби доступу до захищених ресурсів, що суттєво спрощує подальший супровід системи. Для контролю працездатності застосунку впроваджено моніторинг використання ресурсів та часу відповіді, що дає змогу швидко реагувати на потенційні проблеми або деградацію продуктивності.

Завершальним етапом упровадження стала перевірка роботи системи в реальному середовищі. Було виконано тестове розгортання, у межах якого перевірено стабільність роботи всіх функцій, відповідність інтерфейсу вимогам щодо продуктивності та коректність взаємодії між підсистемами. Після цього система була підготовлена до подальшої експлуатації користувачами [2, 27].

Виконані заходи впровадження засвідчили готовність системи до використання в реальному середовищі та забезпечили основу для подальшого супроводу й розвитку програмного продукту.

3.6 Підтримка та супровід системи

Підтримка та супровід програмної системи охоплюють комплекс заходів, спрямованих на забезпечення стабільного функціонування застосунку після його впровадження, а також на створення умов для подальшого розвитку й адаптації до змін вимог користувачів. У межах супроводу здійснюється моніторинг працездатності, аналіз журналів подій, виправлення помилок, оновлення компонентів системи та оптимізація її роботи.

Після розгортання системи важливим завданням є постійний контроль за її станом. Для цього використовується вбудований механізм логування, що забезпечує фіксацію ключових подій, взаємодії з мовною моделлю, операцій користувачів та виняткових ситуацій. Аналіз логів дозволяє оперативно виявляти потенційні проблеми, визначати закономірності їх виникнення та здійснювати корекцію роботи системи без переривання її функціонування. Журналювання також слугує основою для

вдосконалення системи безпеки, зокрема виявлення несанкціонованих спроб доступу чи аномальної активності.

У процесі супроводу виконується оновлення структурних компонентів системи, як-от залежностей серверної частини, бібліотек клієнтського інтерфейсу та конфігурацій середовища. Зважаючи на використання зовнішніх мовних моделей, оновлення охоплює також адаптацію механізму формування промптів до нових або модифікованих API, що забезпечує сумісність системи із сучасними версіями моделей. Окрему увагу приділено питанням продуктивності, оскільки зміни у зовнішніх сервісах можуть вимагати повторного налаштування механізмів кешування чи обробки відповідей.

Підтримка охоплює також роботу з базою даних, яка потребує періодичного оновлення міграцій, оптимізації індексів та контролю за обсягом збережених файлів. У межах системи, що оперує великою кількістю згенерованих компонентів, важливим є забезпечення стабільної роботи механізмів збереження та очищення. Це дає змогу уникнути надмірного накопичення даних, зменшити навантаження на сервер і забезпечити швидкий доступ до сховища.

Підтримка інтерфейсу користувача передбачає аналіз відгуків та можливих проблем взаємодії, що можуть виникати під час використання системи. Оновлення механізмів перегляду компонентів, удосконалення Sandbox Viewer та оптимізація роботи інтерфейсу спрямовані на покращення користувацького досвіду та забезпечення адаптивності системи до різних робочих сценаріїв.

У межах супроводу передбачено можливість масштабування системи, яке може включати розгортання додаткових інстансів серверної частини, перенесення бази даних на окремий сервер або використання балансування навантаження. Така гнучкість забезпечує можливість збільшення кількості користувачів та обсягів даних без зниження продуктивності.

Підтримка системи є безперервним процесом, що поєднує аналіз роботи, виправлення помилок, удосконалення функціональності та адаптацію до нових

технологічних вимог. Завдяки впровадженим механізмам логування, гнучкій архітектурі та можливості поступового масштабування забезпечується довготривала експлуатація застосунку та його подальший розвиток.

4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

У цьому розділі розглянуто питання охорони праці та забезпечення безпеки в надзвичайних ситуаціях під час розроблення та експлуатації програмної системи. Проаналізовано основні потенційні небезпеки, пов’язані з роботою за персональним комп’ютером, а також визначено вимоги до організації безпечного робочого місця розробника. Окрему увагу приділено заходам цивільного захисту та діям персоналу в умовах виникнення надзвичайних ситуацій відповідно до чинних нормативно-правових документів.

4.1 Охорона праці та пожежна безпека під час розроблення програмного забезпечення

Під час розроблення програмного забезпечення автоматизованої системи генерації UI-компонентів для вебзастосунків враховувалися вимоги охорони праці та пожежної безпеки з метою забезпечення безпечних і нешкідливих умов праці розробника, а також мінімізації негативного впливу виробничих факторів, характерних для роботи за персональним комп’ютером.

Проєктування, реалізація та тестування програмної системи здійснювалися з використанням персонального комп’ютера, мережевого обладнання та периферійних пристроїв. Такий вид діяльності належить до інтелектуальної праці, що характеризується підвищеним зоровим, розумовим та нервово-емоційним навантаженням. Відповідно до Закону України «Про охорону праці», охорона праці є системою заходів, «спрямованих на збереження життя, здоров’я і працездатності людини в процесі трудової діяльності» [31]. Зазначене положення визначає необхідність урахування фізіологічних та психофізіологічних особливостей людини під час організації процесу розроблення програмного забезпечення.

Аналіз умов праці показав, що при роботі над програмним продуктом основними шкідливими та небезпечними факторами є тривале перебування в статичному положенні, обмежена рухова активність, інтенсивне зорове навантаження під час тривалої концентрації погляду на екрані монітора, а також підвищене нервово-емоційне напруження, пов’язане з аналізом складних логічних конструкцій, пошуком помилок у коді та необхідністю прийняття технічних рішень у стислі терміни. Крім того, потенційну небезпеку становлять фактори електробезпеки, зумовлені експлуатацією електронної обчислювальної техніки.

У процесі розроблення програмної системи особлива увага приділялася організації робочого місця. Відповідно до вимог санітарного законодавства України, умови праці повинні забезпечувати оптимальні параметри мікроклімату, що не призводять до погіршення самопочуття працівника та зниження його працездатності [32]. Підтримання комфортної температури повітря та допустимого рівня вологості дозволяє зменшити втому, запобігти перегріванню або переохолодженню організму та створити сприятливі умови для тривалої інтелектуальної діяльності.

Важливим чинником охорони праці є правильна організація освітлення робочого місця. Відповідно до ДБН В.2.5-28:2018, освітлення повинно бути достатнім для виконання зорових робіт і не створювати засліплювальної дії або відбитих відблисків на екранах відеотерміналів [33]. Під час роботи над програмним забезпеченням використовувалося комбіноване освітлення, що дозволяло рівномірно освітлювати робочу зону та зменшувати напруження зору, особливо під час тривалого читання програмного коду та технічної документації.

Раціональна організація режиму праці та відпочинку є важливою складовою охорони праці під час виконання робіт, пов’язаних із використанням персонального комп’ютера. Тривала безперервна робота за екраном може призводити до перевтоми, зниження концентрації уваги та помилок у програмному коді. Тому під час розроблення програмної системи дотримувався режим чергування роботи та

короткочасних перерв, що сприяє зниженню зорової і нервово-емоційної напруги та відповідає загальним вимогам гігієни праці, визначеним законодавством України [31].

Окрему увагу було приділено питанням електробезпеки. Відповідно до вимог Закону України «Про охорону праці», працівники повинні бути захищені від дії небезпечних факторів, зокрема електричного струму [31]. Під час експлуатації комп'ютерної техніки використовувалося лише справне обладнання, що підключалося до електромережі із заземленням. Не допускалося використання пошкоджених кабелів, несправних розеток або подовжувачів. Перед початком роботи проводився візуальний огляд технічного стану обладнання, що дозволяло знизити ризик виникнення аварійних ситуацій.

Пожежна безпека під час розроблення програмного забезпечення забезпечувалася відповідно до Закону України «Про пожежну безпеку», який визначає пожежну безпеку як стан захищеності життя та здоров'я людей від пожеж [34]. Приміщення, у якому виконувалися роботи, належить до приміщень з підвищеною електропожежною небезпекою у зв'язку з використанням комп'ютерної техніки, блоків живлення та мережевого обладнання.

Згідно з Правилами пожежної безпеки в Україні, експлуатація електрообладнання повинна здійснюватися з дотриманням установлених вимог, зокрема забороняється перевантаження електромереж, використання несправних електроприладів та залишення увімкненого обладнання без нагляду [35]. Після завершення роботи комп'ютерна техніка відключалася від електромережі, а в приміщенні забезпечувалася можливість оперативного знеструмлення у разі виникнення аварійної ситуації.

У разі виникнення пожежі або загрози загоряння передбачалося негайне припинення роботи, відключення електроживлення та повідомлення відповідних аварійних служб. Дотримання зазначених заходів відповідає вимогам чинних нормативних документів і спрямоване на мінімізацію можливих наслідків надзвичайних ситуацій.

Отже, під час розроблення програмного забезпечення автоматизованої системи генерації UI-компонентів були комплексно враховані вимоги охорони праці та пожежної безпеки відповідно до чинної нормативно-правової бази України. Реалізовані організаційні та технічні заходи сприяють створенню безпечних умов праці, зниженню впливу шкідливих і небезпечних факторів, а також забезпечують збереження здоров'я і працездатності розробника під час виконання програмно-інженерних робіт.

4.2 Особливості роботи та розлади здоров'я користувачів комп'ютерів, що формуються під впливом роботи за комп'ютером

Професійна діяльність користувачів комп'ютерної техніки, зокрема фахівців у галузі програмної інженерії та веброзробки, характеризується тривалим використанням персональних комп'ютерів і екранних пристроїв у поєднанні з високою інтенсивністю розумової праці. Такий вид діяльності належить до робіт із переважанням інтелектуального навантаження, для яких типовими є постійна концентрація уваги, обробка значних обсягів інформації, багатозадачність і обмежена рухова активність. Сукупність зазначених чинників формує специфічні умови праці, що мають безпосередній вплив на функціональний стан організму людини та зумовлюють ризик розвитку стійких розладів здоров'я.

Особливості умов діяльності людини розглядаються як один із визначальних чинників стану її здоров'я та працездатності, оскільки характер і тривалість впливу виробничих факторів безпосередньо відображаються на фізіологічних і психофізіологічних показниках організму [36]. У випадку комп'ютерної праці цей вплив має комплексний характер і обумовлений поєднанням зорових, статичних та психоемоційних навантажень.

Однією з ключових особливостей роботи за комп'ютером є підвищене навантаження на зорову систему. Тривале фокусування погляду на екрані,

необхідність постійного сприйняття текстової та графічної інформації, робота з дрібними елементами інтерфейсу, а також часті зміни об’єктів візуальної уваги створюють умови для розвитку зорової втоми. У користувачів комп’ютерів це проявляється у вигляді сухості та печіння в очах, тимчасового зниження гостроти зору, головного болю та загального дискомфорту. Такі порушення, як правило, мають функціональний характер, однак за тривалої систематичної дії несприятливих факторів вони накопичуються та призводять до зниження якості виконання професійних завдань.

Поряд із зоровим навантаженням суттєвий вплив на здоров’я користувачів комп’ютерної техніки має тривале перебування у статичному положенні. Обмежена рухова активність у поєднанні з нерівномірним навантаженням на м’язи шиї, плечового пояса та спини сприяє розвитку порушень з боку опорно-рухового апарату. Найбільш поширеними проявами є біль у шийному та поперековому відділах хребта, м’язове перенапруження, відчуття скутості та швидка фізична втома. На початкових етапах такі порушення можуть не супроводжуватися вираженими симптомами, однак у довгостроковій перспективі вони знижують працездатність і негативно впливають на загальний стан здоров’я.

Характерною особливістю комп’ютерної праці є також значне навантаження на верхні кінцівки, пов’язане з монотонними повторюваними рухами під час роботи з клавіатурою та маніпуляторами введення. Тривале виконання однотипних операцій призводить до перенапруження м’язів кистей і передпліч, появи больових відчуттів у ділянці зап’ясть та зниження точності рухів. Такі прояви негативно відображаються на ефективності діяльності користувачів і підвищують імовірність помилок у процесі виконання завдань.

Важливим компонентом впливу комп’ютерної праці на здоров’я є психоемоційне навантаження. Інтенсивна розумова діяльність, необхідність постійної концентрації уваги, обробки значних обсягів інформації та прийняття рішень у стислі терміни створюють умови для розвитку нервово-емоційного напруження. У

користувачів комп’ютерів це може проявлятися у вигляді швидкої втомлюваності, дратівливості, зниження концентрації уваги, порушень сну та емоційного виснаження. Функціональний стан людини в таких умовах визначає рівень її працездатності та здатність ефективно виконувати професійні завдання [37].

Психоемоційне навантаження має складну структуру та формується не лише інтенсивністю розумової діяльності, але й специфікою цифрового середовища. Постійне перемикання між завданнями, багатозадачність, робота з великими обсягами інформації та часті переривання робочого процесу сприяють розвитку хронічного когнітивного перенапруження. За даними Всесвітньої організації охорони здоров’я, тривалий вплив психоемоційних факторів робочого середовища пов’язаний зі зниженням працездатності та підвищеним ризиком розвитку порушень психічного здоров’я [38].

Додатковим чинником психоемоційного напруження є високий рівень відповідальності за результати діяльності та необхідність постійного контролю якості виконуваних операцій. У користувачів комп’ютерної техніки це може спричиняти формування станів тривожності, зниження мотивації та емоційне вигорання. У рекомендаціях Міжнародної організації праці психосоціальні ризики визначаються як важлива складова умов праці, що повинна враховуватися нарівні з фізичними та ергономічними чинниками [39].

Зорові, опорно-рухові та психоемоційні розлади, що формуються під впливом роботи за комп’ютером, мають виражений накопичувальний характер. Тривалий вплив навіть помірних за інтенсивністю несприятливих умов поступово знижує адаптаційні можливості організму, що проявляється у стійкому зменшенні працездатності, зростанні загальної втоми та погіршенні когнітивних функцій. У навчальній літературі з безпеки життєдіяльності зазначається, що функціональний стан людини є результатом дії умов діяльності та визначає її можливості ефективно протистояти навантаженням у процесі праці [37].

Особливе значення має взаємозв’язок між фізіологічними та психофізіологічними чинниками комп’ютерної праці. Зорове перенапруження та статичні навантаження підсилюють психоемоційне виснаження, тоді як хронічна втома погіршує якість сприйняття інформації та знижує ефективність зорової роботи. Такий взаємний вплив факторів ускладнює своєчасне виявлення негативних змін у стані здоров’я та сприяє формуванню стійких функціональних порушень.

У межах аналізу небезпечних факторів, що можуть виникати під час експлуатації комп’ютерної техніки та електрообладнання, окрему увагу приділено ризику короткого замикання електричних мереж. «У разі руйнування будівель можливими є обриви проводів у середині приміщень, що при збереженні кабельних мереж призведе до короткого замикання, а ті, у свою чергу, можуть призвести до пожеж» [37]. За таких умов коротке замикання розглядається як вторинний небезпечний фактор, здатний спричинити займання ізоляційних матеріалів та пошкодження електронного обладнання.

Отже, особливості роботи користувачів комп’ютерів формують комплекс умов, що впливають на стан здоров’я та працездатність людини. Зорові, опорно-рухові та психоемоційні розлади, які виникають під впливом комп’ютерної праці, мають поступовий і накопичувальний характер та потребують урахування під час організації професійної діяльності. Аналіз зазначених особливостей є необхідною передумовою для обґрунтування заходів з охорони праці, спрямованих на збереження здоров’я користувачів комп’ютерної техніки та підтримання належного функціонального стану організму.

ВИСНОВКИ

У роботі здійснено комплексне дослідження, проєктування та реалізацію програмної системи автоматизованої генерації UI-компонентів для вебзастосунків із використанням великих мовних моделей. На основі проведеного аналізу, розробленої архітектури та експериментальної перевірки сформульовано такі узагальнені висновки.

У першому розділі проаналізовано сучасний стан предметної області та визначено ключові проблеми, що існують у процесі створення інтерфейсних компонентів у веброзробці. Встановлено, що ручне формування UI-елементів є трудомістким та схильним до помилок, а існуючі автоматизовані рішення не забезпечують достатньої гнучкості, інтеграції з дизайн-системами та адаптивності до специфіки реальних проєктів. Проведений огляд інструментів та технологій підтвердив актуальність застосування генеративних можливостей LLM для автоматизації процесів розробки інтерфейсів. Сформульовано функціональні та нефункціональні вимоги до системи та побудовано концептуальні моделі, що визначили основу подальшого проєктування.

У другому розділі розроблено архітектуру програмної системи, що включає клієнтську та серверну частини, підсистему автентифікації, модуль генерації коду на основі стратегій взаємодії з різними мовними моделями та систему збереження структурованих файлів компонентів. Запропонована багаторівнева архітектура забезпечує модульність, гнучкість і масштабованість програмного продукту. Реалізація серверної частини виконана на основі Next.js та Prisma ORM, що гарантує цілісність і типобезпечність даних. Використання шаблонів «стратегія», «адаптер» і «фасад» дало змогу ізолювати логіку генерації, спростити розширюваність системи та забезпечити підтримку декількох AI-провайдерів. Клієнтська частина реалізована з використанням серверних компонентів і забезпечує інтуїтивну взаємодію користувача з основними функціями системи. У результаті розроблено повнофункціональний

застосунок, що дозволяє створювати, переглядати, редагувати, публікувати й експортувати UI-компоненти.

У третьому розділі проведено модульне, інтеграційне та сценарне тестування програмної системи. Перевірено коректність роботи API-маршрутів, підсистем автентифікації, генерації компонентів, збереження файлів та відображення даних на клієнтській частині. Окремо протестовано взаємодію з великими мовними моделями та поведінку системи у випадках помилок. Тестування підтвердило стабільність функціонування системи в умовах реального використання та відповідність реалізованих механізмів проєктним вимогам. Описано процес впровадження застосунку, налаштування середовища, створення бази даних та забезпечення безперервної підтримки й моніторингу.

У ході виконання магістерської роботи досягнуто поставленої мети – створено програмну систему автоматизованої генерації UI-компонентів, що здатна перетворювати текстові описи користувача на структуровані, типізовані та готові до інтеграції компоненти сучасних вебзастосунків. Розроблена система забезпечує зменшення трудовитрат розробників, підвищення консистентності інтерфейсів і прискорення створення UI-елементів завдяки застосуванню можливостей великих мовних моделей.

Практична цінність роботи полягає у можливості використання створеної системи в процесі розробки вебпродуктів, її інтеграції у CI/CD-конвеєри, застосуванні для автоматизації рутинних операцій і швидкого формування прототипів інтерфейсів. Результати роботи можуть бути основою для подальших досліджень у напрямі генеративного проєктування інтерфейсів, удосконалення механізмів валідації згенерованого коду, автоматизованого аналізу якості та розширення підтримки додаткових фреймворків і дизайн-систем.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Sommerville I. Software Engineering. 10th ed. – Boston : Pearson, 2016. – 816 p.
2. Pressman R. S., Maxim B. R. Software Engineering: A Practitioner’s Approach. 9th ed. – New York : McGraw-Hill, 2020. – 736 p.
3. Bass L., Clements P., Kazman R. Software Architecture in Practice. 4th ed. – Boston : Addison-Wesley, 2021. – 560 p.
4. Martin R. C. Clean Architecture: A Craftsman’s Guide to Software Structure and Design. – Boston : Pearson, 2018. – 432 p.
5. ISO/IEC 25010:2011. Systems and software quality models. – Geneva : ISO, 2011.
6. Vaswani A., Shazeer N., Parmar N. et al. Attention Is All You Need. – In: *Advances in Neural Information Processing Systems*. – 2017.
7. Brown T., Mann B., Ryder N. et al. Language Models are Few-Shot Learners. – In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. – 2020.
8. OpenAI. GPT-4 Technical Report [Електронний ресурс]. – 2023. – Режим доступу: https://openai.com/research/gpt-4.
9. OpenAI. GPT-4o System Card [Електронний ресурс]. – 2024. – Режим доступу: https://openai.com/index/gpt-4o-system-card/.
10. Zhao W., Liu J., Zhou Y. et al. A Survey of Large Language Models [Електронний ресурс]. – 2023. – Режим доступу: https://arxiv.org/abs/2303.18223.
11. Chen M., Tworek J., Jun H. et al. Evaluating Large Language Models Trained on Code [Електронний ресурс]. – 2021. – Режим доступу: https://arxiv.org/abs/2107.03374.
12. GitHub. GitHub Copilot Documentation [Електронний ресурс]. – Режим доступу: https://docs.github.com/copilot.
13. Microsoft. AI-assisted software development overview [Електронний ресурс]. – 2023. – Режим доступу: https://learn.microsoft.com.

14. Li Y., Chen Z., Zhang H. Code Generation with Large Language Models: A Survey [Електронний ресурс]. – 2024. – Режим доступу: https://ieeexplore.ieee.org.
15. Vercel. Next.js Documentation [Електронний ресурс]. – 2024. – Режим доступу: https://nextjs.org/docs.
16. Vercel. Next.js App Router Overview [Електронний ресурс]. – 2024. – Режим доступу: https://nextjs.org/docs/app.
17. Prisma. Prisma ORM Documentation [Електронний ресурс]. – 2024. – Режим доступу: https://www.prisma.io/docs.
18. Prisma. Prisma Schema Reference [Електронний ресурс]. – 2024. – Режим доступу: https://www.prisma.io/docs/orm/prisma-schema.
19. PostgreSQL Global Development Group. PostgreSQL 16 Documentation [Електронний ресурс]. – 2024. – Режим доступу: https://www.postgresql.org/docs/.
20. PostgreSQL Global Development Group. PostgreSQL Architecture Overview [Електронний ресурс]. – Режим доступу: https://www.postgresql.org/docs/current/tutorial-arch.html.
21. Mozilla Developer Network. Modern Web APIs [Електронний ресурс]. – Режим доступу: https://developer.mozilla.org.
22. World Wide Web Consortium. Web Components Specification [Електронний ресурс]. – 2022. – Режим доступу: https://www.w3.org/TR/components-intro/.
23. Radix UI. Accessible UI Primitives for React [Електронний ресурс]. – Режим доступу: https://www.radix-ui.com.
24. Google. Material Design Guidelines [Електронний ресурс]. – 2024. – Режим доступу: https://material.io.
25. Nielsen J. Usability Engineering. – San Diego : Academic Press, 1994. – 362 p.
26. ISO 9241-210:2019. Ergonomics of human-system interaction – Human-centred design for interactive systems. – Geneva : ISO, 2019.
27. OWASP Foundation. OWASP Top 10 Web Application Security Risks [Електронний ресурс]. – 2023. – Режим доступу: https://owasp.org.

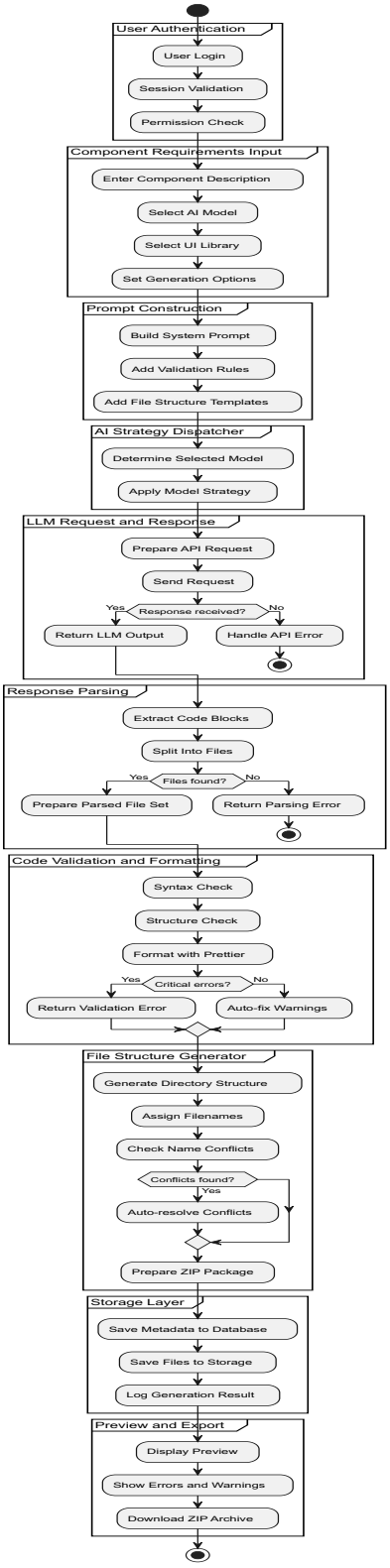
28. Myers G., Sandler C., Badgett T. The Art of Software Testing. 3rd ed. – Hoboken : Wiley, 2011. – 256 p.
29. Newman S. Building Microservices. 2nd ed. – Sebastopol : O'Reilly Media, 2021. – 600 p.
30. Vercel. Deploying and Scaling Next.js Applications [Електронний ресурс]. – 2024. – Режим доступу: https://vercel.com/docs.
31. Про охорону праці : Закон України від 14.10.1992 № 2694-XII (із змінами) [Електронний ресурс]. – Режим доступу: https://zakon.rada.gov.ua/go/2694-12 (дата звернення: 13.12.2025).
32. Державні санітарні норми мікроклімату виробничих приміщень ДСН 3.3.6.042-99 : затв. постановою Головного державного санітарного лікаря України від 01.12.1999 № 42 [Електронний ресурс]. – Режим доступу: https://zakon.rada.gov.ua/go/va042282-99 (дата звернення: 13.12.2025).
33. ДБН В.2.5-28:2018. Природне і штучне освітлення. – Київ : Мінрегіон України, 2018 [Електронний ресурс]. – Режим доступу: https://e-construction.gov.ua/laws_detail/3074958732556240833 (дата звернення: 13.12.2025).
34. Про пожежну безпеку : Закон України від 17.12.1993 № 3745-XII (із змінами) [Електронний ресурс]. – Режим доступу: https://zakon.rada.gov.ua/go/3745-12 (дата звернення: 13.12.2025).
35. Про затвердження Правил пожежної безпеки в Україні : наказ МВС України від 30.12.2014 № 1417 [Електронний ресурс]. – Режим доступу: https://zakon.rada.gov.ua/go/z0252-15 (дата звернення: 13.12.2025).
36. Стручок В. С. Безпека в надзвичайних ситуаціях : методичний посібник для здобувачів освітнього ступеня «магістр». – Тернопіль : ФОП Паляниця В. А., 2022. – 156 с. [Електронний ресурс]. – Режим доступу: https://elartu.tntu.edu.ua/handle/lib/39196.

37. Стручок В. С. Техноекологія та цивільна безпека. Частина «Цивільна безпека» : навчальний посібник. – Тернопіль : ФОП Паляниця В. А., 2022. – 156 с. [Електронний ресурс]. – Режим доступу: http://elartu.tntu.edu.ua/handle/lib/39424.
38. World Health Organization. Guidelines on mental health at work. – Geneva : WHO, 2022 [Електронний ресурс]. – Режим доступу: https://www.who.int/publications/i/item/9789240053052.
39. International Labour Organization. Workplace stress: A collective challenge. – Geneva : ILO, 2016 [Електронний ресурс]. – Режим доступу: https://www.ilo.org/global/topics/safety-and-health-at-work/resources-library/publications/WCMS_466547/lang--en/index.htm.
40. Методичні вказівки до виконання кваліфікаційної роботи магістра для здобувачів спеціальності 121 – Інженерія програмного забезпечення, всіх форм навчання / укладачі Михалик Д.М., Цуприк Г.Б., Бревус В.М., Мудрик І.Я. – Тернопіль Тернопільський національний технічний університет імені Івана Пулюя, 2024. – 44 с.
41. Mudryk I**.** Machine learning models and methods aspects of processing unstructured data [Електронний ресурс]. – 2024. – Режим доступу: https://scholar.google.com/citations?view_op=view_citation&user=YIBK1fgAAAAJ&citation_for_view=YIBK1fgAAAAJ:hFOr9nPyWt4C.
42. Mudryk I. Використання штучного інтелекту для розробки системи відеоспостереження з використанням технологій хмарних вебсервісів aws [Електронний ресурс]. – 2023. – Режим доступу: https://scholar.google.com/citations?view_op=view_citation&user=YIBK1fgAAAAJ&citation_for_view=YIBK1fgAAAAJ:L8Ckcad2t8MC.
43. Mudryk I. Ентерпрайз патерни для кросплатформної розробки [Електронний ресурс]. – 2023. – Режим доступу: https://scholar.google.com/citations?view_op=view_citation&user=YIBK1fgAAAAJ&citation_for_view=YIBK1fgAAAAJ:HDshCWvjkbEC.

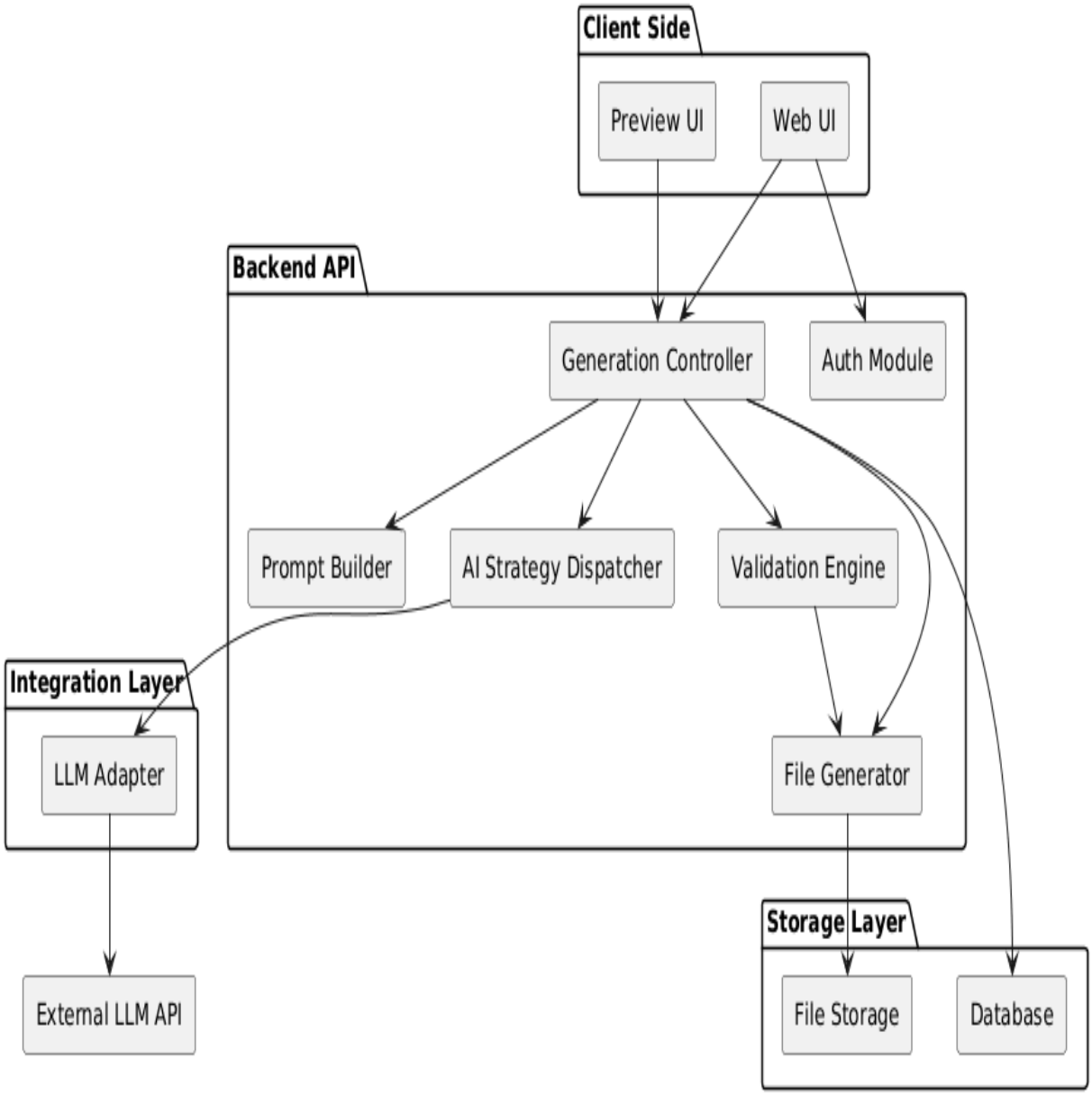
ДОДАТКИ

ДОДАТОК А

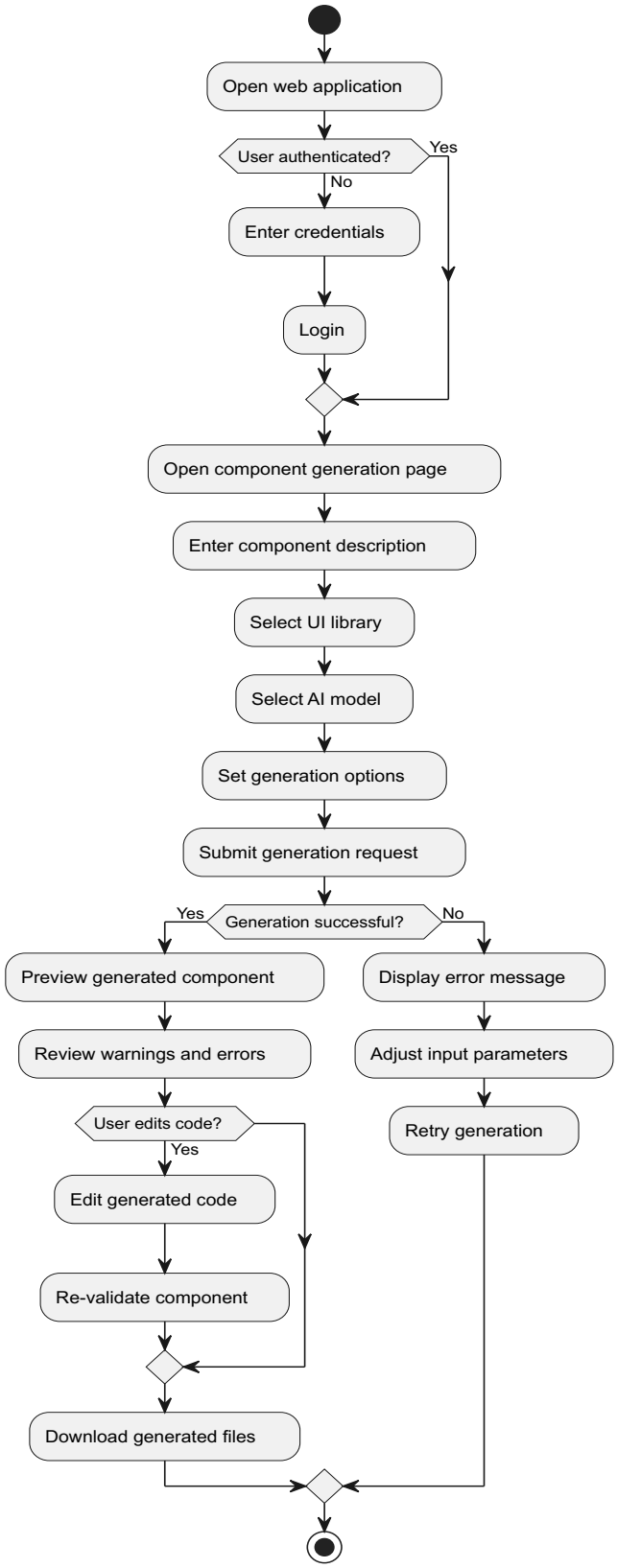
Архітектура системи / Реалізація процесу генерації



Діаграма послідовності процесу генерації UI-компонента



Повна діаграма користувацьких сценаріїв роботи із системою



ДОДАТОК Б

Публікація наукової конференції

УДК 004.41

О. Глух, І. Мудрик
(Тернопільський національний технічний університет імені Івана Пулюя, Україна)

МЕТОДИ ТА ПІДХОДИ ДО АВТОМАТИЧНОЇ ГЕНЕРАЦІЇ ІНТЕРФЕЙСНИХ ЕЛЕМЕНТІВ У ВЕБ-РОЗРОБЦІ НА ОСНОВІ ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ

UDC 004.41

O. Hlukh, I. Mudryk

METHODS AND APPROACHES FOR AUTOMATED GENERATION OF INTERFACE ELEMENTS IN WEB DEVELOPMENT USING LARGE LANGUAGE MODELS

Швидкий розвиток великих мовних моделей (LLM) зумовив появу нових підходів до автоматизації процесів створення інтерфейсів у веброзробці. Зокрема, моделі здатні інтерпретувати текстові описи вимог і генерувати варіанти інтерфейсних рішень, що відкриває можливість часткового або повного усунення ручного написання коду UI-компонентів [1]. Такий підхід дозволяє зменшити трудомісткість початкових етапів проєктування та підтримати розробників у формуванні структури та логіки інтерфейсів.

В основі застосування LLM для генерації елементів інтерфейсу лежать принципи контекстного аналізу та синтезу структурованих відповідей, що є предметом все ширших досліджень у галузі інженерії програмного забезпечення [2]. Актуальними залишаються питання точності генерації, узгодженості створених елементів, можливостей моделі відтворювати стилістичні та функціональні вимоги, а також адаптації до різноманітних технологічних стеків. Додаткову увагу привертають підходи до оцінювання якості створеного коду і визначення умов, за яких моделі можуть ефективно підтримувати процеси побудови інтерфейсів [3].

Перспективи подальших досліджень включають уточнення методів взаємодії між розробниками та мовними моделями, удосконалення інтерпретації вимог, механізми самокорекції згенерованого коду та інтеграцію систем аналізу якості. Використання LLM у формуванні інтерфейсних рішень створює підґрунтя для появи нових інтелектуальних інструментів, здатних підвищити ефективність та доступність веброзробки.

Література

1. Interaction Design Foundation. Generative AI for User Interface Design: Current Capabilities and Emerging Trends, 2024. Електронний ресурс. Режим доступу: https://www.interaction-design.org/literature/topics/generative-ai (дата звернення: 09.12.2025).
2. Wang X., Guo Y., Li J. Large Language Models for Software Engineering: Progress and Open Challenges. IEEE Software, 2024. Електронний ресурс. Режим доступу: https://ieeexplore.ieee.org/document/10512345 (дата звернення: 09.12.2025).
3. Choudhury R., Ben Abacha A., Demner-Fushman D. Evaluating Structured Code and UI Generation with Modern LLMs. arXiv, 2025. Електронний ресурс. Режим доступу: https://arxiv.org/abs/2503.00412 (дата звернення: 09.12.2025).

ДОДАТОК В

Посилання на github https://github.com/OlegHlukh/components-forge

Диск з роботою

ДОДАТОК Г

Лістинг коду 1 – app/page.tsx

```
import { auth } from "@/lib/auth";
import { GeneratorSection } from "@/components/GeneratorSection";
import { HomeContent } from "./HomeContent";
import styles from "./page.module.css";

export default async function Home() {
  const session = await auth();

  return <main className={styles.main}>{session ? <HomeContent /> :
<GeneratorSection />}</main>;
}
```

Лістинг коду 2 – файл Generate.tsx

```
"use client";

import { useState, useEffect, useCallback, useRef } from "react";
import { useRouter, useSearchParams } from "next/navigation";
import { SearchBar, SearchSubmitParams } from
"@/components/SearchBar/SearchBar";
import { AIModel } from "@/types/ai-models";
import { StyleLibrary } from "@/types/style-libraries";
import styles from "./GeneratorSection.module.css";

export function GeneratorSection() {
  const router = useRouter();
  const searchParams = useSearchParams();
  const [isAuthenticated, setIsAuthenticated] = useState<boolean |
null>(null);
  const hasAutoGenerated = useRef(false);

  // Get params from URL (from login redirect)
  const urlPrompt = searchParams.get("prompt") || "";
  const urlModel = (searchParams.get("model") as AIModel) ||
AIModel.GPT4;
  const urlStyleLibrary =
    (searchParams.get("styleLibrary") as StyleLibrary) ||
StyleLibrary.CSS_MODULES;

  const handleGenerate = useCallback(
    async (params: SearchSubmitParams) => {
```

```
    // Check if user is authenticated
      if (isAuthenticated === false) {
        // Save params in URL for after login
        const loginParams = new URLSearchParams({
          returnUrl: "/",
          prompt: params.prompt,
          model: params.model,
          styleLibrary: params.styleLibrary,
        });
        router.push(`/login?${loginParams.toString()}`);
        return;
      }

      // Redirect to generation page
      const queryParams = new URLSearchParams({
        prompt: params.prompt,
        model: params.model,
        styleLibrary: params.styleLibrary,
      });

      router.push(`/components/new?${queryParams.toString()}`);
    },
    [isAuthenticated, router]
  );

  // Check authentication status
  useEffect(() => {
    async function checkAuth() {
      try {
        const response = await fetch("/api/auth/session");
        const data = await response.json();
        setIsAuthenticated(!!data.user);
      } catch {
        setIsAuthenticated(false);
      }
    }
    checkAuth();
  }, []);

  // Auto-generate if authenticated and has URL params (after login
redirect)
  useEffect(() => {
    if (
      isAuthenticated === true &&
      urlPrompt &&
      !hasAutoGenerated.current
    ) {
      hasAutoGenerated.current = true;
      handleGenerate({
```

```
        prompt: urlPrompt,
        model: urlModel,
        styleLibrary: urlStyleLibrary,
      });
    }
  }, [isAuthenticated, urlPrompt, urlModel, urlStyleLibrary,
handleGenerate]);

  return (
    <div className={styles.container}>
      {/* Hero Section */}
      <div className={styles.hero}>
        <div className={styles.glow} />
        <h1 className={styles.title}>
          <span className={styles.gradient}>Create</span> beautiful
components
        </h1>
        <p className={styles.subtitle}>
          Describe your UI component and let AI generate production-
ready React code
        </p>

        <div className={styles.searchWrapper}>
          <SearchBar
            onGenerate={handleGenerate}
            initialPrompt={urlPrompt}
            initialModel={urlModel}
            initialStyleLibrary={urlStyleLibrary}
          />
        </div>
      </div>
    </div>
  );
}
```

Лістинг 3 – файл useEditorState

```
import { useState, useEffect, useCallback } from "react";

interface EditorState {
  isFullscreen: boolean;
  isChatCollapsed: boolean;
  showCode: boolean;
  hasChanges: boolean;
  saveSuccess: boolean;
}
```

```
interface EditorHandlers {
  toggleFullscreen: () => void;
  toggleChat: () => void;
  toggleCode: () => void;
  exitFullscreen: () => void;
  markChanged: () => void;
  markSaved: () => void;
  resetChangeState: () => void;
}

export function useEditorState(): [EditorState, EditorHandlers] {
  const [isFullscreen, setIsFullscreen] = useState(false);
  const [isChatCollapsed, setIsChatCollapsed] = useState(false);
  const [showCode, setShowCode] = useState(false);
  const [hasChanges, setHasChanges] = useState(false);
  const [saveSuccess, setSaveSuccess] = useState(false);

  // Handle Escape key to exit fullscreen
  useEffect(() => {
    const handleEscape = (e: KeyboardEvent) => {
      if (e.key === "Escape" && isFullscreen) {
        setIsFullscreen(false);
      }
    };

    document.addEventListener("keydown", handleEscape);
    return () => document.removeEventListener("keydown", handleEscape);
  }, [isFullscreen]);

  const toggleFullscreen = useCallback(() => {
    setIsFullscreen((prev) => !prev);
  }, []);

  const exitFullscreen = useCallback(() => {
    setIsFullscreen(false);
  }, []);

  const toggleChat = useCallback(() => {
    setIsChatCollapsed((prev) => !prev);
  }, []);

  const toggleCode = useCallback(() => {
    setShowCode((prev) => !prev);
  }, []);

  const markChanged = useCallback(() => {
    setHasChanges(true);
    setSaveSuccess(false);
  }, []);
```

```
  const markSaved = useCallback(() => {
    setHasChanges(false);
    setSaveSuccess(true);
    setTimeout(() => setSaveSuccess(false), 2000);
  }, []);

  const resetChangeState = useCallback(() => {
    setHasChanges(false);
    setSaveSuccess(false);
  }, []);

  const state: EditorState = {
    isFullscreen,
    isChatCollapsed,
    showCode,
    hasChanges,
    saveSuccess,
  };

  const handlers: EditorHandlers = {
    toggleFullscreen,
    toggleChat,
    toggleCode,
    exitFullscreen,
    markChanged,
    markSaved,
    resetChangeState,
  };

  return [state, handlers];
}
```

Лістинг 4 – файл Component.tsx

```
"use client";

import { useRef, useCallback } from "react";
import { useParams, useRouter } from "next/navigation";
import { Loader2, PanelRightOpen } from "lucide-react";
import { ComponentPreview, ComponentPreviewRef } from
"@/components/ComponentPreview";
import { ComponentToolbar } from "@/components/ComponentToolbar";
import { EditChat } from "@/components/EditChat/EditChat";
import { useComponent } from "@/hooks/useComponent";
import { useEditorState } from "@/hooks/useEditorState";
import { PreviewFile } from "@/types/components";
```

```
import styles from "./page.module.css";

const HEADER_HEIGHT = 48;
const PAGE_PADDING = 140;

export default function ComponentPage() {
  const params = useParams();
  const router = useRouter();
  const previewRef = useRef<ComponentPreviewRef>(null);

  const [componentState, componentHandlers] = useComponent(params.id as
string);
  const [editorState, editorHandlers] = useEditorState();

  const { component, isLoading, error, isSaving, isDeleting,
isPublishing } = componentState;
  const { updateFiles, saveFiles, deleteComponent, togglePublish } =
componentHandlers;

  const { isFullscreen, isChatCollapsed, showCode, hasChanges,
saveSuccess } = editorState;
  const { toggleFullscreen, toggleChat, toggleCode, markChanged,
markSaved, resetChangeState } =
    editorHandlers;

  const handleSave = async () => {
    const files = previewRef.current?.getFiles();
    if (!files) return;

    const success = await saveFiles(files);
    if (success) {
      markSaved();
    }
  };

  const handleDelete = async () => {
    if (!component) return;

    const success = await deleteComponent();
    if (success) {
      router.push("/");
    } else {
      alert("Failed to delete component");
    }
  };

  const handleComponentUpdate = (updatedFiles: PreviewFile[]) => {
    updateFiles(updatedFiles);
    resetChangeState();
```

```
  };

  const getEditorFiles = useCallback(() => {
    return previewRef.current?.getFiles() ?? component?.files ?? [];
  }, [component?.files]);

  const previewHeight = isFullscreen
    ? window.innerHeight - HEADER_HEIGHT
    : window.innerHeight - PAGE_PADDING;

  if (isLoading) {
    return (
      <div className={styles.loading}>
        <Loader2 size={32} className={styles.spinner} />
        <span>Loading component...</span>
      </div>
    );
  }

  if (error || !component) {
    return (
      <div className={styles.error}>
        <h2>Error</h2>
        <p>{error || "Component not found"}</p>
        <button onClick={() => router.push("/")}
className={styles.backButton}>
          Go Back
        </button>
      </div>
    );
  }

  return (
    <div className={styles.page}>
      <div
        className={`${styles.content} ${isFullscreen ?
styles.fullscreen : ""} ${isChatCollapsed ? styles.chatCollapsed :
""}`}
      >
        <div className={styles.previewSection}>
          <ComponentToolbar
            onBack={() => router.push("/")}
            showCode={showCode}
            onToggleCode={toggleCode}
            onSave={handleSave}
            isSaving={isSaving}
            hasChanges={hasChanges}
            saveSuccess={saveSuccess}
            downloadUrl={`/api/components/${component.id}/download`}
```

```
            isPublic={component.isPublic}
            onTogglePublish={togglePublish}
            isPublishing={isPublishing}
            onDelete={handleDelete}
            isDeleting={isDeleting}
            isChatVisible={!isChatCollapsed}
            onHideChat={toggleChat}
            isFullscreen={isFullscreen}
            onToggleFullscreen={toggleFullscreen}
          />
          <div className={styles.previewContainer}>
            <ComponentPreview
              ref={previewRef}
              files={component.files}
              componentName={component.name}
              styleLibrary={component.styleLib}
              showEditor={showCode}
              height={previewHeight}
              onFilesChange={markChanged}
            />
          </div>
        </div>

        {!isChatCollapsed && (
          <div className={styles.chatSection}>
            <EditChat
              componentId={component.id}
              currentFiles={component.files}
              styleLibrary={component.styleLib}
              onUpdate={handleComponentUpdate}
              getEditorFiles={getEditorFiles}
            />
          </div>
        )}

        {isChatCollapsed && (
          <button onClick={toggleChat}
className={styles.expandChatButton} title="Show chat">
            <PanelRightOpen size={20} />
            <span>Show AI Chat</span>
          </button>
        )}
      </div>
    </div>
  );
}
```

Лістинг 5 – файл ai-factory.ts

```
import { AIModel } from "@/types/ai-models";
import { AIStrategy } from "./base.strategy";
import { OpenAIStrategy } from "./openai.strategy";
import { AnthropicStrategy } from "./anthropic.strategy";
import { GoogleStrategy } from "./google.strategy";
import { GroqStrategy } from "./groq.strategy";

/**
 * Фабрика для створення відповідної AI стратегії на основі моделі
 */
export class AIStrategyFactory {
  /**
   * Створює стратегію для конкретної моделі
   */
  static createStrategy(model: AIModel): AIStrategy {
    switch (model) {
      // OpenAI моделі
      case AIModel.GPT4:
        return new OpenAIStrategy("gpt-4");
      case AIModel.GPT4_TURBO:
        return new OpenAIStrategy("gpt-4-turbo");
      case AIModel.GPT35_TURBO:
        return new OpenAIStrategy("gpt-3.5-turbo");

      // Anthropic моделі
      case AIModel.CLAUDE_3_OPUS:
        return new AnthropicStrategy("claude-3-opus-20240229");
      case AIModel.CLAUDE_3_SONNET:
        return new AnthropicStrategy("claude-3-sonnet-20240229");

      // Google моделі
      case AIModel.GEMINI_PRO:
        return new GoogleStrategy("gemini-pro");

      // Groq моделі (безкоштовний tier)
      case AIModel.LLAMA_3_3_70B:
        return new GroqStrategy("llama-3.3-70b-versatile");
      case AIModel.LLAMA_3_1_8B:
        return new GroqStrategy("llama-3.1-8b-instant");
      case AIModel.MIXTRAL_8X7B:
        return new GroqStrategy("mixtral-8x7b-32768");

      default:
        // За замовчуванням використовуємо GPT-4
        return new OpenAIStrategy("gpt-4");
    }
  }
```

```
  /**
   * Перевіряє чи підтримується модель
   */
  static isModelSupported(model: AIModel): boolean {
    try {
      this.createStrategy(model);
      return true;
    } catch {
      return false;
    }
  }
}
```

Лісниг 5 – component-generation.facade.ts

```
import { AIModel } from "@/types/ai-models";
import { StyleLibrary } from "@/types/style-libraries";
import { logger } from "@/lib/logger";
import { validatePrompt, buildComponentPrompt } from
"./prompt.service";
import { generateComponent as generateAI } from "./ai.service";
import { createComponent, ComponentWithFiles } from
"./component.service";

const log = logger.child("GenerationFacade");

export interface GenerationRequest {
  prompt: string;
  model: AIModel;
  styleLibrary: StyleLibrary;
  userId: number;
}

export interface GenerationResponse {
  success: true;
  component: {
    id: string;
    name: string;
    files: ComponentWithFiles["files"];
  };
}

export interface GenerationError {
  success: false;
  error: string;
  type: "VALIDATION_ERROR" | "AI_ERROR" | "STORAGE_ERROR" |
"UNKNOWN_ERROR";
}
```

```
export type GenerationResult = GenerationResponse | GenerationError;

//
================================================================================
// FACADE
//
================================================================================

/**
 * Фасад для генерації компонентів
 *
 * Інкапсулює складну логіку взаємодії між сервісами:
 * - PromptService (валідація та побудова промптів)
 * - AIService (генерація через AI моделі)
 * - ComponentService (збереження в БД та на диск)
 *
 * Забезпечує єдиний простий інтерфейс для клієнтського коду
 */
export class ComponentGenerationFacade {
  /**
   * Генерує новий компонент на основі промпта користувача
   *
   * Етапи:
   * 1. Валідація промпта (перевірка на UI-компонент)
   * 2. Побудова системного промпта з інструкціями для AI
   * 3. Виклик AI моделі для генерації коду
   * 4. Парсинг відповіді та збереження компонента
   */
  async generate(request: GenerationRequest): Promise<GenerationResult>
{
    const { prompt, model, styleLibrary, userId } = request;

    // Етап 1: Валідація промпта
    const validation = this.validateRequest(prompt);
    if (!validation.isValid) {
      return {
        success: false,
        error: validation.error!,
        type: "VALIDATION_ERROR",
      };
    }

    // Етап 2: Побудова системного промпта
    const systemPrompt = this.buildPrompt(prompt, model, styleLibrary);

    // Етап 3: Генерація через AI
    const aiResult = await this.callAI(systemPrompt, model);
```

```
    if (!aiResult.success) {
      return {
        success: false,
        error: aiResult.error!,
        type: "AI_ERROR",
      };
    }

    // Етап 4: Збереження компонента
    const saveResult = await this.saveComponent({
      aiResponse: aiResult.code!,
      prompt,
      model,
      styleLibrary,
      userId,
    });

    if (!saveResult.success) {
      return {
        success: false,
        error: saveResult.error!,
        type: "STORAGE_ERROR",
      };
    }

    return {
      success: true,
      component: saveResult.component!,
    };
  }

  /**
   * Валідує промпт користувача
   */
  private validateRequest(prompt: string): { isValid: boolean; error?:
string } {
    return validatePrompt(prompt);
  }

  /**
   * Будує системний промпт для AI
   */
  private buildPrompt(userPrompt: string, model: AIModel, styleLibrary:
StyleLibrary): string {
    return buildComponentPrompt({
      userPrompt,
      model,
      styleLibrary,
    });
```

```
  }

  /**
   * Викликає AI модель для генерації
   */
  private async callAI(
    systemPrompt: string,
    model: AIModel
  ): Promise<{ success: boolean; code?: string; error?: string }> {
    const result = await generateAI({
      systemPrompt,
      model,
    });

    return {
      success: result.success,
      code: result.code,
      error: result.error,
    };
  }

  /**
   * Зберігає згенерований компонент
   */
  private async saveComponent(input: {
    aiResponse: string;
    prompt: string;
    model: AIModel;
    styleLibrary: StyleLibrary;
    userId: number;
  }): Promise<{
    success: boolean;
    component?: { id: string; name: string; files:
ComponentWithFiles["files"] };
    error?: string;
  }> {
    try {
      const component = await createComponent(input);

      return {
        success: true,
        component: {
          id: component.id,
          name: component.name,
          files: component.files,
        },
      };
    } catch (error) {
      log.error("Failed to save component", error);
```

```
      return {
        success: false,
        error: error instanceof Error ? error.message : "Failed to save
component",
      };
    }
  }
}

//
================================================================================
=====
// SINGLETON INSTANCE
//
================================================================================
=====

/**
 * Singleton інстанс фасаду для використання в API routes
 */
export const componentGenerationFacade = new
ComponentGenerationFacade();
```

Лістиг 6 – file-parse.service.ts

```
import { StyleLibrary } from "@/types/style-libraries";
import { logger } from "@/lib/logger";

const log = logger.child("FileParser");

export interface ParsedFile {
  filename: string;
  content: string;
  fileType: "tsx" | "css" | "types" | "other";
}

export interface ParseResult {
  componentName: string;
  files: ParsedFile[];
}

/**
 * Парсить відповідь AI та витягує файли компонента
 * @param existingFiles - якщо передано, намагається зберегти
оригінальні назви файлів
 */
```

```
export function parseAIResponse(
  response: string,
  styleLibrary: StyleLibrary,
  existingFiles?: Array<{ filename: string; fileType: string }>
): ParseResult {
  const files: ParsedFile[] = [];
  let componentName = "Component";

  // Якщо є існуючі файли, витягуємо назву компонента з них
  if (existingFiles && existingFiles.length > 0) {
    const mainFile = existingFiles.find(
      (f) => f.fileType === "tsx" &&
!f.filename.toLowerCase().includes("app")
    );
    if (mainFile) {
      componentName = mainFile.filename.replace(/\.(tsx|ts|jsx|js)$/,
"");
    }
  }

  log.debug("Parsing AI response", { responseLength: response.length,
componentName });

  // Спочатку шукаємо code blocks з назвою файлу у форматі
```filename.tsx
  const codeBlocksWithFilename = /```([a-zA-Z0-9_.-
]+\.(?:tsx?|jsx?|css|scss|module\.css))\n([\s\S]*?)```/g;

  let match;
  while ((match = codeBlocksWithFilename.exec(response)) !== null) {
    const [, filename, content] = match;
    const trimmedContent = content.trim();

    if (filename && trimmedContent) {
      const fileType = getFileTypeFromFilename(filename);
      files.push({ filename, content: trimmedContent, fileType });
    }
  }

  // Якщо знайшли файли з назвами, витягуємо componentName і повертаємо
  if (files.length > 0) {
    // Знаходимо головний tsx файл (не App.tsx)
    const mainFile = files.find(
      (f) => f.fileType === "tsx" &&
!f.filename.toLowerCase().includes("app")
    );

    if (mainFile) {
      // Назва з файлу (ProfileCard.tsx -> ProfileCard)
```
```

```
      const filenameBasedName =
mainFile.filename.replace(/\.(tsx|ts|jsx|js)$/, "");

      // Пробуємо витягнути з контенту
      const extracted = extractComponentName(mainFile.content);

      // Якщо з контенту отримали generic "Component", використовуємо
назву файлу
      if (extracted && extracted !== "Component") {
        componentName = extracted;
      } else {
        componentName = filenameBasedName;
      }
    }

    log.debug("Parsed files with filenames", { files: files.map((f) =>
f.filename), componentName });
    return { componentName, files };
  }

  // Шукаємо code blocks з коментарем назви файлу на першому рядку
  // Формат: ```tsx\n// Button.tsx або ```tsx\n/* Button.tsx */
  const codeBlocksWithComment =

/```(\w+)\n(?:\/\/\s*([^\n]+\.(?:tsx?|jsx?|css|scss))|\/\*\s*([^\n]+\.(
?:tsx?|jsx?|css|scss))\s*\*\/)\n([\s\S]*?)```/g;

  while ((match = codeBlocksWithComment.exec(response)) !== null) {
    const [, language, filename1, filename2, content] = match;
    const filename = (filename1 || filename2 || "").trim();
    const trimmedContent = content.trim();

    if (filename && trimmedContent) {
      const fileType = getFileType(language, filename);

      // Витягуємо назву компонента
      if (fileType === "tsx" &&
!filename.toLowerCase().includes("app")) {
        const extracted = extractComponentName(trimmedContent);
        if (extracted) componentName = extracted;
      }

      files.push({ filename, content: trimmedContent, fileType });
    }
  }

  // Якщо не знайшли з коментарями, шукаємо звичайні code blocks
  if (files.length === 0) {
    const simpleCodeBlocks = /```(\w+)?\n([\s\S]*?)```/g;
```

```
    while ((match = simpleCodeBlocks.exec(response)) !== null) {
      const [, language = "tsx", content] = match;
      const trimmedContent = content.trim();

      if (!trimmedContent) continue;

      // Перевіряємо чи контент схожий на React компонент
      const isReactCode =
        trimmedContent.includes("import") ||
        trimmedContent.includes("export") ||
        trimmedContent.includes("function") ||
        trimmedContent.includes("const");

      const isCss =
        ["css", "scss", "sass"].includes(language) ||
        (trimmedContent.includes("{") &&
          trimmedContent.includes("}") &&
          !trimmedContent.includes("import") &&
          !trimmedContent.includes("export"));

      let fileType: ParsedFile["fileType"] = "other";
      let filename = "";

      if (
        ["tsx", "jsx", "ts", "js", "typescript",
"javascript"].includes(language) ||
        isReactCode
      ) {
        fileType = "tsx";
        const extracted = extractComponentName(trimmedContent);
        if (extracted && extracted !== "App") {
          componentName = extracted;
        }

        // Визначаємо чи це App.tsx
        if (
          trimmedContent.includes("export default function App") ||
          trimmedContent.includes("export default App")
        ) {
          filename = "App.tsx";
        } else {
          filename = `${componentName}.tsx`;
        }
      } else if (isCss) {
        fileType = "css";
        filename = getStyleFilename(styleLibrary, componentName);
      } else {
        filename = `${componentName}.${language || "txt"}`;
```

```
      }

      // Уникаємо дублікатів файлів
      const existingFile = files.find((f) => f.filename === filename);
      if (!existingFile) {
        files.push({ filename, content: trimmedContent, fileType });
      }
    }
  }

  // Якщо все ще немає файлів, пробуємо розпарсити як plain text з
React кодом
  if (files.length === 0 && response.includes("export")) {
    log.debug("Trying to parse as plain text");
    const extracted = extractComponentName(response);
    if (extracted) componentName = extracted;

    files.push({
      filename: `${componentName}.tsx`,
      content: response.trim(),
      fileType: "tsx",
    });
  }

  log.debug("Parsed files", { files: files.map((f) => f.filename) });

  return { componentName, files };
}

/**
 * Витягує назву компонента з коду
 */
function extractComponentName(content: string): string | null {
  // Шукаємо export function/const ComponentName
  const patterns = [
    /export\s+(?:default\s+)?function\s+(\w+)/,
    /export\s+(?:default\s+)?const\s+(\w+)/,
    /function\s+(\w+)\s*\(/,
    /const\s+(\w+)\s*[:=]\s*(?:\([^)]*\)|[^=])*=>\s*[{(]/,
  ];

  for (const pattern of patterns) {
    const match = content.match(pattern);
    if (match && match[1] && !["styles", "App"].includes(match[1])) {
      return match[1];
    }
  }

  return null;
```

```
}

/**
 * Визначає тип файлу з назви файлу
 */
function getFileTypeFromFilename(filename: string):
ParsedFile["fileType"] {
  if (/\.(tsx?|jsx?)$/.test(filename)) {
    return "tsx";
  }
  if (/\.(css|scss|sass)$/.test(filename)) {
    return "css";
  }
  return "other";
}

/**
 * Визначає тип файлу
 */
function getFileType(language: string, filename: string):
ParsedFile["fileType"] {
  if (["tsx", "jsx", "ts", "js", "typescript",
"javascript"].includes(language)) {
    return "tsx";
  }
  if (["css", "scss", "sass"].includes(language)) {
    return "css";
  }
  if (
    filename.endsWith(".tsx") ||
    filename.endsWith(".ts") ||
    filename.endsWith(".jsx") ||
    filename.endsWith(".js")
  ) {
    return "tsx";
  }
  if (filename.endsWith(".css") || filename.endsWith(".scss")) {
    return "css";
  }
  return "other";
}

/**
 * Отримує назву файлу стилів відповідно до бібліотеки
 */
function getStyleFilename(styleLibrary: StyleLibrary, componentName:
string): string {
  switch (styleLibrary) {
    case StyleLibrary.CSS_MODULES:
```

```
      return `${componentName}.module.css`;
    case StyleLibrary.STYLED_COMPONENTS:
      return `${componentName}.styles.ts`;
    case StyleLibrary.VANILLA_CSS:
      return `${componentName}.css`;
    default:
      return `${componentName}.module.css`;
  }
}

/**
 * Генерує компонентну назву з опису користувача
 */
export function generateComponentName(description: string): string {
  // Беремо перші 2-3 значущі слова
  const words = description
    .toLowerCase()
    .replace(/[^a-zA-Za-яА-Я0-9\s]/g, "")
    .split(/\s+/)
    .filter(
      (word) =>
        word.length > 2 &&
        !["the", "and", "for", "with", "that", "this", "create",
"make", "build"].includes(word)
    )
    .slice(0, 3);

  if (words.length === 0) {
    return "Component";
  }

  // Конвертуємо в PascalCase
  return words.map((word) => word.charAt(0).toUpperCase() +
word.slice(1)).join("");
}
```