Національний лісотехнічний університет України
(повна назва закладу вищої освіти)

Навчально-науковий інститут комп'ютерних наук та інформаційних технологій
(повна назва інституту, назва факультету (відділення))

Кафедра комп'ютерних наук
(повна назва кафедри (предметної, циклової комісії))

Пояснювальна записка

до дипломної роботи

перший (бакалаврський)
(рівень вищої освіти)

на тему: *Розроблення системи генерації прототипів користувацького інтерфейсу за допомогою великих мовних моделей*

Виконала: студентка 4 курсу групи КН–41
спеціальності
122 "Комп'ютерні науки"
(шифр і назва напряму підготовки, спеціальності)

Ковтуненко Л. С.
(прізвище та ініціали)

Керівник *Процик Ю. С.*
(прізвище та ініціали)

Рецензент *Стороженук О. Л.*
(прізвище та ініціали)

Львів – 2025

Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)

ННІ комп'ютерних наук та інформаційних технологій
Кафедра комп'ютерних наук
Рівень вищої освіти перший (бакалаврський)
Спеціальність 122 "Комп'ютерні науки"
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри КН
Борецька І. Б.
"10" червня 2025 року

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Ковтуненко Лідії Сергіївній
(прізвище, ім'я, по батькові)

1. Тема роботи *Розроблення системи генерації прототипів користувацького інтерфейсу за допомогою великих мовних моделей*
керівник роботи *Процик Юрій Степанович, к.ф.-м.н.*
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджена наказом вищого навчального закладу від "*15*" *листопада 2024 року* № *С-882*
2. Термін подання студентом роботи *10 червня 2025 р.*
3. Вихідні дані до роботи *Аналіз шляхів вирішення задачі, організаційна структура системи*

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
Вступ. Розділ 1. Стан проблемної області.
Розділ 2. Інформаційне та математичне забезпечення.
Розділ 3. Програмне та технічне забезпечення.
Висновки. Список використаних джерел. Додатки.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
слайди для доповіді (підготовка матеріалу для доповіді загальним обсягом 10-15 слайдів)

6. Дата видачі завдання *18 листопада 2024 р.*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1.	Огляд літературних джерел	*18.11.2024 р.* *23.12.2025 р.*	Виконано
2.	Аналіз проблемної області та постановка задачі	*16.02.2025 р.* *22.02.2025 р.*	Виконано
3.	Проектування інформаційного та математичного забезпечення	*22.02.2025 р.* *16.03.2025 р.*	Виконано
4.	Програмна реалізація проекту	*17.03.2025 р.* *27.05.2025 р.*	Виконано
5.	Тестування та усунення помилок програмної реалізації	*11.05.2025 р.* *03.06.2025 р.*	Виконано
6.	Оформлення пояснювальної записки та здача на рецензування	*03.06.2025 р.* *10.06.2025 р.*	Виконано

Студент _____ Ковтуненко Л. С.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Процик Ю. С.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дипломна робота містить 54 сторінок пояснювальної записки, 26 рисунків, 7 таблиць, 1 додаток та 17 джерел.

Розроблено вебплатформу UI Crafter для генерації прототипів користувацьких інтерфейсів із використанням великих мовних моделей GPT-4 та локальної донавченої LoRA-моделі на основі TinyLlama. Система формує валідний HTML+CSS код на основі текстових описів інтерфейсів. Платформа реалізована як Flask-застосунок з інтеграцією OpenAI API та локальної моделі через HuggingFace Transformers і PEFT. Передбачено постобробку HTML-коду для очищення службових вставок та стабілізації структури. Рішення призначене для швидкого прототипування UX/UI-дизайну на ранніх етапах розробки. Перспективною є інтеграція системи як плагіну для Figma або інших дизайнерських середовищ.

Ключові слова: генерація HTML, великі мовні моделі, прототип інтерфейсу, GPT-4, LoRA, TinyLlama, Flask-застосунок, low-fidelity дизайн.

ABSTRACT

The thesis consists of 54 pages of explanatory text, 26 figures, 7 tables, 1 appendix, and 17 references.

The thesis presents the development of the **UI Crafter** web platform for automated UI prototyping using large language models, including GPT-4 and a locally fine-tuned TinyLlama-1.1B model with LoRA adaptation. The system generates valid HTML+CSS from user-provided interface descriptions. It is implemented as a Flask web application integrated with OpenAI API and local models via HuggingFace Transformers and PEFT. Post-processing removes service markers and stabilizes HTML structure. The solution is intended for rapid UX/UI prototyping at early design stages. The system may further be integrated as a plugin for Figma or other design environments.

Keywords: HTML generation, large language models, user interface prototyping, GPT-4, LoRA, TinyLlama, Flask application, low-fidelity design.

ТЕХНІЧНЕ ЗАВДАННЯ

Розробити систему генерації прототипів користувацького інтерфейсу за допомогою великих мовних моделей, що передбачає створення вебплатформи для автоматизованого перетворення текстових описів (prompts) у низькорівневі HTML+CSS-прототипи інтерфейсів із використанням моделей GPT-4 (через API) та кастомізованої LoRA-моделі на базі TinyLlama.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

API (Application Programming Interface) – інтерфейс прикладного програмування; набір засобів для взаємодії між різними програмними компонентами.
HTML (HyperText Markup Language) – мова розмітки гіпертексту, яка використовується для створення структури вебсторінок.
CSS (Cascading Style Sheets) – каскадні таблиці стилів; мова опису зовнішнього вигляду HTML-елементів.
UI (User Interface) – користувацький інтерфейс; засіб взаємодії користувача з системою.
UX (User Experience) – досвід користувача; враження від використання цифрового продукту.
LoRA (Low-Rank Adaptation) – метод донавчання великих мовних моделей через вставки низькорангових матриць.
LLM (Large Language Model) – велика мовна модель; тип моделі штучного інтелекту, натренованої на великих корпусах тексту.
REST (Representational State Transfer) – стиль архітектури вебсервісів, що використовує стандартні HTTP-методи для взаємодії з ресурсами.
JSON (JavaScript Object Notation) – текстовий формат зберігання структурованих даних, що легко читається людиною і машиною.
GPT-4 (Generative Pre-trained Transformer 4) – четверте покоління генеративної трансформерної моделі від OpenAI.
JS (JavaScript) – мова програмування для створення інтерактивності у вебсторінках.
Flask – мікрофреймворк для побудови вебдодатків на Python.
Jinja2 – шаблонізатор для Python, який використовується для генерації HTML-сторінок на основі шаблонів.

ВСТУП

Актуальність

У сучасному циклі розробки цифрових продуктів дизайн користувацького інтерфейсу (UI) виступає однією з ключових фаз, що безпосередньо впливає на зручність, ефективність та конкурентоспроможність програмного забезпечення. Прототипування, особливо на ранніх стадіях, дозволяє командам розробників і дизайнерів швидко оцінити структуру й логіку взаємодії до початку повноцінної реалізації. Проте навіть створення базових lo-fi прототипів потребує значних затрат часу та професійної експертизи у візуальному дизайні. В умовах зростаючої складності застосунків та коротких дедлайнів, актуальним стає використання інтелектуальних систем генерації інтерфейсів на основі природної мови.

Завдяки стрімкому розвитку великих мовних моделей (LLM), таких як GPT-4 [10, 13], а також методів їх донавчання (наприклад, LoRA [2]), з'явилась можливість автоматизувати генерацію HTML-коду на основі коротких текстових описів. Це відкриває нові перспективи для дизайнерських інструментів майбутнього, де первинні UI-ідеї можуть бути втілені у код за лічені секунди. Саме тому створення системи, що поєднує сучасні мовні моделі з інтерфейсом генерації вебпрототипів, є надзвичайно актуальним завданням на перетині штучного інтелекту та розробки інтерфейсів.

Об'єкт дослідження

Процес автоматизованого проєктування користувацьких інтерфейсів на основі текстових запитів**.**

Предмет дослідження

Методи та інструменти генерації HTML/CSS-прототипів за допомогою великих мовних моделей, а також архітектура системи, яка реалізує цей процес.

Мета роботи

Розробити вебсистему, що приймає текстовий опис інтерфейсу та генерує відповідний HTML+CSS код з використанням великих мовних моделей, з можливістю вибору між зовнішнім API (GPT-4) та локальною кастомізованою LoRA-моделлю.

Завдання

- Провести аналіз сучасних підходів до генерації UI-прототипів за допомогою LLM.
- Побудувати архітектуру вебзастосунку з підтримкою двох джерел
- генерації: GPT-4 (через API) та LoRA-модель на базі TinyLlama.
- Реалізувати інтерактивний web-інтерфейс з формою для вводу prompt'у та блоком рендерингу згенерованого HTML.
- Розробити модуль очищення та нормалізації HTML-коду.
- Провести порівняльне тестування якості генерації для обох моделей.
- Підготувати технічну та експлуатаційну документацію.

Практичне значення

Розроблена система дозволяє значно скоротити час на створення первинних (lo-fi) прототипів вебінтерфейсів, усуваючи потребу у ручному кодуванні на початковій стадії проєктування. Такий підхід може бути інтегрований у дизайнерські середовища (наприклад, як плагін для Figma), а також використаний як основа для комерційних low-code/no-code платформ. У перспективі система здатна підвищити продуктивність UI/UX-команд, автоматизувавши рутинні етапи генерації структури інтерфейсу.

РОЗДІЛ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ

1.1. Проблематика автоматизованого UI-прототипування

Проєктування користувацьких інтерфейсів є однією з найважливіших складових циклу створення веб- або мобільних застосунків. На ранніх етапах розробки зазвичай створюють lo-fi прототипи, які допомагають визначити загальну структуру, взаємодію користувача з інтерфейсом та пріоритети вмісту. Проте навіть така «спрощена» версія вимагає ручної роботи дизайнера або front-end розробника, що ускладнює швидке ітеративне тестування ідей.

Сучасні інструменти, як Figma, Adobe XD, Penpot тощо, спрощують візуалізацію UI, але залишають на користувача відповідальність за наповнення структури, узгодження стилів і забезпечення доступності. Це створює потребу в системах, які могли б автоматично згенерувати кодовий прототип із мінімального текстового опису — тобто здійснювати трансформацію prompt → HTML/CSS [4, 5, 8].

1.2. Потенціал великих мовних моделей (LLM)

Великі мовні моделі (LLM) — це потужні системи, здатні аналізувати та створювати текст, зокрема й у форматі програмного коду. Завдяки навчанню на великих обсягах даних, вони можуть генерувати HTML і CSS, відтворювати базові структури вебінтерфейсів, компонування блоків та візуальні стилі[1, 8, 14]. Це робить їх привабливими для автоматизованого створення прототипів користувацьких інтерфейсів.

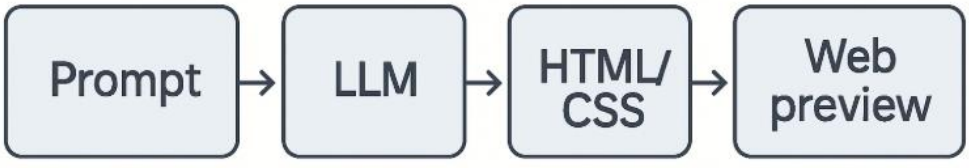


Рисунок 1.1 - Архітектура генерації інтерфейсу за допомогою LLM

Проте на практиці застосування LLM для генерації UI має низку обмежень, які впливають на якість результатів:

- Зайвий текст у відповіді : Моделі часто вставляють пояснення, приклади або додаткові коментарі, які не потрібні в кінцевому HTML-коді. Це ускладнює автоматичне використання результату.
- Помилки в структурі коду : В деяких відповідях відсутні закриваючі теги, неправильна вкладеність елементів або дублювання структур. Такий код може не працювати або виглядати некоректно в браузері.
- Невизначеність у стилях : Якщо явно не вказати, якою системою стилів користуватися (наприклад, TailwindCSS), модель може змішати інлайн-стилі, класичні CSS або навіть стилі з інших бібліотек.
- Відсутність узгодженості : Один і той самий запит може дати дуже різні результати: іноді з повною структурою інтерфейсу, іноді — лише з окремими частинами.
- Потреба в очищенні результату : Згенерований HTML зазвичай потребує додаткової обробки: видалення зайвого тексту, упорядкування класів, приведення коду до чистого та зручного формату[11].

Ці труднощі характерні як для платних API моделей, так і для відкритих рішень. Без спеціальної адаптації або донавчання великі мовні моделі не здатні стабільно генерувати якісний HTML-код для інтерфейсів. Саме тому для побудови надійної системи генерації UI потрібні додаткові рішення: правильна структура запиту (prompt), фільтрація результатів та вибір оптимальної моделі.

1.3. Обмеження open-source моделей у задачах генерації інтерфейсів

Хоча велика частина досліджень зосереджена на комерційних LLM, інтерес до відкритих рішень зростає завдяки їх доступності, локальності та можливості модифікацій. У ході роботи було протестовано кілька сучасних open-source моделей: **Zephyr-7B**, **Mistral-7B (base)** та **DeepSeek-Coder** , які позиціонуються як універсальні генератори тексту або коду [4, 5] .

Zephyr-7B

Ця модель створена на базі Mistral, є однією з популярних моделей для чат-інтерфейсів. Вона здатна генерувати HTML, однак її відповіді часто включають супровідний текст, пояснення, вставки в markdown-форматі, а також узагальнені фрази на кшталт «Here is your code». Подібна поведінка значно ускладнює автоматичне використання результатів у системах, які очікують лише чистий HTML-код без додаткового контексту. Навіть за наявності чітких інструкцій модель іноді ігнорує вимоги щодо форматування або повертає надлишкову структуру, що порушує семантику інтерфейсу.

Окремо варто відзначити й технічні обмеження, з якими я стикнулася при роботі з данною моделлю. Наприклад, при тестуванні моделі Zephyr-7B Beta в умовах типової задачі (створення простого HTML+CSS прототипу to-do списку без зображень і JavaScript) модель не змогла повернути відповідь, і платформа повернула повідомлення fetch failed, а це свідчить про нестабільність роботи або невідповідність запиту внутрішнім обмеженням моделі, й вказує на її обмежену придатність у задачах генерації коду для UI. Ситуація наведена на рисунку 1.2.

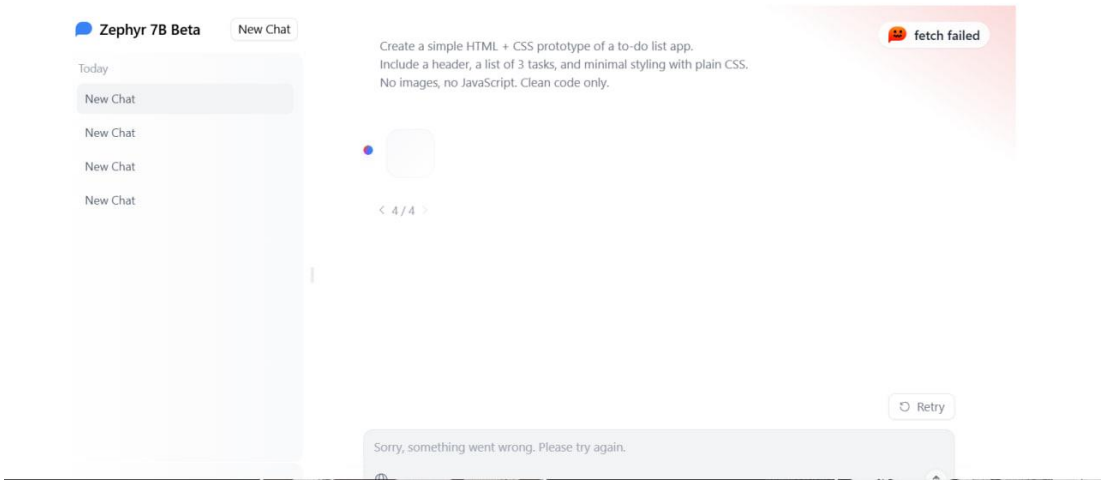


Рисунок 1.2 - Випадок невдалої генерації HTML-коду моделлю Zephyr-7B

Mistral-7B

Ця модель дійсно демонструє здатність генерувати валідний HTML, однак її відповіді залежать від точності й формулювання запиту. Якщо prompt подано у вільній формі, без інструкції крок за кроком, результати часто є неповними: відсутні

закриваючі теги, змішуються блоки, структура виглядає фрагментованою. До того ж, стиль оформлення є нестабільним — зустрічається як інлайн-стилізація, так і випадкові CSS-класи без узгодженості [4]. Це знижує якість генерації і потребує постійної перевірки вручну або доопрацювання шаблонів.

У тестовому прикладі генерації простого to-do списку модель все ж змогла створити повноцінний HTML-документ з мінімальним CSS-оформленням. Результат виглядав охайно: заголовок, три задачі з чекбоксами, блоки з паддингами і тінями — усе було візуально узгодженим. Проте стилі були прописані вручну у блоці <style>, без використання CSS-фреймворків, а сама структура — надто загальна для гнучкого повторного використання. Такий код підходить для демонстрації, але не є масштабованим або зручним для інтеграції в реальні UI-системи. Результат генерації зображений на рисунку 1.3. та рисунку 1.4.

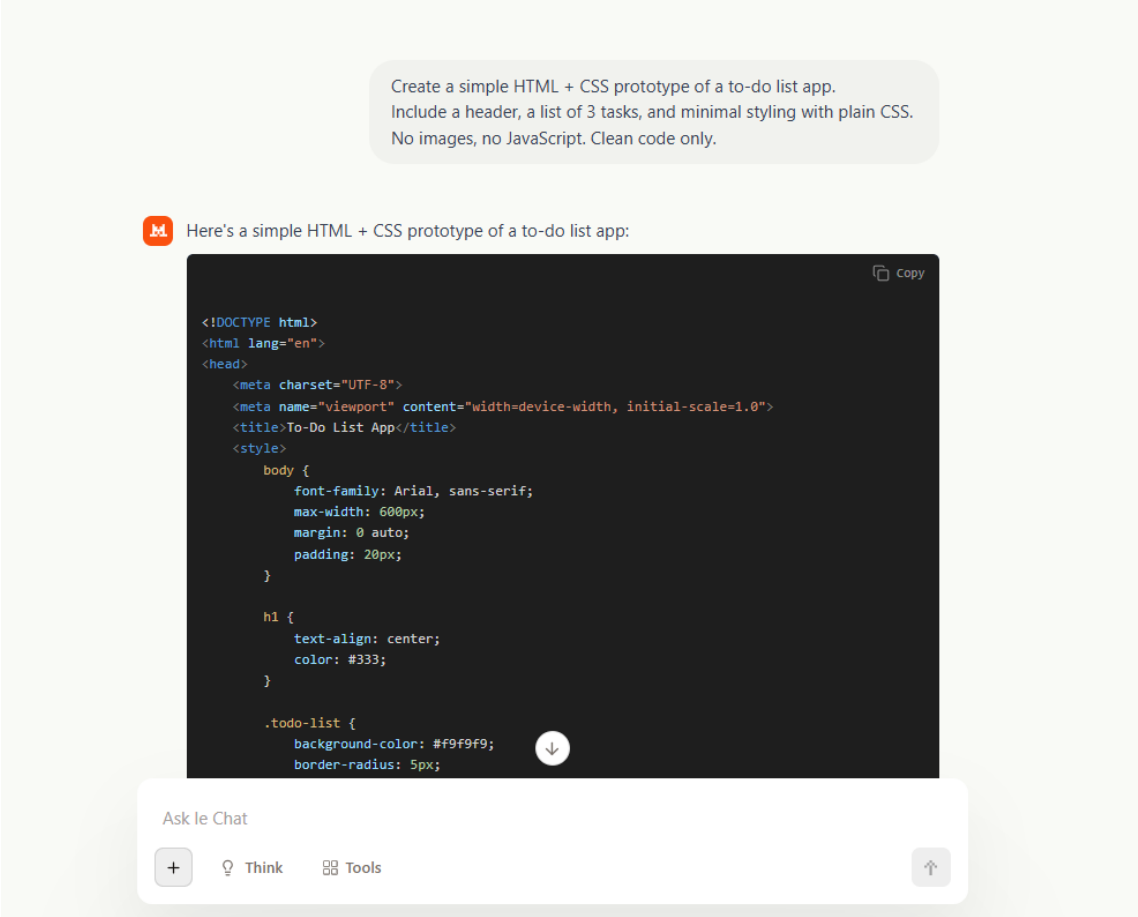


Рисунок 1.3 - Відповідь у чаті Mistral

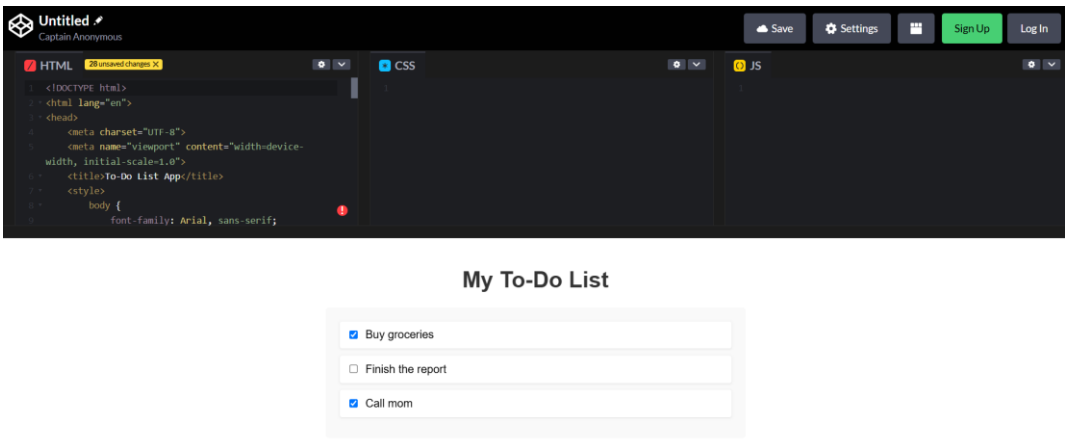


Рисунок 1.4 - Відображення згенерованого коду з відповіді моделі Mistral на платформі codepen.io

DeepSeek-Coder

Ще одна протестована модель, DeepSeek-Coder, є цікавою тим, що спеціалізується на генерації програмного коду. Проте її оптимізація орієнтована переважно на логіку й синтаксис, а не на дизайн або візуальну структуру інтерфейсів. У відповідях вона часто вставляє службові коментарі, описує кроки реалізації або супроводжує код рекомендаціями щодо інтеграції. Навіть якщо результат виглядає структуровано, як-от у випадку з генерацією to-do списку (див. рисунок 1.5), — він залишається надто простим і статичним: без чекбоксів, без інтерактивних елементів, лише базовий список завдань у <ul>. Стилі прописані вручну, без використання CSS-фреймворків, що обмежує повторне використання коду. У контексті задач швидкого прототипування така модель вимагає доопрацювання і не забезпечує достатньої композиційної гнучкості.

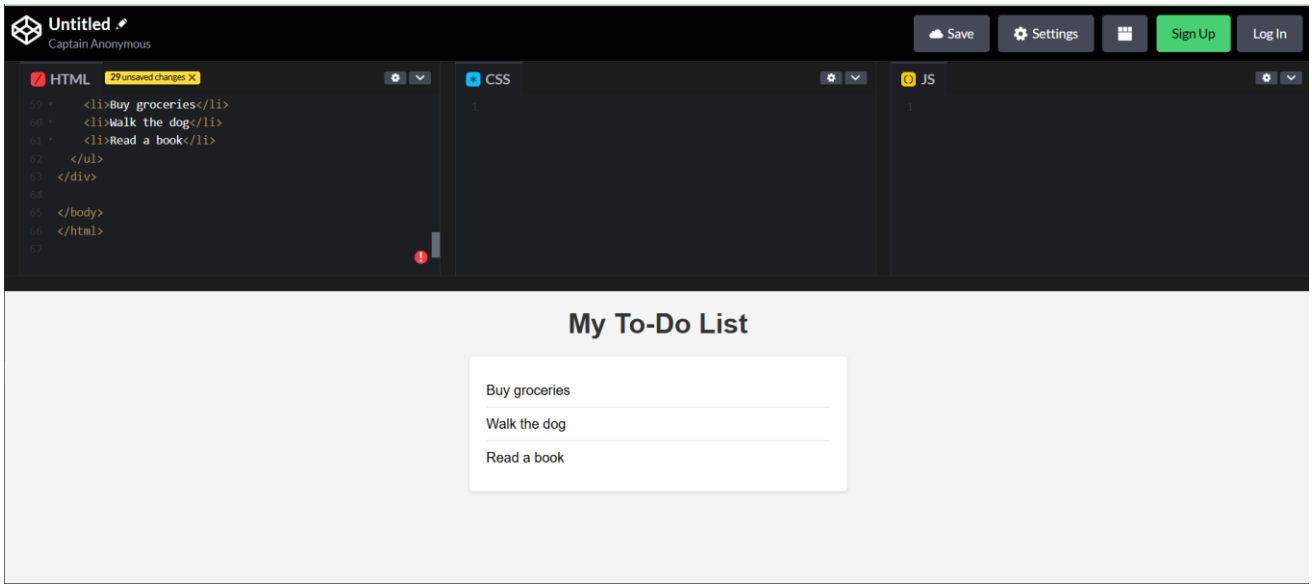


Рисунок 1.5 - Відображення згенерованого коду з відповіді моделі DeepSeek-Coder на платформі codepen.io

У результаті тестування стало очевидно, що жодна з протестованих open-source моделей — Zephyr-7B, Mistral-7B чи DeepSeek-Coder — не демонструє стабільної якості генерації повноцінного HTML+CSS для UI-прототипів без додаткового налаштування. Вони або не дотримуються інструкцій, або створюють неповну чи надто спрощену структуру, або вставляють зайвий супровідний текст. Це підкріплює доцільність використання спеціалізованих або донавчених моделей у задачах автоматизованого UI-генерування.

1.4. Проблемні аспекти й обґрунтування створення UI Crafter

Аналіз існуючих моделей генерації інтерфейсів показав, що жодна з доступних open-source моделей не забезпечує належного балансу між структурною точністю, візуальною повнотою та контрольованістю вихідного HTML [4, 5]. Моделі часто не дотримуються інструкцій, змішують стилі, порушують семантику розмітки або вставляють зайві текстові пояснення [11]. Такі відповіді вимагають постійної ручної перевірки й доопрацювання, що суперечить ідеї автоматизованого прототипування.

Крім того, використання API-комерційних LLM, таких як GPT-4 [10, 13], хоч і забезпечує якісний результат, має власні обмеження — насамперед залежність від

зовнішніх серверів, необхідність підключення до Інтернету, обмеження за швидкістю відповіді та витрати на кожен запит.

У цих умовах постала потреба у створенні інструменту, який:

- дозволяє вибирати джерело генерації (локальне або хмарне),
- забезпечує стабільну якість HTML-виводу,
- виключає пояснення, markdown та службовий контент,
- підтримує стилі, орієнтовані на TailwindCSS.

Відповіддю на ці вимоги стала розробка власної системи UI Crafter — вебплатформи, яка поєднує зручний інтерфейс для введення промту з двома каналами генерації: GPT-4 (через OpenAI API) та кастомною LoRA-моделлю на базі TinyLlama [2, 3], fine-tuned спеціально під HTML/UI макети.

Таке рішення дозволяє:

- забезпечити високу якість генерації для складних запитів за допомогою GPT-4 [10, 13],
- реалізувати автономну локальну генерацію через LoRA-модель без зовнішніх залежностей,
- інтегрувати модуль автоматичного очищення коду (html_utils.py), який усуває всі типові артефакти генерації [11].

UI Crafter орієнтований на вирішення конкретної задачі: швидке, точне та чисте перетворення текстового опису інтерфейсу в придатний до рендерингу HTML+CSS код. Його створення є логічною відповіддю на недоліки існуючих підходів і спробою об'єднати найкраще з двох світів — якість генерації та локальну контрольованість.

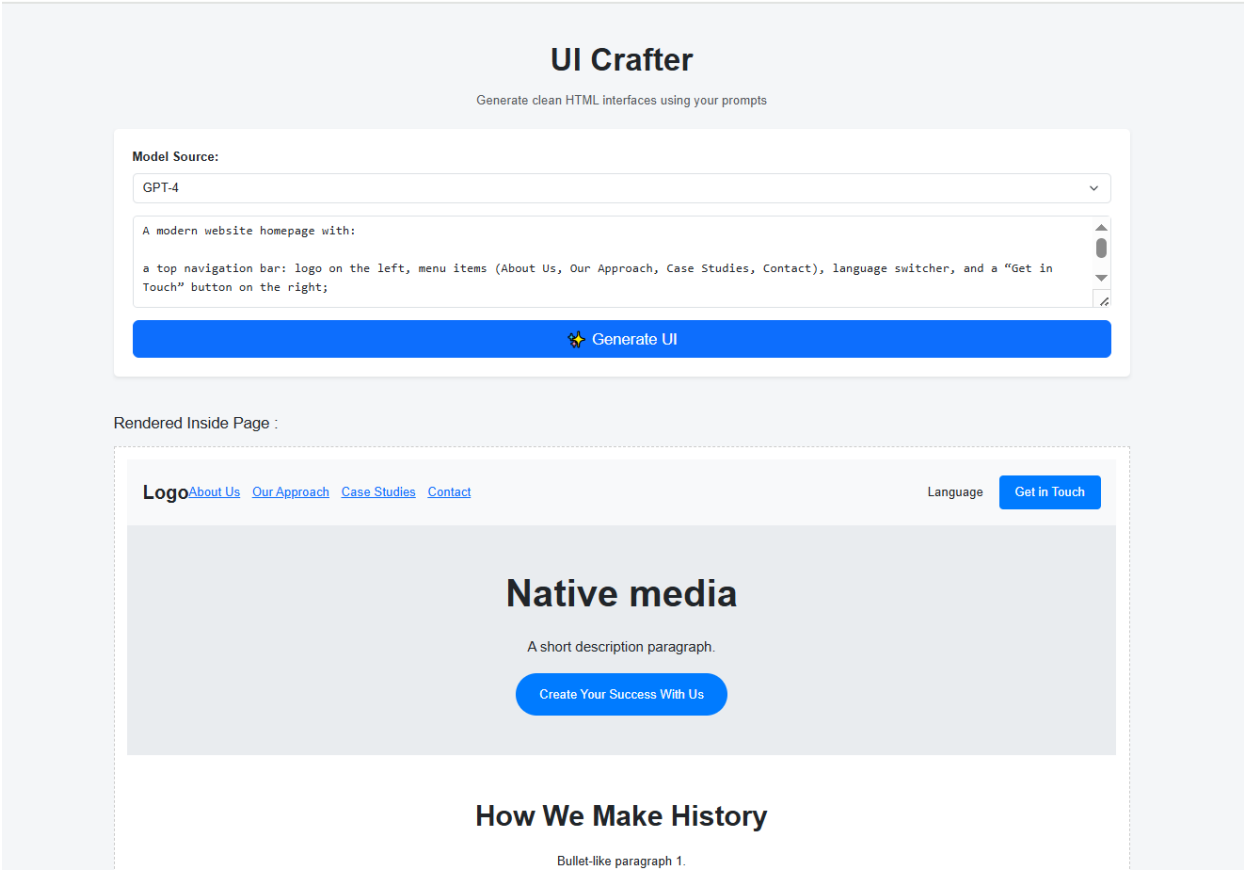


Рисунок 1.6 - Інтерфейс системи UI Crafter

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

2.1 Інформаційна модель

Система UI Crafter виконує послідовну трансформацію текстового запиту користувача у готову HTML-структуру, яка виводиться у браузері. Користувач вводить інструкцію англійською мовою, наприклад: *“Create a landing page with a hero section and a call-to-action button”*. Цей текст є початковим джерелом інформації — так званим prompt’ом, який далі передається в обрану велику мовну модель (GPT-4 або локальна LoRA-модель TinyLlama) [2, 3, 10, 13].

Модель обробляє запит та формує HTML-код, у якому описано елементи інтерфейсу: заголовки, секції, кнопки, сітки тощо. Але оскільки навіть найкращі генерації часто містять зайві частини — наприклад, фрази “Sure, here is the code” або markdown-блоки, система автоматично очищує цей результат за допомогою спеціального скрипта html_utils.py. Він виконує нормалізацію структури: видаляє все зайве, вирівнює відступи, замінює конфліктні класи, та гарантує, що результат узгоджений зі стилістикою TailwindCSS.

Після очищення HTML-код передається в модуль візуалізації — це звичайне відображення результату в межах вебінтерфейсу. Користувач бачить сформовану сторінку у вікні попереднього перегляду без потреби відкривати редактор коду або тестувати вручну.

Такий підхід дозволяє забезпечити простий і зрозумілий ланцюг обробки: від введення до виводу — prompt → генерація → очищення → перегляд. Усі етапи побудовано як окремі незалежні модулі, які передають результат один одному. Це спрощує обслуговування системи, дозволяє легко замінити, наприклад, генератор або тип виводу, і забезпечує стабільну роботу навіть при великій кількості запитів.

Основу моделі становлять п’ять ключових компонентів: Prompt, Controller, LLMGenerator, PostProcessor та Viewer. Кожен із них виконує строго визначену функцію:

- Prompt — текстовий запит, що описує структуру інтерфейсу.
- Controller — модуль, який приймає prompt і передає його в генератор.

- LLMGenerator — інтерфейс, реалізований у вигляді GPT-4 або LoRA-моделі.
- PostProcessor — очищує HTML від зайвих фрагментів.
- Viewer — рендерить результат у вебінтерфейсі.

Передача даних між цими блоками відбувається послідовно, у чітко визначеному порядку (див. рисунок 2.1) . Контролер приймає інструкцію, викликає відповідну модель, отримує «сирий» HTML, очищує його і передає на візуалізацію. Така структура забезпечує як логічну цілісність, так і можливість масштабування — наприклад, заміни генератора на іншу модель без впливу на решту системи.

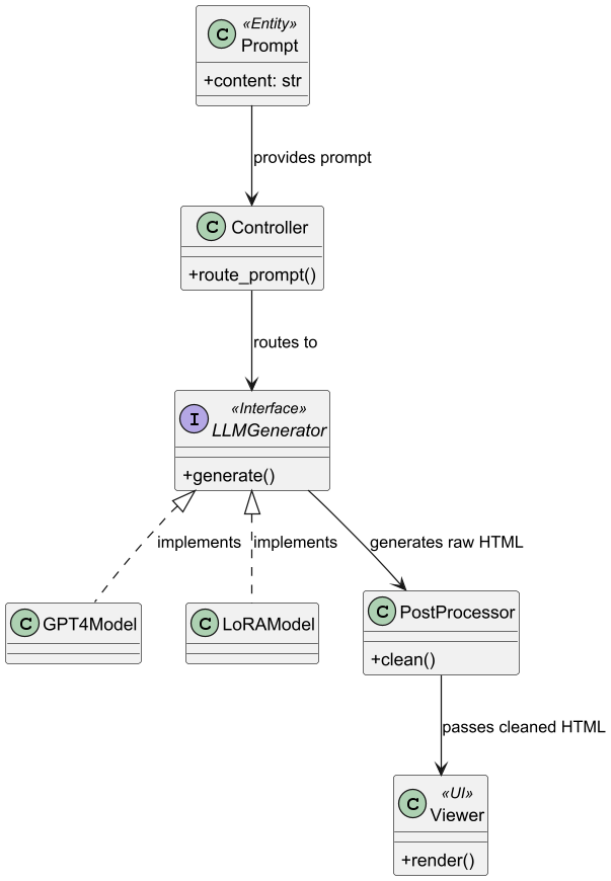


Рисунок 2.1 - UML-діаграма структури інформаційного потоку системи UI Crafter

Пояснення: На рисунку 2.1 представлено повну UML-діаграму сутностей системи UI Crafter, включно з генераторами GPT-4 і LoRA, які реалізують функцію текстової трансформації у HTML-код. Діаграма відображає не лише логіку виклику, але й структуру методів і відповідальність кожного компонента в межах системи.

2.2 Обмеження на вхідні та вихідні дані

Щоб генерація HTML-коду працювала стабільно і передбачувано, всі дані, які надсилаються у систему або отримуються з неї, повинні відповідати певним структурним та змістовим вимогам. У проекті такі обмеження були сформульовані окремо для вхідних запитів і вихідних результатів з урахуванням технічних обмежень мовної моделі, а також специфіки подальшої обробки коду.

Вхідні дані, що надходять від користувача, являють собою текстовий опис бажаного інтерфейсу. Цей текст має бути чітким, стиснутим та сформульованим англійською мовою. Оскільки моделі можуть некоректно трактувати неоднозначні інструкції або вводити зайву інформацію, усі запити подаються у спеціальному шаблоні, який починається з позначки ### Instruction: та завершується ### Response: (див. рисунок 2.2). Такий шаблон формує стабільне середовище для генерації й дозволяє контролювати стиль та структуру відповідей [14]. В рамках системи прийнято обмеження на довжину prompt'a — зазвичай не більше 80 токенів. Це дозволяє уникнути перенавантаження вхідного буфера моделі, зберегти якість генерації та забезпечити швидкість обробки запитів.

```
def generate_html_with_lora(prompt: str) -> str:
    full_prompt = f"""### Instruction:
{prompt.strip()}

Write only HTML and CSS. Do not include lorem ipsum. Do not explain anything.
### Response:"""
```

Рисунок 2.2 - Структура вхідного запиту у форматі prompt-response

На виході користувач отримує HTML-код, який повинен відповідати ряду вимог. Найважливіше — це відсутність будь-якого додаткового тексту. Модель не повинна пояснювати, що вона робить, чи супроводжувати код вступними або завершальними фразами. Також суворо заборонено використовувати “lorem ipsum” або будь-які інші шаблонні тексти, які не несуть семантичного навантаження. Код має містити тільки HTML і CSS, при цьому бажано у вигляді класів TailwindCSS, які підтримуються візуальним рендером системи.

Ще одним обмеженням є відмова від JavaScript. Хоча в багатьох UI-сценаріях JS міг би бути корисним, для задач генерації статичних прототипів у рамках проекту він не є обов'язковим. Його наявність у відповіді ускладнює обробку, створює потенційні проблеми із безпекою, і не підтримується в межах автоматичного прев'ю. Аналогічно, заборонено використовувати зовнішні стилі, зокрема теги <link rel="stylesheet"...>. Усі стилі повинні бути або inline, або реалізовані через TailwindCSS.

Всі вхідні й вихідні дані подаються у форматі UTF-8 plain text, що забезпечує їхню повну сумісність з токенізатором трансформерної архітектури [1, 8]. Для зберігання прикладів prompt → completion використовується формат .jsonl — JSON Lines, де кожен рядок є окремим повноцінним прикладом.

Псевдокод прикладу одного елемента датасету у форматі JSON:

```
{"prompt":
       "Create a user profile card with name, photo, and short bio.",
"completion":
      "\n<div class=\"profile-card\">\n
      <imgsrc=\"https://via.placeholder.com/100\" alt=\"User Photo\">\n
      <h2>Jane Doe</h2>\n  <p>Frontend Developer</p>\n</div>\n"  }
```

Це зручно як для донавчання, так і для автоматизованого тестування чи формування наборів задач. Перед подачею на модель кожен такий приклад вставляється у вищезазначений шаблон, після чого токенізується з паддінгом до 512 токенів. Це дозволяє забезпечити однакову довжину вхідного тензора та уникнути непередбачуваних помилок при генерації [6, 14].

Обмеження, запроваджені в системі UI Crafter, виконують кілька ключових функцій: по-перше, забезпечують стабільну роботу моделі на inference-етапі; по-друге, дозволяють отримувати чистий та структурований HTML, придатний до рендерингу без ручного втручання; по-третє, спрощують автоматичну перевірку результатів, адже уніфіковані шаблони дають змогу програмно виявляти помилки. Нарешті, саме завдяки цим вимогам можливо проводити коректне порівняння якості генерацій між різними моделями та налаштуваннями, що особливо важливо під час донавчання та тюнінгу LoRA-версії [2, 3].

2.3 Модель генерації HTML-коду

Однією з ключових особливостей генерації HTML-коду у системі UI Crafter є використання причинно-наслідкової autoregressive архітектури, що забезпечує послідовний прогноз наступних токенів на основі вже згенерованої послідовності [1, 8, 10]. Це особливо важливо для задач із чіткими синтаксичними вимогами, до яких відноситься HTML-розмітка. Кожен елемент HTML має відкриваючий і закриваючий теги, сувору вкладеність, відповідність атрибутів, а також допустимі комбінації елементів, що створює додаткове навантаження на модель в аспекті контекстного контролю.

При генерації HTML структура документу формується поступово. Спочатку модель відкриває зовнішні обгортки сторінки, такі як <html>, <head>, <body>, а далі поступово формує внутрішні секції: блоки навігації, форми, картки, кнопки, таблиці тощо. Кожен новий тег залежить від попередніх виборів моделі, що дозволяє забезпечити логічну цілісність при коротких інструкціях. Однак зі зростанням складності завдання (наприклад, при побудові складних форм чи багатоетапних макетів) виникає ефект накопичення помилок у вкладеності, неповному закритті тегів, або змішанні стилів.

Цю проблему частково знімає сам принцип autoregressive моделювання, оскільки кожен новий токен прогнозується з урахуванням уже згенерованих елементів, зберігаючи локальну узгодженість вкладень. Однак суто статистична природа LLM залишає певні ризики помилок, особливо у довгих запитах або нестандартних конструкціях. Для компенсації цих помилок в системі UI Crafter впроваджено додаткові механізми стабілізації:

- Стандартизовані шаблони prompt-response – Передача інструкцій у стабільному форматі з чіткою сегментацією ### Instruction: / ### Response: дозволяє зменшити варіативність відповідей.
- Укорочення інструкцій – Обмеження довжини prompt'ів до 80 токенів дозволяє уникати перевантаження контексту і сприяє генерації більш компактних, структурованих відповідей.

- Навчання на вузькій предметній області – Використання спеціалізованих прикладів prompt → HTML при донавчанні LoRA дає змогу моделі фокусуватися на правилах HTML-синтаксису замість загальної генерації тексту [2, 3].
- Детермінізація стилів – Під час підготовки навчальних даних використовувалися приклади з фокусом на TailwindCSS, що дозволяє моделі стабільніше генерувати класи для оформлення елементів.

Крім власне генерації HTML, важливим аспектом роботи є також очищення відповідей. Як GPT-4, так і LoRA можуть повертати службовий текст, наприклад: «Here is your code:», «The following HTML implements your request:», додавати markdown-блоки з ```html, або генерувати коментарі. Усі ці вставки не є валідними частинами HTML-документу, і тому мають бути вилучені ще до подачі коду в модуль візуалізації. Саме для цього у UI Crafter реалізовано окремий модуль PostProcessor реалізований у вигляді скрипта html_utils.py), який автоматизує очистку, нормалізацію відступів, видалення службових маркерів та базову перевірку вкладеності тегів.

Варто зазначити, що autoregressive-моделі демонструють помітно вищу стабільність генерації HTML при роботі з простими, повторюваними структурами — такими як кнопки, картки, прості таблиці чи форми. Для більш складних макетів (dashboard, multi-column layouts, модальні вікна тощо) якість генерації сильно залежить від чіткості сформульованого prompt'a. Це ще раз підкреслює важливість якісного формування навчальних даних для підвищення експертності моделі в предметній області.

Ключовою перевагою застосування autoregressive-LLM в UI Crafter є здатність моделі зберігати високий ступінь контекстної послідовності, забезпечуючи синтаксичну логіку та візуальну цілісність згенерованих прототипів, навіть при обмежених обсягах донавчання LoRA на спеціалізованих корпусах даних [2, 3, 6].

2.4 Проектування інформаційної системи

Розробка системи UI Crafter передбачала створення програмної архітектури, яка б не лише забезпечувала коректну генерацію HTML-коду, але й була адаптована для подальшого розвитку та інтеграції з новими моделями. Для досягнення цієї мети було обрано модульний принцип побудови системи [6, 14].

На відміну від монолітних підходів, модульна архітектура дозволяє чітко розділити відповідальність між компонентами, забезпечуючи тим самим ізоляцію логіки кожного етапу обробки запиту. Це спрощує підтримку системи, дозволяє розширювати її функціонал без внесення змін у ядро та дає можливість легко адаптувати різні генеративні моделі до єдиного інтерфейсу.

Кожен модуль системи UI Crafter працює із власною зоною даних, оперуючи строго визначеним обсягом інформації:

- Контролер (Controller, app.py) — Відповідає за маршрутизацію запитів, вибір моделі генерації та передачу результатів між модулями. Він є центральною точкою взаємодії між компонентами системи.
- Генератор (LLMGenerator) — Забезпечує взаємодію із мовною моделлю: GPT-4 (через API, модуль llm_gpt.py) [10, 13] або LoRA TinyLlama (локально, модуль llm_lora.py) [2, 3]. Обидва генератори реалізують єдиний інтерфейс виклику — це дозволяє легко переключати джерело генерації, не змінюючи структуру запитів.
- Постобробник (PostProcessor, html_utils.py) — Очищає згенерований код від супровідного тексту, маркерів, коментарів, нормалізує вкладеність тегів та забезпечує відповідність синтаксису HTML.
- Інтерфейс візуалізації (Viewer, templates/index.html) — Відображає фінальний HTML-код у вебінтерфейсі користувача.

Такий підхід дозволяє системі залишатись гнучкою до змін. Наприклад, у разі появи нової LLM-моделі достатньо реалізувати для неї окремий адаптер, що імплементує інтерфейс генератора, без необхідності втручання у контролер чи інші компоненти.

Особливістю інформаційного забезпечення в системі є уніфікація форматів обміну даними між модулями. Всі текстові об’єкти передаються у форматі UTF-8 у вигляді простих рядків — це спрощує обробку, усуває проблеми сумісності кодувань та дозволяє використовувати однаковий формат як для генерації, так і для збереження навчальних даних [1, 8].

Такий принципово ізольований і стандартизований підхід до організації інформаційних потоків дозволяє UI Crafter залишатися стабільною системою навіть при значному ускладненні функціоналу чи нарощуванні обсягів генерації.

2.5 Організація навчальних даних для fine-tuning LoRA

Оскільки завдання генерації HTML-коду належить до структурно-складних задач з високими синтаксичними вимогами, особливе значення у процесі fine-tuning локальної LoRA-моделі відіграє якість та специфіка навчального корпусу [2, 3, 6, 14]. У випадку розробки системи UI Crafter створення навчальних даних здійснювалось поетапно, з поступовим ускладненням завдань та розширенням обсягу даних.

Джерела та етапи формування навчального корпусу

Процес створення навчальних даних розпочався із формування базових тестових наборів, що дозволяли перевірити працездатність механізму донавчання LoRA та оцінити реакцію моделі на прості інструкції. Для цих початкових експериментів використовувалися невеликі вручну створені корпуси на 50-100 прикладів, що містили базові HTML-компоненти без ускладненої стилістики та без застосування сучасних CSS-фреймворків.

На цьому етапі основною метою було надати моделі уявлення про базову структуру HTML-документів, вкладеність тегів, організацію простих таблиць, форм, списків і секцій. Стилізація елементів виконувалася або мінімальною (inline), або взагалі була відсутня.

Типовий приклад запису з цього етапу:

```
{"prompt": "Create a web page with pricing table and feature list",
```

"completion": "<html><body><section> Basic Pro Enterprise </section>
Sharable full-range standardization
Thought finish agree clear business.
future
</body></html>"}

Після тестування роботи LoRA на початкових даних було прийнято рішення перейти до наповнення більш структурованого корпусу з використанням реалістичних прикладів сучасних UI-компонентів. З цією метою було частково використано приклади з відкритого набору Wiregen [4], який містив генеративні фрагменти HTML з розміткою форм, карток, кнопок, секцій тощо. На відміну від початкових ручних даних, у прикладах Wiregen вже зустрічались складніші компоненти з ширшою вкладеністю, однак стилістика оформлення залишалась різнорідною — з переважанням звичайних класів CSS, часто без стандартизації.

Вихідні дані Wiregen також потребували глибокої ручної очистки, оскільки містили певну кількість синтаксичних артефактів, зайвих службових вставок, а також випадки використання шаблонних наповнювачів типу Lorem Ipsum [11]. Після проведеної стандартизації на базі цих даних було створено корпус обсягом 500 прикладів (wiregen_merged.jsonl), який ліг в основу основної частини донавчання моделі.

Приклад з даного етапу:

{"prompt": "responsive contact form using Tailwind CSS. Example #1",
"completion": "<section class=\"max-w-xl mx-auto p-6 bg-white rounded-lg shadow\">\n <h2 class=\"text-2xl font-bold mb-4\">Contact Us</h2>\n <form class=\"space-y-4\">\n \n <label class=\"block text-sm font-medium text-gray-700\">Name</label>\n <input type=\"text\" class=\"mt-1 block w-full border border-gray-300 rounded-md p-2\" placeholder=\"Your name\" />\n \n \n <label class=\"block text-sm font-medium text-gray-700\">Email</label>\n <input type=\"email\" class=\"mt-1 block w-full border border-gray-300 rounded-md p-2\" placeholder=\"you@example.com\" />\n \n \n <label class=\"block text-sm font-medium text-gray-700\">Message</label>\n <textarea rows=\"4\" class=\"mt-1 block w-full border border-gray-300 rounded-md p-2\" placeholder=\"Write your message...\"></textarea>\n \n <button type=\"submit\" class=\"w-full bg-blue-600 text-white py-2 px-4 rounded hover:bg-blue-700\">Send Message</button>\n </form>\n</section>"}

На фінальному етапі додатково формувався вже цільовий корпус combined.jsonl обсягом близько 600 прикладів, який містив повністю очищені,

синтаксично стабільні приклади з чітко витриманою структурою. Саме на цьому етапі в навчальних даних було вперше запроваджено систематичне використання стилістики TailwindCSS [16], що дозволило стабілізувати генерацію стилів у фінальній версії моделі.

Приклад запису з цього етапу:

{"prompt": "Create a user profile card with name, photo, and short bio.", "completion": "<div class=\"profile-card\">\n <img src=\"https://via.placeholder.com/100\" alt=\"User Photo\">\n <h2>Jane Doe</h2>\n Frontend Developer \n "}

Якість підготовки корпусу та її вплив на результати генерації

Одним із ключових факторів успіху fine-tuning процесу стало не лише формування корпусу, а й його системна стандартизація та валідація. З огляду на синтаксичну чутливість HTML, кожен навчальний приклад проходив ретельну перевірку перед включенням до фінального набору.

Під час очистки особлива увага приділялась правильності закриття тегів, збереженню вкладеності елементів, коректності атрибутів, а також усуненню будь-яких службових вставок: коментарів, пояснювальних фраз, маркерів Markdown або шаблонних текстів типу Lorem Ipsum. Значна робота була виконана і щодо стилістичної уніфікації — всі приклади максимально стандартизовувались під використання класів TailwindCSS [16], що забезпечувало однакову стилістику в подальшій генерації.

Крім змістовної очистки, всі дані додатково проходили токенізацію з паддінгом до довжини 512 токенів [6, 14]. Це дозволяло не лише забезпечити стабільний розмір вхідних тензорів під час тренування, а й сприяло більш рівномірному розподілу обчислювального навантаження на локальних апаратних ресурсах.

Саме висока якість попередньої стандартизації дозволила досягти суттєвого підвищення стабільності та точності роботи донавченої LoRA-моделі. Після завершення навчання модель демонструвала здатність:

- утримувати правильну вкладеність складних HTML-структур;
- уникати появи службового тексту у згенерованих відповідях;

- зберігати стабільність TailwindCSS-класів навіть при складних запитах;
- суттєво знизити кількість синтаксичних та структурних помилок у кінцевому HTML-коді.

Таким чином, саме якісна підготовка навчального корпусу стала фундаментальним інформаційним забезпеченням для ефективної роботи системи генерації UI-прототипів у межах UI Crafter.

2.6 Порівняння генерації GPT-4 та LoRA-моделі TinyLlama у задачах генерації HTML-коду

У системі UI Crafter передбачено можливість використання двох альтернативних джерел генерації HTML-коду: комерційної великої мовної моделі GPT-4 (через зовнішній API OpenAI) [10, 13] та локальної донавченої моделі TinyLlama-1.1B, модифікованої за допомогою LoRA fine-tuning [2, 3, 6]. Вибір генератора здійснюється безпосередньо на етапі формування запиту, що дозволяє оцінювати переваги й обмеження кожної моделі у реальних умовах функціонування системи.

Таблиця 2.1 - Характеристики моделей

Параметр	**GPT-4**	**LoRA TinyLlama**
Архітектура	GPT-4 (закритий протокол)	TinyLlama 1.1B (open-source)
Доступність	API-зовнішній сервер	Повністю локальна
Обсяг параметрів	+\ - 1 трлн	+ \ - 1.1 млрд
Наявність донавчання під задачу	Стандартна генерація	Fine-tuned під HTML
Потреба у підключенні до Інтернету	Так	Ні
Швидкість відповіді	Висока, але з latency	Швидше для локальних задач
Вартість генерації	Оплата за кожен запит	Разові витрати на fine-tuning

Якість синтаксису і стабільність

GPT-4: Демонструє високу якість побудови HTML-структур навіть при складних запитах, добре утримує вкладеність тегів, майже повністю відсутні синтаксичні помилки у стандартних інтерфейсних компонентах. Однак навіть при чітко сформульованих інструкціях GPT-4 нерідко повертає службовий текст на кшталт «Here is your code:», маркери Markdown (```html) або пояснювальні вставки. Саме ці артефакти ускладнюють автоматичне використання результатів у системах з подальшою обробкою HTML [11].

LoRA TinyLlama: Завдяки спеціалізованому донавчанню на стабілізованому корпусі prompt → HTML система демонструє більшу чистоту відповідей — практично повністю усунуто службові вставки, стабілізована вкладеність тегів, вироблено сталість у використанні класів TailwindCSS [16]. При цьому у складних довгих запитах LoRA все ще може демонструвати деяку нестабільність у завершенні вкладеності при перевищенні оптимальної глибини, однак якість у типовому діапазоні запитів для UI-прототипів є високою.

У цьому розділі було визначено, які дані необхідні для навчання моделі та як їх підготувати. Я побудувала повний цикл обробки — від формування навчального корпусу до механізмів генерації HTML-коду та його подальшої стабілізації. Також обґрунтовано використання LoRA-адаптації для донавчання на обмежених локальних ресурсах, розроблена структура дозволила досягти стабільної генерації HTML-прототипів та забезпечила можливість подальшого розвитку системи.

РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Обґрунтування вибору технологічного стеку

У процесі розробки системи UI Crafter основним завданням було забезпечити стабільну генерацію повноцінних HTML-прототипів інтерфейсів з використанням великих мовних моделей. Для досягнення цього обрано комбінацію перевірених фреймворків, бібліотек машинного навчання та гнучкої серверної архітектури [6, 14, 7].

В основі розробки системи лежить мова програмування **Python 3.10+** [17], яка забезпечує широкі можливості у сфері роботи з трансформерними моделями, вебсервісами та обробкою текстових даних. Обрано такі основні бібліотеки:

- HuggingFace Transformers— інтеграція з великими мовними моделями, включаючи TinyLlama та GPT-4 [14];
- PEFT— реалізація механізму донавчання моделей через Low-Rank Adaptation (LoRA) [6];
- PyTorch (torch 2.5.1+cu121) — ядро глибокого навчання [7];
- HuggingFace Datasets— обробка та завантаження навчальних датасетів у форматі JSONL [14];
- Flask— побудова серверної частини REST API та вебінтерфейсу [15];
- BeautifulSoup4 та RegExp — постобробка HTML-коду після генерації (модуль html_utils.py) [11];
- dotenv— зберігання API-ключів для безпечної роботи з OpenAI;
- OpenAI API— інтеграція із зовнішнім GPT-4 [10, 13].

Всі бібліотеки централізовано керуються через файл requirements.txt, що забезпечує реплікабельність середовища розробки.

З огляду на необхідність забезпечити стабільний серверний доступ до генеративних моделей та реалізувати користувацький вебінтерфейс, було обрано фреймворк Flask як основу серверної частини. Flask дозволив організувати легковаговий backend для маршрутизації запитів, реалізувати API для взаємодії з моделями, а також інтегрувати систему шаблонізації Jinja2 для динамічного наповнення вебсторінок [15].

Користувацький інтерфейс було реалізовано максимально просто, з акцентом на функціональність генерації (див. рисунок 3.1). Графічна частина використовує HTML5 та CSS3 із підключенням Bootstrap 5.3.3 через CDN для створення адаптивної верстки та покращення візуальної складової без складних фронтенд-фреймворків [16]. Водночас генерація самих UI-прототипів у HTML-коді забезпечується за допомогою стилів TailwindCSS, які інтегруються безпосередньо у згенеровану модельню HTML-розмітку на основі отриманого prompt [16].

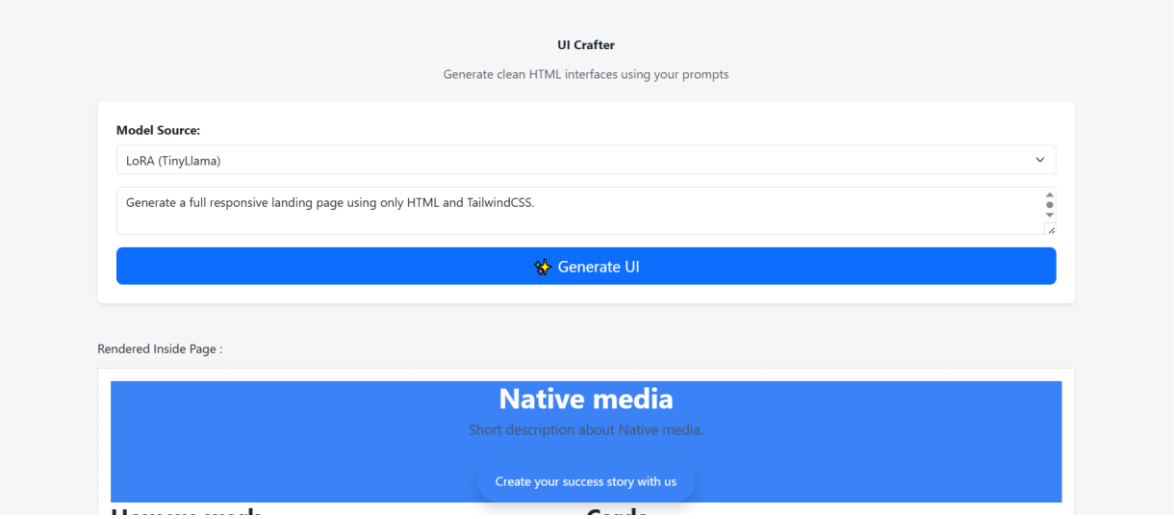


Рисунок 3.1 - Приклад користувацького інтерфейсу UI Crafter

Щодо генеративних моделей, у системі реалізовано дві незалежні лінії генерації. Перша — це SaaS-рішення GPT-4, що працює через OpenAI API [10, 13] та дозволяє отримати високоякісні результати вже при стандартних параметрах. Друга — локальна модель TinyLlama-1.1B Chat із застосуванням LoRA-адаптації, яка проходила донавчання на власноруч підготовлених корпусах даних [2, 3, 4, 5]. До навчальних датасетів входили: dataset_wire100.jsonl як тестовий початковий набір, wiregen_augmented.jsonl та wiregen_merged_better.jsonl з автоматично згенерованими

прикладами з Wiregen [4], wirefigma_extended.jsonl з вручну розміченими даними з Figma, а також фіналізовані очищені датасети combined.jsonl та final_dataset_500.jsonl.

Всі ці корпуси формувались у форматі prompt-completion, де вхідні запити будувались за уніфікованою структурою. Зокрема, кожен запит містив блок Instruction, у якому користувач формулює завдання, наприклад: *«Create a TailwindCSS layout for a FAQ accordion»*, після чого модель генерує відповідь у блоці Response, де міститься лише очікуваний HTML-код, без службових вставок чи додаткових коментарів [4] (див. рисунок 3.2).

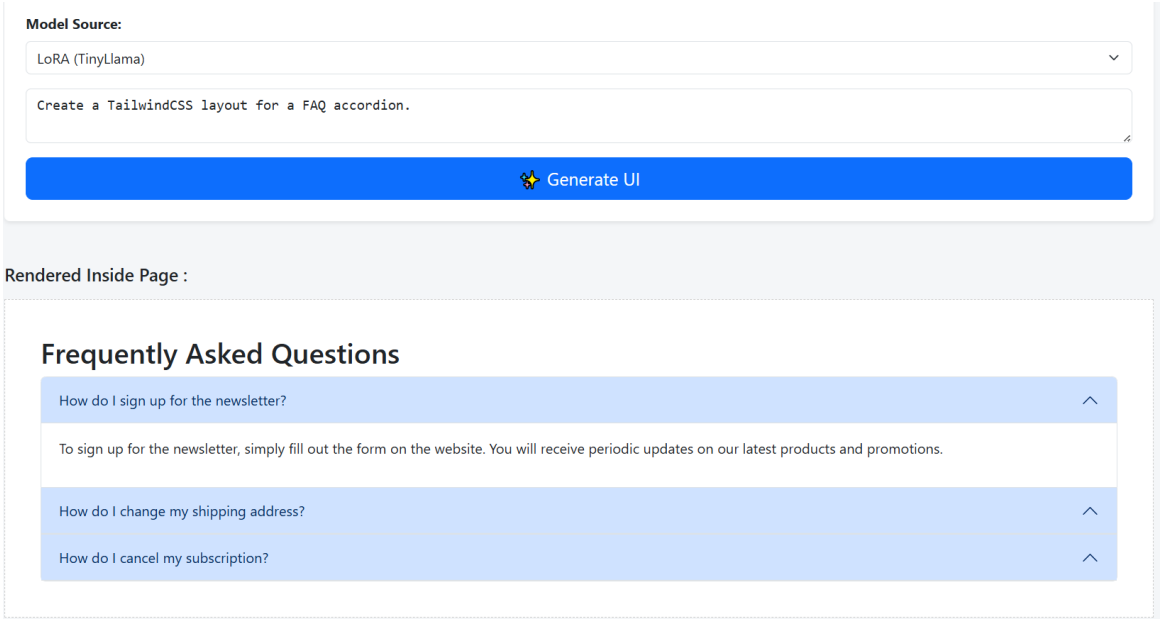


Рисунок 3.2 - Приклад згенерованого прототипу інтерфейсу за запитом LoRA-моделі

Такий вибір технологій дозволив створити гнучку, стабільну та функціонально завершену систему генерації прототипів UI на основі LLM, яка поєднує сучасні генеративні алгоритми, модульну архітектуру, локальну автономність та можливість подальшого масштабування.

3.2 Апаратне забезпечення та середовище розробки

Розробка та навчання системи UI Crafter здійснювались із розрахунку на локальне розгортання усіх компонентів, включаючи повний цикл fine-tuning моделей з використанням LoRA-адаптації [6]. Такий підхід дозволив уникнути залежності від сторонніх хмарних платформ, забезпечити повний контроль над навчальним процесом і даними, а також зменшити витрати на обчислювальні ресурси.

Для реалізації донавчання мовної моделі TinyLlama-1.1B у локальному середовищі застосовувалась персональна обчислювальна станція із процесором Intel Core i7 дванадцятого покоління, оперативною пам’яттю 32 гігабайти та графічною картою NVIDIA RTX 4070. Навчання моделі виконувалось під керуванням операційної системи Windows 10 із встановленим середовищем CUDA [9], що забезпечило сумісність із бібліотеками глибокого навчання PyTorch та HuggingFace [7, 14]. Обрана конфігурація дозволила повністю реалізувати процес донавчання моделей у локальних умовах без залучення хмарних сервісів.

Навчені ваги моделей зберігались у відповідних каталогах, зокрема: tinyllama_wiregen_lora, tinyllama_lora_augmented, tinyllama_lora_htmlgen. Така організація дозволяла ізолювати різні фази донавчання та зберігати повну історію експериментів.

Розгортання самої системи генерації HTML-коду після завершення навчання можливе як на графічному процесорі для досягнення максимальної продуктивності, так і на центральному процесорі при обмежених ресурсах. Запуск локальної генерації через LoRA-модель здійснюється незалежно від доступу до мережі Інтернет, у той час як генерація за допомогою GPT-4 потребує наявності активного API-з’єднання з сервісами OpenAI [10, 13].

Розроблена архітектура середовища забезпечила повну автономність роботи системи, гнучкість експериментів із навчальними даними, стабільність генерації та можливість розширення функціоналу в майбутньому.

3.3 Технічні вимоги до запуску системи

UI Crafter не потребє складної інфраструктури для запуску та може бути розгорнута на стандартних персональних комп'ютерах залежно від обраного режиму роботи. У разі використання генерації через GPT-4 система працює через API-сервіс OpenAI [10, 13], тому основні обчислення виконуються на стороні сервера OpenAI. Для запуску цього режиму достатньо мати актуальну версію Python не нижче 3.10 [17], коректно встановлені бібліотеки із requirements.txt, стабільне підключення до мережі Інтернет та діючий API-ключ OpenAI.

У режимі локальної генерації через донавчену LoRA-модель система також може бути розгорнута на звичайних персональних комп'ютерах із встановленим середовищем Python та повним набором бібліотек [6, 7, 14]. Використання центрального процесора можливе навіть без наявності окремої графічної карти, однак у такому випадку генерація HTML-коду виконується із суттєво меншою швидкістю. Для забезпечення оптимальної продуктивності при локальному запуску бажаною є наявність графічного прискорювача з підтримкою CUDA [9] та сумісних драйверів, що дозволяє значно прискорити генерацію у випадку роботи з більшими запитами.

Завдяки повній автономності архітектури користувач системи має змогу запускати генерацію як локально на власних обчислювальних ресурсах, так і через зовнішнє API, без потреби у додаткових складних налаштуваннях середовища.

3.4 Структура програмної реалізації

Програма реалізована у вигляді локального серверного застосунку з гнучкою архітектурою, що дозволяє інтегрувати різні моделі генерації та забезпечує повний цикл обробки запитів користувача — від введення prompt до отримання згенерованого HTML-коду [6, 14]. Уся логіка роботи системи поділена на ряд взаємодіючих модулів, кожен із яких виконує окрему функціональну роль у загальній архітектурі.

Файлова структура проекту побудована таким чином, щоб чітко розділити компоненти генерації, навчання, інтерфейсу та зберігання моделей (див. рисунок 3.3). У кореневій директорії проєкту розташовані основні програмні модулі, файли з

навчальними корпусами та каталоги з донавченими вагами моделей. Серверна частина системи реалізована у вигляді модуля app.py, який відповідає за прийом вхідних запитів від користувача, вибір джерела генерації, передачу запиту у відповідну модель, отримання відповіді та передачу її у модуль постобробки.

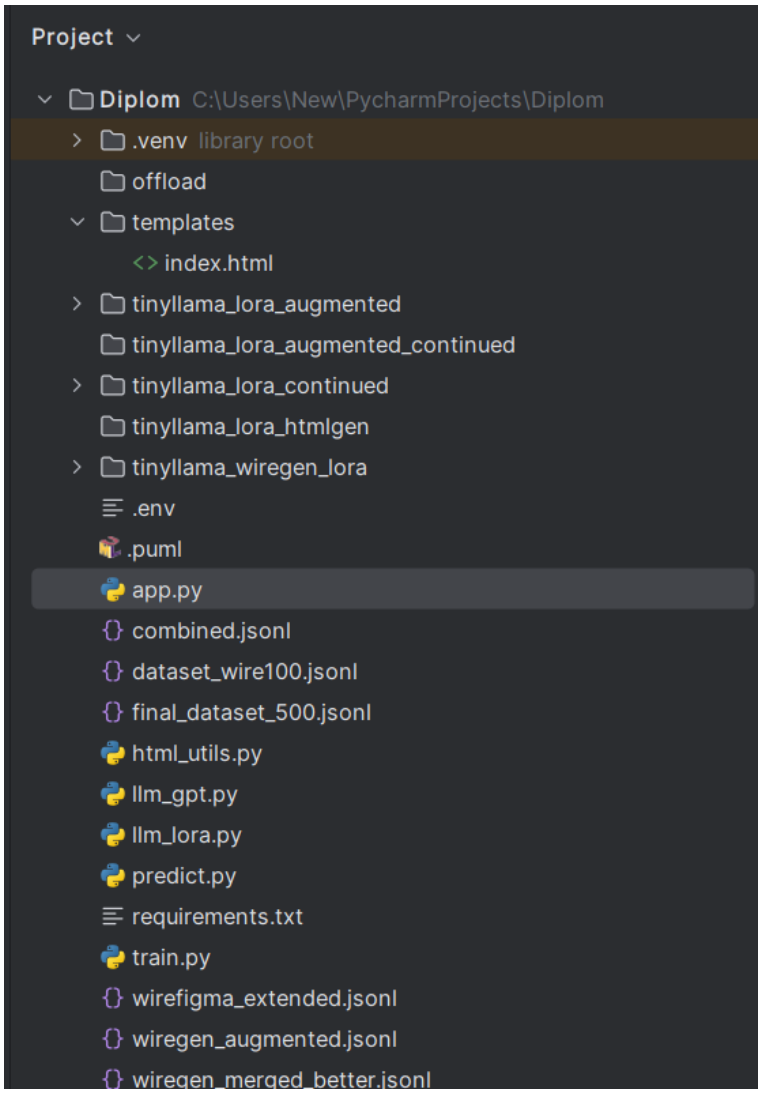


Рисунок 3.3 - Структура файлів проекту

Обробка запитів до GPT-4 виконується за допомогою модуля llm_gpt.py, який реалізує взаємодію з OpenAI API [10, 13], формує запит у відповідному форматі та приймає відповідь від моделі. Для генерації за допомогою локальної LoRA-моделі реалізовано окремий модуль llm_lora.py, який здійснює завантаження збережених ваг моделі, формування вхідної послідовності токенів та генерацію HTML-коду у режимі inference [2, 3].

Окрему роль у структурі системи виконує модуль html_utils.py, який реалізує алгоритми стабілізації та постобробки згенерованого HTML. Він відповідає за видалення службових маркерів, очищення markdown-вставок, нормалізацію структури тегів, усунення можливих артефактів у вигляді lorem ipsum, а також стандартизацію класів TailwindCSS, що використовуються при генерації макетів [11].

У структурі проекту також передбачено наявність окремого модуля для навчання моделі. Модуль train.py відповідає за процес донавчання моделі TinyLlama-1.1B із застосуванням Low-Rank Adaptation через бібліотеку PEFT [6]. У цьому файлі реалізовано підготовку навчальних даних, токенізацію prompt-completion пар, створення навчальних датасетів у форматі JSONL [14] та запуск процесу оптимізації параметрів моделі із збереженням нових ваг. Під час експериментів різні фази навчання зберігались у вигляді окремих директорій: tinyllama_wiregen_lora, tinyllama_lora_augmented, tinyllama_lora_htmlgen та інших, що дозволяло ізолювати кожен етап fine-tuning. Після завершення донавчання генерація відповідей за навченою LoRA-моделлю виконується через модуль llm_lora.py, який завантажує збережені ваги та обробляє нові запити користувачів у процесі роботи системи. Перевірка стабільності роботи моделі здійснювалась безпосередньо через основний серверний додаток у файлі app.py, де навчена модель інтегрується у загальний цикл генерації інтерфейсів без використання окремих скриптів тестування.

Шаблон вебінтерфейсу системи реалізовано у каталозі templates, у файлі index.html. Даний шаблон обробляється сервером Flask із використанням системи Jinja2-шаблонізації, що дозволяє динамічно формувати сторінку відображення результатів генерації HTML-коду [15] (див. рисунок 3.4).

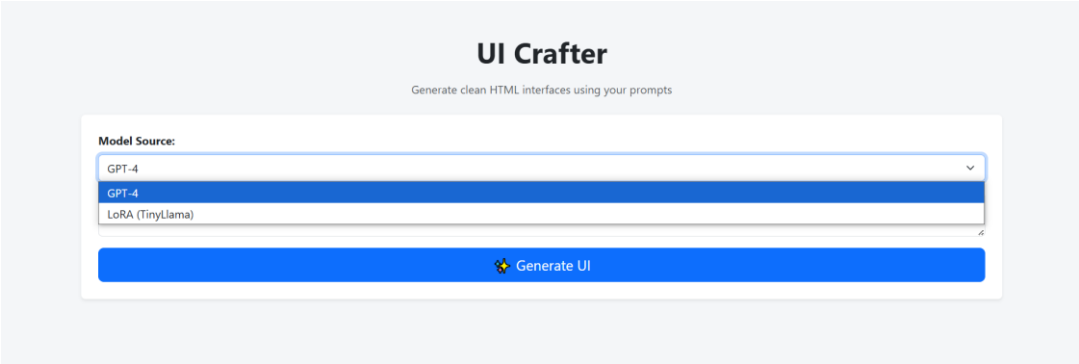


Рисунок 3.4 - Основна форма генерації

Навчальні дані організовано у вигляді текстових файлів формату JSONL, зокрема: dataset_wire100.jsonl, wiregen_augmented.jsonl, wiregen_merged_better.jsonl, wirefigma_extended.jsonl, combined.jsonl та final_dataset_500.jsonl [4, 5]. Кожен із цих файлів містить підготовлені приклади у форматі prompt-completion, що використовувалися під час різних фаз донавчання LoRA-моделі.

3.5 Алгоритм роботи системи

Робота системи починається з взаємодії користувача із вебінтерфейсом, де у спеціально відведеному полі форми, реалізованої через шаблон index.html, вводиться текстовий запит. Запит містить детальний опис необхідного макету інтерфейсу, зокрема розташування елементів, їхню структуру, стилізацію та логіку відображення з використанням класів TailwindCSS [16].

Після подання запиту, серверна частина системи, реалізована у модулі app.py на базі Flask [15], приймає текст prompt і визначає джерело генерації. Залежно від вибору моделі генерації запит передається або у модуль llm_gpt.py для генерації через зовнішній API OpenAI GPT-4 [10, 13], або у модуль llm_lora.py для генерації через локально донавчену модель TinyLlama-1.1B з LoRA-адаптацією [2, 3, 6].

Отримана відповідь моделі у вигляді raw output — тобто сирого HTML-коду згенерованого LLM — повертається назад до контролера app.py. Враховуючи те, що навіть після генерації часто присутні службові вставки, службовий текст або додаткові фрагменти, на наступному етапі цей код передається до модуля постобробки html_utils.py [11]. У цьому модулі реалізовано повний цикл очистки

результатів генерації, який включає видалення markdown-блоків, вирізання службових вставок на кшталт head, html або пояснювальних коментарів, обрізання залишкових некоректних фрагментів після закриття HTML-документу, очищення вставок lorem ipsum, а також корекцію потенційних конфліктних CSS-класів.

Після завершення етапу постобробки очищений та стабілізований HTML-код повертається у серверну частину і виводиться у браузері для попереднього перегляду. Користувач бачить результат генерації безпосередньо у вікні браузера, при цьому у нижній частині сторінки також виводиться сирий (raw) HTML-код, що дозволяє здійснювати додатковий контроль роботи моделі або дебаг при потребі.

Система працює у повністю "одноразовому циклі" без використання баз даних, збереження історії запитів чи кешування результатів. Кожен запит обробляється незалежно: prompt → генерація → постобробка → візуалізація результату.

3.6 Архітектурні особливості та модульність

Взаємодія окремих компонентів у системі UI Crafter реалізована через послідовне проходження запиту усіма необхідними етапами обробки. Така організація дозволяє забезпечити чіткий розподіл обов’язків між модулями та спрощує як поточне обслуговування системи, так і її можливе розширення [6, 14].

Процес роботи починається з контролера, розташованого у файлі app.py [15]. Саме він приймає вхідний запит з вебінтерфейсу, аналізує параметри запуску та визначає, яка саме модель буде використана для генерації. Подальша маршрутизація залежить від вибору користувача: у разі роботи з віддаленим сервісом OpenAI активується модуль llm_gpt.py [10, 13], тоді як для локальної генерації система передає запит у llm_lora.py [2, 3].

Отриманий у результаті роботи моделі HTML-код потребує подальшої обробки, оскільки генератори можуть додавати службові вставки, пояснювальні конструкції чи нестандартні синтаксичні фрагменти. Для вирішення цього завдання призначений модуль html_utils.py [11]. Він проводить повну очистку коду, усуває

зайві фрагменти, коригує вкладеність тегів, стабілізує структуру документа та уніфікує стилізацію за допомогою класів TailwindCSS [16].

Логіку донавчання моделей локальної генерації винесено у спеціалізований скрипт train.py - тут реалізовано підготовку навчальних датасетів, їх токенізацію та сам процес оптимізації параметрів LoRA-моделі із збереженням результатів для подальшого використання [6, 14].

Інтерфейс користувача побудований із застосуванням шаблонів Flask. Шаблон index.html забезпечує взаємодію з користувачем, прийом запитів, відображення результатів генерації та надає можливість перегляду як фінального HTML, так і сирого згенерованого коду.

Всі зовнішні залежності системи згруповані у файлі requirements.txt, що спрощує розгортання середовища розробки. Конфіденційна інформація (зокрема API-ключі OpenAI) зберігається у файлі .env, що дозволяє ізолювати параметри доступу від основного коду.

3.7 Експериментальні дослідження роботи системи

З метою оцінки роботи системи UI Crafter проведено серію експериментальних тестувань генерації HTML-прототипів інтерфейсів із використанням двох моделей: донавченої локальної LoRA-моделі TinyLlama-1.1B та моделі GPT-4 від OpenAI [2, 3, 6, 10, 13]. Мета експерименту — дослідити якість генерації, повноту відтворення структури, а також стабільність роботи моделей на ідентичних запитах.

Для тестування використовувались різні типи запитів (prompt), що відображають реальні сценарії створення вебінтерфейсів [4, 5]. Зокрема було сформовано серії завдань на створення повноцінних landing page, FAQ-компонентів та багатосекційних сторінок з різними блоками.

Експеримент 1: Генерація landing page

В обох моделях застосовувався prompt:

Generate a full responsive landing page using only HTML and TailwindCSS.

General layout:

- White background with modern minimalistic design.

- Use only TailwindCSS utility classes. No external styles or JavaScript.

- Use default Tailwind font.

- Apply appropriate padding and spacing between all sections.

Header (Navigation Bar):

- Use flex justify-between layout.

- Left: text placeholder "Logo".

- Center: navigation menu with links: "Про нас", "Наш підхід", "Кейси", "Контакти".

- Right: language selector "EN | UA" and button "Зв'яжіться з нами" (bg-blue-500 text-white rounded px-4 py-2 shadow).

Hero Section:

- Center vertically and horizontally.

- Title: "Native media" (text-4xl font-bold).

- Subtitle: "Short description about Native media." (text-gray-600 text-lg mt-2).

- Call-to-action button: text "Створити свій успіх разом з нами" (bg-blue-500 text-white rounded-full px-6 py-3 shadow mt-4).

How We Work Section:

- Title: "Як ми творимо історію" (text-3xl font-bold mt-16 mb-4 text-center).

- Below: three horizontally aligned paragraphs: "Paragraph 1", "Paragraph 2", "Paragraph 3" (flex justify-center gap-6 text-gray-700).

Cards Section:

- Title: "Креатив у цифрах" (text-3xl font-bold mt-16 mb-4 text-center).

- Display 4 cards in grid layout: grid grid-cols-1 md:grid-cols-4 gap-6.

- Each card: gray placeholder box (w-full h-40 bg-gray-200 mb-2) and label "Card 1", "Card 2", "Card 3", "Card 4".

Important:

- Output only valid HTML with embedded TailwindCSS classes.

- Do not add any explanations, comments, or markdown.

- Do not include lorem ipsum or example filler texts.

- The output should start directly with HTML.

У результаті генерації LoRA модель (див. рисунок 3.5) сформувала повноцінний макет з усіма передбаченими секціями. Проте візуальне наповнення залишилось спрощеним: застосовані більш базові стилі, відчувалась менша глибина опрацювання адаптивності та міжсекційних відступів. Структура блоків збережена, і

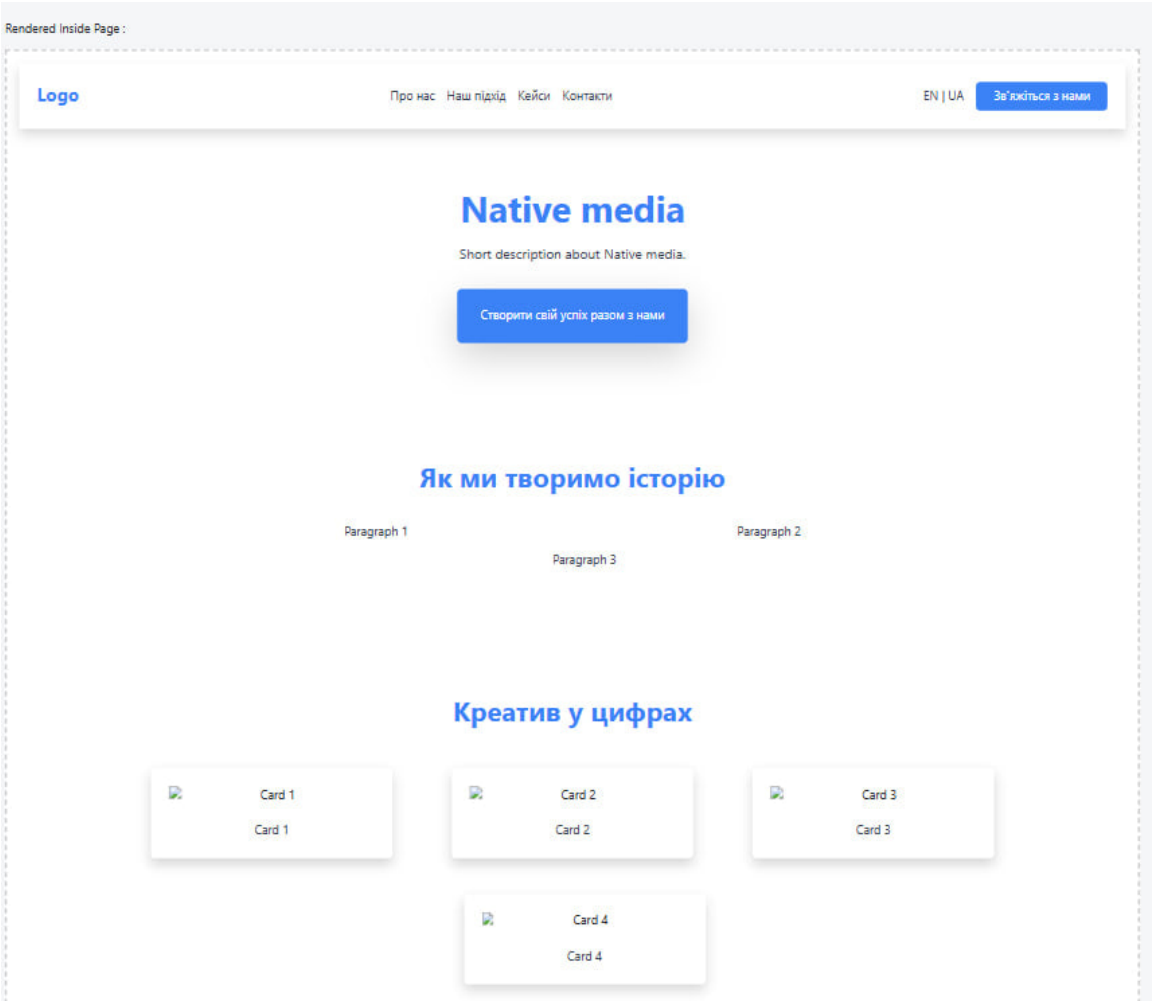


Рисунок 3.5 - Результат генерації landing page (LoRA)

У той же час GPT-4 (див. рисунок 3.6) продемонстрував глибшу генерацію з повноцінним відображенням всіх блоків сторінки. Була коректно побудована адаптивна сітка карток статистики, центрування контенту в hero section, наповнення усіх кнопок стилями, застосування семантичних TailwindCSS-класів відповідно до завдання. Загальний вигляд GPT-4 наближався до реального production-макету [16].

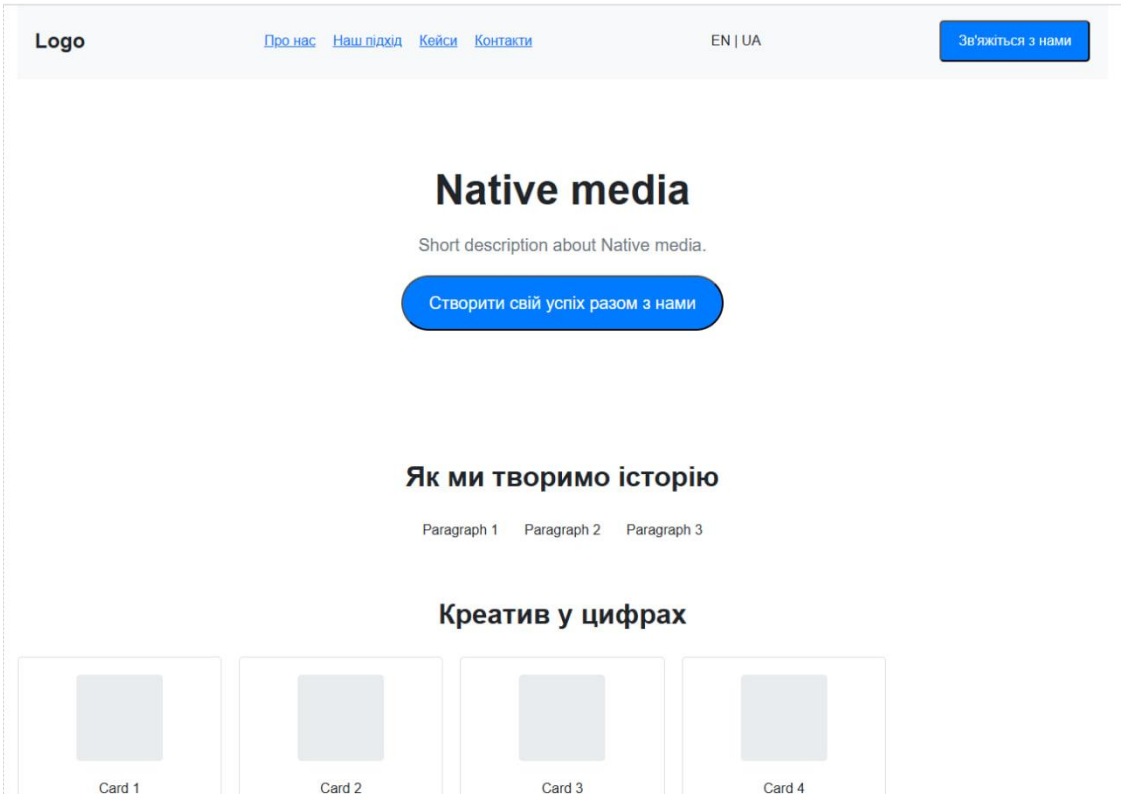


Рисунок 3.6 - Результат генерації landing page (GPT-4)

Експеримент 2: Генерація FAQ акордеону

Для другого експерименту обидві моделі отримали однаковий prompt: Create a layout for a FAQ accordion.

У випадку роботи LoRA (TinyLlama-1.1B з LoRA-адаптацією), модель згенерувала базову текстову структуру (див. рисунок 3.7). Було побудовано заголовок секції, перелік запитань і відповідей, а також застосовано базову стилізацію за допомогою TailwindCSS-класів для кольору фону та відступів. Однак функціональна поведінка акордеону — динамічне відкривання і закривання відповідей — не була реалізована. Крім цього, у коді LoRA залишились службові вставки markdown-формату (```html), які є типовим артефактом при генерації моделей, що не мають вбудованих механізмів фільтрації таких службових блоків [11]. Отриманий результат відображав лише статичний контент без інтерактивності.

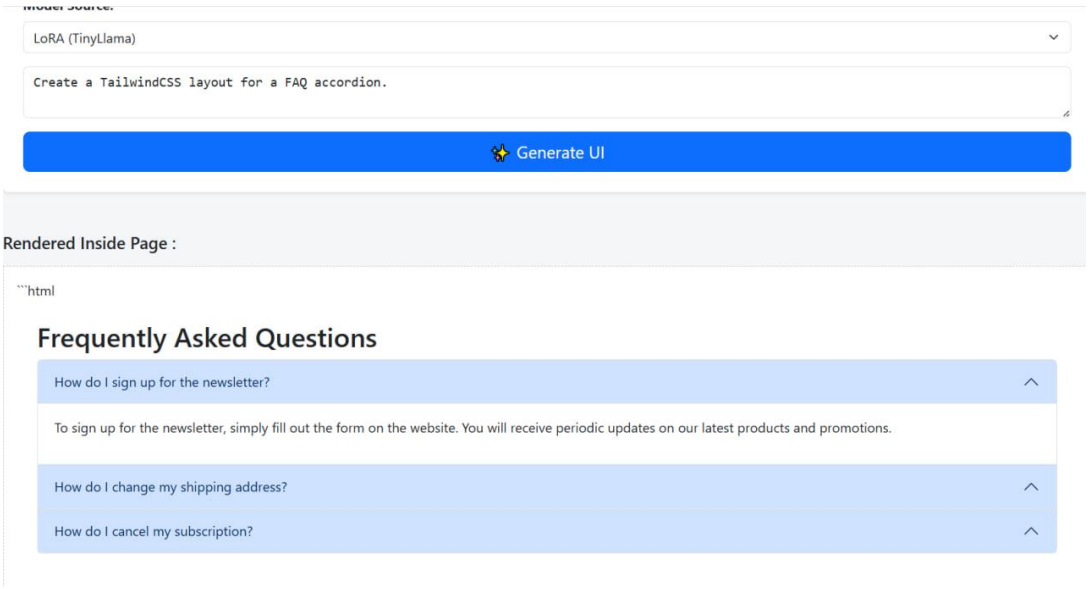


Рисунок 3.7 - Результат генерації FAQ (LoRA)

У свою чергу, GPT-4 (див. рисунок 3.8) згенерував повноцінний функціональний компонент FAQ акордеону. Було коректно сформовано заголовок, усі запитання з відповідями та реалізовано повну поведінкову логіку розгортання та згортання блоків. Крім того, GPT-4 застосував розширені можливості TailwindCSS [16], забезпечивши адаптивний вигляд компоненту, приємну візуальну стилізацію та чистий валідний HTML-код без службових вставок. Завдяки ширшому контекстуальному розумінню завдання, GPT-4 зміг самостійно сформувати логіку поведінки акордеону навіть без додаткового уточнення з боку користувача.

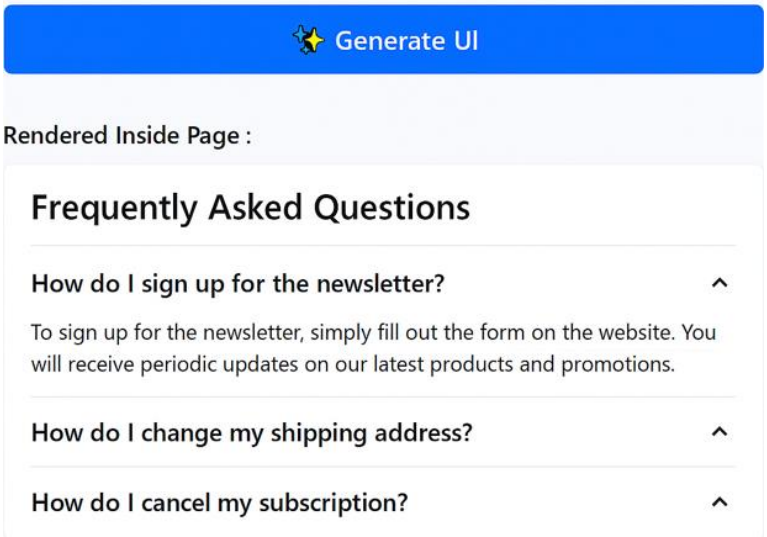


Рисунок 3.8 - Результат генерації FAQ (GPT-4)

На цьому етапі сформовано технічні рішення, що забезпечують роботу системи UI Crafter. Визначено роль кожного модуля в процесі генерації HTML-прототипів — від обробки запиту до фінальної візуалізації. Чіткий розподіл обов’язків між модулями дозволив побудувати зрозумілу та стабільну структуру системи без зайвих ускладнень [6, 14].

Запропонована архітектура підтримує роботу з двома джерелами генерації — локальною донавченою моделлю та зовнішнім GPT-4 — та зберігає гнучкість для подальшого розширення. Модульна побудова дає можливість розвивати окремі частини системи без змін її загальної логіки.

Реалізоване рішення створює основу для подальших експериментів: удосконалення генерації, розширення навчального корпусу та впровадження нових сценаріїв використання. Проведені експериментальні дослідження підтвердили ефективність побудованої архітектури та продемонстрували характерні відмінності в поведінці моделей при генерації UI-прототипів. Зокрема, GPT-4 показав високу гнучкість у створенні повноцінних багатосекційних макетів із глибокою стилізацією та інтерактивними компонентами [10, 13, 16], тоді як LoRA-модель забезпечила стабільну генерацію базових HTML-структур при чітко сформульованих запитах [2, 3, 4, 5]. Отримані результати створюють підґрунтя для подальшого вдосконалення моделей та розширення їхнього застосування в задачах автоматизованої генерації користувацьких інтерфейсів.

ВИСНОВКИ

У роботі реалізовано систему UI Crafter для автоматизованої генерації прототипів користувацьких інтерфейсів у HTML-форматі з використанням великих мовних моделей. Система підтримує комбіновану генерацію: через зовнішній API GPT-4 та локальну донавчену модель TinyLlama-1.1B із LoRA-адаптацією. Навчальний корпус сформовано на основі публічних та власноруч зібраних датасетів prompt-completion, що дало змогу підлаштувати модель до завдань генерації HTML з урахуванням синтаксису та стилів TailwindCSS.

Архітектура системи модульна: генерація, постобробка та візуалізація винесені в окремі компоненти. Ключовим є етап стабілізації коду: очищення службових вставок, корекція вкладеностей та уніфікація CSS-класів, що дозволяє отримувати валідний HTML, придатний до рендерингу у браузері.

У ході тестування встановлено: LoRA-модель забезпечує стабільну структуру коду, тоді як GPT-4 краще працює зі складними макетами при гнучких запитах. Обидва підходи продемонстрували стабільну роботу та доцільність комбінованого використання.

Перспективним є розширення навчального корпусу складнішими структурами, вдосконалення постобробки, інтеграція генерації стилів, а також розробка плагіну для Figma для прямої генерації інтерфейсів у середовищі дизайнерської розробки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

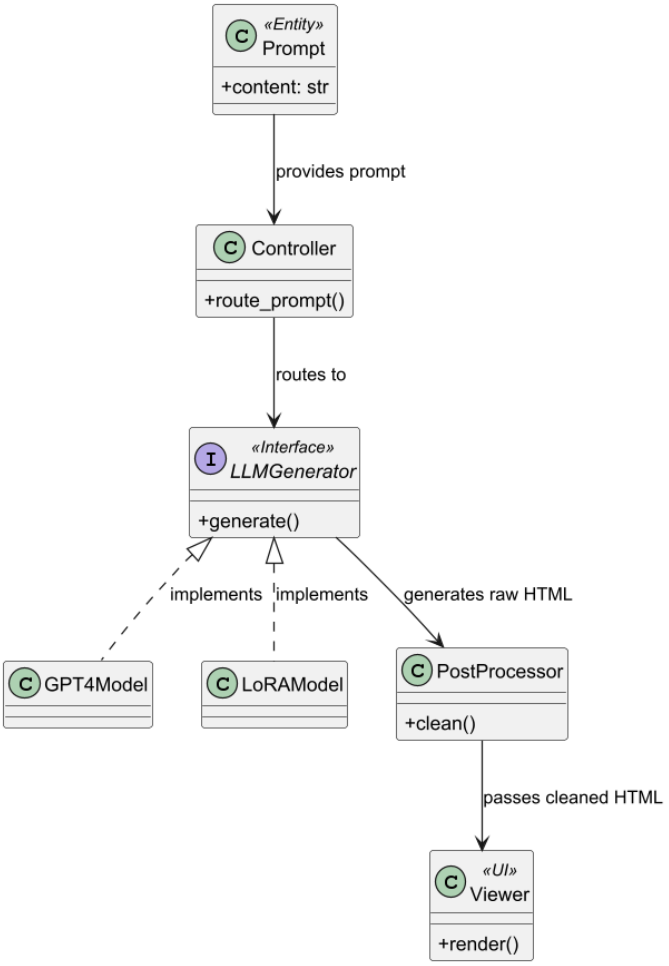
1. Vaswani A., Shazeer N., Parmar N. et al. Attention is all you need // Advances in Neural Information Processing Systems. — 2017. — Vol. 30. — P. 5998–6008.
2. Hu E., Shen Y., Wallis P. et al. LoRA: Low-Rank Adaptation of Large Language Models // arXiv preprint arXiv:2106.09685, 2021.
3. TinyLlama. TinyLlama-1.1B Chat v1.0. — Hugging Face, 2024. — URL: https://huggingface.co/TinyLlama/TinyLlama-1.1B-Chat-v1.0
4. Chen M., Xie X., Xu P., Liu S. WireGen: A Large-Scale Multi-Modal Dataset for Web UI Code Generation // arXiv preprint arXiv:2303.01728, 2023. — URL: https://arxiv.org/abs/2303.01728
5. Li L., Zhang C., Wang J. MultiUI: Multi-task UI Code Generation with Unified Representation // Proceedings of the 32nd ACM International Conference on Information and Knowledge Management (CIKM 2023). — 2023. — P. 4857–4866.
6. Hugging Face. PEFT: Parameter Efficient Fine-Tuning Documentation. — URL: https://huggingface.co/docs/peft/
7. Paszke A., Gross S., Massa F. et al. PyTorch: An imperative style, high-performance deep learning library // Advances in Neural Information Processing Systems. — 2019. — Vol. 32. — P. 8026–8037.
8. Tunstall L., von Werra L., Wolf T. Natural Language Processing with Transformers. Revised Edition. — Sebastopol: O’Reilly Media, 2022. — 478 p. — ISBN 978-1-098-13679-6.
9. NVIDIA CUDA Toolkit Documentation. — URL: https://developer.nvidia.com/cuda-toolkit
10. Brown T.B. et al. Language Models are Few-Shot Learners // NeurIPS 2020. — Vol. 33. — P. 1877–1901.
11. Kochetov M. et al. HTML Data Cleaning Techniques for LLM-Generated Content // ACM Conference on Web Science. — 2022.

12. Microsoft Research. TableBank: A Benchmark Dataset for Table Detection and Recognition // arXiv preprint arXiv:1903.01949, 2019. — URL: https://arxiv.org/abs/1903.01949
13. OpenAI. GPT-4 Technical Report. — OpenAI, 2023. — URL: https://openai.com/research/gpt-4
14. Hugging Face. Transformers Documentation. — URL: https://huggingface.co/docs/transformers/
15. Flask Documentation. Flask Web Development, 2023. — URL: https://flask.palletsprojects.com/
16. Bootstrap 5.3 Documentation. — URL: https://getbootstrap.com/docs/5.3/getting-started/introduction/
17. Python Software Foundation. Python 3.10 Documentation. — URL: https://docs.python.org/3.10/

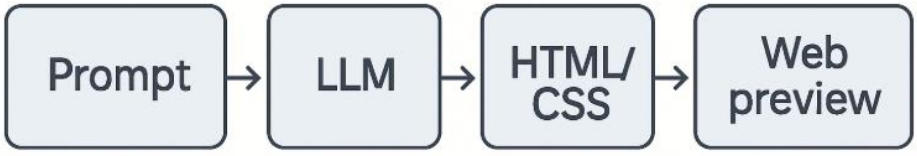
ДОДАТКИ

ДОДАТОК А

UML-діаграма структури інформаційного потоку системи UI Crafter



Архітектура генерації інтерфейсу за допомогою LLM



ДОДАТОК Б

Програмний код App.py

```
from flask import Flask, request, render_template
from llm_gpt import generate_html_with_gpt4
from llm_lora import generate_html_with_lora

app = Flask(__name__)


@app.route("/", methods=["GET", "POST"])
def index():
    html = ""
    prompt = ""
    model_source = "gpt"

    if request.method == "POST":
        prompt = request.form.get("prompt", "").strip()
        model_source = request.form.get("model_source", "gpt")

        if prompt:
            if model_source == "gpt":
                html = generate_html_with_gpt4(prompt)
            else:
                html = generate_html_with_lora(prompt)

    return render_template("index.html", prompt=prompt, html=html,
model_source=model_source)


if __name__ == "__main__":
    app.run(debug=True)
```

Програмний код train.py

```
from transformers import AutoTokenizer, AutoModelForCausalLM, TrainingArguments,
Trainer
from peft import LoraConfig, get_peft_model, prepare_model_for_kbit_training,
PeftModel
from datasets import load_dataset
import torch

base_model_id = "TinyLlama/TinyLlama-1.1B-Chat-v1.0"
lora_model_path = "./tinyllama_wiregen_lora"
new_data_path = "wirefigma_extended.jsonl"

tokenizer = AutoTokenizer.from_pretrained(base_model_id)
tokenizer.pad_token = tokenizer.eos_token

base_model = AutoModelForCausalLM.from_pretrained(
    base_model_id,
    torch_dtype=torch.float16,
    device_map="auto"
)

model = PeftModel.from_pretrained(base_model, lora_model_path)
model = get_peft_model(model, LoraConfig(
    r=8,
    lora_alpha=32,
```

```
    target_modules=["q_proj", "k_proj", "v_proj", "o_proj"],
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM"
))


model.print_trainable_parameters()

dataset = load_dataset("json", data_files=new_data_path, split="train")


def format_example(example):
    return {
        "text": f"### Instruction:\n{example['prompt']}\n\n###
Response:\n{example['completion']}"
    }


formatted_dataset = dataset.map(format_example)


def tokenize(example):
    encodings = tokenizer(
        example["text"],
        truncation=True,
        padding="max_length",
        max_length=512,
        return_tensors="pt"
    )
    encodings = {k: v.squeeze(0) for k, v in encodings.items()}
    encodings["labels"] = encodings["input_ids"].clone()
    return encodings


tokenized_dataset = formatted_dataset.map(tokenize)

training_args = TrainingArguments(
    output_dir="./tinyllama_lora_continued",
    per_device_train_batch_size=2,
    num_train_epochs=5,
    learning_rate=2e-4,
    fp16=True,
    logging_steps=10,
    save_steps=50,
    save_total_limit=1,
    report_to="none"
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_dataset,
    tokenizer=tokenizer
)
model.print_trainable_parameters()

trainer.train()

model.save_pretrained("tinyllama_wiregen_lora")
tokenizer.save_pretrained("tinyllama_wiregen_lora")
```

Програмний код llm_lora.py

```
from transformers import pipeline, AutoTokenizer, AutoModelForCausalLM
from peft import PeftModel
import torch
from html_utils import full_html_cleanup

MODEL_DIR = "./tinyllama_wiregen_lora"

_tokenizer = AutoTokenizer.from_pretrained(MODEL_DIR)

base_model = AutoModelForCausalLM.from_pretrained(
    "TinyLlama/TinyLlama-1.1B-Chat-v1.0",
    torch_dtype=torch.float16,
    device_map="auto"
)

_model = PeftModel.from_pretrained(base_model, MODEL_DIR)

_pipe = pipeline("text-generation", model=_model, tokenizer=_tokenizer)


def generate_html_with_lora(prompt: str) -> str:
    full_prompt = f"""### Instruction:
{prompt.strip()}

Write only HTML and CSS. Do not include lorem ipsum. Do not explain anything.
### Response:"""

    output = _pipe(full_prompt, max_new_tokens=768, do_sample=False,
temperature=0.0)[0]["generated_text"]

    if "### Response:" in output:
        raw_html = output.split("### Response:")[-1]
    elif "### Instruction:" in output:
        raw_html = output.split("### Instruction:")[-1]
    else:
        raw_html = output

    return full_html_cleanup(raw_html)
```

Програмний код llm_gpt.py

```
from openai import OpenAI
import os
from html_utils import full_html_cleanup
from dotenv import load_dotenv
load_dotenv()

client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))


def generate_html_with_gpt4(prompt: str) -> str:
    system_instruction = """
    You are an expert UI developer.
    You will receive a prompt describing the desired interface.
    Return only HTML+CSS code, without <html>, <head> or <body> tags.
    Output must begin with <div>, <style>, or <section>.
    """
```

```
    response = client.chat.completions.create(
        model="gpt-4",
        messages=[
            {"role": "system", "content": system_instruction},
            {"role": "user", "content": prompt}
        ],
        temperature=0.5,
        max_tokens=4096
    )

    raw = response.choices[0].message.content.strip()
    cleaned = full_html_cleanup(raw)
    return cleaned
```

Програмний код html_utils.py

```
import re
from bs4 import BeautifulSoup


def strip_markdown_blocks(text: str) -> str:
    text = re.sub(r"```html\s*", "", text, flags=re.IGNORECASE)
    text = re.sub(r"```", "", text)
    return text.strip()


def remove_html_head_blocks(text: str) -> str:
    text = re.sub(r"<html.*?>", "", text, flags=re.IGNORECASE | re.DOTALL)
    text = re.sub(r"</html>", "", text, flags=re.IGNORECASE)
    text = re.sub(r"<head>.*?</head>", "", text, flags=re.IGNORECASE | re.DOTALL)
    return text.strip()


def remove_leading_commentary(text: str) -> str:
    match =
re.search(r"<(div|style|form|section|table|ul|ol|input|button|textarea|header|main|bo
dy)[\s>]", text)
    return text[match.start():].strip() if match else text


def rewrite_conflicting_classes(text: str) -> str:
    text = text.replace(".container", ".generated-container")
    text = text.replace('class="container"', 'class="generated-container"')
    text = text.replace("body {", ".generated-body {")
    text = text.replace("<body", '<div class="generated-body"')
    text = text.replace("</body>", "</div>")
    return text


def remove_lorem_ipsum(text: str) -> str:
    return re.sub(r'(lorem ipsum|sed euismod)[^<]{50,}', '', text,
flags=re.IGNORECASE)


def cut_after_html(text: str) -> str:
    match = re.search(r'(</html>|</div>)', text, flags=re.IGNORECASE)
    if match:
        end_pos = match.end()
        return text[:end_pos]
    return text
```

```
def clean_html_with_bs4(text: str) -> str:
    cleaned = text.strip()
    cleaned = re.sub(r'<link[^>]+stylesheet[^>]+>', '', cleaned)
    cleaned = cleaned.replace("<!DOCTYPE html>", "").strip()

    if "<html" not in cleaned and "<body" in cleaned:
        cleaned = "<html>" + cleaned[cleaned.index("<body>"):] + "</html>"

    soup = BeautifulSoup(cleaned, "html5lib")
    return str(soup)


def full_html_cleanup(raw: str, model_type: str = "lora") -> str:

    text = strip_markdown_blocks(raw)

    if model_type == "lora":
        text = remove_html_head_blocks(text)
    elif model_type == "gpt":
        pass

    text = remove_leading_commentary(text)
    text = rewrite_conflicting_classes(text)
    text = remove_lorem_ipsum(text)
    text = cut_after_html(text)
    return text
```