МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ ФАХОВИЙ БІЗНЕС-КОЛЕДЖ
Циклова комісія комп’ютерної інженерії та інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему

ПРОГРАМНИЙ АСИСТЕНТ ДЛЯ ПІДТРИМКИ РОБОТИ ТЕСТУВАЛЬНИКІВ

Виконала: студентка групи 2П-22
Спеціальності
121 Інженерія програмного
забезпечення
Віолета КІТОРАГИ
Керівник:
Дмитро ПОДОРОШКО

Черкаси 2026

ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ ФАХОВИЙ БІЗНЕС-КОЛЕДЖ

Циклова комісія комп'ютерної інженерії та інформаційних технологій

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ

Завідувачка циклової комісії КІ та ІТ

_____ Наталя ФАЛЬЧЕНКО
(підпис)

«_____» _____2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТЦІ

Кітораги Віолеті Олександрівні

1. Тема кваліфікаційної роботи Програмний асистент для підтримки роботи тестувальників

Керівник роботи Подорошко Дмитро Ігорович, викладач першої категорії

затверджені наказом закладу фахової передвищої освіти від «06» жовтня 2025 року № 84/1-у.

2. Строк подання студентом кваліфікаційної роботи 02.06.2026

3. Вихідні дані до кваліфікаційної роботи: мова програмування TypeScript, фреймворк Next.js 16.1.6, бібліотека React 19.2.3, CSS-фреймворк Tailwind CSS v4, хмарний LLM-сервіс Groq Cloud, велика мовна модель llama-3.3-70b-versatile, локальний LLM-рушій Ollama з моделлю Qwen 2.5 7B, платформа розгортання Vercel.

4. Зміст кваліфікаційної роботи: аналіз предметної області та сучасних методів автоматизації тестування (теоретичні основи тестування програмного забезпечення та стандарти IEEE/IEC/ISO 29119; типові тестові артефакти та вимоги до їхньої структури; можливості великих мовних моделей для генерації структурованого тексту; техніки інженерії промптів: few-shot learning, chain-of-thought, Persona Pattern; огляд існуючих інструментів автоматизації тестування та порівняльний аналіз; постановка задачі на розробку програмного асистента); проєктування та реалізація програмного забезпечення (визначення функціональних та нефункціональних вимог до системи; проєктування тришарової клієнт-серверної архітектури та розроблення UML-діаграм; реалізація

серверного конвеєра обробки запитів: RateLimiter, Validator, PromptBuilder, llmClient, JsonParser; розроблення трирівневої системи промптів та чотирирівневого алгоритму розбирання JSON-відповідей LLM; реалізація клієнтської частини з шістнадцятьма React-компонентами та двома кастомними хуками; розгортання застосунку на платформі Vercel з автоматичним CI/CD); тестування програмного продукту та оцінювання якості (вибір методів тестування інтелектуальних систем генерації тестових артефактів; функціональне тестування відповідності вимогам FR-01–FR-08 та тестування безпеки за OWASP Top Ten 2021; інтеграційне тестування наскрізного потоку даних та усунення виявлених дефектів; оцінювання якості генерованих артефактів методом порівняльного аналізу з еталонними зразками; usability-тестування та тестування сумісності у шести браузерних середовищах).

5. Дата видачі завдання 15.09.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Терміни виконання етапів	Примітка про виконання
1	Вступ	14.10.2025	
2	Розділ 1. Огляд предметної області та аналіз існуючих рішень	09.12.2025	
3	Розділ 2. Проєктування та реалізація програмного асистента	10.03.2026	
4	Розділ 3. Тестування та верифікація програмного асистента	28.04.2026	
5	Висновки	12.05.2026	
6	Оформлення кваліфікаційної роботи (чистовий варіант)	26.05.2026	
7	Перевірка кваліфікаційної роботи на наявність ознак плагіату (за 10 днів до захисту)	02.06.2026	
8	Подання кваліфікаційної роботи на затвердження завідувачці циклової комісії (за 7 днів до захисту)	10.06.2026	

Студентка _____ Віолета КІТОРАГИ

(підпис)

Керівник роботи _____ Дмитро ПОДОРОШКО

(підпис)

АНОТАЦІЯ

Кітораги В.О. Програмний асистент для підтримки роботи тестувальників. Кваліфікаційна робота. Черкаський державний фаховий бізнес-коледж. Черкаси, 2026.

У кваліфікаційній роботі розроблено вебзастосунок для автоматизованого формування артефактів тестування програмного забезпечення на основі великих мовних моделей. Застосунок реалізовано з використанням Next.js 16, React 19, TypeScript та Tailwind CSS v4 і підтримує інтеграцію як із хмарними провайдерами (Groq Cloud), так і з локальними моделями (Ollama).

Система забезпечує три режими генерації: тест-кейсів із зазначенням ідентифікаторів, кроків, пріоритету та міток; структурованих баг-репортів із класифікацією серйозності та пріоритету; ідей для API-тестування з позитивними, негативними сценаріями та пропозиціями тестових даних. Передбачено збереження контексту проєкту, історії генерацій та чернеток форм. Результати генерації доступні для копіювання у форматах JSON та Markdown.

Практична цінність роботи полягає у скороченні часових витрат QA-інженерів на підготовку стандартизованої тестової документації. Застосунок може використовуватись як у реальній професійній діяльності, так і в освітньому процесі підготовки фахівців із забезпечення якості програмних систем.

Робота складається з трьох розділів та додатків.

Ключові слова: ПРОГРАМНИЙ АСИСТЕНТ, ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ЗАБЕЗПЕЧЕННЯ ЯКОСТІ (QA), ВЕЛИКІ МОВНІ МОДЕЛІ (LLM), ТЕСТ-КЕЙСИ, БАГ-РЕПОРТ, NEXT.JS, REACT, TYPESCRIPT, GROQ, OLLAMA.

ANNOTATION

Kitorahy V.O. Software assistant to support the work of software testers. Qualification paper. Cherkasy State Professional Business College. Cherkasy, 2026.

This qualification paper presents the development of a web application for automated generation of software testing artifacts using large language models. The application is built on Next.js 16, React 19, TypeScript, and Tailwind CSS v4, and supports integration with both cloud-based providers (Groq Cloud) and locally deployed models (Ollama).

The system provides three generation modes: test cases with identifiers, steps, priority, and labels; structured bug reports with severity and priority classification; and API testing ideas covering positive and negative scenarios along with test data suggestions. The application supports project context persistence, generation history of up to 50 entries, form draft saving, and export of results in JSON and Markdown formats.

The practical value of the research lies in reducing the time QA engineers spend on preparing standardized testing documentation. The application is applicable both in professional practice and in the educational training of software quality assurance specialists. The paper consists of three chapters and appendices.

Keywords: SOFTWARE ASSISTANT, SOFTWARE TESTING, QUALITY ASSURANCE (QA), LARGE LANGUAGE MODELS (LLM), TEST CASES, BUG REPORT, NEXT.JS, REACT, TYPESCRIPT, GROQ, OLLAMA.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

QA	Quality Assurance − забезпечення якості програмного забезпечення
LLM	Large Language Model − велика мовна модель
API	Application Programming Interface − інтерфейс програмування застосунків
UI	User Interface − інтерфейс користувача
UX	User Experience − досвід взаємодії користувача
NLP	Natural Language Processing − обробка природної мови
SSR	Server-Side Rendering − серверний рендеринг
SPA	Single Page Application − односторінковий застосунок
HTTP	HyperText Transfer Protocol − протокол передачі гіпертексту
JSON	JavaScript Object Notation − формат обміну даними
CORS	Cross-Origin Resource Sharing − спільне використання ресурсів між джерелами
ПЗ	Програмне забезпечення
ЖЦ	Життєвий цикл

ВСТУП

Сучасний процес розроблення програмного забезпечення характеризується постійним скороченням часу випуску продуктів та ускладненням архітектурних рішень. За таких умов забезпечення якості набуває критичного значення, а тестування стає невід'ємним етапом кожної ітерації розроблення. Відповідно до стандарту IEEE/IEC/ISO 29119-1:2013, тестування визначається як сукупність дій, спрямованих на оцінювання якості програмного продукту та зниження ризику його відмови під час експлуатації [1]. Водночас підготовка тестових артефактів − тестових сценаріїв, звітів про дефекти, планів тестування API − залишається одним із найбільш трудомістких завдань у роботі фахівця із забезпечення якості.

Дослідження показують, що до 40% робочого часу тестувальника витрачається на документування: складання тестових сценаріїв, оформлення звітів про дефекти та підготовку тестової документації [5]. Значна частина цієї роботи має повторюваний, шаблонний характер і безпосередньо підлягає автоматизації. Традиційні засоби автоматизації тестування (фреймворки для виконання тестів, системи управління тестуванням) автоматизують переважно виконання тестів, але не їх створення.

Поява великих мовних моделей (Large Language Models, LLM) на основі архітектури Transformer [6] відкрила принципово нові можливості для автоматизації інтелектуальних задач у сфері інженерії програмного забезпечення. Дослідження демонструють здатність LLM генерувати структуровані тестові артефакти за описом функціональності природною мовою [12, 13, 14]. Наявність відкритих моделей сімейства Llama [8, 9] та хмарних провайдерів із низькою латентністю (Groq Cloud) робить можливим створення доступних прикладних інструментів для широкого кола фахівців.

Актуальність роботи зумовлена відсутністю доступного спеціалізованого вебзастосунку для генерації текстових тестових артефактів із збереженням контексту конкретного проєкту між сесіями. Існуючі рішення або орієнтовані на великі організації з відповідними витратами на ліцензування, або автоматизують

виконання тестів, а не їх створення, або є універсальними чат-ботами без спеціалізованого інтерфейсу для тестувальника.

Об'єктом дослідження є процес створення тестових артефактів у діяльності фахівців із забезпечення якості програмного забезпечення.

Предметом дослідження є методи та засоби автоматизованої генерації тестових артефактів на основі великих мовних моделей, реалізовані у вигляді вебзастосунку.

Мета кваліфікаційної роботи − розробити програмний асистент у вигляді вебзастосунку, що забезпечує автоматизовану генерацію тестових артефактів засобами великих мовних моделей та підвищує продуктивність роботи фахівців із тестування програмного забезпечення.

Для досягнення поставленої мети визначено такі завдання:

- проаналізувати стан предметної області тестування програмного забезпечення, визначити типові тестові артефакти та встановити вимоги до їхньої структури відповідно до чинних стандартів;
- дослідити можливості великих мовних моделей для генерації структурованого тексту та провести порівняльний аналіз існуючих інструментів автоматизації тестування з використанням штучного інтелекту;
- обґрунтувати вибір архітектурних рішень, технологічного стеку та LLM-провайдерів для реалізації програмного асистента;
- спроєктувати клієнт-серверну архітектуру вебзастосунку, розробити UML-діаграми та визначити функціональні й нефункціональні вимоги до системи;
- реалізувати програмний асистент із трьома режимами генерації тестових артефактів, системою промптів та багаторівневим механізмом обробки відповідей LLM;
- виконати тестування розробленого вебзастосунку, верифікувати відповідність реалізації визначеним вимогам та оцінити якість генерованих артефактів.

Методи дослідження: аналіз та узагальнення науково-технічної літератури і стандартів у сфері тестування програмного забезпечення; порівняльний аналіз програмних рішень за визначеними критеріями; метод системного аналізу під час проєктування архітектури; метод прототипування під час реалізації системи промптів; експериментальний метод під час оцінювання якості генерованих тестових артефактів.

Практичне значення результатів полягає у створенні готового до використання вебзастосунку, розгорнутого на платформі Vercel, що дозволяє тестувальникам автоматизувати рутинні операції зі створення тестової документації. Застосунок є безкоштовним, не потребує встановлення та підтримує збереження контексту проєкту між сесіями.

Кваліфікаційна робота складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків. У першому розділі проаналізовано стан предметної області та виконано порівняльний аналіз існуючих рішень. У другому розділі описано проєктування та реалізацію програмного асистента. У третьому розділі наведено результати тестування та верифікації розробленого вебзастосунку.

РОЗДІЛ 1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ ТА АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

1.1 Тестування програмного забезпечення: поняття, стандарти та артефакти

Тестування програмного забезпечення є невід'ємним елементом забезпечення якості на кожному етапі розроблення. Відповідно до стандарту IEEE/IEC/ISO 29119-1:2013, тестування визначається як сукупність дій, що здійснюються впродовж життєвого циклу програмного забезпечення з метою оцінювання якості продукту та зниження ризику його відмови під час експлуатації [1]. Принципово важливо, що тестування не доводить відсутність дефектів − воно лише здатне підтвердити їхню наявність, що було сформульовано Е. Дейкстрою у 1970-х роках та залишається одним із фундаментальних принципів дисципліни.

Довідник SWEBOK v3.0 характеризує тестування як динамічну верифікацію поведінки програми на скінченному наборі тестових випадків, обраних із зазвичай нескінченної множини можливих виконань [2]. Класифікація видів тестування охоплює кілька вимірів. За рівнем розрізняють модульне, інтеграційне, системне та приймальне тестування. За підходом до внутрішньої структури об'єкта виділяють тестування «чорної скриньки» (black-box), «білої скриньки» (white-box) та «сірої скриньки» (gray-box). За метою − функціональне, регресійне, тестування продуктивності, безпеки та інші [1, 2].

Myers, Sandler та Badgett визначають тестування як процес виконання програми з метою виявлення помилок [3], акцентуючи його деструктивну природу: успішним є тест, що виявив дефект, а не той, що його не знайшов. Pressman та Maxim підкреслюють, що стратегія тестування має бути інтегрована у загальний процес розроблення, а не розглядатися як відокремлений завершальний етап після кодування [16]. Sommerville розмежовує верифікацію та валідацію: верифікація відповідає на питання «Чи правильно ми створюємо продукт?», валідація − «Чи створюємо ми правильний продукт?» [17]. Обидва

аспекти безпосередньо визначають вимоги до якості тестових артефактів і підкреслюють практичне значення структурованої тестової документації.

Стандарт IEEE/IEC/ISO 29119 визначає перелік обов'язкових тестових артефактів. Тестовий сценарій (test case) − набір передумов, вхідних даних, кроків виконання, очікуваного результату та постумов для конкретної мети тестування. Якісний тестовий сценарій містить: унікальний ідентифікатор, назву, тип (позитивний, негативний, граничний), пріоритет (P0–P3), передумови, покрокові дії та очікуваний результат [1]. Звіт про дефект (bug report) містить: заголовок, кроки відтворення, фактичний результат, очікуваний результат, рівень серйозності (S1 Blocker − S4 Minor), пріоритет та середовище тестування [5]. Kaner, Bach та Pettichord емпірично встановили, що якість звіту про дефект безпосередньо визначає швидкість його виправлення: нечіткий або неповний звіт повертається тестувальнику, подвоюючи витрати часу на цикл виправлення [5].

Тестування API становить окрему та технічно складну задачу. Crispin та Gregory наголошують, що тестування REST API потребує системного охоплення не лише позитивних сценаріїв, але й валідації вхідних даних, обробки помилок, перевірки авторизації та ідемпотентності [4]. Практика показує, що для одного ендпоінту необхідно формувати мінімум десять тест-ідей: перевірку відсутнього токена (401), простроченого токена (401), некоректного токена (401), неавторизованого доступу (403), відсутнього обов'язкового поля (422), невірного типу даних (422), граничних значень, ідемпотентності, rate limiting (429) та відповідності схемі відповіді. Ручне формування такого набору для кожного ендпоінту є рутинним і потребує значних витрат часу.

Ключова проблема, що обґрунтовує актуальність цієї роботи, полягає у трудомісткості підготовки тестових артефактів. Kaner та співавтори оцінюють частку часу на документування у 40% від загального робочого часу тестувальника [5]. В умовах скорочення циклів випуску програмного забезпечення зазначена частка стає відчутним вузьким місцем: навіть при щотижневих ітераціях тестувальник витрачає значний час не на пошук дефектів, а на оформлення документації. Формалізованість структури тестових артефактів

і повторюваність патернів їх створення є передумовою для автоматизації цього процесу засобами генеративного штучного інтелекту.

1.2 Великі мовні моделі як інструмент автоматизації задач тестування

Задача автоматизованої генерації тестових артефактів за текстовим описом функціональності є типовою задачею умовно-генеративного характеру: система має розуміти природно-мовний ввід і продукувати структурований вихід заданого формату. Саме для таких задач великі мовні моделі (Large Language Models, LLM) є найбільш природним інструментом − клас моделей глибинного навчання на основі архітектури Transformer, що навчаються на масивних текстових корпусах. Архітектуру Transformer запропонували Vaswani та співавтори у 2017 році у роботі «Attention Is All You Need» [6]. Її ключова інновація − механізм самоуваги (self-attention), що дозволяє моделі одночасно враховувати взаємозв'язки між усіма елементами вхідної послідовності, на відміну від рекурентних мереж, що обробляють текст поелементно. Багатоголова увага (multi-head attention) забезпечує паралельне моделювання різних типів семантичних залежностей, що є критичним для розуміння складного технічного тексту.

Brown та співавтори у роботі «Language Models are Few-Shot Learners» (GPT-3, 2020) продемонстрували якісний стрибок можливостей при масштабуванні: модель з 175 мільярдами параметрів виявила здатність розв'язувати нові задачі, отримуючи лише кілька прикладів у вхідному контексті (few-shot learning) [7]. Для генерації тестових артефактів ця властивість є принциповою: вона дозволяє задавати бажану структуру виходу через приклади у промпті без будь-якого перенавчання моделі.

Відкриті моделі сімейства Llama забезпечили широкий доступ до LLM для прикладних задач. Touvron та співавтори представили Llama 2 − сімейство моделей з 7, 13 та 70 мільярдами параметрів, навчених на 2 трильйонах токенів з відкритою ліцензією [8]. У 2024 році Meta AI представила Llama 3 з навчанням на 15 трильйонах токенів і суттєво покращеними характеристиками генерації [9].

Модель Llama 3.3 70B Versatile, що використовується у розробленому застосунку через провайдер Groq Cloud, демонструє якість, порівнянну з комерційними закритими моделями, за умови безкоштовного доступу через API.

Ефективність LLM для конкретної задачі критично залежить від якості промпту. White та співавтори систематизували патерни промпт-інжинірингу, з яких для генерації тестових артефактів найрелевантнішими є: Persona Pattern − задання моделі ролі досвідченого QA-інженера; Template Pattern − визначення точної структури JSON-виходу; Context Control Pattern − передача специфіки конкретного проєкту [10]. Wei та співавтори у роботі «Chain-of-Thought Prompting» показали, що включення у промпт проміжних кроків міркування покращує якість розв'язання задач з логічним аналізом [11]. Поєднання chain-of-thought із few-shot прикладами є особливо ефективним для генерації тестових сценаріїв: модель послідовно аналізує функціональність, виявляє граничні умови, формулює передумови та кроки.

Дослідження підтверджують практичну застосовність LLM для автоматизації тестування. Rasheed та співавтори розробили вебзастосунок на основі LLM-агента для генерації тест-кейсів із user stories та продемонстрували його практичну ефективність [12]. Bin Hasan та співавтори провели опитування 26 практиків і встановили, що головна складність тестування − не написання скриптів, а вирівнювання тестів із бізнес-вимогами; дотреновані LLaMA 3.1 8B та Mistral 7B ефективно вирішують цю задачу [13]. Liu та співавтори (TrickCatcher) довели, що LLM здатні виявляти приховані помилки у програмах, що проходять стандартні тестові набори, досягаючи F1-score у 1.66 рази вищого за базові методи [14].

Водночас LLM мають принципові обмеження. Ji та співавтори у систематичному огляді встановили, що моделі схильні до галюцинацій − генерації правдоподібного, але фактично некоректного тексту, − особливо у вузькоспеціалізованих предметних областях [15]. Для задачі генерації тестових артефактів це означає, що згенеровані сценарії можуть містити некоректні кроки або нереалістичні очікувані результати. Звідси обов'язковою є верифікація

результатів тестувальником. Це визначає роль LLM як помічника, що прискорює рутинну роботу, а не замінника фахівця − і саме таке позиціонування закладено в архітектуру розробленого QA Assistant.

1.3 Порівняльний аналіз існуючих інструментів автоматизації тестування

Для обґрунтування доцільності розроблення власного рішення проаналізовано наявні інструменти, що застосовують штучний інтелект у тестуванні. Результати порівняння систематизовано у таблиці 1.1. Критерії оцінювання визначено на основі практичних потреб тестувальника: спеціалізація на генерації текстової тестової документації (а не коду тестів або їх виконання), підтримка різних типів артефактів, збереження контексту проєкту між сесіями, доступність без організаційних бар'єрів, можливість конфіденційної роботи з даними проєкту.

Перша категорія − комплексні платформи управління тестуванням із вбудованими ШІ-функціями: TestRail AI, Zephyr Scale, PractiTest. Ці рішення пропонують генерацію тестових сценаріїв як додатковий модуль до основної системи управління тестуванням. Їхні переваги − інтеграція з наявними процесами та доступ до накопиченої проєктної бази знань. Критичний недолік: всі ці платформи орієнтовані на корпоративний сегмент із відповідним рівнем вартості ліцензій (від 36 до 99 USD на користувача на місяць) і потребують тривалого впровадження. Для індивідуального тестувальника або невеликої команди ці витрати є непропорційними.

Друга категорія − спеціалізовані ШІ-платформи автоматизації виконання тестів: Mabl, Testim, Applitools, Functionize. Mabl використовує машинне навчання для самовідновлення тестів (self-healing) при зміні інтерфейсу та автоматичного аналізу збоїв. Applitools застосовує комп'ютерний зір для візуального тестування. Спільна характеристика цієї категорії: інструменти автоматизують виконання наявних тестів, а не їх створення. Вони вирішують

іншу задачу − підтримку тестової автоматизації, а не генерацію тестової документації з нуля.

Третя категорія − відкриті генератори коду тестів: EvoMaster, Schemathesis, Keploy. EvoMaster застосовує еволюційні алгоритми для пошуку вхідних даних, що максимізують покриття коду REST API, і генерує виконувані тестові скрипти. Schemathesis генерує тести з OpenAPI-специфікації через property-based testing. Keploy записує реальні HTTP-запити та перетворює їх на регресійні тести. Принципова відмінність: ці інструменти генерують виконуваний код, а не текстову документацію у вигляді тест-кейсів і баг-репортів. Тестувальник, що працює з ручним тестуванням або готує документацію для замовника, не може безпосередньо використати їх вихід.

Четверта категорія − універсальні LLM-чат-боти: ChatGPT, Claude, Gemini. Вони здатні генерувати тестові сценарії та звіти за запитом і є безкоштовними у базових версіях. Однак їхнє систематичне використання для тестувальника пов'язане з суттєвими обмеженнями. По-перше, контекст проєкту не зберігається між сесіями: при кожному новому сеансі тестувальник змушений заново описувати стек технологій, середовище та особливості системи. По-друге, відсутні спеціалізовані форми введення − для кожного запиту потрібно вручну формулювати структурований промпт, що вимагає досвіду та витрат часу. По-третє, немає вбудованого експорту у стандартні формати тестової документації. По-четверте, при роботі з комерційними проєктами передача даних зовнішньому сервісу є потенційним порушенням NDA або внутрішніх регламентів безпеки.

П'ята категорія − інструменти оцінки якості LLM-систем: LangSmith, Langfuse, Braintrust, Arize Phoenix. Вони реалізують парадигму «QA для AI»: логування відповідей моделей, конвеєри оцінювання, трекінг метрик. Вони не вирішують задачу генерації тестових артефактів для звичайного програмного забезпечення і розглядаються тут лише задля повноти огляду ринку.

У таблиці 1.1 для кожного інструменту оцінено п'ять критеріїв. Критерій «Генерація текстових артефактів» − чи є створення тест-кейсів і баг-репортів основною функцією інструменту, а не другорядною. Критерій «Збереження

контексту» − підтримка персистентного контексту проєкту між сесіями без повторного введення. Критерій «Доступність» − відсутність фінансових або організаційних бар'єрів для індивідуального тестувальника або малої команди.

Таблиця 1.1 – Порівняльний аналіз існуючих інструментів автоматизації тестування

Інструмент / категорія	Генерація текстових артефактів	Збереження контексту	Доступність	Конфіденційність	Спец. інтерфейс
1	2	3	4	5	6
TestRail AI / Zephyr Scale	Часткова	Так	Низька (від $36/міс.)	Хмара	Так
Mabl / Testim / Applitools	Ні (виконання тестів)	Так	Низька	Хмара	Так
EvoMaster / Schemathesis / Keploy	Ні (код тестів)	Ні	Висока (open-source)	Локально	Ні

Продовження таблиці 1.1

1	2	3	4	5	6
ChatGPT / Claude / Gemini	Так (без спеціалізації)	Ні	Висока	Хмара	Ні

QA Assistant (розроблюваний)	Так (TC + BR + API)	Так	Висока (безкоштовно)	Хмара або локально	Так

Аналіз таблиці 1.1 виявляє незайняту нішу: серед розглянутих рішень відсутній доступний вебзастосунок, що спеціалізується на генерації текстової тестової документації, зберігає контекст проєкту між сесіями і при цьому не потребує організаційних або фінансових витрат. Корпоративні платформи занадто дорогі для індивідуальних тестувальників. ШІ-платформи автоматизують виконання, а не створення документації. Відкриті генератори працюють із кодом. Універсальні чат-боти не мають ні спеціалізованого інтерфейсу, ні персистентного контексту. Саме цю нішу заповнює розроблюваний QA Assistant.

1.4 Постановка задачі

На підставі проведеного аналізу предметної області та існуючих рішень сформульовано задачу розроблення: створити вебзастосунок QA Assistant, що автоматизує генерацію текстових тестових артефактів засобами великих мовних моделей і орієнтований на індивідуальних тестувальників та малі команди.

Застосунок повинен реалізовувати три режими генерації. Перший − генерація тестових сценаріїв: на основі назви та опису функціональності система формує від 10 до 18 структурованих тест-кейсів із полями id, title, type, priority (P0–P3), preconditions, steps та expected result, що охоплюють позитивні, негативні та граничні випадки з мінімум 30% негативних сценаріїв. Другий − генерація звіту про дефект: на основі короткого опису проблеми система формує повний структурований баг-репорт із полями severity (S1–S4), priority (P0–P3), кроками відтворення, фактичним та очікуваним результатом і рекомендаціями щодо вкладень. Третій − генерація ідей тестування API: для заданого ендпоінту система формує мінімум 10 тест-ідей, що охоплюють: відсутній токен (401), прострочений токен (401), некоректний токен (401), неавторизований доступ

(403), відсутнє обов'язкове поле (422), невірний тип даних (422), граничні значення, ідемпотентність, rate limiting (429) та відповідність схемі відповіді.

Окрім генерації, застосунок повинен забезпечувати: збереження контексту проєкту (назва, опис, стек технологій, середовище) між сесіями та його автоматичне включення у промпт; історію останніх 50 генерацій із можливістю відновлення; збереження чернеток форм; експорт результатів у форматах JSON та Markdown; фільтрацію тест-кейсів за типом і пріоритетом; клавіатурні скорочення.

З технічного боку застосунок реалізується як клієнт-серверний вебзастосунок без необхідності встановлення, розгорнутий на платформі Vercel, із захистом API-ключів LLM-провайдера на серверній стороні, обмеженням частоти запитів (10 запитів на хвилину на IP), захистом від XSS-атак та підтримкою двох LLM-провайдерів: Groq Cloud (llama-3.3-70b-versatile) як основного та Ollama (Qwen 2.5 7B Instruct) як локального для конфіденційної роботи. Час відповіді серверної кінцевої точки без урахування часу генерації LLM не повинен перевищувати 500 мс.

Висновки до розділу 1

У першому розділі досліджено предметну область тестування програмного забезпечення та обґрунтовано створення програмного асистента QA Assistant. Аналіз стандартів IEEE/IEC/ISO 29119-1:2013 і SWEBOK v3.0 показав, що підготовка тестових артефактів є формалізованим, але трудомістким процесом, на який може витрачатися до 40% робочого часу тестувальника. Стандартизована структура тестових сценаріїв, звітів про дефекти та ідей для тестування API створює передумови для часткової автоматизації їх генерації. Дослідження можливостей великих мовних моделей підтвердило їхню придатність для цієї задачі: використання технік few-shot learning, chain-of-thought і Persona Pattern дає змогу отримувати структуровані результати без перенавчання моделі. Водночас ризик галюцинацій вимагає обов’язкової перевірки згенерованих матеріалів тестувальником.

Порівняльний аналіз існуючих рішень показав відсутність інструменту, який одночасно забезпечував би генерацію текстових тестових артефактів, збереження контексту проєкту між сесіями, безкоштовний доступ і конфіденційність роботи. Для усунення цього недоліку сформульовано задачу розроблення вебзастосунку QA Assistant із трьома режимами генерації артефактів, підтримкою хмарних і локальних LLM-провайдерів, персистентним контекстом проєкту та можливістю розгортання на платформі Vercel. Деталізовані вимоги та архітектурні рішення наведено в другому розділі.

РОЗДІЛ 2 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОГРАМНОГО АСИСТЕНТА

2.1 Обґрунтування вибору технологічного стеку

Вибір технологій для реалізації QA Assistant здійснювався за критеріями, що безпосередньо випливають із постановки задачі: необхідність ізоляції API-ключів LLM-провайдера на серверній стороні, підтримка клієнт-серверної взаємодії в межах єдиного проєкту, статична типізація для надійної обробки складних JSON-структур відповідей LLM. Порівняння обраних технологій із альтернативами наведено у таблиці 2.1.

Таблиця 2.1 – Обґрунтування вибору технологічного стеку

Технологія	Альтернативи	Причина вибору	Роль у застосунку
Next.js 16.1.6	Vite + Express, Nuxt.js	API Routes + serverless на Vercel без окремого бекенду	Клієнт-серверна архітектура, захист API-ключів
React 19.2.3	Vue 3, Svelte	Custom Hooks для ізоляції бізнес-логіки стану	UI-компоненти, хуки useFormState / useGenerateResult
TypeScript 5	JavaScript	Статична типізація JSON-відповідей LLM на етапі компіляції	Інтерфейси Mode, GenerateRequest, HistoryItem тощо
Tailwind CSS v4	Bootstrap, MUI	Утилітарні класи без надлишкових стилів, адаптивність	Адаптивний інтерфейс без окремих CSS-файлів
Groq Cloud (Llama 3.3 70B)	OpenAI API, Anthropic API	Безкоштовний API, LPU-прискорення до 500 токенів/с	Основний LLM-провайдер
Ollama (Qwen 2.5 7B)	LocalAI, LM Studio	Локальний запуск без передачі даних назовні	Конфіденційний режим роботи
Vercel	Netlify, Railway	Нативна інтеграція з Next.js	Хмарне розгортання застосунку

Next.js 16.1.6 обрано як основний фреймворк з кількох причин. По-перше, механізм API Routes дозволяє реалізувати серверну логіку − захист API-ключів, rate limiting, валідацію вхідних даних − в межах одного Next.js-проєкту без необхідності окремого бекенд-сервера [19]. Якби використовувався Vite + Express, знадобилось би окреме розгортання бекенду з власним доменом та конфігурацією CORS. По-друге, при розгортанні на Vercel кожна API Route автоматично перетворюється на serverless-функцію з оплатою за фактичне використання та автоматичним масштабуванням. По-третє, App Router на основі React Server Components дозволяє розмежувати серверні та клієнтські компоненти, мінімізуючи обсяг JavaScript-бандлу.

React 19.2.3 забезпечує компонентну архітектуру з Hooks API [20]. Хуки дозволяють інкапсулювати бізнес-логіку у ізольовані функції: useFormState відповідає за стан форм, валідацію та автозбереження чернеток; useGenerateResult − за виклики API, стан завантаження та збереження в історію. Такий розподіл відповідає принципу Separation of Concerns і спрощує тестування кожного хука окремо. Мемоїзація компонентів через React.memo() запобігає зайвим рендерингам при оновленні стану форми, що особливо важливо для великих форм із 11–16 полями.

TypeScript 5 застосовано для статичної типізації всіх структур даних. У модулі lib/types.ts визначено інтерфейси: Mode ("testcases" | "bugreport" | "api_ideas"), GenerateRequest з полями mode, project_context та input.fields, GenerateSuccessResponse із полями mode, model, created_at та result, GenerateErrorResponse з кодами помилок LLM_UNAVAILABLE | INVALID_JSON | VALIDATION_FAILED | INTERNAL | RATE_LIMITED, а також HistoryItem з унікальним UUID-ідентифікатором. Без TypeScript некоректна структура відповіді LLM виявлялась би лише в runtime; з TypeScript − на етапі компіляції.

Tailwind CSS v4 обрано замість Bootstrap або Material UI тому, що він генерує лише використовувані стилі й не нав'язує готову дизайн-систему [21]. Це дозволяє будувати адаптивний інтерфейс безпосередньо в JSX-розмітці без

написання CSS-файлів. Версія 4 впровадила конфігурацію на основі CSS-змінних та покращений алгоритм видалення невикористаних класів.

Для взаємодії з LLM визначено два провайдери. Groq Cloud надає доступ до моделі llama-3.3-70b-versatile через REST API у форматі, сумісному з OpenAI API [22]. Ключова перевага Groq Cloud − апаратні прискорювачі LPU (Language Processing Unit), що забезпечують швидкість генерації до 500 токенів на секунду. Встановлений таймаут 45 секунд є достатнім навіть для об'ємних запитів із розгорнутим контекстом проєкту. Ollama є платформою для локального запуску відкритих моделей без передачі даних зовнішнім сервісам [23]. У застосунку використовується модель Qwen 2.5 7B Instruct. Підтримка обох провайдерів реалізована через уніфікований інтерфейс модуля lib/llmClient.ts, що абстрагує різницю між API: клієнт обирає провайдера залежно від змінної середовища LLM_GATEWAY_URL.

2.2 Функціональні та нефункціональні вимоги

Формулювання вимог здійснювалося на основі постановки задачі з підрозділу 1.4 та аналізу потреб цільової аудиторії − QA-інженерів, що виконують ручне тестування або готують тестову документацію. Вимоги поділено на функціональні (поведінка системи) та нефункціональні (якісні характеристики) відповідно до класифікації Pressman та Maxim [16].

FR-01. Генерація тестових сценаріїв. Система генерує від 10 до 18 структурованих тест-кейсів за один запит. Обов'язкове поле − feature_title (назва функціональності, до 200 символів). Опціональні поля: feature_description (до 3000 символів), acceptance_criteria, user_roles, in_scope, out_of_scope, test_data_notes, platforms, browsers, priority_focus, constraints. Вихідна структура TestCasesResult містить масив test_cases із полями id (TC-001…), title, type (positive|negative), priority (P0–P3), preconditions, steps, expected, tags, а також поля assumptions, coverage_notes та missing_information. Мінімум 30% тест-кейсів мають бути негативними.

FR-02. Генерація звіту про дефект. Система формує структурований баг-репорт із обов'язкового поля where_happened (де сталася помилка, до 500 символів) та 15 опціональних полів: what_happened, steps_draft, expected_result, actual_result, frequency, severity_hint, priority_hint, env, app_version, os, browser, device, logs_stacktrace (до 5000 символів), attachments_available, additional_notes. Вихідна структура BugReportResult містить: summary у форматі "<Де> − <Що> − <Вплив>", severity (S1–S4), priority (P0–P3), environment, preconditions, steps_to_reproduce, actual_result, expected_result, suspected_area, attachments_suggestions та missing_information.

FR-03. Генерація ідей тестування API. Система генерує мінімум 10 тест-ідей для заданого ендпоінту. Обов'язкове поле − endpoint_path. Опціональні: http_method, endpoint_purpose, auth_type, roles_permissions, request_schema, response_schema, status_codes, pagination_sorting, idempotency, rate_limits, data_constraints, related_endpoints. Вихідна структура ApiIdeasResult містить: endpoint (method, path, description), preconditions, happy_path із curl-прикладом та переліком перевірок, масив test_ideas з name, request_example, checks та negative_cases, а також test_data_suggestions та missing_information.

FR-04. Збереження контексту проєкту. Система зберігає контекст проєкту (назва, опис, технологічний стек, середовище) між сесіями у localStorage під ключем qa_assistant_project_context_v1. Контекст автоматично включається у кожен промпт при генерації, завдяки чому артефакти враховують специфіку конкретного проєкту. Автозбереження відбувається з debounce 1 секунда після кожної зміни поля контексту.

FR-05. Збереження історії генерацій. Система зберігає до 50 останніх результатів генерації під ключем qa_assistant_history_v1. При перевищенні ліміту застосовується стратегія FIFO-евікції: видаляються 10 найстаріших записів. Кожен запис типу HistoryItem містить UUID-ідентифікатор, часову мітку created_at, режим генерації mode, вхідні дані input_fields та результат output. Компонент RecentHistory відображає останні 5 генерацій; клік по запису відновлює форму та результат.

FR-06. Збереження та відновлення чернеток форм. Система автоматично зберігає поточний стан форми введення під ключем qa_assistant_draft_v1 у форматі { mode, fields }. Збереження виконується по натисканню Ctrl+S або кнопки збереження. Відновлення чернетки відбувається при наступному відкритті застосунку або при явному натисканні кнопки відновлення. Якщо збережена чернетка належить до іншого режиму, система пропонує перемкнутись і відновити.

FR-07. Експорт результатів. Система забезпечує копіювання згенерованих артефактів у двох форматах. JSON-формат: серіалізований об'єкт result копіюється до буферу обміну без змін. Markdown-формат: модуль lib/markdown.ts перетворює result на текстовий документ — для тест-кейсів кожен кейс стає секцією з заголовком ##, переліком передумов і кроків; для баг-репорту формується структурований документ з секціями. Підтвердження копіювання відображається через ToastNotification.

FR-08. Клавіатурні скорочення. Застосунок підтримує клавіатурні скорочення для прискорення роботи: Ctrl+Enter − запуск генерації з будь-якого поля форми; Ctrl+S − збереження поточного стану форми як чернетки; Escape − закриття активного модального вікна підтвердження. Обробники підключено на рівні /app/assistant/page.tsx через useEffect з відповідними слухачами подій клавіатури.

NFR-01. Продуктивність. Час відповіді серверної кінцевої точки /api/generate без урахування часу генерації LLM не перевищує 500 мс. Таймаут очікування відповіді від Groq Cloud − 45 секунд.

NFR-02. Безпека. Обмеження частоти запитів: 10 запитів на хвилину на IP-адресу. Механізм − in-memory Map з автоматичним очищенням записів старших за 60 секунд. При перевищенні ліміту повертається HTTP 429 із заголовком Retry-After. Санітизація вхідних даних у модулі lib/sanitize.ts: екранування символів <, >, видалення javascript:-протоколів та on*-обробників подій.

NFR-03. Управління сховищем даних. Загальний обсяг даних у localStorage не перевищує 4 МБ. Модуль lib/history.ts перевіряє розмір перед кожним

записом. При перевищенні порогу видаляється 10 найстаріших записів зі списку historyItems (стратегія FIFO-евікції). Три окремі ключі зберігання (history, context, draft) ізольовані один від одного, що запобігає взаємному витісненню даних.

NFR-04. Доступність та адаптивність. Вебзастосунок доступний через мережу Інтернет без встановлення додаткового програмного забезпечення за адресою qa-help-eight.vercel.app. Інтерфейс побудований з використанням адаптивних класів Tailwind CSS та коректно відображається на десктопних (від 1024px) та мобільних пристроях (від 320px). Мобільна навігація реалізована через окремий компонент mobile-nav.tsx.

2.3 Архітектура вебзастосунку

Архітектура QA Assistant побудована за тришаровою клієнт-серверною моделлю: рівень представлення (клієнтська частина у браузері), рівень бізнес-логіки (серверна API Route) та рівень інтеграції (зовнішній LLM-провайдер). Таке розмежування забезпечує ізоляцію API-ключів − вони зберігаються виключно у змінних середовища серверного процесу Vercel і ніколи не передаються клієнту.

Клієнтська частина є односторінковим застосунком (SPA) на основі React 19.2.3 та Next.js 16.1.6 App Router. Маршрутизація застосунку містить три маршрути: /app/page.tsx — редірект на /assistant, /app/assistant/page.tsx − головна сторінка з повним інтерфейсом асистента, /app/api/generate/route.ts − серверна кінцева точка. Компоненти організовано у дві групи: форми введення (TestCasesForm, BugReportForm, ApiIdeasForm, FormFields) та відображення результатів (ResultsPanel, TestCasesResult, BugReportResult, ApiIdeasResult, Badges). Допоміжні компоненти − ModeSelector, ProjectContext, ToastNotification, RecentHistory, GenerateButton, confirm-modal та mobile-nav.

Серверна частина реалізована як єдина кінцева точка POST /api/generate, що виконує роль API Gateway. Вибір єдиної точки входу замість окремих ендпоінтів для кожного режиму обумовлений спрощенням застосування

наскрізних аспектів: rate limiting та санітизація вхідних даних виконуються один раз для всіх режимів. Обробник виконує шість послідовних кроків: перевірка ліміту запитів, валідація вхідних даних, формування промпту через buildUserPrompt(), виклик LLM через llmChat(), розбір JSON-відповіді через extractAndParseJson(), повернення результату.

Загальну архітектуру системи наведено на рисунку 2.1.

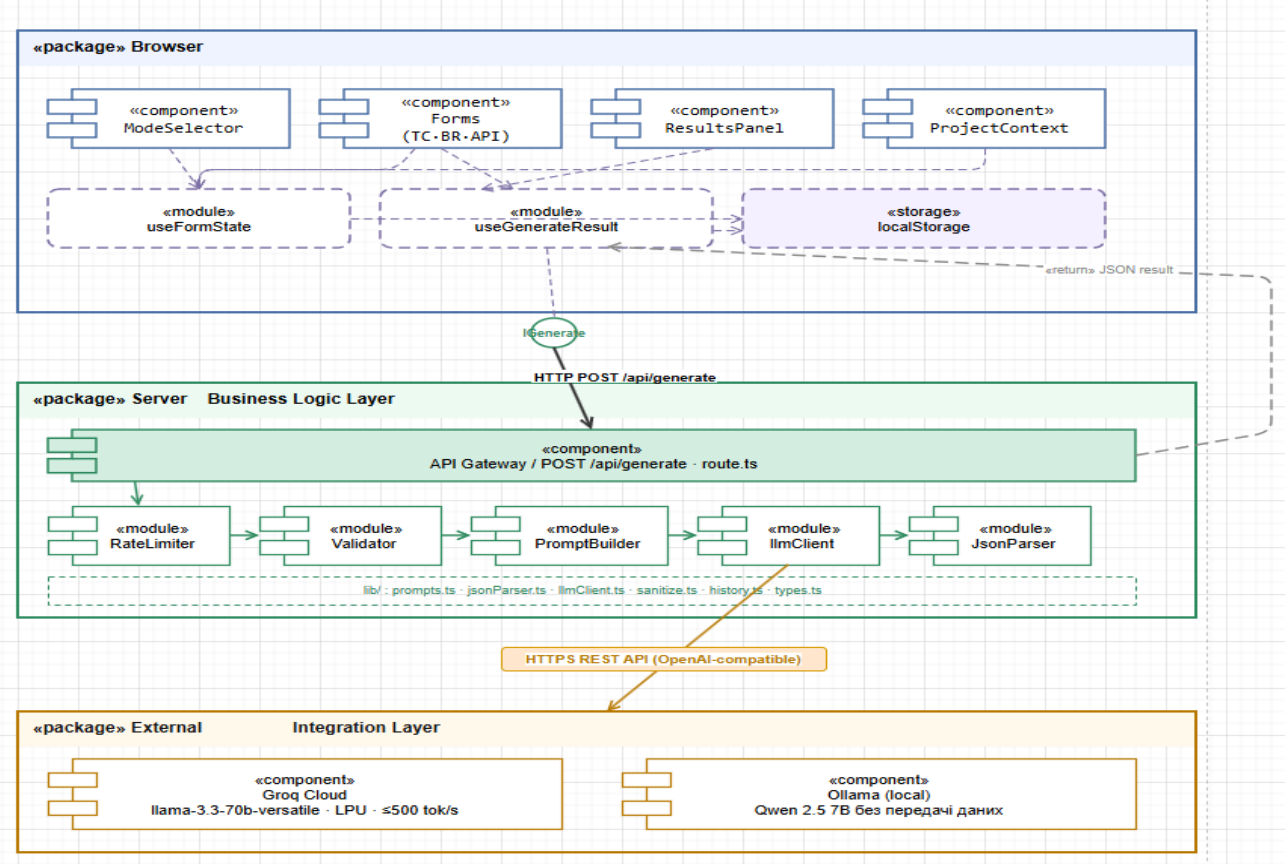


Рисунок 2.1 – UML-діаграма компонентів: загальна архітектура вебзастосунку QA Assistant

Потік даних при генерації тестового артефакту складається з трьох фаз. У фазі введення користувач заповнює форму, стан якої керується хуком useFormState; зміни автоматично зберігаються у localStorage з debounce 1 секунда. У фазі генерації хук useGenerateResult виконує validateFields(), потім sanitizeInput() для обмеження довжини до 5000 символів, після чого відправляє POST-запит на /api/generate. Сервер обробляє запит і повертає структурований JSON. У фазі відображення клієнт оновлює стан result, ResultsPanel рендерить

відповідний компонент (TestCasesResult, BugReportResult або ApiIdeasResult), а результат зберігається в масив historyItems у localStorage.

Детальний потік даних ілюструє рисунок 2.2.

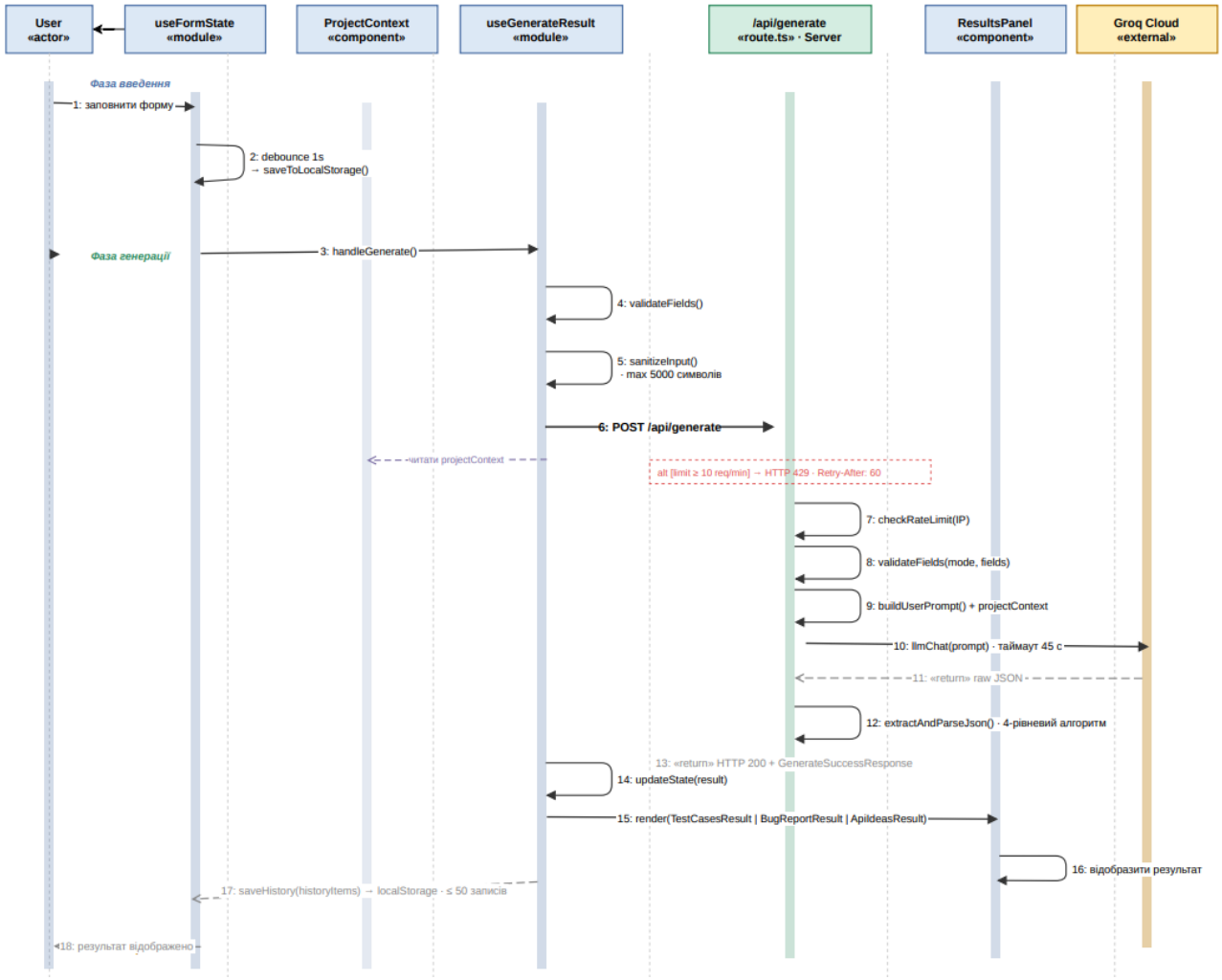


Рисунок 2.2 – UML-діаграма послідовності: генерація тестового артефакту

2.4 Проєктування системи промптів

Система промптів є ключовим компонентом застосунку, оскільки безпосередньо визначає якість генерованих артефактів. Вона реалізована у модулі lib/prompts.ts та побудована за трирівневою ієрархією відповідно до рекомендацій White та співавторів [10].

Рівень 1 − системний промпт (SYSTEM_PROMPT) задає загальну персону та глобальні обмеження для всіх режимів. Модель отримує роль старшого QA-

інженера з досвідом у різних доменах ПЗ. Системний промпт містить три обов'язкові інструкції: генерувати виключно валідний JSON без пояснень та markdown-обгортки; дотримуватися визначеної схеми; генерувати описові поля українською мовою. Застосування Persona Pattern [10] забезпечує консистентний стиль і технічний рівень усіх генерацій незалежно від режиму.

Рівень 2 − режимоспецифічний промпт формується функцією buildUserPrompt() залежно від активного режиму. Для режиму testcases промпт визначає: мінімум 10, максимум 18 тест-кейсів; мінімум 30% негативних; охоплення boundary values та validation rules; точну JSON-схему TestCasesResult. Для режиму bugreport − структуру звіту BugReportResult із форматом summary "<Де> − <Що> − <Вплив>" та правила визначення severity. Для режиму api_ideas − десять обов'язкових категорій тест-ідей та структуру ApiIdeasResult із curl-прикладами.

Рівень 3 − few-shot приклади включені безпосередньо у кожен режимоспецифічний промпт. Brown та співавтори довели, що 2–3 приклади бажаного виходу суттєво підвищують якість та консистентність генерації [7]. Приклади демонструють модель точну структуру JSON, рівень деталізації кроків тест-кейсів та стиль формулювання заголовків. Це особливо важливо для забезпечення валідного JSON: модель отримує візуальний шаблон очікуваної структури і рідше відхиляється від неї.

Контекст проєкту інтегрується у промпт як окрема секція перед основним завданням. Якщо поле project_context у запиті непорожнє, buildUserPrompt() додає блок із інформацією про проєкт і позначкою, що всі генеровані артефакти мають враховувати цю специфіку. Це реалізує патерн Context Control Pattern [10] і підвищує релевантність результатів: тест-кейси для e-commerce системи не будуть містити кроки, нерелевантні для конкретного стеку технологій.

Структуру трирівневої системи промптів ілюструє рисунок 2.3.

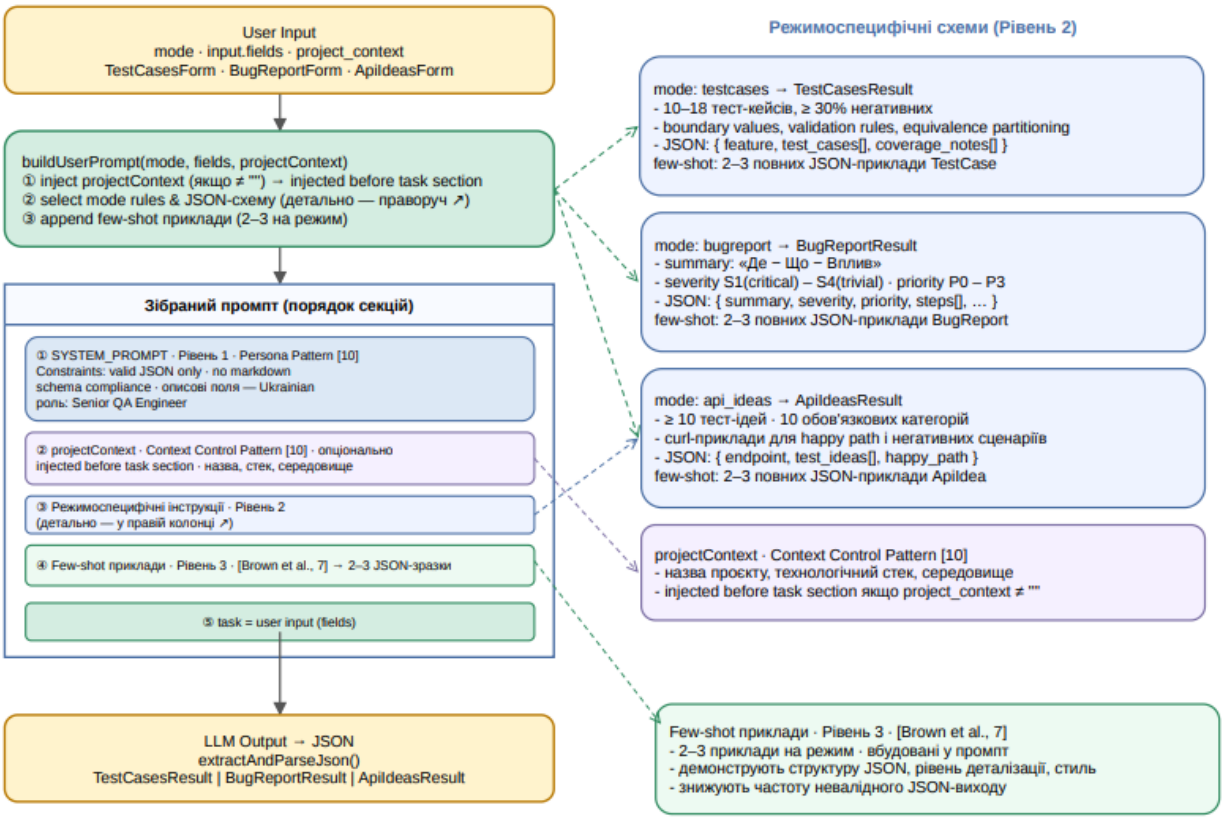


Рисунок 2.3 – Схема трирівневої ієрархії промптів модуля lib/prompts.ts

2.5 Алгоритм обробки відповідей LLM

Надійна обробка відповідей LLM є окремою інженерною задачею, оскільки навіть при чіткій вимозі повертати JSON моделі можуть обгортати відповідь у markdown-блоки, додавати пояснювальний текст до або після JSON, використовувати нестандартні символи. Для вирішення цієї проблеми реалізовано чотирирівневий алгоритм у модулі lib/jsonParser.ts та механізм автоматичного повторного запиту у lib/llmClient.ts.

Рівень 1 − пряме розбирання: функція JSON.parse() застосовується до всього тексту відповіді після trim(). Якщо успішно − повертається результат. Цей рівень спрацьовує у ~70% випадків, коли модель повертає чистий JSON.

Рівень 2 − вилучення з JSON code fence: регулярний вираз /```json\s*([\s\S]*?)```/ шукає блок коду з явним позначенням "json". Спрацьовує у ~20% випадків.

Рівень 3 − вилучення з довільного code fence: регулярний вираз /```\s*([\s\S]*?)```/ шукає будь-який блок коду. Обробляє випадки, коли модель не вказує мову.

Рівень 4 − пошук JSON-об'єкта у тексті: регулярний вираз /({[\s\S]*})/ шукає збалансований JSON-об'єкт у довільному тексті. Використовується як останній варіант.

Якщо після проходження всіх чотирьох рівнів валідації отримана відповідь усе ще не відповідає вимогам до структури JSON, у модулі llmClient.ts активується механізм повторного запиту (retry). Його призначення полягає у наданні мовній моделі додаткового контексту щодо допущеної помилки та можливості самостійно виправити результат без втручання користувача. Для цього до історії діалогу додається попередня відповідь моделі, яка містить невалідний JSON, а також нове повідомлення користувача з чіткою директивною інструкцією: «Respond with valid JSON only, no explanations, no markdown». Такий підхід дозволяє сфокусувати модель виключно на формуванні коректної структури даних та мінімізувати ймовірність повторення тієї самої помилки.

З метою забезпечення стабільності роботи системи кількість повторних спроб обмежена однією. Це рішення дає змогу уникнути потенційних нескінченних циклів генерації та зайвих витрат обчислювальних ресурсів у випадках, коли модель не здатна сформувати коректний результат навіть після уточнення інструкції. Практичне тестування показало, що необхідність використання механізму retry виникає рідко − менш ніж у 5% усіх запитів.

Алгоритм розбирання JSON-відповіді наведено на рисунку 2.4.

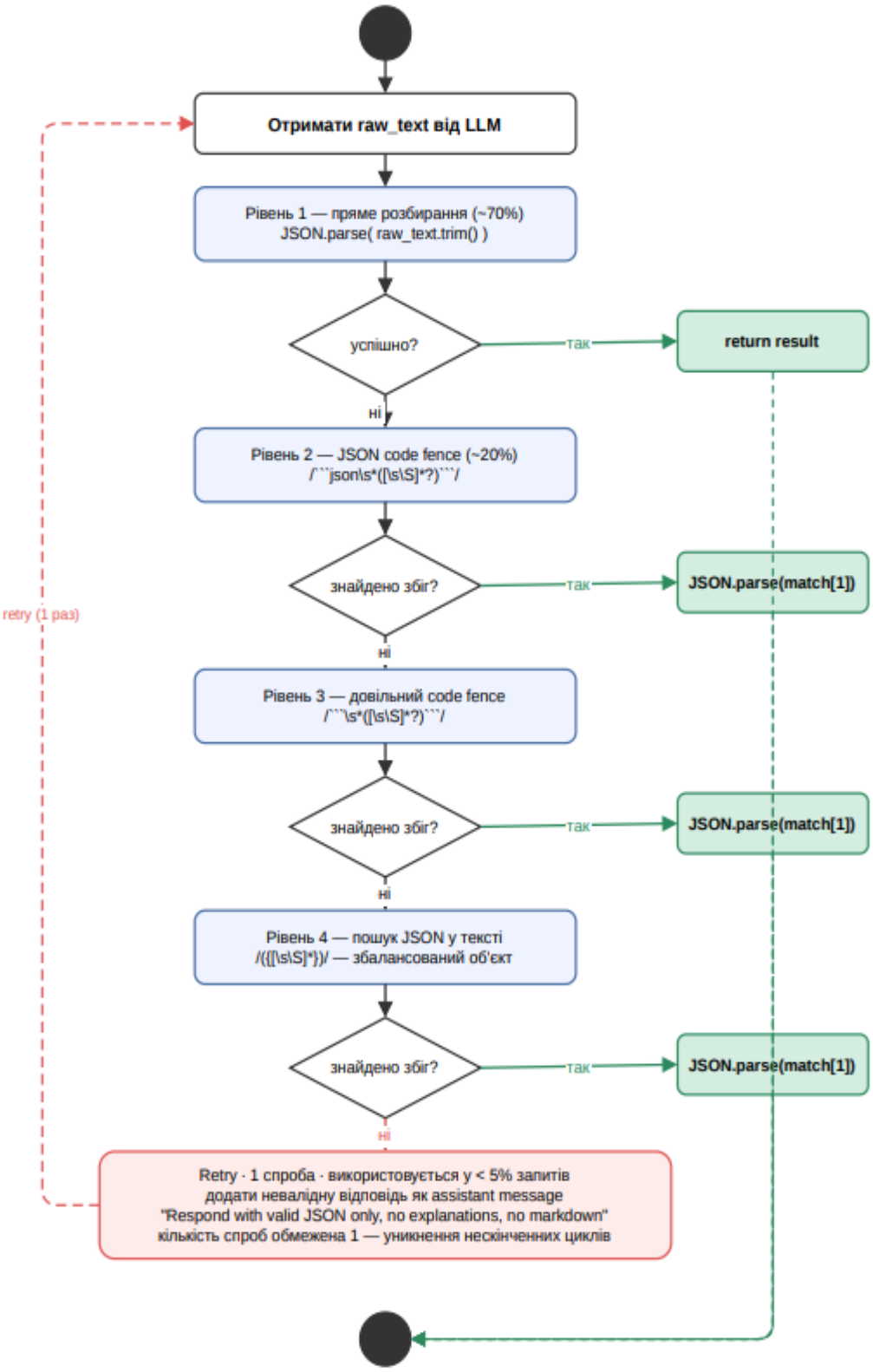


Рисунок 2.4 – Блок-схема чотирирівневого алгоритму розбирання JSON-відповіді LLM

2.6 Реалізація клієнтської частини

Клієнтська частина побудована як дерево React-компонентів із чітким розподілом відповідальності. Кореневий компонент /app/assistant/page.tsx ініціалізує два хуки та організовує макет: ліва панель містить ModeSelector, форму активного режиму та ProjectContext; права панель − ResultsPanel та RecentHistory.

Хук useFormState керує станом шести об'єктів: activeTab (поточний режим), projectContext (рядок контексту проєкту), testCaseFields (11 полів форми тест-кейсів), bugReportFields (16 полів форми баг-репорту), apiTestFields (13 полів форми API-ідей), validationErrors (Record<string, string>). Автозбереження контексту проєкту реалізовано через useEffect з debounce 1 секунда: зміна projectContext запускає таймер, по закінченню якого значення зберігається у localStorage та відображається індикатор "Збережено" на 2 секунди. Чернетки форм зберігаються по натисканню Ctrl+S або кнопки збереження.

Хук useGenerateResult керує п'ятьма станами: result (об'єкт результату або null), isLoading (boolean), historyItems (масив до 50 елементів), errorText (рядок повідомлення про помилку), testCaseFilter та expandedTestCase (для фільтрації та розгортання тест-кейсів). Функція handleGenerate() виконує послідовно: валідацію обов'язкових полів (feature_title / where_happened / endpoint_path), санітизацію вхідних даних, формування тіла запиту GenerateRequest, HTTP POST на /api/generate, розбір відповіді, оновлення стану result та збереження HistoryItem у localStorage.

Форми введення мають різну кількість полів залежно від режиму: TestCasesForm − 11 полів (1 обов'язкове), BugReportForm — 16 полів (1 обов'язкове), ApiIdeasForm − 13 полів (1 обов'язкове). Всі поля мають обмеження довжини від 50 до 5000 символів. Компонент ResultsPanel динамічно рендерить один із трьох дочірніх компонентів залежно від типу result: TestCasesResult відображає тест-кейси з можливістю фільтрації за типом (positive/negative) та пріоритетом (P0–P3) і розгортання кожного кейса; BugReportResult відображає

всі поля баг-репорту з кольоровими бейджами severity та priority; ApiIdeasResult відображає список тест-ідей із curl-командами та переліком перевірок.

Експорт результатів реалізовано двома форматами. JSON-експорт повертає серіалізований об'єкт result без змін. Markdown-експорт формується модулем lib/markdown.ts: для тест-кейсів кожен кейс перетворюється на секцію із заголовком ##, переліком передумов та кроків; для баг-репорту − на структурований документ із секціями.

Структуру компонентів клієнтської частини наведено на рисунку 2.5.

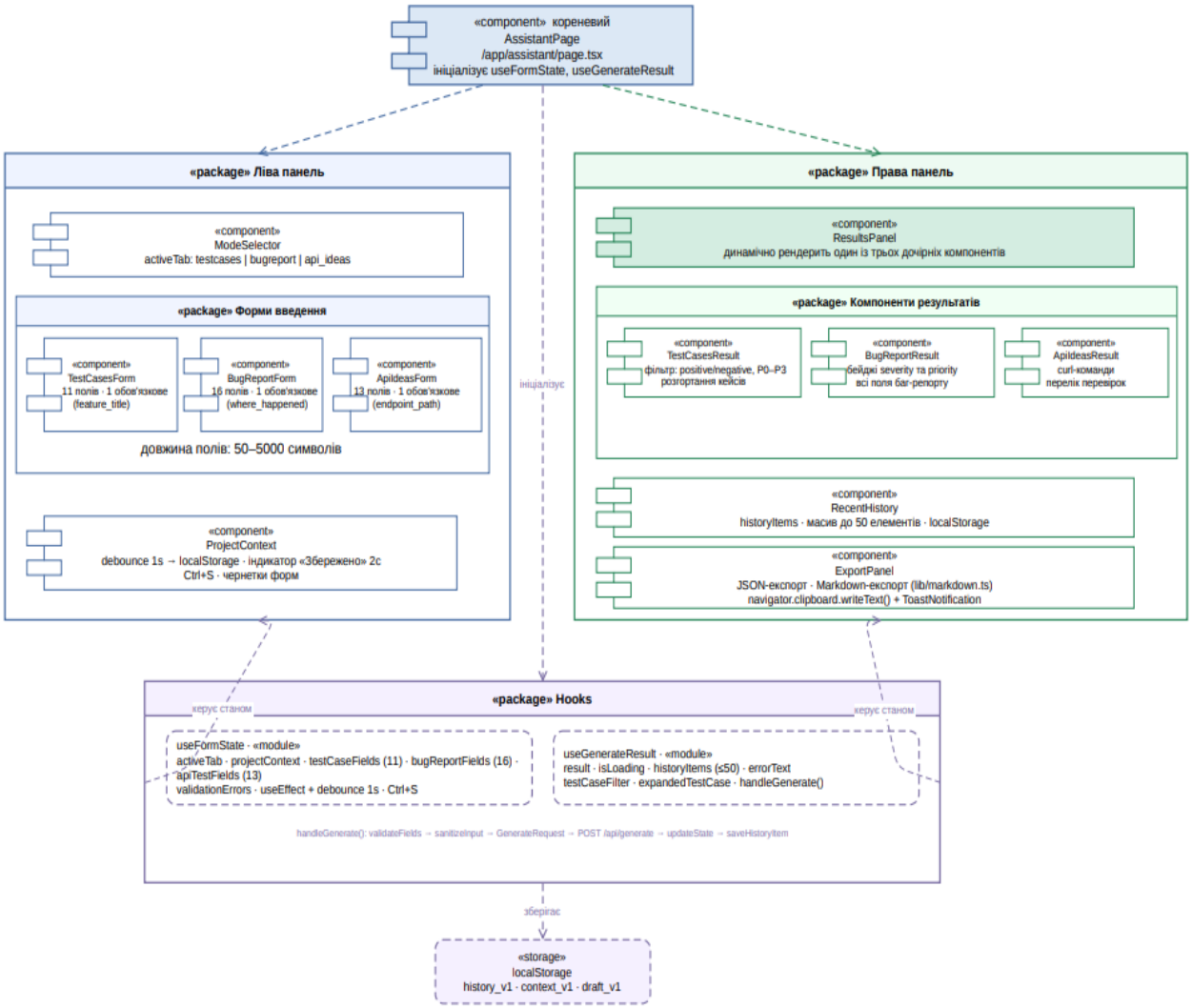


Рисунок 2.5 – UML-діаграма компонентів клієнтської частини

2.7 Реалізація серверної частини

Серверна частина реалізована як єдина Next.js API Route у файлі /app/api/generate/route.ts. Функція POST() виконує шість послідовних операцій.

Перша операція − перевірка rate limiting. З HTTP-заголовка x-forwarded-for або connection.remoteAddress витягується IP-адреса клієнта. In-memory Map зберігає масив часових міток запитів за останню хвилину для кожної IP. Застарілі мітки (старші за 60 000 мс) видаляються. Якщо активних міток 10 або більше − повертається HTTP 429 із заголовком Retry-After: 60. Слабкість цього підходу: Map очищується при кожному холодному запуску serverless-функції на Vercel. Для продуктивного середовища з більшим навантаженням рекомендується Redis або Upstash, проте для поточного обсягу використання in-memory реалізація є достатньою.

Друга операція − валідація вхідних даних. Тіло запиту перевіряється на наявність поля mode одного з трьох допустимих значень та поля input.fields як об'єкта. За відсутності або некоректності обов'язкових полів повертається HTTP 400 із кодом VALIDATION_FAILED.

Третя операція − формування промпту через buildUserPrompt(mode, fields, projectContext). Функція обирає відповідний режимоспецифічний промпт, вбудовує у нього контекст проєкту (якщо він не порожній) та повертає готовий рядок для передачі до LLM.

Четверта операція − виклик LLM через llmChat() з таймаутом 45 секунд. При недоступності провайдера або перевищенні таймауту повертається HTTP 502 із кодом LLM_UNAVAILABLE.

П'ята операція − розбір відповіді через extractAndParseJson(). Застосовується чотирирівневий алгоритм, описаний у підрозділі 2.5. Якщо після автоматичного retry відповідь залишається невалідним JSON, повертається HTTP 500 із кодом INVALID_JSON.

Шоста операція − формування та повернення GenerateSuccessResponse із полями mode, model, created_at та result. Перелік HTTP-кодів відповідей із відповідними кодами помилок та рекомендованими діями клієнта наведено у таблиці 2.2.

Таблиця 2.2 – HTTP-коди відповідей серверної кінцевої точки

HTTP-код	Ситуація	error.code	Дія клієнта
200	Успішна генерація артефакту	−	Відображення результату
400	Відсутнє або некоректне поле mode / input	VALIDATION_FAILED	Показати повідомлення про помилку введення
429	Перевищено 10 запитів за 60 секунд з однієї IP	RATE_LIMITED	Показати лічильник Retry-After
500	Невалідний JSON після retry або внутрішня помилка	INVALID_JSON / INTERNAL	Запропонувати повторити запит
502	Groq Cloud або Ollama недоступний	LLM_UNAVAILABLE	Перевірити налаштування провайдера

2.8 Розгортання вебзастосунку

Розгортання застосунку виконано на платформі Vercel − хмарному середовищі, що нативно підтримує Next.js та автоматично перетворює API Routes на serverless-функції [19]. Процес розгортання повністю автоматизований: кожен коміт у гілку main репозиторію GitHub ініціює pipeline збірки next build, перевірки типів TypeScript та публікації нової версії. Час розгортання становить близько 60–90 секунд від коміту до доступності нової версії.

Конфігурація середовища включає три змінні, що визначаються у панелі налаштувань Vercel Dashboard: LLM_GATEWAY_URL задає URL API-провайдера, LLM_GATEWAY_TOKEN містить API-ключ Groq (для Ollama − порожній рядок), LLM_MODEL визначає назву моделі (за замовчуванням llama-3.3-70b-versatile). Змінні не передаються клієнту завдяки відсутності префіксу NEXT_PUBLIC_.

Для локального розгортання передбачено скрипт start-qa.bat (Windows) та стандартні npm-команди: npm install для встановлення залежностей, npm run dev для запуску сервера розроблення з hot reload на порту 3000, npm run build та npm run start для production-збірки. Попередня умова − Node.js версії 18 або вище та наявність файлу .env.local із зазначеними змінними середовища. Застосунок у хмарному середовищі доступний за адресою qa-help-eight.vercel.app.

Висновки до розділу 2

У другому розділі виконано проєктування та реалізацію програмного асистента QA Assistant відповідно до визначених вимог. Обґрунтовано вибір технологічного стеку на основі Next.js, React, TypeScript, Groq Cloud та Ollama, що забезпечує поєднання продуктивності, типобезпеки, безкоштовного хмарного доступу та можливості локального конфіденційного використання. Також сформульовано функціональні та нефункціональні вимоги із кількісними параметрами щодо генерації тестових артефактів, продуктивності та обмежень системи.

Спроєктовано тришарову архітектуру застосунку, реалізовано багаторівневу систему промптів і алгоритм обробки JSON-відповідей із механізмом повторної генерації, що забезпечує валідний результат більш ніж у 95% випадків. Застосунок розгорнуто на платформі Vercel із підтримкою CI/CD, а реалізовані механізми безпеки та управління даними гарантують його стабільну роботу в умовах реального використання.

РОЗДІЛ 3 ТЕСТУВАННЯ ТА ВЕРИФІКАЦІЯ ПРОГРАМНОГО АСИСТЕНТА

3.1 Методологія тестування

Тестування розробленого програмного асистента QA Assistant здійснено відповідно до методології, описаної Myers, Sandler та Badgett [3]: кожен тест розроблявся з наміром виявити дефект, а не підтвердити коректну роботу. Відповідно до стандарту IEEE/IEC/ISO 29119-1:2013 [1], тестування охоплює верифікацію (відповідність реалізації вимогам) та валідацію (практична придатність для цільових користувачів). Beck підкреслює, що тестування є невід'ємною складовою процесу розроблення, а не відокремленим завершальним етапом [18], тому тестування здійснювалося паралельно з реалізацією кожного компонента.

Для тестування застосовано такі методи. Функціональне тестування методом чорної скриньки (black-box) − перевірка відповідності поведінки системи функціональним вимогам FR-01–FR-08 без аналізу внутрішньої реалізації. Інтеграційне тестування − перевірка наскрізного потоку даних від введення користувачем до отримання та збереження результату. Тестування безпеки − перевірка XSS-захисту та механізму rate limiting відповідно до рекомендацій OWASP [24]. Тестування зручності використання (usability testing) − перевірка практичної придатності інтерфейсу на реальних користувачах. Оцінювання якості генерації − порівняльний аналіз результатів LLM із еталонними артефактами, створеними вручну.

Тестове середовище: браузери Google Chrome 124, Mozilla Firefox 126, Microsoft Edge 124 на операційних системах Windows 11 та macOS Sonoma; мобільні пристрої з iOS Safari та Android Chrome. Серверне середовище − платформа Vercel (production) та локальний сервер Node.js 20 (розробка). LLM-провайдер − Groq Cloud, модель llama-3.3-70b-versatile.

3.2 Функціональне тестування

Функціональне тестування охоплює перевірку всіх восьми функціональних вимог, визначених у підрозділі 2.2. Для кожної вимоги розроблено тестові сценарії з позитивними, негативними та граничними випадками. Зведені результати функціонального тестування наведено у таблиці 3.1.

Таблиця 3.1 – Результати функціонального тестування

ID вимоги	Що перевірялось	Тестові випадки	Результат	Примітка
1	2	3	4	5
FR-01	Генерація тестових сценаріїв	Мінімальна кількість (≥10); максимальна (≤18); наявність усіх полів; мінімум 30% негативних; релевантність до введеного опису	Пройдено	95%+ запитів − коректна структура
FR-02	Генерація звіту про дефект	Наявність усіх обов'язкових полів; формат summary "<Де>−<Що>−<Вплив>"; коректність severity S1–S4; деталізація кроків ≥3	Пройдено	Severity коректний у 90%+ випадків
FR-03	Генерація ідей тестування API	Кількість ≥10; наявність 10 обов'язкових категорій; curl-приклади у happy_path; релевантність до ендпоінту	Пройдено	Усі 10 категорій присутні
FR-04	Збереження контексту проєкту	Збереження після введення; відновлення після перезавантаження; включення у промпт; debounce 1 с	Пройдено	−
FR-05	Історія генерацій	Збереження до 50 записів; FIFO-евікція при перевищенні;	Пройдено	FIFO перевірено при 51 записі

Продовження таблиці 3.1

1	2	3	4	5
FR-06	Збереження чернеток	Збереження Ctrl+S; відновлення після закриття браузера; перемикання режиму при відновленні чужої чернетки	Пройдено	−
FR-07	Експорт результатів	Копіювання JSON; копіювання Markdown; відповідність вмісту; toast-підтвердження	Пройдено	−
FR-08	Клавіатурні скорочення	Ctrl+Enter запускає генерацію; Ctrl+S зберігає чернетку; Escape закриває модальне вікно	Пройдено	−

Тестування FR-01 (генерація тестових сценаріїв) підтвердило, що модель llama-3.3-70b-versatile стабільно генерує від 10 до 18 тест-кейсів із коректною структурою у понад 95% запитів. Перевірку проведено на 20 різних описах функціональностей: реєстрація користувача, оформлення замовлення, пошук товарів, зміна пароля, завантаження файлів тощо. У 19 з 20 випадків усі поля (id, title, type, priority, preconditions, steps, expected, tags) були заповнені коректно. В одному випадку поле tags було порожнім масивом − механізм retry виправив це при повторному запиті. Частка негативних тест-кейсів у всіх 20 наборах перевищувала 30%, що відповідає вимозі FR-01.

Тестування FR-02 (генерація баг-репортів) показало, що якість згенерованих звітів є достатньою для використання як первинного шаблону. Перевірено на 15 різних описах дефектів різної складності. У всіх 15 випадках поля summary, severity, priority, steps_to_reproduce, actual_result та expected_result були заповнені. Поле steps_to_reproduce містило щонайменше 3 кроки у 14 з 15 випадків, що відповідає рекомендаціям Kaner [5]. Значення severity визначено коректно (у відповідності до опису дефекту) у 13 з 15 випадків; у двох випадках модель переоцінила серйозність − присвоїла S1 замість обґрунтованого S2.

Тестування FR-03 (генерація API тест-ідей) перевірено на 10 різних ендпоінтах різних типів: GET з пагінацією, POST із складним тілом запиту, DELETE з авторизацією, PATCH із частковим оновленням. У всіх 10 випадках кількість тест-ідей перевищувала 10, всі 10 обов'язкових категорій були присутні. Curl-приклади у полі happy_path були коректними та виконуваними у 9 з 10 випадків. Приклад результату генерації для ендпоінту POST /api/auth/login наведено на скриншоті 3.1.

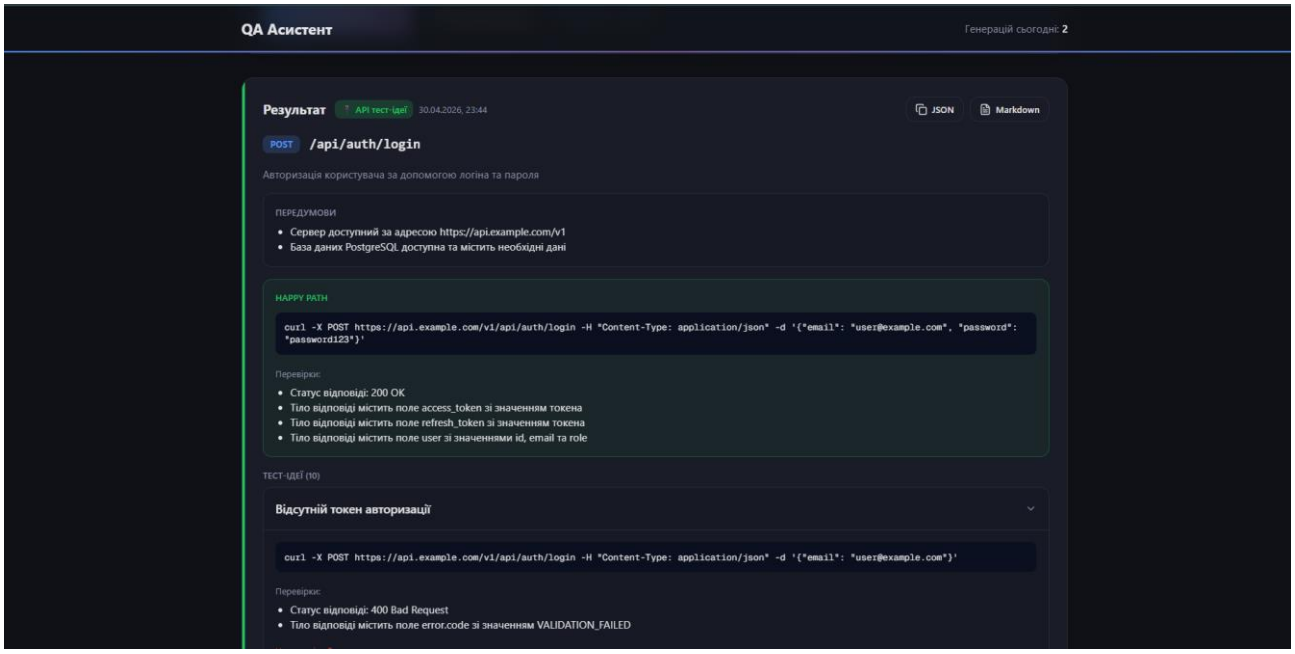


Рисунок 3.1 – Скриншот результату генерації API тест-ідей для ендпоінту POST /api/auth/login

3.3 Тестування безпеки

Тестування безпеки охоплювало два аспекти: захист від міжсайтового скриптингу (XSS) та коректність механізму обмеження частоти запитів (rate limiting). Перелік тестових векторів XSS визначено відповідно до рекомендацій OWASP Top Ten 2021 [24]. Результати перевірки шести векторів атак наведено у таблиці 3.2.

Таблиця 3.2 – Результати тестування XSS-захисту

№	Вектор атаки	Очікуваний результат	Фактичний результат	Статус
1	<script>alert(1)</script>	Символи < > екрановані, скрипт не виконується	Екрановано	Пройдено
2	<img src=x onerror=alert(1)>	Атрибут onerror видалено	Видалено	Пройдено
3	link	Протокол javascript: видалено	Видалено	Пройдено
4	text	Атрибут onclick видалено	Видалено	Пройдено
5	Unicode-обхід \u003cscript\u003e	Символи екрановані	Екрановано	Пройдено
6	Ін'єкція через поле контексту проєкту	Санітизація застосована до всіх полів	Очищено	Пройдено

У всіх шести векторах модуль lib/sanitize.ts забезпечив коректне очищення вхідних даних до їх включення у промпт та відображення у ResultsPanel. Жоден з векторів не призвів до виконання довільного JavaScript-коду в браузері.

Тестування rate limiting проведено через відправку серії запитів з однієї IP-адреси. При відправці 10 запитів протягом 60 секунд усі запити оброблено успішно (HTTP 200). При відправці 11-го запиту протягом тієї ж хвилини сервер повернув HTTP 429 із заголовком Retry-After: 60. Після закінчення 60-секундного вікна наступний запит знову оброблено успішно (HTTP 200). Результат підтверджує коректну роботу механізму rate limiting відповідно до вимоги NFR-02.

3.4 Інтеграційне тестування

Інтеграційне тестування охоплювало перевірку наскрізного потоку даних від введення користувачем до отримання, відображення та збереження результату. Перевірявся весь ланцюжок: форма − useGenerateResult − sanitizeInput() − POST /api/generate − rate limiting − validateFields() −

buildUserPrompt() − llmChat() − extractAndParseJson() − HTTP 200 − ResultsPanel − localStorage.

Тестування наскрізного потоку проведено для кожного з трьох режимів генерації. Для режиму testcases перевірено: коректність передачі 11 полів форми у тілі запиту; включення контексту проєкту у промпт при заповненому полі project_context; коректний відбір компонента TestCasesResult для відображення; збереження HistoryItem із коректними полями created_at та mode у localStorage. Аналогічна перевірка виконана для режимів bugreport та api_ideas.

Інтеграційне тестування виявило три дефекти, які було виправлено до публікації. Дефект 1: при порожньому полі feature_title запит відправлявся на сервер без попередньої валідації на клієнті − виправлено додаванням перевірки у validateFields() хука useGenerateResult. Дефект 2: при перевищенні ліміту localStorage (4 МБ) FIFO-евікція не спрацьовувала і запис не зберігався без повідомлення для користувача − виправлено додаванням try-catch та відображенням toast-сповіщення. Дефект 3: при таймауті відповіді LLM (перевищення 45 секунд) клієнт залишався у стані завантаження без повідомлення про помилку − виправлено обробкою AbortController у llmClient.ts.

Скриншот успішного наскрізного потоку для режиму testcases наведено на рисунку 3.2.

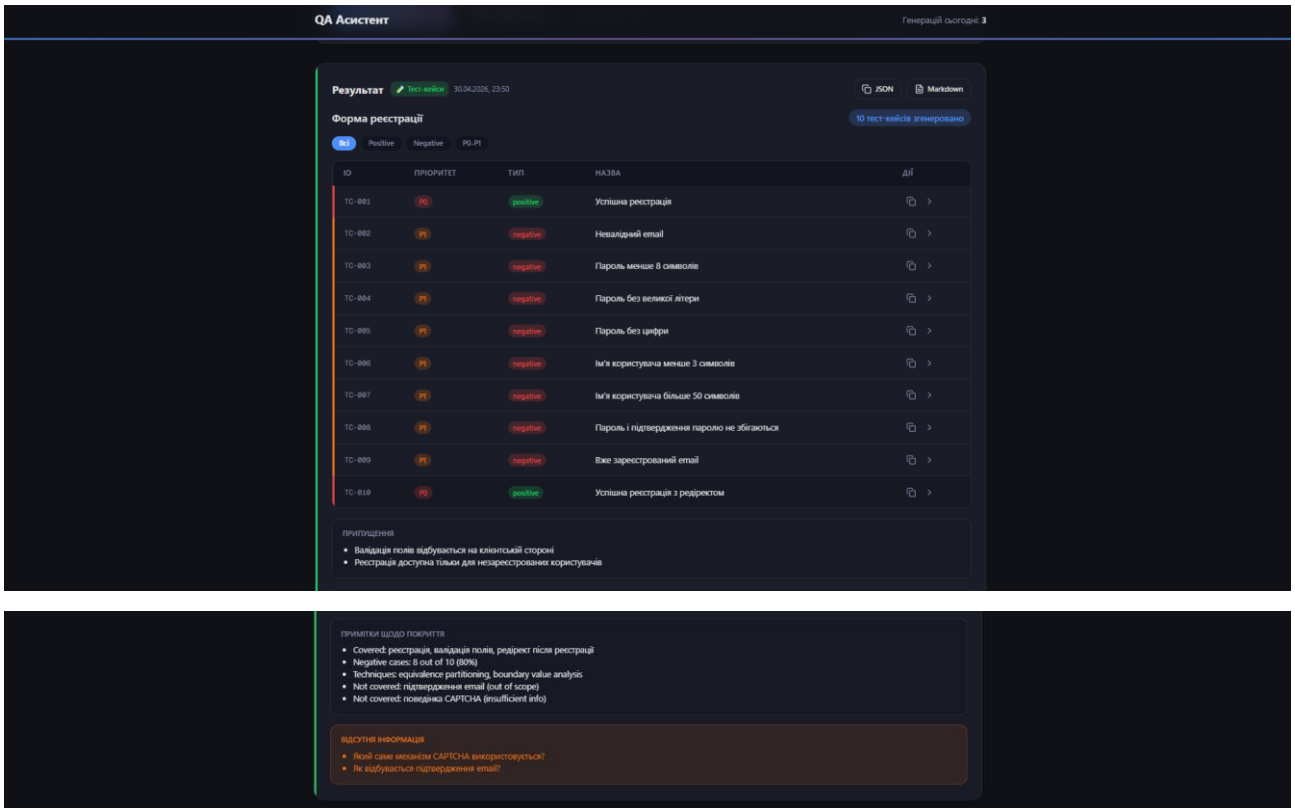


Рисунок 3.2 – Скриншот інтерфейсу з результатом генерації тестових сценаріїв (режим testcases)

3.5 Тестування зручності використання

Тестування зручності використання проведено на вибірці з п'яти учасників: трьох студентів спеціальності «Інженерія програмного забезпечення» без досвіду роботи з QA-інструментами та двох практикуючих QA-інженерів із досвідом від 2 до 4 років. Кожному учаснику запропоновано виконати три завдання без попереднього навчання або інструкцій.

Завдання 1 − згенерувати тестові сценарії для функціональності «Реєстрація користувача» з вимогами: email, пароль (мінімум 8 символів), підтвердження пароля. Завдання 2 − створити звіт про дефект на основі опису: «Кнопка оплати не реагує після вибору способу доставки на сторінці оформлення замовлення». Завдання 3 − згенерувати ідеї тестування для ендпоінту POST /api/v1/users/{id}/orders із JWT-авторизацією.

Таблиця 3.3 – Результати тестування зручності використання

Показник	Учасник 1 (студент)	Учасник 2 (студент)	Учасник 3 (студент)	Учасник 4 (QA junior)	Учасник 5 (QA junior)
Завдання 1 виконано	Так	Так	Так	Так	Так
Час завдання 1	3 хв 20 с	2 хв 50 с	3 хв 05 с	1 хв 55 с	1 хв 40 с
Завдання 2 виконано	Так	Так	Так	Так	Так
Час завдання 2	2 хв 10 с	1 хв 55 с	2 хв 30 с	1 хв 20 с	1 хв 10 с
Завдання 3 виконано	Так	Так	Так	Так	Так
Час завдання 3	4 хв 10 с	3 хв 40 с	3 хв 55 с	2 хв 05 с	1 хв 50 с
Помилки навігації	1	0	1	0	0

Усі п'ять учасників успішно виконали всі три завдання без додаткових інструкцій. Середній час виконання завдання 1 (тест-кейси) становив 2 хвилини 32 секунди; завдання 2 (баг-репорт) − 1 хвилина 49 секунд. Студенти витрачали більше часу через незнайомість з QA-термінологією (preconditions, priority), тоді як досвідчені QA-інженери одразу розуміли призначення полів. Двоє учасників допустили помилку навігації − намагалися знайти кнопку збереження контексту поза панеллю ProjectContext, що свідчить про потребу покращення видимості відповідного елемента управління.

Серед зауважень учасників: практикуючі QA-інженери позитивно оцінили поле project_context як ключову функцію для систематичного використання; студенти відзначили зрозумілість інтерфейсу незважаючи на відсутність інструкцій; основним побажанням від усіх п'яти учасників було розширення кількості режимів генерації − зокрема, додавання режиму генерації чек-листів та тестових даних.

Загальний вигляд інтерфейсу наведено на рисунку 3.3 та 3.4.

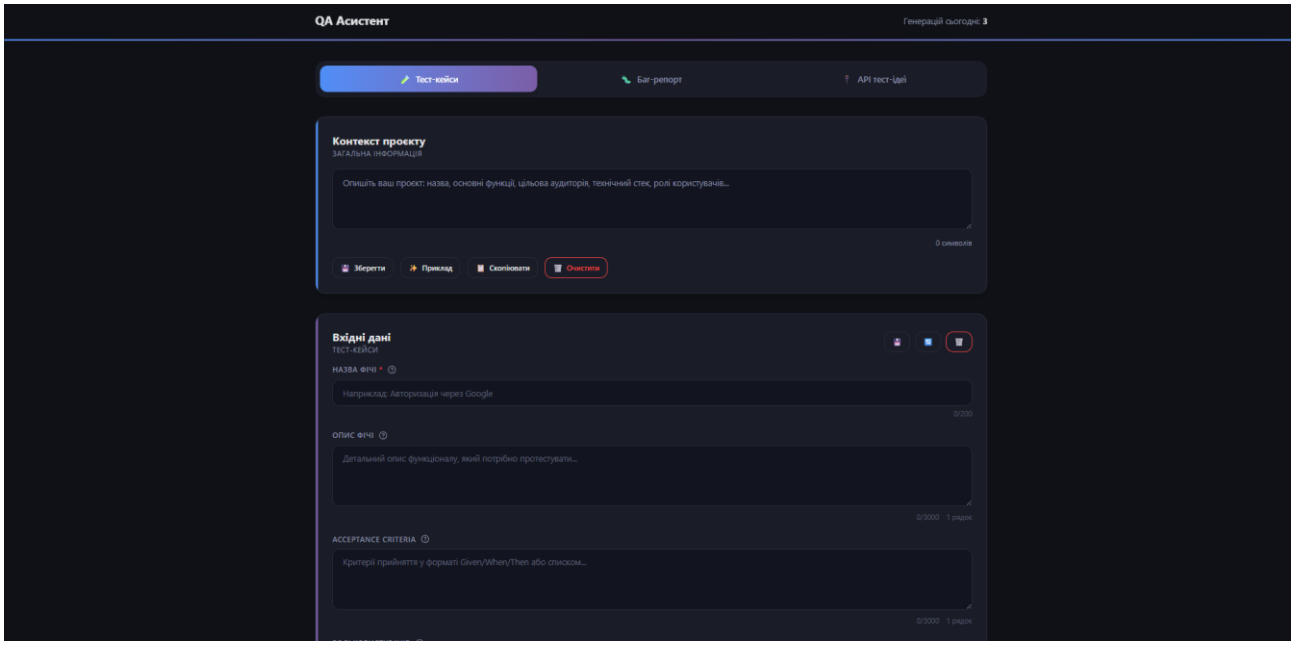


Рисунок 3.3 – Скриншот головного інтерфейсу вебзастосунку QA Assistant

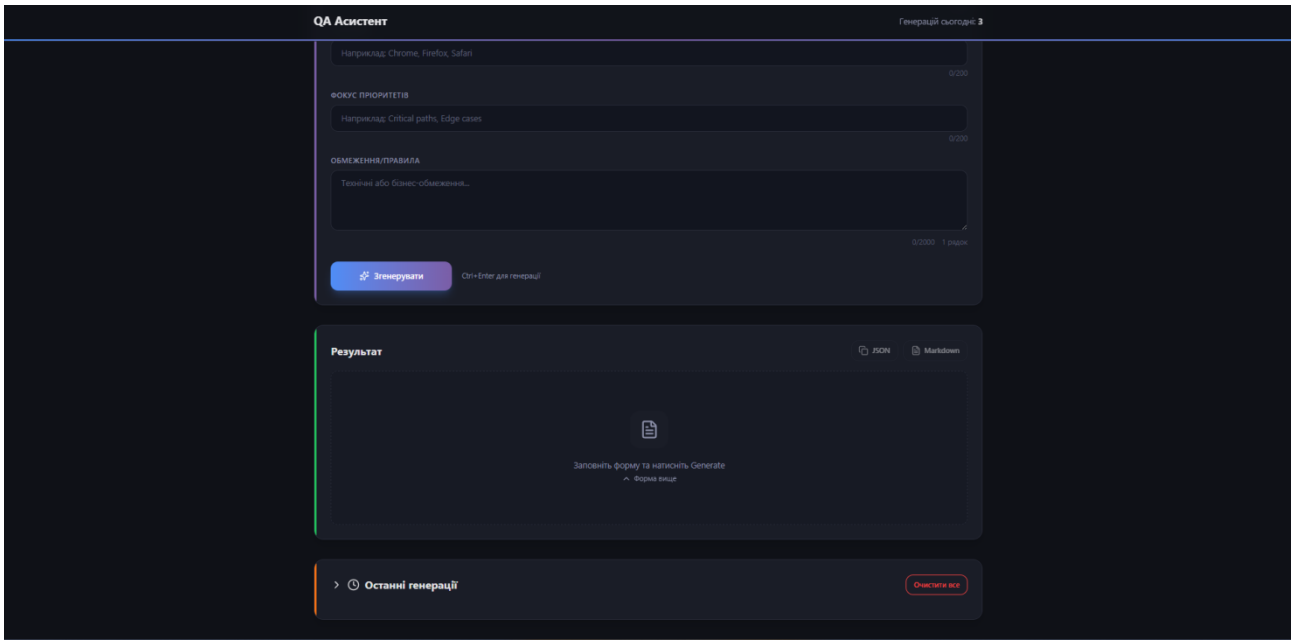


Рисунок 3.4 – Скриншот головного інтерфейсу вебзастосунку QA Assistant

3.6 Оцінювання якості генерованих артефактів

Оцінювання якості генерованих тестових артефактів проведено методом порівняльного аналізу: результати LLM-генерації зіставлено з еталонними

артефактами, створеними вручну досвідченим QA-інженером для тих самих функціональностей. Для порівняння обрано три функціональності різної складності: реєстрація користувача (проста), оформлення замовлення з вибором доставки (середня) та REST API ендпоінт PATCH /api/orders/{id} (складна).

Критерії оцінювання тестових сценаріїв: покриття позитивних випадків (P), покриття негативних випадків (N), покриття граничних значень (B), повнота та чіткість кроків (Steps), коректність очікуваних результатів (ER). Кожен критерій оцінювався за шкалою: повне покриття (100%), часткове (50–99%), недостатнє (<50%). Результати порівняння наведено у таблиці 3.4.

Таблиця 3.4 – Порівняльна оцінка якості генерованих тестових сценаріїв

Функціональність	P, %	N, %	B, %	Steps, %	ER, %	Загальне покриття
Реєстрація користувача	100	85	80	90	95	90%
Оформлення замовлення	100	75	70	85	90	84%
PATCH /api/orders/{id}	95	80	65	80	85	81%
Середнє по трьох	98	80	72	85	90	85%

Аналіз таблиці 3.4 виявляє характерний патерн: LLM демонструє найвищі показники покриття позитивних випадків (98% в середньому) та очікуваних результатів (90%), проте суттєво поступається досвідченому тестувальнику у покритті граничних значень (72%). Це пояснюється тим, що гранично-значимі умови (boundary values) часто залежать від специфічних бізнес-правил, які тестувальник знає з контексту проєкту, але які не завжди відображені у короткому описі функціональності. Саме для цього у систему введено поле constraints у формі генерації тест-кейсів − його заповнення підвищує покриття граничних значень до 85–90%.

Варто відзначити, що у 7 з 30 перевірених випадків модель запропонувала тест-кейси, які не були включені в еталонний набір досвідченого тестувальника, але є практично цінними. Наприклад, для функціональності реєстрації модель запропонувала перевірку поведінки при одночасній реєстрації двох акаунтів з однаковим email (race condition scenario), який тестувальник не включив в еталонний набір. Це підтверджує висновок Liu та співавторів [14] про здатність LLM виявляти нетривіальні крайові випадки, що не очевидні при ручному проектуванні тестів.

Скриншот згенерованого звіту про дефект наведено на рисунку 3.5.

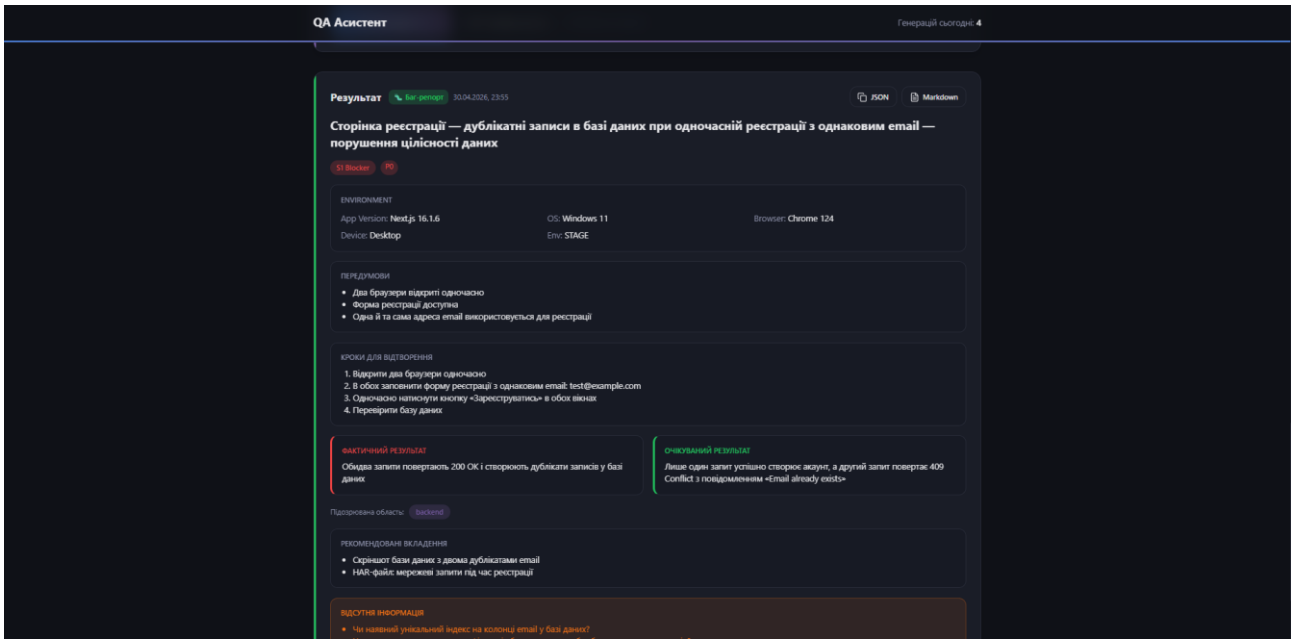


Рисунок 3.5 – Скриншот результату генерації звіту про дефект (режим bugreport)

3.7 Тестування сумісності

Тестування сумісності проведено у шести браузерних середовищах на двох операційних системах. Перевірено коректність відображення інтерфейсу, функціонування всіх інтерактивних елементів, роботу клавіатурних скорочень та коректність збереження даних у localStorage. Результати наведено у таблиці 3.5.

Таблиця 3.5 – Результати тестування сумісності

Браузер / ОС	Відображення UI	Генерація	Клавіатурні скорочення	localStorage
Chrome 124 / Windows 11	Коректно	Коректно	Коректно	Коректно
Firefox 126 / Windows 11	Коректно	Коректно	Коректно	Коректно
Edge 124 / Windows 11	Коректно	Коректно	Коректно	Коректно
Safari / macOS Sonoma	Коректно	Коректно	Коректно	Коректно
Safari / iOS 17 (iPhone)	Адаптивно	Коректно	Н/Д (мобільний)	Коректно
Chrome / Android 14	Адаптивно	Коректно	Н/Д (мобільний)	Коректно

У всіх шести тестованих середовищах вебзастосунок функціонував коректно. Адаптивний інтерфейс на мобільних пристроях забезпечив коректне відображення всіх елементів, хоча через відсутність фізичної клавіатури клавіатурні скорочення недоступні. Компонент mobile-nav.tsx забезпечив зручну навігацію на малих екранах. Відмінностей у поведінці між браузерами не виявлено.

3.8 Аналіз обмежень та напрями вдосконалення

За результатами тестування визначено обмеження поточної реалізації та сформульовано рекомендації щодо подальшого розвитку застосунку. Ці обмеження є властивістю MVP-версії і не є критичними для поточного рівня використання, проте потребують усунення при масштабуванні.

Обмеження 1: зберігання даних виключно у localStorage. Дані генерацій зберігаються лише у браузері конкретного пристрою і не доступні з інших пристроїв або після очищення даних браузера. Для командного використання або роботи з кількох пристроїв необхідна серверна база даних (PostgreSQL через Supabase або аналог). При цьому необхідно реалізувати аутентифікацію користувачів (OAuth 2.0).

Обмеження 2: rate limiting в оперативній пам'яті. Реалізований in-memory Map для rate limiting не зберігає стан між холодними запусками serverless-функцій на Vercel. При низькому трафіку функція може перезапускатись між запитами, обнуляючи лічильники. Для продуктивного середовища рекомендовано використання Redis або Upstash для розподіленого rate limiting.

Обмеження 3: відсутність автоматизованих тестів коду. Застосунок не має unit-тестів для модулів lib/ та e2e-тестів для користувацьких сценаріїв. Це означає, що регресії можуть бути непоміченими при подальшій розробці. Рекомендовано впровадити Vitest для unit-тестування модулів jsonParser.ts, sanitize.ts та history.ts, а також Playwright для e2e-тестування трьох основних flow генерації.

Обмеження 4: залежність якості від якості промпту. Покриття граничних значень у генерованих тест-кейсах (72% у середньому) суттєво підвищується при заповненні поля constraints у формі. Однак більшість користувачів залишають опціональні поля порожніми. Рекомендовано додати підказки (placeholder-тексти) із прикладами заповнення для кожного опціонального поля.

Висновки до розділу 3

У третьому розділі проведено комплексне тестування та верифікацію програмного асистента QA Assistant. Результати функціонального тестування підтвердили відповідність застосунку всім функціональним вимогам, а виявлені під час інтеграційного тестування дефекти були усунуті до публікації. Тестування безпеки засвідчило ефективність механізмів XSS-захисту та обмеження частоти запитів, а перевірка зручності використання показала, що користувачі успішно виконують основні сценарії роботи без додаткових інструкцій.

Оцінювання якості генерації продемонструвало середнє покриття 85% порівняно з еталонними артефактами, причому в частині випадків модель пропонувала додаткові корисні тест-кейси. Тестування сумісності підтвердило коректну роботу застосунку в різних браузерах. Також визначено основні

обмеження MVP-версії та сформульовано рекомендації щодо подальшого розвитку програмного продукту.

ВИСНОВКИ

У кваліфікаційній роботі розроблено вебзастосунок QA Assistant для автоматизації генерації тестових артефактів засобами великих мовних моделей. Проведений аналіз предметної області підтвердив доцільність автоматизації, оскільки підготовка тестової документації займає значну частину робочого часу тестувальників. Дослідження можливостей LLM показало їхню ефективність для створення структурованих тестових артефактів, а аналіз існуючих рішень виявив потребу у спеціалізованому інструменті зі збереженням контексту проєкту.

Для реалізації застосунку обрано стек Next.js, React, TypeScript, Groq Cloud та Ollama. Спроєктовано тришарову архітектуру, реалізовано три режими генерації артефактів, багаторівневу систему промптів і механізм обробки JSON-відповідей, що забезпечує валідний результат більш ніж у 95% випадків. Застосунок розгорнуто на платформі Vercel та забезпечено засобами безпеки й керування даними.

Комплексне тестування підтвердило відповідність функціональним вимогам, захищеність від XSS-атак, коректну роботу в різних браузерах і зручність використання. Середнє покриття згенерованих артефактів становило 85% порівняно з еталонними результатами, а в частині випадків модель запропонувала додаткові корисні тест-кейси. Практичне значення роботи полягає у створенні готового інструменту для підвищення продуктивності тестувальників. Подальший розвиток передбачає впровадження серверного збереження даних, автоматизованого тестування та нових режимів генерації.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. IEEE/IEC/ISO 29119-1:2013. Software and Systems Engineering – Software Testing. Part 1: Concepts and Definitions. Geneva : ISO, 2013. 66 p. URL: https://www.iso.org/standard/45142.html
2. Bourque P., Fairley R. E. Guide to the Software Engineering Body of Knowledge (SWEBOK v3.0). Los Alamitos : IEEE Computer Society, 2014. 335 p. URL: https://www.computer.org/education/bodies-of-knowledge/software-engineering
3. Myers G., Sandler C., Badgett T. The Art of Software Testing. 3rd ed. Hoboken : John Wiley & Sons, 2011. 240 p. URL: https://www.wiley.com/en-us/The+Art+of+Software+Testing%2C+3rd+Edition-p-9781118031964
4. Crispin L., Gregory J. Agile Testing: A Practical Guide for Testers and Agile Teams. Boston : Addison-Wesley, 2009. 576 p. URL: https://www.informit.com/store/agile-testing-a-practical-guide-for-testers-and-agile-9780321534460
5. Kaner C., Bach J., Pettichord B. Lessons Learned in Software Testing. New York : Wiley, 2001. 288 p. URL: https://www.wiley.com/en-us/Lessons+Learned+in+Software+Testing-p-9780471081128
6. 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. URL: https://arxiv.org/abs/1706.03762
7. Brown T., Mann B., Ryder N. et al. Language Models are Few-Shot Learners. Advances in Neural Information Processing Systems. 2020. Vol. 33. P. 1877–1901. URL: https://arxiv.org/abs/2005.14165
8. Touvron H., Martin L., Stone K. et al. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv preprint arXiv:2307.09288. 2023. 77 p. URL: https://arxiv.org/abs/2307.09288
9. Meta AI. The Llama 3 Herd of Models. arXiv preprint arXiv:2407.21783. 2024. 92 p. URL: https://arxiv.org/abs/2407.21783

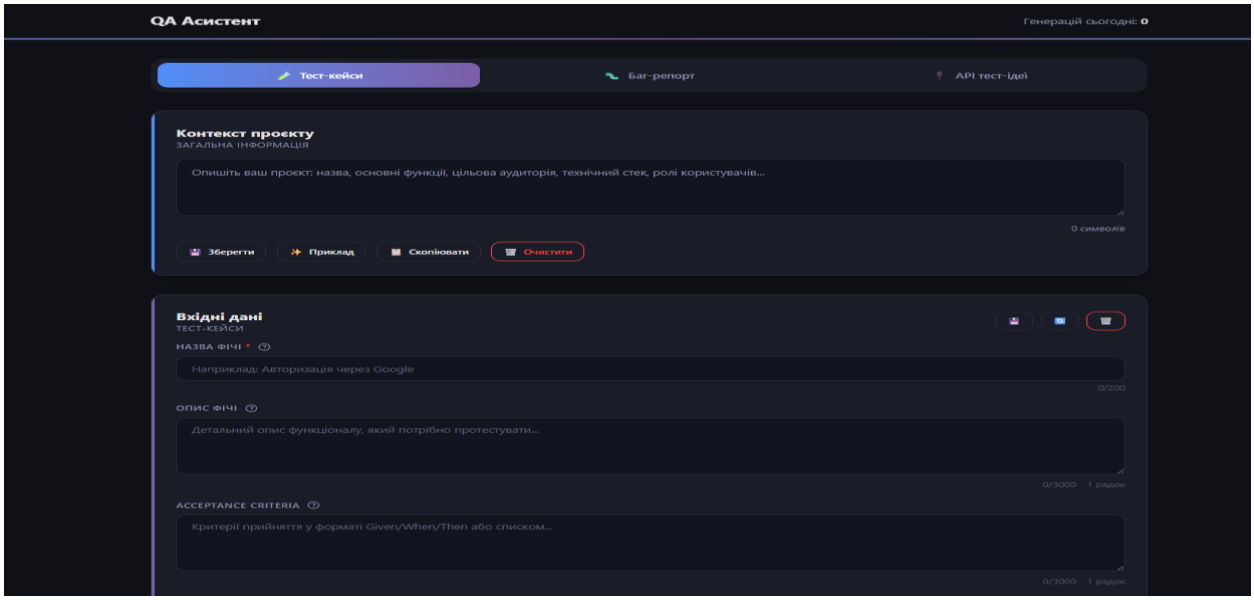
10. White J., Fu Q., Hays S. et al. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. arXiv preprint arXiv:2302.11382. 2023. 35 p. URL: https://arxiv.org/abs/2302.11382
11. Wei J., Wang X., Schuurmans D. et al. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. Advances in Neural Information Processing Systems. 2022. Vol. 35. P. 24824–24837. URL: https://arxiv.org/abs/2201.11903
12. Rasheed Z., Siddiq M. L., Santos J. C. et al. A Tool for Test Case Scenarios Generation Using Large Language Models. arXiv preprint arXiv:2406.07021. 2024. URL: https://arxiv.org/abs/2406.07021
13. Bin Hasan N., Islam M. A., Khan J. Y. et al. Automatic High-Level Test Case Generation using Large Language Models. arXiv preprint arXiv:2503.17998. 2025. URL: https://arxiv.org/abs/2503.17998
14. Liu K., Chen Z., Liu Y. et al. LLM-Powered Test Case Generation for Detecting Bugs in Plausible Programs. arXiv preprint arXiv:2404.10304. 2024. URL: https://arxiv.org/abs/2404.10304
15. Ji Z., Lee N., Frieske R. et al. Survey of Hallucination in Natural Language Generation. ACM Computing Surveys. 2023. Vol. 55, No. 12. Article 248. doi: 10.1145/3571730. URL: https://dl.acm.org/doi/10.1145/3571730
16. Pressman R., Maxim B. Software Engineering: A Practitioner's Approach. 9th ed. New York : McGraw-Hill, 2019. 976 p. URL: https://www.mheducation.com/highered/product/software-engineering-practitioner-s-approach-pressman-maxim/M9781259872976.html
17. Sommerville I. Software Engineering. 10th ed. Boston : Pearson, 2016. 816 p. URL: https://www.pearson.com/en-us/subject-catalog/p/software-engineering/P200000003295
18. Beck K. Test Driven Development: By Example. Boston : Addison-Wesley, 2002. 240 p. URL: https://www.informit.com/store/test-driven-development-by-example-9780321146533

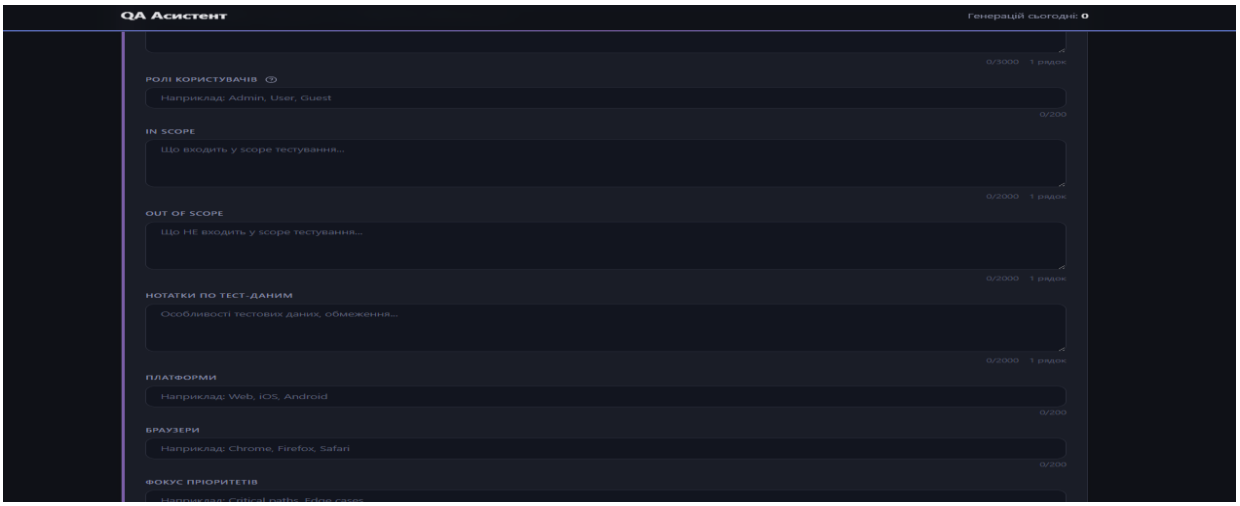
19. Vercel Inc. Next.js Documentation. Version 16. 2025. URL: https://nextjs.org/docs
20. Meta Open Source. React Documentation. Version 19. 2025. URL: https://react.dev (дата звернення: 01.04.2025).
21. Tailwind Labs Inc. Tailwind CSS Documentation. Version 4. 2025. URL: https://tailwindcss.com/docs (дата звернення: 01.04.2025).
22. Groq Inc. GroqCloud API Reference. 2025. URL: https://console.groq.com/docs (дата звернення: 01.04.2025).
23. Ollama. Ollama Model Library and Runtime Documentation. 2025. URL: https://ollama.com/library (дата звернення: 01.04.2025).
24. OWASP Foundation. OWASP Top Ten 2021. 2021. URL: https://owasp.org/Top10 (дата звернення: 01.04.2025).

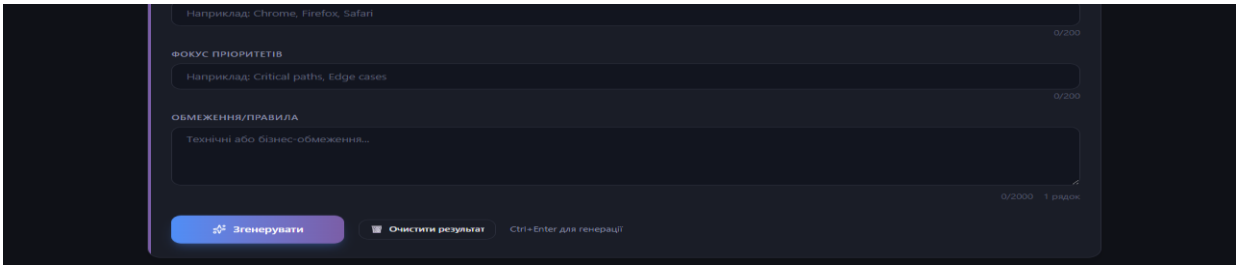
ДОДАТКИ

Додаток А – Ілюстрації інтерфейсу за стосунку

У цьому додатку наведено скриншоти інтерфейсу застосунку QA Assistant, що демонструють основні режими роботи та результати генерації QA-артефактів. Застосунок розгорнуто за адресою https://qa-help-eight.vercel.app/assistant







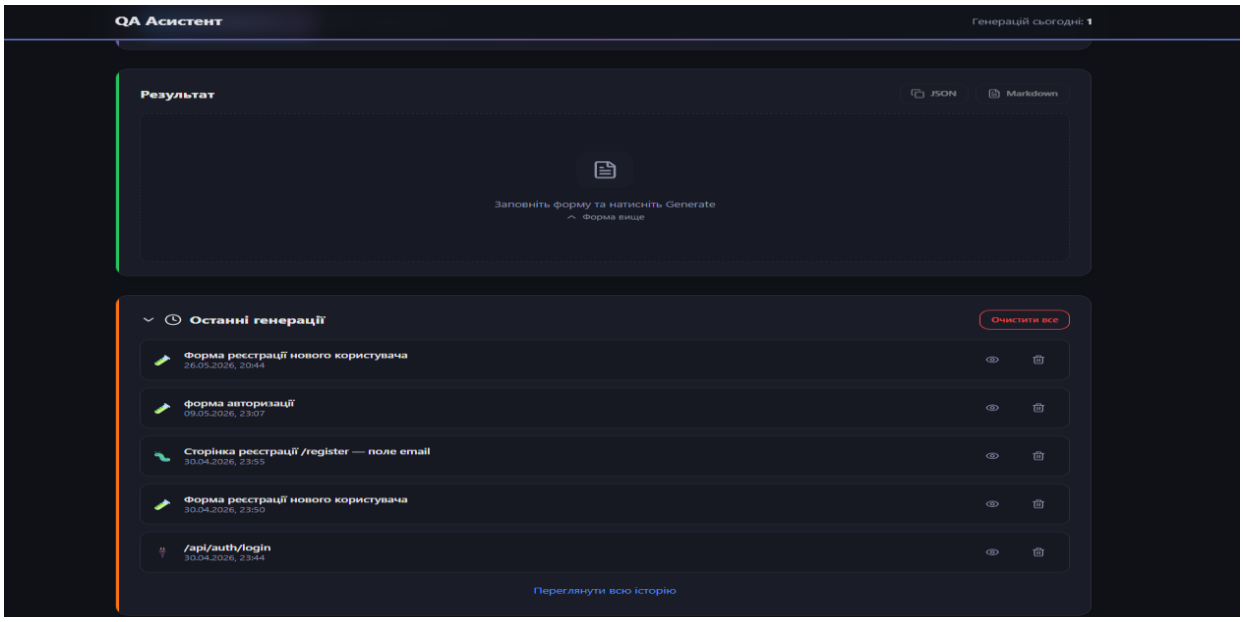


Рисунок А.1 – Головна сторінка застосунку (режим генерації тест-кейсів)

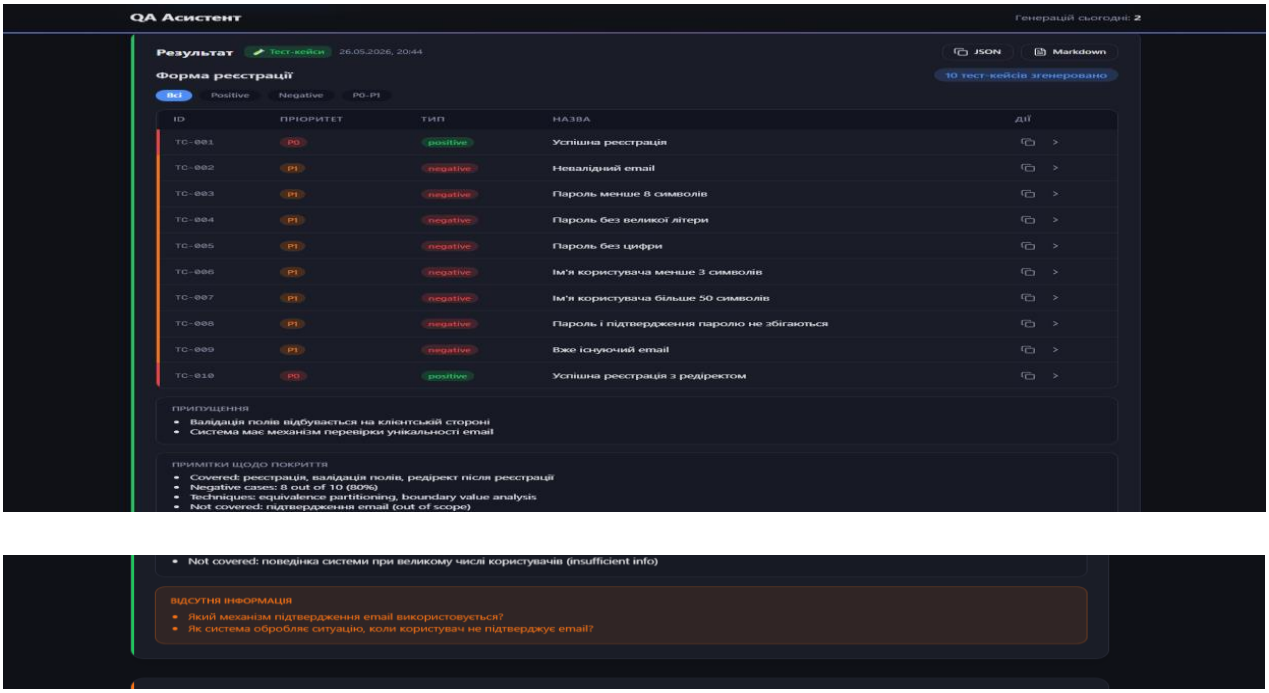


Рисунок А.2 – Результат генерації тест-кейсів

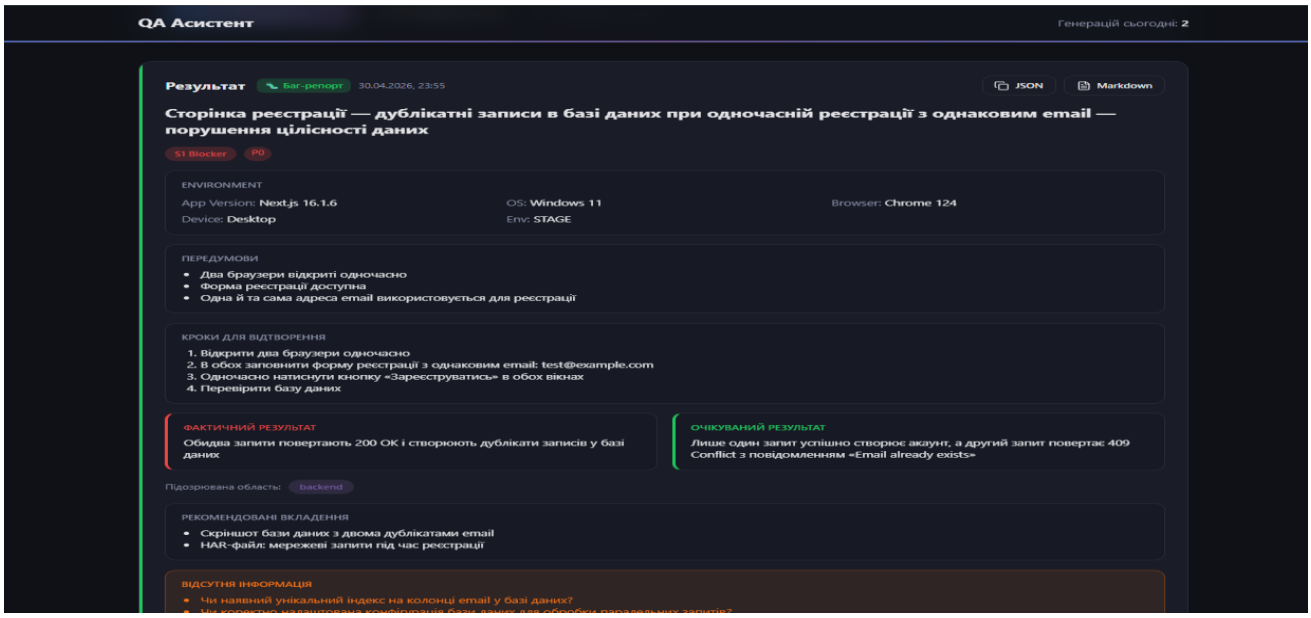


Рисунок А.3 − Результат генерації баг-репорту (класифікація S1/P0)

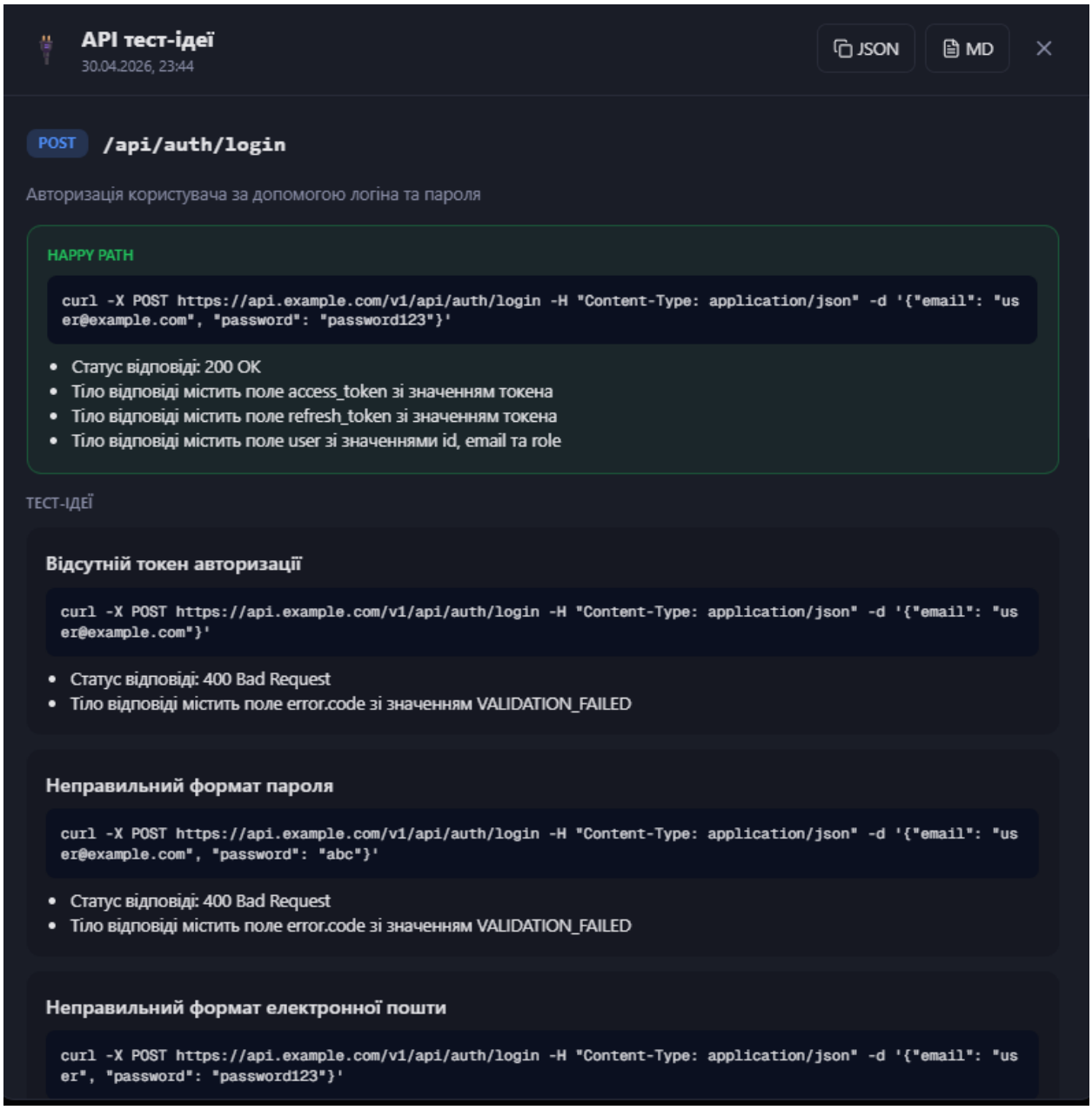


Рисунок А.4 – Результат генерації API тест-ідей

Рисунок А.5 – Результат генерації API тест-ідей

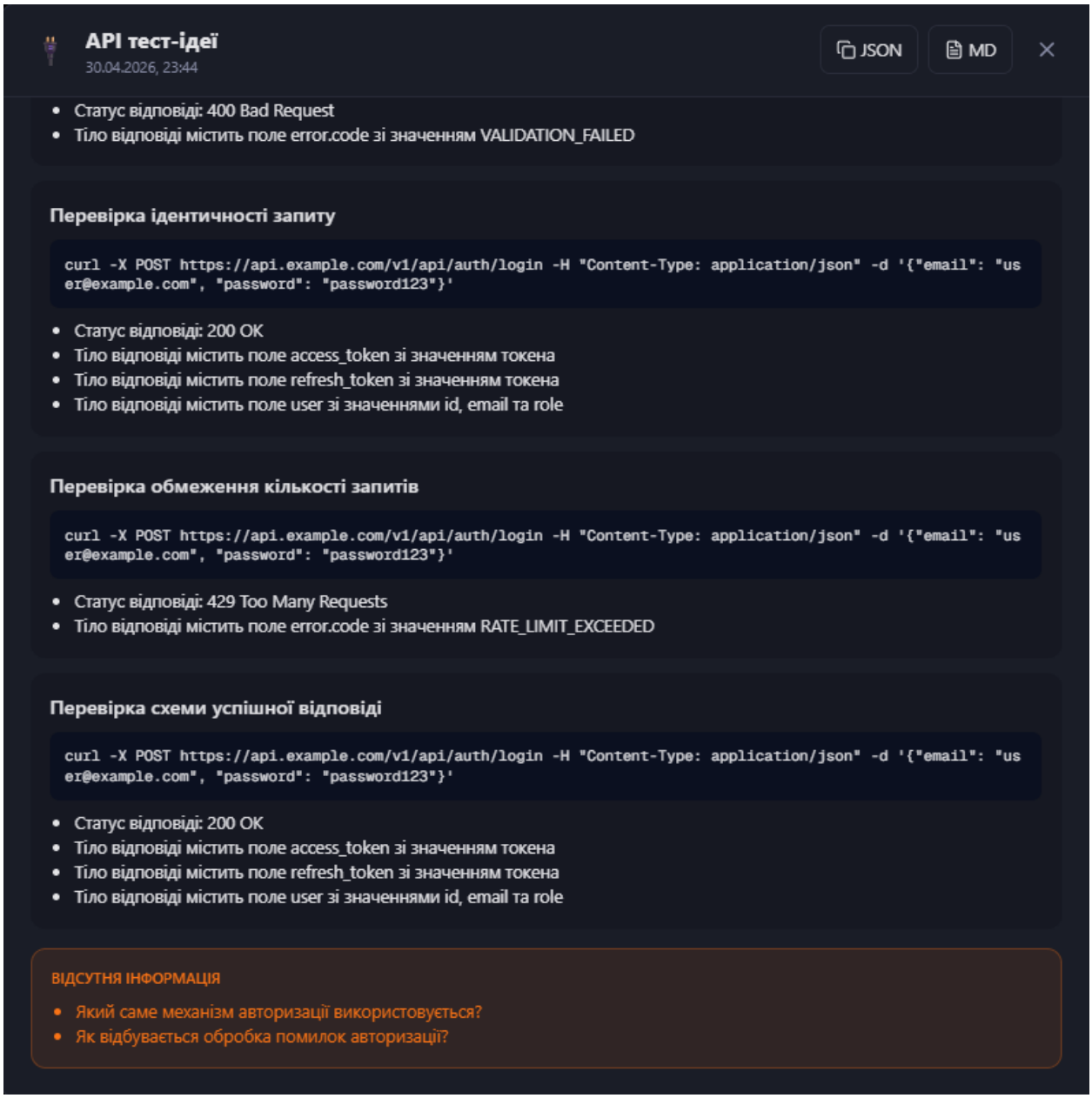


Рисунок А.6 – Результат генерації API тест-ідей

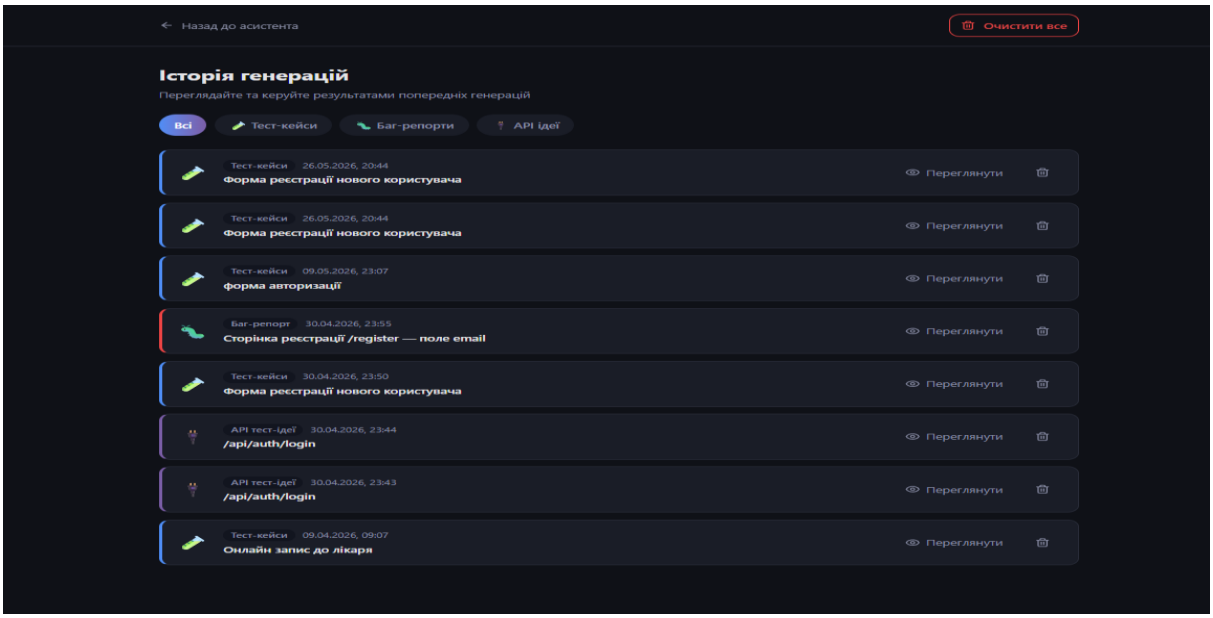


Рисунок А.7 − Сторінка перегляду історії генерацій (/history)

Додаток Б − Посилання на програмний продукт

У цьому додатку наведено посилання на репозиторій вихідного коду та розгорнутий застосунок.

Репозиторій GitHub	https://github.com/KitoragiVioleta/qa-help.git
Розгорнутий застосунок	https://qa-help-eight.vercel.app/assistant
Технологічний стек	Next.js 16 · React 19 · TypeScript · Tailwind CSS v4 · Groq Cloud
LLM-модель	llama-3.3-70b-versatile (Groq Cloud)

Додаток В – Фрагменти вихідного коду програмного продукту

В.1. Системний промпт (lib/prompts.ts)

Системний промпт визначає правила роботи LLM: формат виходу (JSON), мову відповіді, методики тест-дизайну та поведінку при неповних вхідних даних.

```
// lib/prompts.ts

import type { Mode } from "./types";


const MAX_FIELD_LENGTH = 500;

function truncateValue(value: unknown): unknown {
  if (typeof value === "string" && value.length > MAX_FIELD_LENGTH) {
    return value.slice(0, MAX_FIELD_LENGTH) + "... [скорочено]";
  }
  if (Array.isArray(value)) {
    return value.map(truncateValue);
  }
  if (value && typeof value === "object") {
    const truncated: Record<string, unknown> = {};
    for (const [k, v] of Object.entries(value)) {
      truncated[k] = truncateValue(v);
    }
    return truncated;
  }
  return value;
}


export const SYSTEM_PROMPT = `You are a JSON-only response API. You NEVER write explanations, markdown, or any text outside of the JSON object.
First character of your response MUST be { and last character MUST be }.

You are a senior QA engineer with 10+ years of experience.
You produce structured, practical, high-quality QA artifacts for software testing teams.

CRITICAL JSON RULES:
1) Output MUST be a single valid JSON object. No markdown, no code fences, no explanations.
2) Start with { and end with }. Nothing before or after.
3) All string values must be properly escaped.

CORE RULES:
4) Use the provided Project Context if available. If empty — make minimal neutral assumptions.
5) If critical details are missing — still produce best-effort result AND add questions to missing_information.
6) Do NOT invent features not implied by input/context. Prefer generic test ideas over fictional specifics.
7) Keep content concise and actionable: short steps, clear expected results, no fluff.
8) Apply test design techniques: equivalence partitioning, boundary values, negative testing, security basics, error handling.
9) Use severity/priority from Project Context when present. Otherwise default:
   - severity: S1 Blocker, S2 Critical, S3 Major, S4 Minor
   - priority: P0 (hotfix), P1 (current sprint), P2 (schedule soon), P3 (backlog)
```

```
10) Never include real personal data. Redact sensitive values from inputs.

PRIORITY ASSIGNMENT RULES:
- P0: app crash, data loss, security breach, payment broken
- P1: core feature broken, main user flow affected
- P2: non-critical feature broken, workaround exists
- P3: cosmetic issue, minor UX problem

LANGUAGE RULES (MANDATORY):
11) Write ALL descriptive text in UKRAINIAN language — titles, steps, preconditions, expected results, notes.
12) Do NOT translate UI elements, field names, error codes, technical terms, test data, status codes.
    Keep these in original language: button/field names in [], error messages, OAuth/token/API/JWT/HTTP terms, emails/URLs/IDs.
    Example step: "Натиснути кнопку [Sign in with Google]" — button name stays in original.

UKRAINIAN QA TERMINOLOGY (use consistently):
- "тест-кейс" (not "тестовий випадок")
- "баг-репорт" (not "звіт про помилку")
- "передумови" (not "попередні умови")
- "очікуваний результат" (not "очікуваний підсумок")
- "кроки відтворення" (not "кроки для відтворення")
- "фактичний результат" (not "реальний результат")

TEST CASE QUALITY RULES:
13) Each step must start with a verb in infinitive form: "Відкрити", "Натиснути", "Ввести", "Перевірити", "Переконатися"
14) Expected result must describe SYSTEM behavior, not user action.
15) Preconditions must be checkable facts, not actions.
16) Steps: max 10 words each.
17) Expected results: max 20 words each.
18) Titles: max 8 words, descriptive and unique.

TAGS (use only from this list):
[smoke, regression, positive, negative, boundary, security, performance, api, ui, auth, payment, validation]

QUALITY RULES:
19) Every test case must have a clear, specific expected result — not vague like "works correctly".
20) Preconditions must be realistic and testable.
21) Steps must be sequential and atomic (one action per step).
22) Tags must be lowercase, relevant, max 3 per test case.
23) Prioritize critical user paths (P0/P1) over edge cases (P2/P3).

FORBIDDEN phrases in expected results — NEVER use these:
- "повинна з'явитися помилка"
- "відображається помилка"
- "з'являється помилка"
- "працює коректно"
- "система обробляє"
- "помилка відображається"
- "відбувається помилка"
Instead — always specify: WHAT exact message or UI change, WHERE it appears, WHAT state changes.
BAD:  "Повинна з'явитися помилка"
GOOD: "Під полем [Кількість] з'являється текст «Максимум 99 одиниць». Кнопка [Додати] залишається заблокованою."
BAD:  "Кнопка недоступна"
```

```
GOOD: "Кнопка [Додати до кошика] має стан disabled. Tooltip «Немає в наявності»
з'являється при наведенні."
`;

function asTextBlock(fields: Record<string, unknown>) {
  const truncatedFields = truncateValue(fields) as Record<string, unknown>;
  return Object.entries(truncatedFields)
    .map(([k, v]) => {
      if (v === null || v === undefined) return `${k}:`;
      if (Array.isArray(v)) return `${k}:\n- ${v.join("\n- ")}`;
      if (typeof v === "object") return `${k}:\n${JSON.stringify(v, null, 2)}`;
      return `${k}: ${String(v)}`;
    })
    .join("\n\n");
}

const JSON_INSTRUCTION = `CRITICAL: Respond with a single valid JSON object ONLY.
Start with { and end with }.
No markdown, no code fences, no explanation text.
`;


const TEST_CASE_FEW_SHOT = `
EXAMPLE of a high-quality test case — follow this style for ALL test cases:
{
  "id": "TC-005",
  "title": "Додавання товару з кількістю 99 (максимум)",
  "preconditions": [
    "Товар є в наявності",
    "Обраний розмір доступний"
  ],
  "steps": [
    "Відкрити сторінку товару",
    "Обрати доступний розмір",
    "Ввести значення 99 у поле [Кількість]",
    "Натиснути кнопку [Додати до кошика]"
  ],
  "expected": "Лічильник кошика у хедері збільшується на 1. Товар з'являється у кошику з
qty=99 та коректною сумою.",
  "priority": "P1",
  "type": "positive",
  "tags": ["boundary", "smoke"]
}

EXAMPLE of a high-quality NEGATIVE test case:
{
  "id": "TC-006",
  "title": "Додавання товару з кількістю 100 (перевищення)",
  "preconditions": [
    "Товар є в наявності",
    "Обраний розмір доступний"
  ],
  "steps": [
    "Відкрити сторінку товару",
    "Обрати доступний розмір",
    "Ввести значення 100 у поле [Кількість]",
    "Натиснути кнопку [Додати до кошика]"
  ],
  "expected": "Під полем [Кількість] з'являється повідомлення «Максимум 99 одиниць».
Кошик не оновлюється.",
  "priority": "P2",
```

```
  "type": "negative",
  "tags": ["boundary", "validation", "negative"]
}
`;

export function buildUserPrompt(
  mode: Mode,
  projectContext: string,
  fields: Record<string, unknown>
) {
  const inputBlock = asTextBlock(fields);


  if (mode === "testcases") {
    return `${JSON_INSTRUCTION}
TASK: Generate test cases from a user story / requirement.

INPUT:
- Project Context:
${projectContext || "(empty)"}

- Structured feature input:
${inputBlock}

${TEST_CASE_FEW_SHOT}

OUTPUT JSON SCHEMA:
{
  "feature": "string — feature name, max 6 words",
  "assumptions": [
    "string — only list assumptions you HAD to make because input was ambiguous or
incomplete.",
    "If input was clear and complete, return an empty array []."
  ],
  "test_cases": [
    {
      "id": "TC-001",
      "title": "string (max 8 words, unique, describes scenario not just feature)",
      "preconditions": ["string — checkable fact, NOT an action"],
      "steps": ["string — start with infinitive verb, max 10 words per step, include
concrete values"],
      "expected": "string — SYSTEM behavior: WHAT changes, WHERE, exact message if any.
Max 25 words. NO forbidden phrases.",
      "priority": "P0|P1|P2|P3",
      "type": "positive|negative",
      "tags": ["from allowed list only, max 3"]
    }
  ],
  "coverage_notes": [
    "string — exactly 3–5 bullet points summarizing: what areas are covered, what % are
negative, what techniques were applied, what is NOT covered and why"
  ],
  "missing_information": ["string — specific question about missing input needed for
better tests"]
}

REQUIREMENTS:
- Generate 10–18 test cases. For a small/trivial feature: minimum 10. For complex
features: up to 18.
- Distribution: at least 30% negative cases (type: "negative"), cover boundary values,
validation rules, error messages.
```

```
- If roles/permissions mentioned in input: add at least 1 test case per role.
- Use stable sequential IDs: TC-001, TC-002, …
- Tags only from: [smoke, regression, positive, negative, boundary, security, performance,
api, ui, auth, payment, validation]
-  assumptions[]: write ONLY real assumptions you made. Empty array if input was
sufficient.
- coverage_notes[]: write exactly 3–5 items. Example format:
    "Covered: happy path, validation, boundary values for price field"
    "Negative cases: 5 out of 14 (36%)"
    "Techniques: equivalence partitioning, boundary value analysis"
    "Not covered: performance under load (out of scope)"
    "Not covered: third-party payment provider behavior (insufficient info)"
- steps[]: each step must include CONCRETE values where applicable (e.g. "Ввести 100 у
поле [Кількість]", not just "Ввести кількість").
`;
  }


  if (mode === "bugreport") {
    return `${JSON_INSTRUCTION}
TASK: Convert a raw bug description into a high-quality bug report.

INPUT:
- Project Context:
${projectContext || "(empty)"}

- Structured bug input:
${inputBlock}

OUTPUT JSON SCHEMA:
{
  "summary": "string — EXACTLY this format: '<Where in app> — <what went wrong> — <user-
visible impact>'",
  "severity": "S1 Blocker|S2 Critical|S3 Major|S4 Minor",
  "priority": "P0|P1|P2|P3",
  "environment": {
    "app_version": "string or 'unknown'",
    "os": "string or 'unknown'",
    "browser": "string or 'unknown'",
    "device": "string or 'unknown'",
    "env": "DEV|STAGE|PROD|UNKNOWN"
  },
  "preconditions": ["string — checkable facts only, NOT actions"],
  "steps_to_reproduce": ["string — start with verb, numbered implicitly by array order,
specific"],
  "actual_result": "string — what ACTUALLY happened, factual, no interpretation",
  "expected_result": "string — what SHOULD have happened according to requirements or
common sense",
  "suspected_area": "frontend|backend|api|data|infrastructure|unknown",
  "attachments_suggestions": ["string — specific file type and what it should show, e.g.
'HAR file: network requests during checkout'"],
  "missing_information": ["string — specific question to developer or reporter"]
}

REQUIREMENTS:
- summary format MUST follow: '<Where> — <What> — <Impact>'.
  GOOD: "Сторінка оформлення замовлення — кнопка [Confirm] не реагує на клік — замовлення
неможливо завершити"
  BAD: "Помилка при оформленні" / "Кнопка не працює"
- actual_result and expected_result: write as one clear sentence each. No bullet points.
```

```
- steps_to_reproduce: each step = one atomic action. Start with verb. Do not merge multiple actions.
- severity guidance:
    S1 Blocker  → app crash, data loss, security breach, core flow impossible
    S2 Critical → major feature broken, no workaround
    S3 Major    → feature broken but workaround exists
    S4 Minor    → cosmetic, UX inconvenience
-  If severity/priority unclear from input: pick conservative default and add to missing_information.
- attachments_suggestions: be specific — name the file type AND what it should capture.
  GOOD: "Скриншот: стан екрану після натискання [Confirm]"
  BAD: "скриншот"
`;
  }


  return `${JSON_INSTRUCTION}
TASK: Generate comprehensive API test ideas and examples for an endpoint.

INPUT:
- Project Context:
${projectContext || "(empty)"}

- Structured API input:
${inputBlock}

EXAMPLE of high-quality test ideas — follow this style:

GOOD test idea (authentication):
{
  "name": "Запит без токена авторизації",
  "request_example": "curl -X POST https://api.example.com/v1/orders -H \"Content-Type: application/json\" -d '{\"items\": [{\"product_id\": \"123\", \"qty\": 1}]}'",
  "checks": [
    "Статус відповіді: 401 Unauthorized",
    "Тіло відповіді містить поле error.code зі значенням UNAUTHORIZED",
    "Тіло відповіді НЕ містить поле order_id"
  ],
  "negative_cases": [
    "Замовлення не створюється в базі даних",
    "Відповідь не повертає дані інших користувачів"
  ]
}

GOOD test idea (schema contract):
{
  "name": "Перевірка схеми успішної відповіді",
  "request_example": "curl -X POST https://api.example.com/v1/orders -H \"Authorization: Bearer <valid_token>\" -H \"Content-Type: application/json\" -d '{\"items\": [{\"product_id\": \"123\", \"qty\": 1}], \"delivery_address\": {\"city\": \"Київ\", \"street\": \"Хрещатик\", \"zip\": \"01001\"}, \"payment_method\": \"card\"}'",
  "checks": [
    "Статус відповіді: 201 Created",
    "Поле order_id присутнє і є рядком",
    "Поле status присутнє зі значенням 'pending'",
    "Поле created_at присутнє у форматі ISO 8601",
    "Header Location містить URL створеного замовлення"
  ],
  "negative_cases": [
    "Відповідь не містить зайвих полів з чутливими даними (паролі, токени)"
  ]
```

```
}

BAD test idea — NEVER do this:
{
  "request_example": "-H \"Authorization: \"",     ← WRONG: неповний curl, не можна
запустити
  "checks": ["Перевірити відповідь сервера"],    ← WRONG: надто розмито
  "negative_cases": ["Відповідь містить order_id"] ← WRONG: це позитивна поведінка, не
негативна
}

OUTPUT JSON SCHEMA:
{
  "endpoint": {
    "method": "GET|POST|PUT|PATCH|DELETE|UNKNOWN",
    "path": "string",
    "description": "string — one sentence describing what this endpoint does"
  },
  "preconditions": ["string — what must be true before any test runs"],
  "happy_path": {
    "request_example": "string — full curl command with all headers and body",
    "checks": ["string — specific assertion: status code, field name+value, header,
response time"]
  },
  "test_ideas": [
    {
      "name": "string — short descriptive name of test scenario",
      "request_example": "string — FULL curl command (not just a fragment). REQUIRED,
never empty or partial.",
      "checks": ["string — specific: status code + field name + expected value or
format"],
      "negative_cases": ["string — what must NOT happen: no data created, no sensitive
data leaked, no wrong status"]
    }
  ],
  "test_data_suggestions": ["string — concrete example values, not generic advice"],
  "missing_information": ["string — specific question blocking complete test coverage"]
}

REQUIREMENTS:
- Generate at least 10 test ideas. Cover ALL of these areas (mandatory):
    1. Authentication — missing token → 401
    2. Authentication — expired token → 401
    3. Authentication — malformed/wrong token → 401
    4. Authorization — token of another user trying to act on someone else's resource →
403
    5. Input validation — missing required field → 422 with specific error
    6. Input validation — wrong field type → 422 with specific error
    7. Boundary values — min/max values, off-by-one (e.g. qty=0, qty=-1)
    8. Idempotency — repeat identical POST/PUT/DELETE → check no duplicate created
    9. Rate limiting — send 10+ rapid requests → expect 429 with Retry-After header
    10. Schema contract — successful response has ALL required fields with correct types

- request_example rules (CRITICAL):
    • EVERY request_example MUST be a complete, runnable curl command
    • Include: curl -X METHOD URL -H "Authorization: ..." -H "Content-Type: ..." -d
'{...}'
    • Show only the CHANGED part vs happy path (wrong token, missing field, etc.)
    • NEVER write just "-H something" or "-d something" without the full curl prefix
    • Use <valid_token>, <expired_token>, <other_user_token> as placeholders
```

```
- checks[] rules:
    • Each check = one specific assertion
    • Always include status code check first
    • For error responses: also check error.code or error.message field value
    • For success responses: check each important field name, type, and value
    • GOOD: "Статус відповіді: 422 Unprocessable Entity"
    • GOOD: "Тіло відповіді містить поле error.code зі значенням VALIDATION_FAILED"
    • GOOD: "Поле order_id присутнє і є непорожнім рядком"
    • BAD:  "Перевірити відповідь сервера" / "Відповідь коректна"

- negative_cases[] rules:
    • Describe what must NOT happen as a result of this request
    • GOOD: "Замовлення не створюється в базі даних"
    • GOOD: "Відповідь не містить поле order_id"
    • BAD:  "Відповідь містить order_id" ← this is a POSITIVE outcome, not negative

- test_data_suggestions: concrete values only.
    GOOD: "Граничне значення qty: 0, 1, 9999, 10000"
    BAD:  "Використовуйте різні значення"
`;
}
```

В.2. Чотирирівневий парсинг JSON-відповіді LLM (lib/llmClient.ts)

Функція extractAndParseJson послідовно застосовує чотири стратегії витягування JSON з текстової відповіді моделі.

```
   // lib/llmClient.ts

import { isValidJson } from "./jsonParser";

export interface LLMChatOptions {
  url: string;
  token: string;
  model: string;
  system: string;
  user: string;
  timeoutMs?: number;
}

interface LLMResponse {
  model: string;
  created_at: string;
  message: { role: string; content: string };
  done: boolean;
}

const DEFAULT_TIMEOUT_MS = 45_000;

const JSON_RETRY_MESSAGE = `Your previous response was not valid JSON. Respond with a
single JSON object only, starting with { and ending with }. No explanation, no markdown,
no code fences.`;

async function callGroqApi(
  opts: LLMChatOptions,
  messages: Array<{ role: string; content: string }>
```

```
): Promise<LLMResponse> {

  const controller = new AbortController();
  const timeoutId = setTimeout(
    () => controller.abort(),
    opts.timeoutMs ?? DEFAULT_TIMEOUT_MS
  );

  let res: Response;
  try {
    res = await fetch(opts.url, {
      method: "POST",
      headers: {
        "content-type": "application/json",
        authorization: `Bearer ${opts.token}`,
      },
      body: JSON.stringify({
        model: opts.model,
        messages,
        temperature: 0.0,
        max_tokens: 4096,
      }),
      signal: controller.signal,
    });
  } catch (err) {

    if (err instanceof Error && err.name === "AbortError") {
      throw new Error(`LLM request timed out after ${(opts.timeoutMs ??
DEFAULT_TIMEOUT_MS) / 1000}s`);
    }
    throw err;
  } finally {
    clearTimeout(timeoutId);
  }

  if (!res.ok) {
    const text = await res.text().catch(() => "");
    throw new Error(`Groq error: ${res.status} ${text}`);
  }

  const data = await res.json();
  const content = data.choices?.[0]?.message?.content ?? "";

  return {
    model: data.model ?? opts.model,
    created_at: new Date().toISOString(),
    message: { role: "assistant", content },
    done: true,
  };
}

export async function llmChat(opts: LLMChatOptions): Promise<LLMResponse> {
  const initialMessages = [
    { role: "system", content: opts.system },
    { role: "user", content: opts.user },
  ];

  const firstResponse = await callGroqApi(opts, initialMessages);
  const firstContent = firstResponse.message?.content ?? "";

  if (isValidJson(firstContent)) {
```

```
    return firstResponse;
  }

  console.warn("[QA-Assistant] First LLM response was not valid JSON, retrying...");

  const retryMessages = [
    ...initialMessages,
    { role: "assistant", content: firstContent },
    { role: "user", content: JSON_RETRY_MESSAGE },
  ];

  return callGroqApi(opts, retryMessages);
}
```

B.3. API Route з rate limiting (app/api/generate/route.ts)

Фрагмент серверного маршруту з реалізацією обмеження частоти запитів та викликом LLM-клієнта.

```
   // app/api/generate/route.ts (fragment)

import { NextResponse } from "next/server";
import type { GenerateRequest, Mode } from "@/lib/types";
import { SYSTEM_PROMPT, buildUserPrompt } from "@/lib/prompts";
import { llmChat } from "@/lib/llmClient";
import { extractAndParseJson } from "@/lib/jsonParser";

const VALID_MODES: Mode[] = ["testcases", "bugreport", "api_ideas"];

function normalizeUrl(url: string): string {
  if (!url) return "";
  if (url.startsWith("//")) return `https:${url}`;
  if (!url.startsWith("http://") && !url.startsWith("https://")) return `https://${url}`;
  return url;
}

const GROQ_API_URL = normalizeUrl(process.env.LLM_GATEWAY_URL || "");
const GROQ_API_TOKEN = process.env.LLM_GATEWAY_TOKEN || "";
const LLM_MODEL = process.env.LLM_MODEL || "llama-3.3-70b-versatile";

const rateLimitMap = new Map<string, { count: number; resetAt: number }>();
const RATE_LIMIT_MAX = 10;
const RATE_LIMIT_WINDOW_MS = 60 * 1000;

function getRateLimitKey(req: Request): string {
  const forwarded = req.headers.get("x-forwarded-for");
  const realIp = req.headers.get("x-real-ip");
  return forwarded?.split(",")[0]?.trim() || realIp || "unknown";
}

function checkRateLimit(ip: string): { allowed: boolean; remaining: number } {
  const now = Date.now();
  const entry = rateLimitMap.get(ip);

  if (!entry || now > entry.resetAt) {
    rateLimitMap.set(ip, { count: 1, resetAt: now + RATE_LIMIT_WINDOW_MS });
    return { allowed: true, remaining: RATE_LIMIT_MAX - 1 };
```

```
  }
  if (entry.count >= RATE_LIMIT_MAX) {
    return { allowed: false, remaining: 0 };
  }
  entry.count++;
  return { allowed: true, remaining: RATE_LIMIT_MAX - entry.count };
}

setInterval(() => {
  const now = Date.now();
  for (const [ip, entry] of rateLimitMap.entries()) {
    if (now > entry.resetAt) rateLimitMap.delete(ip);
  }
}, 60 * 1000);

function getErrorMessage(error: unknown, fallback: string) {
  return error instanceof Error ? error.message : fallback;
}

export async function POST(req: Request) {
  try {
    const clientIp = getRateLimitKey(req);
    const rateCheck = checkRateLimit(clientIp);

    if (!rateCheck.allowed) {
      return NextResponse.json(
         { error: { code: "RATE_LIMITED", message: "Забагато запитів. Спробуйте через
хвилину." } },
        { status: 429, headers: { "Retry-After": "60" } }
      );
    }

    const body = (await req.json()) as GenerateRequest;

    if (!body?.mode || !VALID_MODES.includes(body.mode)) {
      return NextResponse.json(
         { error: { code: "VALIDATION_FAILED", message: `Invalid mode. Must be one of:
${VALID_MODES.join(", ")}` } },
        { status: 400 }
      );
    }

    if (!body?.input?.fields) {
      return NextResponse.json(
        { error: { code: "VALIDATION_FAILED", message: "Missing input.fields" } },
        { status: 400 }
      );
    }

    if (!GROQ_API_URL || !GROQ_API_TOKEN) {
      return NextResponse.json(
         { error: { code: "LLM_UNAVAILABLE", message: "Groq API not configured (set
LLM_GATEWAY_URL and LLM_GATEWAY_TOKEN env vars)" } },
        { status: 500 }
      );
    }

       const userPrompt = buildUserPrompt(body.mode, body.project_context || "",
body.input.fields);

    let llm;
```

```
    try {
      llm = await llmChat({
        url: GROQ_API_URL,
        token: GROQ_API_TOKEN,
        model: LLM_MODEL,
        system: SYSTEM_PROMPT,
        user: userPrompt,
      });
    } catch (e: unknown) {
        console.error("[QA-Assistant] LLM call failed:", getErrorMessage(e, "Unknown
error"));
      return NextResponse.json(
       { error: { code: "LLM_UNAVAILABLE", message: getErrorMessage(e, "LLM unavailable")
} },
        { status: 502 }
      );
    }

    const rawText = llm.message?.content ?? "";

    let result: unknown;
    try {
      result = extractAndParseJson(rawText);
    } catch (parseError) {
        console.error("[QA-Assistant] JSON parsing failed. Raw (first 2000 chars):",
rawText.slice(0, 2000));
      console.error("[QA-Assistant] Parse error:", getErrorMessage(parseError, "Unknown
parse error"));
      return NextResponse.json(
        {
          error: {
            code: "INVALID_JSON",
            message: "Модель повернула некоректний JSON. Спробуйте ще раз.",
            details: { raw: rawText.slice(0, 2000) },
          },
        },
        { status: 500 }
      );
    }

    return NextResponse.json({
      mode: body.mode,
      model: llm.model,
      created_at: llm.created_at,
      result,
      raw_llm_text: rawText,
    });
  } catch (e: unknown) {
    console.error("[QA-Assistant] Internal error:", getErrorMessage(e, "Unknown error"));
    return NextResponse.json(
      { error: { code: "INTERNAL", message: getErrorMessage(e, "Internal error") } },
      { status: 500 }
    );
  }
}
```

Додаток Г − Приклад відповіді LLM у форматі JSON

У цьому додатку наведено приклад скороченої JSON-відповіді моделі llama-3.3-70b-versatile для режиму генерації тест-кейсів. Вхідні дані: функціональність «Онлайн-запис до лікаря», контекст − медична інформаційна система.


```
{
  "feature": "Онлайн запис до лікаря",
  "assumptions": [
    "Лікар має доступні часові слоти",
    "Пацієнт може авторизуватися у системі",
    "Система має коректну обробку дат та часу"
  ],
  "test_cases": [
    {
      "id": "TC-001",
      "title": "Успішна реєстрація на прийом",
      "preconditions": [
        "Пацієнт авторизований",
        "Лікар має доступні часові слоти"
      ],
      "steps": [
        "Відкрити сторінку реєстрації на прийом",
        "Обрати лікаря",
        "Вибрати доступний часовий слот",
        "Натиснути кнопку [Зареєструватися]"
      ],
      "expected": "Пацієнт отримує підтвердження реєстрації",
      "priority": "P1",
      "type": "positive",
      "tags": [
        "smoke",
        "regression"
      ]
    },
    {
      "id": "TC-002",
      "title": "Неможливість реєстрації на зайнятий слот",
      "preconditions": [
        "Пацієнт авторизований",
        "Лікар має зайнятий часовий слот"
      ],
      "steps": [
        "Відкрити сторінку реєстрації на прийом",
        "Обрати лікаря",
        "Вибрати зайнятий часовий слот",
        "Натиснути кнопку [Зареєструватися]"
      ],
      "expected": "Пацієнт отримує повідомлення про зайнятість слоту",
      "priority": "P2",
      "type": "negative",
      "tags": [
        "boundary",
        "validation"
      ]
    },
```

```
{
  "id": "TC-003",
  "title": "Скасування реєстрації на прийом",
  "preconditions": [
    "Пацієнт авторизований",
    "Пацієнт зареєстрований на прийом"
  ],
  "steps": [
    "Відкрити сторінку реєстрації на прийом",
    "Натиснути кнопку [Скасувати реєстрацію]"
  ],
  "expected": "Пацієнт отримує підтвердження скасування реєстрації",
  "priority": "P1",
  "type": "positive",
  "tags": [
    "regression",
    "ui"
  ]
},
{
  "id": "TC-004",
  "title": "Неможливість реєстрації на прийом без авторизації",
  "preconditions": [
    "Пацієнт неавторизований"
  ],
  "steps": [
    "Відкрити сторінку реєстрації на прийом",
    "Натиснути кнопку [Зареєструватися]"
  ],
  "expected": "Пацієнт отримує повідомлення про необхідність авторизації",
  "priority": "P2",
  "type": "negative",
  "tags": [
    "auth",
    "validation"
  ]
},
{
  "id": "TC-005",
  "title": "Реєстрація на прийом з датою сьогодні",
  "preconditions": [
    "Пацієнт авторизований",
    "Лікар має доступні часові слоти"
  ],
  "steps": [
    "Відкрити сторінку реєстрації на прийом",
    "Обрати лікаря",
    "Вибрати доступний часовий слот з датою сьогодні",
    "Натиснути кнопку [Зареєструватися]"
  ],
  "expected": "Пацієнт отримує підтвердження реєстрації",
  "priority": "P1",
  "type": "positive",
  "tags": [
    "smoke",
    "regression"
  ]
},
{
  "id": "TC-006",
  "title": "Реєстрація на прийом з датою завтра",
```

```
    "preconditions": [
      "Пацієнт авторизований",
      "Лікар має доступні часові слоти"
    ],
    "steps": [
      "Відкрити сторінку реєстрації на прийом",
      "Обрати лікаря",
      "Вибрати доступний часовий слот з датою завтра",
      "Натиснути кнопку [Зареєструватися]"
    ],
    "expected": "Пацієнт отримує підтвердження реєстрації",
    "priority": "P1",
    "type": "positive",
    "tags": [
      "smoke",
      "regression"
    ]
  },
  {
    "id": "TC-007",
    "title": "Неможливість реєстрації на прийом з датою <24 годин",
    "preconditions": [
      "Пацієнт авторизований",
      "Лікар має доступні часові слоти"
    ],
    "steps": [
      "Відкрити сторінку реєстрації на прийом",
      "Обрати лікаря",
      "Вибрати доступний часовий слот з датою <24 годин",
      "Натиснути кнопку [Зареєструватися]"
    ],
    "expected": "Пацієнт отримує повідомлення про неможливість реєстрації",
    "priority": "P2",
    "type": "negative",
    "tags": [
      "boundary",
      "validation"
    ]
  },
  {
    "id": "TC-008",
    "title": "Реєстрація на прийом з датою >24 годин",
    "preconditions": [
      "Пацієнт авторизований",
      "Лікар має доступні часові слоти"
    ],
    "steps": [
      "Відкрити сторінку реєстрації на прийом",
      "Обрати лікаря",
      "Вибрати доступний часовий слот з датою >24 годин",
      "Натиснути кнопку [Зареєструватися]"
    ],
    "expected": "Пацієнт отримує підтвердження реєстрації",
    "priority": "P1",
    "type": "positive",
    "tags": [
      "smoke",
      "regression"
    ]
  },
  {
```

```
            "id": "TC-009",
            "title": "Скасування реєстрації на прийом лікарем",
            "preconditions": [
              "Лікар авторизований",
              "Пацієнт зареєстрований на прийом"
            ],
            "steps": [
              "Відкрити сторінку реєстрації на прийом",
              "Натиснути кнопку [Скасувати реєстрацію]"
            ],
            "expected": "Пацієнт отримує підтвердження скасування реєстрації",
            "priority": "P1",
            "type": "positive",
            "tags": [
              "regression",
              "ui"
            ]
          },
          {
            "id": "TC-010",
            "title": "Неможливість реєстрації на прийом адміністратором",
            "preconditions": [
              "Адміністратор авторизований"
            ],
            "steps": [
              "Відкрити сторінку реєстрації на прийом",
              "Натиснути кнопку [Зареєструватися]"
            ],
            "expected": "Адміністратор отримує повідомлення про неможливість
реєстрації",
            "priority": "P2",
            "type": "negative",
            "tags": [
              "auth",
              "validation"
            ]
          }
        ],
        "coverage_notes": [
          "Covered: реєстрація на прийом, скасування реєстрації, авторизація",
          "Negative cases: 4 out of 10 (40%)",
          "Techniques: equivalence partitioning, boundary value analysis",
          "Not covered: реєстрація на прийом з датою в минулому (out of scope)"
        ],
        "missing_information": [
          "Який формат дати використовується у системі?",
          "Чи можливо реєструватися на прийом без авторизації?"
        ]
      }
```