Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра комп'ютерних наук

КВАЛІФІКАЦІЙНА РОБОТА
ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

ВЕБЗАСТОСУНОК З ПІДТРИМКОЮ СМАРТ-КОНТРАКТІВ ДЛЯ БЕЗПЕЧНОГО ВИКОНАННЯ КРИПТОТРАНЗАКЦІЙ

WEB APPLICATION WITH SMART CONTRACTS SUPPORT FOR SECURE CRYPTO TRANSACTIONS

спеціальність 122 Комп'ютерні науки

освітня програма «Комп'ютерні науки»

Виконав: здобувач вищої освіти
групи КН-41
Сорочук Юрій Вікторович

_____
(підпис)

Керівник: асистент
Неділько Ольга Володимирівна

_____
(підпис)

Кваліфікаційну роботу
допущено до захисту
«___» _____ 2026 р.
Гарант освітньої програми:
д.пед.н., професор
Тулашвілі Юрій Йосипович

_____
(підпис)

Луцьк – 2026 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет комп'ютерних та інформаційних технологій
Кафедра комп'ютерних наук
Ступінь вищої освіти: бакалавр
Галузь знань: 12 Інформаційні технології
Спеціальність: 122 Комп'ютерні науки
Освітня програма: «Комп'ютерні науки»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____Валерій ЛІЩИНА
«__»_____2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧА
ПЕРШОГО (БАКАЛАВРСЬКОГО) РІВНЯ ВИЩОЇ ОСВІТИ

Сорочука Юрія Вікторовича
(прізвище, ім'я, по-батькові)

1. Тема кваліфікаційної роботи «Вебзастосунок з підтримкою смарт-контрактів для безпечного виконання криптотранзакцій».

Керівник асистент Неділько Ольга Володимирівна,
затверджені наказом закладу вищої освіти від *« 16 » січня 2026 р.* № *490/01-02*

2. Строк подання здобувачем вищої освіти кваліфікаційної роботи *«__» _____ 2026 р.*

3. Вихідні дані до роботи: *Опубліковані зарубіжні та вітчизняні дослідження в галузі децентралізованих інформаційних систем, концепції Web 3.0 та блокчейн-технологій, методи проектування гібридних архітектрук dApp, механізми безпечної синхронізації on-chain та off-chain даних, сучасні фреймворки та інструментарій для розробки клієнтської частини (React, Vite), а також середовища, мови та бібліотеки інтеграції смарт-контрактів у мережу Ethereum (Solidity, Ethers.js, MetaMask, Supabase).*

4. Зміст розрахунково-пояснювальної записки: *Аналіз сучасного стану проблеми, існуючих методів і засобів її розв'язання, аналіз і вибір засобів проєктування, опис функціонального наповнення об'єкта проектування, розробка й обґрунтування системного наповнення, оцінка ергономічних та надійністних параметрів проектованої системи, функціонально структурна схема роботи об'єкта проектування.*

5. Перелік графічного матеріалу: *24 рисунки, презентація.*

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз проблеми за темою роботи та постановка завдань дослідження	*Неділько О. В.*		
Теоретичне дослідження	*Неділько О. В.*		
Практична реалізація	*Неділько О. В.*		
Спеціальні розрахунки	*Неділько О. В.*		
Показник запозичень тексту	%		
Інструментальна перевірка	*Кошелюк В. А.*		
Нормоконтроль	*Сачук В. О.*		
Гарант ОПП	*Тулашвілі Ю. Й.*		

7. Дата видачі завдання *« 16 » січня 2026 р.*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	*Провести огляд літературних джерел по темі кваліфікаційної роботи*	*до 17.02.2026 р.*	
2	*Провести аналіз загальної проблеми і вибір напрямків дослідження*	*до 28.02.2026 р.*	
3	*Розробити функціональну схему роботи програмного продукту*	*до 12.03.2026 р.*	
4	*Описати засоби розробки об'єкта проектування*	*до 26.03.2026 р.*	
5	*Практична реалізація об'єкта проектування*	*до 30.04.2026 р.*	
6	*Провести спеціальні розрахунки*	*до 18.05.2026 р.*	
7	*Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедру*	*до 02.06.2026 р.*	

Здобувач вищої освіти _____ Юрій СОРОЧУК

Керівник роботи _____ Ольга НЕДІЛЬКО

АНОТАЦІЯ

Сорочук Ю. В. Вебзастосунок з підтримкою смарт-контрактів для безпечного виконання криптотранзакцій. Рукопис.

Кваліфікаційна робота бакалавра ОП «Комп'ютерні науки». Луцький національний технічний університет. Луцьк, 2026.

Кваліфікаційна робота бакалавра складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків.

У першому розділі виконано аналіз предметної області, наведено приклади існуючих рішень та визначено основні функціональні потреби користувачів. Другий розділ містить специфікацію функціональних і нефункціональних вимог до інформаційної системи, її логічну структуру та ключові компоненти. У третьому розділі було здійснено розробку вебзастосунку з підтримкою смарт-контрактів для безпечного виконання криптотранзакцій на базі бібліотеки React та мови програмування Solidity, доповнивши цей технологічний стек хмарною платформою Supabase та бібліотекою Ethers.js для взаємодії з криптогаманцем MetaMask. У четвертому розділі наведено спеціальні розрахунки, зокрема здійснено ергономічну оцінку функціональних компонентів програмного продукту з використанням математичних моделей, розраховано час на виконання повного циклу замовлення в Web3-системі з урахуванням мережевих затримок та фізичного введення даних, а також обґрунтовано високу ефективність спроєктованого інтерфейсу. У висновках узагальнено результати виконаної роботи та визначено перспективи подальшого розвитку системи.

Ключові слова: Web3, вебзастосунок, блокчейн, смарт-контракт, децентралізація, Ethereum, Solidity, React, Supabase, MetaMask, ергономіка, зручність.

ANNOTATION

Yurii Sorochuk. Web application with smart contract support for secure crypto transactions. Manuscript.

Bachelor’s qualification paper in the study program "Computer Science". Lutsk National Technical University. Lutsk, 2026.

The bachelor’s qualification paper consists of an introduction, four chapters, conclusions, a list of references, and appendices. The first chapter provides an analysis of the subject area, presents examples of existing solutions, and defines the main functional user needs. The second chapter contains a specification of functional and non-functional requirements for the information system, its logical structure, and key components. The third chapter presents the development of a web application with smart contract support for secure crypto transactions based on the React library and the Solidity programming language, supplementing this technology stack with the Supabase cloud platform and the Ethers.js library for interaction with the MetaMask crypto wallet. The fourth chapter presents special calculations, including an ergonomic assessment of the software product’s functional components using mathematical models, a calculation of the execution time for a full order cycle in the Web3 system considering network latencies and physical data entry, as well as justification for the high efficiency of the designed interface. The conclusions summarize the results of the work performed and define the prospects for further system development.

Keywords: Web3, web application, blockchain, smart contract, decentralization, Ethereum, Solidity, React, Supabase, MetaMask, ergonomics, usability.

ЗМІСТ

ВСТУП

У сучасному світі питання децентралізації фінансових потоків та забезпечення безпеки цифрових транзакцій набувають критичного значення. Підприємства та окремі розробники перебувають у постійному пошуку надійних технологічних рішень, які дозволять мінімізувати посередництво сторонніх організацій та забезпечити прозорість виконання угод. Особливого значення в цьому контексті набуває технології блокчейну та смарт-контрактів, які здатні автоматизувати складні фінансові операції, виключаючи людський фактор та забезпечуючи незмінність даних у режимі реального часу.

Сучасні вебзастосунки часто стикаються з проблемою інформаційної асиметрії, технологічні можливості Web3 випереджають якість користувацьких інтерфейсів, через що складність взаємодії з протоколами перетворюється на бар'єр для масового користувача. Існуючі платформи для фрилансу та надання послуг зазвичай працюють як централізовані системи з прихованою фінансовою логікою. Тому розробка децентралізованих систем, які поєднують надійність блокчейну з інтуїтивно зрозумілою ергономікою, є надзвичайно актуальним завданням для забезпечення довіри та фінансової незалежності споживачів.

Метою кваліфікаційної роботи бакалавра є розробка вебзастосунку з підтримкою смарт-контрактів для безпечного виконання криптотранзакцій шляхом інтеграції фронтенд-технологій React, хмарної бази даних Supabase та блокчейн-протоколів мережі Ethereum.

Об'єктом дослідження є процес проектування та функціонування децентралізованих інформаційних систем, що забезпечують проведення криптотранзакцій у мережі Ethereum.

Предметом дослідження є методи та архітектурні підходи до побудови гібридних вебзастосунків, інтеграція смарт-контрактів Solidity з клієнтським інтерфейсом React та синхронізація on-chain та off-chain даних у реальному часі.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

– проаналізувати сучасний стан проблеми, існуючі аналоги та методи забезпечення довіри в децентралізованих середовищах;

– сформувати та обґрунтувати вибір шляхів, технологій та засобів вирішення поставленого завдання;

– розробити функціональну схему та архітектуру вебзастосунку, що забезпечує коректну взаємодію між клієнтським інтерфейсом та блокчейн-мережею;

– реалізувати смарт-контракт для безпечного виконання фінансових операцій та інтегрувати його з інтерфейсом через гаманець MetaMask;

– розробити вебзастосунок з інтуїтивно зрозумілим інтерфейсом, що забезпечує прозору історію транзакцій та зручне управління профілем користувача.

Наукова новизна роботи полягає у розробці гібридної архітектури застосунку, де реалізовано безшовну синхронізацію між реляційною базою даних та блокчейном через механізм збереження хешів транзакцій, що дозволяє забезпечити верифікованість off-chain даних без надлишкових витрат комісій. Також вперше застосовано класичні методи ергономічної оцінки, закони Хіка та Фіттса та модель KLM-GOMS до архітектури Web3-інтерфейсу, що дозволило кількісно довести оптимальність розробленого користувацького шляху.

Практична цінність роботи полягає у створенні готового до експлуатації програмного продукту, який забезпечує прозорість фінансових взаємовідносин між сторонами, виключає маніпуляції з даними оплати та може бути використаний як база для створення сучасних сервісів з автоматизації послуг на базі Web3.

Результати дослідження були представлені на міжнародній науково-практичній конференції «Сучасні інформаційні технології: від теорії до практики» (додаток А).

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз сучасного стану проблеми

Спостерігаючи за темпом розвитку інформаційних технологій та визначаючи проблематику об’єкту дослідження, відзначається чітка трансформація цифрового ринку. Чим досконаліше стають блокчейн-технології, тим складнішою здається взаємодія між користувачем та кінцевим продуктом. Сьогодні децентралізація часто викликає страх та недорозуміння в суспільстві, перетворюючись на закрите середовище, де фінансові операції стають складним завданням, доступним лише вузькому колу фахівців. Власне кажучи, сучасні вебзастосунки мають повернутися до своєї первісної мети – бути зручним інструментом, а не джерелом когнітивного навантаження. У статті, в пам’ять про винахідника всесвітньої павутини Тіма Бернерса-Лі, згадували його слова: «Веб – це більшою мірою соціальне творіння, ніж технічне. Я проєктував його задля соціального ефекту – щоб допомогти людям працювати разом, а не як технічну іграшку» [1]. Це твердження стало головним орієнтиром при проєктуванні вебзастосунку, адже кінцеве прагнення полягає у створенні соціально значущого продукту, а не просто чергового додатка для бізнесу.

У процесі дослідження предметної області було помічено тенденцію серед розробників щодо тотального використання конвеєрних шаблонів, які часто генерує штучний інтелект заради економії ресурсів. Хоча такий підхід значно пришвидшує реліз, він породжує бездушні інтерфейси, де за технологічністю ховається абсолютна відсутність емпатії до замовника. Натомість було обрано шлях індивідуального проєктування, де візуальна естетика веб-дизайну не конфліктує з жорсткою структурою коду, а підкреслює його надійність. У роботі було свідомо відмовлено від швидких рішень на користь поєднання зручності бібліотеки React та швидкості Supabase, що дозволило створити оболонку, яка плавно занурює клієнта в екосистему Ethereum, не лякаючи його складною термінологією.

Безумовно, Web 3.0 розглядається як простір реального повернення права власності людині. Це не просто складні алгоритми, а нова ідеологія володіння даними та активами. Як зазначає Кріс Діксон, партнер венчурної компанії a16z та один із провідних ідеологів та інвесторів у Web3: «Web3 – це інтернет, яким володіють розробники та користувачі, координований за допомогою токенів» [2]. У розробленому вебзастосунку цей принцип реалізується через кожен виклик смарт-контракту, який відбувається прозоро та автономно. Коли користувач здійснює оплату в тестовій мережі через MetaMask, повністю знімається відчуття тягаря блокчейн-протоколів. Систему було спроєктовано так, щоб крипто-транзакції сприймалися як природний та безпечний процес, де довіра до системи базується не на ілюзіях, а на перевіреному коді.

Крім того, варто згадати про інформаційну асиметрію. Сьогодні розрив між тими, хто розуміє механіку крипто-транзакцій, і тими, хто просто хоче отримати послугу, стає новою формою цифрової нерівності. У цьому вбачається серйозний соціальний виклик, і є всі підстави вважати, що без адаптування інтерфейсів для користувача Web 3.0 ризикує залишитися відомим тільки для ентузіастів, так і не ставши тим справедливим інтернетом, про який мріють ідеологи. Проведений аналіз предметної області підтверджує, що оптимізація та адаптація технологій є не просто трендом, а необхідністю. Істинна інновація полягає у здатності розробника зробити складне простим, орієнтуючись при цьому на програмну візію організації Web3 Foundation, покликання якої, за їхніми словами, полягає: «В реалізації Web 3.0 – децентралізованого та справедливого інтернету, у якому користувачі самостійно контролюють власні дані, ідентичність та долю» [3]. Запропонований підхід доводить, що вебзастосунок повинен бути зручним помічником, який полегшує процеси життєдіяльності, гарантуючи при цьому повну фінансову незалежність та безпеку в децентралізованому середовищі. Тільки через подолання технологічного бар’єру стає можливим відкрити справжній потенціал Web 3.0 для широкого загалу.

1.2 Аналіз аналогічних програм, баз даних, систем, обґрунтування вибраного напряму та вибір методів рішення

Перед розробкою власного продукту необхідно поглянути на ринок тверезо, без ілюзій щодо першості у вирішенні подібних задач. Аналіз готових рішень – це не просто формальність, а спосіб знайти ті самі слабкі місця, де існуючі системи втрачають користувача. Дослідження ринку було розпочато з вивчення платформ для надання послуг у сфері розробки. Першою платформою варто виділити Upwork, який сьогодні фактично є еталоном для фрилансу [4]. Було проаналізовано їхню систему безпечних угод та виявлено, що головною перевагою тут є високий рівень довіри до бренду. Проте технічно ця довіра базується на закритих базах даних, де користувач не має жодного доступу до логіки прийняття рішень. Також було звернуто увагу на Fiverr [5], який пропонує максимально швидкий шлях від пошуку до покупки. Але під час порівняння цих двох систем було помічено спільну ваду: вони обидві працюють як чорні скриньки, де фінансова логіка прихована для користувача.

На противагу централізованим гігантам було розглянуто Gitcoin [6]. Це найбільш релевантний аналог із децентралізованого світу, який вивчався з точки зору механіки оплат. Перевага Gitcoin у тому, що він використовує відкритий код смарт-контрактів, що повністю виключає маніпуляції з грошима. Проте під час аналізу недоліків було виявлено, що система занадто прив’язана до специфічних розробницьких задач і має дуже високий поріг входження через перевантажений деталями інтерфейс. У цьому вбачається головна перешкода: технологічна досконаліть Gitcoin конфліктує з його ергономікою. Саме це стало спонукальним фактором для пошуку методів, які дозволять залишити надійність блокчейну, але прибрати складність для замовника.

Окрім згаданих лідерів, було вивчено досвід платформ на кшталт Toptal [7], яка позиціонує себе як мережа для елітарних фахівців. Хоча вона гарантує високу якість, її модель закритих спільнот створює штучне обмеження доступу до ринку та надвисокі ціни, що є неприйнятним для малого та

середнього бізнесу. Також було проаналізовано Braintrust [8], що намагається впровадити елементи володіння платформою через токени, проте навіть там зберігається відчуття великого агрегатора, де індивідуальність конкретної студії розмивається серед тисяч однотипних профілів.

Аналізуючи ці аналоги, було зроблено висновок, що сучасним вебзастосункам для продажу послуг з розробки бракує саме прямого діалогу з користувачем та прозорості фінансового розрахунку. Більшість розробників сьогодні змушені або миритися з величезними комісіями централізованих майданчиків, або будувати власні системи оплати, які часто виглядають ненадійно в очах замовника. Було помічено, що існує величезна прірва між простим сайтом-візиткою та складним блокчейн-протоколом. Замовнику бракує інструменту, який би дозволив відчути безпеку угоди на рівні коду, але при цьому не змушував би проходити складні процедури реєстрації чи тривалі періоди виведення коштів, що зазвичай тривають тижнями у Web 2.0 системах. Обґрунтування обраного напряму базується на вирішенні цієї системної кризи.

Існують вагомі підстави вважати, що ідеальне рішення повинно поєднувати автономність смарт-контрактів із легкістю звичайного веб-сайту. Основна проблема конкурентів – це надмірна концентрація влади та інформації в руках власника платформи. Натомість головне прагнення полягає у створенні середовища, де технологія виступає не наглядачем, а гарантом справедливості. Це зумовило вибір гібридного методу проєктування, де надійність децентралізованих фінансів стане фундаментом для зручного користувацького інтерфейсу спроєктованого застосунку.

1.3 Аналіз і вибір засобів проектування вебзастосунку та смарт-контракту

Проєктування системи вимагає не просто набору випадкових програм, а створення цілісної екосистеми, де кожен інструмент доповнює інший. Вибір було розпочато з пошуку мов програмування, які стали б фундаментом для

фронтенд- та бекенд-частин. Під час аналізу варіантів для клієнтської сторони було здійснено порівняння Angular, Vue.js та React. Angular пропонує сувору структуру, яка більше підходить для масштабних корпоративних систем [9], проте для гнучкого проєкту цей фреймворк виявився занадто громіздким. З іншого боку, Vue.js приваблює своєю лаконічністю та легким порогом входу, що робить його зручним для швидкого візуального проєктування [10]. Проте було помічено суттєвий недолік – інструментальна підтримка Vue.js все ще дещо поступається в розмаїтті готових рішень для специфічної інтеграції з децентралізованими мережами. Саме тому вибір було зупинено на бібліотеці React. Саме React забезпечує необхідний рівень гнучкості та має величезну кількість Web3-бібліотек, які дозволяють безперешкодно взаємодіяти зі смарт-контрактами [11]. Що стосується серверної логіки, то замість використання Python або Go було обрано Node.js у зв’язці з інструментом збірки Vite. Це дозволяє підтримувати високу швидкість розробки, де кожна зміна в коді миттєво відображається в браузері, що критично для збереження загальної продуктивності.

Особливе місце в проєктуванні займає розробка смарт-контрактів. Було вивчено можливість використання мови Vyper, яка позиціонує себе як безпечніша альтернатива, або навіть Rust для мереж на кшталт Solana. Проте для досягнення поставленої мети було обрано Solidity. Solidity – це мова з величезною спільнотою та найкращою документацією, яка дозволяє без зайвих проблем описувати складну логіку фінансових операцій [12]. У ній вбачається ідеальний баланс між функціональністю та легкістю деплою. Більше того, Solidity найприродніше взаємодіє з гаманцем MetaMask, що значно спрощує процес авторизації транзакцій.

Важливим етапом проєктування став також вибір програмного засобу для забезпечення зв’язку між клієнтським інтерфейсом та блокчейн-мережею. У цій сфері розглядалися два найбільш розповсюджені рішення: Web3.js та Ethers.js. Бібліотека Web3.js є перевіреним часом інструментом із великою кількістю напрацювань, проте її визнано занадто громіздкою та складною в управлінні

провайдерами для динамічного проєкту. Натомість було обрано Ethers.js. Ця технологічна бібліотека пропонує більш сучасний та лаконічний підхід до взаємодії зі смарт-контрактами, забезпечуючи при цьому високу стабільність роботи. Головна перевага полягає в чіткому розділенні функцій провайдера та підписанта (signer), що ідеально відповідає концепції безпечного управління транзакціями через MetaMask. Використання Ethers.js дозволяє підтримувати легкість програмного коду та гарантувати точне виконання асинхронних запитів до мережі Ethereum без зайвого навантаження на браузер користувача.

Питання збереження інформації зумовило порівняння різних типів баз даних. Було розглянуто класичні відомі всім реляційні системи на кшталт MySQL та NoSQL-рішення, як-от MongoDB. Традиційний підхід вимагає підняття власного сервера та постійної турботи про його безпеку та масштабування. У сучасних умовах це визнано нераціональним. Натомість було обрано хмарну платформу Supabase. Вона надає потужність PostgreSQL у хмарі, що забезпечує надійність збереження метаданих проєктів. Вагомим аргументом є те, що Supabase має hook-систему для React та дозволяє легко синхронізувати дані, які недоцільно записувати в дорогий блокчейн. Ця платформа використовується як міст, що поєднує швидкість звичайного вебу та безпеку смарт-контрактів.

Що стосується середовища розробки, завдання було розділено на два напрями. Для загальної роботи з кодом проєкту було обрано Visual Studio Code. Здійснювалося порівняння з IntelliJ IDEA та Sublime Text, але VS Code виявився оптимальним рішенням: він легкий, швидкий та має тисячі плагінів для підтримки React та Solidity і вище згаданих рішень. Проте для безпосередньої роботи з смарт-контрактами було залучено Remix IDE. Хоча існують консольні фреймворки на кшталт Hardhat або Truffle, перевагу було віддано Remix для етапу прототипування. Його головна відмінність – це миттєвий візуальний фідбек. З’являється можливість одразу побачити результати виконання функцій контракту, не витрачаючи час на написання складних тестів у терміналі.

Нарешті, було обрано MetaMask як основний крипто-гаманець для розроблюваної системи. Під час розгляду аналогів, як-от Trust Wallet чи Coinbase Wallet, було визначено, що саме MetaMask має найкращу інтеграцію з Remix IDE. Роботу було організовано так, щоб тестова мережа (наприклад, Sepolia) розгорталася в кілька кліків. Це дозволяє проводити безпечні випробування фінансової логіки без витрат реальних коштів. Таким чином, було створено комплекс засобів, де VS Code, Remix, React та Supabase працюють як єдиний механізм, дозволяючи швидко пройти шлях від ідеї до готового до використання вебзастосунку.

1.4 Постановка завдання на кваліфікаційну роботу бакалавра

Підсумовуючи результати аналізу предметної області та технічного інструментарію, було поставлено завдання трансформувати складні протоколи на прикладний та зрозумілий сервіс, де висока технологічність блокчейну використовується виключно для зручності людини. Безперечно, головне прагнення полягає в доведенні того, що децентралізовані фінанси можуть бути доступними для кожного, не створюючи бар'єрів у вигляді надмірної складності. Відповідно, було сформульовано конкретні цілі, які мають бути досягнуті в результаті виконання роботи:

– дослідити архітектурні принципи Web 3.0 та на основі аналізу розробити концепцію м'якого занурення користувача в децентралізоване середовище, мінімізуючи когнітивне навантаження при роботі з крипто-транзакціями;

– спроєктувати гібридну архітектуру вебзастосунку, яка поєднує гнучкість бібліотеки React, надійність хмарної бази даних Supabase та прозорість смарт-контрактів у мережі Ethereum;

– розробити та розгорнути смарт-контракт на мові Solidity в тестовій мережі, забезпечивши автоматизовану та безпечну оплату послуг токеном Ethereum, без залучення сторонніх фінансових посередників;

– реалізувати інтуїтивно зрозумілий інтерфейс для користувача, який включає функціональне портфоліо з готовими проєктами, сторінку контактів та персоналізований профіль із надійною реєстрацією через електронну пошту;

– інтегрувати систему управління крипто-гаманцем MetaMask, надавши користувачу можливість не лише підключатися до платформи, а й гнучко керувати даними – змінювати або відключати гаманець без втрати доступу до сервісу;

– налаштувати механізм збереження та відображення даних, що дозволить користувачам переглядати історію покупок завдяки інтеграції з базою даних Supabase;

– провести комплексну перевірку працездатності системи, підтвердивши надійність зв’язку між фронтенд-частиною та блокчейн-протоколом, а також оцінивши зручність реалізованого клієнтського шляху.

З огляду на вищезазначене, стає очевидним, що виконання цих завдань дозволить трансформувати абстрактні ідеї децентралізації у зрозумілий та функціональний сервіс, де довіра між сторонами базується не на обіцянках, а на перевіреному програмному коді. Впровадження такого інноваційного підходу суттєво змінить саму парадигму взаємодії в цифровому просторі. Спроєктована екосистема стане своєрідним мостом між традиційними користувачами та новітніми інструментами блокчейн-мережі. Власне кажучи, безкомпромісний фокус на ергономіці та прозорості автоматично виводить подібні рішення на якісно новий рівень – туди, де складні криптографічні процеси відбуваються глибоко в системі, залишаючись непомітними під час роботи. При цьому гарантується абсолютний захист персональних активів.

Крім того, цілком закономірно припустити, що результати даного дослідження слугуватимуть міцним концептуальним фундаментом для подальшого масштабування. Запропоновані архітектурні патерни легко адаптуються під спеціалізовані маркетплейси, платформи вільного найму чи системи автоматизованого розподілу коштів.

РОЗДІЛ 2
СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ

2.1 Обґрунтування вибору шляхів, технологій (алгоритмів) і засобів вирішення поставленого завдання

Проєктування системи було розпочато з глибокого аналізу архітектурної моделі, оскільки успіх Web3-застосунку повністю залежить від балансу між швидкістю реального світу та безпекою децентралізованих мереж. Цілком очевидно, що сьогодні неможливо створити ефективний сервіс, не розуміючи чіткої межі між off-chain та on-chain процесами. Концептуальна схема взаємодії, представлена на рисунку 2.1, відображає фундаментальний поділ, який використовується в розроблюваному проєкті. На цій схемі чітко простежується класична модель переходу інформації із «реального світу» до «світу блокчейну».

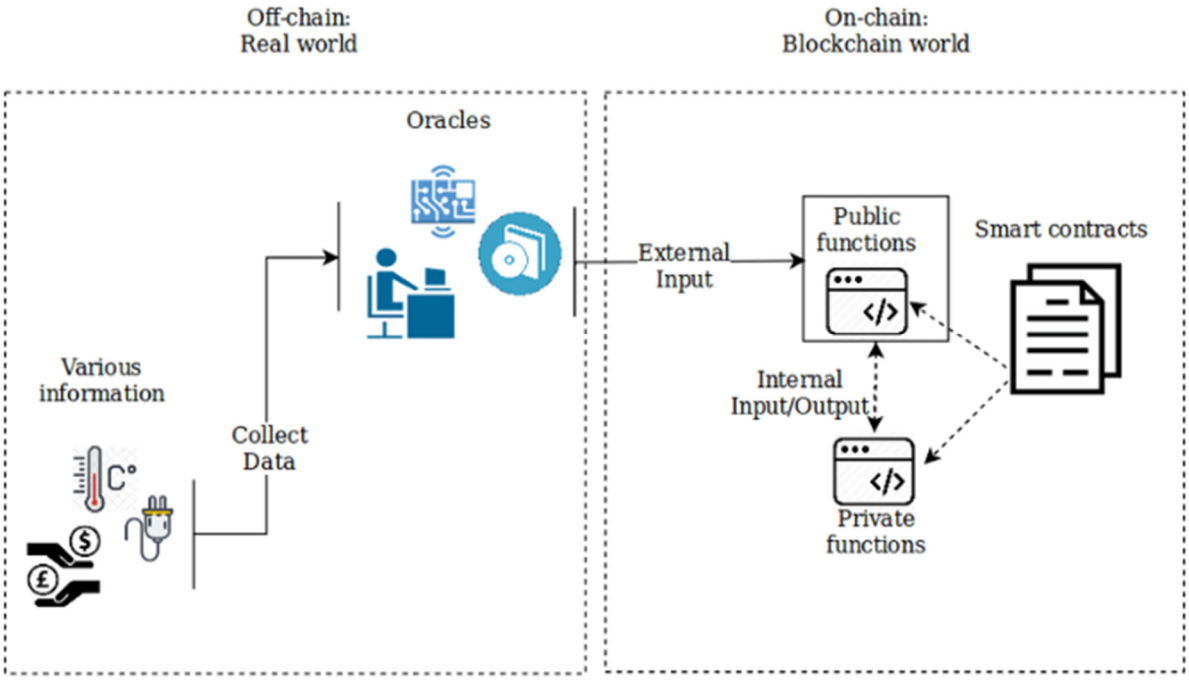

Рисунок 2.1 – Процес комунікації між on-chain та off-chain [13]

Як можна спостерігати на рисунку, дані проходять послідовний шлях від збору інформації (collect data) до її обробки та передачі через зовнішні входи (external input) у смарт-контракти. Цю модель для вебзастосунку інтерпретовано

наступним чином: усе розмаїття даних користувача (various information) – від профілів до метаданих портфоліо – надійно зберігається в off-chain середовищі на базі платформи Supabase. Таке архітектурне рішення дозволяє повністю уникнути надмірних витрат газу та забезпечити миттєву реакцію користувацького інтерфейсу. У цьому випадку роль оракулів та механізмів збору даних покладено на клієнтську логіку React, яка структурує запити та готує їх до безпосередньої взаємодії з децентралізованим блокчейн-протоколом.

Для об’єктивної оцінки обраного підходу вбачається за доцільне провести чітку паралель між існуючими архітектурними рішеннями. Безумовно, постає питання, чи не було б простіше перенести абсолютно всі дані безпосередньо в блокчейн. З технічного погляду – це цілком можливо, проте з прагматичного – такий крок став би критичним для загальної зручності клієнта. Повністю децентралізоване збереження кожного окремого елемента інтерфейсу чи текстового опису перетворило б звичайну зміну статусу замовлення на складну фінансову операцію з тривалим очікуванням консенсусу. Саме тому обраний варіант реалізації є раціональним компромісом, де висока швидкість звичних інтернет-технологій поєднується з непохитною чесністю децентралізованих мереж.

Окрему увагу приділено безпосередньому механізму потрапляння даних у «blockchain world». На рисунку цей етап позначений як external input, що в межах проєкту реалізується через взаємодію бібліотеки Ethers.js та криптогаманця MetaMask. Такий метод було обрано завдяки можливості безпечно викликати публічні функції смарт-контракту, забезпечуючи фінансову прозорість та чесність оплати. Порівнюючи цей підхід із традиційними схемами, головна перевага вбачається в тому, що основна бізнес-логіка, яка потребує високого рівня довіри, інкапсульована в приватних та публічних функціях на мові Solidity. Це дозволяє створити надійний захисний контур, де жодна зовнішня маніпуляція не здатна змінити умови вже підписаної та валідованої транзакції.

Водночас слід наголосити на надійності збереження тієї інформації, що залишається поза ланцюгом блоків. Потужність реляційної бази даних

PostgreSQL, інтегрованої в хмарну платформу Supabase, використовується не лише заради швидкодії. Це дозволяє забезпечити сувору цілісність та зв'язаність інформації про замовлення, яка, хоч і не потребує обов'язкового фінансового підтвердження в Ethereum, є критично важливою для ведення повноцінної історії в профілі клієнта. Власне кажучи, безпека даних у системі базується на ефективному багаторівневому підході: конфіденційність та структурованість у Supabase надійно доповнюються публічною незмінністю в блокчейні.

З упевненістю можна стверджувати, що такий гібридний метод, де чітко розмежовуються real world (Supabase) та blockchain world (Ethereum), є найбільш раціональним для успішного вирішення поставленого завдання. Як демонструє рисунок 2.1, смарт-контракти не існують ізольовано – вони потребують постійного структурованого входу інформації. Саме тому для побудови високошвидкісного фронтенду було залучено інструмент Vite, який стає ефективним засобом для збору та передачі даних. Таким чином, обґрунтування обраного підходу базується не лише на загальній зручності розробки, а й на суворому дотриманні перевіреної часом архітектурної моделі, що гарантує стабільність, цілісність та безпеку всього децентралізованого сервісу.

2.2 Функціонально-структурна схема роботи об'єкта проектування (блок-схеми алгоритмів, UML діаграми класів, діаграми компонентів)

Головна мета системи визначається як створення надійного середовища для взаємодії між замовниками та вебзастосунком, де кожна фінансова операція підкріплена прозорістю блокчейн-технологій. Для вичерпного опису архітектури застосунку необхідно проаналізувати його з двох боків: через динаміку руху даних (інформаційні потоки) та через статичну організацію програмного коду (компонентна структура). Основними задачами, які вирішуються в межах цього проєкту, є забезпечення інтелектуальної підтримки процесу замовлення, розмежування прав доступу до персоналізованих даних та гарантування цілісності фінансових транзакцій.

Під час аналізу функціональних характеристик системи виокремлюється декілька критичних рівнів. Початкове введення інформації та реєстрація здійснюються через хмарну платформу Supabase, що дозволяє забезпечити віддалений доступ за протоколом TCP/IP з будь-якої точки світу. Було реалізовано механізм контролю введеної інформації на рівні фронтенд-валідації та серверних правил бази даних, що надійно гарантує цілісність PostgreSQL. Для наочної візуалізації логіки взаємодії цих рівнів було розроблено структурно-функціональну схему, представлену на рисунку 2.2.

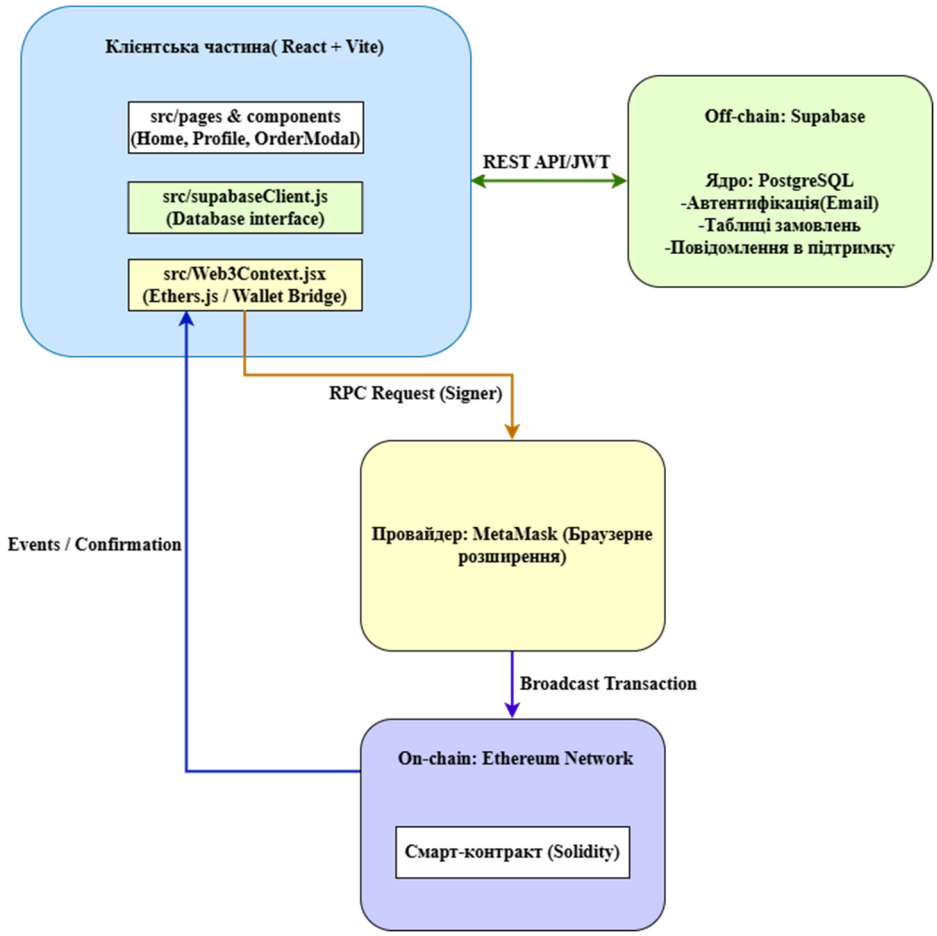

Рисунок 2.2 – Структурно-функціональна схема інформаційних потоків

Джерело: розроблено автором

На рисунку чітко простежується, що ядром клієнтської частини є React-компоненти, які через спеціалізовані модулі supabaseClient.js та Web3Context.jsx успішно керують потоками інформації. Особливе місце також посідає забезпечення надійного захисту даних, де залучено сучасні стандарти

авторизації (JWT), які дозволяють розмежовувати права доступу таким чином, щоб історія оплат та особисті налаштування в профілі були доступні виключно власнику автентифікованого сеансу. Цілком логічно, що в межах розробки застосовано архітектуру, де логіка Web3-з'єднання повністю інкапсульована в окремому контексті, завдяки чому підтримується стабільний зв'язок із MetaMask та мережею Ethereum. Взаємодія з off-chain сховищем відбувається через RESTful-інтерфейс, де кожен запит супроводжується JWT-токеном, який, власне кажучи, дозволяє виводити знайдену інформацію миттєво – без тривалого очікування підтвердження від блокчейн-вузлів.

Процес безпосереднього оформлення замовлення та проведення оплати відображено у вигляді блок-схеми алгоритму, який беззаперечно слугує точним відображенням закладеної бізнес-логіки, де кожен крок має чітко визначену умову та кінцевий результат.

Зазначений процес розпочинається із обов'язкової перевірки стану підключення гаманця в модулі OrderModal.jsx, після чого, за умови успішного встановлення з'єднання, системою формується відповідний запит до смарт-контракту на мові Solidity. В архітектурі передбачено чітке розгалуження алгоритму залежно від отриманої відповіді провайдера, тому в разі відхилення транзакції користувачем система миттєво повертає інформаційне сповіщення. З цього логічно випливає, що такий детермінований контроль гарантує абсолютну консистентність гібридної платформи. Оффчейн-сховище зберігає виключно ті операції, які вже здобули строгий математичний консенсус у децентралізованій мережі. Це фундаментально унеможливлює навіть мінімальну розсинхронізацію між реальним фінансовим станом рахунку та його візуальним відображенням у клієнтському інтерфейсі. При успішному підписанні ініціюється запуск механізму очікування події (Event), і як тільки підтвердження остаточно отримано, алгоритм в автоматичному режимі звертається до Supabase для фіксації метаданих замовлення, повністю замикаючи наскрізний цикл обслуговування (рис. 2.3).

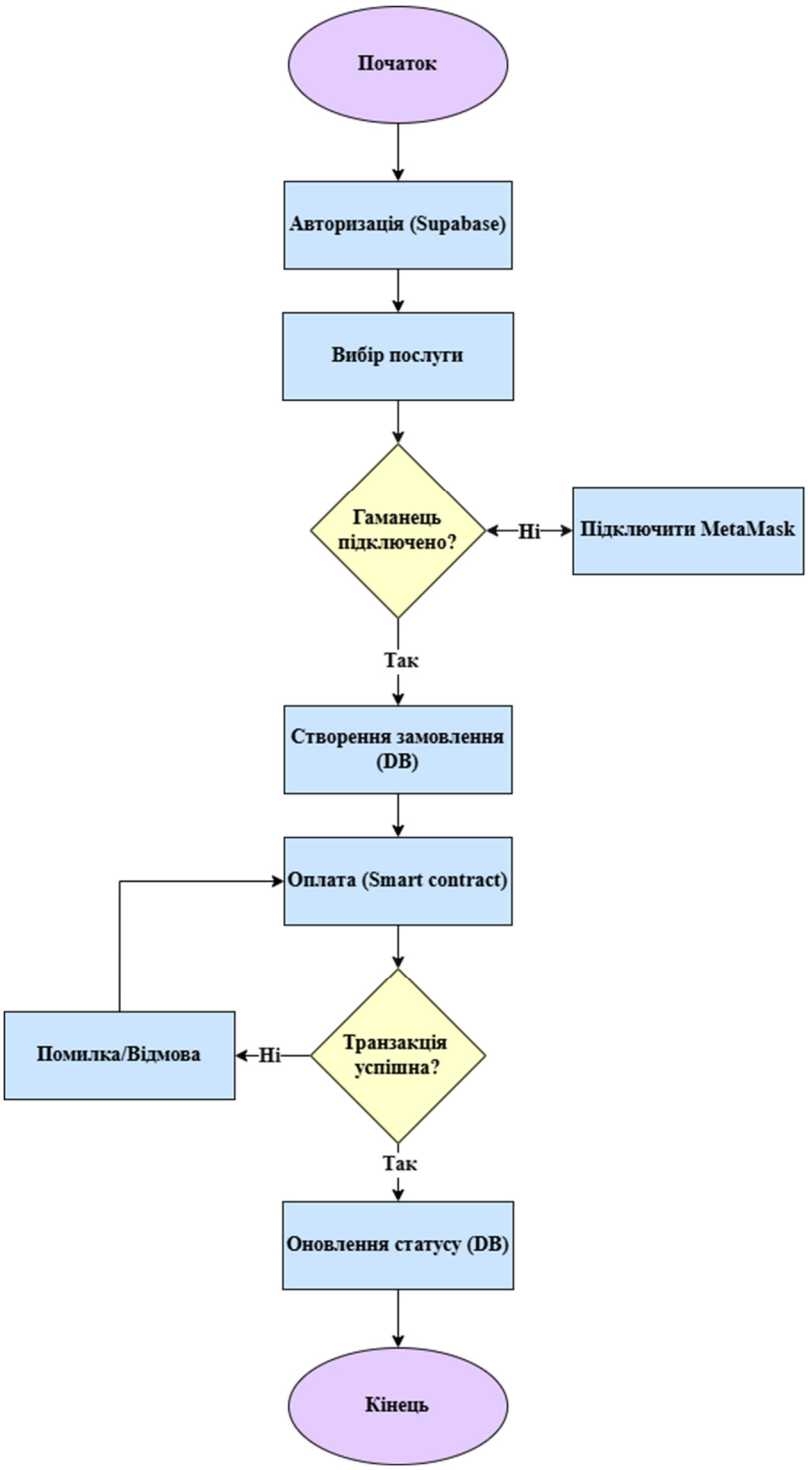

Рисунок 2.3 – Блок-схема алгоритму процесу оформлення та оплати замовлення

Джерело: розроблено автором

Проте для розуміння того, як ці алгоритми та потоки даних розподілені всередині самого програмного коду, була розроблена UML-діаграма компонентів, що наведена на рисунку 2.4. Вона демонструє статичну архітектуру проєкту та залежності між конкретними файлами.

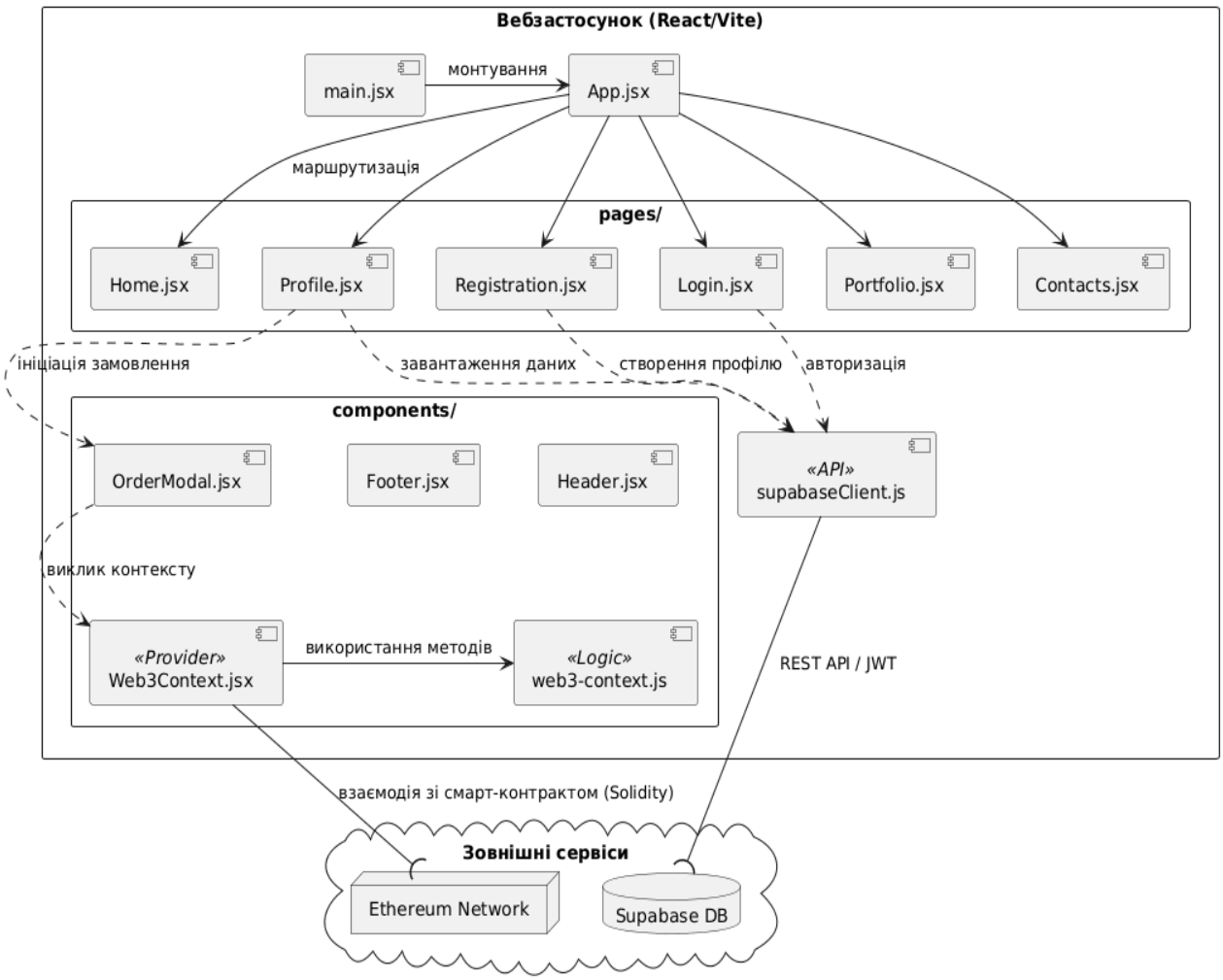

Рисунок 2.4 – UML-діаграма програмних компонентів системи

Джерело: розроблено автором

Було візуалізовано ієрархічну структуру застосунку, побудованого на базі Vite. На схемі чітко простежується, як головний модуль App.jsx керує маршрутизацією між сторінками з папки pages/, тоді як Web3Provider огортає всю систему, надаючи глобальний доступ до Web3-контексту. Такий підхід дозволяє ефективно відокремити візуальну логіку сторінок (як-от Profile.jsx чи Portfolio.jsx) від службових модулів supabaseClient.js та web3-context.js. Вбачається цілком очевидним, що використання UML-діаграми дозволяє наочно довести високу модульність розробки, де кожен програмний компонент має чітко визначену зону відповідальності.

Таким чином, запропонована функціонально-структурна модель повністю задовольняє вимоги до безпеки, швидкодії та надійності інформаційної системи.

Було створено комплексне технологічне рішення, де програмна структура ідеально відповідає логіці інформаційних потоків, забезпечуючи користувачу зручний інструмент управління в реальному часі.

2.3 Створення моделі бази даних

Проєктування бази даних починається з аналізу існуючих технологій. Вибір СКБД має ґрунтуватися на балансі між гнучкістю та надійністю. Сучасні системи керування базами даних прийнято класифікувати за типом моделі даних, яку вони підтримують:

– реляційні (SQL), що базуються на суворих табличних структурах та зв’язках (PostgreSQL, MySQL, MS SQL Server). Вони забезпечують високу цілісність даних завдяки відповідності принципам ACID;

– документо-орієнтовані (NoSQL), які зберігають дані у форматі гнучких документів (MongoDB, CouchDB), що зручно для неструктурованої інформації;

– ключ-значення, які оптимізовані для високої швидкодії при простих запитах (Redis, DynamoDB);

– графові, що спеціалізуються на складних взаємозв’язках між об’єктами (Neo4j).

Враховуючи особливості бізнес-логіки розроблюваного проєкту, де критично важливим є розмежування прав доступу та точність фінансових транзакцій, було обрано PostgreSQL на базі платформи Supabase. Це реляційна СКБД, що вирізняється підтримкою складних типів даних, потужним механізмом Row Level Security (RLS) та можливістю масштабування [14]. Її технічна характеристика включає повну підтримку стандартів SQL, високу стійкість до збоїв та вбудовані засоби авторизації, що ідеально інтегруються зі створеним React-фронтендом. Інфологічна модель бази даних фокусується на поняттях «Користувач», «Профіль», «Платіж» та «Підтримка». Відповідно, було спроєктувано логічну структуру, де кожне поняття має чітко визначені атрибути, що наочно відображено на рисунку 2.5 у вигляді каскадної моделі.

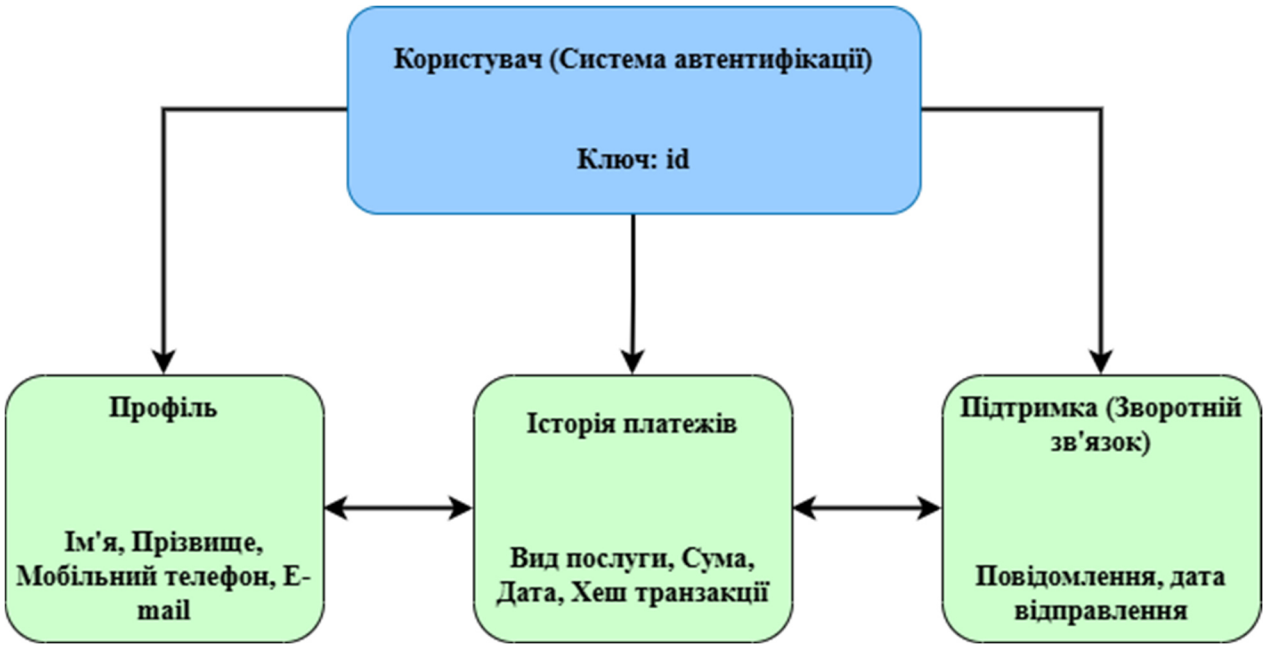

Рисунок 2.5 – Логічна структура взаємозв’язків об’єктів бази даних

Джерело: розроблено автором

Саме такий ієрархічний підхід дозволяє реалізувати принцип єдиного входу, де ідентифікатор користувача (id) стає кореневим елементом для всіх інших таблиць. Вбачається цілком очевидним, що дані профілю, історія платежів та звернення до підтримки логічно витікають із сутності авторизованого користувача, що забезпечує чистоту архітектури та відсутність дублювання інформації. Перехід до фізичного рівня вимагає обов’язкового врахування специфіки PostgreSQL. Відповідно, було розроблено фізичну структуру, де використовуються типи даних text для описів та timestamptz для точної фіксації часу подій. Візуалізація цієї структури представлена на ERD-діаграмі.

Цілком логічно розглядати рисунок нижче як технічну карту, де кожен зв’язок та тип даних має своє функціональне призначення. Центральним вузлом усієї системи є таблиця auth.users, яка інтегрована в ядро Supabase (рис. 2.6). Саме від цієї захищеної сутності каскадно розгалужуються всі інші реляційні зв’язки, що дозволяє в автоматичному режимі верифікувати операційні права кожного окремого облікового запису.

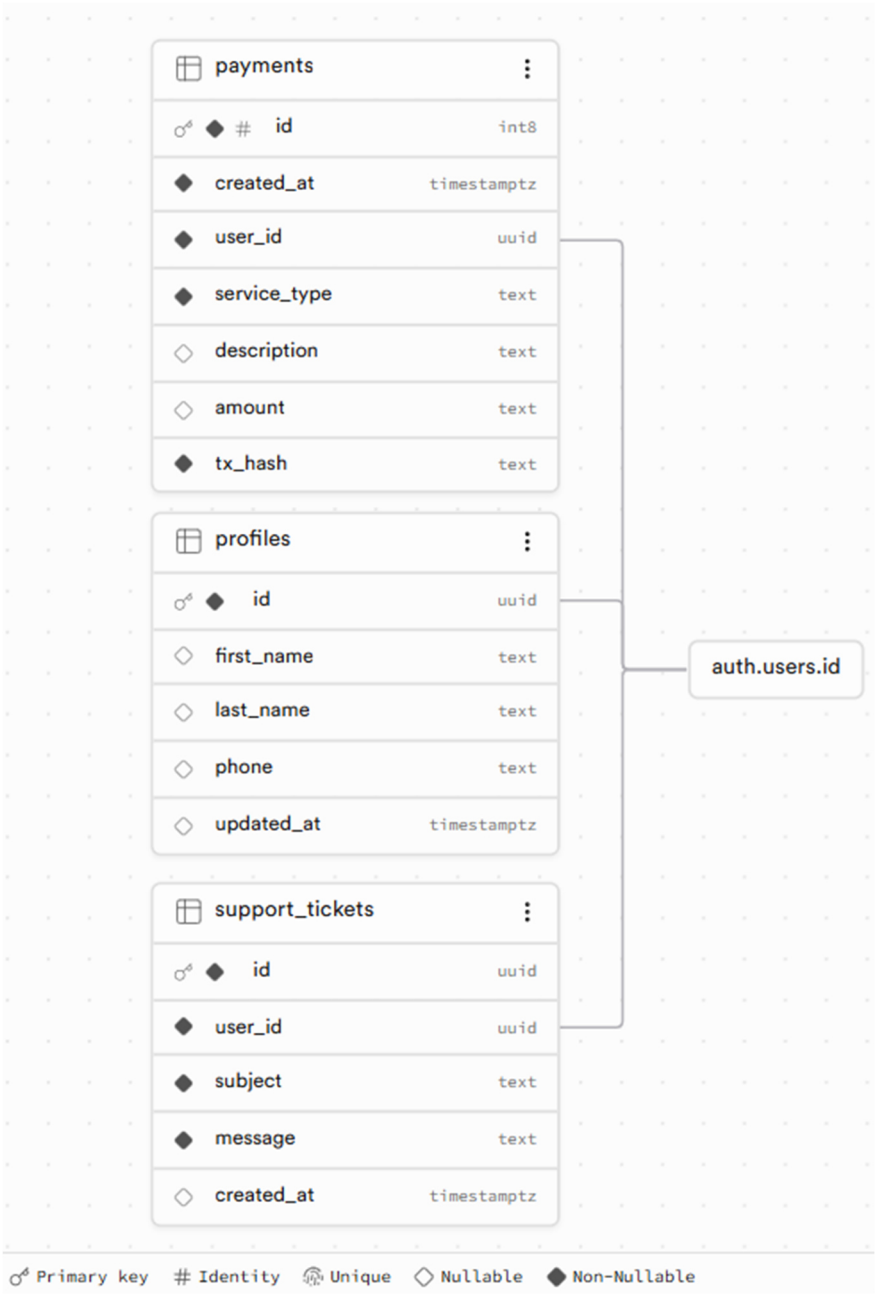

Рисунок 2.6 – ERD-діаграма фізичної структури бази даних

Джерело: розроблено автором

Для полів id у всіх таблицях використовується тип uuid. Це надійно гарантує, що кожне замовлення чи профіль отримують унікальний у часі та просторі маркер, який абсолютно неможливо підробити чи випадково продублювати.

Таблиця profiles розширює базову сутність користувача, додаючи персональні атрибути, такі як ім’я, прізвище та номер телефону. На діаграмі

чітко простежується, що поле id у цій таблиці одночасно є первинним ключем та зовнішнім ключем до auth.users, що забезпечує абсолютну нерозривність даних.

Таблиці payments та support_tickets пов’язані з користувачем через відповідні відношення. Це означає, що один клієнт може мати необмежену історію замовлень та звернень, при цьому система завжди точно ідентифікує, кому саме вони належать. Цілком закономірно, що до таблиці оплат було спеціально додано поле tx_hash задля забезпечення прозорості, яка визначалася як одна з головних цілей системи: відтепер будь-який запис у базі можна миттєво зіставити з конкретною транзакцією в блокчейні.

Очевидно, що такий підхід до моделювання, де всі дані консолідовані навколо єдиної точки автентифікації, дозволяє реалізувати максимально зручну сторінку профілю, де історія оплат та звернення відображаються паралельно з персональними даними. Водночас для кожної операції застосовуються типи timestamptz, що дає змогу вести точний хронологічний аудит дій користувача.

Таким чином, ERD-діаграма демонструє не просто набір таблиць, а готову до масштабування архітектуру. Було створено систему, де логіка збереження даних повністю відповідає програмній структурі, забезпечуючи цілісність, прозорість та безпеку на кожному етапі роботи застосунку.

Крім того, варто зазначити, що подібна організація реляційних зв’язків створює ідеальне підґрунтя для впровадження суворих політик безпеки на рівні окремих записів (Row Level Security). Завдяки чіткій та безперервній прив’язці кожної сутності до унікального ідентифікатора авторизованого клієнта, з’являється можливість налаштувати правила доступу безпосередньо в ядрі PostgreSQL. Це фундаментально гарантує, що навіть у випадку гіпотетичної програмної вразливості на фронтенд-частині застосунку сторонні особи не зможуть отримати дозвіл на перегляд чужої історії транзакцій чи конфіденційної інформації. Спроєктована база даних виступає не лише як пасивне інформаційне сховище, а як активний та незалежний ешелон захисту, що органічно доповнює бездоганну безпеку децентралізованих протоколів.

РОЗДІЛ 3
РОЗРОБКА ВЕБЗАСТОСУНКУ З ПІДТРИМКОЮ СМАРТ-КОНТРАКТІВ

3.1 Практична реалізація об’єкта проектування. Побудова блок-схем модулів програми

Переходячи до безпосередньої практичної реалізації вебзастосунку, варто усвідомлювати, що будь-який програмний код є лише логічним продовженням закладеної раніше архітектури. Саме тому розробку було розпочато з налаштування професійних середовищ, де чітко розмежовується робота над клієнтським інтерфейсом та блокчейн-протоколом. Для створення візуальної частини та налаштування локального сервера було обрано легкий, але надзвичайно зручний редактор Visual Studio Code. Цей інструмент дозволив безшовно інтегрувати сучасний фреймворк Vite, який забезпечує миттєве оновлення компонентів React під час написання коду. Відбулася свідома відмова від застарілих конвеєрних рішень із зосередженням на модульній побудові системи. Як зазначалося раніше, під час проєктування UML-діаграми програмних компонентів, ієрархія файлів починається з головного модуля App.jsx, який керує всією маршрутизацією. Цей статичний каркас став тим міцним фундаментом, на якому розпочалося поступове нарощування інтерактивних елементів.

Паралельно з побудовою фронтенд-частини було розгорнуто середовище Remix IDE. Цей крок виявився абсолютно необхідним для швидкого прототипування фінансової логіки на мові Solidity. Оскільки децентралізовані мережі не прощають помилок, практичні експерименти розпочалися саме з написання та розгортання смарт-контракту в тестовій мережі Sepolia. Це дало змогу безпечно перевірити механізми прийому платежів через розширення MetaMask, абсолютно не ризикуючи реальними цифровими активами.

Коли базові компоненти інтерфейсу були готові, а смарт-контракт успішно скомпільований, відбувся перехід до етапу синхронізації. Було підключено хмарну платформу Supabase для забезпечення надійного збереження off-chain

даних, таких як персональні профілі та історія замовлень. Як проілюстровано на структурно-функціональній схемі інформаційних потоків, яка була задекларована раніше, зв’язок між React-додатком та базою даних PostgreSQL здійснюється через спеціалізований модуль supabaseClient.js за допомогою REST API та JWT-токенів. Важливо зазначити, що ці токени залишаються прихованими, оскільки їхнє генерування та перевірка інтегровані безпосередньо в саму структуру Supabase. Тобто ці процеси відбуваються в автономному режимі, тоді як на рівні розробки залишається лише створювати SQL-запити для взаємодії із застосунком. Цілком закономірно, що така комплексна зв’язка технологій React, Supabase, Solidity та Ethers.js дозволила втілити концепцію гібридного dApp у реальні рядки програмного коду.

Аналізуючи безпосередній процес конструювання, виникає потреба розбити застосунок на чіткі логічні шари. Першим і найважливішим кроком є вивчення архітектурного хребта клієнтської частини, яка консолідує всі ізольовані модулі в єдину цілісну систему. Розгалужена структура сучасного dApp вимагає централізованого управління переходами, щоб користувач не відчував затримок під час навігації. Плавна зміна контексту без перезавантаження сторінки є базовою основою комфортної взаємодії. Власне кажучи, саме цю логіку було закладено в основу головного модуля, який ефективно керує потоками даних та відображенням візуальних елементів.

Як уже зазначалося, ієрархія розробленого застосунку повністю розгортається з кореневого файла App.jsx. У лістингу 3.1 наведено його програмну реалізацію, яка наочно демонструє декларативний підхід до побудови маршрутизації та інтеграції Web3-контексту.

Лістинг 3.1 – Програмний код головного модуля App.jsx

```
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";
import Header from "./components/Header";
import Footer from "./components/Footer";
import Home from "./pages/Home";
import Portfolio from "./pages/Portfolio";
import Contacts from "./pages/Contacts";
import Login from "./pages/Login";
```

```
import Registration from "./pages/Registration";
import Profile from "./pages/Profile";
import { Web3Provider } from "./components/Web3Context";
import ScrollToTop from "./components/ScrollToTop";
function App() {
  return (
    <Web3Provider>
      <Router>
        <ScrollToTop />
        <div className="app-container">
          <Header />
          <main>
            <Routes>
              <Route path="/" element={<Home />} />
              <Route path="/portfolio" element={<Portfolio />} />
              <Route path="/contacts" element={<Contacts />} />
              <Route path="/login" element={<Login />} />
              <Route path="/registration" element={<Registration />} />
              <Route path="/profile" element={<Profile />} />
            </Routes>
          </main>
          <Footer />
        </div>
      </Router>
    </Web3Provider>
  );
}
export default App;
```

кінець лістингу 3.1

Розглядаючи наведену структуру, видно, що весь цифровий простір застосунку огорнутий у глобальний компонент Web3Provider. Це дозволяє зробити стан підключеного крипто-гаманця доступним на будь-якому рівні вкладеності компонентів. Окрему роль відіграє модуль ScrollToTop, який виконує важливе ергономічне завдання – примусово повертає вікно перегляду в нульові координати під час зміни маршруту, ліквідуючи неприємний ефект збереження позиції прокрутки при переході між довгими сторінками портфоліо. У лістингу наведений перелік розроблюваних сторінок, але для більшого розуміння структури варто звернути увагу на рисунок 3.1.

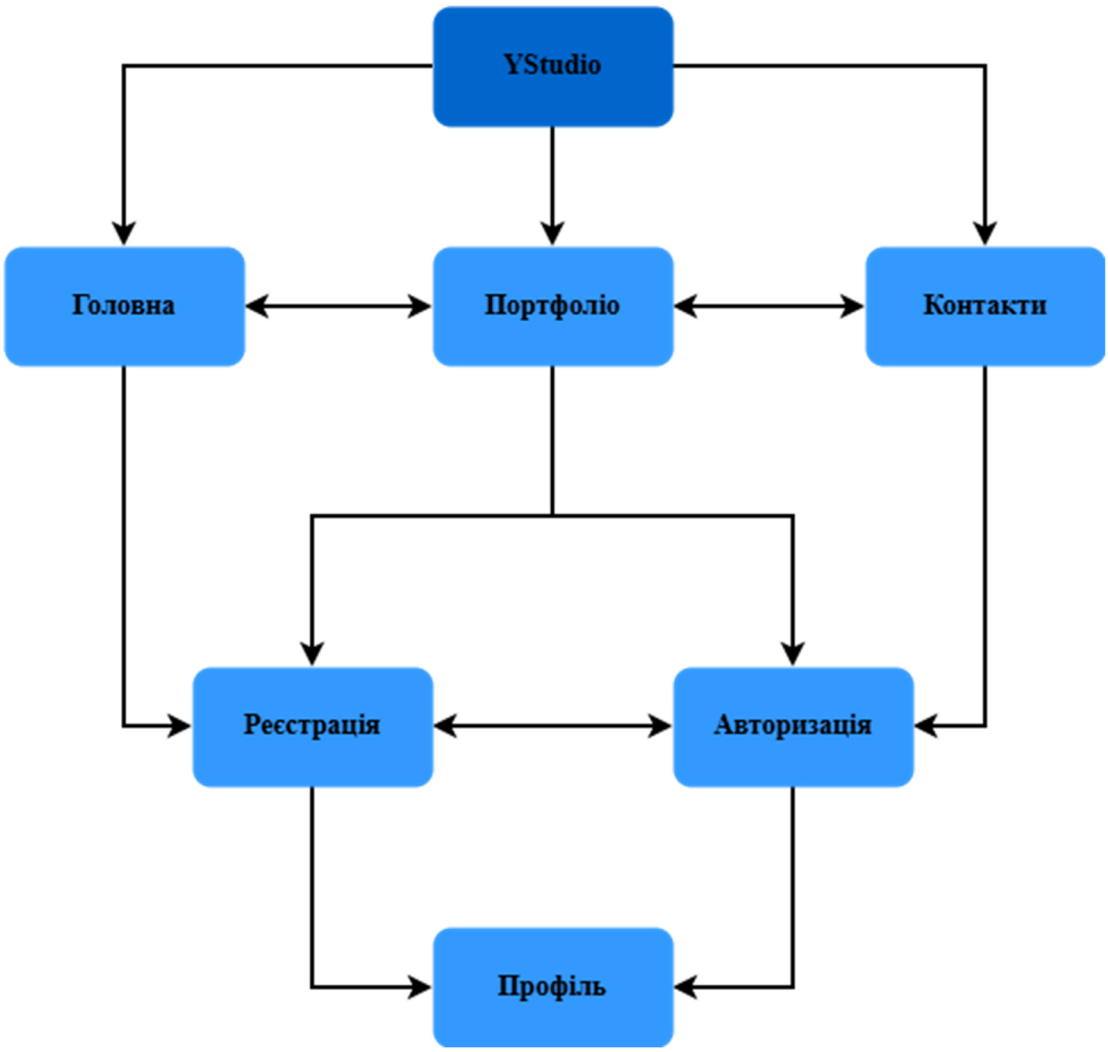

Рисунок 3.1 – Структурна схема вебзастосунку

Джерело: розроблено автором

На рисунку вище візуалізовано структурну схему сторінок вебзастосунку YStudio, яка в подальшому підлягатиме детальному розбору з фокусом на кожному програмному модулі у відповідних лістингах. Взаємодія клієнта з платформою ініціюється через базові інформаційні сторінки. Кожна з них інкапсулює чітко ізольовану бізнес-логіку.

Модуль головної сторінки Home.jsx виступає первинною точкою контакту. Саме тут було імплементовано асинхронне відстеження сесії користувача за допомогою вбудованого методу supabase.auth.onAuthStateChange. Не викликає сумнівів той факт, що такий архітектурний підхід дає змогу динамічно трансформувати візуальний стан інтерфейсу. Якщо відвідувач не пройшов процедуру автентифікації, система превентивно та м’яко блокує ініціалізацію

модального вікна замовлення послуги. При цьому генерується відповідне інформаційне попередження. Власне кажучи, цей механізм слугує надійним бар'єром, який ефективно захищає глибинну логіку dApp від некоректних або несанкціонованих запитів. Безпосередня програмна реалізація описаного функціоналу наведена у лістингу 3.2.

Лістинг 3.2 – Фрагмент коду, що відповідає за моніторинг сесії та валідацію доступу користувачів

```
import { useState, useEffect } from "react";
import { supabase } from "../supabaseClient";
import OrderModal from "../components/OrderModal";

const Home = () => {
  const [isModalOpen, setIsModalOpen] = useState(false);
  const [user, setUser] = useState(null);

  useEffect(() => {
    const checkUser = async () => {
      const { data: { session } } = await supabase.auth.getSession();
      setUser(session?.user ?? null);
    };
    checkUser();

    const { data: { subscription } } = supabase.auth.onAuthStateChange((_event, session) => {
      setUser(session?.user ?? null);
    });

    return () => subscription.unsubscribe();
  }, []);

  const handleOrderClick = () => {
    if (!user) {
      alert("Будь ласка, увійдіть в акаунт, щоб замовити послугу через блокчейн.");
      return;
    }
    setIsModalOpen(true);
};
```

кінець лістингу 3.2

Переглядаючи наведений фрагмент коду, можна побачити, що використання хука useEffect дозволяє організувати безперервний моніторинг

стану автентифікації без надлишкового навантаження на систему. Метод onAuthStateChange створює активну підписку на події зміни сесії, яка автоматично анулюється при розмонтуванні компонента завдяки виклику функції subscription.unsubscribe(). Безперечно, такий підхід забезпечує максимальну синхронізацію клієнтського інтерфейсу із хмарним сховищем. Автоматичне відсікання неавторизованих запитів у методі handleOrderClick за допомогою умовної перевірки є надійною гарантією стабільності системи. Ця логічна ізоляція абсолютно необхідна. Вона повністю запобігає виникненню критичних помилок під час спроби передачі порожнього об’єкта користувача у функціонал децентралізованого гаманця.

Далі користувач зазвичай переходить до вивчення представлених робіт у модулі Portfolio.jsx. Реалізація цієї сторінки спирається на внутрішній стан React (activeCategory), який здійснює миттєве фільтрування масиву проєктів безпосередньо на клієнтському рівні. Відбулася свідома відмова від повторних запитів до сервера при перемиканні категорій. З цього логічно випливає безпрецедентна швидкість відклику інтерфейсу, оскільки повністю нівелюється мережева затримка. У лістингу 3.3 наведено програмну реалізацію цього модуля в частині деструктуризації станів та алгоритму селекції даних.

Лістинг 3.3 – Фрагмент коду логіки фільтрації в Portfolio.jsx

```
import { useState } from “react”;
const Portfolio = () => {
  const [activeCategory, setActiveCategory] = useState(“all”);
  const portfolioItems = [
    { id: 1, category: “website”, img: “/images/33echno.jpg”, title: “Технокряк”, desc: “Веб-сайт” },
  ];
  const filteredItems = activeCategory === “all”
    ? portfolioItems
    : portfolioItems.filter((item) => item.category === activeCategory);
  return (
    <main>
      <section className=”filters”>
        <h1 className=”visually-hidden”>Наші роботи</h1>
        <ul className=”list-filters”>
          <li>
            <button
```

```
              type=”button”
              className={`filter ${activeCategory === “all” ? “active-
filter” : “”}`}
              onClick={() => setActiveCategory(“all”)}
            >
              Усі
            </button>
          </li>
        </ul>
        <ul className=”list-portfolio”>
          {filteredItems.map((item) => (
            <li key={item.id} className=”filter-block active”>
              <img src={item.img} alt={item.title} className=”photo” />
              <h3 className=”filter-caption”>{item.title}</h3>
              <p className=”filter-description”>{item.desc}</p>
            </li>
          ))}
        </ul>
      </section>
    </main>
  );
};
export default Portfolio;
```

кінець лістингу 3.3

Тут спостерігається висока обчислювальна лаконічність взаємодії внутрішніх механізмів React. Розрахунок значення константи filteredItems відбувається абсолютно декларативно під час кожного нового циклу рендерингу, що дозволяє повністю ліквідувати часові затримки. Особлива увага була приділена процесу циклічного розгортання даних за допомогою методу map, де базовим інструментом оптимізації виступає спеціальний службовий атрибут key. Кожен згенерований вузол списку отримує обов’язковий ідентифікатор key={item.id}. Це рішення є критично важливим для стабільної роботи віртуального DOM-дерева, оскільки воно дозволяє алгоритму узгодження React точково відстежувати структурні зміни в пам’яті й уникати повторного рендерингу та повного перемальовування всього графічного масиву при зміні категорій.

Безперечно, динамічне підставлення властивостей ітератора безпосередньо в атрибути тегу img демонструє всі переваги дата-орієнтованого проєктування інтерфейсів. Замість жорсткої прив’язки до статичного контенту,

система адаптивно зчитує відносний шлях до графічного файла та його текстовий опис з об’єкта item. Це забезпечує бездоганну гнучкість коду при подальшому масштабуванні портфоліо. Водночас наявність динамічного опису в атрибуті alt гарантує високий рівень доступності розробленого інтерфейсу та його коректну інтеграцію з екранними читцями.

Завершує базовий інформаційний блок сторінка Contacts.jsx, де було спроєктувано форму підписки на e-mail розсилку та представлено загальну інформацію про компанію. У лістингу 3.4 наведено критично важливий фрагмент логіки цього компонента, який відповідає за життєвий цикл сторінки та первинне перехоплення користувацьких метрик.

Лістинг 3.4 – Фрагмент коду що відповідає за обробку подій у Contacts.jsx

```
import { useEffect } from "react";
import "../styles/contacts.css";
const Contacts = () => {
  useEffect(() => {
    window.scrollTo(0, 0);
  }, []);
  const handleSubscribe = (e) => {
    e.preventDefault();
    const email = e.target.email.value;
    alert(`Ми надіслали підтвердження на пошту: ${email}`);
  };
// ... декларативна розмітка інтерфейсу
```

кінець лістингу 3.4

У лістингу спостерігається висока інженерна доцільність застосування базових інструментів React. Цілком закономірно, що використання хука useEffect із порожнім масивом залежностей дозволяє повністю нівелювати неприємний ефект пам’яті скролу при переході між різними маршрутами, примусово повертаючи вікно перегляду в нульові координати строго один раз під час монтування компонента.

Водночас функція handleSubscribe забезпечує повний контроль над поведінкою екранної форми. Виклик методу e.preventDefault(), як відомо, жорстко блокує стандартне перезавантаження вкладки браузера, що дає змогу

асинхронно перехопити введені дані без порушення цілісності поточного сеансу. Отже, було збережено максимальну імерсивність користувацького досвіду, що повністю узгоджується із загальною структурою інформаційних потоків системи, яка була описана раніше.

Наступним критично важливим архітектурним вузлом спроєктованої інформаційної системи є модуль реєстрації нових користувачів, реалізований у компоненті Registration.jsx. Безперечно, надійність будь-якої бази даних безпосередньо залежить від якості вхідних метрик, які приймаються на рівні клієнтського інтерфейсу. Саме тому цей модуль було спроєктувано як комплексний багаторівневий фільтр, де користувацький ввід піддається суворим криптографічним та логічним перевіркам ще до моменту відправлення мережевого запиту. У лістингу 3.5 наведено розширену програмну структуру цього модуля, яка наочно демонструє систему станів, методи управління вводом та двоетапний алгоритм збереження облікових даних.

Лістинг 3.5 – Фрагмент коду що відповідає за модуль валідації та реєстрації користувача

```
const handlePhoneChange = (e) => {
  let value = e.target.value;
  if (!value.startsWith("+380")) value = "+380";
  setFormData({ ...formData, phone: value });
};
const handleSubmit = async (e) => {
  e.preventDefault();
  if (formData.password !== formData.confirmPassword) {
    alert("Паролі не співпадають!");
    return;
  }
  setLoading(true);
  const { data: authData, error: authError } = await supabase.auth.signUp({
    email: formData.email,
    password: formData.password,
  });
  if (authError) {
    alert("Помилка реєстрації: " + authError.message);
    return setLoading(false);
  }
  if (authData.user) {
    const { error: profileError } = await
supabase.from("profiles").insert([
```

```
      {
        id: authData.user.id,
        first_name: formData.firstName,
        last_name: formData.lastName,
        phone: formData.phone,
      },
    ]);
    if (!profileError) {
      alert("Реєстрація успішна! Ви можете відразу увійти до свого
акаунту.");
      navigate("/login");
    }
  }
  setLoading(false);
  };
```

кінець лістингу 3.5

У наведеному коді чітко простежується специфіка реалізації клієнтської валідації, яка базується на поєднанні декларативних станів React та нативних інструментів браузера. Насамперед було розроблено спеціалізований метод handlePhoneChange, який виконує роль інтелектуального обмежувача. Під час спроби користувача стерти або модифікувати початкові символи поля, умовна конструкція миттєво блокує зміну стану, примусово повертаючи міжнародний телефонний префікс.

Не менш важливим етапом валідації є перевірка цілісності паролів безпосередньо всередині функції handleSubmit. Логічне зіставлення полів password та confirmPassword було свідомо винесено на найперший рядок обробника подій, щоб у разі банальної друкарської помилки користувача миттєво перервати виконання скрипта за допомогою директиви return. Такий підхід дозволяє повністю уникнути марних асинхронних звернень до мережі.

Коли клієнтські фільтри успішно пройдено, алгоритм переходить до виконання двоетапного транзакційного комплексу, що гарантує надійне збереження інформації у хмарній інфраструктурі Supabase. Спочатку викликається метод supabase.auth.signUp, який реєструє особу у службовій схемі автентифікації, де пароль обов’язково піддається алгоритмам одностороннього хешування. Тільки після успішного отримання унікального ідентифікатора

user.id, який виступає первинним ключем, системою ініціюється другий крок – виклик інструкції .insert() для користувацької таблиці profiles. Ці дві ключові операції пов’язані на основі суворого реляційного зв’язку, що цілком і повністю відповідає задекларованій раніше фізичній моделі бази даних. Запропонований механізм гарантує абсолютну консистентність даних, де кожна off-chain сутність профілю жорстко прив’язана до криптографічно захищеного акаунту авторизації.

Логічним завершенням процедури ідентифікації виступає процес авторизації вже зареєстрованих користувачів, який було інкапсульовано у спеціалізованому модулі Login.jsx. Оскільки архітектурна суть цього етапу багато в чому дублює концепцію реєстрації, відбулася свідома відмова від громіздких клієнтських перевірок на фронтенді, довіривши основну валідаційну роботу виключно хмарному серверу. Існують вагомі підстави вважати, що повторне нагромадження логіки лише перевантажило б інтерфейс. На відміну від створення акаунта, на цьому етапі система виконує одну чітку функцію – звіряє введені дані із відповідними записами у захищеній схемі СКБД. Відповідно, у лістингу 3.6 наведено стислий фрагмент цього алгоритму, який детально демонструє обробку поточної сесії.

Лістинг 3.6 – Фрагмент коду що відповідає за логіку авторизації користувача

```
const handleLogin = async (e) => {
  e.preventDefault();
  setLoading(true);
  const { data, error } = await supabase.auth.signInWithPassword({
    email: email,
    password: password,
  });
  if (!error && data.user) {
    alert("Вхід успішний!");
    navigate("/profile");
  } else {
    alert("Помилка входу: " + error.message);
  }
  setLoading(false);
};
```

кінець лістингу 3.6

Метод supabase.auth.signInWithPassword асинхронно транслює введену пару логіна та пароля у хмару, де сервер самостійно звіряє криптографічні хеші. Якщо верифікація проходить успішно, вбудовані механізми клієнтської бібліотеки автоматично генерують JWT-токен та записують параметри активної сесії у локальне сховище браузера. Відповідно, було застосовано метод Maps("/profile") для миттєвого перенаправлення верифікованої особи на сторінку профілю, що повністю виключає тривалі затримки інтерфейсу. Таким чином, забезпечено надійний захист off-chain даних від несанкціонованого доступу, створюючи стабільну платформу для подальшого розгортання Web3-інструментарію.

Після успішної авторизації користувач автоматично отримує доступ до центрального хабу розробленої системи – сторінки особистого кабінету Profile.jsx. Під час проєктування цього модуля головне прагнення полягало у створенні гнучкого цифрового середовища, яке одночасно оперує традиційними реляційними метаданими та відображає поточний стан децентралізованого простору. Існують усі підстави вважати, що кабінет не повинен бути статичною вітриною. Саме тому архітектура відображення цієї сторінки повністю базується на реактивних станах та умовному відображенні блоків розмітки. Для забезпечення автономної роботи особистого кабінету користувача було реалізовано комплексний React-компонент, який відповідає за безпечну синхронізацію локальних даних клієнта з хмарною інфраструктурою бази даних. Детальний фрагмент коду асинхронного контролера, що інкапсулює логіку верифікації сесії, агрегації історії смарт-контрактів та маршрутизації службових запитів, наведено в Додатку Б. У лістингу 3.7 наведено оптимізовану структуру компонента, яка демонструє алгоритми асинхронного виведення даних та селекції екранних вкладок.

Лістинг 3.7 – Фрагмент коду що відповідає за логіку сторінки Profile.jsx

```
import { useState, useEffect } from "react";
import { supabase } from "../supabaseClient";
```

```
import { useWeb3 } from "../components/Web3Context";
const Profile = ({ user }) => {
  const [activeTab, setActiveTab] = useState("general");
  const [profileData, setProfileData] = useState(null);
  const [transactions, setTransactions] = useState([]);
  const { walletAddress } = useWeb3();
  useEffect(() => {
    if (!user) return;
    const fetchUserData = async () => {
      const { data: profile } = await
supabase.from("profiles").select("*").eq("id", user.id).single();
      const { data: payments } = await
supabase.from("payments").select("*").eq("user_id",
user.id).order("created_at", { ascending: false });
      if (profile) setProfileData(profile);
      if (payments) setTransactions(payments);
    };

    fetchUserData();
}, [user]);
```

кінець лістингу 3.7

Процес ініціалізації сторінки запускається за допомогою хука useEffect, який реагує на зміну об’єкта сесії user. Безперечно, всередині цього тригера було реалізовано послідовне витягування даних за допомогою методів побудови запитів клієнтської бібліотеки. З таблиці profiles за допомогою фільтра .eq("id", user.id) успішно витягується персональна інформація клієнта, зокрема ім’я, прізвище та номер телефону, які були занесені туди під час попередньої реєстрації. Паралельно система звертається до таблиці payments, ефективно вилучаючи повний перелік фінансових операцій, пов’язаних із цим акаунтом. Завдяки модифікатору .order() здійснюється сортування транзакцій у зворотному хронологічному порядку, що дає змогу користувачу бачити останні блокчейн-платежі безпосередньо на самому початку списку. Відповідно, отримані масиви записуються у локальні стани profileData та transactions, ініціюючи автоматичне оновлення інтерфейсу React.

Візуальне відображення інформації повністю підпорядковане змінній стану activeTab. Цілком логічно, що для рендерингу конкретних секцій було використано логічний оператор короткого замикання &&. Якщо користувач

обирає вкладку історії, система миттєво розгортає масив transactions у вигляді динамічної таблиці, де кожен рядок наочно демонструє тип замовленої послуги, суму в ETH та унікальний криптографічний хеш транзакції (tx_hash). З іншого боку, кабінет слугує не лише для читання, а й для безпосереднього запису нових off-chain метрик. При переході на вкладку служби підтримки активується екранна форма, де передбачена можливість введення теми та тексту повідомлення. При відправленні цієї форми алгоритм автоматично викликає інструкцію supabase.from("support_tickets").insert(), заносячи у базу даних ідентифікатор користувача, поточний час та зміст звернення.

Важливим елементом архітектури є інтеграція глобального контексту через кастомний хук useWeb3(). Змінна walletAddress вилучається безпосередньо з децентралізованого провайдера та виводиться у верхній частині профілю. Якщо MetaMask підключений, користувач бачить власну публічну адресу, яка є надійним підтвердженням готовності системи до on-chain операцій. Таким чином, сторінка профілю виступає своєрідною точкою конвергенції, де вилучені з PostgreSQL реляційні записи безшовно поєднуються з динамічним блокчейн-станом, гарантуючи клієнту повний контроль над обліковим записом.

Після успішної активації цифрового гаманця та верифікації облікового запису, здійснюється повернення до аналізу архітектури головної сторінки Home.jsx, де користувач безпосередньо ініціює комерційну взаємодію з платформою. Вбачається цілком закономірним, що момент замовлення послуги є кульмінаційною точкою роботи всього dApp, оскільки саме на цьому етапі класичний веб-інтерфейс повністю поступається місцем децентралізованій бізнес-логіці. Натискання кнопки замовлення ініціює активацію інтерактивного модуля OrderModal.jsx, який було спроєктувано як головний інтеграційний шлюз. Цей компонент одночасно зчитує стан сесії користувача, звертається до блокчейн-провайдера та здійснює ефективну координацію між двома абсолютно різними технологічними середовищами. Відповідно, у лістингу 3.8 наведено ключовий фрагмент асинхронного алгоритму, який забезпечує виконання платежу та його подальшу off-chain фіксацію.

Лістинг 3.8 – Фрагмент коду що відповідає за інтеграцію смарт-контракту та синхронізацію з БД

```
const handleOrderSubmit = async (e) => {
  e.preventDefault();
  if (!walletAddress) return;
  setLoading(true);
  try {
    const provider = new ethers.BrowserProvider(window.ethereum);
    const signer = await provider.getSigner();
    const contract = new ethers.Contract(CONTRACT_ADDRESS, CONTRACT_ABI,
signer);
    const tx = await contract.payForService(serviceType, description, {
      value: ethers.parseEther("0.01"),
    });
    const receipt = await tx.wait();
    await supabase.from("payments").insert([
      {
        user_id: user.id,
        service_type: serviceType,
        amount: 0.01,
        description: description,
        tx_hash: receipt.hash,
      },
    ]);
    onClose();
  } catch (error) {
    console.error("Критичний збій виконання операції:", error);
  } finally {
    setLoading(false);
  }
};
```

кінець лістингу 3.8

Представлений код дозволяє детально простежити логіку виконання транзакційного конвеєра. Процес побудований як строга послідовність асинхронних операцій, де безпека фінансів є абсолютним пріоритетом. Спочатку ми створюємо екземпляр класу ethers.Contract, передаючи туди мережеву адресу задеплоєного контракту та його специфікацію ABI. Виклик децентралізованої функції payForService автоматично викликає вікно MetaMask, змушуючи користувача підтвердити списання коштів у розмірі 0,01 ETH. Ми використали інструкцію await tx.wait(), яка призупиняє клієнтський алгоритм, очікуючи на беззаперечне підтвердження від майнерів тестової мережі Sepolia.

Тільки після того, як блокчейн повертає валідний об’єкт квитанції receipt, система переходить до етапу off-chain синхронізації. Використовуючи метод .insert(), в таблицю payments бази даних PostgreSQL заноситься детальна інформація про замовлення. Ключовим архітектурним рішенням тут є обов’язкове збереження унікального хешу транзакції receipt.hash. Це повністю унеможливлює ситуацію, коли послуга фіксується як оплачена без фактичного надходження коштів на смарт-контракт розробника.

Оскільки хмарна база даних оновлюється миттєво, цей транзакційний запис стає доступним для всіх суміжних модулів системи. Як вже було зазначено під час аналізу кабінету користувача, при наступному переході на сторінку профілю асинхронний метод fetchUserData автоматично вилучить цей рядок із таблиці. Клієнт миттєво побачить нове замовлення у своїй фінансовій історії, де зафіксований хеш виступатиме незаперечним цифровим чеком. Таким чином, було успішно замкнуто цикл гібридної архітектури застосунку, де кожен крок користувача на головній сторінці математично точно синхронізується з децентралізованим сховищем та відображається в особистому профілі.

Завершивши конструювання клієнтських інтерфейсів та децентралізованих протоколів, було логічним підійти до питання надійного депонування інформації. Традиційні інструменти адміністрування, такі як phpMyAdmin, поступово відходять у минуле, поступаючись місцем сучасним хмарним екосистемам. Для реалізації фізичної моделі бази даних було обрано платформу Supabase, яка надає повноцінний доступ до ядра потужної СКБД PostgreSQL. Цей вибір дозволив нам відмовитися від написання громіздких бекенд-серверів та ручного створення API, делегувавши управління даними та їхній багаторівневий захист безпосередньо системі керування базами даних.

Проєктування інформаційної архітектури було розпочато зі створення таблиці profiles, яка виступає центральним сховищем персональних метаданих. У лістингу 3.9 наведено SQL-скрипт генерації та налаштування відповідних тригерів безпеки.

Лістинг 3.9 – SQL-запит для створення таблиці profiles

```
const handleOrderSubmit = async (e) => {
  e.preventDefault();
  if (!walletAddress) return;
  setLoading(true);
  try {
    const provider = new ethers.BrowserProvider(window.ethereum);
    const signer = await provider.getSigner();
    const contract = new ethers.Contract(CONTRACT_ADDRESS, CONTRACT_ABI,
signer);
    const tx = await contract.payForService(serviceType, description, {
      value: ethers.parseEther("0.01"),
    });
    const receipt = await tx.wait();
    await supabase.from("payments").insert([
      {
        user_id: user.id,
        service_type: serviceType,
        amount: 0.01,
        description: description,
        tx_hash: receipt.hash,
      },
    ]);
    onClose();
  } catch (error) {
    console.error("Критичний збій виконання операції:", error);
  } finally {
    setLoading(false);
  }
};
```

кінець лістингу 3.9

Під час аналізу наведеного коду стає абсолютно очевидним, що первинний ключ id має тип UUID та жорстко посилається на системну таблицю auth.users. Зазначений підхід надійно гарантує строгу реляційну цілісність системи – жоден профіль фізично не здатен існувати без відповідного криптографічного облікового запису. Проте постає закономірне питання щодо того, чи достатньо просто створити такий зв’язок для комплексного забезпечення приватності. Безумовно, ні, оскільки фундаментальною основою закладеної архітектури безпеки виступає потужний механізм Row Level Security (RLS). За допомогою спеціалізованої інструкції CREATE POLICY було успішно впроваджено непорушне правило: кожен клієнт отримує беззаперечне право виконувати

маніпуляції SELECT, UPDATE та INSERT виключно для тих рядків, де значення id повністю збігається з його поточним маркером автентифікації auth.uid(). Як наслідок, наповнена інформацією таблиця залишається абсолютно невразливою до будь-яких спроб горизонтальних атак.

Відповідно, логічним наступним кроком стала безпосередня реалізація дієвого механізму збереження фінансової історії. Для досягнення цієї мети було спроєктувано та розгорнуто таблицю payments, яка успішно виконує роль надійного off-chain реєстратора транзакцій. Безпосередня логіка її інтеграції детально продемонстрована у лістингу 3.10.

Лістинг 3.10 – SQL-запит для створення таблиці payments

```
create table payments (
  id bigint generated by default as identity primary key,
  created_at timestamp with time zone default timezone('utc'::text, now())
not null,
  user_id uuid references auth.users not null,
  service_type text not null,
  description text,
  amount text,
  tx_hash text not null
);
alter table payments enable row level security;
create policy "Users can insert their own payments"
  on payments for insert
  with check (auth.uid() = user_id);
create policy "Users can view their own payments"
  on payments for select
  using (auth.uid() = user_id);
using (auth.uid() = user_id);
```

кінець лістингу 3.10

Ключова особливість цієї архітектурної структури полягає у зберіганні текстового поля tx_hash. Воно успішно виступає своєрідним надійним цифровим мостом між традиційною реляційною базою даних та децентралізованою блокчейн-мережею. Вбачається цілком закономірним, що політики RLS на цьому рівні налаштовано ще більш консервативно. Відповідно, клієнтам надано дозвіл виключно на перегляд власних платежів та ініціювання нових записів, тоді як можливість їхньої подальшої модифікації чи безповоротного видалення було

навмисно та категорично виключено безпосередньо на рівні системи. Такий суворий підхід надійно цементує фінансову історію. Нарешті, для гарантування безперебійної та ефективної комунікації із замовниками було спроєктувано та розгорнуто спеціалізовану таблицю support_tickets, безпосередній SQL-код якої детально подано в лістингу 3.11.

Лістинг 3.11 – SQL-запит для створення таблиці support_tickets

```
create table support_tickets (
  id uuid default gen_random_uuid() primary key,
  user_id uuid references auth.users not null,
  subject text not null,
  message text not null,
  created_at timestamp with time zone default timezone('utc'::text, now())
);
alter table support_tickets enable row level security;
create policy "Користувачі можуть створювати власні запити." on
support_tickets
  for insert with check (auth.uid() = user_id);
for insert with check (auth.uid() = user_id);
```

кінець лістингу 3.11

Тут було застосовано нативну функцію gen_random_uuid() для автоматичної генерації унікальних ідентифікаторів клієнтських звернень. Політика безпеки в цьому випадку зведена до абсолютного мінімуму – клієнт отримує право виключно ініціювати інструкцію INSERT, створюючи новий запит. Натомість зчитування цих даних залишається суворою прерогативою адміністратора бази даних. Це цілком ефективно усуває ризики та повністю виключає витік чутливої інформації між різними акаунтами.

Розглядаючи механізми захисту спроєктованих баз даних, варто окремо зупинитися на фундаментальному архітектурному підході до ідентифікації клієнтів. Окрім безпосереднього конструювання користувацьких сутностей, міцним фундаментом безпеки децентралізованого застосунку виступає ізольована підсистема управління сесіями. З огляду на це, під час проєктування інформаційної моделі процеси криптографічного захисту було свідомо делеговано спеціалізованому хмарному модулю Supabase Auth. Тобто замість

власноручної розробки громіздкого бекенд-сервера для хешування паролів та управління життєвим циклом JWT-токенів, перевагу було віддано імплементації готового надійного інфраструктурного рішення як сервісу (BaaS).

Відповідно до обраної моделі, коли клієнтський інтерфейс ініціює функцію реєстрації, хмарне ядро бази даних самостійно перехоплює електронну пошту, створює криптостійкий зліпок пароля та автоматично генерує унікальний ідентифікатор користувача (UID). Вбачається абсолютно очевидним, що саме цей згенерований інфраструктурою криптографічний ідентифікатор стає тим самим глобальним маркером auth.uid(), на якому безпосередньо базується вся вищеописана логіка Row Level Security. Отже, абстрактний механізм хмарної авторизації нерозривно пов’язується з конкретними правами доступу до фінансових записів та персональних профілів на рівні SQL-тригерів. Практика переконливо доводить, що застосування такого підходу дозволило гарантувати абсолютну цілісність реляційних зв’язків між публічними даними та захищеними акаунтами, повністю усунувши вектори атак, притаманні застарілим самописним системам шифрування.

3.2 Тестування та налагодження системи вебзастосунку з підтримкою смарт-контрактів

Процес розгортання, комплексного тестування та експлуатаційного налагодження децентралізованого веб-застосунку YStudio зумовив об’єктивну необхідність застосування багаторівневої інженерної стратегії, оскільки гібридна Web3-архітектура спроєктованої системи органічно поєднує в собі як класичні хмарні сервіси, так і розподілені криптографічні протоколи. Вбачається цілком закономірним, що на відміну від звичайних монолітних програмних продуктів, де процес налагодження здебільшого обмежується перевіркою локальних функцій, розроблений веб-застосунок функціонує в умовах постійної асинхронної взаємодії між клієнтським інтерфейсом React, реляційним ядром PostgreSQL на платформі Supabase та віртуальною машиною Ethereum (EVM) у

тестовій мережі Sepolia. Цілком очевидно, що саме такий складний розподілений стек зумовив перехід до етапу верифікації коду з максимальною педантичністю, залучаючи широкий спектр інструментів моніторингу, автоматизованого контролю та глибокого профілювання.

З огляду на це, під час обґрунтування методології верифікації, в основу інженерної стратегії було закладено фундаментальну концепцію захисного програмування, а також наскрізне тестування поведінкових факторів, динамічну валідацію й обробку виняткових ситуацій у реальному часі (runtime testing). Існують усі підстави вважати, що для архітектури з глибоким асинхронним стеком саме цей підхід є найбільш репрезентативним. Він ефективно забезпечує безперервний автоматичний контроль цілісності даних та стабільності системи безпосередньо під час взаємодії користувача з інтерфейсом, повністю нівелюючи ризик виникнення прихованих логічних аномалій чи неконтрольованих збоїв.

Практика переконливо доводить, що основну частину локального налагодження та рефакторингу цих процесів найдоцільніше було виконати безпосередньо в інтегрованому середовищі розробки Visual Studio Code. Такий безперервний моніторинг стану виконання коду виступає критично важливою умовою для фінальної стабілізації гібридного застосунку перед його повноцінним розгортанням у виробничому середовищі. Відповідно, було успішно розгорнуто комплексну систему інспектування асинхронних процесів (рис. 3.2).

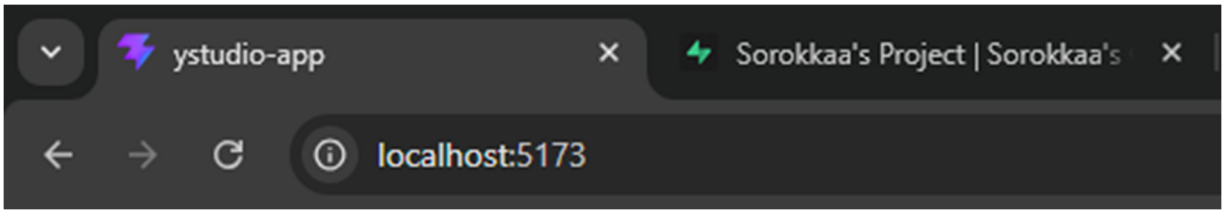

Рисунок 3.2 – Запуск проєкту в локальному середовищі розробки

Джерело: розроблено автором

Особливу увагу було приділено аналізу консольного виводу браузера та відстеженню життєвого циклу React-хуків за допомогою розширення React

Developer Tools у вкладці Components. Інспектування деревоподібної структури інтерфейсу та профілювання внутрішнього стану компонента Profile дозволило у реальному часі перевірити точність збереження метаданих. Системний моніторинг підтвердив, що після проходження автентифікації внутрішній стан React-компонента безпомилково ініціалізує об’єкт користувача із реальними атрибутами профілю «Юрій Сорочук», а також динамічно підтягує відповідні масиви даних профілю з бази (рис. 3.3).

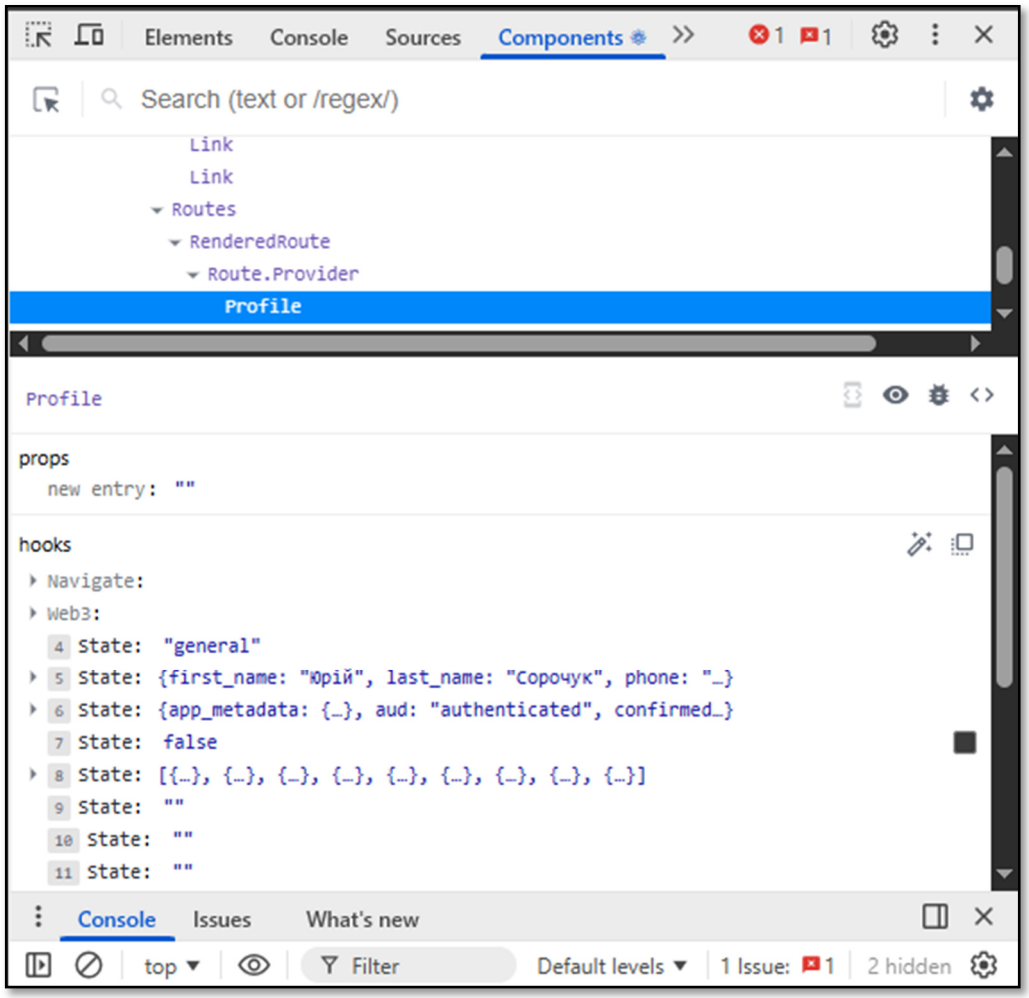

Рисунок 3.3 – Інспектування стану компонента Profile у React DevTools

Джерело: розроблено автором

Будь-яка асинхронна дія, пов’язана з автентифікацією в Supabase чи відправкою транзакцій, обов’язково проходить через захисні блоки try/catch. Програма автоматично тестує мережеві відповіді: у разі збою код не руйнується,

а помилка перехоплюється, реєструється в системному журналі через console.error та безпечно виводиться на екран у вигляді клієнтських сповіщень (рис. 3.4).

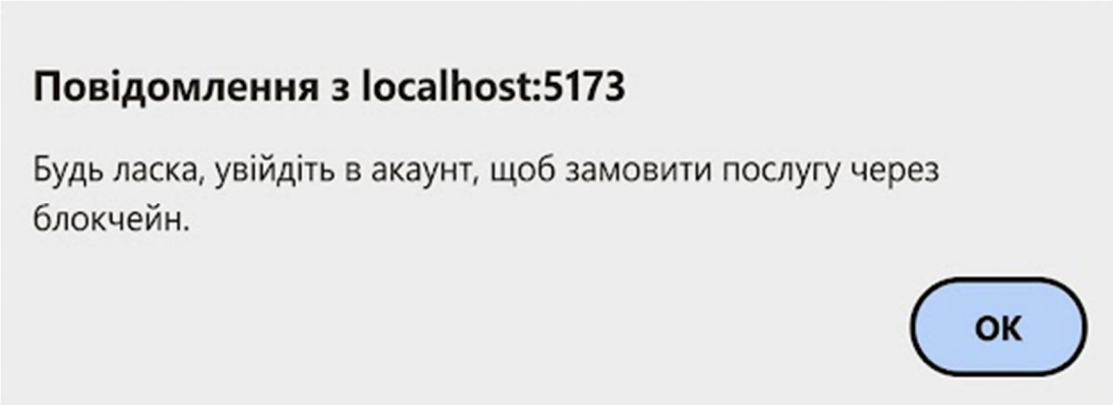

Рисунок 3.4 – Приклад клієнтських сповіщень

Джерело: розроблено автором

Перед ініціалізацією Web3-сценаріїв інтерфейс самостійно перевіряє наявність плагіна гаманця через конструкцію if (!window.ethereum), запобігаючи виконанню завідомо невалідних операцій.

Для більш глибокого інструментального аналізу мережевої активності вебресурсу було додатково використана панель інструментів розробника Chrome DevTools, зокрема її спеціалізовану вкладку Network. Загалом система здійснила 79 мережевих запитів, забезпечивши передачу 19,8 кБ оптимізованого трафіка із загального обсягу ресурсів у 5,9 МБ. Була зафіксована серія послідовних асинхронних HTTP-запитів до таблиць profiles та payments, які успішно отримали від хмарного сервера статус 200 (OK). При цьому швидкість завантаження статичних графічних елементів інтерфейсу склала всього 6-16 мілісекунд, тоді як складні аналітичні вибірки даних профілю оброблялися в діапазоні від 100 до 217 мілісекунд, що беззаперечно підтверджує відсутність надлишкового навантаження (рис. 3.5).

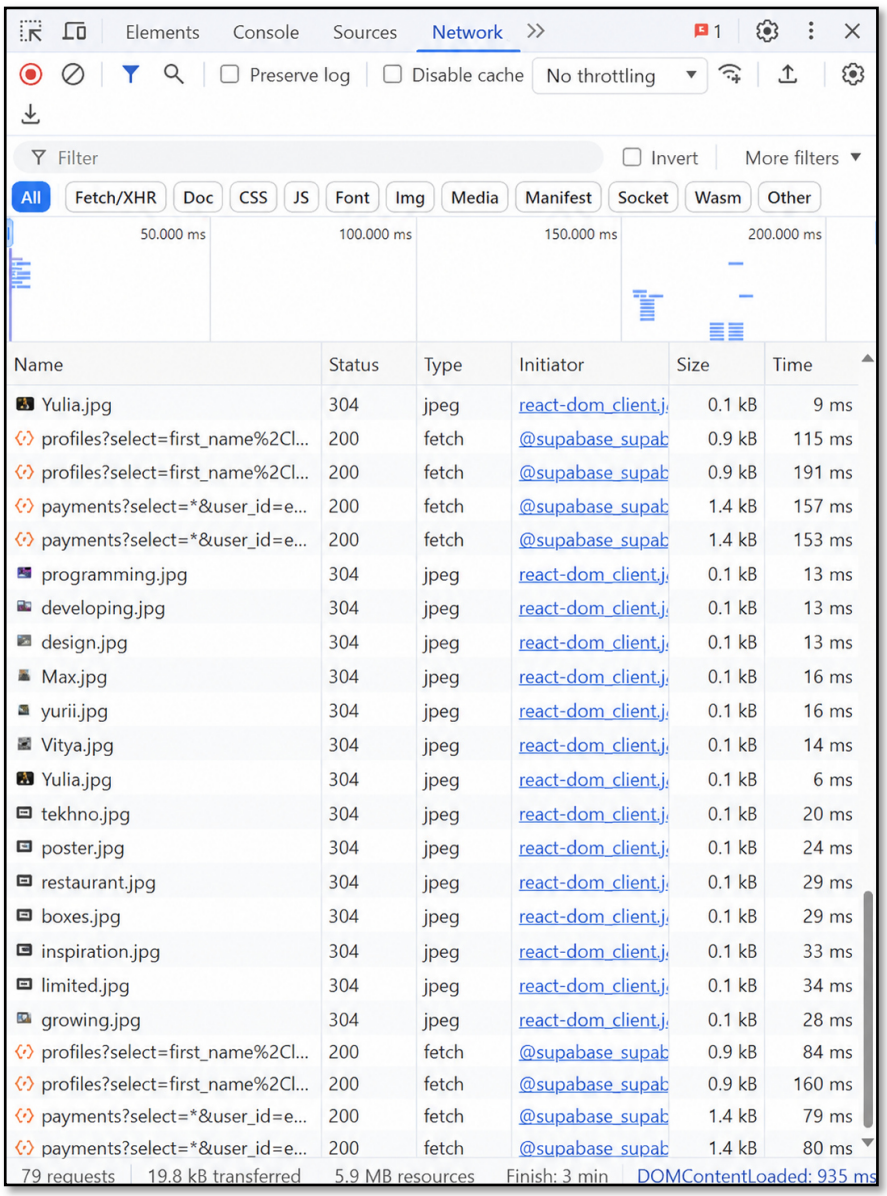

Рисунок 3.5 – Результати аналізу мережевих запитів у вкладці Network

Джерело: розроблено автором

Додатковим підтвердженням високої якості інтерфейсу стали результати аналізу метрик швидкодії у вкладці Performance, де ключовий показник швидкості відображення основного контенту Largest Contentful Paint (LCP) склав усього 2,29 секунди, а індекс зсуву макета Cumulative Layout Shift (CLS) зафіксовано на еталонній позначці 0,01. Під час проведення експлуатаційних тестів локальне значення INP для застосунку зафіксувалося на позначці у 40 мс, що згідно з офіційними критеріями оптимізації Core Web Vitals є еталонним результатом (рис. 3.6).

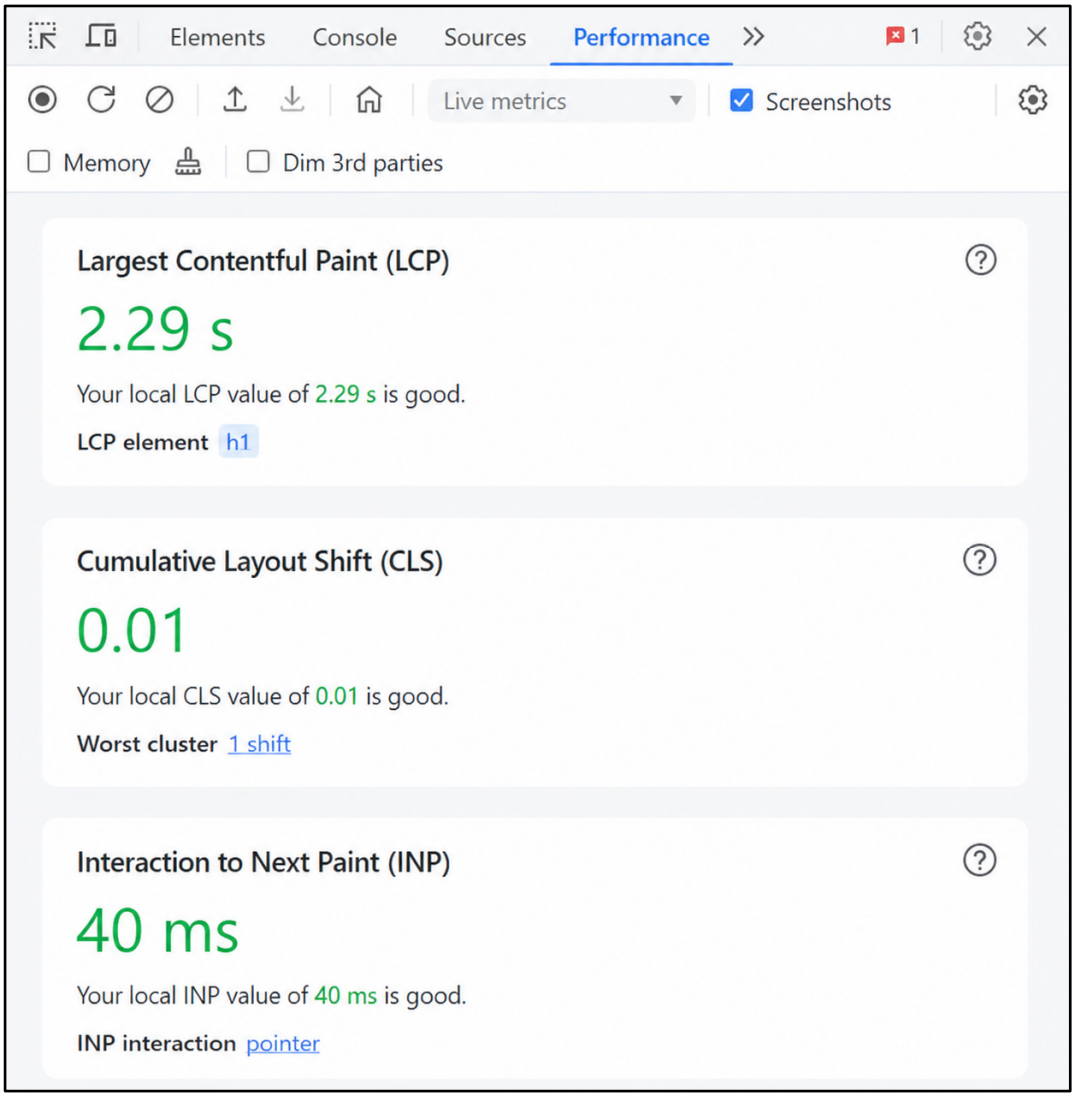

Рисунок 3.6 – Показники швидкодії інтерфейсу у вкладці Performance

Джерело: розроблено автором

Паралельно з інтерфейсним аналізом варто провести всебічне профілювання хмарної інфраструктури Supabase за допомогою інтегрованого аналітичного дашборду. Протягом контрольного вікна експлуатаційного моніторингу за 24 години платформа зафіксувала сумарно 371 запит. З них переважна більшість припала на класичні звернення до реляційних таблиць бази даних PostgreSQL (300 запитів), тоді як підсистема керування сесіями та реєстрації нових користувачів згенерувала 71 автентифікаційний запит. Кількість запитів можна переглянути на рисунку 3.7.

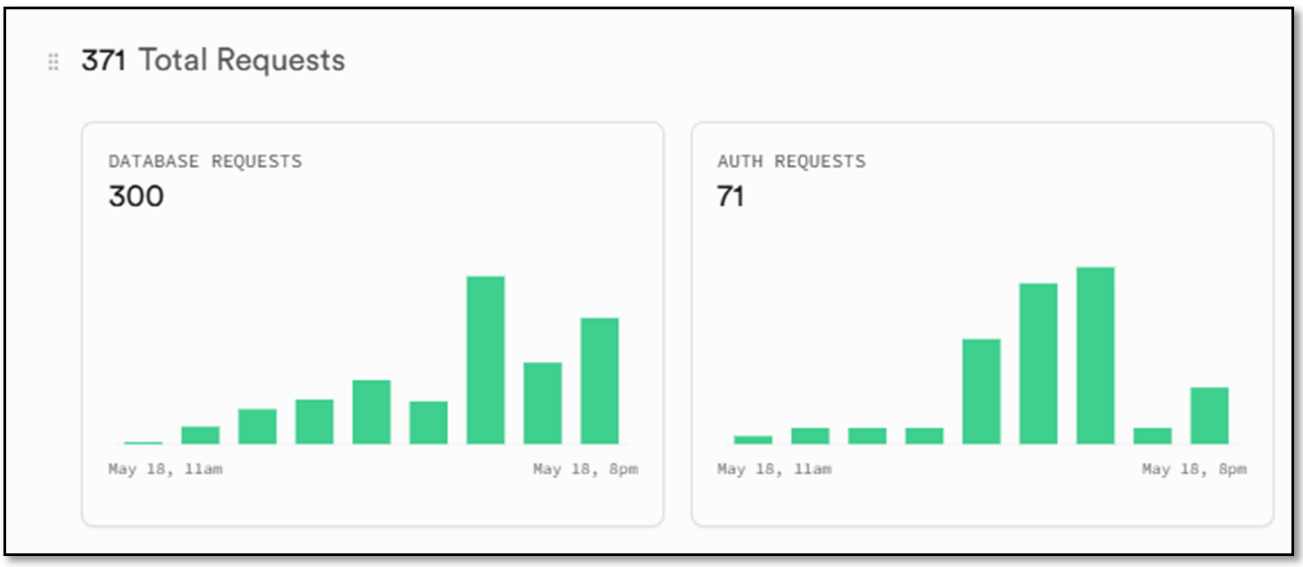

Рисунок 3.7 – Розподіл навантаження та статистика запитів у Supabase

Джерело: розроблено автором

Аналізуючи внутрішні процеси СУБД-сервера через низькорівневі системні журнали Postgres, вдалося провести детальний аудит безпеки. Логи наочно продемонстрували, що кожне підключення до системи ініціювалося з використанням сучасного криптографічного методу автентифікації scram-sha-256 для облікового запису postgres. Весь мережевий обмін даними відбувався виключно через захищений шифрований протокол SSL версії v1.3 з використанням стійкого шифру TLS_AES_256_GCM_SHA384. З цього логічно випливає, що така комплексна криптографічна конфігурація ефективно блокує будь-які спроби мережевого перехоплення трафіку. Безперечно, інформаційний периметр бази даних залишається абсолютно герметичним. Отже, результати проведеного аудиту переконливо доводять готовність розгорнутої інфраструктури до безпечної експлуатації, оскільки надійний реляційний бекенд виступає фундаментальною передумовою для подальшої безшовної інтеграції з децентралізованим блокчейн-середовищем. Поодинокі системні сповіщення про відсутність міграційних таблиць схеми були ідентифіковані як некритичні та успішно нівельовані шляхом фінальної синхронізації структури бази даних (рис. 3.8).

Рисунок 3.8 – Системний журнал СУБД із логами безпеки підключень

Джерело: розроблено автором

Окремим важливим аспектом під час моніторингу інтерфейсної частини у консолі розробника став аналіз інформаційних сповіщень системи безпеки. Зокрема, у журналі Issues було зафіксовано попередження від механізму Content Security Policy (рис. 3.9).

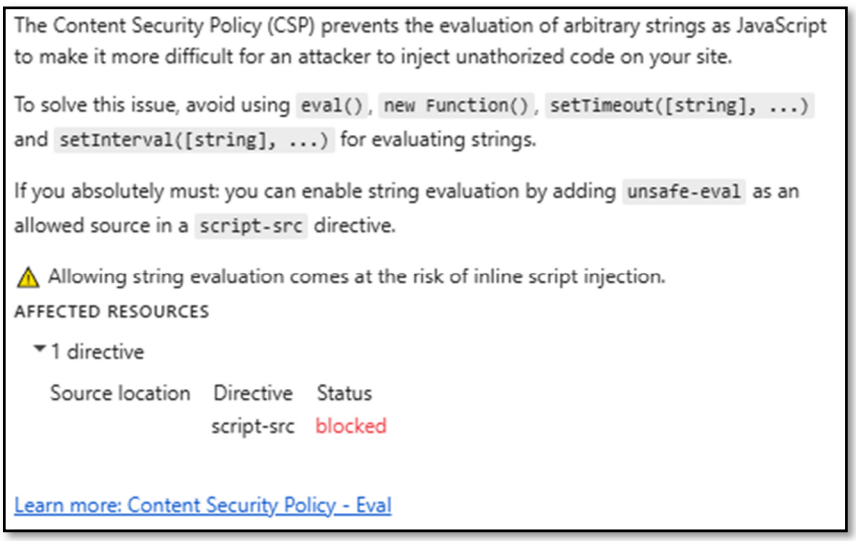

Рисунок 3.9 – Попередження механізму CSP в консолі розробника

Джерело: розроблено автором

Вбудовані захисні алгоритми сучасного браузера автоматично заблокували потенційну можливість виконання довільних текстових рядків як JavaScript-коду через застарілу команду eval() або конструктор new Function(). Поява цього запису є абсолютно стандартною реакцією системи захисту на фонові діагностичні скрипти сторонніх Web3-розширень у середовищі хостингу Vercel, який за замовчуванням виставляє суворі заголовки безпеки для публічних сайтів. Оскільки написаний React-код не використовує інструментів динамічної компіляції, це автоматичне блокування жодним чином не порушило працездатність інтерфейсу, підтвердивши повну стійкість веб-застосунку до міжсайтових скриптових ін'єкцій класу Cross-Site Scripting (XSS).

Фінальним та найвідповідальнішим етапом налагодження стала перевірка блокчейн-компонента створеної екосистеми. Написаний смарт-контракт ServicePayment на мові Solidity пройшов серію суворих транзакційних випробувань у тестовій мережі Sepolia. За допомогою реєстру Etherscan було зафіксовано повну історію з семи успішно проведених операцій (рис. 3.10), надісланих на адресу контракту.

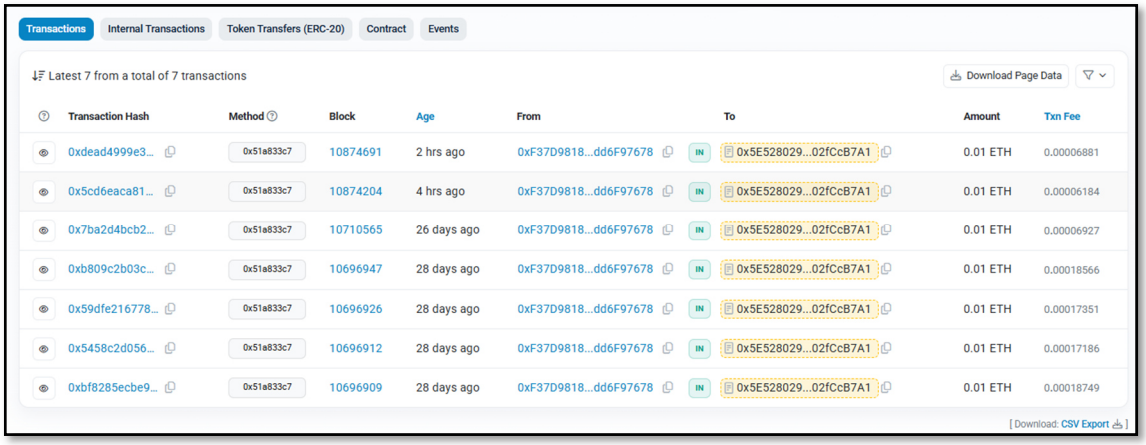

Рисунок 3.10 – Список транзакцій смарт-контракту на платформі Etherscan [15]

Емпіричний аналіз конкретної контрольної транзакції у вікні детального перегляду провідника Etherscan, яка отримала 533 підтвердження блоків, надав беззаперечні матеріальні докази працездатності. Журнал внутрішніх транзакцій

(Internal Transactions) чітко зафіксував автоматичний розподіл коштів: вхідний платіж обсягом 0,01 ETH з хешем 0xdead4999e3bb08e924fbfe6ae1463c6a83bc4751ac5506e0cc4b6b8b8b3e1dc12 був миттєво перенаправлений з адреси смарт-контракту на кінцевий гаманець розробника 0xF37D98187b1d6E3Da40465Ec7065E29dd6F97678 (рис. 3.11). Ця метрика повністю підтверджує, що закладена логіка безпечного фінансового транзиту функціонує бездоганно, виключаючи ризик заморожування або компрометації цифрових активів.

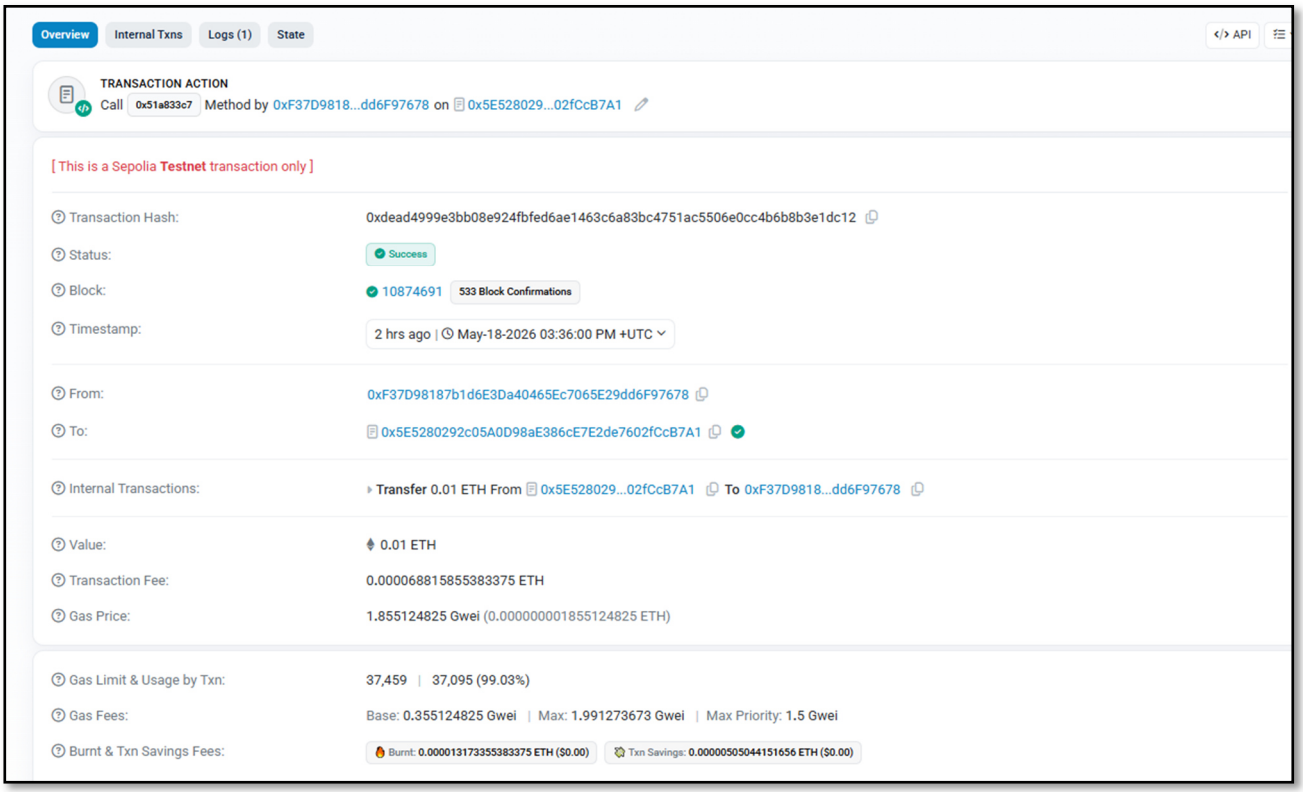

Рисунок 3.11 – Аудит та верифікація внутрішнього транзиту коштів [15]

Завершальним кроком практичної розробки стало глобальне розгортання клієнтської частини веб-застосунку на хмарі Vercel, що дозволило перенести проєкт із локального комп’ютера в реальний інтернет-простір. Після усунення початкових розбіжностей у регістрі назв імпортованих файлів-компонентів, викликаних чутливістю Linux-серверів до регістру символів, автоматизований конвеєр CI/CD на Vercel успішно завершив збірку проєкту з приватного GitHub-репозиторію. Веб-застосунок отримав офіційний статус «Ready» та розгорнув

повністю функціонуюче Web3-середовище під унікальним доменним ім’ям ystudio-app.vercel.app (рис. 3.12).

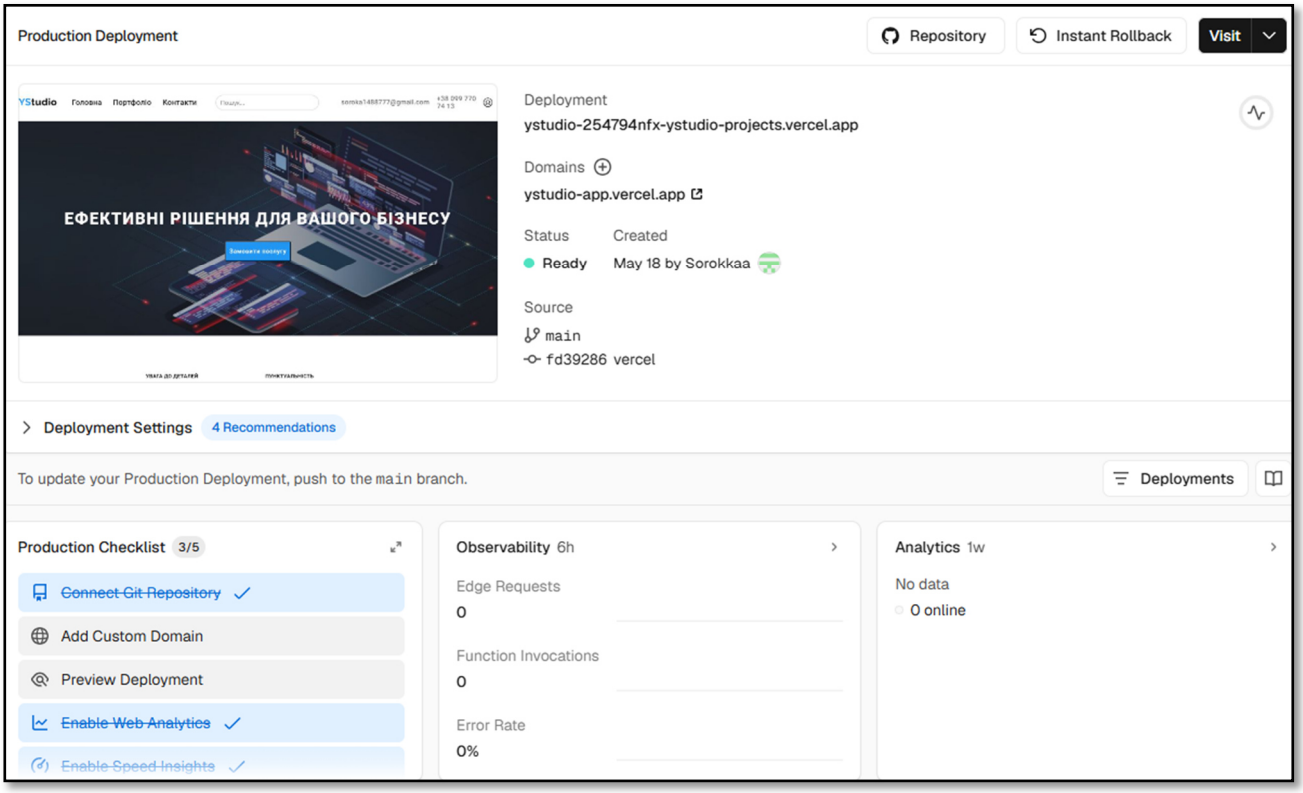

Рисунок 3.12 – Успішне глобальне розгортання проєкту на платформі Vercel [16]

Завершуючи комплексний аналіз експлуатаційних параметрів розробленого децентралізованого веб-застосунку YStudio, був сформований чіткий перелік технічних вимог, необхідних для стабільної та швидкодіючої взаємодії кінцевого користувача з розробленою цифровою екосистемою.

Мінімальні системні вимоги:

– операційна система Windows 10, macOS 11, Linux (Ubuntu 20.04+);

– двоядерний процесор Intel Core i3 або AMD Ryzen 3 з тактовою частотою від 1.8 ГГц;

– оперативна пам’ять від 4 гб;

– стабільний доступ до мережі інтернет на швидкості не менше 5 Мбіт/с;

– встановлене браузерне розширення MetaMask або будь-який сумісний Web3-гаманець, веб-браузер на базі Chromium (Google Chrome версії 90+, Brave, Microsoft Edge) або Mozilla Firefox.

Рекомендовані системні вимоги:

– актуальні версії операційної системи Windows 11 або macOS Sonoma із обов’язково активованим апаратним прискоренням графіки в системі;

– чотириядерний процесор Intel Core i5 чи AMD Ryzen 5 або енергоефективні чіпи Apple Silicon серії M;

– 8 гб оперативної пам’яті або більше для безперешкодного рендерингу динамічних елементів інтерфейсу та паралельної обробки криптографічних підписів транзакцій;

– широкосмуговий інтернет-канал зі швидкістю від 20 Мбіт/с та низьким рівнем затримки (ping) до розподілених вузлів мережі Ethereum Sepolia;

– остання актуальна версія браузера Google Chrome із активованим та авторизованим гаманцем MetaMask, що налаштований на роботу з тестовою мережею Sepolia.

3.3 Інструкція користувача з інсталяції та використання програмного продукту

Оскільки розроблена система функціонує як сучасна хмарно-орієнтована SPA-платформа (Single Page Application), було повністю нівельовано потребу у складній локальній інсталяції програмних компонентів, конфігурації серверних модулів чи синхронізації блокчейн-вузлів на комп’ютері кінцевого споживача. Як уже згадувалося, весь скомпільований виробничий код клієнтської частини було успішно розгорнуто на хмарі безкоштовного високопродуктивного хостингу Vercel. Відповідно, це інженерне рішення дозволяє отримати миттєвий глобальний доступ до інструментів створеної екосистеми з будь-якого пристрою за допомогою звичайного інтернет-браузера.

Цілком логічно, що безпосередній процес використання системи зводиться до простого переходу за єдиним уніфікованим URL: https://ystudio-app.vercel.app/. При цьому для повноцінної взаємодії з децентралізованими смарт-контрактами єдиною технічною вимогою до клієнта залишається наявність встановленого браузерного розширення MetaMask.

Вбачається цілком очевидним, що експлуатаційний алгоритм практичного застосування програмного продукту є виключно лінійним і складається з кількох логічних інтерактивних кроків. Насамперед, під час первинного відвідування вебресурсу користувачеві необхідно пройти швидку процедуру реєстрації, обов’язково вказавши унікальну електронну адресу, персональні дані та надійний пароль (рис. 3.13).

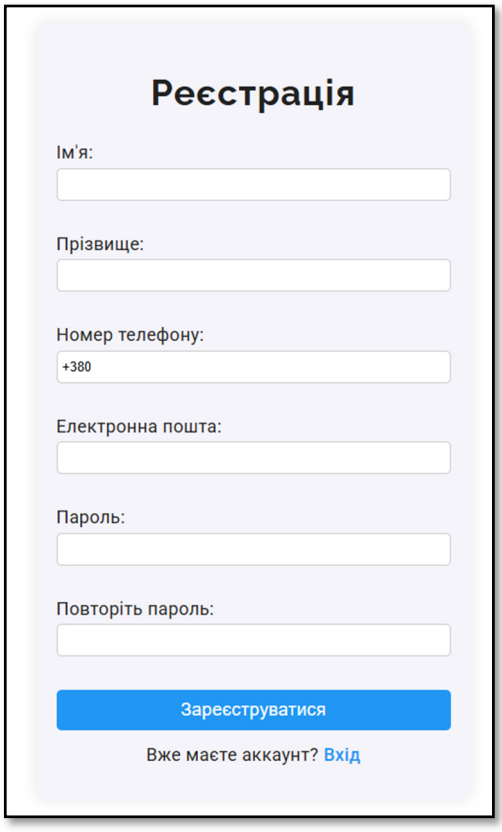

Рисунок 3.13 – Реєстрація користувача

Джерело: розроблено автором

Якщо користувач вже має обліковий запис, то може використати авторизацію, натиснувши на посилання нижче та використавши свою

електронну пошту та пароль (рис. 3.14). Цей етап повністю обслуговується безпечним хмарним модулем Supabase Auth, відповідно дані користувача знаходяться в безпеці.

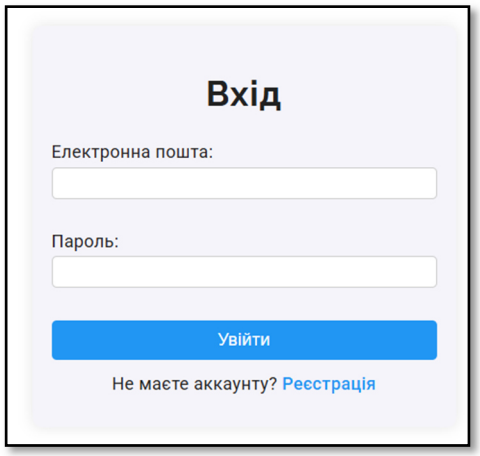

Рисунок 3.14 – Авторизація користувача

Джерело: розроблено автором

Після успішного входу в систему здійснюється автоматичний перехід до інтерфейсу особистого профілю. Для активації платіжних можливостей платформи користувачеві необхідно натиснути кнопку підключення гаманця MetaMask, який має бути заздалегідь переключений на тестову блокчейн-мережу Sepolia (рис. 3.15). Це пов’язує хмарний акаунт із криптографічною адресою в розподіленому реєстрі.

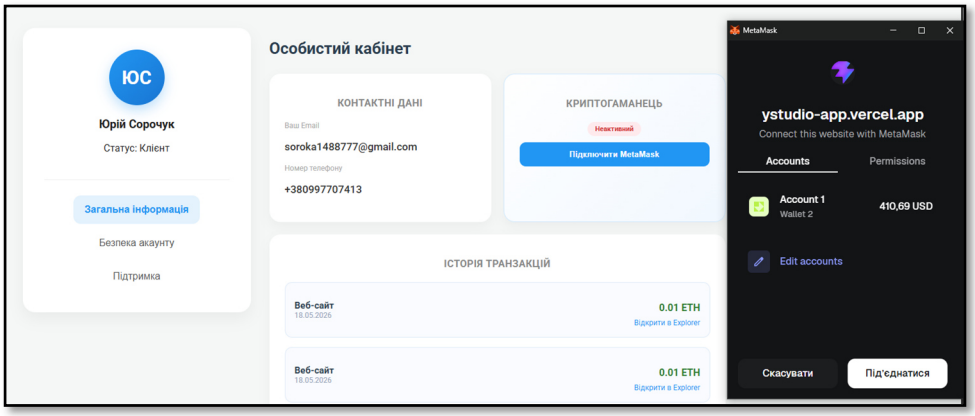

Рисунок 3.15 – Підключення криптогаманця MetaMask

Джерело: розроблено автором

Оскільки в межах проєкту використовується тестова блокчейн-мережа Sepolia, для повноцінної демонстрації роботи застосунку застосовуються виключно несправжні цифрові активи. Вбачається цілком закономірним, що такий підхід до прототипування дозволяє проводити вичерпну валідацію платіжних шлюзів в умовах, максимально наближених до реальної експлуатації (Mainnet). Це ефективно нівелює будь-які фінансові ризики, пов'язані з можливою непередбачуваною поведінкою експериментального коду на етапі налагодження (рис. 3.16).

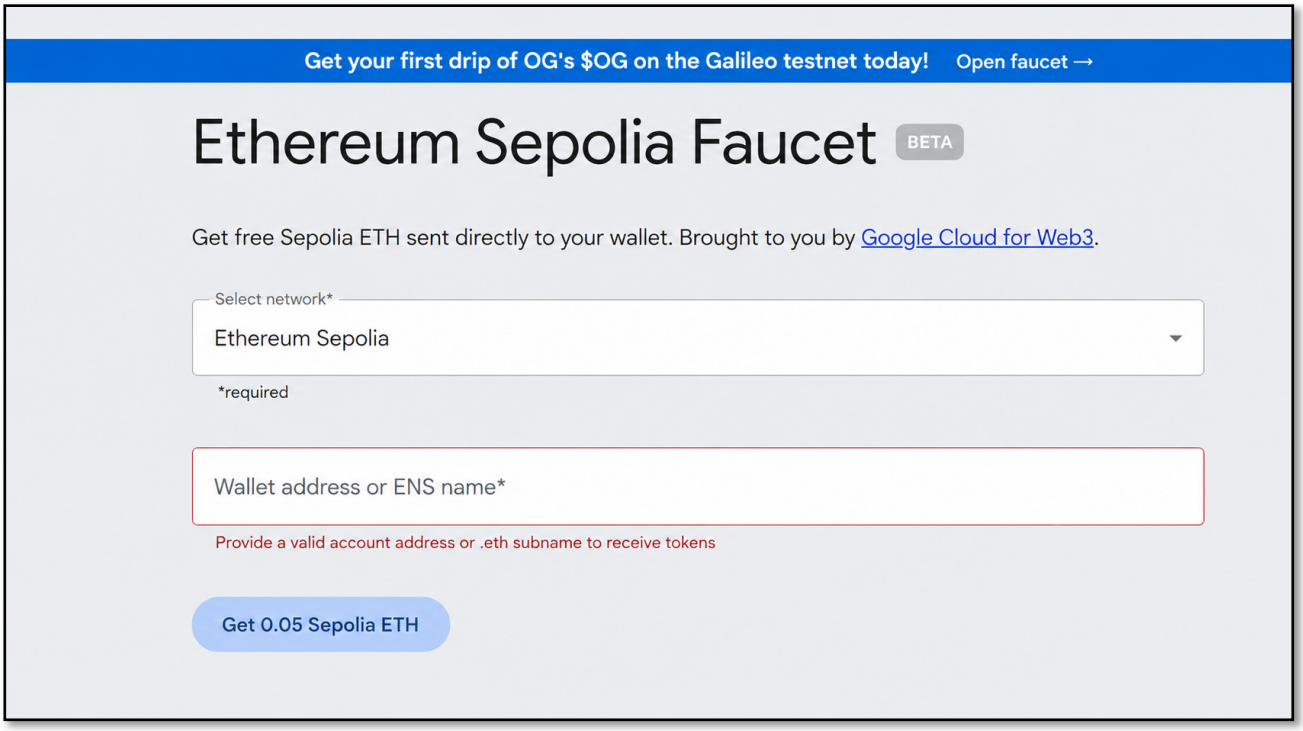

Рисунок 3.16 – Отримання тестових токенів [17]

Отримавши токени, підключивши гаманець та повернувшись на головну сторінку інформаційної системи, клієнт отримує доступ до інтерактивного каталогу послуг студії. Вибравши потрібний сервіс, користувач натискає кнопку замовлення, що миттєво ініціює виклик смарт-контракту та відкриває вікно MetaMask для підписання транзакції та оплати послуги у розмірі 0,01 ETH. На цьому етапі браузерне розширення виконує надзвичайно важливу роль захищеного посередника. Автономна система автоматично розраховує

актуальну комісію блокчейн-мережі за виконання обчислень та очікує фінального схвалення операції від власника закритого ключа (рис. 3.17).

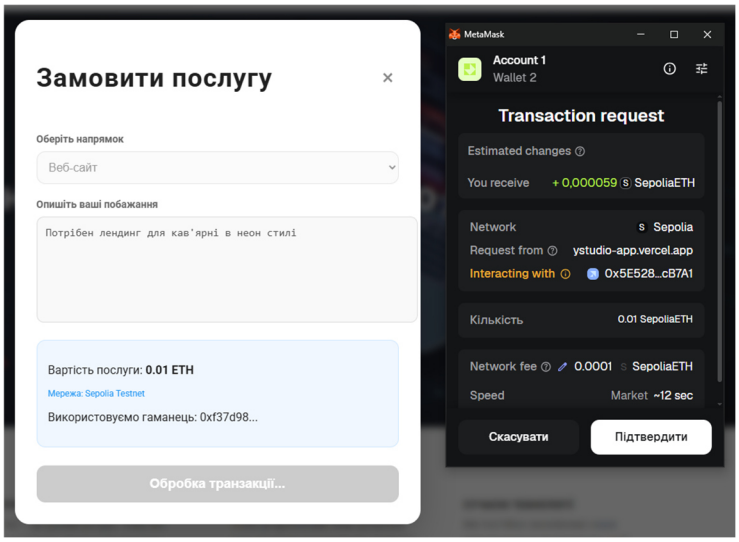

Рисунок 3.17 – Замовлення послуги та оплата через смарт-контракт

Джерело: розроблено автором

Користувач може знову відкрити свій особистий профіль, де у відповідній таблиці миттєво з'являється деталізована історія замовлень із зафіксованими сумами, статусами та унікальними криптографічними хешами операцій (рис. 3.18).

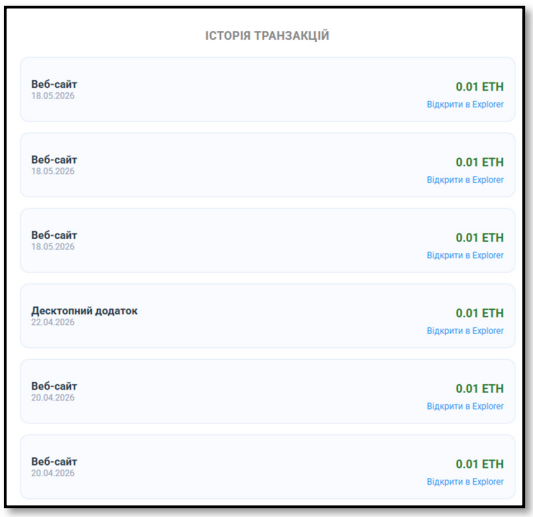

Рисунок 3.18 – Історія транзакцій користувача

Джерело: розроблено автором

РОЗДІЛ 4
СПЕЦІАЛЬНІ РОЗРАХУНКИ

Під час розробки графічного інтерфейсу децентралізованого веб-застосунку YStudio відбулася свідома відмова від виключно суб'єктивних естетичних критеріїв оцінки. Безперечно, натомість було імплементовано суворий інженерний підхід до ергономіки, що ґрунтується на точних математичних моделях та фундаментальних законах когнітивної психології. Вбачається цілком закономірним, що такий вектор проєктування дозволив не просто сформувати візуально приємний ресурс, а й кількісно довести його високу операційну ефективність. Цей аспект є абсолютно критичним. Адже успішна та стабільна взаємодія непідготовленого клієнта зі складним Web3-середовищем вимагає бездоганної архітектурної логіки.

Відповідно, першим етапом практичного дослідження стала оптимізація когнітивного навантаження під час навігації за допомогою закону Хіка [18]. Ця математична модель дозволяє надзвичайно точно передбачити час, необхідний користувачеві для прийняття остаточного рішення, що прямо залежить від кількості доступних альтернатив на екрані. З аналітичного погляду, еквівалент цього закону виражається логарифмічним рівнянням у формулі (4.1).

$$T = a + b\,log_2(\,n + 1)\,, \tag{4.1}$$

де T – середній час реакції користувача;

a – базова константа, що визначає час, необхідний людині на нейрофізіологічну реакцію без процесу вибору (час на усвідомлення самої появи елементів інтерфейсу, що для середньостатистичної людини становить 200 мс;

b – емпірична константа, що описує швидкість обробки одного біта інформації людським мозком під час здійснення вибору (в ергономіці інтерфейсів приймається за 150 мс);

n – кількість рівноцінних варіантів вибору (пунктів меню).

Аналізуючи архітектуру верхньої навігаційної панелі, було свідомо застосовано принцип угруповання та приховування другорядних маршрутів. Замість одночасного виведення всіх існуючих компонентів системи на екран, кількість ключових розділів було раціонально обмежено до чотирьох базових візуальних елементів: посилань «Головна», «Портфоліо», «Контакти» та інтерактивного блоку особистого профілю користувача. Системні сторінки авторизації та реєстрації залишаються динамічно прихованими й замінюють блок профілю виключно для неавторизованих сесій. Підставивши значення n = 4 в рівняння, можна спостерігати наступний результат обчислення: T = 200 + + $150\log_2(4 + 1) = 200 + 150 \times 2{,}32 = 548$ мілісекунд. Цей показник математично підтверджує, що архітектура розробленого фронтенду цілком і повністю позбавлена надлишкового інформаційного шуму. З огляду на вищезазначене, вбачається цілком закономірним: клієнту потрібна лише половина секунди, щоб свідомо просканувати меню і зробити правильний вибір. Це успішно дозволяє орієнтуватися в системі практично на рівні підсвідомих рефлексів.

Наступним кроком було оцінено фізичну ергономіку взаємодії з ключовими інтерактивними елементами із застосуванням закону Фіттса [18]. Для забезпечення максимальної репрезентативності обчислень, моделювання візуальних метрик здійснювалося з урахуванням еталонного робочого простору, зокрема типового співвідношення площі екрана для 17-дюймового монітора. Цей фундаментальний закон надійно моделює час, необхідний для точного наведення курсора миші на цільовий об’єкт, суворо зважаючи на його безпосередній розмір та просторову віддаленість. Відповідно, розрахунок індексу складності руху базується на математичній формулі (4.2).

$$T = a + b \log_2 \left(1 + \frac{D}{W} \right), \tag{4.2}$$

де T – середній час руху кисті руки;

a – константа, що описує фізіологічну затримку на старт та зупинку руху курсора (в інженерії приймається як 50 мс);

b – константа, що відображає швидкість позиціонування, притаманну конкретному маніпулятору (для стандартної комп'ютерної миші цей показник є найвищим і дорівнює 150 мс/біт);

D – відстань від початкової точки курсора до геометричного центру об'єкта в пікселях;

W – ширина цільової області вздовж осі руху в пікселях.

Для тестування було обрано головну кнопку конверсії «Замовити послугу», яку стратегічно розміщено в центрі першого екрана. Враховуючи стандартну роздільну здатність Full HD, притаманну 17-дюймовим моніторам, що використовувалися під час розробки, середня діагональна відстань пробігу курсора від елементів верхнього навігаційного меню до центру екрана становить приблизно D = 750 пікселів, тоді як оптимізована ширина самої кнопки дорівнює W = 180 пікселів. Проведено розрахунок: $T = 50 + 150\log_2(1+ 750/180) = 50 + 150\log_2\times(5,16)$, що становить приблизно $50 + 150 \times 2,36 = 404$ мілісекунди. Отже, завдяки вдало підібраним пропорціям та компонуванню цільової зони, користувач витрачає всього чотири десятих секунди на безпомилкове наведення курсора. Безперечно, такий показник суттєво знижує фізичну втому під час навігації сторінкою та зводить імовірність випадкового кліку повз активний елемент до мінімальної статистичної похибки.

Фінальним акордом ергономічної оцінки стало моделювання життєвого циклу взаємодії з платформою за допомогою прогнозуючої системи GOMS, а саме її ключової метрики Keystroke-Level Model [19]. Цей аналітичний інструмент розбиває будь-який складний користувацький сценарій на серію атомарних мікро-дій. Було змодельовано абсолютно повний, наскрізний платіжний шлях: від первинного виклику форми замовлення, введення кастомного опису завдання до остаточного занесення успішно проведеної операції в базу даних після тривалої блокчейн-верифікації. Загальний час

виконання цього комплексного завдання обчислюється як сума стандартних операторів відповідно до формули (4.3).

$$T_{заг} = \sum T_K + \sum T_P + \sum T_M + \sum T_R, \quad (4.3)$$

де $T_{заг}$ – розрахунковий загальний час виконання повного користувацького сценарію від першої фізичної дії до остаточного відгуку системи;

T_K – час одного фізичного кліку мишею або натискання клавіші на клавіатурі (0,2 с);

T_P – час наведення курсора на об’єкт на екрані (1,1 с);

T_M – час ментальної підготовки, під час якого користувач усвідомлює систему та приймає рішення про наступну дію (1,35 с);

T_R – час відгуку системи на дії користувача (об’єктивна затримка програмного забезпечення або мережі).

Розкладаючи реальний Web3-сценарій на послідовні кроки, було отримано строгу черговість операцій. Зокрема, перший етап включає наведення курсора та клік на центральну кнопку виклику форми послуг ($T_P + T_K$). Водночас другий етап фіксує ментальну підготовку під час вибору конкретного сервісу в модальному вікні, переміщення курсора на потрібну позицію та безпосередню фіксацію вибору ($T_M + T_P + T_K$). Третій, об’єктивно найбільш ресурсомісткий для клієнта етап, передбачає фокусування на текстовому полі та введення технічного опису замовлення. Відповідно до закладених параметрів, середній обсяг опису становить сімдесят п’ять символів, що закономірно додає до формули серію натискань клавіш ($T_M + T_K + T_K + 75 \times T_K$). Зі свого боку, четвертий етап полягає у наведенні та кліку на фінальну кнопку «Замовити» ($T_P + T_K$), що ініціює миттєве системне очікування відгуку Web3-провайдера, де часовий показник відповідає метриці INP, визначеній раніше ($T_R = 0{,}04$ с). П’ятий етап докладно відображає когнітивну перевірку параметрів фінансової транзакції у спливаючому вікні MetaMask, переміщення курсора на елемент підтвердження

та фінальний клік підписання смарт-контракту ($T_M + T_P + T_K$). Безперечно, завершальним інфраструктурним етапом виступає тривале технологічне очікування консенсусу, під час якого розподілені вузли мережі Sepolia генерують блок, а спеціалізований асинхронний хук синхронізує ці дані з хмарною платформою Supabase ($T_R = 15,0$ с).

Підсумовуючи всі константи для зафіксованих кроків процесу – п’ять операторів наведення T_P, вісімдесят операторів кліку та натискання клавіш T_K, три оператори ментальної підготовки T_M та два оператори системної затримки T_R – отримується такий підсумковий математичний вираз: $5 \times 1,1 + 80 \times 0,2 + 3 \times$ $\times 1,35 + 15,04 = 5,5 + 16,0 + 4,05 + 15,04 = 40,59$ секунди.

Вбачається цілком очевидним, що отриманий результат у 40,59 секунди є надзвичайно показовим з погляду інженерної ергономіки децентралізованих систем. Проведений аналіз наочно демонструє, що значна частка часу – рівно 15 секунд – витрачається на об’єктивну затримку самої блокчейн-мережі під час криптографічної валідації контракту, тоді як ще 15 секунд займає суто фізичне введення тексту опису користувачем на клавіатурі. Натомість сумарний час безпосередньої навігації інтерфейсом зведено до абсолютного мінімуму. Існують вагомі підстави стверджувати, що така комплексна математична метрика беззаперечно доводить високу ефективність спроєктованого програмного рішення. Адже навіть з урахуванням ручного набору тексту та специфічних архітектурних затримок блокчейну, загальна тривалість повного циклу оформлення кастомного Web3-замовлення не перевищує психологічно комфортних меж, гарантуючи плавний користувацький досвід абсолютно без жодного когнітивного перевантаження.

Отже, результати проведеного теоретичного моделювання формують надійний еталонний фундамент для ініціалізації подальшого емпіричного тестування. З огляду на вищезазначене, стає цілком очевидним, що інтеграція складних децентралізованих механізмів у звичні користувацькі інтерфейси є абсолютно реальним завданням.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи бакалавра було успішно реалізовано вебзастосунок із підтримкою смарт-контрактів для безпечного виконання крипто-транзакцій, що стало закономірним підсумком комплексного архітектурного та програмного проєктування. Безперечно, досягнення поставленої мети стало можливим завдяки створенню інтегрованої гібридної системи, яка органічно поєднує гнучкість сучасних фронтенд-технологій, надійність хмарного зберігання даних та абсолютну незмінність логіки децентралізованих протоколів. Під час розробки основна увага була зосереджена на вирішенні ключових інженерних викликів, зокрема на створенні клієнтської частини на базі бібліотеки React. Цілком очевидно, що такий підхід дозволив побудувати модульний, чутливий до станів інтерфейс із високою швидкістю відгуку, яка є критично важливою під час роботи з Web3-транзакціями.

Відповідно, було імплементовано механізм безшовної взаємодії з децентралізованим гаманцем MetaMask за допомогою спеціалізованої бібліотеки Ethers.js. Практика переконливо доводить, що це інженерне рішення дозволило мінімізувати технічний поріг входження для кінцевого споживача, суттєво спростивши процес підключення, перевірки балансів та підписання транзакцій до кількох інтуїтивно зрозумілих кроків. Для ефективного управління даними було успішно інтегровано хмарну платформу Supabase, де фундаментальним досягненням стала реалізація суворих політик Row Level Security. Завдяки цим налаштуванням доступ до профілів та історії транзакцій надійно захищений безпосередньо на рівні бази даних, створюючи міцний міст між централізованим зберіганням метаданих та децентралізованою логікою виконання угод. Вбачається цілком закономірним, що технологічним ядром спроєктованої системи став розроблений смарт-контракт на мові Solidity, який безапеляційно гарантує виконання фінансових операцій за жорстко визначеним алгоритмом. Це абсолютно унеможливлює стороннє втручання або будь-які маніпуляції з активами, що додатково підтверджує наукову новизну застосованого підходу до

верифікації даних. Аналіз досягнутих показників переконливо довів, що застосування методів кількісної ергономічної оцінки до розробленого інтерфейсу дозволило докорінно оптимізувати користувацький шлях. З цього логічно випливає беззаперечне підтвердження високої якості спроєктованого візуального та функціонального середовища.

Здобуті результати роботи мають високу практичну цінність і без проблем можуть бути масштабовані для сервісів електронної комерції, фриланс-платформ або складних систем автоматизації платежів, оскільки реалізована архітектура виступає готовим та надійним шаблоном для інтеграції блокчейн-мережі в традиційні вебзастосунки. Перспективи подальшого технологічного розвитку вбачаються у стратегічному переході на рішення Layer 2 задля кардинального зменшення вартості комісій при виконанні транзакцій, а також у впровадженні розширеного функціоналу для створення децентралізованих автономних організацій на базі наявної екосистеми. Існують усі підстави вважати, що запропонований підхід до структурного синтезу React, Supabase та Solidity є найбільш ефективним шляхом до створення безпечних та зручних Web3-сервісів нового покоління, що неодмінно сприятиме масовій демократизації доступу до децентралізованої економіки.

Крім того, подальша еволюція подібних платформ відкриває широкі горизонти для впровадження інтелектуальних алгоритмів з метою предиктивної аналітики та автоматичного виявлення аномальних патернів поведінки смарт-контрактів. Інтеграція модулів машинного навчання дозволить перетворити статичний аудит на динамічну систему моніторингу ризиків, здатну превентивно блокувати підозрілі блокчейн-транзакції ще до моменту їхнього остаточного підтвердження вузлами мережі. Отже, розроблений програмний продукт виступає не просто як локальне інженерне рішення, а як фундаментальна база для розгортання масштабованих фінансових екосистем майбутнього, де максимальна прозорість безшовно поєднується з безпрецедентним рівнем криптографічного захисту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. What is Web3. Harvard Business Review. URL: https://hbr.org/2022/05/what-is-web3 (дата звернення: 02.02.2026).

2. Conflutech: «Thanks to Tim Berners-Lee for the happy Internaut Day». URL: https://conflutech.com/thanks-tim-berners-lee-happy-internaut-day/ (дата звернення: 05.02.2026).

3. Web3 Foundation. URL: https://web3.foundation/about/ (дата звернення: 05.02.2026).

4. Upwork. URL: https://www.upwork.com/i/how- it- works/client/ (дата звернення: 15.02.2026).

5. Fiverr. URL: https://www.fiverr.com/ (дата звернення: 15.02.2026).

6. Gitcoin – fund what matters. URL: https://gitcoin.co/ (дата звернення: 15.02.2026).

7. Toptal – hire talent from the top. URL: https://www.toptal.com/ (дата звернення: 15.02.2026).

8. Braintrust AIR – the best AI recruiting software to empower hr teams. URL: https://www.usebraintrust.com/ (дата звернення: 20.02.2026).

9. What is Angular. URL: https://angular.dev/overview (дата звернення: 22.02.2026).

10. Vue.js – the progressive JavaScript framework. URL: https://vuejs.org/ (дата звернення: 22.02.2026).

11. React. URL: https://react.dev/learn (дата звернення: 22.02.2026).

12. Solidity programming language. URL: https://www.soliditylang.org/ (дата звернення: 27.02.2026).

13. HalScience. URL: https://hal.science/hal-02108987/document (дата звернення 03.03.2026)

14. Supabase: the postgres development platform. URL: https://supabase.com/ (дата звернення: 10.03.2026).

15. Etherscan. URL: https://sepolia.etherscan.io/ (дата звернення: 18.03.2026)

16. Vercel: build and deploy the best web experiences with the AI Cloud. URL: https://vercel.com/ (дата звернення: 27.03.2026).

17. Ethereum Sepolia faucet. URL: https://cloud.google.com/application/web3/faucet/ethereum/sepolia (дата звернення: 05.04.2026).

18. Shapoval Agency: «Топ-7 законів у UX-дизайні». URL: https://cases.media/article/top-7-zakoniv-u-ux-dizaini?srsltid=AfmBOop4DA35dUA5RNmCjvWT2hqFloYfpdG9bneemqrb_6U5Pwsyumyc (дата звернення: 10.04.2026).

19. Key Lime Interactive: «What is the Keystroke-Level Model?». URL: https://info.keylimeinteractive.com/what-is-the-keystroke-level-model (дата звернення: 13.04.2026).

ДОДАТКИ

Додаток А

Сертифікат про участь у міжнародній науково-практичній конференції

Додаток Б

Фрагмент коду сторінки Profile.jsx

```
import { useEffect, useState } from "react";
import { useNavigate } from "react-router-dom";
import { supabase } from "../supabaseClient";
import { useWeb3 } from "../components/web3-context";
import "../styles/profile.css";

const Profile = () => {
  const navigate = useNavigate();
  const { walletAddress, connectWallet, disconnectWallet } = useWeb3();
  const [activeTab, setActiveTab] = useState("general");
  const [profile, setProfile] = useState(null);
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(true);
  const [transactions, setTransactions] = useState([]);
  const [newPassword, setNewPassword] = useState("");
  const [confirmPassword, setConfirmPassword] = useState("");
  const [supportSubject, setSupportSubject] = useState("");
  const [supportMessage, setSupportMessage] = useState("");
  const fetchTransactions = async (userId) => {
    const { data, error } = await supabase
      .from("payments")
      .select("*")
      .eq("user_id", userId)
      .order("created_at", { ascending: false });

    if (!error) {
      setTransactions(data);
    } else {
      console.error("Ми не змогли завантажити історію:", error);
    }
  };

  useEffect(() => {
    const getProfile = async () => {
      const {
        data: { session },
      } = await supabase.auth.getSession();

      if (!session) {
        navigate("/login");
        return;
      }

      setUser(session.user);
      const { data, error } = await supabase
        .from("profiles")
        .select("first_name, last_name, phone")
```

```
        .eq("id", session.user.id)
        .single();

      if (!error) setProfile(data);
      await fetchTransactions(session.user.id);

      setLoading(false);
    };

    getProfile();
  }, [navigate]);

  const handlePasswordUpdate = async (e) => {
    e.preventDefault();
    if (newPassword !== confirmPassword) {
      alert("Введені паролі не збігаються!");
      return;
    }
    const { error } = await supabase.auth.updateUser({ password:
newPassword });
    if (error) alert("Помилка оновлення: " + error.message);
    else {
      alert("Ми успішно оновили ваш пароль!");
      setNewPassword("");
      setConfirmPassword("");
    }
  };

  const handleSupportSubmit = async (e) => {
    e.preventDefault();
    const { error } = await supabase
      .from("support_tickets")
      .insert([
        { user_id: user.id, subject: supportSubject, message:
supportMessage },
      ]);

    if (error) alert("Виникла помилка при відправці.");
    else {
      alert("Ми отримали ваше повідомлення і скоро відповімо!");
      setSupportSubject("");
      setSupportMessage("");
    }
  };

  ……
};

export default Profile;
```