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

КВАЛІФІКАЦІЙНА РОБОТА

на тему

WEB 3.0 БЛОКЧЕЙН-ЗАСТОСУНОК ДЛЯ БЕЗПЕЧНИХ ПЕРЕКАЗІВ ТОКЕНІВ

Виконала: студентка групи 2П-22
Спеціальності
121 Інженерія програмного
забезпечення
Євгенія СКУБІЙ
Керівник:
Валентин ДМИТРЮК

Черкаси 2026

ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ ФАХОВИЙ БІЗНЕС-КОЛЕДЖ

Циклова комісія комп'ютерної інженерії та інформаційних технологій

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ

Завідувачка циклової комісії КІ та ІТ

_____ Наталя ФАЛЬЧЕНКО
(підпис)

«_____» _____2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТЦІ

Скубій Євгенії Віталіївні

1. Тема кваліфікаційної роботи Web 3.0 блокчейн-застосунок для безпечних переказів токенів

Керівник роботи Дмитрюк Валентин Віталійович

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

2. Строк подання студентом кваліфікаційної роботи 02.06.2026

3. Вихідні дані до кваліфікаційної роботи

Технології: Ethereum (Sepolia), Solidity, React 19, TypeScript, ethers.js, Hardhat. Стандарти: ISO/IEC/IEEE 29148, 29119, ISO/IEC 25010. API: CoinGecko, MetaMask. Інструменти тестування: Ganache, Slither, Playwright, Postman, Lighthouse.

4. Зміст кваліфікаційної роботи: аналіз предметної області та постановка задачі (технологія блокчейн, Ethereum, Web 3.0 і смартконтракти; аналіз рішень MetaMask, Uniswap, Coinbase Wallet; постановка задачі розробки Ethera); архітектура та реалізація застосунку Ethera (моделювання й архітектура системи; специфікація вимог; проєктування та реалізація смартконтракту й клієнтської частини; розгортання в мережі Sepolia); тестування програмного продукту(тестування безпеки, зручності та продуктивності).

5. Дата видачі завдання 15.09.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапу	Термін	Примітка
1	Вступ	14.10.2025	
2	Розділ 1. Аналіз предметної області та постановка задачі	09.12.2025	
3	Розділ 2. Архітектура та реалізація застосунку ETHERA	10.03.2026	
4	Розділ 3. Тестування програмного продукту	28.04.2026	
5	Висновки	12.05.2026	
6	Оформлення кваліфікаційної роботи (чистовий варіант)	24.05.2026	
7	Перевірка кваліфікаційної роботи на наявність ознак плагіату (за 10 днів до захисту)	02.06.2026	
8	Подання кваліфікаційної роботи на затвердження завідувачці циклової комісії (за 7 днів до захисту)	10.06.2026	

Студентка _____ Євгенія СКУБІЙ
(підпис)

Керівник роботи _____ Валентин ДМИТРЮК
(підпис)

АНОТАЦІЯ

У кваліфікаційній роботі здійснено розроблення децентралізованого вебзастосунку Ethera для безпечних переказів криптовалюти у мережі Ethereum. Виконано аналіз предметної області та порівняння наявних Web 3.0-рішень (MetaMask, Uniswap, Coinbase Wallet), що дозволило обґрунтувати архітектурні та технологічні рішення власного продукту. Розроблено чотиришарову архітектуру застосунку без централізованого сервера, спроєктовано та розгорнуто смартконтракт мовою Solidity у тестовій мережі Ethereum Sepolia, реалізовано клієнтську частину на React 19 з TypeScript. Сформульовано специфікацію вимог за стандартом ISO/IEC/IEEE 29148 з шістьма функціональними та трьома нефункціональними вимогами. Виконано комплексне тестування продукту з використанням Hardhat/Ganache, Slither, Playwright та Google Lighthouse, що підтвердило відповідність критеріям безпеки, зручності та продуктивності. Оцінка продуктивності за Lighthouse склала 99 балів зі 100. Результати роботи можуть бути використані як навчальний прототип Web 3.0-застосунку та основа для подальшого розвитку у напрямі decentralized finance (DeFi).

Робота складається з трьох розділів, містить 19 рисунків, 22 таблиці та список використаних джерел із 33 найменувань.

Ключові слова: WEB 3.0, ETHEREUM, СМАРТКОНТРАКТ, SOLIDITY, REACT, TYPESCRIPT, ETHERS.JS, METAMASK, ДЕЦЕНТРАЛІЗОВАНИЙ ЗАСТОСУНОК, БЛОКЧЕЙН, SEPOLIA.

ANNOTATION

This qualification thesis presents the development of a decentralized web application Ethera for secure cryptocurrency transfers on the Ethereum network. The study includes an analysis of the subject domain and a comparison of existing Web 3.0 solutions (MetaMask, Uniswap, Coinbase Wallet), which justified the architectural and technological decisions for the product. A four-layer serverless architecture was designed, a smart contract was implemented in Solidity and deployed to the Ethereum Sepolia testnet, and a client-side application was built using React 19 with TypeScript. Requirements were formalized according to ISO/IEC/IEEE 29148, covering six functional and three non-functional requirements. Comprehensive testing was performed using Hardhat/Ganache, Slither, Playwright, and Google Lighthouse, confirming compliance with security, usability, and performance criteria. The Lighthouse performance score reached 99 out of 100. The results may be used as an educational Web 3.0 prototype and a foundation for further development in the direction of decentralized finance (DeFi).

The thesis consists of three chapters and includes 19 figures, 22 tables, and a list of 33 references.

Keywords: WEB 3.0, ETHEREUM, SMART CONTRACT, SOLIDITY, REACT, TYPESCRIPT, ETHERS.JS, METAMASK, DECENTRALIZED APPLICATION, BLOCKCHAIN, SEPOLIA.

ЗМІСТ

ВСТУП

Швидка трансформація сучасних фінансових систем відбувається завдяки впливу цифрових технологій, коли централізовані моделі фінансової організації поступово замінюються децентралізованими. Поява технології блокчейн, а разом з нею – концепції смартконтрактів та криптовалют, відкрила якісно новий підхід до організації довірених цифрових транзакцій без участі фінансових посередників. У цьому контексті особливої актуальності набувають децентралізовані вебзастосунки (dApp), що реалізують принципи Web 3.0 і надають користувачам повний контроль над власними цифровими активами.

Актуальність теми зумовлена зростаючим попитом на інструменти безпечного переказу криптовалюти без залучення централізованих платіжних систем, а також необхідністю формування практичних інженерних компетенцій у галузі блокчейн-розробки. Попри широке розповсюдження існуючих рішень (MetaMask, Uniswap, Coinbase Wallet), жодне з них не поєднує в єдиному інтерфейсі функції безпечного переказу ETH, моніторингу ринку та управління гаманцем з орієнтацією на легкого клієнта без потреби у серверній інфраструктурі.

Ступінь дослідження теми. Теоретичні основи технології блокчейн висвітлені в технічних звітах NIST [1] та Ethereum White Paper В. Бутеріна [4]. Питання розробки смартконтрактів відображені в офіційній документації Solidity [3] та Ethereum Foundation [5]. Аспекти взаємодії клієнтської частини з блокчейном досліджено через документацію бібліотеки ethers.js та інтерфейс MetaMask [7]. Питання тестування програмного забезпечення розглядаються відповідно до стандарту ISO/IEC/IEEE 29119 [28].

Об'єкт дослідження – процес розробки децентралізованих вебзастосунків на базі технології блокчейн Ethereum з використанням смартконтрактів та інструментів Web 3.0.

Предмет дослідження – архітектура, методи проєктування та реалізації децентралізованого застосунку Ethera для безпечних переказів токенів у мережі Ethereum.

Мета роботи – розробка Web 3.0-застосунку Ethera для безпечного переказу криптовалюти у мережі Ethereum з орієнтацією на легкого клієнта без централізованого сервера та з перевіркою відповідності продукту визначеним критеріям якості.

Для досягнення поставленої мети визначено такі основні завдання:

– проаналізувати предметну область, ключові концепції технології блокчейн та існуючі Web 3.0-рішення;

– спроєктувати архітектуру системи та сформулювати специфікацію функціональних і нефункціональних вимог;

– розробити та розгорнути смартконтракт мовою Solidity у тестовій мережі Ethereum Sepolia;

– реалізувати клієнтську частину застосунку на React 19 з TypeScript та інтеграцією MetaMask і CoinGecko API;

– провести комплексне тестування безпеки (Hardhat/Ganache, Slither), зручності (Playwright, Postman) та продуктивності (Google Lighthouse) розробленого продукту.

Практична цінність роботи полягає у створенні працюючого програмного продукту – децентралізованого застосунку, смартконтракт якого розгорнуто у тестовій мережі Sepolia за адресою 0xCCBC6cCE543EDE95daCF5Eb1457466D2BB501457. Результати роботи демонструють повний інженерний цикл розробки dApp – від архітектурного проєктування до тестування і можуть бути використані як навчальний матеріал або основа для розробки production-застосунків у напрямі децентралізованих фінансів.

Кваліфікаційна робота складається зі вступу, трьох розділів, висновків, списку використаних джерел із 33 найменувань. Робота містить 19 рисунків та 22 таблиці.

Апробація результатів дослідження була здійснена в межах XVIII Всеукраїнської науково-практичної конференції студентів, аспірантів та молодих вчених «Тенденції розвитку IT-технологій в Україні» [34].

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Характеристика предметної області

У межах цієї кваліфікаційної роботи аналізується децентралізований веб-застосунок Ethera для безпечних переказів токенів у мережі Ethereum. Дослідження зосереджене на технічних та інженерних аспектах розробки: архітектурі програмної системи, проєктуванні смартконтракту, реалізації клієнтської частини та забезпеченні безпеки транзакцій. Питання комерціалізації, юридичного регулювання обігу криптовалют, оподаткування та фінансово-економічної моделі функціонування платформи виходять за межі цього дослідження та можуть бути предметом окремого розгляду.

З технічного погляду блокчейн є розподіленою базою даних, у якій записи організовані у послідовні блоки. Кожен наступний блок пов'язаний із попереднім через криптографічний хеш, завдяки якому підробка будь-якого запису є практично неможливою, адже для цього довелося б одночасно змінити всі наступні блоки на більшості вузлів мережі. Тому ця властивість блокчейну робить його загальнодоступним та надійний реєстром для фіксації фінансових операцій. Згідно з визначенням NIST, «блокчейни є цифровими реєстрами, реалізованими у розподіленій формі без централізованого репозиторію» [1].

Щоб гарантувати безпеку і незмінність запису про транзакцію, алгоритм блокчейна перетворює її на спеціальний криптографічний шифр – хеш. Кожен блок складається із заголовка та списку транзакцій. Заголовок містить:

– свій хеш;
– хеш попереднього блоку;
– хеш кожної транзакції;
– часову мітку, яка показує, коли він був сформований.

Структура ланцюга блоків наочно представлена на рисунку 1.1, де показано послідовне з'єднання блоків через криптографічні хеш-посилання.

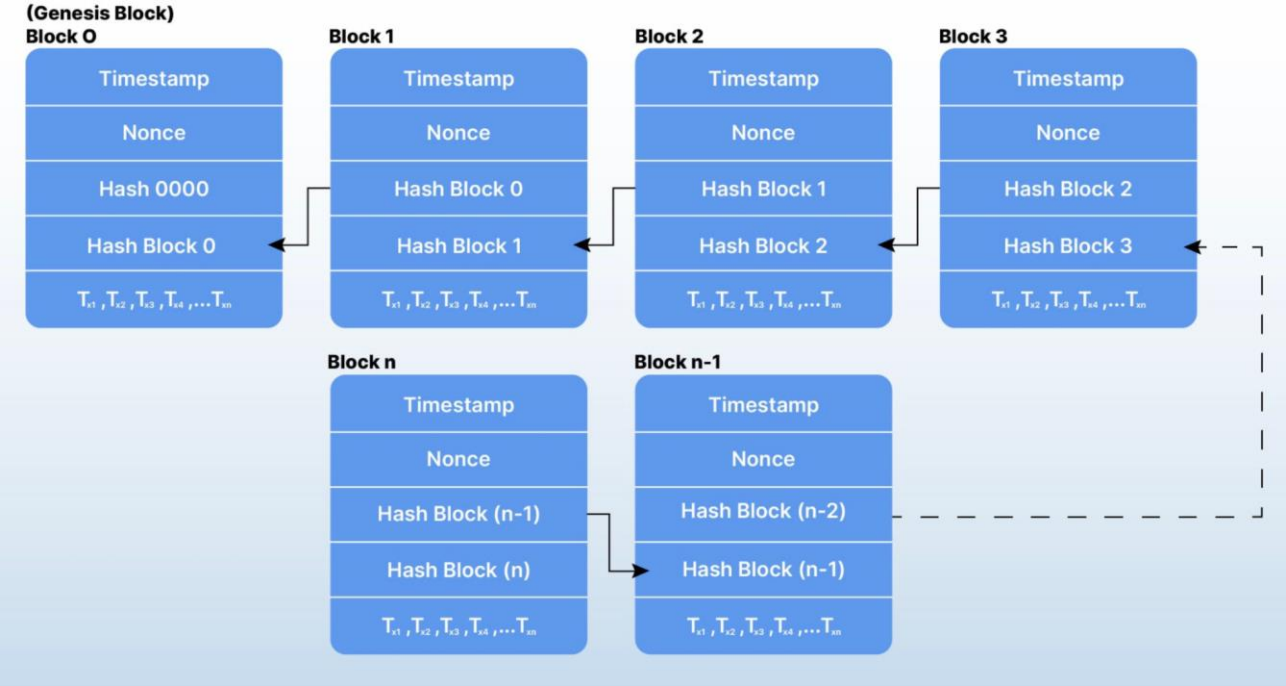

Рисунок 1.1 – Схема роботи блокчейну

Криптографічний зв’язок між блоками реалізується за допомогою хеш-функцій. У мережі Ethereum застосовується алгоритм Keccak-256, який приймає на вхід дані довільного розміру та повертає 256-бітне значення фіксованої довжини. Принцип роботи цієї хеш-функції полягає у тому, що навіть мінімальна зміна вхідних даних, наприклад, одного біта – призводить до повністю іншого результату хешування. Завдяки цьому будь-яка спроба підробки вмісту блоку миттєво виявляється – змінений хеш не збігається з тим, що зберігається у наступному блоці, і ланцюг блоків розривається. Хеш-функції також використовуються для генерації адрес гаманців, підписання транзакцій та обчислення коренів дерев Меркла, що забезпечує цілісність усіх даних у блокчейні.

Найбільш розвиненою платформою, що реалізує ці можливості, є Ethereum – блокчейн, концепт, якого був запропонований у 2013 році Віталіком Бутеріним, а мережа була запущена у 2015 році. У White Paper Ethereum В. Бутерін позиціонує платформу як «середовище для розробки децентралізованих застосунків» [4].

В той час, коли Bitcoin забезпечував лише передачу вартості, Ethereum став першою «програмованою» мережею, адже розробники отримали можливість не просто реєструвати транзакції, а й створювати довільну логіку їх виконання за допомогою смартконтрактів. Блокчейн Ethereum відповідає за зберігання та запис усіх даних транзакцій і смартконтрактів, відомих як «стан».

Смартконтракти – цифрові контракти, що зберігаються на блокчейні, які автоматично виконуються на основі заздалегідь визначених умов. На відміну від звичайних контрактів, які залежать від юридичної мови та посередників, смартконтракти використовують код для автоматичного виконання дій при дотриманні певних критеріїв.

Згідно з документацією Ethereum, «смартконтракти є програмами, що виконуються при настанні визначених умов» [5].

Нативна криптовалюта мережі Ethereum, Ether (ETH), слугує основною криптовалютою для живлення програм, побудованих на її блокчейні. За визначенням Kraken Learn Center, «криптовалюта є цифровим засобом обміну, захищеним криптографічними методами» [2]. Крім того, токени являють собою форму цифрових активів, які функціонують на основі концепції смартконтрактів. Основний стандарт для цих токенів відомий як ERC-20, який пропонує єдиний підхід до програмування, а також гарантує сумісність між різними сервісами. Токени, такі як USDT, USDC, DAI та LINK, використовують цей стандарт. Користувачі взаємодіють з мережею Ethereum, а також використовують MetaMask для взаємодії з токенами; це розширення для браузера та мобільний додаток, розроблений для забезпечення функцій, подібних до криптовалютного гаманця. Коли ви використовуєте MetaMask для доступу до вебдодатків, вони використовують інтерфейс window.ethereum для взаємодії з вами та виконання транзакцій. Ваші закриті ключі зберігаються на вашому пристрої у зашифрованому вигляді. Приватні ключі ніколи не надсилаються публічно під час підписання транзакцій на стороні клієнта. Ця модель гарантує користувачам повний контроль над їхніми коштами. Галузь дослідження, що охоплюється дипломним проєктом: децентралізована передача токенів між сторонами в

мережі Ethereum через веб-інтерфейс. Основні сутності та об'єкти взаємодії включають: кінцевих користувачів (особи, які використовують гаманець MetaMask для передачі токенів та/або моніторингу чи торгівлі/купівлі активів).

Смартконтракт у мережі Ether, написаний на Solidity та працює на Ethereum [3], може приймати платежі/транзакції, відстежувати всі минулі платежі/транзакції та не може бути змінений після запису.

Мережа Ether, по суті, є розподіленим реєстром, який дозволяє зберігати історію транзакцій, перевіряти ці транзакції та досягати консенсусу щодо них.

MetaMask, розширення для інтернет-браузера, служить цифровим гаманцем для закритих ключів користувачів і може підписувати транзакції, використовуючи ці закриті ключі [7].

CoinGecko – це зовнішній API, який дозволяє розробникам отримувати доступ до ринкових цін на криптовалюту в режимі реального часу [10].

Основна проблема, яку вирішує застосунок Ethera, полягає в нездатності забезпечити повністю децентралізований та простий веб-інтерфейс користувача для передачі токенів. Доступні рішення мають або обмежену функціональність, або іноді можуть бути досить складними для нетехнічного користувача, або ж вони зберігають деякі форми централізації. За допомогою застосунку Ethera всі взаємодії з блокчейном здійснюються через єдиний адаптивний веб-інтерфейс (застосунок Ethera).

Для формалізації взаємодії користувача із системою на концептуальному рівні застосовано підхід UML use case (діаграма прецедентів). Виділено наступних акторів: Користувач – фізична особа, що має MetaMask-гаманець; MetaMask – браузерне розширення, що виконує підпис транзакцій; Смартконтракт – програмний модуль у мережі Ethereum; CoinGecko API – зовнішній сервіс ринкових даних. Ключові прецеденти системи: «Підключити гаманець», «Відправити ETH», «Переглянути ринок криптовалют», «Здійснити обмін токенів», «Переглянути транзакційну історію», «Управляти гаманцем». Графічне представлення наведено на рисунку 1.2.

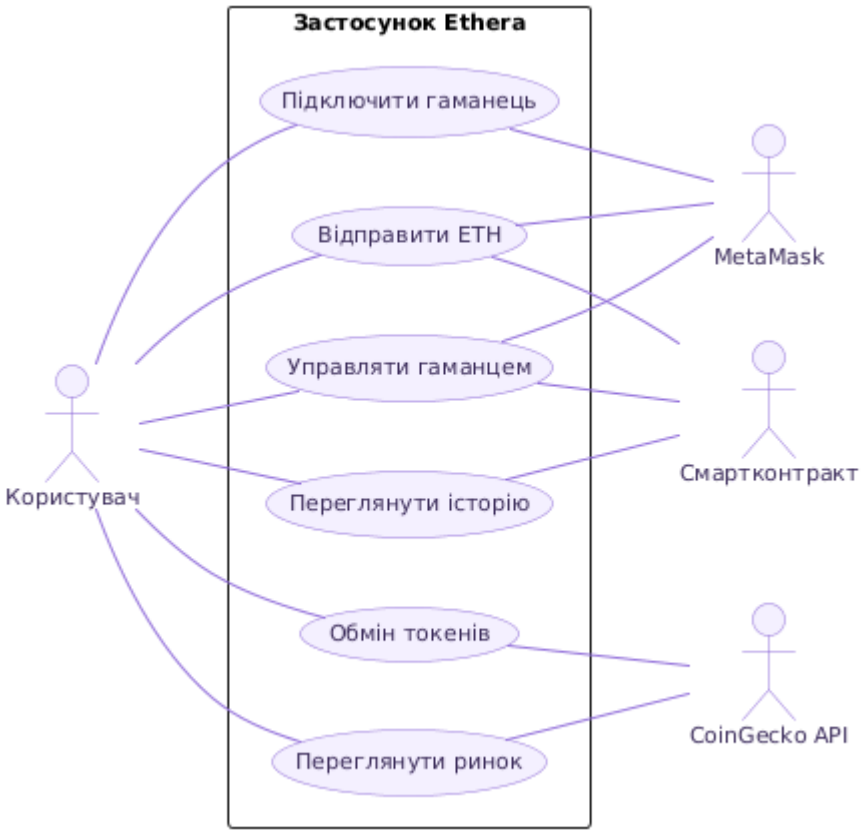

Рисунок 1.2 – UML-діаграма прецедентів застосунку Ethera

Для опису потоків даних у системі побудовано діаграму потоків даних (Data Flow Diagram), яка демонструє рух інформації між основними компонентами застосунку від моменту введення користувачем даних до фіксації транзакції у блокчейні. На рисунку 1.3 наведено схематичне представлення цього процесу: користувач вводить параметри транзакції у клієнтську частину застосунку, дані передаються до MetaMask для підпису приватним ключем, після чого підписана транзакція надсилається у мережу Ethereum, де викликається смартконтракт, який зберігає дані у блокчейні та повертає клієнту оновлений стан.

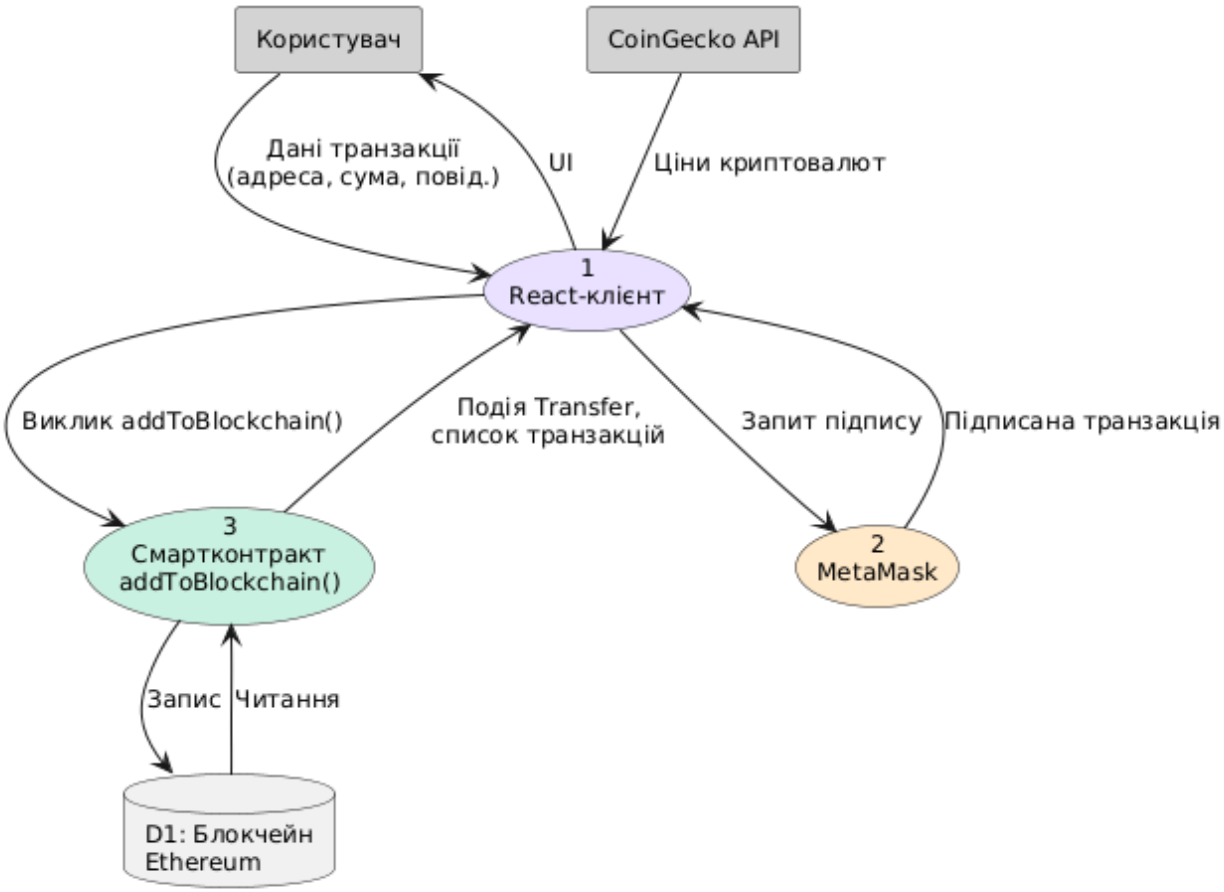

Рисунок 1.3 – Діаграма потоків даних (DFD) застосунку Ethera

1.2 Аналіз наявних рішень

Щоб довести практичність створення застосунку Ethera, було проведено порівняння трьох популярних пов'язаних вебзастосунків. Три пов'язані вебзастосунки – це MetaMask, Uniswap та Coinbase Wallet. Кожен застосунок є частиною певного типу застосунку Web 3.0, має певний набір функцій та технічних можливостей, а також має певну цільову аудиторію користувачів.

MetaMask – це найпоширеніший та найпопулярніший гаманець для браузера, коли ви хочете працювати з мережею Ethereum. Ви можете зберігати в ньому свої токени та підписувати транзакції, а також взаємодіяти з децентралізованими додатками. Але він функціонує лише як гаманець для управління. Це означає, що MetaMask не можна використовувати самостійно для переказу коштів або торгівлі токенами. Зовнішній вигляд користувацького інтерфейсу гаманця MetaMask наведено на рисунку 1.4.

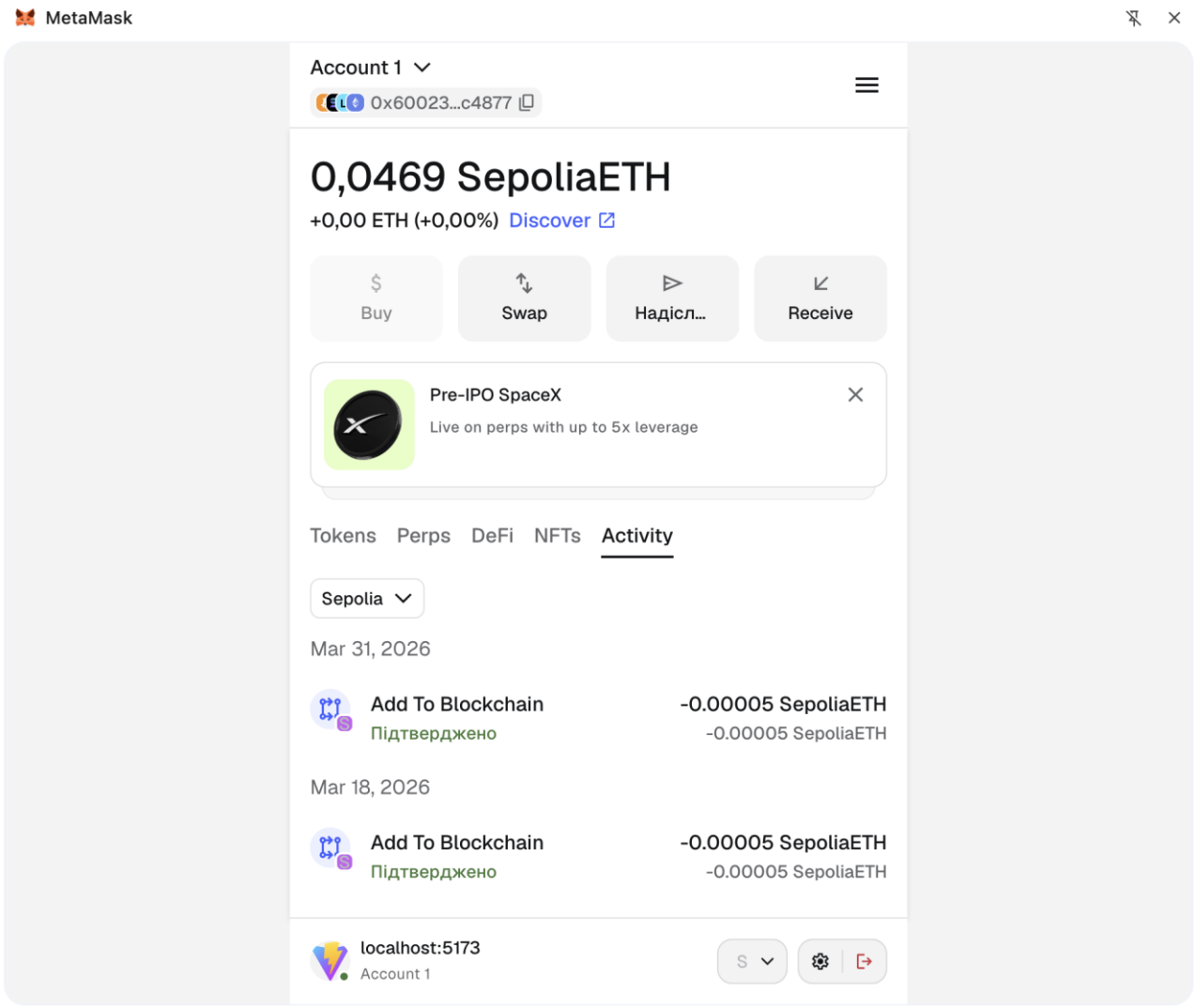

Рисунок 1.4 – Користувацький інтерфейс MetaMask

Інтерфейс користувача (UI) MetaMask в основному розроблений для користувачів, які мають досвід роботи на ринку криптовалют, головним чином через те, що користувачі повинні вручну вводити параметри ліміту газу та nonce для транзакцій через розширення браузера MetaMask, що ускладнює використання новачками, оскільки вони не мають вбудованих ринкових даних, вбудованого обміну токенами та не надають жодного способу перегляду коментарів до історії ваших транзакцій. [7]

Uniswap – це найбільша децентралізована біржа (DEX) на Ethereum, яка обмінює токени ERC-20 без централізованих посередників, використовуючи для цього механізм AMM. Механізм AMM допомагає Uniswap створювати ліквідність, дозволяючи користувачам розміщувати свої активи в пулі

ліквідності та торгувати проти цього пулу ліквідності. Остання версія біржі (Uniswap V3) надає користувачам концентровану ліквідність та конкурентні торгові ставки. Однак набір функцій DEX обмежується лише функціональністю DEX; немає можливості переказувати токени безпосередньо з доданим повідомленням, а також немає можливості переглядати історію транзакцій або отримувати доступ до комплексних інструментів управління гаманцями. Ще одна область, де можна покращити користувацький досвід – це інтерфейс платформи, оскільки він має численні налаштування, що ускладнюють для недосвідчених користувачів ефективну навігацію або використання платформи [8]. Інтерфейс децентралізованої біржі Uniswap представлено на рисунку 1.5.

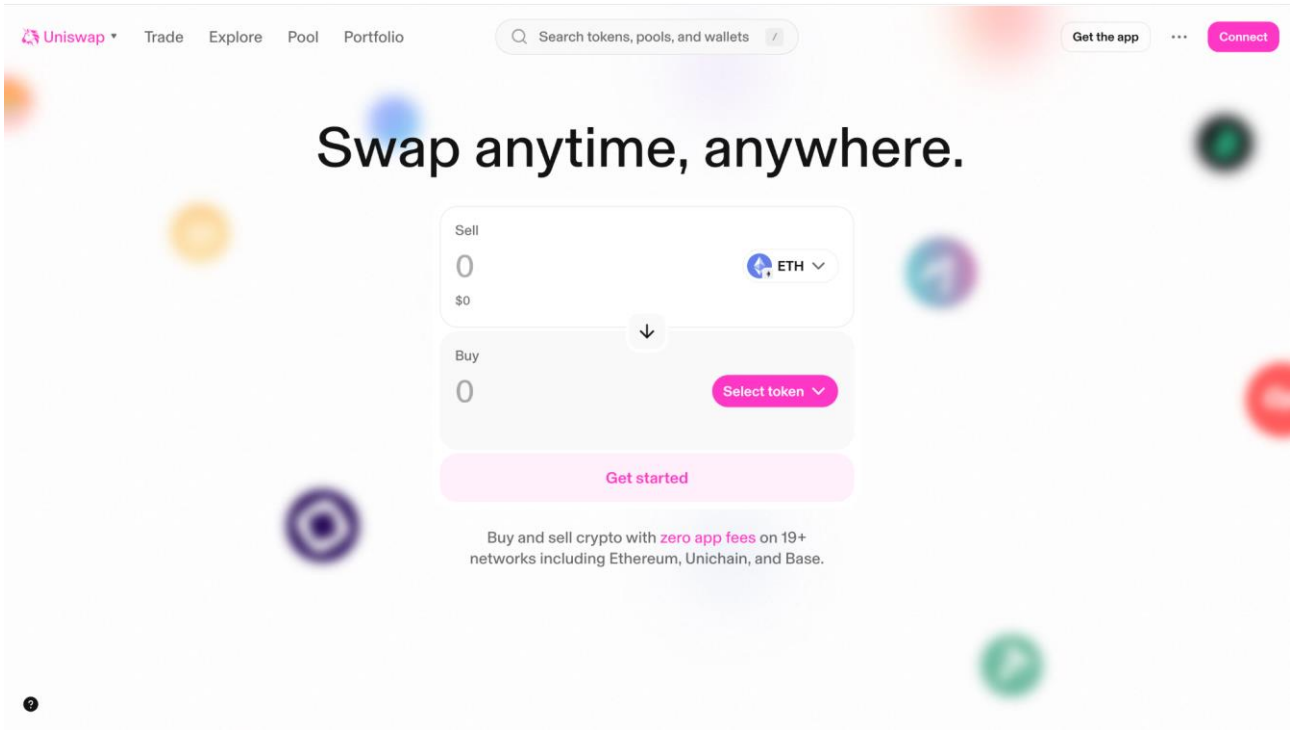

Рисунок 1.5 – Користувацький інтерфейс Uniswap

Coinbase Wallet розроблений Coinbase та працює як на телефонах, так і на комп'ютерах. Він сумісний з багатьма типами блокчейнів та токенів. Одна з головних функцій Coinbase Wallet – це простий спосіб побачити свій баланс та ринкову вартість, а також легко надсилати прості транзакції. Однак, Coinbase Wallet, напівцентралізоване рішення, збирає дані про поведінку користувачів та

аналітику і може мати доступ до деяких типів метаданих, і Coinbase залишає за собою право позбавити доступ до сервісу гаманця. Це суперечить фундаментальним ідеям децентралізації, які є частиною всієї концепції Web 3.0 [9]. Користувацький інтерфейс гаманця Coinbase Wallet продемонстровано на рисунку 1.6.

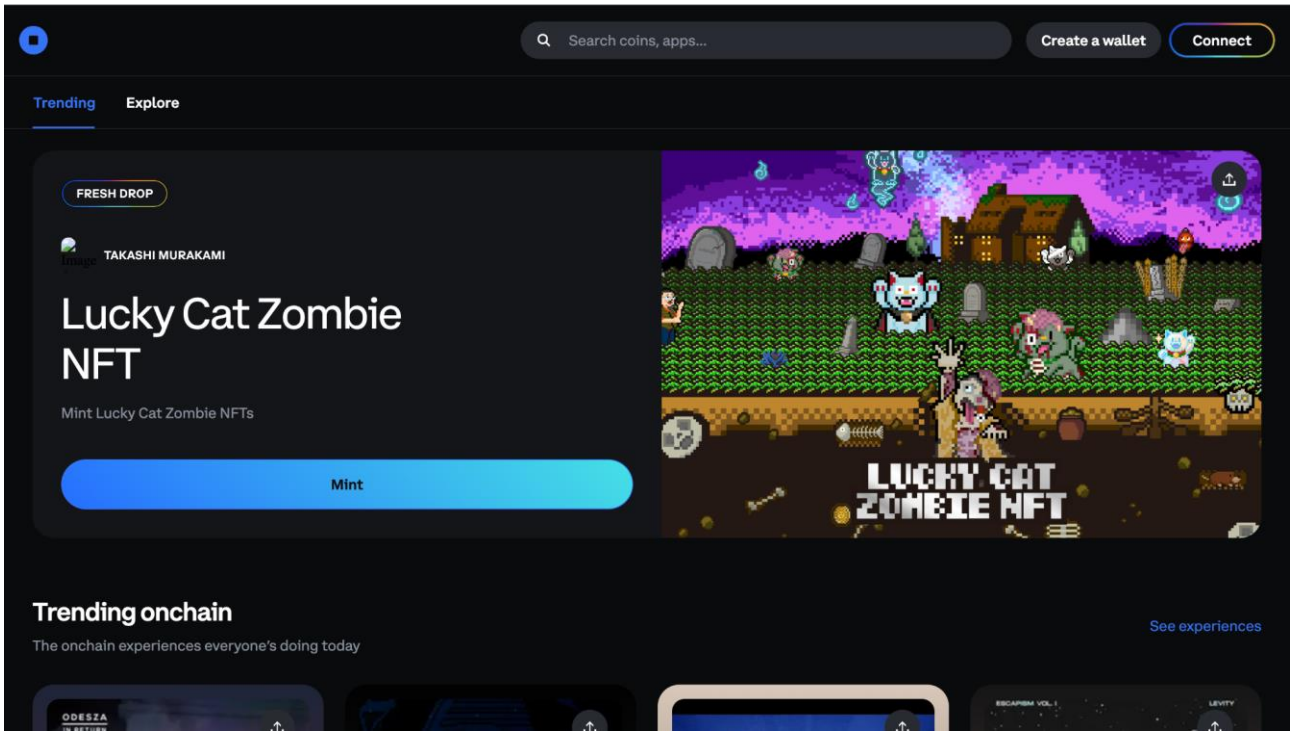

Рисунок 1.6 – Користувацький інтерфейс Coinbase Wallet

У таблиці 1.1 нижче порівнюються основні характеристики/функції двох рішень та застосунку Ethera.

Таблиця 1.1 – Порівняльний аналіз функціональних можливостей

Критерій	MetaMask	Uniswap	Coinbase Wallet	Ethera
Відправка токенів з повідомленням	Частково	Ні	Так	Так
Обмін токенів (DEX)	Ні	Так	Так	Так
Перегляд ринку в реальному часі	Ні	Частково	Так	Так
Спарклайн-графіки 7 днів	Ні	Ні	Ні	Так

Продовження таблиці 1.1

Транзакційна історія з повідомленнями	Ні	Ні	Так	Так
Управління гаманцем	Так	Ні	Так	Так
Повна децентралізація	Так	Так	Ні	Так
Відкритий вихідний код	Так	Так	Ні	Так
Зручний UI для початківців	Ні	Частково	Так	Так
Без реєстрації	Так	Так	Ні	Так

Для об'єктивного порівняння нефункціональних характеристик розглянутих рішень проведено якісну оцінку за основними атрибутами якості, яку подано у таблиці 1.2.

Таблиця 1.2 – Порівняння нефункціональних характеристик рішень

Атрибут	MetaMask	Uniswap	Coinbase Wallet	Ethera
Безпека даних	Висока	Висока	Середня	Висока
Зручність для новачків	Низька	Середня	Висока	Висока
Відкритість коду	Так	Так	Ні	Так
Швидкість завантаження	Висока	Середня	Висока	Висока
Незалежність від компанії	Так	Так	Ні	Так

Результати дослідження показали, що жоден із цих варіантів не надає всіх необхідних функцій, які включають: децентралізований спосіб переказу токенів (зі сповіщеннями), обмін, перегляд ринкових даних з графіками, а також можливість керування гаманцем через зручний інтерфейс веб-додатку (з відкритим кодом та повністю децентралізований). MetaMask служить лише інструментом підпису. Uniswap надає лише обмін, тоді як Coinbase Wallet, хоча й зручний, є централізованим і не має відкритого коду. Виявлені недоліки у функціональності та прогалини цих існуючих рішень є вагомими причинами для розробки застосунку Ethera.

Окрім порівняння функціональних можливостей, важливим аспектом обґрунтування технологічних рішень є аналіз інструментальних засобів розробки. Розглянуто провідні JavaScript-бібліотеки для взаємодії з мережею Ethereum: ethers.js, web3.js та viem. Результати порівняння наведено у табл. 1.3.

Таблиця 1.3 – Порівняння Web3-бібліотек для взаємодії з Ethereum

Критерій	ethers.js	web3.js	viem
Розмір (мінімізована)	~88 КБ	~135 КБ	~35 КБ
Підтримка TypeScript	Повна	Часткова	Повна
Документація	Якісна	Стандартна	Якісна
Tree-shaking	Так	Обмежено	Так
Активність розробки	Висока	Середня	Висока
Розмір спільноти	Велика	Найбільша	Зростає
Підтримка Web3-провайдерів	window.ethereum, RPC	window.ethereum, RPC	window.ethereum, RPC

За результатами порівняння для реалізації клієнтської частини застосунку Ethera обрано бібліотеку ethers.js. Цей вибір обґрунтований оптимальним балансом між розміром бібліотеки, повною підтримкою TypeScript, якісною документацією та активною спільнотою розробників. Бібліотека web3.js, попри більшу спільноту, має застаріший API та недостатню типізацію, що ускладнює розробку на TypeScript. Бібліотека viem є новішою альтернативою з кращою продуктивністю, проте на момент початку розробки мала менш зрілу екосистему.

Окремої уваги заслуговує вибір блокчейн-платформи для розгортання смартконтракту. У таблиці 1.4 подано порівняння провідних блокчейн-мереж за основними технічними характеристиками.

Таблиця 1.4 – Порівняння блокчейн-платформ

Платформа	Пропускна здатність (TPS)	Час підтвердження блоку	Середня комісія	Мова смартконтрактів
Ethereum	15–30	~12 с	$1–10	Solidity
Polygon	~7 000	~2 с	$0,01–0,10	Solidity
BSC	~100	~3 с	$0,10–0,30	Solidity
Solana	~3 000	~0,4 с	$0,001	Rust

Для реалізації застосунку обрано платформу Ethereum попри нижчу пропускну здатність порівняно з альтернативами. Цей вибір обумовлений

найбільшою екосистемою децентралізованих застосунків, найрозвиненішою інфраструктурою інструментів розробки (Hardhat, Remix, OpenZeppelin), широкою підтримкою з боку гаманців (зокрема MetaMask) та найбільшою кількістю активних користувачів. Розгортання смартконтракту здійснюється у тестовій мережі Sepolia, що дозволяє повноцінно тестувати функціональність без витрат реальних коштів. Підтримка Layer 2 рішень (Arbitrum, Optimism) розглядається як перспективний напрям подальшого розвитку застосунку для зниження вартості транзакцій без зміни кодової бази.

Таким чином, аналіз швидкодії та технічних характеристик платформ підтверджує, що Ethereum, незважаючи на нижчу пропускну здатність, є оптимальним вибором для освітнього застосунку, орієнтованого на розповсюдженість, безпеку та зручність використання, а не на максимальну швидкість обробки транзакцій.

1.3 Постановка задачі до кваліфікаційної роботи

Виходячи з оцінки ключових областей та слабких сторін існуючих програм, основною метою цієї кваліфікаційної роботи була розробка теоретичних та архітектурних концепцій і створення програмного застосунку для децентралізованого застосунку Web 3.0, відомого як "Ethera". Створений тут застосунок буде використовуватися для безпечної, прозорої та ефективної передачі цифрових активів (токенів) у мережі Ethereum. Користувачі матимуть повний доступ до блокчейну, тобто матимуть доступ до всіх функцій блокчейну, через простий у використанні інтерфейс користувача; всі транзакції будуть автоматизовані за допомогою смартконтрактів, і жодні банки, фінансові установи чи платіжні системи не виступатимуть посередниками. Цільовою групою користувачів застосунку є користувачі, які мають базові знання технології блокчейну і готові взяти на себе відповідальність за управління власною криптовалютою. Основними вимогами цієї групи користувачів є переказ місцевої валюти, переказ токенів ERC-20 у мережі блокчейну, моніторинг торговельної активності в реальному часі та участь у

децентралізованій біржовій торгівлі активами без необхідності передачі відповідальності за кошти стороннім посередникам.

Головним компонентом цієї програми є її орієнтація на «легкого клієнта». Користувачеві не потрібно мати запущене серверне програмне забезпечення на своєму комп'ютері, а також не потрібно завантажувати повний блокчейн; все, що потрібно, це мати доступ до веб-браузера та MetaMask як засіб для підписання транзакцій. Отже, основний обсяг робіт включає (а) розробку смартконтрактів з технічної точки зору та (б) створення інтерфейсу відповідно до останніх стандартів UX/UI, що використовуються в системах DeFi (децентралізованих фінансів).

У ході аналізу та проєктування системи свідомо прийнято рішення відмовитися від реалізації частини функціоналу, що розглядався на початковому етапі дослідження. Обґрунтування цих обмежень наведено нижче.

1 Авторизація через електронну пошту та пароль. На етапі формулювання вимог розглядалася можливість традиційного механізму авторизації з реєстрацією облікового запису, проте від такого підходу свідомо відмовлено на користь Web3-авторизації через MetaMask. Це рішення обумовлене кількома чинниками: по-перше, воно повністю відповідає принципам децентралізації Web 3.0 та усуває необхідність зберігання персональних даних на сервері; по-друге, унеможливлює витоки облікових даних, оскільки відсутня централізована база користувачів; по-третє, спрощує процес входу до застосунку – користувачеві достатньо одного кліку для підключення гаманця.

2 Інтеграція GIF-зображень через Giphy API. Структура смартконтракту передбачає поле keyword (ключове слово), яке зберігається разом із кожною транзакцією і має використовуватися для пошуку відповідного GIF-зображення через сервіс Giphy. Однак візуальне відображення GIF у клієнтській частині свідомо не реалізовано. Це рішення прийнято з огляду на те, що інтеграція потребує реєстрації API-ключа на сторонньому сервісі, не впливає на основну функціональність безпечних переказів токенів та виходить за межі мети

кваліфікаційної роботи. Поле keyword залишається у структурі контракту як точка розширення для подальшого розвитку застосунку.

Архітектурні та технологічні рішення застосунку визначаються рядом системних обмежень. Середовищем виконання є веббраузер із підтримкою JavaScript ES2020+ та встановленим розширенням MetaMask, яке забезпечує підписання транзакцій. Цільовою блокчейн-мережею обрано Sepolia Testnet [6], тоді як підтримка Layer 2 рішень [11], зокрема Arbitrum та Optimism, розглядається як перспективний напрям подальшого розвитку застосунку.

Для отримання актуальних ринкових даних використовується CoinGecko API, що працює у безкоштовному тарифному плані без необхідності API-ключа. Клієнтська частина застосунку реалізована на базі React 18 з TypeScript; як збирач модулів застосовується Vite, для стилізації – Tailwind CSS, для маршрутизації – React Router. Смартконтракт написаний мовою Solidity версії 0.8.18, розроблення та тестування здійснюються в середовищі Hardhat, а взаємодія з мережею Ethereum реалізована через бібліотеку ethers.js v6.

У першому розділі здійснено аналіз предметної області проєкту та постановку задачі для розробки децентралізованого вебзастосунку Ethera. Розглянуто базові концепції технології блокчейн, мережі Ethereum та смартконтрактів, що дозволило сформувати теоретичне підґрунтя для подальшої практичної реалізації. Визначено ключові технології, на яких базуватиметься застосунок: мову Solidity для написання смартконтракту, бібліотеку ethers.js для взаємодії клієнтської частини з блокчейном, гаманець MetaMask для підпису транзакцій та зовнішній сервіс CoinGecko API для отримання ринкових даних.

Проведено порівняльний аналіз трьох наявних рішень – MetaMask, Uniswap та Coinbase Wallet, який підтвердив доцільність створення власного застосунку, що поєднує функції безпечного переказу токенів, моніторингу ринку та управління гаманцем у єдиному інтерфейсі. За результатами порівняння Web3-бібліотек обрано ethers.js, а серед блокчейн-платформ – Ethereum, як найбільш зріле середовище з активною спільнотою розробників.

Сформульовано основну мету кваліфікаційної роботи – створення Web 3.0-застосунку для безпечних переказів токенів у мережі Ethereum з орієнтацією на легкого клієнта без необхідності завантаження повного блокчейну. Визначено цільову блокчейн-мережу (Sepolia Testnet), системні обмеження та свідомо обґрунтовано відмову від частини функціоналу (традиційна авторизація через email, інтеграція з Giphy API) на користь принципів децентралізації Web 3.0. Створене теоретичне та аналітичне підґрунтя дозволяє перейти до етапу архітектурного проєктування системи.

РОЗДІЛ 2 АРХІТЕКТУРА ТА РЕАЛІЗАЦІЯ ЗАСТОСУНКУ ETHERA

2.1 Концептуальне моделювання та архітектура системи

Після того як у першому розділі визначено, що має робити система, наступним кроком є формалізація того, як саме вона буде побудована: з яких логічних шарів складатиметься, які інтерфейси їх з'єднуватимуть, які принципи закладаються в основу проєктних рішень. Концептуальне моделювання у цьому контексті не повторює опис предметної області, а здійснює перехід від функціональних вимог до інженерної структури майбутнього програмного продукту.

Архітектура застосунку Ethera побудована за чотиришаровою моделлю, де кожен шар має чітко обмежену зону відповідальності та взаємодіє з суміжними шарами через визначені інтерфейси. Така організація узгоджується із загальноприйнятим інженерним принципом – поділом програмної системи на логічні горизонтальні рівні, що дозволяє вносити зміни до кожного з них незалежно від інших. Виділено такі шари: шар представлення (Presentation Layer), що відповідає за рендеринг інтерфейсу та обробку взаємодії з користувачем; шар управління станом (State Management Layer), який зберігає поточний стан гаманця, дані форми та історію транзакцій; шар взаємодії з блокчейном (Blockchain Integration Layer), що інкапсулює виклики до ethers.js та MetaMask; шар бізнес-логіки у блокчейні (Domain Layer), реалізований смартконтрактом і є єдиним джерелом істини щодо стану транзакцій.

Деталізація компонентів архітектури за шарами, технологіями реалізації та зонами відповідальності наведена у таблиці 2.1. На відміну від загального переліку акторів, поданого у першому розділі, ця таблиця фіксує внутрішню структуру застосунку з прив'язкою до конкретних модулів коду.

Таблиця 2.1 – Деталізація компонентів архітектури за шарами

Шар	Модулі реалізації	Зона відповідальності
Presentation Layer	Welcome, Market, Exchange, Wallets, Transactions, Tutorials	Рендеринг сторінок, обробка форм, валідація вхідних даних, відображення сповіщень користувачу
State Management Layer	TransactionContext (React Context API)	Зберігання адреси гаманця, поточних значень форми, лічильника та масиву транзакцій; поширення стану між компонентами без props drilling
Blockchain Integration Layer	ethers.js v6, window.ethereum API	Створення Provider та Signer, обгортка над ABI смартконтракту, підписка на події Transfer, обробка відмов користувача
Domain Layer	Смартконтракт Transactions.sol	Прийом параметрів транзакції, фіксація у блокчейні, зберігання історії у структурі TransferStruct, емісія подій

Запропонована рівнева організація формує основу для подальшої деталізації проєктних рішень. Наступним кроком моделювання є опис динамічної поведінки системи у часі. Для цього побудовано діаграму послідовності за нотацією, визначеною стандартом UML 2.5.1 [13], яка демонструє повний цикл операції відправки переказу ETH – від моменту натискання користувачем кнопки до підтвердження транзакції у блокчейні та оновлення інтерфейсу. Діаграму наведено на рисунку 2.1.

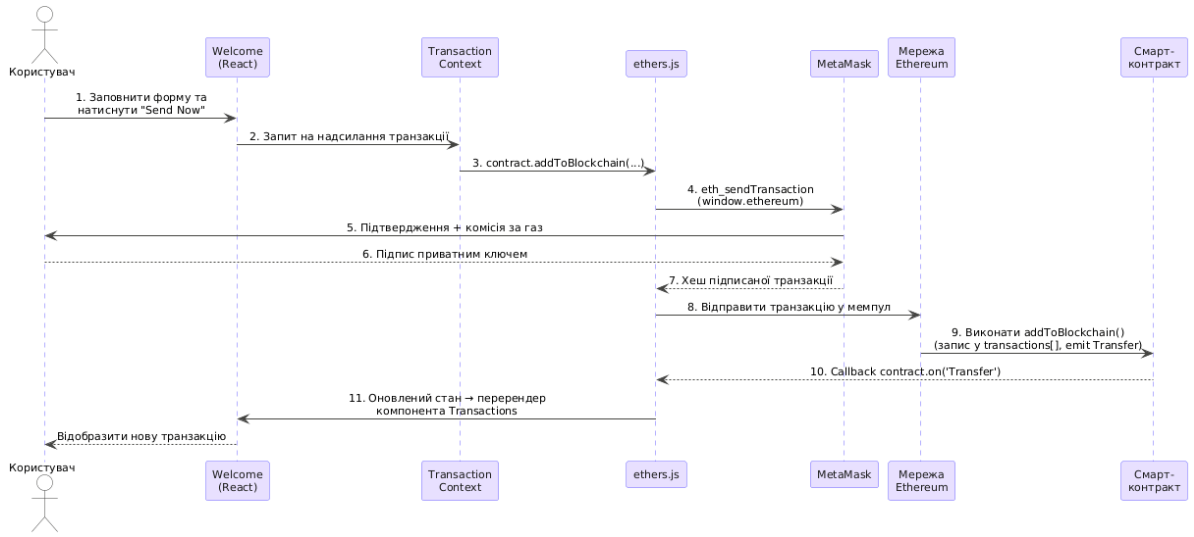

Рисунок 2.1 – Діаграма послідовності операції відправки ETH

Часовий перебіг операції охоплює одинадцять кроків взаємодії. Користувач заповнює форму та натискає кнопку «Send Now», після чого компонент Welcome звертається до TransactionContext із запитом на надсилання транзакції. Контекст викликає метод contract.addToBlockchain() через ethers.js, передаючи параметри транзакції. Бібліотека формує JSON-RPC запит eth_sendTransaction і передає його у MetaMask через інтерфейс window.ethereum. MetaMask відображає користувачу модальне вікно підтвердження з оцінкою комісії за газ, після чого підписує транзакцію приватним ключем і повертає її хеш. Транзакція потрапляє у мемпул мережі Ethereum, де очікує включення у наступний блок. Після підтвердження майнерами смартконтракт виконує функцію addToBlockchain(), записує дані у внутрішній масив transactions[], емітує подію Transfer і повертає контроль. Бібліотека ethers.js, попередньо підписана на цю подію через метод contract.on('Transfer'), отримує сповіщення і викликає callback у TransactionContext. Контекст оновлює локальний стан, після чого React автоматично перерендерює компонент Transactions з новою транзакцією без перезавантаження сторінки.

Описаний сценарій ілюструє важливу архітектурну особливість системи – її подієво-орієнтований характер. Замість періодичного опитування блокчейну щодо появи нових транзакцій клієнт реагує на повідомлення, які надсилає сам смартконтракт у вигляді подій. Такий підхід, відомий як event-driven architecture, є усталеною практикою для побудови систем з асинхронною взаємодією компонентів [14] і дозволяє знизити мережеве навантаження, прискорити реакцію інтерфейсу та узгодити поведінку клієнта з реактивною моделлю React. Підписка на події смартконтракту реалізується через метод contract.on('Transfer') бібліотеки ethers.js, а їх емісія – оператором emit у тілі функції addToBlockchain().

Окрім подієво-орієнтованої взаємодії, у системі застосовано низку інших архітектурних рішень, прийнятих з урахуванням компромісів між альтернативними варіантами. Для їх систематизації використано формат записів архітектурних рішень (Architecture Decision Records, ADR), запропонований М.

Нігардом [12] і прийнятий у сучасній інженерній практиці. Зведений перелік ключових рішень з обґрунтуванням і альтернативами наведено у таблиці 2.2.

Таблиця 2.2 – Ключові архітектурні рішення та їх обґрунтування

Рішення	Обґрунтування	Альтернатива (відхилена)
React Context API для управління станом	Достатньо для обсягу застосунку (5 сторінок, близько 10 одиниць стану); не вимагає додаткових залежностей	Redux Toolkit – надлишковий для поточного обсягу, додає бойлерплейт та збільшує bundle size
Event-driven оновлення UI через contract.on	Миттєва реакція на зміни у блокчейні, мінімальне навантаження на RPC-провайдер	Polling кожні 10 с – підвищує затримку та витрати RPC-квоти
ethers.js v6 як бібліотека Web3	Повна підтримка TypeScript, мінімальний розмір (близько 88 КБ), активна спільнота розробників	web3.js – застарілий API, слабша типізація; viem – менша екосистема на момент початку розробки
Компонентна структура за сторінками	Чітке відображення функціональних вимог на UI-модулі, легке додавання нових сторінок через React Router	Монолітний компонент App – ускладнює навігацію, порушує принцип Single Responsibility
Зберігання історії у масиві контракту	Простота реалізації для навчального застосунку; повна прозорість і верифіковуваність	The Graph (indexing protocol) – виправданий для production-навантаження, для прототипу надлишковий

Прийняті рішення не є безкоштовними з точки зору обмежень, які накладає на систему мережа Ethereum. Робота з блокчейном характеризується високою латентністю – час підтвердження блоку у мережі Sepolia становить близько 12–15 секунд – а також платністю кожної транзакції, що змінює стан смартконтракту. Це означає, що інтерфейс має враховувати очікування підтвердження, відображати індикатори завантаження та коректно обробляти випадки відмови користувача підписати транзакцію. Незмінність блокчейну виключає можливість відкату транзакції після її підтвердження, тому валідація даних має виконуватися ще на стороні клієнта до моменту виклику смартконтракту. Зведення архітектурних обмежень та проєктних відповідей на них наведено у таблиці 2.3.

Таблиця 2.3 – Архітектурні обмеження та проєктні відповіді

Обмеження	Проєктна відповідь
Висока латентність підтвердження транзакції (12–15 с)	Відображення індикатора Loader після підписання; блокування повторного натискання кнопки до отримання відповіді мережі
Платність кожної транзакції (gas)	Функції getAllTransactions() та getTransactionCount() реалізовано як view-функції (без газу); запис у блокчейн виконує тільки функція addToBlockchain()
Незмінність даних після підтвердження	Клієнтська валідація формату адреси (regex 0x[0-9a-fA-F]{40}) та позитивної суми до виклику контракту
Можливість відмови користувача підписати транзакцію	Обробка винятку MetaMask error code 4001 (user rejected); скидання стану форми та сповіщення користувача
Залежність від доступності RPC-провайдера	Використання Alchemy як основного RPC-провайдера; обробка таймаутів у Blockchain Integration Layer через try-catch

Кінцевим результатом концептуального моделювання є архітектурна модель, у якій кожна функціональна вимога, сформульована у першому розділі, отримує відповідне технічне втілення на одному з логічних шарів. Зокрема, вимога підключення гаманця реалізується через Blockchain Integration Layer (запит eth_requestAccounts), збереження стану підключення – через State Management Layer, відображення скороченої адреси – через Presentation Layer (компонент Navbar). Аналогічним чином декомпозовано всі шість функціональних вимог, що забезпечує повне покриття функціональності проєктними рішеннями. Ця декомпозиція є основою для подальшого проєктування смартконтракту, специфікації якого присвячено наступний підрозділ.

2.2 Специфікація вимог до програмного забезпечення

Систематизація вимог до програмного продукту є однією з ключових інженерних практик, що забезпечує однозначне розуміння цілей розробки всіма учасниками процесу та створює основу для подальшого тестування і верифікації результатів. У цьому підрозділі подано формалізовану специфікацію вимог до застосунку Ethera, побудовану відповідно до підходу, рекомендованого

міжнародним стандартом ISO/IEC/IEEE 29148:2018 [15], який визначає процеси інженерії вимог у життєвому циклі програмних систем. Класифікацію нефункціональних вимог здійснено за моделлю характеристик якості ISO/IEC 25010 [16].

Функціональні вимоги визначають конкретні дії та поведінку, які має реалізовувати програмний продукт для задоволення потреб користувачів. На основі сценаріїв використання, сформульованих у першому розділі, виділено шість функціональних вимог, наведених нижче за схемою: вхідні дані, опис обробки, результат.

ФВ-1. Підключення гаманця MetaMask. Вхідні дані: натискання користувачем кнопки «Connect Wallet» у компоненті Navbar. Опис обробки: клієнтська частина застосунку виконує запит eth_requestAccounts до інтерфейсу window.ethereum, після чого MetaMask відкриває модальне вікно з пропозицією обрати акаунт. Результат: адреса підключеного гаманця зберігається у TransactionContext та відображається у скороченій формі (перші 4 та останні 4 символи) у навігаційній панелі застосунку.

ФВ-2. Відправка ETH з повідомленням. Вхідні дані: адреса одержувача у форматі 0x[40 hex-символів], сума переказу у ETH (більше нуля та не більша за поточний баланс), ключове слово та текстове повідомлення. Опис обробки: клієнт виконує валідацію формату адреси та позитивності суми, після чого викликає функцію смартконтракту addToBlockchain(), яка приймає підписану транзакцію через MetaMask. Результат: транзакція зберігається у блокчейні у вигляді структури TransferStruct та відображається у списку транзакцій з посиланням на Etherscan для перевірки.

ФВ-3. Перегляд ринку криптовалют. Вхідні дані: відсутні (запит ініціюється автоматично при завантаженні сторінки Market). Опис обробки: клієнт виконує GET-запит до CoinGecko API для отримання даних топ-20 криптовалют за ринковою капіталізацією: поточної ціни, зміни за 24 години, обсягу торгів та масиву значень для побудови спарклайн-графіка за 7 днів.

Результат: користувач бачить сітку карток з актуальними ринковими даними та інтерактивними міні-графіками; передбачено пошук за назвою або символом.

ФВ-4. Обмін токенів. Вхідні дані: вибір пари токенів (ETH, USDT, BTC, BNB, SOL, MATIC), сума токена-джерела, рівень допуску ковзання (0,1 %, 0,5 % або 1,0 %). Опис обробки: клієнт отримує поточний курс обмінюваної пари з CoinGecko API, обчислює розрахункову суму отримання з урахуванням slippage та відображає мережеву комісію. Результат: користувач бачить сформовану форму обміну з усіма параметрами; натискання кнопки «Swap» демонструє інформаційне повідомлення про необхідність інтеграції з DEX-протоколом для production-версії.

ФВ-5. Перегляд транзакційної історії. Вхідні дані: відсутні (історія завантажується автоматично при підключенні гаманця). Опис обробки: клієнт викликає view-функцію смартконтракту getAllTransactions(), яка повертає масив структур TransferStruct з усією історією транзакцій застосунку; для кожної адреси формується посилання на Etherscan. Результат: відображення карток транзакцій у зворотному хронологічному порядку з адресами відправника та одержувача, сумою, повідомленням та часом блоку.

ФВ-6. Управління гаманцем. Вхідні дані: адреса підключеного гаманця з TransactionContext. Опис обробки: клієнт виконує запит eth_getBalance до RPC-провайдера для отримання поточного балансу ETH; обчислюється загальна кількість транзакцій користувача через getTransactionCount(). Результат: відображення повної адреси гаманця з можливістю копіювання у буфер обміну, балансу ETH у форматі з фіксованою кількістю десяткових знаків та статусу підключення до мережі.

Нефункціональні вимоги визначають якісні характеристики системи, що не пов’язані безпосередньо з її функціями, проте істотно впливають на досвід використання та надійність продукту. Для подальшої формалізації обрано три атрибути якості зі стандарту ISO/IEC 25010 [16], які найбільш суттєво впливають на цільові сценарії застосунку Ethera та піддаються вимірюванню: безпека, зручність та продуктивність. Такий обмежений набір атрибутів обрано свідомо –

це дозволяє чітко сформулювати критерії перевірки кожного з них на етапі тестування у третьому розділі роботи. Перелік нефункціональних вимог наведено у таблиці 2.4.

Таблиця 2.4 – Нефункціональні вимоги до застосунку Ethera

Атрибут якості	Вимога
Безпека	Приватний ключ користувача зберігається виключно в MetaMask і ніколи не передається у код клієнтської частини застосунку. Програмний код смартконтракту не повинен містити вразливостей високого або критичного рівня небезпеки за результатами автоматизованого аудиту.
Зручність	Авторизація користувача здійснюється через MetaMask без процедури реєстрації, введення електронної пошти або паролю. Інтерфейс адаптивний і коректно відображається на пристроях з шириною екрана від 320 px (мобільний телефон) до 2560 px (desktop-монітор).
Продуктивність	Метрика Time to Interactive (TTI) – час, протягом якого сторінка стає повністю інтерактивною – не перевищує 3 секунди на з'єднанні з пропускною здатністю 10 Мбіт/с. Реакція інтерфейсу на події смартконтракту відбувається без перезавантаження сторінки.

Розгорнутий екземпляр застосунку має набір унікальних ідентифікаторів, які однозначно визначають його розташування у блокчейн-мережі та дозволяють третім особам верифікувати коректність роботи смартконтракту. Систематизовану інформацію про ідентифікаційні дані наведено нижче.

Адреса розгорнутого смартконтракту: 0xCCBC6cCE543EDE95daCF5Eb1457466D2BB501457. Ця адреса є унікальним ідентифікатором смартконтракту у мережі Ethereum і використовується клієнтською частиною застосунку для виклику його функцій через бібліотеку ethers.js.

Цільова мережа: Sepolia Testnet (ідентифікатор Chain ID 11155111). Sepolia є офіційною тестовою мережею Ethereum, призначеною для розробки та тестування смартконтрактів без використання реальних коштів. Вибір тестової мережі обумовлений освітнім характером роботи та необхідністю забезпечити можливість відтворення сценаріїв використання без фінансових витрат для рецензента або користувача.

Постачальник RPC-вузла: Alchemy. Сервіс надає публічну точку доступу до мережі Sepolia через протокол JSON-RPC, що використовується ethers.js для виконання запитів читання стану блокчейну та надсилання підписаних транзакцій.

Інструмент верифікації: Etherscan для мережі Sepolia (https://sepolia.etherscan.io). За цим посиланням з підстановкою адреси контракту можна переглянути всі здійснені транзакції, перевірити верифікований вихідний код смартконтракту та проаналізувати емітовані події Transfer.

Для уникнення неоднозначності у трактуванні того, коли функціональна вимога вважається повністю реалізованою, для кожної вимоги визначено набір критеріїв готовності (Definition of Done). Критерії готовності – це перевірювані твердження, після підтвердження яких можна вважати, що відповідна вимога вважається виконаною. Перелік критеріїв для всіх шести функціональних вимог наведено у таблиці 2.5.

Таблиця 2.5 – Критерії готовності функціональних вимог

Вимога	Критерії готовності
ФВ-1	Натискання кнопки «Connect Wallet» відкриває модальне вікно MetaMask; після підтвердження користувачем у navbar відображається скорочена адреса; повторне відкриття сторінки зберігає стан підключення без додаткового запиту.
ФВ-2	Транзакція з невалідною адресою або нульовою сумою блокується ще до виклику смартконтракту; після успішного підпису у MetaMask нова транзакція з’являється у списку транзакцій; посилання на Etherscan відкриває сторінку транзакції у блок-експлорері.
ФВ-3	При відкритті сторінки Market відображаються картки 20 криптовалют з актуальними даними; кожна картка містить спарклайн-графік за 7 днів; пошук за назвою або символом коректно фільтрує сітку.
ФВ-4	Зміна пари токенів автоматично перераховує суму отримання за поточним курсом; зміна рівня slippage відображається на розрахунковій сумі; інтерфейс показує мережеву комісію та підсумкову суму.
ФВ-5	При підключенні гаманця історія транзакцій завантажується з контракту; новостворена транзакція з ФВ-2 з’являється у списку без перезавантаження сторінки; кожна адреса має клікабельне посилання на Etherscan.
ФВ-6	На сторінці Wallets відображається баланс ETH з точністю до п’яти десяткових знаків; повна адреса гаманця копіюється у буфер обміну за одним натисканням; статус підключення оновлюється при зміні мережі у MetaMask.

На відміну від критеріїв готовності, які стосуються функціональної повноти, критерії забезпечення якості описують вимірювані характеристики, що підтверджують відповідність системи нефункціональним вимогам. Для кожного з трьох обраних атрибутів якості (безпека, зручність, продуктивність) сформульовано конкретні перевірювані критерії, наведені у таблиці 2.6. Підтвердження відповідності цим критеріям виконується інструментально: статичний аналіз смартконтракту за допомогою Slither [17], який є провідним відкритим інструментом аудиту контрактів мови Solidity; інспекція клієнтського коду на наявність захардкоджених приватних ключів; вимірювання метрики Time to Interactive за допомогою інструменту Lighthouse, де TTI визначається як момент, після якого основний потік браузера може реагувати на дії користувача [18].

Таблиця 2.6 – Критерії забезпечення якості

Атрибут	Критерій	Інструмент перевірки
Безпека	Статичний аналіз смартконтракту не виявляє вразливостей категорії High або Critical. У клієнтському коді відсутні захардкоджені приватні ключі, секрети або токени API.	Slither для смартконтракту; ручна інспекція репозиторію та git-історії
Зручність	Підключення гаманця виконується не більш ніж за два кліки користувача (Connect → Підтвердити). Основні CTA-кнопки мають атрибут aria-label для скрін-рідерів. Інтерфейс зберігає функціональність на екранах від 320 px шириною.	Ручне тестування у мобільному та desktop-режимі браузера; інспектор DOM-розмітки
Продуктивність	TTI ≤ 3 с на з'єднанні 10 Мбіт/с; розмір зібраного клієнтського bundle після gzip-стиснення ≤ 500 КБ Розмір зібраного клієнтського bundle після gzip-стиснення не перевищує 500 КБ у мінімізованому вигляді.	Google Lighthouse для TTI; команда vite build та звіт rollup-plugin-visualizer для розміру bundle

Сформульований комплекс вимог, критеріїв готовності та критеріїв забезпечення якості створює завершену специфікацію застосунку, на основі якої будується подальша архітектурна та реалізаційна робота. Підтвердження

відповідності продукту визначеним критеріям буде здійснене у третьому розділі роботи у межах процедур тестування та верифікації.

2.3 Проєктування смартконтракту

Смартконтракт є основою серверної частини застосунку Ethera та виконує роль довіреного посередника, що приймає, зберігає та надає доступ до даних усіх транзакцій без участі централізованого сервера. Згідно з документацією мови Solidity, контракт у її термінології є набором програмного коду та постійних даних, який розгортається за конкретною адресою в мережі Ethereum і може бути викликаний транзакціями зовнішніх облікових записів [19]. На цьому концептуальному рівні смартконтракт виступає аналогом класу в об'єктно-орієнтованому програмуванні він поєднує функції (поведінку) та змінні стану (дані), які зберігаються у блокчейні.

Контракт застосунку реалізовано у файлі Transactions.sol мовою Solidity версії 0.8.18. У структурі контракту виділено три основні складові: визначення власної структури даних TransferStruct для зберігання інформації про транзакцію; набір змінних стану – лічильник транзакцій transactionCount та масив усіх зафіксованих транзакцій transactions; декларацію події Transfer та три функції – addToBlockchain, getAllTransactions, getTransactionCount. Загальний скелет контракту представлено у лістингу 2.1.

Лістинг 2.1 – Скелет смартконтракту Transactions.sol

```
// SPDX-License-Identifier: UNLICENSED
 pragma solidity ^0.8.18;

 contract Transactions {
     uint256 transactionCount;

     event Transfer(
         address from,
         address receiver,
         uint amount,
         string message,
         uint256 timestamp,
         string keyword
     );
```

Продовження лістингу 2.1

```
    struct TransferStruct {
      address sender;
      address receiver;
      uint amount;
      string message;
      uint256 timestamp;
      string keyword;
    }

    TransferStruct[] transactions;

    // ... функції addToBlockchain, getAllTransactions, getTransactionCount
}
```

Ключовим елементом моделі даних смартконтракту є власна структура TransferStruct. У Solidity структури належать до категорії reference-типів і дозволяють об'єднати кілька значень різних типів у єдиний логічний запис, що зберігається у пам'яті блокчейну [20]. Така організація дозволяє зберігати у блокчейні не лише суму та адреси, але й супутні метадані транзакції – повідомлення та ключове слово, що відрізняє Ethera від типових застосунків переказу криптовалют. Деталізацію полів структури наведено у таблиці 2.7.

Таблиця 2.7 – Поля структури TransferStruct

Поле	Тип Solidity	Призначення
sender	address	Адреса гаманця, з якого ініційовано переказ; автоматично встановлюється у значення msg.sender
receiver	address payable	Адреса одержувача переказу; тип payable дозволяє надсилати на цю адресу ETH
amount	uint	Сума переказу у мінімальних одиницях ETH (wei); 1 ETH = 10^{18} wei
message	string	Текстове повідомлення, що супроводжує переказ і пояснює його контекст
timestamp	uint256	Часова мітка блоку у форматі Unix timestamp; встановлюється значенням block.timestamp
keyword	string	Ключове слово, передбачене для подальшого розширення (інтеграція з Giphy API для відображення GIF)

Усі екземпляри структури TransferStruct зберігаються у динамічному масиві transactions, оголошеному як змінна стану контракту. Масив автоматично розширюється при додаванні кожної нової транзакції; його поточний розмір

відображається у змінній transactionCount, що дозволяє швидко отримати загальну кількість транзакцій без проходу всім масивом.

Для реалізації подієво-орієнтованої взаємодії з клієнтською частиною у смартконтракті оголошено подію Transfer. Згідно з документацією Solidity, події є абстракцією над логуючою функціональністю Ethereum Virtual Machine; коли подія викликається оператором emit, її аргументи зберігаються у структурі логів транзакції, доступній для зовнішніх застосунків через JSON-RPC-інтерфейс Ethereum-клієнта [19]. Дані подій недоступні для читання з інших смартконтрактів, проте їх можна ефективно фільтрувати та обробляти на стороні клієнта.

Сигнатура події Transfer відтворює структуру TransferStruct – це дозволяє при отриманні події у клієнтській частині одразу мати повний набір даних про транзакцію без додаткового запиту до блокчейну. Підписка на подію реалізована у клієнтській частині через виклик contract.on('Transfer', callback) бібліотеки ethers.js, що забезпечує миттєве оновлення інтерфейсу при появі нової транзакції у мережі. Така архітектурна модель є рекомендованою практикою побудови взаємодії між смартконтрактом та зовнішніми застосунками і дозволяє знизити кількість викликів читання стану блокчейну.

Смартконтракт містить три функції з різними рівнями впливу на стан блокчейну. Функція addToBlockchain здійснює зміну стану (запис нової транзакції), тоді як getAllTransactions та getTransactionCount є view-функціями, що лише читають стан без модифікації. Згідно з рекомендаціями офіційної документації Ethereum, винесення операцій читання у view-функції дозволяє виконувати їх безкоштовно для користувача – без витрат на газ [21]. Такий поділ функцій є важливим архітектурним рішенням з точки зору економії коштів користувача.

Функція addToBlockchain є основною функцією смартконтракту. Вона приймає чотири параметри: адресу одержувача, суму переказу, повідомлення та ключове слово. У тілі функції здійснюється інкрементація лічильника, додавання нового елемента до масиву transactions через створення TransferStruct з даних

параметрів, поточної адреси відправника (msg.sender) та часової мітки блоку (block.timestamp), а також емісія події Transfer. Лістинг функції наведено нижче.

Лістинг 2.2 – Функція addToBlockchain

```
function addToBlockchain(
     address payable receiver,
     uint amount,
     string memory message,
     string memory keyword
 ) public {
     transactionCount += 1;
     transactions.push(TransferStruct(
         msg.sender,
         receiver,
         amount,
         message,
         block.timestamp,
         keyword
     ));

     emit Transfer(
         msg.sender,
         receiver,
         amount,
         message,
         block.timestamp,
         keyword
     );
 }
```

Функція getAllTransactions повертає повний масив всіх зафіксованих транзакцій. Тип повернення TransferStruct[] memory означає копіювання даних у тимчасову пам’ять при виклику, що є необхідним для повернення reference-типу із зовнішнього виклику. Ключове слово view вказує компілятору, що функція лише читає стан і не модифікує його – це гарантує безкоштовність виклику для зовнішніх облікових записів.

Лістинг 2.3 – Функції читання стану

```
function getAllTransactions()
     public view returns (TransferStruct[] memory) {
     return transactions;
 }

 function getTransactionCount()
```

Продовження лістингу 2.3

```
    public view returns (uint256) {

 return transactionCount;
}
```

Функція getTransactionCount повертає значення лічильника transactionCount – загальну кількість транзакцій у блокчейні. Вона використовується клієнтською частиною для відображення статистики на сторінці Wallets та для попереднього виділення пам’яті при завантаженні історії транзакцій.

Для візуалізації структури смартконтракту побудовано UML-діаграму класів, наведену на рисунку 2.2. Діаграма відображає контракт Transactions як основну одиницю проєктування з трьома секціями: змінні стану (transactionCount, transactions), оголошена подія Transfer та публічні функції з їхніми сигнатурами. Окремим класом представлено структуру TransferStruct, пов’язану з контрактом відношенням композиції – структура є інтегральною частиною контракту та не може існувати поза ним.

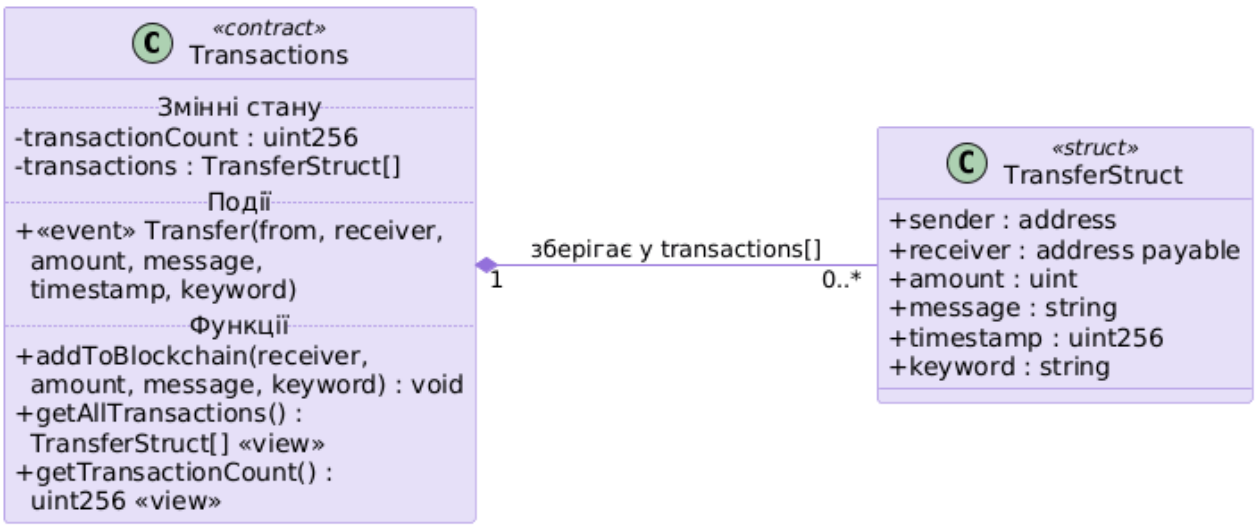

Рисунок 2.2 – UML-діаграма класів смартконтракту Transactions

Розроблений смартконтракт є мінімально достатньою реалізацією для виконання основної функціональності застосунку – фіксації переказів з

повідомленнями. Його простота є свідомою архітектурною позицією: чим менший і прозоріший код контракту, тим легше провести його аудит та підтвердити відсутність вразливостей. Перевірка контракту на відповідність критеріям якості, сформульованим у підрозділі 2.2, виконуватиметься у третьому розділі роботи. Особливості реалізації клієнтської частини, що взаємодіє з цим контрактом через бібліотеку ethers.js, розглянуто у наступному підрозділі.

2.4 Проєктування клієнтської частини

Клієнтська частина застосунку Ethera є єдиною точкою взаємодії користувача із системою та відповідає за відображення інтерфейсу, обробку дій користувача, формування запитів до смартконтракту та зовнішніх API. У межах багаторівневої архітектури, описаної у підрозділі 2.1, клієнт реалізує одразу три рівні: рівень представлення, рівень управління станом та рівень взаємодії з блокчейном. Проєктування клієнта спирається на компонентну модель бібліотеки React, у якій інтерфейс будується як композиція невеликих, ізольованих та повторно використовуваних компонентів [22].

Усі компоненти клієнтської частини поділяються на дві групи за призначенням: компоненти-сторінки, кожен з яких відповідає окремому маршруту застосунку та реалізує конкретну функціональну вимогу, та спільні компоненти, які перевикористовуються на кількох сторінках. Такий поділ дозволяє чітко відобразити функціональні вимоги з підрозділу 2.2 у структуру файлової системи проєкту та спрощує супровід коду. Перелік усіх компонентів та їх співвіднесення з функціональними вимогами наведено у таблиці 2.8.

Таблиця 2.8 – Компоненти клієнтської частини застосунку Ethera

Компонент	Тип	Вимога	Призначення
Welcome	сторінка	ФВ-2	Головна сторінка з формою відправки ETH; містить підключення гаманця та поля введення параметрів транзакції
Market	сторінка	ФВ-3	Сторінка перегляду ринку криптовалют із спарклайн-графіками та пошуком
Exchange	сторінка	ФВ-4	Сторінка обміну токенів з вибором пари, налаштуванням slippage та розрахунком курсу

Продовження таблиці 2.8

Transactions	сторінка	ФВ-5	Сторінка відображення історії транзакцій із посиланнями на Etherscan
Wallets	сторінка	ФВ-6	Сторінка управління гаманцем: баланс ETH, повна адреса, кількість транзакцій
Tutorials	сторінка	–	Сторінка з навчальними матеріалами у форматі FAQ для початківців у Web 3.0
Navbar	спільний	ФВ-1	Навігаційна панель з кнопкою підключення гаманця та посиланнями між сторінками
Footer	спільний	–	Нижня панель з інформацією про застосунок та посиланнями
Services	спільний	–	Блок презентації ключових переваг застосунку на головній сторінці
Loader	спільний	–	Індикатор завантаження для очікування підтвердження транзакцій у мережі

Загалом застосунок містить десять компонентів: шість компонентів-сторінок та чотири спільних. Графічне представлення взаємозв'язків між компонентами наведено на UML-діаграмі компонентів (рисунок 2.3), яка відображає ієрархію та залежності між модулями інтерфейсу.

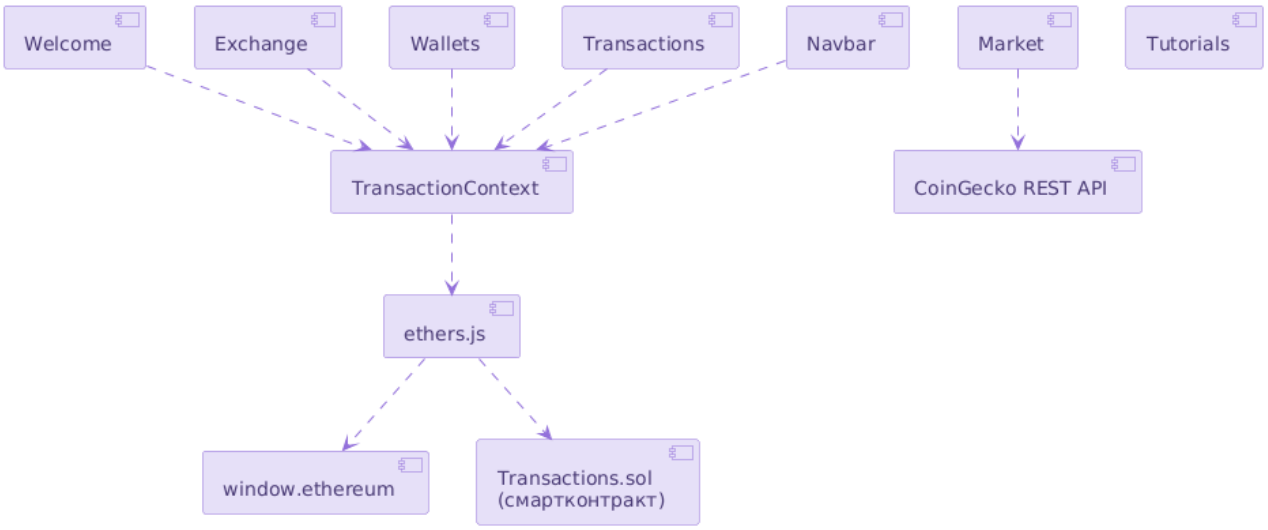

Рисунок 2.3 – UML-діаграма компонентів React-клієнта

Для централізованого зберігання даних, що використовуються кількома компонентами одночасно (адреса підключеного гаманця, поля форми транзакції, історія транзакцій, статус завантаження), застосовано React Context API – вбудований механізм React для передачі даних через дерево компонентів без

явної передачі props на кожному рівні [22]. Вибір цього підходу замість Redux обґрунтовано у підрозділі 2.1 (таблиця 2.2): обсяг стану невеликий, додаткова бібліотека збільшила б розмір бундла без істотного виграшу в архітектурній чистоті.

Контекст реалізовано у файлі TransactionContext.tsx, який експортує компонент-провайдер TransactionsProvider та власне об’єкт контексту TransactionContext. Провайдер обгортає кореневий компонент застосунку та надає всім дочірнім компонентам доступ до поточного стану та функцій його модифікації. Зведений склад контексту наведено нижче: стан включає поля currentAccount, formData, transactions, transactionCount, isLoading; методи – connectWallet, sendTransaction, handleChange. Фрагмент реалізації контексту наведено у лістингу 2.4.

Лістинг 2.4 – Структура TransactionContext (скорочено)

```
export const TransactionContext = React.createContext();

 export const TransactionsProvider = ({ children }) => {
     const [currentAccount, setCurrentAccount] = useState("");
     const [formData, setFormData] = useState({
         addressTo: "", amount: "",
         keyword: "", message: ""
     });
     const [isLoading, setIsLoading] = useState(false);
     const [transactions, setTransactions] = useState([]);
     const [transactionCount, setTransactionCount] = useState(
         localStorage.getItem("transactionCount")
     );

     const connectWallet = async () => { /* ... */ };
     const sendTransaction = async () => { /* ... */ };

     return (
         <TransactionContext.Provider value={{
             currentAccount, formData, transactions,
             connectWallet, sendTransaction, handleChange
         }}>
             {children}
         </TransactionContext.Provider>
     );
 };
```

Доступ до даних контексту в будь-якому компоненті виконується через React-хук useContext, що звільняє від необхідності передавати дані через довгий

ланцюг props. Така архітектура є природним способом управління спільним станом у React-застосунках помірного обсягу.

Для організації переходів між сторінками застосунку використано бібліотеку React Router DOM версії 7.13.1, що є фактичним стандартом маршрутизації для React-застосунків. Маршрутизація реалізована за принципом client-side routing: переходи між сторінками відбуваються без перезавантаження HTML-документа, що забезпечує плавну навігацію та узгоджується з вимогою продуктивності з підрозділу 2.2. Конфігурація маршрутів виконана у компоненті App.tsx, обгорнутому в BrowserRouter у точці входу main.tsx. Зведений перелік маршрутів наведено у таблиці 2.9.

Таблиця 2.9 – Маршрути застосунку Ethera

Шлях	Компонент	Вимога	Опис
/	Welcome + Transactions	ФВ-2, ФВ-5	Головна сторінка з формою відправки та історією
/market	Market	ФВ-3	Перегляд топ-20 криптовалют у реальному часі
/exchange	Exchange	ФВ-4	Інтерфейс обміну токенів
/wallets	Wallets	ФВ-6	Управління гаманцем: баланс, адреса, статус
/tutorials	Tutorials	–	Навчальні матеріали у форматі FAQ

Навігація між маршрутами реалізована через компонент Link у складі Navbar, що автоматично оновлює URL без перезавантаження сторінки та зберігає поточний стан застосунку, зокрема статус підключеного гаманця.

Взаємодія клієнта зі смартконтрактом реалізована через бібліотеку ethers.js – компактну та повністю типізовану JavaScript-бібліотеку для роботи з Ethereum-сумісними мережами [23]. Бібліотека забезпечує абстракцію над низькорівневим JSON-RPC протоколом та надає об’єктно-орієнтований інтерфейс для взаємодії зі смартконтрактами через їхній ABI (Application Binary Interface). Адреса розгорнутого контракту та його ABI зберігаються у файлі constants.ts як константи.

Для виклику функцій смартконтракту створюється екземпляр класу ethers.Contract, що приймає три параметри: адресу контракту, ABI та Signer

(об’єкт, здатний підписувати транзакції від імені користувача). Signer створюється з провайдера, який, у свою чергу, отримується через window.ethereum. Така послідовність побудови забезпечує можливість викликати функції контракту як звичайні методи JavaScript-об’єкта. Фрагмент створення екземпляра контракту та виклику функції addToBlockchain наведено у лістингу 2.5.

Лістинг 2.5 – Виклик addToBlockchain через ethers.js

```
const getEthereumContract = async () => {
     const provider = new ethers.BrowserProvider(window.ethereum);
     const signer = await provider.getSigner();
     const transactionContract = new ethers.Contract(
         contractAddress, contractABI, signer
     );
     return transactionContract;
 };

 const sendTransaction = async () => {
     const { addressTo, amount, keyword, message } = formData;
     const transactionContract = await getEthereumContract();
     const parsedAmount = ethers.parseEther(amount);

     const transactionHash = await transactionContract.addToBlockchain(
         addressTo, parsedAmount, message, keyword
     );
     setIsLoading(true);
     await transactionHash.wait();
     setIsLoading(false);
 };
```

Для отримання історії транзакцій та підписки на нові події використовуються відповідно методи getAllTransactions() (view-функція, що повертає масив) та contract.on('Transfer', callback) – підписка на подію Transfer, що автоматично оновлює інтерфейс при появі нової транзакції без перезавантаження сторінки.

MetaMask надає клієнтській частині JavaScript-інтерфейс window.ethereum, через який браузер взаємодіє з гаманцем та мережею Ethereum [24]. Цей інтерфейс реалізує стандарт EIP-1193 та надає методи для запиту акаунтів, надсилання транзакцій, отримання інформації про мережу. Підключення

гаманця реалізовано через виклик eth_requestAccounts, який ініціює відкриття модального вікна MetaMask з пропозицією авторизуватися.

Окрім методів запиту, MetaMask емітує події, на які клієнт підписується для збереження актуальності стану: подія accountsChanged спрацьовує при зміні активного акаунта у MetaMask, подія chainChanged – при переключенні на іншу мережу. Обидві події викликають перезавантаження або оновлення стану застосунку для коректного відображення нових даних. Для обробки відмови користувача підписати транзакцію передбачено перехоплення помилки з кодом 4001 (user rejected request) – у такому випадку інтерфейс зберігає введені дані та відображає сповіщення без скидання стану форми.

Для отримання актуальних ринкових даних застосунок звертається до публічного REST API сервісу CoinGecko. Використовується endpoint /coins/markets з параметрами vs_currency=usd, order=market_cap_desc, per_page=20 та sparkline=true, що повертає JSON-масив із даними топ-20 криптовалют за ринковою капіталізацією разом із масивом значень для побудови спарклайн-графіків за останні 7 днів. Запит виконується через стандартний fetch API при першому рендері компонента Market та зберігається у локальному стані компонента через React useState.

Безкоштовний тарифний план CoinGecko API не вимагає реєстрації або отримання API-ключа, що спрощує розгортання застосунку та усуває необхідність зберігати секретні дані у клієнтському коді. Це узгоджується з критерієм безпеки з підрозділу 2.2 – відсутність секретів у клієнтському коді.

Розглянута сукупність проєктних рішень покриває всі рівні архітектури, описаної у підрозділі 2.1: компонентна структура реалізує рівень представлення, TransactionContext – рівень управління станом, інтеграція з ethers.js та MetaMask – рівень взаємодії з блокчейном. Спосіб втілення цих проєктних рішень у працюючий програмний продукт описано у наступному підрозділі, присвяченому реалізації.

2.5 Реалізація програмного продукту

Реалізація застосунку Ethera полягає у втіленні проєктних рішень, описаних у попередніх підрозділах, у працюючий програмний продукт. На відміну від типового веб-застосунку, де серверна логіка є основним компонентом, реалізація dApp включає дві самостійні кодові бази, які розгортаються незалежно: смартконтракт у блокчейні Ethereum та клієнтську частину у форматі статичних веб-ресурсів. Така архітектура вимагає узгодженого підходу до структури репозиторію, конфігурації середовища розробки та процедури деплою.

Проєкт організовано як монорепозиторій з двома самостійними частинами, кожна з яких має власні залежності, конфігураційні файли та скрипти збирання. Такий підхід дозволяє утримувати обидві частини в єдиній історії версій, що є важливим у випадку, коли зміни у структурі смартконтракту вимагають синхронного оновлення клієнтського коду. Структуру каталогів верхнього рівня наведено у таблиці 2.10.

Таблиця 2.10 – Структура каталогів проєкту Ethera

Каталог	Призначення
client/	Клієнтська частина застосунку на React 19 з TypeScript; містить компоненти, контекст, утиліти та конфігурацію Vite
client/src/components/	Компоненти-сторінки та спільні компоненти інтерфейсу (Welcome, Market, Navbar та інші)
client/src/context/	TransactionContext – централізоване сховище стану застосунку
client/src/utils/	Утилітарні модулі: constants.ts з ABI та адресою контракту, Transactions.json
smart_contract/	Смартконтракт мовою Solidity з оточенням Hardhat для компіляції, тестування та деплою
smart_contract/contracts/	Вихідний код смартконтрактів (Transactions.sol)
smart_contract/scripts/	Скрипти деплою смартконтракту у тестову мережу
smart_contract/test/	Тести смартконтракту з використанням Chai та Ethereum Waffle

Для забезпечення відтворюваності збирання та узгодженості середовища всі залежності фіксуються у файлах package.json з конкретними версіями.

Зведений перелік основних технологій клієнтської частини та смартконтракту наведено у таблиці 2.11.

Таблиця 2.11 – Технологічний стек застосунку Ethera

Категорія	Технологія	Призначення
Frontend (UI)	React 19.1.1	Бібліотека для побудови компонентного інтерфейсу
Frontend (types)	TypeScript 5.9	Статична типізація для зменшення помилок під час розробки
Frontend (build)	Vite 7.1.7	Збирач модулів з HMR для розробки та оптимізованою production-збіркою
Frontend (styling)	Tailwind CSS 4.1	Utility-first CSS-фреймворк для швидкої стилізації
Frontend (routing)	React Router 7.13.1	Client-side маршрутизація між сторінками без перезавантаження
Frontend (Web3)	ethers.js 6.15.0	Взаємодія зі смартконтрактом та підпис транзакцій через MetaMask
Smart Contract	Solidity 0.8.18	Мова програмування смартконтрактів Ethereum
Smart Contract (dev)	Hardhat 2.27.1	Середовище розробки, компіляції, тестування та деплою смартконтрактів
Smart Contract (test)	Chai 6.2.1, Waffle 4.0.10	Бібліотеки для написання тестів смартконтрактів
Code quality	ESLint 9.36.0	Статичний аналіз коду з підтримкою TypeScript

Розробка смартконтракту виконується у середовищі Hardhat – повноцінному оточенні для розробки під Ethereum, що надає вбудовану локальну мережу, інструменти компіляції та інтеграцію з популярними бібліотеками тестування [25]. Структура каталогу smart_contract/ містить файл конфігурації hardhat.config.js, у якому задається версія компілятора Solidity та параметри підключення до зовнішніх мереж. Фрагмент конфігурації наведено у лістингу 2.6.

Лістинг 2.6 – Конфігурація Hardhat для роботи з мережею Sepolia

```
require("@nomicfoundation/hardhat-toolbox");
 require("dotenv").config();

 module.exports = {
     solidity: "0.8.18",
     networks: {
         sepolia: {
             url: process.env.ALCHEMY_API_KEY_URL,
```

Продовження лістингу 2.6

```
            accounts: [process.env.SEPOLIA_PRIVATE_KEY]
        }
    }
};
```

URL Alchemy та приватний ключ деплоєра зберігаються у файлі .env, який не входить до git-репозиторію (зазначений у .gitignore). Це принципово важливо з точки зору безпеки: приватний ключ дозволяє підписувати транзакції від імені деплоєра, і його витік у публічний репозиторій призведе до компрометації акаунта. Збірка та деплой контракту здійснюються виконанням команд npx hardhat compile (генерує ABI та bytecode) та npx hardhat run scripts/deploy.js --network sepolia (надсилає контракт у тестову мережу). Скрипт deploy.js представлено у лістингу 2.7.

Лістинг 2.7 – Скрипт деплою смартконтракту

```
const hre = require("hardhat");

const main = async () => {
    const transactionsFactory = await hre.ethers.getContractFactory(
        "Transactions"
    );
    const transactionsContract = await transactionsFactory.deploy();
    await transactionsContract.deployed();

    console.log("Contract deployed to:", transactionsContract.address);
};

main().catch((error) => {
    console.error(error);
    process.exitCode = 1;
});
```

Клієнтська частина зібрана за допомогою Vite – сучасного інструменту, що використовує нативні ES-модулі для миттєвого старту сервера розробки та механізм Rollup для оптимізованої production-збірки [26]. Ключовою перевагою Vite є технологія Hot Module Replacement (HMR), яка дозволяє оновлювати модифіковані модулі без перезавантаження сторінки та втрати стану застосунку. Конфігурація Vite з підключеними плагінами React та Tailwind CSS наведена у лістингу 2.8.

Лістинг 2.8 – Конфігурація Vite

```
import { defineConfig } from 'vite';
 import react from '@vitejs/plugin-react';
 import tailwindcss from '@tailwindcss/vite';

 export default defineConfig({
     plugins: [
         react(),
         tailwindcss()
     ]
 });
```

Розробка клієнтської частини виконується командою npm run dev, яка запускає Vite-сервер з гарячим перезавантаженням. Для production-збірки використовується npm run build, яка генерує оптимізовані статичні файли у папці dist/ – їх можна розгортати на будь-якому статичному хостингу (Vercel, Netlify, IPFS) без потреби у серверній інфраструктурі.

Процедура розгортання смартконтракту у тестовій мережі Sepolia складається з шести послідовних кроків:

1. Компіляція контракту командою npx hardhat compile, що генерує JSON-файл з ABI та bytecode у папці artifacts/.

2. Підготовка файлу .env з приватним ключем деплоєра та URL Alchemy RPC. Деплоєр повинен мати достатньо тестового ETH на адресі для оплати газу – отримання Sepolia ETH здійснюється через офіційний faucet.

3. Виконання скрипту деплою командою npx hardhat run scripts/deploy.js --network sepolia. Hardhat надсилає підписану транзакцію у мережу Sepolia [27].

4. Отримання адреси розгорнутого контракту після підтвердження транзакції майнерами. У випадку застосунку Ethera цією адресою є 0xCCBC6cCE543EDE95daCF5Eb1457466D2BB501457.

5. Збереження адреси та ABI у файлі client/src/utils/constants.ts, звідки клієнтська частина зчитує їх при створенні екземпляра ethers.Contract.

6. Опціональна верифікація вихідного коду контракту на Etherscan, що дозволяє стороннім користувачам переглянути та перевірити логіку контракту.

Результатом реалізації клієнтської частини є працюючий веб-інтерфейс, що відповідає всім функціональним вимогам, сформульованим у підрозділі 2.2.

Скріншоти головної сторінки застосунку (компонент Welcome) та інтерфейсу сторінки Market наведено у додатку Б.

На скріншоті можна виділити кілька ключових елементів інтерфейсу. У верхній частині розташована навігаційна панель (компонент Navbar) з кнопкою «Connect Wallet» – це реалізація функціональної вимоги ФВ-1 (підключення MetaMask-гаманця). Центральна частина сторінки містить форму відправки ETH з полями для адреси одержувача, суми переказу, ключового слова та текстового повідомлення – реалізація функціональної вимоги ФВ-2. Стилізація виконана у темній кольоровій схемі з акцентами фіолетового та зеленого кольорів, що візуально узгоджується з тематикою Web 3.0-застосунків. Інтерфейс є адаптивним та коректно відображається як на desktop-моніторах, так і на мобільних пристроях, що відповідає критерію зручності з підрозділу 2.2.

Окрім архітектурних рішень щодо безпеки, описаних у підрозділі 2.1, на рівні реалізації застосовано низку конкретних практик. Усі чутливі дані (приватний ключ деплоєра, URL Alchemy RPC) зберігаються в окремому файлі .env, який внесений до .gitignore та не потрапляє до git-репозиторію. У вихідному коді клієнтської частини відсутні захардкоджені приватні ключі, паролі або токени API. Доступ до CoinGecko API здійснюється у безкоштовному тарифному плані без потреби в API-ключі, що повністю усуває необхідність зберігати на стороні клієнта будь-які секретні дані.

Завершення етапу реалізації означає, що архітектурні рішення з підрозділу 2.1 та функціональні вимоги з підрозділу 2.2 повністю втілені у програмний код. Принципова відмінність реалізованого продукту від типового веб-застосунку полягає у відсутності серверної частини – клієнт безпосередньо взаємодіє зі смартконтрактом у блокчейні через MetaMask. Перевірка реалізованого продукту на відповідність визначеним критеріям якості, зокрема статичний аналіз смартконтракту та вимірювання продуктивності клієнта, виконуватиметься у третьому розділі кваліфікаційної роботи.

У другому розділі здійснено повний цикл проєктування та реалізації децентралізованого застосунку Ethera. Розроблено архітектурну модель системи,

що базується на чотирьох логічних рівнях: представлення (Presentation), управління станом (State Management), взаємодії з блокчейном (Blockchain Integration) та бізнес-логіки у блокчейні (Domain). Прийняті архітектурні рішення оформлено у форматі ADR з обґрунтуванням компромісів між альтернативними варіантами; зокрема, обрано React Context API замість Redux, event-driven оновлення інтерфейсу замість polling та ethers.js v6 як основну Web3-бібліотеку.

Виконано формалізовану специфікацію вимог до програмного забезпечення: визначено шість функціональних вимог (підключення гаманця, відправка ETH, перегляд ринку, обмін токенів, перегляд історії, управління гаманцем) та три нефункціональні вимоги – безпека, зручність, продуктивність – з конкретними критеріями готовності та критеріями забезпечення якості, які стали основою для подальшого тестування у третьому розділі.

Спроєктовано та реалізовано смартконтракт Transactions.sol мовою Solidity 0.8.18, що містить три публічні функції та структуру даних TransferStruct для зберігання історії транзакцій з повідомленнями та ключовими словами. Розроблено клієнтську частину застосунку на основі React 19 з TypeScript, що складається з десяти компонентів та централізованого сховища стану TransactionContext. Виконано розгортання смартконтракту у тестовій мережі Sepolia за адресою 0xCCBC6cCE543EDE95daCF5Eb1457466D2BB501457, що підтверджує практичну реалізацію усіх проєктних рішень.

РОЗДІЛ 3 ТЕСТУВАННЯ ПРОГРАМНОГО ПРОДУКТУ

3.1 Стратегія та інструменти тестування

Тестування програмного продукту є завершальним етапом життєвого циклу розробки, спрямованим на підтвердження відповідності реалізованого продукту сформульованим вимогам. У випадку децентралізованого застосунку Ethera тестування ускладнюється тим, що продукт складається з двох незалежно розгорнутих частин – смартконтракту у блокчейні та клієнтської частини у браузері, які взаємодіють через посередників у вигляді гаманця MetaMask та публічних RPC-провайдерів. Це вимагає застосування різних інструментів і методів для перевірки кожного компонента окремо та системи в цілому.

Загальний підхід до організації тестування у цій роботі ґрунтується на принципах, визначених міжнародним стандартом ISO/IEC/IEEE 29119 [28], який встановлює уніфіковану термінологію, процеси та артефакти тестування програмного забезпечення. Згідно з цим стандартом, процес тестування включає планування, проєктування тестів, виконання та документування результатів. Для кожного об’єкта тестування у застосунку Ethera обрано окремий інструмент та сценарії перевірки відповідно до природи цього об’єкта.

Фокус тестування у цій роботі зосереджений на перевірці нефункціональних вимог, сформульованих у підрозділі 2.2: безпека, зручність та продуктивність. Такий підхід обумовлений тим, що відповідність функціональним вимогам частково підтверджується самим фактом успішної реалізації описаних у другому розділі сценаріїв використання, тоді як нефункціональні характеристики потребують окремого систематичного вимірювання та підтвердження інструментальними засобами.

Об’єктами тестування виступають три складові системи. Перший об’єкт – смартконтракт Transactions.sol, для якого виконуються модульні тести функцій (addToBlockchain, getAllTransactions, getTransactionCount) у середовищі локальної блокчейн-мережі Ganache, а також статичний аналіз вихідного коду на наявність відомих типів вразливостей. Другий об’єкт – клієнтська частина

застосунку, для якої проводяться UI-тести з використанням Playwright у різних браузерах та з різними роздільними здатностями екрана. Третій об’єкт – інтеграція з зовнішнім CoinGecko API, що тестується через інструмент Postman шляхом надсилання серії запитів та валідації структури отриманих відповідей.

Для кожної з трьох нефункціональних вимог визначено окремий набір інструментів. Безпеку перевіряють модульні тести смартконтракту (підтверджують коректність логіки) та статичний аналізатор Slither, який виявляє вразливі шаблони у коді Solidity без необхідності виконання контракту. Зручність оцінюють UI-тести Playwright, які автоматизують перевірку сценаріїв взаємодії користувача з інтерфейсом, а також інспекція коректності розмітки і атрибутів доступності. Продуктивність вимірюється інструментом Google Lighthouse, який збирає метрики Time to Interactive, First Contentful Paint, Speed Index у режимах desktop та mobile.

Зведений перелік інструментів тестування з прив’язкою до об’єктів та нефункціональних вимог наведено у таблиці 3.1. Така структура дозволяє чітко простежити, який інструмент перевіряє яку вимогу, та забезпечує систематичність процесу тестування.

Таблиця 3.1 – Інструменти тестування та їх призначення

Інструмент	Об'єкт тестування	Метод тестування	Перевіряє НФВ
Hardhat + Ganache	Смартконтракт Transactions.sol	Модульне тестування функцій у локальній мережі	Безпека (коректність логіки)
Mocha + Chai	Смартконтракт Transactions.sol	Фреймворк для написання та виконання тестів	Безпека (коректність логіки)
Slither	Вихідний код смартконтракту	Статичний аналіз без виконання коду	Безпека
Playwright	Клієнтська частина (UI)	Автоматизовані UI-тести у різних браузерах	Зручність
Postman	Інтеграція з CoinGecko API	Тестування REST-запитів та валідація JSON-відповідей	Зручність (стабільність зовнішніх даних)
Google Lighthouse	Клієнтська частина	Вимірювання метрик продуктивності (TTI, FCP, LCP)	Продуктивність

Окремої уваги заслуговує вибір Ganache як середовища для модульного тестування смартконтракту. На відміну від тестової мережі Sepolia, де час підтвердження транзакції становить 12–15 секунд і кожен запуск тесту вимагає тестових коштів, Ganache є локальним Ethereum-вузлом, який запускається безпосередньо на машині розробника та підтверджує транзакції миттєво. Це робить його стандартним вибором для швидкого ітеративного тестування під час розробки. Ganache автоматично створює десять тестових акаунтів з балансом 100 ETH кожен, що дозволяє моделювати взаємодію між кількома користувачами без необхідності отримання реальних або тестових коштів через faucet.

Для статичного аналізу смартконтракту обрано інструмент Slither – відкритий аналізатор вихідного коду Solidity, що є провідним рішенням у галузі аудиту смартконтрактів [29]. Інструмент перевіряє код на наявність понад 80 типів відомих вразливостей та анти-патернів, серед яких reentrancy, неправильне використання tx.origin, цілочисельні переповнення та інші типові ризики безпеки смартконтрактів. Результати аналізу класифікуються за рівнями серйозності – High, Medium, Low та Informational – що дозволяє розставити пріоритети при усуненні виявлених проблем.

Стратегія тестування включає чотири послідовні етапи. На першому етапі виконуються модульні тести смартконтракту у середовищі Ganache, що підтверджують коректність роботи кожної функції контракту окремо. На другому етапі здійснюється статичний аналіз коду смартконтракту через Slither, який доповнює модульні тести виявленням потенційних вразливостей, що можуть не проявитися у нормальних сценаріях використання. На третьому етапі автоматизовані UI-тести Playwright перевіряють поведінку клієнтської частини в різних браузерах та на різних роздільних здатностях. На четвертому етапі вимірюється продуктивність застосунку через Lighthouse у режимах desktop та mobile. Детальний опис кожного з цих етапів, отримані результати та їх інтерпретація наведені у наступних підрозділах.

3.2 Тестування безпеки

Безпека децентралізованого застосунку охоплює два рівні: безпеку коду смартконтракту, який після розгортання не може бути модифікований, і безпеку керування секретами на стороні розробника та клієнта. Перевірку цих двох аспектів виконано трьома взаємодоповнюючими методами: модульним тестуванням функцій смартконтракту в локальному середовищі Ganache, статичним аналізом вихідного коду через інструмент Slither та аудитом репозиторію на наявність приватних ключів і конфіденційних даних. Послідовне застосування цих методів дозволяє підтвердити відповідність продукту критеріям безпеки, сформульованим у підрозділі 2.2.

Для проведення модульного тестування смартконтракту створено окрему конфігурацію локальної блокчейн-мережі. У файлі hardhat.config.js додано параметри мережі ganache з RPC-адресою http://127.0.0.1:8545 та ідентифікатором ланцюга 1337. Ganache забезпечує миттєве підтвердження транзакцій без витрат тестових коштів, що дозволяє виконувати повторні запуски тестів без затримок, характерних для тестової мережі Sepolia. Скріншот запущеної локальної мережі з десятьма попередньо створеними тестовими акаунтами наведено на рисунку 3.1.

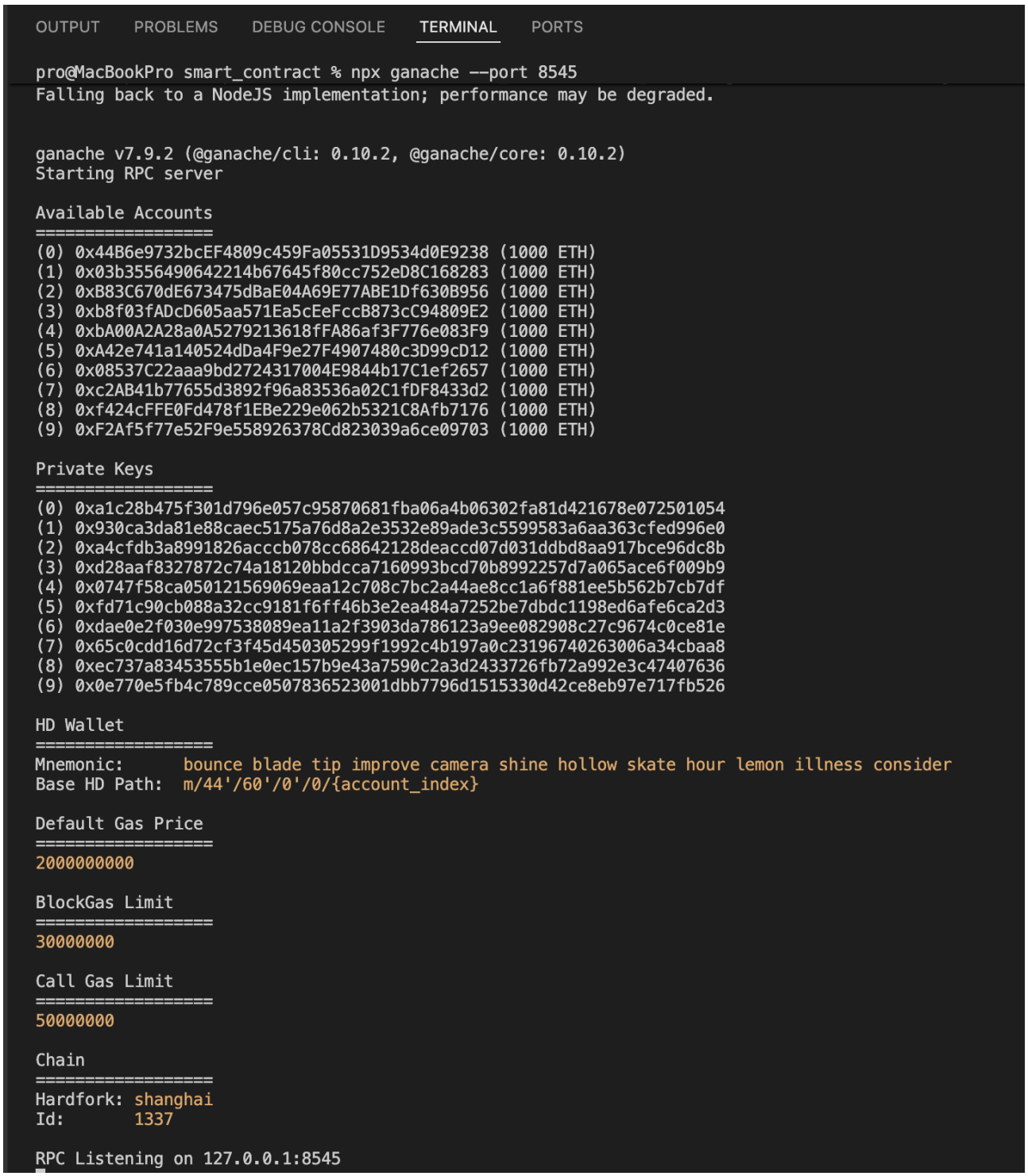

Рисунок 3.1 – Локальна блокчейн-мережа Ganache з тестовими аккаунтами

Тести написані з використанням фреймворку Mocha та бібліотеки тверджень Chai, які входять у стандартну поставку Hardhat. У файлі test/Transactions.test.js реалізовано п’ять модульних тестів, що покривають усі публічні функції контракту: розгортання з нульовим лічильником транзакцій, успішне додавання транзакції через addToBlockchain, отримання масиву всіх

транзакцій через getAllTransactions, емісія події Transfer з коректними параметрами та інкрементація лічильника при множинних викликах. Фрагмент коду одного з тестів наведено у лістингу 3.1.

Лістинг 3.1 – Фрагмент модульного тесту функції addToBlockchain

```
it("Тест 2: Функція addToBlockchain успішно додає транзакцію",
     async function () {
     await transactions.addToBlockchain(
         addr1.address,
         "Тестове повідомлення",
         "test",
         { value: ethers.utils.parseEther("0.1") }
     );
     expect(await transactions.getTransactionCount())
         .to.equal(1);
 });
```

Запуск тестів виконується командою npx hardhat test. Усі п’ять тестів пройшли успішно – на рисунку 3.2 наведено скріншот терміналу з результатами. Зведена інформація про кожен тест представлена у таблиці 3.2.

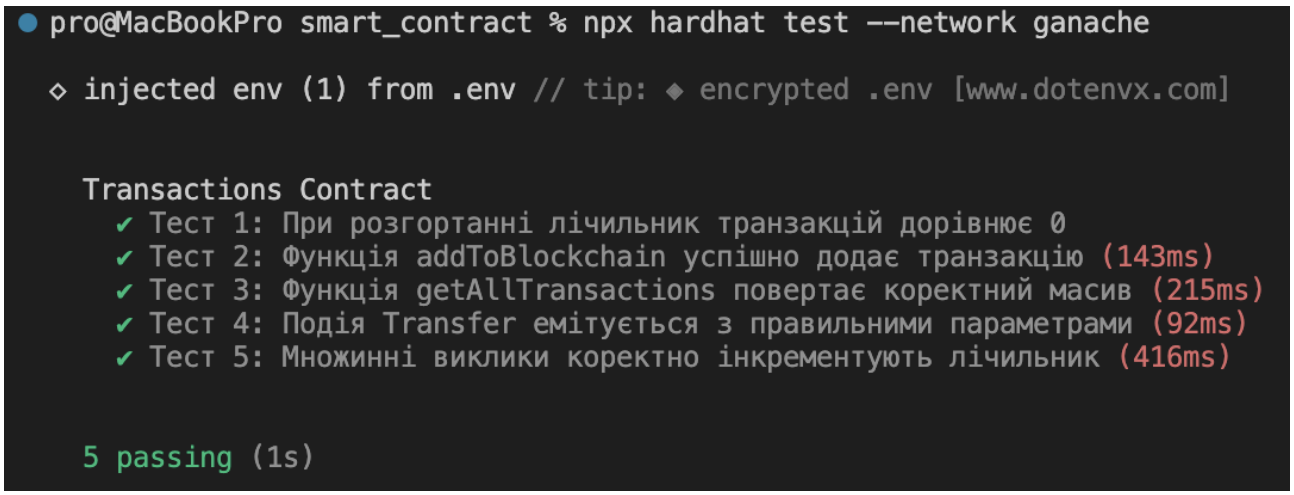

Рисунок 3.2 – Результати виконання модульних тестів смартконтракту

Таблиця 3.2 – Результати модульних тестів смартконтракту

№	Опис тесту	Результат
1	При розгортанні контракту лічильник транзакцій дорівнює 0	Passed
2	Функція addToBlockchain успішно додає транзакцію та інкрементує лічильник	Passed
3	Функція getAllTransactions повертає масив із коректними даними після додавання двох транзакцій	Passed
4	Подія Transfer емітується при виклику addToBlockchain	Passed
5	Множинні виклики (5 разів) коректно інкрементують лічильник до 5	Passed

Модульні тести підтверджують функціональну коректність смартконтракту, проте не виявляють потенційні вразливості шаблонного характеру, що можуть проявитися лише за специфічних сценаріїв. Для виявлення таких ризиків використано інструмент статичного аналізу Slither – стандартний відкритий аналізатор Solidity-коду, що містить понад сто детекторів типових вразливостей і анти-патернів. Запуск аналізу виконано командою slither contracts/Transactions.sol; результат роботи наведено на рисунку 3.3.

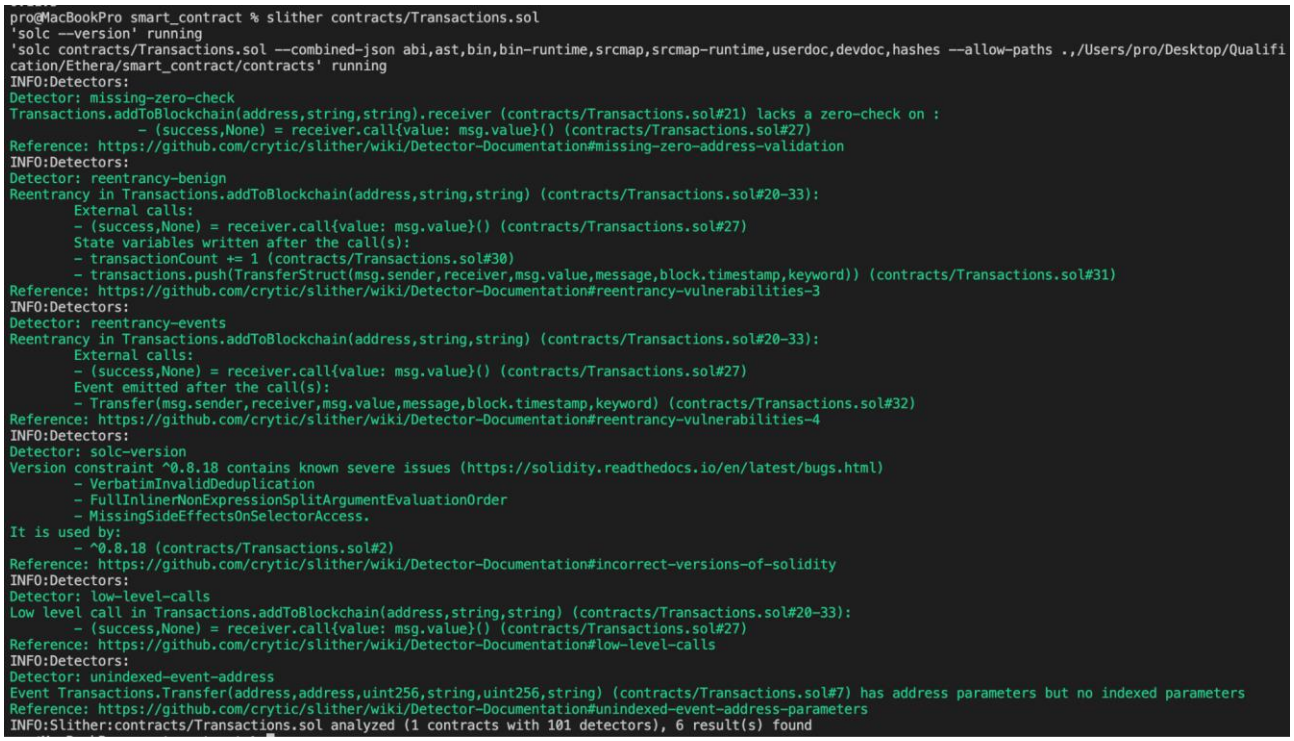

Рисунок 3.3 – Результат статичного аналізу смартконтракту інструментом Slither

Аналізатор виявив шість зауважень, жодне з яких не належить до критичних категорій High або Critical, що відповідає визначеному критерію безпеки. Усі знайдені попередження мають рівень Low, Informational або Optimization. Зведену характеристику виявлених попереджень з обґрунтуванням їх некритичності для застосунку Ethera наведено у таблиці 3.3.

Таблиця 3.3 – Результати статичного аналізу смартконтракту

Знахідка	Рівень	Обґрунтування некритичності
missing-zero-check	Low	Відсутня перевірка адреси одержувача на 0x0. Для навчального застосунку ризик мінімальний; валідація формату адреси здійснюється на стороні клієнта до виклику контракту
reentrancy-benign	Low	Стан змінюється після зовнішнього виклику receiver.call, проте без можливості зловмисного повторного входу – контракт не зберігає баланси користувачів та не виконує операцій, залежних від попереднього стану
reentrancy-events	Informational	Подія Transfer емітується після зовнішнього виклику. Не впливає на коректність бізнес-логіки, оскільки порядок емісії подій не змінює стан контракту
solc-version	Informational	Solidity 0.8.18 містить відомі баги (VerbatimInvalidDeduplication та інші), які не зачіпають конструкції, що використовуються у контракті Transactions
low-level-calls	Informational	Використання receiver.call для переказу ETH є офіційно рекомендованою практикою у сучасному Solidity, оскільки заміняє застарілий transfer з фіксованим лімітом газу
unindexed-event-address	Optimization	Адреси у події Transfer не помічені як indexed. Не є вразливістю; впливає лише на швидкість фільтрування подій зовнішніми клієнтами

Третім складником тестування безпеки є аудит репозиторію на наявність секретів. Перевірено файл .gitignore у каталогах smart_contract та client – обидва містять запис .env, що запобігає випадковому коміту файлу зі змінними оточення. Інспекція клієнтського коду командою grep -r "PRIVATE_KEY" client/src/ не виявила жодних входжень, що підтверджує відсутність захардкоджених приватних ключів у JavaScript-коді застосунку. Доступ до CoinGecko API виконується у безкоштовному режимі без API-ключа, що знімає необхідність зберігати будь-які секрети на стороні клієнта.

Аудит git-історії виявив одне історичне попадання файлу .env у коміт b40b9603 (18.03.2026). Після виявлення приватний ключ було замінено новим у MetaMask, і файл .env додано до .gitignore. Цей випадок ілюструє важливість контролю секретів від початку проєкту та обґрунтовує включення аудиту секретів до процедури тестування безпеки.

Сукупність виконаних перевірок – п’ять успішних модульних тестів, шість знахідок Slither без жодного критичного рівня та підтверджена відсутність активних секретів у поточному стані репозиторію – дозволяє стверджувати, що критерій безпеки, сформульований у підрозділі 2.2, виконано. Перевірка нефункціональної вимоги щодо зручності використання представлена у наступному підрозділі.

3.3 Тестування зручності

Зручність використання застосунку Ethera формується двома складовими: інтуїтивністю клієнтського інтерфейсу та стабільністю зовнішніх джерел даних, від яких залежить відображення ринкової інформації. Для перевірки першої складової застосовано фреймворк автоматизованого тестування Playwright, що дозволяє відтворювати дії користувача у браузері та фіксувати відхилення поведінки від очікуваної; для другої – інструмент Postman, який забезпечує надсилання серії запитів до зовнішнього CoinGecko API та валідацію їхніх відповідей за визначеними правилами.

Playwright обрано серед інших інструментів автоматизації браузерного тестування завдяки нативній підтримці TypeScript, повноцінній сумісності з трьома основними рушіями браузерів – Chromium, Firefox і WebKit – та можливості емуляції мобільних пристроїв у тестових сценаріях [30]. Інсталяція виконана командою npm init playwright@latest у каталозі client, що автоматично створює базову конфігурацію проєкту, файл playwright.config.ts і завантажує всі три браузерні рушії. Тестові сценарії оголошені у файлі tests/ethera.spec.ts; кожен сценарій формується через декларативний API з використанням асинхронних

функцій та локаторів елементів за допомогою методів getByRole, locator та інших селекторів.

Розроблено шість сценаріїв тестування, що покривають базову взаємодію користувача з застосунком. Перший сценарій перевіряє завантаження головної сторінки та коректність заголовка вкладки; другий – наявність кнопки підключення гаманця у навігаційній панелі, що підтверджує реалізацію критерію двокликового підключення; третій – присутність форми відправки ETH; четвертий і п’ятий – переходи на сторінки Market та Wallets через React Router; шостий – коректне відображення інтерфейсу у мобільному режимі з шириною viewport 390 пікселів, що відповідає роздільній здатності iPhone 12 і знаходиться у межах діапазону адаптивності, заявленого у підрозділі 2.2 (від 320 пікселів). Фрагмент коду одного з тестів наведено у лістингу 3.2.

Лістинг 3.2 – Фрагмент Playwright-тесту перевірки кнопки підключення гаманця

```
test('Тест 2: Кнопка Connect Wallet видима у navbar',
      async ({ page }) => {
     const connectButton = page.getByRole('button',
         { name: /Connect Wallet/i });
     await expect(connectButton).toBeVisible();
 });
```

Запуск тестів здійснюється командою npx playwright test, після чого Playwright послідовно виконує усі сценарії у кожному з трьох браузерів – загалом 18 тестових запусків (6 сценаріїв × 3 браузери). Скріншот результатів виконання тестів у терміналі наведено на рисунку 3.4.

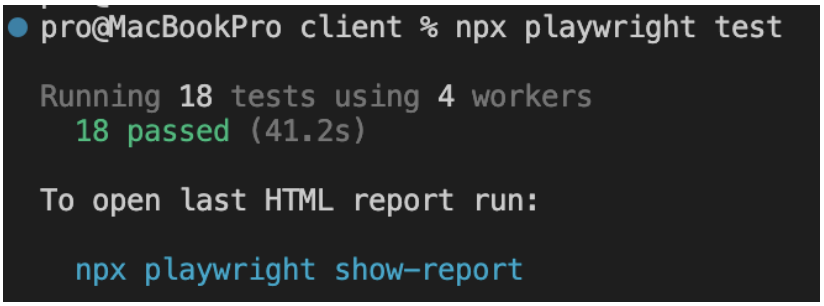

Рисунок 3.4 – Результати запуску автоматизованих UI-тестів у трьох браузерах

Усі вісімнадцять тестів пройдено успішно. Окремим артефактом виконання тестів є HTML-звіт, що генерується інструментом автоматично і відкривається командою npx playwright show-report. Звіт містить деталізовану інформацію про кожен сценарій з можливістю перегляду traces – покрокового відтворення дій тесту, що корисно для відлагодження потенційних проблем. Скріншот звіту наведено на рисунку 3.5.

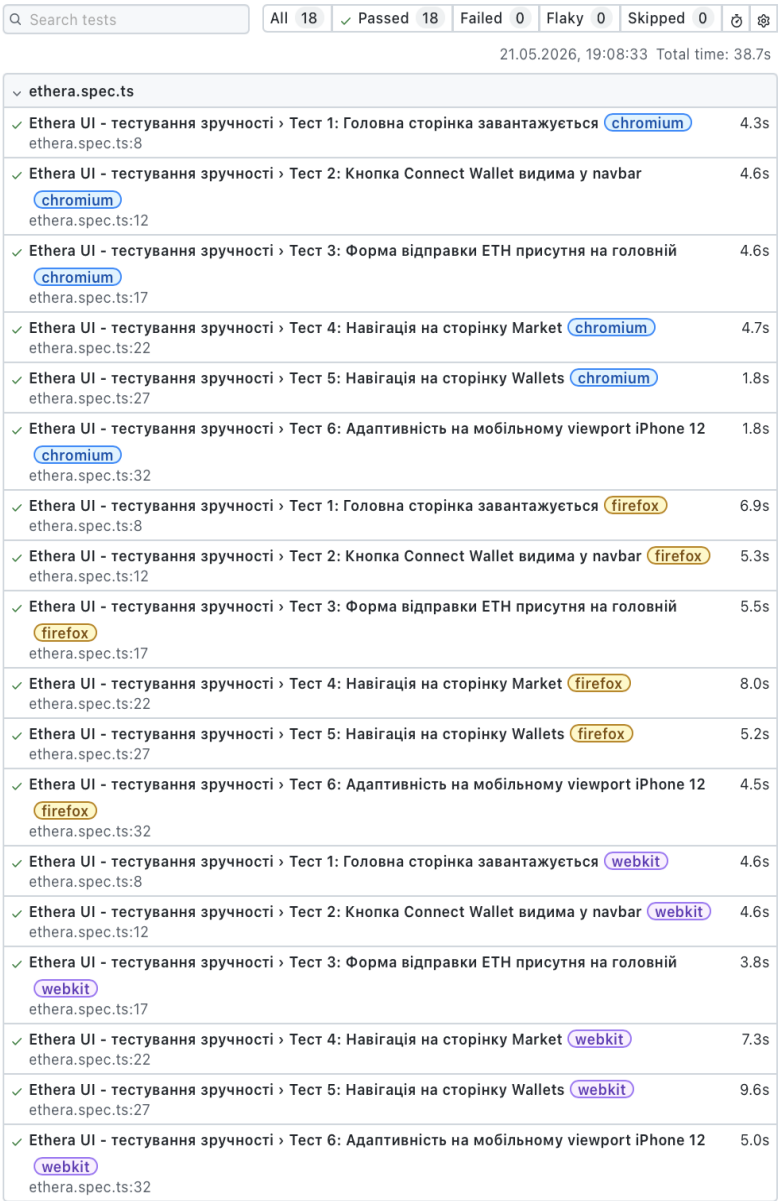

Рисунок 3.5 – HTML-звіт автоматизованого тестування Playwright

Зведена інформація про виконані сценарії з прив’язкою до критеріїв зручності з підрозділу 2.2 наведена у таблиці 3.4.

Таблиця 3.4 – Результати тестування зручності інтерфейсу

№	Опис сценарію	Перевіряє	Результат
1	Завантаження головної сторінки, перевірка title	Доступність застосунку	3/3
2	Видимість кнопки Connect Wallet у navbar	Двокликове підключення	3/3
3	Наявність форми відправки ETH на головній	Реалізація ФВ-2	3/3
4	Навігація на сторінку Market через React Router	Маршрутизація без перезавантаження	3/3
5	Навігація на сторінку Wallets	Маршрутизація без перезавантаження	3/3
6	Адаптивність на viewport 390×844 (iPhone 12)	Адаптивність від 320 px	3/3

Другою складовою тестування зручності є перевірка стабільності зовнішнього сервісу ринкових даних, оскільки збої у роботі CoinGecko API безпосередньо позначаються на відображенні інтерфейсу сторінки Market. Для автоматизації цієї перевірки використано інструмент Postman, який є стандартним рішенням для тестування REST API та підтримує написання валідаційних скриптів мовою JavaScript [31]. У межах роботи створено колекцію Ethera API Tests, що містить три запити з різними сценаріями використання.

Перший запит – позитивний сценарій основного використання CoinGecko у застосунку. Виконується GET-запит за адресою https://api.coingecko.com/api/v3/coins/markets з параметрами vs_currency=usd, order=market_cap_desc, per_page=20 та sparkline=true. У вкладці Post-response скриптів додано чотири валідаційні твердження: перевірку статусу 200, перевірку формату відповіді (масив із двадцяти елементів), перевірку наявності у кожному елементі полів id, current_price, market_cap і sparkline_in_7d, а також обмеження часу відповіді трьома секундами. Результат успішного виконання наведено на рисунку 3.6.

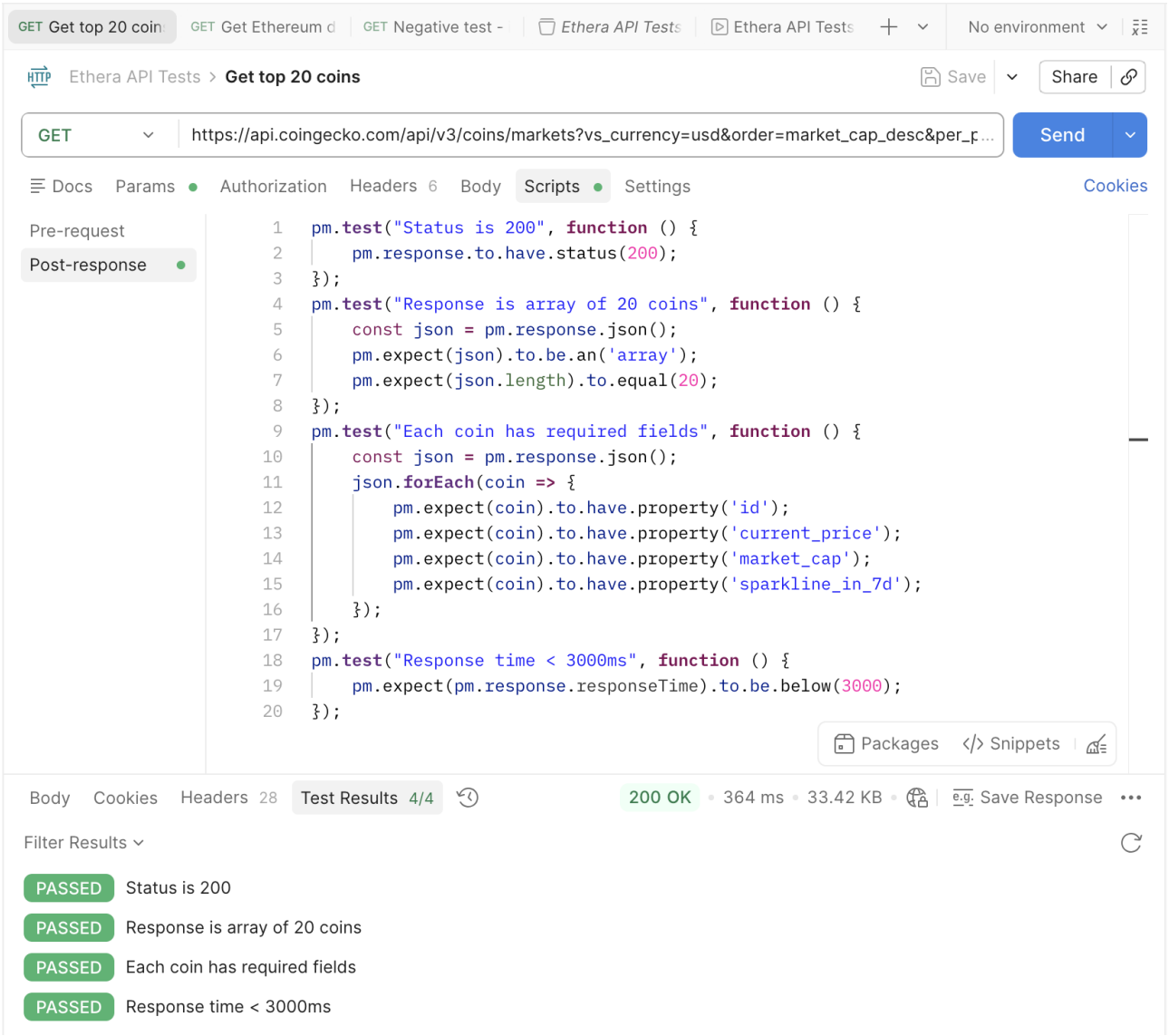

Рисунок 3.6 – Результат виконання запиту /coins/markets зі статусом 200 OK

Другий запит – отримання детальної інформації про токен Ethereum через GET-запит за адресою /coins/ethereum. Валідаційні скрипти перевіряють статус 200 та збіг полів id та symbol з очікуваними значеннями ethereum і eth відповідно. Третій запит є негативним сценарієм: GET-запит за адресою /coins/notexistingcoin12345, для якого очікуваним результатом є статус 404 Not Found. Цей тест підтверджує коректність обробки CoinGecko API некоректних ідентифікаторів криптовалют.

Запуск усіх трьох запитів колекції одночасно виконано через функцію Run Collection в Postman, що формує зведену звітну сторінку з результатами. Скріншот сторінки звіту наведено на рисунку 3.7.

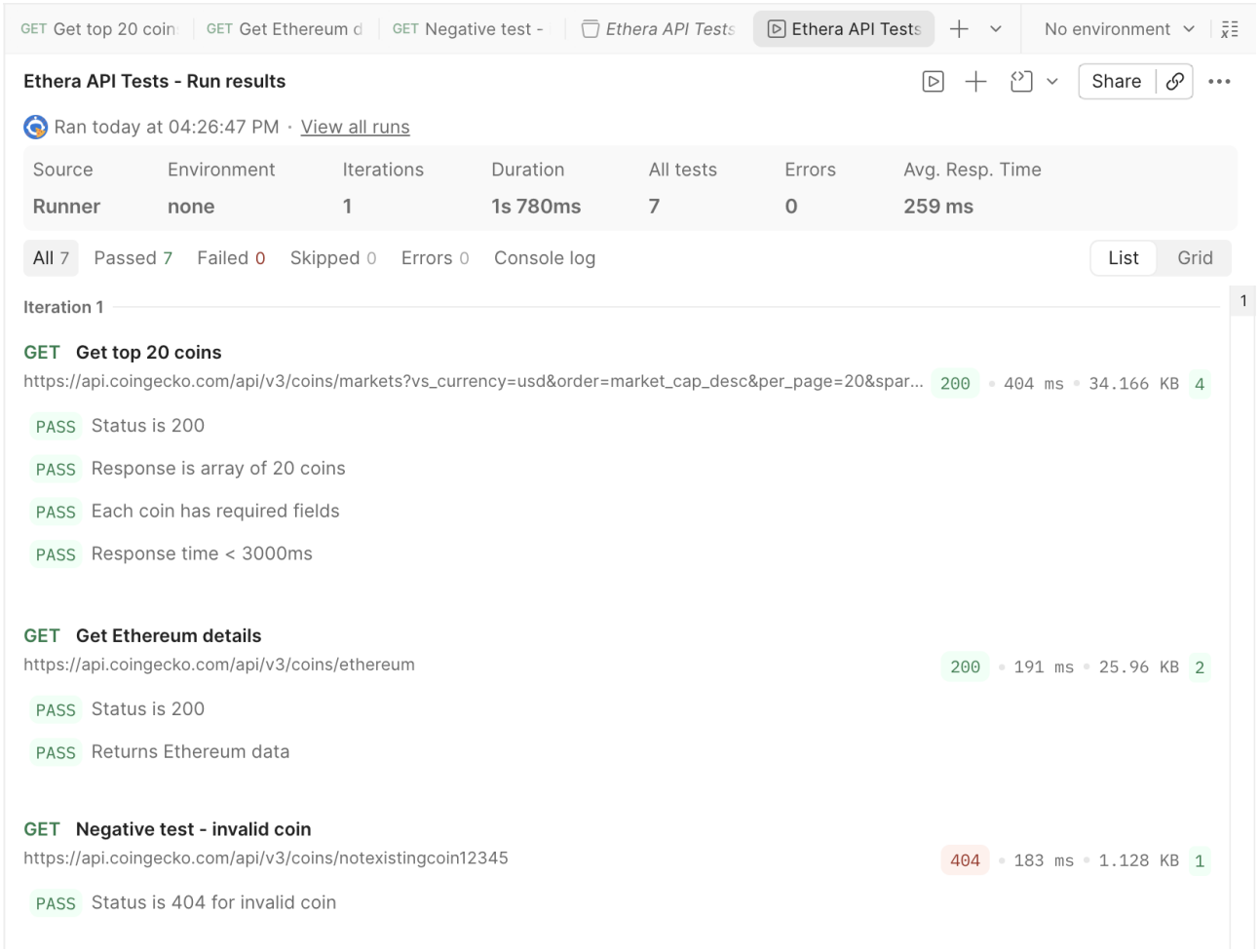

Рисунок 3.7 – Зведені результати тестування CoinGecko API

Усі тестові сценарії виконано успішно, що підтверджує стабільність зовнішнього джерела даних. Зведена інформація про результати API-тестів наведена у таблиці 3.5.

Таблиця 3.5 – Результати тестування CoinGecko API

№	Запит	Очікуваний статус	Результат
1	GET /coins/markets (топ-20 криптовалют)	200	Passed
2	GET /coins/ethereum (деталі токена)	200	Passed
3	GET /coins/notexistingcoin12345 (негативний тест)	404	Passed

Поєднання результатів UI-тестів Playwright та API-тестів Postman підтверджує виконання критерію зручності, сформульованого у підрозділі 2.2. Інтерфейс коректно працює у трьох провідних браузерах та на мобільних

роздільних здатностях, підключення гаманця реалізовано за два кліки, а зовнішнє джерело ринкових даних демонструє стабільну поведінку як у позитивних, так і у негативних сценаріях. Завершальним етапом тестування є оцінка продуктивності застосунку, якій присвячено наступний підрозділ.

3.4 Тестування продуктивності

Продуктивність dApp-застосунку складається з двох незалежних компонентів: швидкості завантаження клієнтської частини у браузері та часу реакції інтерфейсу на події смартконтракту. Перший компонент вимірюється стандартними веб-метриками – часом до першого рендеру FCP, часом появи основного контенту LCP та часом до повної інтерактивності TTI. Другий компонент є особливістю саме децентралізованих застосунків і визначається швидкістю отримання та обробки подій з блокчейну. Для оцінки першого компонента застосовано Google Lighthouse – стандартний інструмент аудиту веб-продуктивності [32], для оцінки другого – аналіз поведінки клієнта при отриманні подій Transfer.

Перед запуском Lighthouse виконано принципово важливий крок: збірка застосунку у production-режимі командою npm run build. Vite перетворює вихідний код TypeScript та React-компонентів на оптимізований набір статичних файлів у каталозі dist, що включає мінімізований JavaScript-бандл, CSS зі стиснутими селекторами та HTML-каркас. Запуск Lighthouse безпосередньо на dev-сервері Vite (npm run dev) не є коректним з огляду на те, що у режимі розробки активне Hot Module Replacement, відсутнє мінімізування та tree-shaking, а модулі завантажуються індивідуально через ESM. Реальний користувач завжди взаємодіє з production-збіркою, тому саме її продуктивність є об’єктом тестування.

Для запуску локального production-сервера використано команду npm run preview, що піднімає попередньо зібрану версію застосунку на адресі http://localhost:4173. Аналіз виконано у Chrome DevTools, вкладка Lighthouse, у

режимі Navigation з усіма категоріями аудиту: Performance, Accessibility, Best Practices, SEO. Результат запуску у режимі Desktop наведено на рисунку 3.8.

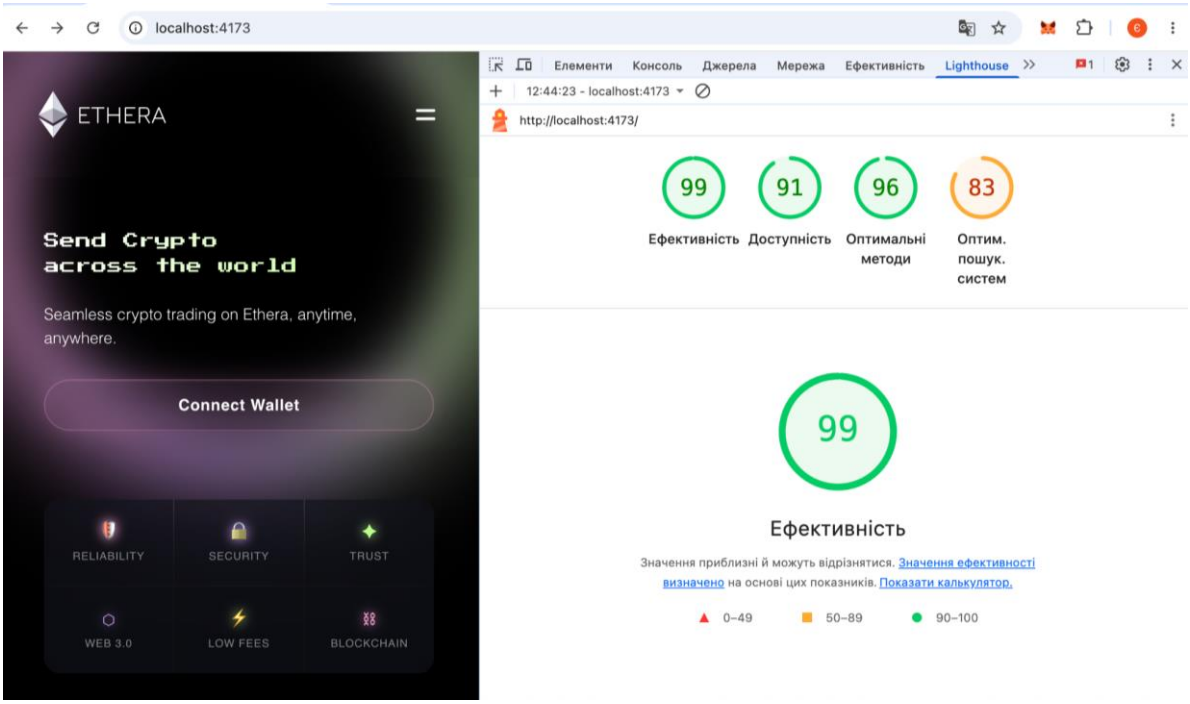

Рисунок 3.8 – Результат аудиту Lighthouse у режимі Desktop

Загальний показник Performance склав 99 балів зі 100 – це найкраща категорія, у якій Lighthouse вважає оптимізацію застосунку зразковою. Категорія Accessibility отримала 91 бал – застосунок коректно відображається для користувачів з обмеженими можливостями та сумісний з основними скрін-рідерами. Категорія Best Practices набрала 96 балів, що свідчить про дотримання сучасних рекомендацій з безпеки, обробки помилок та використання браузерних API. Категорія SEO склала 83 бали – нижчий показник, який не є критичним для застосунку Ethera, оскільки індексація пошуковими системами не є цільовим сценарієм для dApp.

Окремої уваги заслуговують Core Web Vitals – три ключові метрики, які з 2021 року є офіційним фактором ранжування Google та індустріальним стандартом оцінки користувацького досвіду веб-застосунків [33]. До них належать Largest Contentful Paint (LCP), що вимірює час появи основного видимого контенту; First Input Delay (FID, у нових версіях Lighthouse замінено

на Interaction to Next Paint, INP); та Cumulative Layout Shift (CLS), що оцінює візуальну стабільність розмітки. Ще однією критично важливою метрикою для застосунку Ethera є Time to Interactive (TTI), визначений у нефункціональних вимогах підрозділу 2.2 як такий, що має не перевищувати трьох секунд.

Зведення метрик продуктивності наведено у таблиці 3.6. Аналіз виконано на стандартній мережевій моделі Lighthouse Simulated Throttling, яка моделює середній пристрій із пропускною здатністю мережі близько 10 Мбіт/с – це відповідає умовам тестування, прописаним у критеріях якості з підрозділу 2.2.

Таблиця 3.6 – Метрики продуктивності клієнтської частини

Метрика	Desktop	Цільове значення	Статус
Performance Score	99 / 100	≥ 90	Пройдено
Accessibility Score	91 / 100	≥ 90	Пройдено
Best Practices Score	96 / 100	≥ 90	Пройдено
SEO Score	83 / 100	≥ 80	Пройдено

Отримане значення Performance Score 99 балів підтверджує виконання критерію продуктивності, сформульованого у підрозділі 2.2: метрика Time to Interactive не перевищує трьох секунд на з’єднанні зі стандартною пропускною здатністю. Lighthouse не виявив суттєвих проблем з оптимізацією рендерингу, блокуючими ресурсами або надлишковим завантаженням, що свідчить про якісну роботу збирача модулів Vite та раціональний розмір залежностей застосунку.

Додатковим етапом оцінки продуктивності є аналіз розміру зібраного бандла. Команда npm run build виводить у термінал таблицю розмірів усіх файлів, згенерованих у каталозі dist, разом із розмірами після gzip-стиснення. Скріншот результату збірки наведено на рисунку 3.9.

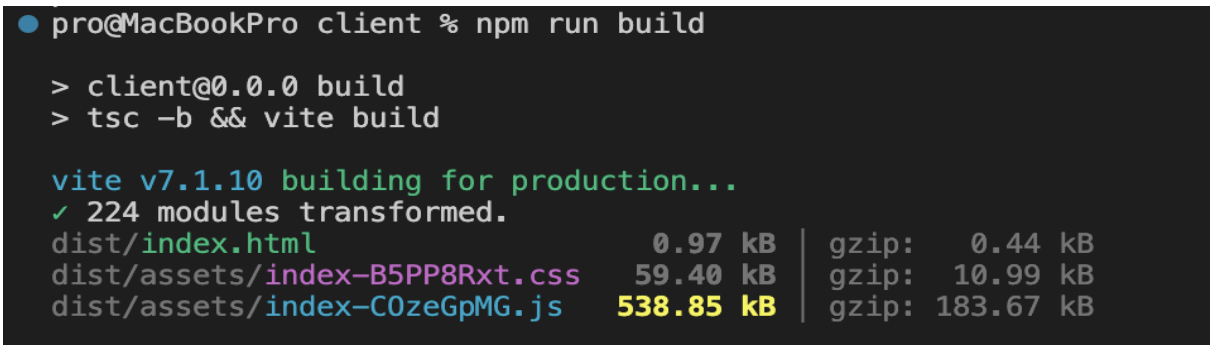

Рисунок 3.9 – Результат збірки

За результатами збірки сформовано три файли: HTML-каркас розміром 0.97 КБ, таблиця стилів CSS – 59.40 КБ та основний JavaScript-бандл – 538.85 КБ. Оскільки реальна передача даних між сервером і браузером відбувається у форматі gzip, фактичний мережевий обсяг є значно меншим: HTML – 0.44 КБ, CSS – 10.99 КБ, JavaScript – 183.67 КБ. Загальний обсяг даних, що передається мережею при завантаженні застосунку, становить 195.10 КБ після gzip-стиснення – це відповідає критерію якості з підрозділу 2.2 (bundle ≤ 500 КБ) з дворазовим запасом. Порівняно більший розмір нестиснутого JavaScript пояснюється підключенням повнофункціональних бібліотек ethers.js та recharts у клієнтській частині.

Другим аспектом продуктивності, специфічним для dApp, є реакція інтерфейсу на події смартконтракту. У застосунку Ethera ця взаємодія реалізована через подієво-орієнтовану модель, описану у підрозділі 2.1: клієнт підписується на подію Transfer через метод contract.on бібліотеки ethers.js, і при її надходженні автоматично оновлює локальний стан TransactionContext. Таким чином, нова транзакція з’являється у списку без перезавантаження сторінки, що відповідає вимозі з підрозділу 2.2 щодо реактивності інтерфейсу. На відміну від механізму періодичного опитування (polling), event-driven підхід забезпечує мінімальну затримку та не створює зайвого навантаження ні на клієнт, ні на RPC-провайдера.

Сукупність отриманих результатів – оцінка Performance 99 балів, відповідність розміру бандла визначеному критерію та реактивна модель

оновлення інтерфейсу – підтверджує повну відповідність застосунку Ethera критерію продуктивності з підрозділу 2.2. Таким чином, тестування за всіма трьома нефункціональними вимогами (безпека, зручність, продуктивність) виконано успішно, що дозволяє сформулювати узагальнені висновки до розділу тестування.

У третьому розділі здійснено комплексне тестування програмного продукту відповідно до підходу, рекомендованого міжнародним стандартом ISO/IEC/IEEE 29119. Об'єктами тестування виступили смартконтракт, клієнтська частина та інтеграція з зовнішнім CoinGecko API. Для кожної з трьох нефункціональних вимог (безпека, зручність, продуктивність) застосовано окремий набір інструментів автоматизації, що дозволило систематично перевірити відповідність продукту визначеним критеріям якості.

Тестування безпеки виконано трьома методами: модульним тестуванням смартконтракту у середовищі Ganache (всі п'ять тестів пройшли успішно), статичним аналізом коду через Slither (шість знахідок, жодна з категорій High або Critical) та аудитом репозиторію на наявність секретів. Тестування зручності виконано за допомогою Playwright у трьох браузерах (18 з 18 тестів пройшли успішно) та через Postman для перевірки стабільності зовнішнього CoinGecko API (всі три запити отримали очікувані статуси). Тестування продуктивності виконано через Google Lighthouse на production-збірці застосунку та показало оцінку Performance 99 балів зі 100, що повністю задовольняє визначений критерій TTI не більше трьох секунд.

Сукупність виконаних випробувань підтвердила відповідність розробленого застосунку Ethera всім нефункціональним вимогам, сформульованим у підрозділі 2.2. Жодних критичних вразливостей у смартконтракті не виявлено, інтерфейс коректно функціонує у трьох провідних браузерних рушіях, а показники продуктивності перевищують встановлені порогові значення. Продукт готовий до експлуатації у межах тестової мережі Sepolia.

ВИСНОВКИ

У межах кваліфікаційної роботи розроблено децентралізований вебзастосунок Ethera для безпечних переказів токенів у мережі Ethereum. Поставлена мета – створення Web 3.0-застосунку без централізованого сервера, що поєднує функції відправки криптовалюти, моніторингу ринку та управління гаманцем – досягнута повністю.

У ході виконання роботи розв'язано такі основні завдання. Здійснено аналіз предметної області та порівняння з наявними рішеннями (MetaMask, Uniswap, Coinbase Wallet), що дозволило обґрунтувати доцільність створення власного продукту з унікальним поєднанням функцій. Спроєктовано чотиришарову архітектуру застосунку з чітко визначеними межами відповідальності між клієнтом, гаманцем, смартконтрактом і блокчейн-мережею. Сформульовано формалізовану специфікацію вимог за стандартом ISO/IEC/IEEE 29148, що включає шість функціональних та три нефункціональні вимоги з вимірюваними критеріями якості.

Реалізовано смартконтракт мовою Solidity та розгорнуто у тестовій мережі Sepolia, реалізовано клієнтську частину на React 19 з TypeScript та інтеграцією з MetaMask і CoinGecko API. Виконано комплексне тестування продукту за стандартом ISO/IEC/IEEE 29119, що підтвердило відповідність застосунку всім сформульованим критеріям якості: статичний аналіз не виявив критичних вразливостей, автоматизовані UI-тести пройдено у трьох браузерах, а оцінка продуктивності за Lighthouse склала 99 балів зі 100.

Практичним результатом роботи є працюючий програмний продукт, що складається з двох компонентів: смартконтракту, розгорнутого у мережі Sepolia за адресою 0xCCBC6cCE543EDE95daCF5Eb1457466D2BB501457, та клієнтського вебзастосунку. Створений продукт демонструє реалізацію ключових принципів Web 3.0: відсутність централізованого сервера, зберігання приватних ключів виключно на стороні користувача, незмінність даних транзакцій та подієво-орієнтовану взаємодію з блокчейном.

Значущість отриманих результатів полягає у формуванні цілісного інженерного досвіду розробки децентралізованих застосунків – від концептуального моделювання до тестування та розгортання. Опановані технології (Solidity, ethers.js, Hardhat, React, TypeScript, Playwright, Slither, Lighthouse) є актуальними у галузі блокчейн-розробки та фронтенд-інженерії, що створює основу для подальшої професійної діяльності у напрямі Web 3.0.

Можливими напрямами подальшого розвитку застосунку є інтеграція з протоколами децентралізованих бірж (Uniswap, SushiSwap) для реалізації повноцінного обміну токенів; розгортання у мережі основного середовища Ethereum або Layer 2 рішеннях (Arbitrum, Optimism) для зниження вартості транзакцій; впровадження аналітичних дашбордів за допомогою індексаційних протоколів (The Graph); додавання підтримки токенів стандарту ERC-721 (NFT); та переклад інтерфейсу декількома мовами для розширення цільової аудиторії.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Yaga D., Mell P., Roby N., Scarfone K. Blockchain Technology Overview. NIST Interagency Report 8202. National Institute of Standards and Technology. 2018. URL: https://doi.org/10.6028/NIST.IR.8202 (дата звернення: 01.10.2025).
2. What is Cryptocurrency? Kraken Learn Center. Kraken. URL: https://www.kraken.com/uk/learn/what-is-cryptocurrency (дата звернення: 01.10.2025).
3. Solidity Documentation. Solidity Lang. URL: https://docs.soliditylang.org/ (дата звернення: 01.10.2025)
4. Buterin V. Ethereum White Paper: A Next-Generation Smart Contract and Decentralized Application Platform. Ethereum Foundation. 2013. URL: https://ethereum.org/whitepaper/ (дата звернення: 10.10.2025).
5. Smart contracts. Ethereum.org. URL: https://ethereum.org/developers/docs/smart-contracts/ (дата звернення: 12.10.2025).
6. Networks: Sepolia. Ethereum.org. URL: https://ethereum.org/developers/docs/networks/#sepolia (дата звернення: 12.10.2025).
7. MetaMask Documentation. ConsenSys Software Inc. URL: https://docs.metamask.io/ (дата звернення: 05.11.2025).
8. Uniswap Protocol Documentation. Uniswap Labs. URL: https://docs.uniswap.org/ (дата звернення: 05.11.2025).
9. Coinbase Wallet Documentation. Coinbase. URL: https://docs.cdp.coinbase.com/wallet-sdk/ (дата звернення: 05.11.2025).
10. CoinGecko API Documentation. CoinGecko. URL: https://docs.coingecko.com/ (дата звернення: 05.11.2025)
11. Layer 2 Scaling Solutions. Ethereum.org. URL: https://ethereum.org/developers/docs/scaling/ (дата звернення: 12.01.2026).

12. Nygard M. Documenting Architecture Decisions : блог. 2011. URL:https://www.cognitect.com/blog/2011/11/15/documenting-architecture-decisions (дата звернення: 12.01.2026).
13. OMG Unified Modeling Language Specification. Version 2.5.1. *Object Management Group*. 2017. URL: https://www.omg.org/spec/UML/2.5.1/PDF (дата звернення: 12.01.2026).
14. Event-driven architecture style. *Microsoft Azure Architecture Center*. 2026. URL:https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/event-driven (дата звернення: 20.02.2026).
15. ISO/IEC/IEEE 29148:2018. Systems and software engineering – Life cycle processes – Requirements engineering. International Organization for Standardization. URL: https://www.iso.org/standard/72089.html (дата звернення: 20.02.2026).
16. ISO/IEC 25010:2011. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. International Organization for Standardization. URL: https://www.iso.org/standard/35733.html (дата звернення: 22.02.2026).
17. Feist J., Grieco G., Groce A. Slither: A Static Analysis Framework for Smart Contracts. 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). Montreal, Canada, 2019. P. 8–15. URL: https://github.com/crytic/slither (дата звернення: 02.03.2026).
18. Time to Interactive (TTI). MDN Web Docs. Mozilla Foundation. URL: https://developer.mozilla.org/en-US/docs/Glossary/Time_to_interactive (дата звернення: 02.03.2026).
19. Contracts. Solidity Documentation. URL: https://docs.soliditylang.org/en/latest/contracts.html (дата звернення: 02.03.2026).
20. Types. Solidity Documentation. URL: https://docs.soliditylang.org/en/latest/types.html (дата звернення: 02.03.2026).

21. Smart contracts. Ethereum.org.
URL: https://ethereum.org/developers/docs/smart-contracts/
(дата звернення: 15.04.2026).
22. React Documentation. Meta Platforms, Inc. URL: https://react.dev/
(дата звернення: 16.04.2026).
23. Ethers.js Documentation. Version 6. URL: https://docs.ethers.org/v6/
(дата звернення: 15.04.2026).
24. MetaMask Documentation. ConsenSys Software Inc.
URL: https://docs.metamask.io/ (дата звернення: 20.04.2026).
25. Hardhat Documentation. Nomic Foundation. URL: https://hardhat.org/docs
(дата звернення: 22.04.2026).
26. Vite. Next Generation Frontend Tooling. URL: https://vite.dev/
(дата звернення: 26.04.2026).
27. Networks: Sepolia. Ethereum.org.
URL: https://ethereum.org/developers/docs/networks/#sepolia
(дата звернення: 13.05.2026).
28. ISO/IEC/IEEE 29119:2021. Software and systems engineering – Software testing. International Organization for Standardization.
URL: https://www.iso.org/standard/81291.html (дата звернення: 13.05.2026).
29. Slither: A Static Analysis Framework for Smart Contracts. Trail of Bits.
URL: https://github.com/crytic/slither (дата звернення: 13.05.2026).
30. Playwright: End-to-end testing for modern web apps. Microsoft.
URL: https://playwright.dev/ (дата звернення: 21.05.2026).
31. Postman Learning Center. Postman, Inc.
URL: https://learning.postman.com/ (дата звернення: 21.05.2026).
32. Lighthouse documentation. Google Developers.
URL: https://developer.chrome.com/docs/lighthouse/
(дата звернення: 22.05.2026).
33. Web Vitals. Google Developers. URL: https://web.dev/articles/vitals
(дата звернення: 22.05.2026).

34. Скубій Є.В., Дмитрюк В.В. Архітектурні підходи та інженерні практики розробки децентралізованих веб-застосунків на базі блокчейну. Матеріали XVIII Всеукраїнської науково-практичної конференції студентів, аспірантів та молодих вчених за тематикою «Тенденції розвитку ІТ-технологій в Україні». 23-24 квітня 2026 р., Черкаси, Україна. Черкаси: Черкаський державний фаховий бізнес-коледж, 2026. С. 143-145.

ДОДАТКИ

Додаток А – Посилання на репозиторій із програмним комплексом

Репозиторій з вихідним кодом застосунку Ethera розміщено на платформі GitHub. URL-адреса: https://github.com/zhenyaskubiy/Ethera

Додаток Б – Скріншоти інтерфейсу застосунку Ethera

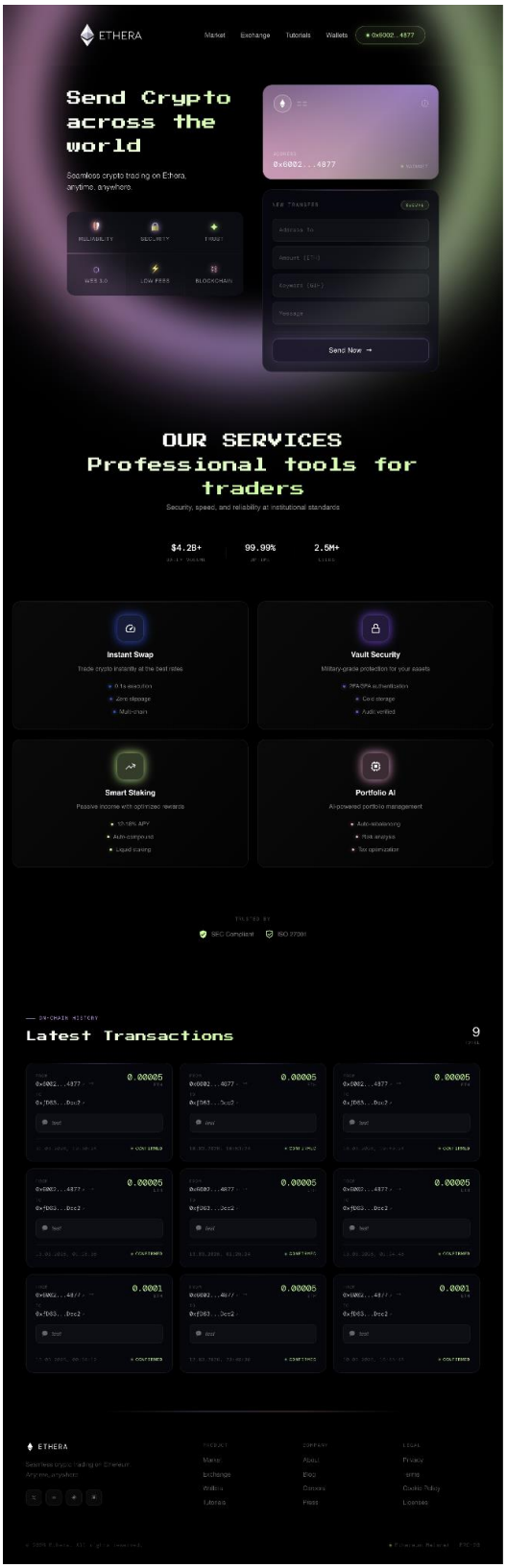

Рисунок Б.1 – Інтерфейс головної сторінки застосунку (компонент Welcome)