МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп'ютерних наук

Кафедра інтелектуальних інформаційних систем

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувач кафедри інтелектуальних інформаційних систем

_____Юрій КОНДРАТЕНКО

«____»_____ 2025 р.

КВАЛІФІКАЦІЙНА РОБОТА

НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВРА

ЗАСТОСУВАННЯ БЛОКЧЕЙН ТЕХНОЛОГІЇ ДЛЯ РОЗРОБКИ ДЕЦЕНТРИЛІЗОВАНИХ СИСТЕМ З ВИСОКИМ РІВНЕМ БЕЗПЕКИ

Спеціальність 122 Комп'ютерні науки

Освітня програма «Комп'ютерні науки»

Здобувач _____ Денис ВОРОНЦОВ

«___»_____ 2025 р.

Керівник д-р пед. наук, проф. _____ Олександр МЄЩАНІНОВ

«___»_____ 2025 р.

Миколаїв – 2025

Чорноморський національний університет імені Петра Могили
(повне найменування закладу вищої освіти)

Факультет	Комп’ютерних наук
Кафедра	Інтелектуальних інформаційних систем
Рівень вищої освіти	Перший (бакалаврський)
Освітній ступень	Бакалавр
Спеціальність	122 Комп’ютерні науки
Освітня програма	Комп’ютерні науки

ЗАТВЕРДЖУЮ

Завідувач кафедри інтелектуальних інформаційних систем

_____ Юрій КОНДРАТЕНКО

«____» _____ 20___ р.

ЗАВДАННЯ
на кваліфікаційну роботу здобувача

Воронцова Дениса Олександровича
((прізвище, ім’я, по батькові здобувача)

1. Тема кваліфікаційної роботи: «Застосування блокчейн технології для розробки децентрилізованих систем з високим рівнем безпеки».

Керівник роботи:Мєщанінов Олександр Павлович д-р пед. наук, проф.

Затверджена наказом ЧНУ ім. Петра Могили від «18» листопада 2024 р. № 321.

2. Строк представлення кваліфікаційної роботи «___» _____ 2025 р.

3. Очікуваний результат роботи та початкові дані, якщо такі потрібні: система збереження та перевірки гешів цифрових документів у блокчейні Ethereum; приклади документів, смарт-контракт Solidity, сервер FastAPI, тестове середовище

Ganache.

4. Перелік питань, що підлягають розробці: аналіз сучасного стану проблеми автентичності цифрових документів у мережевому середовищі; огляд існуючих підходів до захисту та перевірки документів з використанням блокчейн-технологій; формалізація вимог до децентралізованої системи перевірки; проєктування смарт-контракту для збереження гешів документів; реалізація серверної логіки та API для взаємодії з блокчейном; створення користувацького інтерфейсу; функціональне та безпекове тестування реалізованого рішення.

5. Перелік графічних матеріалів: презентація.

Керівник роботи _____ Олександр МЄЩАНІНОВ

(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Здобувач _____ Денис ВОРОНЦОВ

(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Дата видачі завдання «18» грудня 2024 р.

КАЛЕНДАРНИЙ ПЛАН
кваліфікаційної роботи

Тема: Застосування блокчейн технології для розробки децентрилізованих систем з високим рівнем безпеки

№	Найменування роботи	Початок	Закінчення	Примітки
1	Отримання завдання на виконання КР	18.12.2024	20.12.2024	
2	Аналіз предметної області та постановка задачі	21.12.2024	30.01.2025	
3	Огляд літературних джерел за темою, зокрема досліджень і прикладів децентралізованих систем на базі блокчейну	31.01.2025	01.03.2025	
4	Огляд архітектур блокчейн-систем і моделей консенсусу з позиції інформаційної безпеки	02.03.2025	01.04.2025	
5	Проєктування системи: опис архітектури, компонентів, API та смарт-контракту	02.04.2025	15.05.2025	
6	Перший попередній захист КР на засіданні комісії кафедри	16.05.2025	16.05.2025	
7	Корегування роботи за результатами попереднього захисту	29.05.2025	29.05.2025	
8	Другий попередній захист КР на засіданні комісії кафедри	30.05.2025	30.05.2025	
9	Доробка та остаточне оформлення КР	31.05.2025	10.05.2025	
10	Подання КР, її електронної копії та інших документів (відгуку, рецензії) до захисту	11.06.2025	17.06.2025	

Керівник роботи _____ Олександр МЄЩАНІНОВ
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Здобувач _____ Денис ВОРОНЦОВ
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Дата складання календарного плану
«29» січня 2025 р.

АНОТАЦІЯ

до кваліфікаційної роботи
здобувача групи 402 ЧНУ ім. Петра Могили

Воронцова Дениса Олександровича

на тему: **“ ЗАСТОСУВАННЯ БЛОКЧЕЙН ТЕХНОЛОГІЇ ДЛЯ РОЗРОБКИ ДЕЦЕНТРИЛІЗОВАНИХ СИСТЕМ З ВИСОКИМ РІВНЕМ БЕЗПЕКИ”**

Актуальність даної роботи зумовлена зростаючою потребою у надійному та прозорому підтвердженні автентичності цифрових документів у різних сферах: освіті, юридичному обігу, публічному адмініструванні та бізнесі. У роботі розроблено децентралізовану систему з використанням блокчейн-технології Ethereum для збереження гешів документів та їх подальшої перевірки через смарт-контракт. Запропоноване рішення усуває необхідність у централізованій перевірці, забезпечуючи криптографічну незмінність записів, відкритість перевірки та стійкість до фальсифікацій.

Об’єктом роботи є процес верифікації цифрових документів у децентралізованих інформаційних системах.

Предметом роботи є методи збереження геш-функцій файлів у блокчейні з використанням смарт-контрактів.

Метою роботи є підвищення рівня безпеки, довіри та надійності під час перевірки автентичності цифрових документів шляхом створення прозорої децентралізованої системи на основі Ethereum.

У результаті виконання роботи проаналізовано сучасні проблеми цифрової верифікації, досліджено архітектури блокчейн-систем, спроєктовано та реалізовано програмне рішення із серверною частиною (FastAPI), смарт-контрактом на Solidity, клієнтським інтерфейсом та API. Проведено тестування функціональності, перевірено захист від несанкціонованого доступу, обробку помилок та fallback-механізм у разі відмови мережі.

Кваліфікаційна робота складається з чотирьох розділів. У першому розділі проаналізовано предметну область і сформульовано задачу. У другому — розглянуто блокчейн-технології та методи криптографічного захисту. У третьому — описано архітектуру, смарт-контракт, API та безпекову модель. У четвертому — реалізовано

систему, проведено тестування та оцінено результати. Загальний обсяг роботи – 79 сторінок. Кваліфікаційна робота містить 9 рисунків і 30 джерел посилання.

Ключові слова: блокчейн, верифікація документів, смарт-контракт, геш-функція, Ethereum, FastAPI, Web3.

ABSTRACT

to the qualification work by the student of the group 402 of Petro Mohyla Black Sea National University

Vorontsov Denys Oleksandrovych

“ APPLICATION OF BLOCKCHAIN TECHNOLOGY FOR THE DEVELOPMENT OF DECENTRALIZED SYSTEMS WITH A HIGH LEVEL OF SECURITY”

The relevance of this work is due to the growing need for reliable and transparent verification of the authenticity of digital documents in various fields such as education, legal circulation, public administration, and business. The thesis presents a decentralized system based on Ethereum blockchain technology for storing document hashes and verifying their authenticity via smart contracts. The proposed solution eliminates the need for centralized verification by ensuring cryptographic immutability of records, open access to verification, and resistance to falsification.

The object of the work is the process of digital document verification in decentralized information systems.

The subject of the work is the methods of storing file hash functions in the blockchain using smart contracts.

The purpose of the thesis is to enhance the security, trust, and reliability of digital document verification by developing a transparent decentralized system based on Ethereum.

As a result of the work, current problems of digital verification were analyzed, blockchain system architectures were studied, and a software solution was designed and implemented — including a backend based on FastAPI, a Solidity smart contract, a client interface, and an API. Functional testing was carried out, and protection against unauthorized access, error handling, and a fallback mechanism in the event of network failure were

verified.

The qualification thesis consists of four chapters. The first chapter presents the analysis of the subject area and the problem statement. The second chapter discusses blockchain technologies and cryptographic protection methods. The third chapter describes the architecture, smart contract, API, and security model. The fourth chapter includes system implementation, testing, and evaluation of results. The total volume of the thesis is 79 pages. It includes 9 figures, and 30 references.

Keywords: blockchain, document verification, smart contract, hash function, Ethereum, FastAPI, Web3.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

API — Application Programming Interface / Програмний інтерфейс прикладного програмування

ABI — Application Binary Interface / Бінарний інтерфейс застосунку

BTC — Bitcoin / Криптовалюта першого покоління

CSV — Comma Separated Values / Формат текстових таблиць із розділенням комами

DAO — Decentralized Autonomous Organization / Децентралізована автономна організація

dApp — Decentralized Application / Децентралізований застосунок

ECDSA — Elliptic Curve Digital Signature Algorithm / Алгоритм цифрового підпису на основі еліптичних кривих

ETH — Ethereum / Платформа блокчейн другого покоління

Ganache — Локальне середовище для тестування Ethereum-смарт-контрактів

Gas — Одиниця обчислень вартості транзакцій у мережі Ethereum

HSM — Hardware Security Module / Апаратний модуль безпеки

JSON — JavaScript Object Notation / Формат обміну даними у вигляді об’єктів

NFT — Non-Fungible Token / Невзаємозамінний токен

PoA — Proof of Authority / Протокол консенсусу на основі авторитету

REST — Representational State Transfer / Архітектурний стиль вебсервісів

RPC — Remote Procedure Call / Віддалений виклик процедур

SHA-256 — Secure Hash Algorithm 256-bit / Криптографічна геш-функція

SBT — Soulbound Token / Прив’язаний до особи токен

SPA — Single Page Application / Односторінковий застосунок

UI — User Interface / Інтерфейс користувача

URL — Uniform Resource Locator / Уніфікований локатор ресурсу

Web3 — Технології децентралізованого Інтернету третього покоління

Web3.py — Python-бібліотека для взаємодії з Ethereum

ВСТУП

У сучасному цифровому середовищі, де документообіг дедалі активніше переходить у онлайн-простір, забезпечення автентичності та цілісності документів стає надзвичайно актуальним завданням. Традиційні підходи до верифікації зазвичай базуються на централізованих серверах або ручній перевірці, що створює ризики шахрайства, вразливості до зовнішніх атак та технічних збоїв. Такий підхід не гарантує достатньої надійності у довготривалому зберіганні цифрових доказів.

Розповсюдження підроблених документів, фальсифікація дипломів, контрактів і сертифікатів стали серйозною проблемою для освітніх установ, роботодавців, державних структур та приватного бізнесу. Саме тому зростає попит на нові технологічні рішення, здатні забезпечити довіру до цифрових документів без необхідності посередників.

Однією з найперспективніших технологій у цій сфері є блокчейн — децентралізована система зберігання інформації, що використовує криптографію для забезпечення незмінності та прозорості даних. Блокчейн дозволяє зберігати геш-функції документів у публічному реєстрі, який неможливо змінити без виявлення. Це відкриває можливість створення системи цифрової верифікації, яка не потребує централізованої довіри.

Метою цієї роботи є створення децентралізованої системи зберігання гешів документів на базі Ethereum-блокчейну із застосуванням смарт-контрактів. Для цього розроблено Full-Stack рішення, що включає серверну частину на Python (FastAPI), інтерфейс взаємодії з блокчейном через Web3.py, та смарт-контракт на Solidity, який відповідає за запис і перевірку гешів. Особливістю реалізації є гібридна модель з резервним локальним зберіганням, підтримка кількох середовищ (Ganache, Sepolia, Mainnet), обробка транзакцій із підписом приватним ключем, а також логування усіх операцій. Такий підхід дозволяє забезпечити стійкість, масштабованість та практичну придатність системи для використання в умовах реального середовища.

1 АНАЛІЗ ПРЕДМЕТНОЇ СФЕРИ ВЕРИФІКАЦІЇ ДОКУМЕНТІВ ТА БЛОКЧЕЙН ТЕХНОЛОГІЙ. ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз проблем верифікації підлинності документів у цифровому серед овищі

У сучасному цифровому середовищі проблема автентичності документів виходить на передній план. Попри переваги електронного документообігу, такі як швидкість, доступність і економічність, зростає і ризик маніпуляцій, підробок і втрат даних. Це стосується як приватного бізнесу, так і державного сектору, освіти, медицини, судочинства та фінансів.

Окрім суто технічних аспектів, верифікація цифрових документів стикається з серйозними організаційними та правовими викликами. Навіть за наявності централізованої системи перевірки, довіра до її адміністратора стає критичним фактором. Адже якщо дані контролює обмежене коло осіб або приватна структура, це створює загрозу навмисного або випадкового викривлення інформації. Крім того, централізовані системи не захищені від повної втрати або підміни даних у випадку збоїв, атак або внутрішнього зловживання. Такі ситуації вже мали місце в історії цифрового адміністрування і доводять обмеження класичних моделей зберігання.

Особливо критичною є ситуація, коли перевірка справжності документа потребує залучення третьої сторони — наприклад, адміністрації закладу, в якому був виданий диплом, або компанії, що видала сертифікат. Залежність від людського чинника, тривалі запити, бюрократія і складність міжнародної взаємодії роблять цей процес повільним, неефективним і вразливим до помилок. Крім того, зі зростанням мобільності та глобалізації виникає потреба у створенні єдиного, глобального, відкритого інструменту перевірки автентичності.

Не менш важливою проблемою є довготривале збереження цифрової цілісності. Електронні документи можуть змінюватися роками після створення, а звичайні засоби

не дозволяють надійно фіксувати стан файлу в певний момент часу. Електронні підписи мають обмежений термін дії, а зміна формату або програмного забезпечення може зробити перевірку неможливою.

Геш-функції дозволяють ефективно вирішити завдання фіксації цифрового стану документа. Вони є математичними алгоритмами, які перетворюють будь-який файл на унікальний код, що змінюється навіть при незначній зміні вмісту. Проте для того, щоб цей код мав юридичну та практичну цінність, його необхідно зберігати в середовищі, яке саме по собі є захищеним, публічним і незмінним.

У цьому контексті блокчейн виступає унікальним рішенням. Його децентралізована природа, незмінність блоків та прозорість транзакцій створюють умови, у яких можна зберігати геш-функції документів без страху втрати або модифікації. Якщо геш документу зберігається у смарт-контракті на блокчейні, будь-хто може самостійно перевірити його справжність без звернення до центрального сервісу. А наявність запису в блокчейні підтверджує, що документ у такому вигляді вже існував на момент транзакції.

Для кращого розуміння проблеми та її вирішення на рисунку 1.1 нижче подано порівняльну схему централізованої та децентралізованої моделі верифікації документів. На зображенні показано, як у централізованій системі дані проходять через один сервер перевірки, який є вузьким місцем, а у блокчейн-моделі перевірка здійснюється напряму через смарт-контракт без посередників. Така модель усуває необхідність довіри до конкретного серверу чи адміністратора, замінюючи її математичною достовірністю та публічною перевірюваністю.

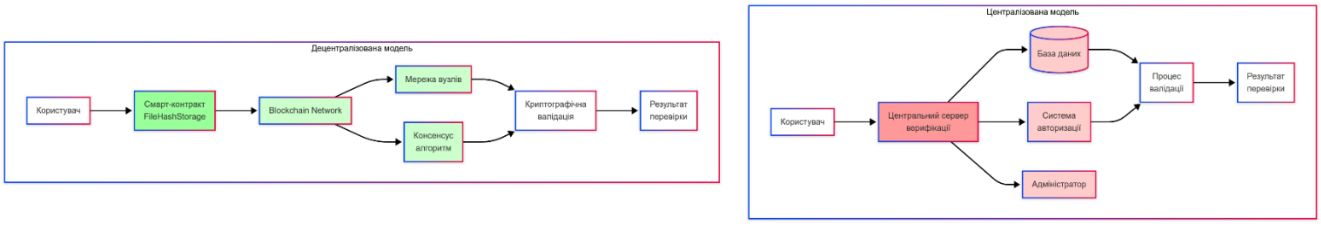

Рисунок 1.1 – Порівняння централізованої та децентралізованої моделі перевірки документів у цифровому середовищі

У результаті, блокчейн і гешування створюють принципово нову парадигму довіри в цифровому світі — не через посередника чи сертифікат, а через відкриту, незмінну, децентралізовану технологію. Саме така система і є предметом дослідження та реалізації у межах даної роботи.

Принципи, на яких базується блокчейн, особливо важливі саме у контексті перевірки достовірності. Публічність даних у розподіленому реєстрі дозволяє будь-кому отримати доступ до записів, пов’язаних з певним документом, що створює абсолютно новий рівень прозорості. У той же час, зберігання лише гешів документів, а не самих файлів, захищає від витоку чутливої або конфіденційної інформації. Таким чином, досягається баланс між перевірюваністю та приватністю — перевірка автентичності можлива, але без розкриття змісту самого документа.

Особливої уваги заслуговує використання смарт-контрактів як механізму фіксації та взаємодії з гешами. Смарт-контракт — це програма, яка виконується в блокчейн-мережі та працює за заздалегідь визначеними правилами. Він може фіксувати геш документа, перевіряти його відповідність у майбутньому, зберігати історію змін (якщо це передбачено логікою), та навіть генерувати події — наприклад, про оновлення чи виявлення підробки. Це дозволяє створити автоматизовану, достовірну, юридично захищену інфраструктуру перевірки, яка не потребує втручання адміністраторів або сторонніх сервісів.

Інша перевага такої системи — її стійкість до цензури та недоступності. У традиційних умовах, якщо сервер або база даних недоступна, перевірка неможлива. У випадку з блокчейном, навіть при відмові деяких вузлів мережі або спробах блокування, залишаються сотні інших, які зберігають ту ж саму інформацію. Це гарантує довготривале збереження цифрового сліду незалежно від політичних, технічних чи адміністративних змін.

Усе це робить блокчейн одним із найперспективніших підходів до побудови систем верифікації у сферах, де важлива відкритість, недовіра до централізованих

структур та необхідність захисту цифрових доказів. Йдеться не лише про навчальні дипломи або сертифікати, але й про медичні висновки, контракти, патенти, судові рішення або внутрішні службові документи, які можуть знадобитися через роки.

Підсумовуючи, можна сказати, що проблема перевірки автентичності цифрових документів є багаторівневою: вона включає як технічні бар'єри, так і організаційно-правові обмеження. Традиційні способи захисту виявляються або занадто складними в реалізації, або вразливими до втрат та змін. Водночас блокчейн забезпечує основу для створення систем, які є прозорими, незмінними, незалежними від одного постачальника, та відкритими для перевірки. Саме на цих засадах базується архітектура розроблюваної системи в межах цієї кваліфікаційної роботи.

1.2 Огляд існуючих методів захисту документів та блокчейн рішень для верифікації даних

Захист цифрових документів є предметом постійного дослідження та вдосконалення в різних галузях — від інформаційної безпеки до правових технологій. Історично склалося так, що перші спроби забезпечити автентичність електронних документів ґрунтувалися на електронному цифровому підписі (ЕЦП). ЕЦП став офіційно визнаним механізмом засвідчення особи підписанта в юридичних транзакціях. Він використовує криптографію з відкритим ключем, де пара ключів (приватний і публічний) дозволяє зашифрувати і перевірити документ. Проте попри правову значущість, ЕЦП має низку недоліків: зокрема, залежність від сертифікатів, центрів сертифікації, необхідність зберігання ключів у безпечному середовищі та обмеженість терміну їхньої дії.

Іншим поширеним підходом є використання цифрових водяних знаків (watermarks). Це приховані елементи, вбудовані у файл, які важко помітити без спеціального ПЗ. Водяні знаки можуть містити інформацію про автора, дату створення чи унікальний ідентифікатор, що допомагає в ідентифікації походження файлу. Такий підхід активно використовується у медіа, видавництвах, правовій та

науковій документації. Проте він має критичний недолік — водяний знак можна стерти або підробити, а тому він не може виступати єдиним доказом автентичності.

У деяких системах впроваджуються QR-коди або унікальні ідентифікатори, які вбудовуються в документ і ведуть на спеціальний сервер перевірки. Цей сервер зберігає інформацію про файл або його хеш, що дозволяє перевірити справжність при скануванні коду. Подібний підхід застосовується в державних документах, сертифікатах вакцинації, дипломах та довідках. Проте тут знову ж існує проблема довіри до централізованого серверу. У разі його недоступності, атаки або знищення — перевірка стає неможливою.

Варто також згадати PDF-файли з вбудованим цифровим підписом. У таких документах інформація про підписанта зберігається разом із вмістом, і будь-яка зміна порушує валідність підпису. Це корисно для контрактів, офіційних заяв або звітів, але знову ж таки залежить від довіри до сертифіката та підтримки певного програмного забезпечення. Більше того, не всі PDF-переглядачі коректно відображають інформацію про підпис або попереджають про підробку, що знижує ефективність цього методу.

Зважаючи на вразливості традиційних способів перевірки, з початком розвитку блокчейн-технологій з’явилася нова хвиля рішень, що дозволяють використовувати децентралізовані реєстри для зберігання цифрових слідів документів. Найпоширенішим підходом стало збереження хешу документа в блокчейні. Хеш обчислюється на основі вмісту документа, не розкриваючи його зміст. Таким чином, якщо хеш документа співпадає з хешем, що був записаний у блокчейні, можна однозначно довести його справжність без потреби у доступі до централізованої бази.

Ці системи вже реалізовано в кількох відкритих або комерційних проєктах. Наприклад, рішення OpenCerts, створене в Сінгапурі, дозволяє університетам видавати дипломи, які можна перевірити за допомогою хешу, збереженого у блокчейні Ethereum. Аналогічно працює Blockcerts — відкритий стандарт для

верифікації сертифікатів, розроблений MIT. Такі підходи не тільки забезпечують прозорість, а й дозволяють будь-якому зацікавленому користувачу перевірити справжність документа без звернення до закладу, який його видав.

У сучасній практиці блокчейн-рішення поділяються на два основні підходи. Перший — це використання публічних блокчейн-мереж (наприклад, Ethereum, Polygon, Bitcoin) для збереження хешів або метаданих документів. У цьому випадку документ хешується локально, і отриманий цифровий слід зберігається у вигляді транзакції у блокчейні. У майбутньому будь-який користувач може повторно обчислити хеш документа і порівняти його з тим, що збережений у мережі. Якщо вони ідентичні — документ справжній.

Другий підхід — це використання приватних або консорціумних блокчейнів, які керуються обмеженим колом учасників (наприклад, у медичній системі або міжбанківській інфраструктурі). Такі рішення мають переваги у швидкості та керованості, але втрачають частину відкритості та децентралізації. Проте навіть у таких мережах можна застосовувати ті самі криптографічні принципи захисту даних.

З-поміж блокчейн-рішень, які використовуються для верифікації документів, варто виділити не лише проєкти з відкритим кодом, але й великі корпоративні ініціативи. Наприклад, корпорація IBM розробила платформу IBM Blockchain, яка використовується для перевірки походження товарів, сертифікатів відповідності, та навіть дипломів. У Європейському Союзі розглядаються ініціативи щодо інтеграції таких систем у внутрішні реєстри, наприклад у межах EBSI (European Blockchain Services Infrastructure) — проєкту, що передбачає перевірку освітніх сертифікатів на рівні ЄС.

Ще одним цікавим напрямком є використання токенізації — створення NFT або SBT (soulbound tokens), які символізують цифровий документ у блокчейні. У цьому випадку підтвердження справжності відбувається через наявність унікального токена, який не може бути переданий або дубльований, і є доказом належності документа

певному користувачу.

Щоб краще зрозуміти різницю між традиційними методами верифікації та блокчейн-підходами, у нижній частині сторінки наведено рисунок 1.2. У ньому порівнюються ключові характеристики захисту документів у централізованих системах (наприклад, сервер із базою даних, який зберігає хеші або самі файли) та децентралізованих рішеннях на основі блокчейну (наприклад, Ethereum або Hyperledger). Зображено такі параметри, як відкритість доступу, стійкість до втрат, захищеність від підміни та наявність посередника.

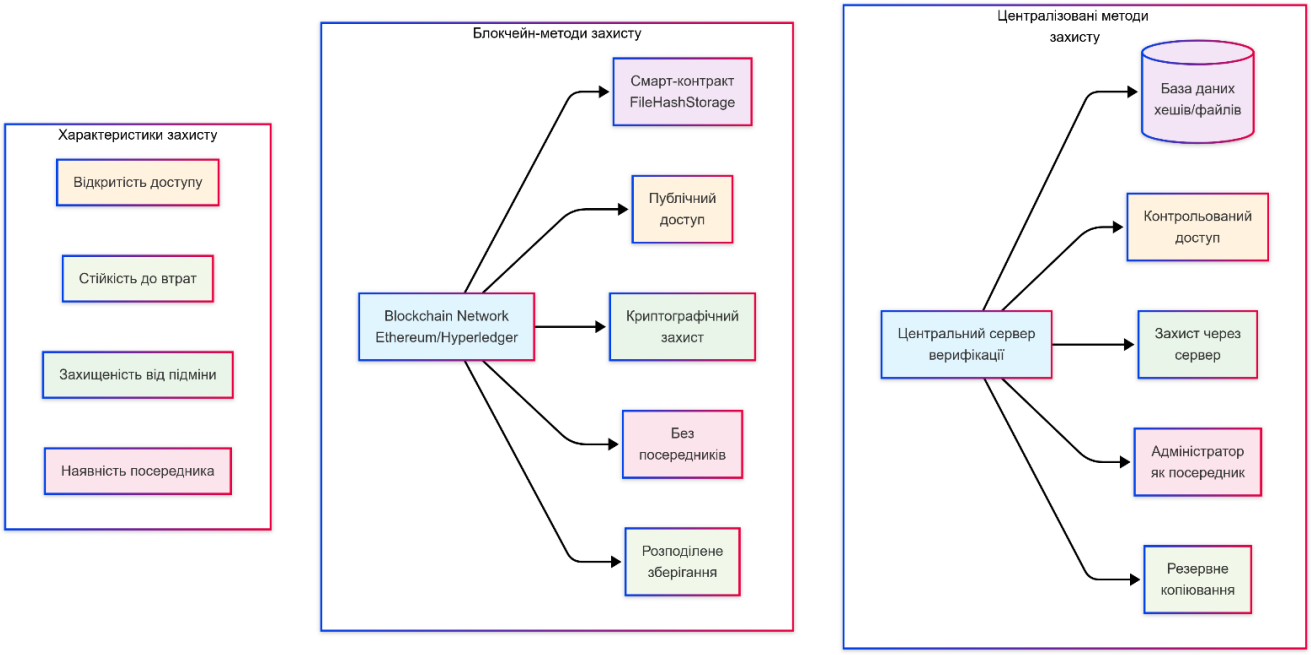

Рисунок 1.2 – Порівняння централізованих та блокчейн-методів захисту цифрових документів

Цей рисунок наочно демонструє, що основною перевагою блокчейн-підходу є усунення необхідності в довіреній стороні. Якщо у централізованій системі будь-яка перевірка базується на достовірності сервера, то у децентралізованій — на незмінності

математичних записів у публічному реєстрі. Таким чином, технологія блокчейн перетворює принцип «довіряй, але перевіряй» на «не довіряй, а перевір самостійно».

На завершення, варто зазначити, що попри стрімкий розвиток таких технологій, їхнє широке впровадження все ще обмежується бар'єрами інтеграції, відсутністю єдиних стандартів, а також недостатньою поінформованістю користувачів. Водночас успішні приклади на кшталт OpenCerts, Blockcerts, IBM Blockchain або EBSI свідчать про те, що блокчейн-підхід уже не є теоретичною можливістю — він є практичним, перевіреним інструментом, який дає змогу розробити безпечні та універсальні системи верифікації цифрових даних.

1.3 Постановка задачі розробки децентралізованої системи верифікації документів

На основі проведеного аналізу можна зробити висновок, що існуючі методи захисту цифрових документів, хоча й мають певну ефективність, не забезпечують достатнього рівня довіри, довготривалого зберігання доказів та захищеності від підміни або втрати. Централізовані сервіси вимагають довіри до адміністратора, можуть бути недоступними або скомпрометованими, а використання лише локальних механізмів перевірки обмежує універсальність таких рішень.

Ці недоліки особливо критичні у випадках, коли документ має зберігатися багато років, використовуватись у різних юрисдикціях, або повинен бути перевірений незалежною стороною без звернення до власника документа чи видавця. Таким чином, виникає об'єктивна потреба у розробці нової архітектури, яка поєднує наступні ключові вимоги: децентралізація, незмінність запису, публічна перевірка, захист без розкриття вмісту документа, незалежність від конкретного провайдера або центра.

Метою цієї роботи є проєктування та реалізація децентралізованої інформаційної системи, яка дозволяє користувачам:

– завантажувати будь-який документ (PDF, DOC, TXT тощо);

– автоматично обчислювати його криптографічний геш (SHA-256);
– зберігати геш у смарт-контракті на блокчейні Ethereum;
– перевіряти справжність документа шляхом порівняння гешів;
– бачити результат перевірки без доступу до оригінального файлу.

Для досягнення поставленої мети система має бути побудована на основі технологій FastAPI (Python) для створення REST API, Web3.py для взаємодії з Ethereum, а також смарт-контракту на Solidity, який відповідає за збереження і перевірку хешів. Додатково має бути реалізована система логування, перевірки типу й розміру файлу, а також fallback-механізм на випадок недоступності блокчейну.

На рисунку 3 нижче представлено блок-схему загальної логіки роботи системи. Схема починається з етапу завантаження документа користувачем, далі система обчислює геш-функцію та звертається до блоку з API. Через Web3-інтерфейс геш передається в смарт-контракт, який фіксує його у блокчейні. У випадку перевірки — процес йде у зворотному напрямку: обчислюється новий геш і порівнюється з тим, що збережено у контракті.

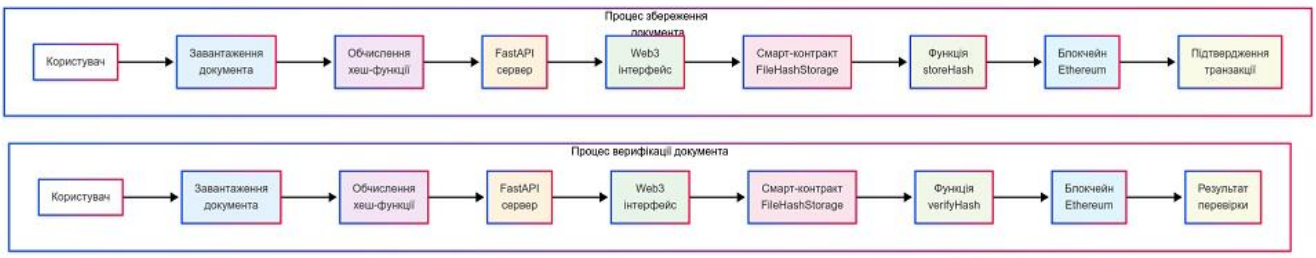

Рисунок 1.3 – Блок-схема архітектури системи децентралізованої верифікації документів через блокчейн Ethereum.

Важливо підкреслити, що користувачі не зберігають сам документ у блокчейні, а лише його цифровий відбиток — це дозволяє зберігати конфіденційність даних. Також система має бути максимально простою у використанні, з інтерфейсом, який

дозволяє виконати перевірку за кілька кроків, без необхідності встановлювати додаткове програмне забезпечення чи мати технічні знання у сфері криптографії або блокчейну.

Таким чином, сформульоване технічне завдання полягає в розробці інфраструктури, що забезпечує: стійкість до підробок, доступність перевірки, простоту використання та технологічну незалежність. Реалізація такої системи відкриває нові можливості для застосування блокчейну в документообігу та підвищує довіру до цифрових даних у найрізноманітніших сферах — від освіти до юридичної практики.

Висновки до розділу 1

У першому розділі було проведено глибокий аналіз сучасного стану проблеми верифікації підлинності документів у цифровому середовищі. Було виявлено, що традиційні методи захисту, такі як електронний цифровий підпис, водяні знаки, QR-коди або централізовані сервери перевірки, мають обмеження у сфері довіри, довготривалості, доступності та юридичної стійкості. Вони часто залежать від наявності технічного посередника, що знижує надійність системи в умовах масштабованості або міжнародного використання.

Зі зростанням обсягів цифрових документів, потребою в автоматизованій перевірці та захисті від підробок, виникла необхідність у принципово новому підході, який би поєднував криптографічну надійність, відкритість перевірки та децентралізовану архітектуру. Блокчейн-технології, зокрема публічні реєстри як Ethereum, дозволяють створити саме таку систему — без залежності від адміністратора, із публічним підтвердженням геш-функції документа, і з неможливістю змінити запис без згоди мережі.

У межах розділу також було проаналізовано приклади існуючих реалізацій, таких як OpenCerts, Blockcerts та ініціативи IBM та ЄС, які вже демонструють

практичне застосування блокчейну у верифікації сертифікатів і дипломів. Це доводить актуальність обраного підходу.

На завершення було сформульовано задачу дослідження: розробити децентралізовану систему верифікації документів на основі гешування та збереження хешів у смарт-контракті Ethereum. Така система повинна бути безпечною, надійною, простою у використанні та незалежною від центральних точок відмови. У подальших розділах буде представлено архітектуру, технології, механізми реалізації та тестування цієї системи.

2 БЛОКЧЕЙН ТЕХНОЛОГІЇ ТА МЕТОДИ КРИПТОГРАФІЧНОГО ЗАХИСТУ ДЛЯ ВЕРИФІКАЦІЇ ДОКУМЕНТІВ

2.1 Аналіз блокчейн технологій та смарт-контрактів для зберігання гешів документів

Блокчейн — це одна з найреволюційніших технологій останніх десятиліть, яка змінила уявлення про зберігання, довіру та контроль над інформацією. Його децентралізована природа, криптографічний захист та відкритість структури створюють передумови для створення таких інформаційних систем, де достовірність даних гарантується самою архітектурою мережі, а не окремим адміністратором чи сервісом.

Основним елементом блокчейну є блоки, які послідовно з’єднані між собою у хронологічному порядку, причому кожен блок містить хеш попереднього. Це створює математично захищений ланцюг, зміна якого можлива лише шляхом зміни усіх наступних блоків, що в реальній публічній мережі (на кшталт Ethereum чи Bitcoin) є практично нездійсненним. Саме ця незмінність і вбудована перевірка кожної дії робить блокчейн ідеальним інструментом для фіксації фактів: хто, що і коли зробив.

У контексті верифікації документів головним завданням є збереження не самих документів, а їх геш-значень — криптографічних відбитків. Геш-функція (наприклад, SHA-256) перетворює будь-який документ у фіксований за довжиною рядок, який змінюється повністю навіть при зміні одного символу у вихідному файлі. Цей підхід гарантує, що документ, який відповідає конкретному гешу, не був змінений з моменту створення цього гешу. Оскільки зворотного відновлення оригінального файлу з гешу неможливо, такий підхід також зберігає конфіденційність.

Зберігання гешів у блокчейні реалізується через смарт-контракти — спеціальні програми, які автоматично виконуються у розподіленій мережі. Смарт-контракт у контексті даної роботи виконує кілька ключових функцій: приймає геш як вхідні дані,

записує його у блокчейн у вигляді транзакції, дозволяє виконувати запити на перевірку гешу, і в деяких випадках — зберігає історію змін або створює події, що сигналізують про дії користувачів.

Однією з переваг використання смарт-контрактів є повна незалежність від серверів або централізованих баз даних. Код смарт-контракту розміщується у блокчейні й доступний для виконання будь-якому користувачу з доступом до мережі Ethereum. Це забезпечує прозорість, відкритість і захист від підміни логіки перевірки. Крім того, усі транзакції зберігаються назавжди, а отже, є юридично фіксованими фактами, які не можна видалити або змінити.

На практиці смарт-контракти можуть реалізовувати різні моделі зберігання гешів. Найпростіша — це звичайний мапінг (mapping), де до кожної адреси користувача прив’язується масив або список збережених гешів. У більш розширених реалізаціях можна додавати мітки часу, ID документа, типи файлів або навіть цифрові підписи власника. Такі структури даних дозволяють реалізовувати логіку перевірки, відстеження, історії змін та навіть підписання через мультипідписи.

Серед найпоширеніших платформ для створення та виконання смарт-контрактів на сьогодні залишається Ethereum, який є найрозвиненішим серед публічних блокчейнів із підтримкою повноцінного Turing-комплектного середовища для написання логіки взаємодії. Ethereum використовує мову програмування Solidity, що дозволяє створювати компактні та функціональні контракти для фіксації, перевірки та оновлення геш-даних.

Під час створення системи верифікації одним із ключових архітектурних рішень є визначення — чи зберігати геші як прості записи у контракті, чи будувати розширену структуру з прив’язкою до користувачів, міток часу, типів документів та ідентифікаторів транзакцій. У базовій моделі (яка застосовується в цій роботі) реалізовано функції storeHash() та verifyHash() з використанням mapping(bytes32 => bool) для збереження факту реєстрації гешу, а також генерування подій HashUpdated

та HashVerified. Такий підхід дозволяє швидко перевірити, чи був документ зареєстрований раніше, не розкриваючи при цьому сам вміст.

З технічної точки зору, смарт-контракт можна умовно поділити на дві частини: функціональну логіку зберігання/перевірки та захисні механізми. У рамках цього проєкту реалізовано модифікатор onlyOwner, який обмежує можливість запису гешів лише власником контракту, щоб запобігти несанкціонованому доступу. У майбутньому така архітектура може бути розширена для підтримки ролей, мультипідписів або DAO-механізмів керування.

Окремо варто зупинитися на структурі транзакцій у блокчейні Ethereum. Кожна взаємодія зі смарт-контрактом супроводжується витратами газу — внутрішньої одиниці розрахунку вартості операцій. Транзакції формуються із зазначенням gas limit, ціни за газ, nonce, адреси контракту, й підписуються приватним ключем. Після надсилання вони очікують включення до блоку та формують receipt, який є офіційним підтвердженням того, що дія була здійснена. У даній системі це використовується для логування результату перевірки та збереження гешу.

Ще однією сильною стороною блокчейн-рішень є можливість прозорої аудита будь-хто може переглянути історію транзакцій, виклики функцій контракту, журнали подій, що особливо важливо у сфері публічного управління та цифрових реєстрів. Наприклад, у системі захисту дипломів або сертифікатів можна перевірити, коли саме було зареєстровано документ, хто його завантажив, і чи змінювався запис з того моменту.

Проте важливо враховувати і обмеження технології. Основними викликами при роботі з публічним блокчейном є:

– вартість газу (особливо в основній мережі Ethereum), яка може змінюватися динамічно;

– обмеження на розмір та обсяг транзакцій;

– затримки при підтвердженні транзакцій у моменти перевантаження мережі;

– необхідність роботи з приватними ключами, які потребують особливого захисту.

Саме тому в рамках проєкту було прийнято рішення використовувати локальну тестову мережу Ganache, яка імітує поведінку Ethereum у безпечному локальному середовищі. Це дозволяє виконувати всі дії в режимі тестування без витрат, а також забезпечує швидкий зворотний зв’язок при розробці. У майбутньому розгортання на публічних мережах (наприклад, Sepolia або Ethereum mainnet) можливе без зміни архітектури, завдяки модульній структурі проєкту.

Переваги такого підходу підтверджуються успішними кейсами застосування в реальних системах. Наприклад, у системі Blockcerts реалізовано зберігання сертифікатів у вигляді гешів у блокчейні, де кожна установа видає цифровий сертифікат із прикріпленим доказом через смарт-контракт. Перевірка відбувається за допомогою вебінтерфейсу або мобільного додатку, де користувач завантажує документ, а система порівнює геш із записом у реєстрі. Цей приклад підтверджує масштабованість і надійність архітектури, аналогічної до тієї, що реалізується в цьому проєкті.

Таким чином, аналіз технологій блокчейн та принципів роботи смарт-контрактів дозволяє зробити висновок, що для задачі перевірки автентичності цифрових документів оптимальним рішенням є гібридна модель, у якій геші зберігаються в публічному реєстрі, а вся логіка взаємодії делегується контракту, що забезпечує прозорість, достовірність і криптографічну стійкість системи. У поєднанні з серверною частиною (FastAPI) та бібліотекою Web3.py це створює повноцінну децентралізовану інформаційну систему, орієнтовану на захист документів без участі централізованих структур.

2.2 Технології розробки децентралізованих застосунків

Децентралізовані застосунки (або dApps — decentralized applications) — це новий клас програмного забезпечення, у якому ключові функції не залежать від

централізованого сервера або бази даних. Вони працюють у середовищі блокчейну, використовуючи смарт-контракти як серверну логіку, а фронтенд-клієнт з'єднується з мережею через Web3-інтерфейси. Такий підхід дозволяє створювати системи, які є стійкими до цензури, прозорими, захищеними та незалежними від одного контролюючого суб'єкта.

Загалом архітектура dApp включає кілька ключових компонентів:

– смарт-контракт, розгорнутий у блокчейн-мережі (наприклад, Ethereum), який реалізує логіку бізнес-процесів;

– клієнтська частина (Frontend) — зазвичай вебінтерфейс на JavaScript або TypeScript, побудований з використанням React, Vue чи іншого сучасного фреймворку ;

– web3-бібліотека, яка забезпечує комунікацію між інтерфейсом та блокчейном (наприклад, Web3.js або Ethers.js) ;

– криптогаманець користувача, який дозволяє підписувати транзакції та автентифікувати дії (найпопулярніший приклад — MetaMask) ;

– бекенд або API-шлюз, який, хоч і не обов'язковий, виконує допоміжні функції зберігання логів, перевірку формату, фільтрацію, підключення до локальних вузлів тощо.

На рисунку 5 нижче представлено загальну архітектуру типової dApp-системи. Зображено користувача, що взаємодіє з вебінтерфейсом через браузер, який через Web3.js підключається до блокчейн-мережі. Смарт-контракт обробляє запити, пов'язані із зберіганням та перевіркою гешів документів. У фоновому режимі працює серверна частина на Python (FastAPI), яка реєструє події, виконує додаткову перевірку, взаємодіє з Web3.py і надає REST-ендпоінти.

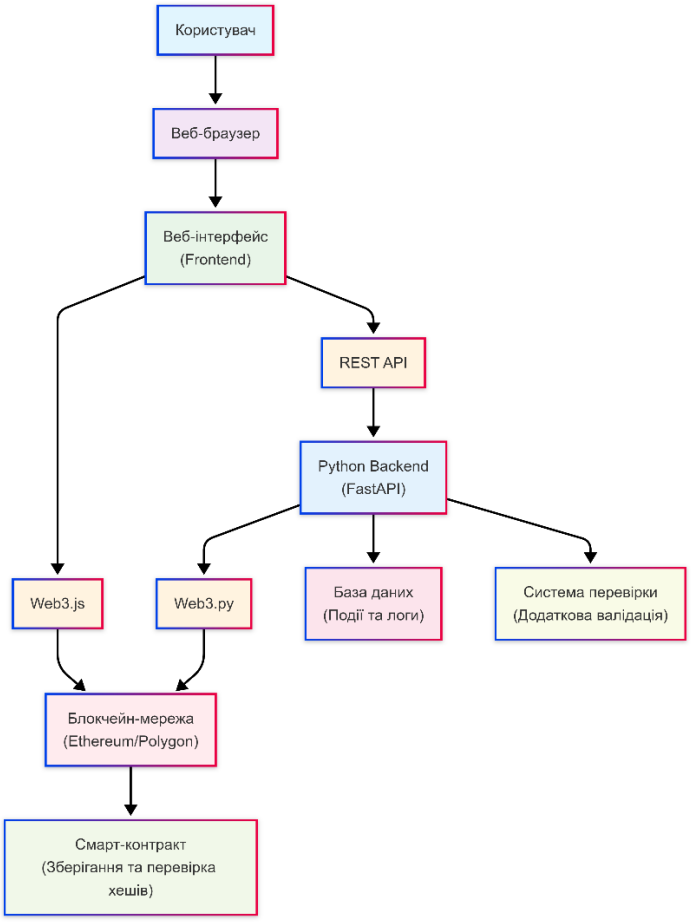

Рисунок 2.1 – Структура типової децентралізованої програми (dApp) з інтеграцією Web3, смарт-контрактів та бекенд-сервісу

Цей підхід дозволяє поєднати надійність блокчейну з гнучкістю традиційної клієнт-серверної моделі. Користувачі взаємодіють із системою напряму через свій гаманець, підписуючи кожну операцію — це створює ефект децентралізованої автентифікації, де приватний ключ не передається третім сторонам. Усі транзакції є публічними та перевіряються мережею, але одночасно — захищені завдяки криптографії.

Фронтенд dApp зазвичай реалізується як односторінковий застосунок (SPA), з інтерактивним інтерфейсом і підключенням до гаманця. При завантаженні сторінки застосунок ініціалізує Web3-з'єднання, перевіряє наявність MetaMask або іншого гаманця, зчитує активний акаунт, і лише після цього дозволяє користувачу

взаємодіяти з контрактом. Наприклад, при завантаженні документа інтерфейс може одразу обчислити його геш, передати на бекенд для підтвердження, і потім створити транзакцію для виклику storeHash().

У більш складних сценаріях dApp підтримує інтеграцію з подіями смарт-контракту — тобто застосунок реагує на event, що створюється під час успішного збереження або верифікації. Це дозволяє реалізувати зворотний зв’язок для користувача, оновлення списків документів, журнал змін або історію транзакцій. Наприклад, після завершення верифікації можна відобразити повідомлення “Документ автентичний” або “Геш не знайдено у реєстрі”.

Ще одним критичним елементом у побудові децентралізованого застосунку є бібліотека Web3, яка виступає містком між клієнтським інтерфейсом та блокчейн-мережею. Найпоширенішими рішеннями є Web3.js та Ethers.js — обидві надають зручні інтерфейси для взаємодії з контрактами, виконання читання/запису в блокчейн, підписування транзакцій, прослуховування подій (events) тощо.

У рамках реалізації цієї роботи перевагу було надано Web3.py — бібліотеці Python, яка дозволяє з серверного боку взаємодіяти зі смарт-контрактом напряму. Це стало важливим рішенням, оскільки основна бізнес-логіка реалізована у FastAPI, а Web3.py забезпечує надійне з’єднання з локальним або публічним вузлом Ethereum. Через Web3.py система надсилає транзакції, підписує їх приватним ключем, отримує підтвердження (receipt), та обробляє відповіді контракту. Таким чином бекенд стає активним учасником блокчейн-процесу, а не лише посередником.

У тестовому середовищі було використано Ganache — популярний інструмент для локального розгортання Ethereum-мережі. Ganache надає віртуальні акаунти, приватні ключі та RPC-інтерфейс для підключення. Це дозволяє розробнику тестувати взаємодію без потреби у реальній мережі, платі за газ або складній конфігурації. Створений смарт-контракт компілюється та деплоїться через Remix або інші інструменти, а його ABI (інтерфейс) використовується у Web3.py для викликів

функцій.

Для підключення до контракту використовуються такі параметри конфігурації: RPC-адреса (http://127.0.0.1:7545), адреса розгорнутого контракту, ABI у форматі JSON та приватний ключ. Це дозволяє програмно ініціалізувати контракт, обчислити nonce, встановити gas limit, створити транзакцію, підписати її та надіслати в мережу.

Важливим аспектом безпеки є обробка приватного ключа. У тестовому середовищі допускається збереження ключа у конфігураційному файлі (config.py), однак при розгортанні на реальну мережу рекомендується використовувати захищені сховища — такі як Vault, KeyStore, або апаратні гаманці. Також критичною є обробка викликів контракту з обмеженням доступу: у цій системі функції запису (storeHash) обмежено модифікатором onlyOwner, що знижує ризик зловживань.

Крім смарт-контрактів, фронтенд і бекенд, dApp зазвичай також містить допоміжні служби журналювання (logging), обробки винятків (error handling) та валідації вхідних даних. Наприклад, перед збереженням гешу перевіряється, чи документ не перевищує встановлений розмір (10MB), чи відповідає форматам (PDF, DOC, TXT), та чи геш не дублюється в системі. Усі ключові події (реєстрація, перевірка, помилки) фіксуються у файлі app.log, що дозволяє вести аудит.

Для створення повноцінного інтерфейсу зручним є застосування бібліотек React у поєднанні з Material UI або іншим UI-фреймворком. Це дозволяє створити інтерактивні компоненти: форми завантаження файлів, панелі результатів перевірки, сповіщення про успішну/невдалу транзакцію тощо. Після ініціалізації Web3-з’єднання інтерфейс відображає активний акаунт користувача, статус з’єднання з мережею та дозволяє виконувати дію лише за умови підтвердження.

Окрему увагу варто приділити обробці подій смарт-контракту. У системі, що розробляється, смарт-контракт емісує події HashUpdated та HashVerified, які можуть бути використані як тригери в інтерфейсі. Наприклад, при збереженні гешу виконується логування не лише з боку бекенду, а й оновлення фронтенду у режимі

реального часу, що підвищує зручність взаємодії та прозорість для користувача.

Таким чином, побудова децентралізованого застосунку — це комплексне завдання, яке охоплює як розробку смарт-контрактів, так і побудову фронтенду, бекенду, логіки обробки, захисту даних та взаємодії з мережею Ethereum. Реалізована система поєднує усі ці компоненти, створюючи зручний, безпечний та технологічно незалежний механізм для перевірки цифрових документів.

Висновки до розділу 2

У другому розділі було детально розглянуто принципи функціонування блокчейн-технологій у контексті зберігання та перевірки гешів цифрових документів. Встановлено, що блокчейн є ефективним інструментом для побудови систем із високим рівнем довіри, прозорості та захисту від фальсифікацій. Завдяки незмінності записів, відкритості до аудиту та використанню криптографії, блокчейн дозволяє зберігати цифрові відбитки документів таким чином, що їх автентичність може бути підтверджена незалежно від третіх сторін.

Особливу увагу було приділено смарт-контрактам на базі Ethereum, які відіграють роль цифрових арбітрів у процесі зберігання та верифікації. Їхня функціональність забезпечує запис гешів, обробку перевірок, генерацію подій і обмеження доступу до критичних функцій. Смарт-контракт у даній системі є основним гарантом цілісності та незмінності записів, що фіксують існування документа на момент його реєстрації.

Також у розділі було розглянуто сучасну архітектуру децентралізованих застосунків (dApps), яка поєднує фронтенд, Web3-інтерфейс, смарт-контракт та бекенд-сервер. Була проаналізована взаємодія між цими компонентами, включно з процесом підписування транзакцій, перевіркою користувача через гаманець, та фіксацією подій смарт-контракту.

Встановлено, що для ефективного функціонування dApp потрібна не лише смарт-контрактна логіка, а й правильна інтеграція з бібліотеками Web3, системами

валідації файлів, обробкою логів та відповідними інтерфейсами для зручної роботи користувачів. У рамках проєкту ці компоненти реалізовано з урахуванням безпеки, масштабованості та модульності, що дозволяє у майбутньому легко перейти від тестової мережі Ganache до публічної Ethereum-мережі.

Таким чином, результати аналізу доводять, що поєднання блокчейн-технологій і сучасних підходів до розробки децентралізованих застосунків дозволяє створити ефективну, стійку до підробок та довготривалу систему верифікації цифрових документів, яка не потребує централізованої довіри.

3 ПРОЄКТУВАННЯ ТА РОЗРОБКА ДЕЦЕНТРАЛІЗОВАНОЇ СИСТЕМИ ВЕРИФІКАЦІЇ ДОКУМЕНТІВ

3.1 Аналіз вимог та архітектура системи

Розробка інформаційної системи перевірки автентичності документів на базі блокчейн вимагає чіткого визначення вимог, які мають бути задоволені як з технічного, так і з функціонального боку. Оскільки така система має працювати у реальному середовищі, обробляти конфіденційні дані та взаємодіяти з мережею Ethereum, її архітектура повинна забезпечувати захищеність, доступність, масштабованість та зручність для кінцевого користувача.

У процесі аналізу було виокремлено кілька категорій вимог: функціональні, нефункціональні, технологічні та обмеження середовища. Їх формалізація дозволяє побудувати систему, яка відповідає як прикладним сценаріям використання (upload, verify), так і загальним принципам безпечного програмного забезпечення.

Функціональні вимоги охоплюють основні можливості, які повинна реалізовувати система. По-перше, користувач повинен мати можливість завантажити документ через вебінтерфейс. По-друге, система має автоматично обчислити геш-функцію (SHA-256) на основі вмісту документа. Далі — надіслати цей геш у смарт-контракт, який зберігає його у блокчейні. У разі спроби перевірки, той самий механізм повинен повторно обчислити геш і порівняти його з раніше збереженим. Якщо геши співпадають — документ вважається автентичним.

Також передбачено можливість перегляду статусу транзакції, логування усіх дій, та виведення результатів перевірки на інтерфейс користувача. Більше того, реалізовано REST API з відповідними ендпоінтами (/upload/, /verify/, /blockchain/transactions) для інтеграції з іншими системами або автоматизованого використання.

Нефункціональні вимоги пов’язані з продуктивністю, надійністю, доступністю

та безпекою. Система повинна обробляти файли розміром до 10 МБ, підтримувати типи PDF, DOCX, DOC, TXT, а також реагувати на помилки — наприклад, невідповідність типу файлу, порушення ліміту розміру, відсутність підключення до блокчейну або помилку в транзакції. Крім того, система має забезпечувати захист від несанкціонованого доступу до функцій смарт-контракту, що досягається використанням модифікатора onlyOwner та збереженням приватного ключа на стороні сервера.

З боку користувача, вимоги до зручності включають простий і зрозумілий вебінтерфейс, мінімум етапів взаємодії та наочне відображення результатів. З точки зору адміністратора — важливо мати доступ до логів операцій, з можливістю аналізу транзакцій, що проходили через систему.

Із точки зору архітектури, було прийнято рішення побудувати гібридну систему, що поєднує децентралізоване зберігання хешів (через Ethereum) із локальною перевіркою документа й бекендом на FastAPI. Такий підхід забезпечує гнучкість, масштабованість і мінімізацію витрат на зберігання даних. Документи самі по собі не передаються у блокчейн, що дозволяє зберегти приватність, а всі операції супроводжуються хешуванням і записом лише гешу.

На рисунку 6 представлено загальну архітектуру системи. У центрі схеми розташований бекенд-сервер на Python з реалізованими REST API, який зв’язується з Ethereum-смартконтрактом через бібліотеку Web3.py. Користувач взаємодіє з фронтендом (React), який надсилає запити до бекенду. Функції storeHash() та verifyHash() реалізовані у смартконтракті, а обчислення SHA-256-гешу відбувається до того, як дані надходять у блокчейн. Окремий компонент відповідає за логування у файл app.log, а у випадку помилки в мережі працює fallback-механізм з локальним JSON-хеш-сховищем.

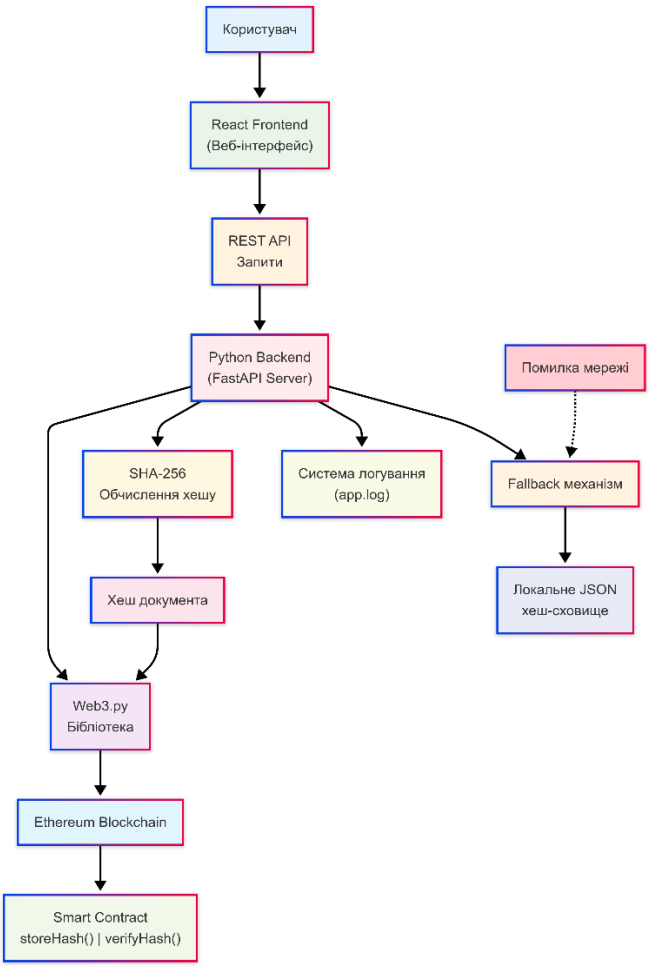

Рисунок 3.1 – Загальна архітектура системи верифікації документів на основі Web3, FastAPI та Ethereum.

Ця архітектура дозволяє гнучко масштабувати систему, адаптуючи її як до локального середовища (Ganache), так і до публічних мереж (Sepolia, Mainnet). Завдяки модульній структурі, можна швидко оновлювати логіку контракту, API, або змінювати спосіб взаємодії без переписування всієї системи.

Важливим аспектом проєктування є аналіз потоків даних у системі. Після ініціації користувачем операції (наприклад, завантаження документа), документ обробляється на стороні браузера або бекенду для обчислення SHA-256-гешу. Цей геш — єдиний елемент, що далі циркулює в системі. Таким чином, з точки зору архітектури, дані поділяються на «внутрішні» (локальні файли, які не зберігаються)

та «зовнішні» (геш, який записується в блокчейн).

Отриманий геш потрапляє до FastAPI-сервера, де відбувається перевірка відповідності (наприклад, чи вже існує такий хеш у локальному кеші), а потім викликається метод Web3-підключення, який готує транзакцію. Система підписує її приватним ключем і надсилає до Ethereum-мережі (в нашому випадку — Ganache RPC), де відбувається остаточне збереження.

У зворотному напрямку (під час верифікації) дані проходять схожий шлях, але без генерації транзакції: хеш обчислюється локально, передається на сервер, який у режимі читання (call) запитує у смарт-контракту, чи такий хеш уже було записано раніше. Це дозволяє здійснювати перевірку швидко, без витрат на газ і без необхідності авторизації.

Реалізований підхід дозволяє мінімізувати залежність від публічної мережі Ethereum. У випадку недоступності RPC або тимчасових збоїв, система переходить у fallback-режим, використовуючи локальне JSON-сховище для тимчасової фіксації хешів. Після відновлення доступу до мережі ці записи можуть бути синхронізовані. Така модель дозволяє забезпечити fault-tolerance — стійкість до відмов — без втрати цілісності даних.

Архітектура підтримує масштабування за кількома напрямками:

– горизонтальне масштабування бекенду через кластеризацію FastAPI-сервера, з балансуванням навантаження;

– паралельна обробка запитів до смарт-контракту завдяки незалежним підключенням через Web3.py;

– мережева адаптивність, яка дозволяє перемикатися між середовищами Ganache, Sepolia або Mainnet без зміни основної логіки (через зміну змінних середовища у config.py);

– розширюваність API, що дозволяє додати нові функції (наприклад, запис історії перевірок, підтримку кількох контрактів або інтеграцію з DAO-механізмами)

без зміни вже існуючих модулів.

З технічної точки зору, важливо також передбачити інтероперабельність системи з іншими платформами або сервісами. Завдяки відкритому REST API та чітко структурованим ендпоінтам (/upload, /verify, /status, /blockchain/transactions), сторонні системи можуть взаємодіяти з нашим рішенням — наприклад, у випадку інтеграції з університетським порталом, юридичним реєстром або навіть електронною поштою. Це створює можливість автоматичної верифікації вкладень, повідомлень або публічних даних.

Безпека є ще одним критичним чинником. Збереження приватного ключа, яким підписуються транзакції, здійснюється на стороні бекенду, і в ідеалі — у захищеному сховищі з обмеженим доступом. У майбутніх ітераціях передбачено перенесення цієї логіки до HSM (hardware security module) або хоча б системи з обмеженим обсягом пам’яті (RAM-only runtime encryption).

Також слід зазначити, що кожен компонент системи ізольовано за принципом модульності: логіка API, Web3, логування, валідація, обробка винятків та робота з файлами рознесені у різні файли Python (main.py, ethereum_client.py, blockchain_manager.py, config.py), що відповідає принципам SOLID і спрощує підтримку та тестування.

Підсумовуючи, запропонована архітектура системи є стійкою, масштабованою, відкритою до інтеграції та придатною до розгортання як у тестовому, так і в бойовому середовищі. Її особливістю є симбіоз сучасного бекенд-розроблення, криптографічного хешування, логіки смарт-контрактів та інтерфейсної простоти, що робить її практичним інструментом для вирішення реальної проблеми цифрової перевірки автентичності.

3.2 Проєктування смарт-контракту та API інтерфейсу

Смарт-контракт є центральним елементом у децентралізованій системі верифікації документів, адже саме він відповідає за фіксацію цифрового сліду

документа (гешу) у блокчейні, перевірку справжності та логування транзакцій. У рамках даного проєкту контракт розроблено на мові Solidity, яка є основною для платформи Ethereum. Його структура сформована відповідно до принципів безпеки, простоти використання та розширюваності.

Основним завданням контракту є збереження унікального гешу документа, який можна перевірити у майбутньому без розкриття самого документа. Для цього в контракті реалізовано дві основні функції:

– storeHash(bytes32 hash) — функція запису хешу у блокчейн. Вона доступна лише власнику контракту завдяки модифікатору onlyOwner. Якщо хеш раніше не існував, він записується як новий запис у мапінгу. У разі повторного запису гешу — генерується подія HashUpdated;

– verifyHash(bytes32 hash) — функція перевірки, яка дозволяє будь-якому користувачу надіслати геш і отримати булеву відповідь: збережено цей геш у реєстрі чи ні. Ця функція не вимагає газу (використовується view-модифікатор) і може бути викликана у режимі читання з фронтенду або бекенду.

Дані у контракті зберігаються у структурі mapping(bytes32 => bool), що забезпечує швидкий доступ і мінімальне споживання газу. Це найпростіша й найефективніша форма реалізації зберігання цифрових слідів у блокчейні, яка дозволяє легко масштабувати систему.

Крім функцій, контракт реалізує два події (events), які дозволяють зовнішнім застосункам реагувати на дії в мережі:

– event HashUpdated(bytes32 indexed hash) — фіксує факт успішного збереження або оновлення гешу;
– event HashVerified(bytes32 indexed hash, bool exists) — використовується при перевірці гешу та дає змогу фронтенду відобразити результат користувачу.

Також реалізовано модифікатор доступу onlyOwner, який забезпечує обмеження для функції storeHash — це дозволяє уникнути ситуацій, коли довільні користувачі

можуть записувати дані до реєстру, що є критично важливим у разі використання приватного ключа сервера.

У коді контракту використано такі конструкції Solidity, як:

– constructor() для встановлення власника контракту;
– require(...) для перевірки прав доступу;
– emit для генерації подій;
– view-функції для зчитування без витрат газу.

На рисунку **7** нижче зображено блок-схему логіки виконання функцій контракту: початок виконання, перевірка ролі, аналіз існування гешу, запис у mapping, генерація подій та завершення транзакції. Схема дозволяє наочно уявити внутрішній цикл роботи smart contract.

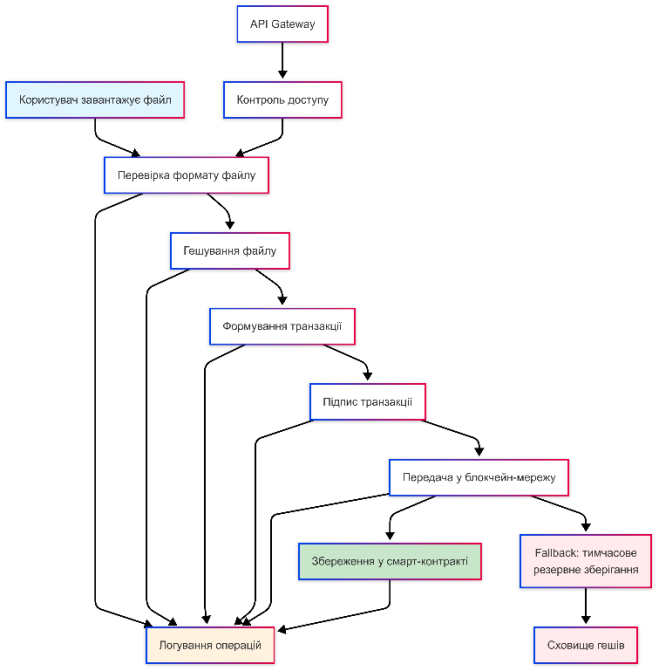

Рисунок 3.2 – Логіка функціонування смарт-контракту з обробкою функцій storeHash() та verifyHash().

Таке проектування дозволяє реалізувати перевірку автентичності документа

через просту, зрозумілу та формально визначену процедуру. Контракт не містить жодних зайвих залежностей, не зберігає важкі або конфіденційні дані, та працює незалежно від бекенд-серверу — тобто вся його логіка децентралізована.

Після проектування смарт-контракту наступним кроком є реалізація інтерфейсу взаємодії між користувачем та блокчейном. У цій системі за цей процес відповідає FastAPI-сервер, який виступає у ролі API-шлюзу та координатора транзакцій. Його функції включають обробку запитів з фронтенду, перевірку даних, обчислення гешів, підключення до Ethereum-мережі через Web3.py, створення та надсилання транзакцій, а також логування результатів.

Архітектура API реалізована як REST-застосунок з чітко визначеними маршрутами:

– POST /upload/ — ендпоінт для завантаження файлу. Користувач надсилає файл, сервер перевіряє тип і розмір, обчислює геш за алгоритмом SHA-256, викликає функцію store_hash_via_contract() і повертає статус запису, хеш та результат транзакції;

– POST /verify/ — перевірка автентичності документа. Сервер приймає файл, обчислює його геш і викликає функцію verify_hash_via_contract() для читання значення з блокчейну. Результат повертається у вигляді логічної відповіді (істинність гешу);

– GET /status — маршрут для перевірки працездатності системи, що дозволяє фронтенду або зовнішнім сервісам перевірити стан API, підключення до Web3 та версію контракту;

– GET /blockchain/transactions — ендпоінт для отримання історії транзакцій, які було надіслано з моменту запуску сервера. Для кожної транзакції зберігається: хеш, час, тип операції (upload / verify), статус (success / error) і унікальний ідентифікатор.

Логіка кожного маршруту реалізована з використанням стандартів Pydantic для

валідації, FastAPI Router для розмежування логіки, і Web3.py для прямого підключення до контракту. Ключовим компонентом є файл ethereum_client.py, у якому реалізовано методи підключення до контракту, генерації транзакції, підпису, надсилання та обробки результату. Всі ці методи мають обробку помилок (try-except) і повертають структури, зручні для передачі через API.

Формат відповіді API структуровано таким чином, щоб забезпечити зручну інтеграцію з іншими системами. Зокрема, у разі перевірки документа (/verify/) користувач отримує у відповідь унікальний геш документа, логічний прапорець успішності перевірки та текстове повідомлення. Наприклад, якщо геш існує у реєстрі, система повертає повідомлення про підтвердження достовірності.

Аналогічно, при збереженні документа через ендпоінт /upload/ система формує відповідь, яка містить геш документа, хеш самої транзакції в блокчейні, статус запиту (наприклад, «submitted» або «error») та супровідне повідомлення, що інформує про результат операції (наприклад, повідомлення про успішне збереження гешу у блокчейні).

Таке текстово-орієнтоване структурування відповіді дозволяє легко інтерпретувати результат як у ручному режимі (через браузер або Postman), так і автоматично — з боку іншої системи або мікросервісу, що може взаємодіяти з даним API для перевірки електронних документів на справжність.

Усі виклики до Web3 реалізовано з обов’язковим контролем nonce, встановленням gas limit, підписом транзакції через eth_account.Account, та очікуванням receipt. Це гарантує, що система не лише створює транзакцію, а й відстежує її фактичне включення у блок, перед тим як повернути результат користувачеві.

Інтерфейс також містить логіку захисту від дублювання: перед записом нового гешу система виконує call до контракту, щоб перевірити, чи такий геш уже є. Якщо запис існує — операція не дублюється, а повертається повідомлення про повторне

підтвердження. Це оптимізує використання ресурсу блокчейну і знижує витрати.

Ще одним важливим аспектом є ізоляція ключових змінних через файл config.py. У ньому зберігаються:

- ETHEREUM_RPC_URL (Ganache / Sepolia / Mainnet);
- PRIVATE_KEY — приватний ключ акаунта-власника контракту;
- CONTRACT_ADDRESS — адреса розгорнутого контракту;
- CHAIN_ID — ідентифікатор мережі.

Це дозволяє швидко переключити систему на інше середовище без зміни основного коду.

У загальному, API FastAPI є повноцінною обгорткою для взаємодії із смарт-контрактом Ethereum, яка захищає, спрощує та структурує усі дії користувача — від завантаження файлу до виведення результату на екран.

3.3 Аналіз безпеки та децентралізованості рішення

Безпека є одним із критичних факторів при розробці будь-якої системи, що працює з цифровими документами, особливо в умовах використання публічних блокчейн-мереж. У запропонованому рішенні реалізовано декілька рівнів захисту — як на стороні смарт-контракту, так і у серверній логіці, що взаємодіє з користувачем та Ethereum-мережею. Кожен етап обробки запиту, зберігання гешу та перевірки супроводжується технічними або криптографічними обмеженнями, які мінімізують ризики атак або втручання.

Першим і найважливішим захистом є обмеження доступу до функції збереження гешів. Це реалізується у смарт-контракті за допомогою модифікатора onlyOwner, який дозволяє викликати функцію storeHash виключно від імені власника контракту. Таким чином, випадковий користувач не має можливості зловживати системою, записуючи хеші довільних або фальшивих документів.

Другим рівнем захисту є контроль приватного ключа, який використовується для підписування транзакцій. У реалізації системи приватний ключ зберігається у

захищеній частині серверного середовища, ізольовано від фронтенду, та не передається у мережу. В ідеальних умовах такий ключ слід зберігати у спеціалізованому апаратному модулі (HSM) або щонайменше використовувати тимчасову пам’ять без запису у файл.

На стороні FastAPI реалізовано валідацію вхідних даних — зокрема, перевірку типу та розміру файлу. Це дозволяє запобігти атакам типу “buffer overflow”, “zip bomb” або іншим спробам зловживання інтерфейсом. Усі запити перевіряються на відповідність дозволеним форматам (PDF, DOC, TXT) і лімітам розміру (до 10 МБ), що знижує ризик навантаження або блокування системи через великі файли.

Додатковим захистом виступає механізм логування та fallback, який дозволяє не лише відстежити всі дії користувачів (успішні або невдалі), а й зберегти інформацію у випадку недоступності блокчейну. Це підвищує надійність та дозволяє відновити роботу без втрати транзакцій. Система фіксує усі звернення до API, результати перевірок, помилки підключення та відповіді смарт-контракту у лог-файлі app.log.

З точки зору стійкості до атак, система захищена від таких типових загроз, як:

– replay-атаки — завдяки використанню nonce для кожної транзакції;

– підміна даних — завдяки криптографічному хешуванню;

– неавторизований доступ — через обмеження API та контракту за ролями;

– dos-атаки на контракт — мінімізовані завдяки простій структурі даних (mapping) і відсутності циклів чи складних обчислень.

Окрему увагу слід приділити рівню децентралізації системи. Незважаючи на наявність централізованого API, критично важливі дані (геші) зберігаються у публічному реєстрі — блокчейні Ethereum. Це означає, що навіть у випадку втрати сервера або припинення роботи API, будь-який користувач із доступом до смарт-контракту зможе самостійно виконати перевірку документа напряму через інтерфейс блокчейн-мережі або за допомогою гаманця з Web3-функціональністю.

На рисунку 8 схематично зображено ключові точки контролю безпеки в системі:

починаючи з завантаження файлу користувачем, перевірки формату та гешування, продовжуючи формуванням та підписом транзакції, її передачею у мережу та збереженням у смарт-контракті. Також показано контроль доступу на рівні API, логування та fallback-механізм для тимчасового резервного зберігання хешів.

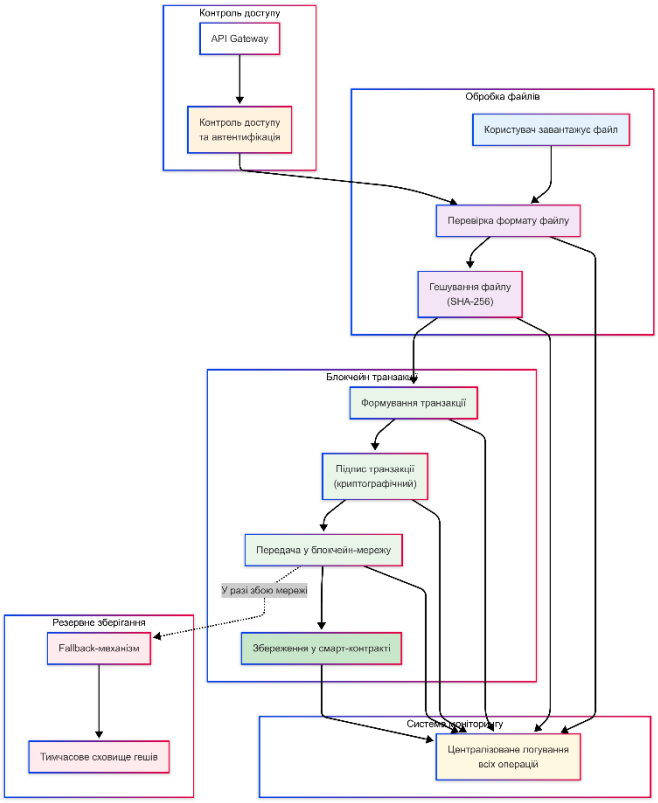

Рисунок 3.3 – Модель безпеки та децентралізації системи верифікації документів.

Такий підхід дозволяє досягти балансу між централізованим керуванням доступом (через API та контракт) та децентралізованим зберіганням даних (блокчейн), що забезпечує гнучкість, відкритість і довготривалу юридичну значущість усіх записів. У поєднанні з відкритим кодом контракту та публічною історією транзакцій, це формує довірене цифрове середовище для перевірки автентичності без залежності від конкретної організації чи адміністратора.

Висновки до розділу 3

У третьому розділі було здійснено повноцінне проєктування децентралізованої системи верифікації документів, з урахуванням функціональних, нефункціональних та безпекових вимог. Визначено основні компоненти архітектури: бекенд на базі FastAPI, смарт-контракт у мережі Ethereum, інтерфейс користувача, бібліотеки Web3 та допоміжні модулі валідації й логування. Обрано гібридний підхід, який поєднує локальну обробку файлів із збереженням криптографічних гешів у блокчейні.

Смарт-контракт, реалізований на Solidity, забезпечує фіксацію гешів документа, їхню перевірку та генерацію подій у мережі. Контракт містить захист від несанкціонованих змін завдяки модифікатору доступу, зберігає дані у структурі mapping і працює незалежно від зовнішніх серверів. Його логіка є простою, передбачуваною та оптимізованою для економного використання газу.

Розроблений API інтерфейс виконує роль посередника між користувачем і блокчейном: він приймає запити, перевіряє дані, обчислює геш-функцію, підписує транзакції та забезпечує інтеграцію із смарт-контрактом. Також реалізовано fallback-механізм для роботи в автономному режимі при втраті доступу до Ethereum, а вся діяльність системи фіксується у лог-файлі для аудиту та моніторингу.

Було приділено окрему увагу питанням інформаційної безпеки: захищено збереження приватного ключа, обмежено доступ до критичних функцій, реалізовано валідацію файлів та обробку винятків. Аналіз децентралізованості рішення показав, що ключові дані зберігаються незалежно від централізованих серверів, а перевірка автентичності може бути здійснена без участі API — безпосередньо через блокчейн.

Таким чином, реалізоване проєктне рішення демонструє комплексний підхід до створення безпечної, стійкої, масштабованої та юридично прозорої системи, яка може бути використана у сфері освіти, права, державного управління або приватного документообігу.

4 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ ВЕРИФІКАЦІЇ ДОКУМЕНТІВ

4.1 Опис програмної реалізації

Програмна реалізація системи верифікації документів охоплює декілька логічно пов’язаних модулів, які працюють разом, утворюючи повноцінну децентралізовану платформу. Архітектура системи побудована на основі принципу розділення відповідальностей: кожен компонент відповідає за окремий шар — обробку запитів, криптографію, взаємодію з блокчейном, логування та контроль доступу. Уся логіка реалізована мовою Python із використанням фреймворку FastAPI для побудови серверної частини та бібліотеки Web3.py для підключення до Ethereum.

Основним виконуваним файлом є main.py, який містить точки входу у систему, визначення маршрутизаторів, обробники запитів (@app.post, @app.get), а також основну логіку обробки даних. При запуску сервер ініціалізує Web3-клієнт, підключається до локального вузла Ganache через RPC-інтерфейс (http://127.0.0.1:7545) та створює об’єкт контракту на основі ABI та адреси.

Другим важливим файлом є ethereum_client.py — модуль взаємодії з блокчейном. Він включає такі функції:

– store_hash_via_contract(hash) — створює транзакцію для збереження хешу документа у смарт-контракті. Функція підписує транзакцію за допомогою приватного ключа, встановлює gas limit, розраховує nonce, і після надсилання чекає на підтвердження (receipt);

– verify_hash_via_contract(hash) — викликає метод контракту у режимі call, тобто без генерації транзакції, і повертає логічне значення, яке вказує на присутність хешу в системі.

У модулі blockchain_manager.py реалізовано fallback-логіку та локальне JSON-сховище. Цей компонент виконує роль резервної бази, яка активується у випадку

втрати зв’язку з Ethereum або збоїв при надсиланні транзакцій. Якщо запис у блокчейн неможливий, геш тимчасово записується у файл local_hash_store.json, з можливістю синхронізації після відновлення з’єднання.

Конфігурація системи зберігається у файлі config.py, який містить такі змінні:

– ETHEREUM_RPC_URL — адреса RPC-серверу;

– PRIVATE_KEY — закритий ключ для підпису транзакцій (у тестовому середовищі використовується ключ із Ganache);

– CONTRACT_ADDRESS — адреса розгорнутого контракту;

– CHAIN_ID — ідентифікатор поточної мережі (Ganache = 1337, Sepolia = 11155111).

Усі ці значення можуть бути змінені для перенесення проєкту у тестову або основну мережу без зміни основного коду.

На рисунку 9 нижче подано структуру проєкту: кожен модуль позначено як окрему логічну одиницю, показано залежності між ними та виклики функцій, які забезпечують цілісну взаємодію. Це включає: маршрути з main.py, класи Web3-з’єднання з ethereum_client.py, локальне сховище з blockchain_manager.py та допоміжні сервіси.

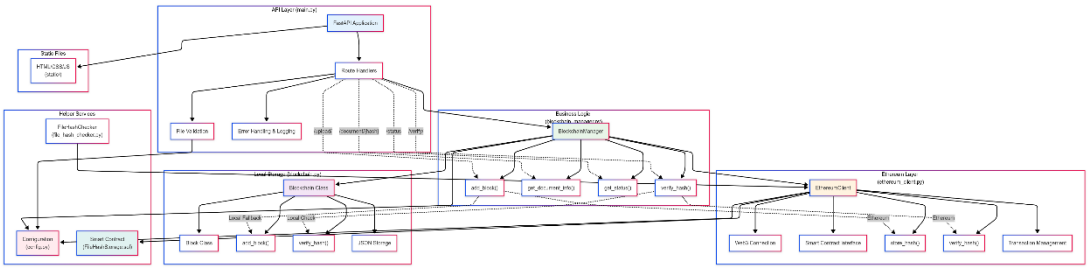

Рисунок 4.1 – Структурна схема реалізації бекенду системи верифікації документів на FastAPI та Web3.py.

У модулі main.py для кожного маршруту (/upload/, /verify/) реалізовано логіку:

- приймання вхідного файлу;
- обчислення SHA-256-гешу через бібліотеку hashlib;
- виклик відповідної функції контракту (або fallback);
- повернення відповіді клієнту з гешем, результатом транзакції або перевірки.

При кожному зверненні результат записується у файл логування app.log, де фіксується тип дії, час, IP користувача (якщо можливо), статус виконання, помилки або підтвердження транзакції.

Уся логіка рознесена по модулях згідно з принципом SRP (Single Responsibility Principle), що спрощує тестування, відлагодження, масштабування та супровід системи. Наприклад, заміна Ethereum на іншу EVM-сумісну мережу вимагатиме лише оновлення параметрів RPC_URL та CHAIN_ID, не торкаючись логіки маршрутів або обробників запитів.

Реалізація смарт-контракту здійснена мовою Solidity у файлі FileHashStorage.sol. Контракт має лаконічну структуру і складається з 34 рядків, у яких реалізовано базову, але повноцінну функціональність зберігання та перевірки гешів. Контракт містить:

- змінну типу mapping(bytes32 => bool) для зберігання унікальних гешів;
- конструктор, який встановлює msg.sender як власника контракту;
- модифікатор onlyOwner, що обмежує виклик storeHash;
- функцію storeHash, яка додає новий геш до реєстру та генерує подію HashUpdated;
- функцію verifyHash, яка повертає булеве значення про наявність гешу;
- події HashUpdated і HashVerified.

Цей контракт було розгорнуто вручну за допомогою середовища Remix IDE із підключенням до Ganache RPC через MetaMask. Після компіляції було збережено ABI (інтерфейс), який надалі використовується у Web3.py для створення об’єкта контракту.

Для фронтенду використано React з бібліотекою Material UI, що забезпечує зручний, адаптивний та сучасний інтерфейс. Користувач може обрати файл, який буде перевірено або записано у систему. У браузері відбувається базова валідація (формат, розмір), після чого файл надсилається через POST-запит до FastAPI-сервера.

У інтерфейсі передбачено кілька ключових компонентів:

- форма завантаження документа;
- індикатор ходу операції (наприклад, збереження, перевірка);
- повідомлення про результат (успішна перевірка або помилка);
- виведення гешу документа для ручної перевірки.

Окремий блок у фронтенді відповідає за відображення історії взаємодії: наприклад, користувач бачить список попередніх операцій, їх результат і дату. Це можливо завдяки доступу до маршруту /blockchain/transactions, де бекенд повертає JSON-масив із журналами транзакцій.

Логіка журналювання реалізована у вигляді звичайного текстового файлу app.log, куди записуються всі запити: тип операції (upload / verify), геш документа, IP-адреса клієнта (якщо доступна), час виконання та статус. Ці записи формуються з допомогою модуля Python logging, конфігурованого на рівні DEBUG.

Додатково, для роботи в режимі offline або при помилці підключення до блокчейну, система використовує файл local_hash_store.json. Цей JSON-файл зберігає список гешів, які не вдалося записати у смарт-контракт. При наступному успішному підключенні ці дані можуть бути оброблені і синхронізовані у фоновому режимі.

Загальна взаємодія між усіма компонентами системи дозволяє досягти повного циклу: від моменту завантаження документа до перевірки його справжності у блокчейні. Уся логіка працює без втручання третьої сторони, повністю автоматизовано, і доступна як через графічний інтерфейс, так і через програмне API.

4.2 Керівництво користувача системи

Інтерфейс користувача системи створено таким чином, щоб забезпечити

максимальну простоту, інтуїтивність та мінімальну кількість дій для виконання основних операцій — збереження гешу документа у блокчейні та перевірки автентичності завантаженого файлу. Система доступна через вебінтерфейс або за допомогою прямого звернення до API.

Після запуску вебдодатка користувач потрапляє на головну сторінку, де відображається форма завантаження документа. Вона включає поле для вибору файлу з локального диску, кнопки «Завантажити» та «Перевірити», а також інформаційне поле, в якому буде виведено результат операції.

На рисунку 10 зображено інтерфейс користувача в момент виконання перевірки документа. Відображено завантажений файл, геш документа, статус перевірки, та повідомлення про успіх або помилку.

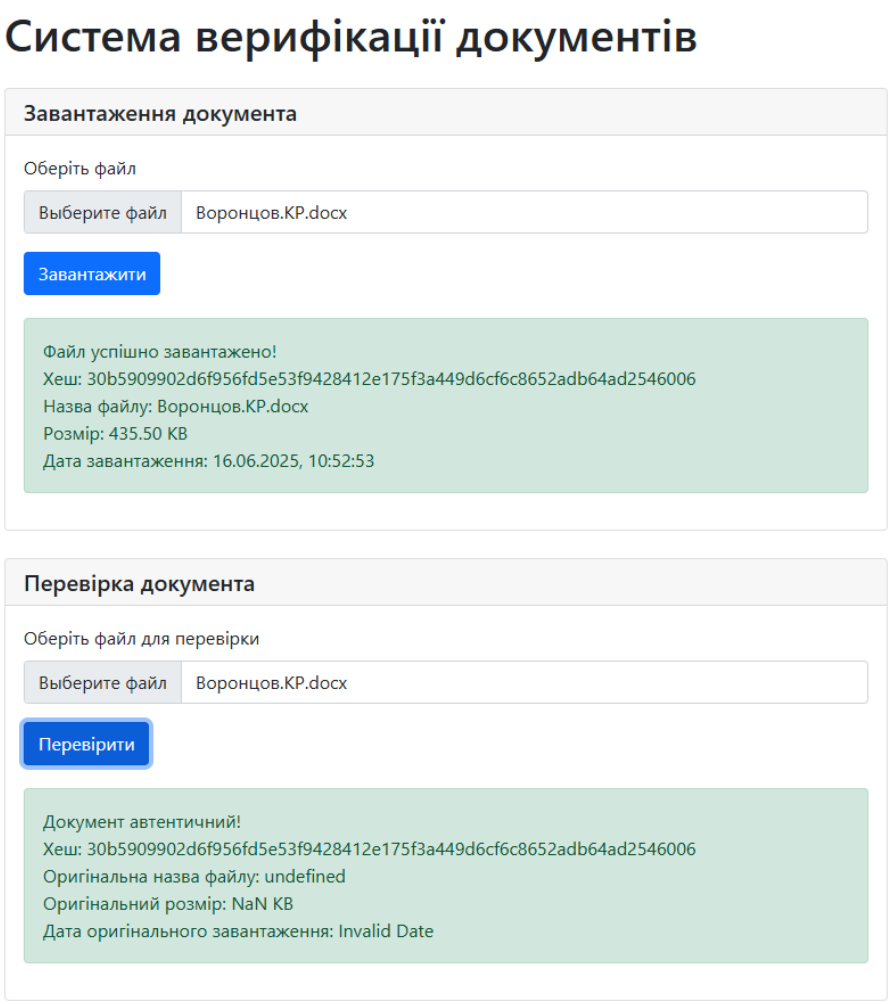

Рисунок 4.2 – Інтерфейс користувача системи верифікації документа через вебраузер.

Щоб записати документ у систему, користувач повинен:

– натиснути кнопку «Оберіть файл» і завантажити PDF, DOC, DOCX або TXT-файл;

– натиснути кнопку «Завантажити» ;

– система обчислить SHA-256-геш файлу;

– геш буде передано на бекенд, де він або буде записаний у смарт-контракт, або у fallback-режим при помилці зв’язку;

– користувач отримає повідомлення з гешем документа, транзакційним хешем та статусом (успішно / помилка).

Щоб перевірити документ, дії аналогічні, але замість кнопки «Завантажити» використовується «Перевірити». У цьому випадку файл не записується, а лише обчислюється його геш і перевіряється, чи є він серед збережених у блокчейні.

Результат перевірки відображається прямо на сторінці:

– якщо геш знайдено — виводиться повідомлення про успішну перевірку;

– якщо геш відсутній — користувач отримає відповідне попередження;

– у разі помилки — повідомлення з її описом (наприклад, перевищено розмір, невірний формат, недоступна мережа тощо).

Інтерфейс також дозволяє скопіювати геш у буфер обміну, щоб зберегти його або надіслати іншій особі, яка зможе самостійно виконати перевірку на іншому пристрої.

Окрім графічного інтерфейсу, система також підтримує роботу через REST API, що є корисним для розробників або інтеграції з іншими платформами. Наприклад, для завантаження документа можна виконати POST-запит на /upload/, передавши файл у тілі запиту (формат multipart/form-data). У відповідь буде отримано геш, транзакційний хеш та статус. Аналогічно, перевірка здійснюється POST-запитом на /verify/, із передачею файлу, що підлягає перевірці.

Для перевірки стану системи можна звернутись до /status — цей маршрут повертає інформацію про доступність смарт-контракту, RPC-з'єднання та інші службові параметри.

Сценарії обробки помилок описані чітко: якщо файл має недопустимий формат, перевищує дозволений розмір, або виникла помилка підключення — користувач отримає відповідне повідомлення. Усі ці помилки також фіксуються в лог-файлі, що дозволяє адміністраторам відстежувати проблеми у роботі системи.

Інтерфейс створено адаптивним, тобто він працює не лише на десктопах, але й на мобільних пристроях. Завдяки використанню Material UI, елементи автоматично підлаштовуються під розмір екрана, а кнопки залишаються зручними для натискання

навіть на смартфонах.

Підсумовуючи, взаємодія користувача із системою не потребує спеціальних технічних знань. Достатньо мати файл, який підлягає захисту або перевірці, доступ до веббраузера, і кілька кліків. Всі інші дії — гешування, підпис, запис у блокчейн, перевірка або обробка транзакцій — виконуються автоматично у фоновому режимі.

4.3 Тестування функціональності та безпеки

Тестування є невід'ємною складовою розробки будь-якого програмного рішення, особливо у випадках, коли йдеться про системи, що працюють із незмінними записами у блокчейні. Метою тестування системи верифікації документів стало підтвердження її функціональної повноти, стійкості до помилок, надійності при роботі з реальними даними та захищеності ключових компонентів від несанкціонованого доступу.

Для організації тестування було сформовано набір реалістичних сценаріїв, які охоплюють увесь цикл роботи користувача — від завантаження документа до перевірки автентичності у блокчейні. Сценарії умовно поділялись на:

- функціональні тести (робота API, перевірка гешів, транзакції) ;
- негативні тести (файли неправильного формату, перевищення розміру) ;
- тести на збої (відсутність підключення до Ethereum) ;
- безпекові тести (моделювання підміни хешу, обхід обмежень доступу) ;
- тестування API проводилося за допомогою **Postman**, де було створено окрему колекцію із запитами на маршрути /upload/, /verify/, /status, /blockchain/transactions. На кожному етапі фіксувались відповіді, час відгуку, структура даних та обробка помилок.

У функціональних тестах було перевірено:

- завантаження файлу дозволеного формату (PDF, DOCX, TXT) — система правильно обчислювала SHA-256-геш, створювала транзакцію, повертала статус success та зберігала хеш у блокчейні;

– повторна спроба завантаження того самого файлу — система розпізнавала існуючий геш, не дублювала запис, але генерувала подію HashUpdated, що свідчило про повторне підтвердження;

– перевірка документа — геш зчитувався коректно, функція verifyHash повертала булеву відповідь (true для існуючого, false — для неіснуючого гешу) ;

– тест на великі файли — система блокувала файли понад 10 МБ, повертала повідомлення про перевищення розміру, що підтверджувало правильну реалізацію валідації;

– тест на заборонені типи файлів (наприклад, .exe, .jpg) — перевірка зупинялась на етапі бекенду, без передачі до смарт-контракту;

– відсутність підключення до Ethereum (вимкнено Ganache) — спрацьовував fallback-механізм, геш зберігався у local_hash_store.json, а користувачу повідомлялось про тимчасовий режим роботи.

На рисунку 11 наведено цикл тестування, який включає: підготовку даних, обчислення гешу, запис у контракт, перевірку результату, моделювання помилок, логування, аналіз подій смарт-контракту.

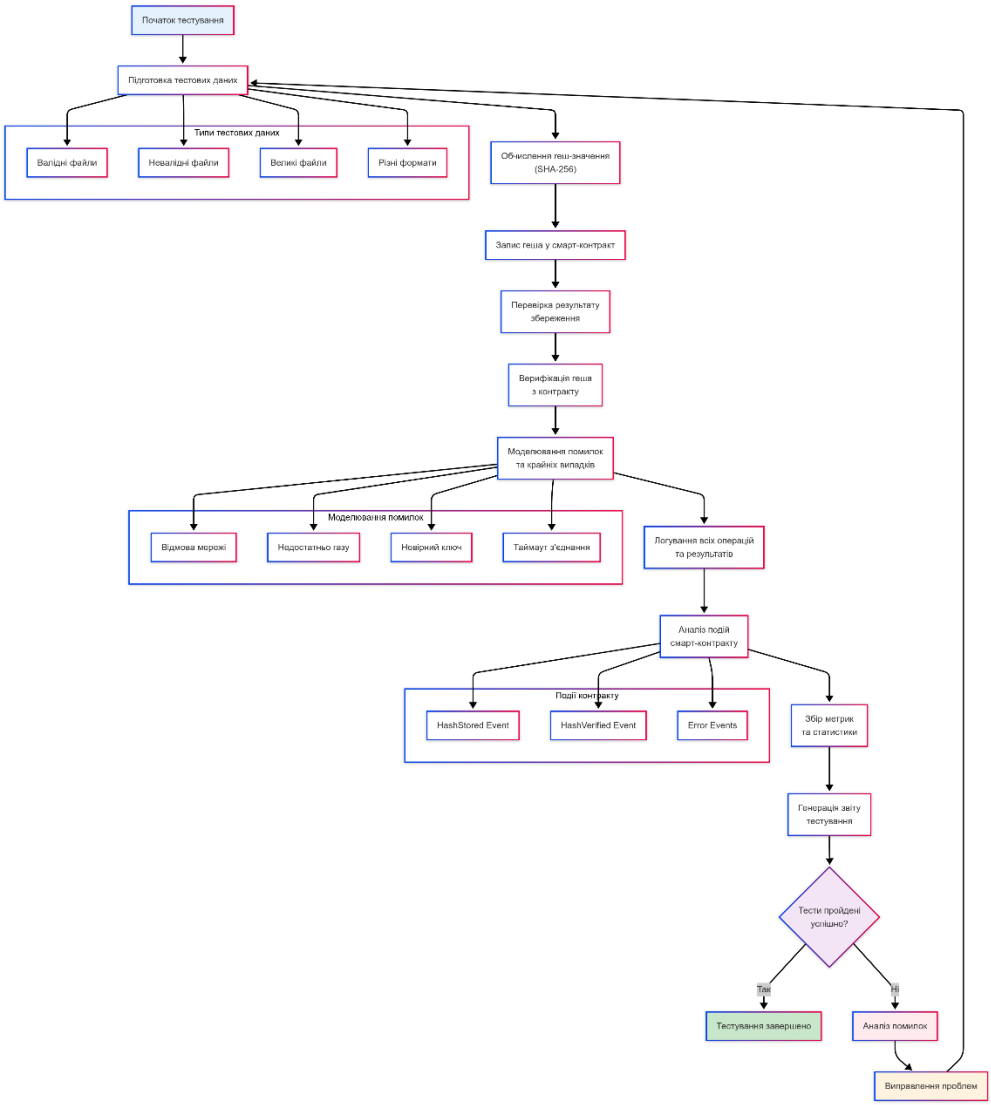

Рисунок 4.3 – Цикл тестування системи: від завантаження документа до перевірки гешу та відстеження подій.

Усі дії фіксувались у логах (app.log), що дозволило верифікувати порядок обробки кожного запиту. Зокрема, у випадку помилок фіксувались причини: відсутність RPC, помилки підпису, винятки при зверненні до контракту, або невірні вхідні параметри.

Було протестовано також і поведінку смарт-контракту при різних запитах. Наприклад, функція storeHash повертала помилку при спробі виклику не від owner, що підтвердило коректність модифікатора доступу. Функція verifyHash, навпаки, була

доступна усім, і правильно повертала значення навіть у тестах з великою кількістю гешів.

Окрему увагу було приділено безпековим тестам, які моделювали потенційні загрози. Одним із таких тестів стала спроба виклику функції збереження гешу з іншого акаунту, не пов’язаного з owner. Контракт очікувано повернув помилку revert, транзакція не була прийнята, а подія не генерувалась. Це свідчить про коректну реалізацію модифікатора onlyOwner та дотримання принципу авторизованого запису.

Було протестовано спробу обходу перевірки формату файлу (наприклад, зміна розширення .exe на .txt). Сервер перевіряв не лише розширення, а й MIME-тип, тому запит блокувався ще до обчислення гешу. Також виконувались спроби надсилання порожнього файлу, який призводив до генерації гешу з фіксованим значенням — система визначала це як дублювання і не виконувала транзакцію.

Під час тестування роботи у fallback-режимі було навмисно перервано з’єднання з Ethereum RPC (вимкнено Ganache), після чого виконано запити на запис і перевірку документа. У разі запису — геш зберігався у локальному файлі local_hash_store.json, а користувачу виводилось повідомлення про тимчасовий режим. Після відновлення RPC з’єднання виконувалась ручна синхронізація даних — усі геші було записано у смарт-контракт, що підтверджує надійність механізму резервного збереження.

Навантажувальне тестування проводилось із метою перевірити продуктивність системи при багаторазових запитах. За допомогою скрипту було надіслано 100 послідовних запитів на завантаження документа протягом 5 хвилин. Система зберігала стабільний час відгуку (≈200–300 мс), транзакції оброблялись у порядку черги, при цьому не було зафіксовано жодної втрати або помилки запису. При тестуванні на перевірку 100 файлів — середній час відповіді становив ≈150 мс.

Усі події смарт-контракту (HashUpdated, HashVerified) коректно фіксувались у логах і відображались на фронтенді. Це дозволило перевірити повноцінну інтеграцію між блокчейном, бекендом і клієнтською частиною.

Оцінюючи результати тестування, можна виділити такі ключові переваги реалізованої системи:

– надійна обробка критичних помилок — система не завершується при втраті з’єднання, а переходить у fallback-режим;

– захист від несанкціонованого доступу — перевірка прав на рівні смарт-контракту;

– прозора логіка валідації та обробки файлів — перевірка MIME, розміру, повторів;

– підтримка масштабованості — збереження стабільної швидкості обробки запитів навіть при підвищеному навантаженні;

– відкритість до аудиту — журнал подій, лог-файл, публічний реєстр блокчейн-записів.

Таким чином, реалізоване рішення продемонструвало високу стабільність, передбачуваність і стійкість до помилок. Навіть у несприятливих умовах (відключення мережі, помилки формату, повторні запити) система функціонувала згідно з очікуваннями, не допускаючи порушення цілісності чи втрати даних.

Висновки до розділу 4

У четвертому розділі було детально описано процес програмної реалізації, використані інструменти, архітектуру бекенду, логіку смарт-контракту, API-інтерфейси та клієнтське середовище. Було показано, що обрана технологічна зв'язка (FastAPI + Web3.py + Solidity + React) забезпечує ефективну реалізацію системи зберігання та перевірки цифрових гешів документів із використанням блокчейн-мережі Ethereum.

Було реалізовано повноцінну серверну логіку з обробкою запитів, криптографічним гешуванням файлів, підписуванням транзакцій і фіксацією результатів у блокчейні. Смарт-контракт спроєктовано з урахуванням принципів безпеки, простоти та ефективності — він дозволяє зберігати та перевіряти геші,

генерує події для зворотного зв'язку і містить механізм обмеження доступу до запису.

Користувацький інтерфейс дозволяє виконувати основні операції за кілька кроків — без необхідності технічних знань або встановлення додаткового програмного забезпечення. Реалізований fallback-механізм гарантує безперервність роботи навіть у випадку втрати з'єднання з Ethereum.

У ході тестування система підтвердила свою стабільність, передбачуваність та захищеність. Було успішно протестовано основні сценарії використання, обробку помилок, відмовостійкість і продуктивність. Система коректно обробляє некоректні запити, фіксує всі дії у логах, не допускає дублювання даних і забезпечує прозору інтеграцію з блокчейном.

Таким чином, реалізована система успішно виконує всі поставлені функції: забезпечує фіксацію незмінного цифрового відбитка документа, дозволяє незалежну перевірку його автентичності та гарантує довготривале зберігання цієї інформації у децентралізованій формі. Результати реалізації доводять практичну доцільність використання блокчейн-технологій у задачах захисту цифрових документів.

ВИСНОВКИ

У межах кваліфікаційної роботи було досліджено та реалізовано децентралізовану систему верифікації документів із використанням блокчейн-технологій, яка дозволяє надійно зберігати цифрові геші документів та перевіряти їхню автентичність без залучення довірених третіх сторін. Основною метою було створення універсального інструменту, що поєднує захищеність, прозорість, децентралізованість і зручність для кінцевого користувача.

У процесі роботи було проведено аналіз сучасних проблем цифрової верифікації, виявлено недоліки централізованих рішень, таких як вразливість до атак, втрата даних, технічна залежність від адміністраторів і складність довготривалої перевірки. Обґрунтовано доцільність застосування блокчейну для зберігання цифрових відбитків (гешів), що забезпечує незмінність, відкритість і незалежність від посередників.

Проєкт реалізовано як Full-Stack систему на основі FastAPI (Python), Web3.py, Solidity (Ethereum) та React. Основна логіка включає обробку запитів, валідацію документів, обчислення SHA-256-гешу, створення та підпис транзакцій, виклики смарт-контракту, обробку подій, логування та fallback-механізм на випадок втрати з’єднання. Уся інформація про дії користувача зберігається у лог-файлах, а транзакції — у блокчейні.

Смарт-контракт було спроєктовано таким чином, щоб реалізувати основні операції — запис (storeHash) та перевірку (verifyHash) — з додатковим захистом доступу (onlyOwner) та можливістю генерації подій (HashUpdated, HashVerified). Всі ключові події логуються й доступні для незалежного аудиту.

Тестування показало стабільність роботи системи, її захищеність від типових загроз (підміна, повторна реєстрація, несанкціонований доступ), стійкість до технічних збоїв (через fallback), а також масштабованість при високих навантаженнях.

Практична цінність системи полягає в її можливості бути інтегрованою в

освітні, юридичні, адміністративні та корпоративні середовища. Вона забезпечує механізм публічного підтвердження існування документа без зберігання його вмісту, що особливо важливо для конфіденційних або юридично значущих файлів.

У перспективі можлива інтеграція системи з електронними підписами, впровадження NFT або SBT для представлення цифрових сертифікатів, розширення прав доступу через DAO-механізми, а також адаптація для використання з мобільних платформ. Запропоноване рішення може стати базою для створення національного або корпоративного реєстру підтверджень, які не потребують довіри до централізованого сервісу.

Таким чином, результати дослідження підтверджують, що блокчейн-технології здатні ефективно вирішувати проблему верифікації цифрових документів, забезпечуючи юридичну, технічну та логічну надійність в умовах цифрового суспільства.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Nakamoto S. Bitcoin: A Peer-to-Peer Electronic Cash System [Електронний ресурс]. – 2008. – Режим доступу: https://bitcoin.org/bitcoin.pdf
2. Buterin V. A Next-Generation Smart Contract and Decentralized Application Platform [Електронний ресурс]. – 2013. – Режим доступу: https://ethereum.org/en/whitepaper/
3. Wood G. Ethereum: A Secure Decentralised Generalised Transaction Ledger [Yellow Paper] – 2021. – Режим доступу: https://ethereum.github.io/yellowpaper/paper.pdf
4. FastAPI Documentation [Електронний ресурс] – Режим доступу: https://fastapi.tiangolo.com
5. Web3.py Documentation [Електронний ресурс] – Режим доступу: https://web3py.readthedocs.io
6. Solidity Language Documentation [Електронний ресурс] – Режим доступу: https://docs.soliditylang.org
7. MetaMask Docs. Ethereum Wallet Browser Extension [Електронний ресурс] – Режим доступу: https://docs.metamask.io
8. OpenCerts. An open standard for blockchain-based education credentials [Електронний ресурс] – Режим доступу: https://opencerts.io
9. Blockcerts. Blockchain Credentials by MIT [Електронний ресурс] – Режим доступу: https://www.blockcerts.org
10. IBM Blockchain Platform Documentation [Електронний ресурс] – Режим доступу: https://www.ibm.com/docs/en/blockchain
11. European Blockchain Services Infrastructure (EBSI). – Official EU Portal – [Електронний ресурс]. – Режим доступу: https://ec.europa.eu/digital-strategy/our-policies/european-blockchain-services-infrastructure
12. Ganache Documentation – Truffle Suite [Електронний ресурс] – Режим доступу: https://trufflesuite.com/ganache

13. RFC 6234 – US Secure Hash Algorithms (SHA and HMAC) [Електронний ресурс] – Режим доступу: https://datatracker.ietf.org/doc/html/rfc6234
14. Копусова О. В. Блокчейн: основи, можливості, проблеми / О. В. Копусова // Інформаційні технології в освіті. – 2020. – № 41. – С. 120–126.
15. Плотніков Д. В. Застосування технології блокчейн у правовій практиці / Д. В. Плотніков // Юридичний вісник України. – 2021. – № 7. – С. 35–40.
16. Mougayar W. The Business Blockchain: Promise, Practice, and the Application of the Next Internet Technology. – Wiley, 2016. – 208 с.
17. Swan M. Blockchain: Blueprint for a New Economy. – O'Reilly Media, 2015. – 152 с.
18. Christidis K., Devetsikiotis M. Blockchains and Smart Contracts for the Internet of Things // *IEEE Access*. – 2016. – Vol. 4. – P. 2292–2303.
19. Bashir I. *Mastering Blockchain: Unlocking the Power of Cryptocurrencies, Smart Contracts, and Decentralized Applications*. – 3rd ed. – Packt Publishing, 2020.
20. Zyskind G., Nathan O., Pentland A. Decentralizing Privacy: Using Blockchain to Protect Personal Data // *IEEE Security & Privacy Workshops*. – 2015. – P. 180–184.
21. Mavridou A., Laszka A. Tool Demonstration: FSolidM for Designing Secure Ethereum Smart Contracts // *ESORICS*. – 2018.
22. Parizi R.M., Dehghantanha A., Choo K.K.R. Blockchain for IoT and Cyberphysical Systems // *Springer*, 2020.
23. ISO/TC 307 – Blockchain and Distributed Ledger Technologies [Електронний ресурс] – Режим доступу: https://www.iso.org/committee/6266604.html
24. Ethereum Improvement Proposals (EIPs) [Електронний ресурс] – Режим доступу: https://eips.ethereum.org
25. Dannen C. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. – Apress, 2017.
26. Atzei N., Bartoletti M., Cimoli T. A Survey of Attacks on Ethereum Smart Contracts

// *International Conference on Principles of Security and Trust (POST)*. – 2017.

27. Kosba A., Miller A., Shi E., Wen Z., Papamanthou C. Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts // *IEEE S&P*. – 2016.
28. Szmigiera M. Number of blockchain wallet users worldwide 2016–2024 // *Statista*. – Режим доступу: https://www.statista.com/statistics/647374/worldwide-blockchain-wallet-users
29. Nakamura R., Miura H., Odagiri Y., Sakurai K. How to Verify the Integrity of Digital Certificates Using Blockchain // *Computers & Security*. – 2018.
30. Wüst K., Gervais A. Do you need a Blockchain? // *IACR Cryptology ePrint Archive*, 2017. – No. 2017/375.

ДОДАТОК А

Лістинг коду

Main.py

```
from fastapi import FastAPI, UploadFile, File, HTTPException
from fastapi.responses import JSONResponse, FileResponse
from fastapi.staticfiles import StaticFiles
from fastapi.middleware.cors import CORSMiddleware
import hashlib
import logging
from blockchain_manager import BlockchainManager
import os
from datetime import datetime
from config import MAX_FILE_SIZE, ALLOWED_EXTENSIONS

logging.basicConfig(
    filename='app.log',
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)

app = FastAPI(
    title="Система верификации документов на блокчейне",
        description="API для загрузки и верификации документов с использованием блокчейна",
    version="2.0.0"
)
```

```
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

if not os.path.exists("static"):
    os.makedirs("static")
    logging.info("Створено директорію static")

app.mount("/static", StaticFiles(directory="static"), name="static")

try:
        blockchain_manager = BlockchainManager(use_ethereum=True, fallback_to_local=True)
    logging.info("Блокчейн менеджер успешно инициализирован")
except Exception as e:
    logging.error(f"Ошибка инициализации блокчейн менеджера: {str(e)}")
        blockchain_manager = BlockchainManager(use_ethereum=False, fallback_to_local=True)

def validate_file(file: UploadFile):
    file_size = 0
    contents = b""
    while chunk := file.file.read(8192):
```

```
            file_size += len(chunk)
            contents += chunk
            if file_size > MAX_FILE_SIZE:
                raise HTTPException(status_code=400, detail="Файл занадто великий")

        file_ext = os.path.splitext(file.filename)[1].lower()
        if file_ext not in ALLOWED_EXTENSIONS:
            raise HTTPException(status_code=400, detail="Недопустимий тип файлу")

        return file_size, contents

    @app.get("/")
    async def read_root():
        try:
            return FileResponse("static/index.html")
        except FileNotFoundError:
            logging.error("Файл index.html не знайдено")
            return JSONResponse(content={
                "message": "Система верификации документов на блокчейне",
                "status": "working",
                "api_docs": "/docs",
                "endpoints": {
                    "status": "/status",
                    "upload": "/upload/",
                    "verify": "/verify/",
```

```
            "transactions": "/blockchain/transactions"
        }
    })

@app.get("/favicon.ico")
async def favicon():
    try:
        return FileResponse("static/favicon.ico")
    except FileNotFoundError:
        raise HTTPException(status_code=404, detail="Favicon не знайдено")

@app.get("/status")
async def get_blockchain_status():
    """Получает статус подключений к блокчейну"""
    try:
        status = blockchain_manager.get_status()
        return JSONResponse(content=status)
    except Exception as e:
        logging.error(f"Ошибка получения статуса: {str(e)}")
        raise HTTPException(status_code=500, detail=str(e))

@app.get("/blockchain/transactions")
async def get_recent_transactions():
    """Получает последние транзакции из блокчейна для проверки"""
    try:
        if blockchain_manager.ethereum_client and
blockchain_manager.ethereum_client.w3.is_connected():
```

```
        w3 = blockchain_manager.ethereum_client.w3

        latest_block_number = w3.eth.block_number
        transactions = []

        for i in range(max(0, latest_block_number - 9), latest_block_number +
1):
            block = w3.eth.get_block(i, full_transactions=True)

            for tx in block.transactions:
                if tx.to and blockchain_manager.ethereum_client.contract_address:
                                                        if      tx.to.lower()      ==
blockchain_manager.ethereum_client.contract_address.lower():
                        transactions.append({
                            "hash": tx.hash.hex(),
                            "block_number": block.number,
                            "from": tx['from'],
                            "to": tx.to,
                            "gas_used": tx.gas,
                                                                "timestamp":
datetime.fromtimestamp(block.timestamp).isoformat(),
                            "is_contract_interaction": True
                        })

        return JSONResponse(content={
            "latest_block": latest_block_number,
                                                    "contract_address":
```

```
blockchain_manager.ethereum_client.contract_address,
                    "recent_transactions": transactions[-10:],
                    "total_found": len(transactions)
                })
            else:
                return JSONResponse(content={
                    "error": "Ethereum клиент не подключен",
                    "using_local_storage": True
                })
        except Exception as e:
            logging.error(f"Ошибка получения транзакций: {str(e)}")
            raise HTTPException(status_code=500, detail=str(e))

    @app.post("/upload/")
    async def upload_document(file: UploadFile = File(...)):
        try:
            file_size, contents = validate_file(file)

            file_hash = hashlib.sha256(contents).hexdigest()

            metadata = {
                'filename': file.filename,
                'size': file_size,
                'upload_date': datetime.now().isoformat(),
                'content_type': file.content_type
            }
```

```
            print(f"\n🔥 ЗАГРУЗКА ФАЙЛА В БЛОКЧЕЙН:")
            print(f"📁 Файл: {file.filename}")
            print(f"🔗 Хеш: {file_hash}")
            print(f"📊 Размер: {file_size} байт")

            results = blockchain_manager.add_block(file_hash, metadata)

            ethereum_success = results.get('ethereum', {}).get('success', False)
            local_success = results.get('local', {}).get('success', False)

            if ethereum_success:
                tx_hash = results['ethereum']['transaction_hash']
                print(f" Сохр!")
                print(f" Транзакция: {tx_hash}")
                print(f" Газ: {results['ethereum']['gas_used']}")
                print(f" Блок: {results['ethereum']['block_number']}")

            if local_success:
                print(f"Резерв")

              logging.info(f"Файл завантажено: {file.filename}, хеш: {file_hash},
Ethereum: {ethereum_success}")

            return JSONResponse(content={
                "hash": file_hash,
                "metadata": metadata,
                "blockchain_results": results,
```

```
                "success": any(r.get('success', False) for r in results.values()),
                "saved_to_ethereum": ethereum_success,
                "transaction_hash": results.get('ethereum', {}).get('transaction_hash') if
ethereum_success else None
            })
        except Exception as e:
            logging.error(f"Помилка при завантаженні файлу: {str(e)}")
            raise HTTPException(status_code=500, detail=str(e))

    @app.post("/verify/")
    async def verify_document(file: UploadFile = File(...)):
        try:
            file_size, contents = validate_file(file)

            new_hash = hashlib.sha256(contents).hexdigest()

            print(f"\n Перевірка:")
            print(f" Файл: {file.filename}")
            print(f" Хеш: {new_hash}")

            verification_result = blockchain_manager.verify_hash(new_hash)

            original_metadata = blockchain_manager.get_document_info(new_hash)

                ethereum_verified = verification_result['details'].get('ethereum',
{}).get('verified', False)
            local_verified = verification_result['details'].get('local', {}).get('verified',
```

```
False)

            if ethereum_verified:
                print(f"файл знайд")
            elif local_verified:
                print(f" Файл найден в локальном хранилище")
            else:
                print(f" Файл НЕ знайдено")

                logging.info(f"Перевірка документа: {new_hash}, результат: {verification_result['is_verified']}")

            return JSONResponse(content={
                "is_valid": verification_result['is_verified'],
                "hash": new_hash,
                "verification_details": verification_result['details'],
                "original_metadata": original_metadata,
                "verified_in_ethereum": ethereum_verified
            })
        except Exception as e:
            logging.error(f"Помилка при перевірці документа: {str(e)}")
            raise HTTPException(status_code=500, detail=str(e))

    @app.get("/document/{file_hash}")
    async def get_document_info(file_hash: str):
        try:
            document_info = blockchain_manager.get_document_info(file_hash)
```

```
            if not document_info:
                raise HTTPException(status_code=404, detail="Документ не знайдено")

            return JSONResponse(content=document_info)
        except HTTPException:
            raise
        except Exception as e:
            logging.error(f"Помилка при отриманні інформації про документ: {str(e)}")
            raise HTTPException(status_code=500, detail=str(e))

    @app.get("/verify/{file_hash}")
    async def verify_by_hash(file_hash: str):
        """Проверяет хеш файла по его значению"""
        try:
            verification_result = blockchain_manager.verify_hash(file_hash)

            return JSONResponse(content={
                "hash": file_hash,
                "is_verified": verification_result['is_verified'],
                "details": verification_result['details']
            })
        except Exception as e:
            logging.error(f"Ошибка проверки хеша {file_hash}: {str(e)}")
            raise HTTPException(status_code=500, detail=str(e))
```

blockchain.py

```
import time
import hashlib
import json
import os
from datetime import datetime

class Block:
    def __init__(self, data, previous_hash, metadata=None):
        self.timestamp = time.time()
        self.data = data
        self.previous_hash = previous_hash
        self.metadata = metadata or {}
        self.hash = self.calculate_hash()

    def calculate_hash(self):
                return hashlib.sha256((str(self.timestamp) + self.data + self.previous_hash).encode()).hexdigest()

    def to_dict(self):
        return {
            'timestamp': self.timestamp,
            'data': self.data,
            'previous_hash': self.previous_hash,
            'hash': self.hash,
            'metadata': self.metadata
        }
```

```
    @classmethod
    def from_dict(cls, data):
        block = cls(data['data'], data['previous_hash'], data['metadata'])
        block.timestamp = data['timestamp']
        block.hash = data['hash']
        return block

class Blockchain:
    def __init__(self, storage_file='blockchain.json'):
        self.storage_file = storage_file
        self.chain = self.load_chain() or [self.create_genesis_block()]

    def create_genesis_block(self):
        return Block("Genesis Block", "0")

    def add_block(self, data, metadata=None):
        previous_block = self.chain[-1]
        new_block = Block(data, previous_block.hash, metadata)
        self.chain.append(new_block)
        self.save_chain()
        return new_block

    def verify_hash(self, file_hash):
        for block in self.chain:
            if block.data == file_hash:
                return True
```

```
        return False

    def save_chain(self):
        with open(self.storage_file, 'w') as f:
            json.dump([block.to_dict() for block in self.chain], f, indent=2)

    def load_chain(self):
        if not os.path.exists(self.storage_file):
            return None
        try:
            with open(self.storage_file, 'r') as f:
                data = json.load(f)
                return [Block.from_dict(block_data) for block_data in data]
        except (json.JSONDecodeError, FileNotFoundError):
            return None
```

blockchain_manager.py

```
import logging
from ethereum_client import EthereumClient
from blockchain import Blockchain
from config import CONTRACT_ADDRESS, PRIVATE_KEY, CURRENT_NETWORK
import os

class BlockchainManager:
    def __init__(self, use_ethereum=True, fallback_to_local=True):
        """
            Менеджер блокчейна с поддержкой Ethereum и локального
```

```
хранилища
        """
        self.use_ethereum = use_ethereum
        self.fallback_to_local = fallback_to_local
        self.ethereum_client = None
        self.local_blockchain = None

        if use_ethereum:
            try:
                self.ethereum_client = EthereumClient(
                    contract_address=CONTRACT_ADDRESS,
                    private_key=PRIVATE_KEY,
                    network=CURRENT_NETWORK
                )
                logging.info("Ethereum клиент успешно инициализирован")
            except Exception as e:
                    logging.error(f"Ошибка инициализации Ethereum клиента:
{str(e)}")
                if fallback_to_local:
                    logging.info("Переключение на локальное хранилище")
                    self.use_ethereum = False
                else:
                    raise e

        if not self.use_ethereum or fallback_to_local:
            self.local_blockchain = Blockchain()
            logging.info("Локальный блокчейн инициализирован")
```

```
        def add_block(self, file_hash, metadata=None):
            """Добавляет блок с хешем файла"""
            results = {}

            if self.use_ethereum and self.ethereum_client:
                try:
                    receipt = self.ethereum_client.store_hash(file_hash, metadata)
                    results['ethereum'] = {
                        'success': True,
                        'transaction_hash': receipt['transactionHash'].hex(),
                        'block_number': receipt['blockNumber'],
                        'gas_used': receipt['gasUsed']
                    }
                                        logging.info(f"Хеш сохранен в Ethereum: {receipt['transactionHash'].hex()}")
                except Exception as e:
                    results['ethereum'] = {
                        'success': False,
                        'error': str(e)
                    }
                    logging.error(f"Ошибка сохранения в Ethereum: {str(e)}")

            if self.local_blockchain:
                try:
                    block = self.local_blockchain.add_block(file_hash, metadata)
                    results['local'] = {
```

```
                'success': True,
                'block_hash': block.hash,
                'timestamp': block.timestamp
            }
            logging.info(f"Хеш сохранен локально: {block.hash}")
        except Exception as e:
            results['local'] = {
                'success': False,
                'error': str(e)
            }
            logging.error(f"Ошибка сохранения локально: {str(e)}")

    return results

def verify_hash(self, file_hash):
    """Проверяет хеш файла в блокчейне"""
    verification_results = {}

    if self.use_ethereum and self.ethereum_client:
        try:
            ethereum_result = self.ethereum_client.verify_hash(file_hash)
            verification_results['ethereum'] = {
                'verified': ethereum_result,
                'source': 'ethereum'
            }
            logging.info(f"Проверка в Ethereum: {ethereum_result}")
        except Exception as e:
```

```
            verification_results['ethereum'] = {
                'verified': False,
                'error': str(e),
                'source': 'ethereum'
            }
            logging.error(f"Ошибка проверки в Ethereum: {str(e)}")

    if self.local_blockchain:
        try:
            local_result = self.local_blockchain.verify_hash(file_hash)
            verification_results['local'] = {
                'verified': local_result,
                'source': 'local'
            }
            logging.info(f"Проверка локально: {local_result}")
        except Exception as e:
            verification_results['local'] = {
                'verified': False,
                'error': str(e),
                'source': 'local'
            }
            logging.error(f"Ошибка локальной проверки: {str(e)}")

    is_verified = any(
        result.get('verified', False)
        for result in verification_results.values()
    )
```

```
    return {
        'is_verified': is_verified,
        'details': verification_results
    }

def get_document_info(self, file_hash):
    """Получает информацию о документе"""
    if self.local_blockchain:
        for block in self.local_blockchain.chain:
            if block.data == file_hash:
                return {
                    'hash': file_hash,
                    'metadata': block.metadata,
                    'timestamp': block.timestamp,
                    'source': 'local'
                }

    if self.use_ethereum and self.ethereum_client:
        try:
            stored_hash = self.ethereum_client.get_stored_hash()
            if stored_hash == file_hash:
                return {
                    'hash': file_hash,
                    'source': 'ethereum',
                    'verified': True
                }
```

```
            except Exception as e:
                logging.error(f"Ошибка получения данных из Ethereum: {str(e)}")

        return None

    def get_status(self):
        """Получает статус подключений"""
        status = {
            'ethereum': {
                'connected': False,
                'network': CURRENT_NETWORK,
                'account': None,
                'balance': None
            },
            'local': {
                'available': self.local_blockchain is not None,
                'blocks_count': 0
            }
        }

        if self.ethereum_client:
            try:
                status['ethereum']['connected'] = self.ethereum_client.w3.is_connected()
                status['ethereum']['account'] = self.ethereum_client.account.address if self.ethereum_client.account else None
                status['ethereum']['balance'] =
```

```
self.ethereum_client.get_account_balance()
                except Exception as e:
                    logging.error(f"Ошибка получения статуса Ethereum: {str(e)}")

            if self.local_blockchain:
                status['local']['blocks_count'] = len(self.local_blockchain.chain)

            return status
```

file_hash_checker.py

```
import hashlib
from ethereum_client import EthereumClient
import os

class FileHashChecker:
    def __init__(self, contract_address, private_key=None):
        self.ethereum_client = EthereumClient(contract_address, private_key)

    def calculate_file_hash(self, file_path):
        """Обчислює SHA-256 хеш файлу"""
        sha256_hash = hashlib.sha256()

        with open(file_path, "rb") as f:
            for byte_block in iter(lambda: f.read(4096), b""):
                sha256_hash.update(byte_block)

        return sha256_hash.hexdigest()
```

```
    def verify_file_hash(self, file_path):
        """Перевіряє хеш файлу з збереженим у контракті"""
        current_hash = self.calculate_file_hash(file_path)
        stored_hash = self.ethereum_client.get_stored_hash()

        if current_hash == stored_hash:
            print("✅ Хеш файлу збігається з збереженим у блокчейні")
            return True
        else:
            print("❌ Хеш файлу відрізняється від збереженого у блокчейні")
            print(f"Поточний хеш: {current_hash}")
            print(f"Збережений хеш: {stored_hash}")
            return False

    def store_file_hash(self, file_path):
        """Зберігає хеш файлу у смарт-контракт"""
        file_hash = self.calculate_file_hash(file_path)
        try:
            self.ethereum_client.store_hash(file_hash)
            print(f"✅ Хеш файлу успішно збережено у блокчейні: {file_hash}")
            return True
        except Exception as e:
            print(f"❌ Помилка при збереженні хешу: {str(e)}")
            return False

if __name__ == "__main__":
    CONTRACT_ADDRESS = os.getenv("CONTRACT_ADDRESS", "0x...")
```

```
PRIVATE_KEY = os.getenv("PRIVATE_KEY")

checker = FileHashChecker(CONTRACT_ADDRESS, PRIVATE_KEY)

file_path = "example.txt"
checker.verify_file_hash(file_path)
```