ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Інформаційна система інтелектуальної генерації та верифікації технічної документації на основі моделей LLM»

на здобуття освітнього ступеня бакалавра

зі спеціальності 126 Інформаційні системи та технології

освітньо-професійної програми Інформаційні системи та технології

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Анастасія ШАБЛЯ

(підпис) *(ім'я, ПРІЗВИЩЕ здобувача)*

Виконав: здобувач вищої освіти група ІСД-41

Анастасія ШАБЛЯ

(ім'я, ПРІЗВИЩЕ)

Керівник Валентина ДАНИЛЬЧЕНКО

PhD,доцент *(ім'я, ПРІЗВИЩЕ)*

Рецензент:

(ім'я, ПРІЗВИЩЕ)

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра Інформаційних систем та технологій
Ступінь вищої освіти «Бакалавр»
Спеціальність 126 Інформаційні системи та технології
Напрям підготовки Інформаційні системи та технології

ЗАТВЕРДЖУЮ
Завідувач кафедри Інформаційних систем та технологій
К.П. СТОРЧАК

"____" _____2026 року

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

Шабля Анастасія Вікторівна
(прізвище, ім'я, по батькові)

1.Тема роботи: Інформаційна система інтелектуальної генерації та верифікації технічної документації на основі моделей LLM
Керівник кваліфікаційної роботи Данильченко В.М. PhD, доцент,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від "20" лютого 2026 р. №51
2. Строк подання кваліфікаційної роботи « 1 червня » 2026 року
3. Вихідні дані до роботи: Технічні специфікації та логи телеметрії IoT-пристроїв у форматах JSON/YAML; Відкриті наукові дослідження із застосування моделей LLM та архітектури RAG в інженерному документуванні; Офіційна документація моделі Llama 3.3, інфраструктури Groq API та веб-фреймворку Flask.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1.Дослідження проблематики супроводу IoT-рішень та методів автоматизації інженерного документування.
2. Проєктування інформаційної системи інтелектуальної генерації та верифікації на базі моделей LLM.
3. Практична реалізація та тестування модулів програмного комплексу.
4. Експериментальна оцінка точності системи та обґрунтування її фінансово-економічної ефективності

5. Перелік графічного матеріалу

1. Презентація

2. Слайди

6. Дата видачі завдання «20» лютого 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково-технічної літератури	20.02.26-10.03.26	Виконано
2	Дослідження архітектурних обмежень LLM та методів автоматизації документування IoT	11.03.26–31.03.26	Виконано
3	Порівняльний аналіз превентивного RAG-підходу та агентного AI-аудиту	01.04.26–15.04.26	Виконано
4	Формулювання технічного завдання, функціональних та нефункціональних вимог до системи	16.04.26–30.04.26	Виконано
5	Програмна реалізація та тестування модулів генерації й ШІ-верифікації	01.05.26–15.05.26	Виконано
6	Програмна реалізація та тестування модулів генерації й ШІ-верифікації	16.05.26–22.05.26	Виконано
7	Розробка демонстраційних матеріалів	23.05.26–28.05.26	Виконано
8	Подання роботи в деканат	01.06.26	Виконано

Студент _____
(підпис) (прізвище та ініціали)

Керівник роботи _____
підпис) (прізвище та ініціали)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 88 стор., 15 рис., 10 табл., 34 джерел.

Мета роботи – підвищення ефективності, точності та швидкості розробки і перевірки технічної документації для IoT-систем шляхом розробки автоматизованої інформаційної системи на основі архітектури великих мовних моделей.

Об'єкт дослідження – процес формування, верифікації та супроводження технічної документації для гетерогенних рішень Інтернету речей.

Предмет дослідження – методи, алгоритми та програмні інструменти автоматизованої генерації та перевірки технічних текстів із застосуванням архітектури великих мовних моделей.

Короткий зміст роботи: Проаналізовано специфіку супроводу технічної документації в галузі IoT та фундаментальні обмеження LLM. Спроєктовано багатошарову модульну архітектуру інформаційної системи (Presentation, Service, DAL) та змодельовано алгоритм інтелектуального аудиту у вигляді детермінованого скінченного автомата. Програмно реалізовано комплекс мовою Python з використанням веб-фреймворку Flask, моделі Llama 3.3 та високопродуктивної інфраструктури Groq LPU. Проведено експериментальну оцінку точності за метриками BERTScore та Cosine Similarity, а також обґрунтовано фінансово-економічну ефективність впровадження розробленої системи.

КЛЮЧОВІ СЛОВА: ІНТЕРНЕТ РЕЧЕЙ, ВЕЛИКІ МОВНІ МОДЕЛІ, АВТОМАТИЗАЦІЯ ДОКУМЕНТАЦІЇ, ШІ-АУДИТ, СКІНЧЕННИЙ АВТОМАТ, RAG-ПІДХІД, FLASK, GROQ LPU, BERTSCORE, COSINE SIMILARITY.

ЗМІСТ

ВСТУП

Актуальність теми. Стрімке розгортання Інтернету речей (IoT) в Industry 4.0 та «розумних» містах обумовлює глибоку гетерогенність мереж, які об'єднують різнорідні сенсори, контролери та протоколи. У таких умовах технічна документація стає критичним елементом контуру безпеки, де будь-яка помилка конфігурації підвищує ризик аварій. Традиційні ручні методи документування є неефективними у динамічному IoT-середовищі, а впровадження великих мовних моделей (LLM) для автоматизації цих процесів вимагає розробки надійних методів верифікації згенерованого тексту для уникнення фактологічних галюцинацій.

Мета дослідження — підвищення ефективності, точності та швидкості розробки і перевірки технічної документації для IoT-систем шляхом розробки автоматизованої інформаційної системи на основі архітектури великих мовних моделей.

Для досягнення поставленої мети необхідно вирішити такі завдання:

1. Проаналізувати сучасний стан, специфіку та проблеми документування IoT-рішень і наявні підходи до застосування LLM в автоматизації інженерних текстів.
2. Дослідити архітектуру, методи промптингу та інтеграції великих мовних моделей для роботи з технічною документацією.
3. Спроєктувати архітектуру інформаційної системи автоматизованої генерації та верифікації документації для IoT.
4. Реалізувати програмний модуль генерації та валідації специфікацій мовою Python із використанням сучасних фреймворків та API.
5. Провести функціональне тестування системи, оцінити точність верифікації текстів та проаналізувати її ефективність.

Об’єкт дослідження — процес формування, верифікації та супроводження технічної документації для гетерогенних рішень Інтернету речей.

Предмет дослідження — методи, алгоритми та програмні інструменти автоматизованої генерації та перевірки технічних текстів із застосуванням архітектури великих мовних моделей.

Методи дослідження. У роботі використано методи системного аналізу (для дослідження предметної області), теорію конструювання програмного забезпечення та об’єктно-орієнтованого програмування (для розробки системи), методи інженерії промптів та обробки природної мови (NLP) (для налаштування логіки LLM).

Практичне значення отриманих результатів полягає у створенні прикладного програмного модуля, який дозволяє суттєво скоротити часові витрати на ручне написання інженерних специфікацій, мінімізувати ризик людських помилок у конфігураціях IoT-пристроїв та забезпечити швидку автоматичну валідацію документів на відповідність технічним стандартам.

Апробація результатів дослідження. Основні положення та науково-практичні результати кваліфікаційної роботи доповідалися, обговорювалися та отримали позитивну оцінку на VII Міжнародній науково-технічній конференції «Сучасний стан та перспективи розвитку IoT» (м. Київ, Державний університет інформаційно-комунікаційних технологій, 2026 рік). Тема тези доповіді: «Автоматизація генерації та верифікації технічної документації для IoT-рішень на основі архітектури LLM».

1 ДОСЛІДЖЕННЯ ПРОБЛЕМАТИКИ СУПРОВОДУ IOT-РІШЕНЬ ТА МЕТОДІВ АВТОМАТИЗАЦІЇ

1.1 Специфіка та життєвий цикл технічної документації в галузі Internet of Things

Фундаментальною ознакою сучасних IoT-ініціатив є екстремально високий рівень гетерогенності апаратного та програмного забезпечення. У межах типової системи моніторингу одночасно функціонують різнорідні масиви сенсорів, мережеві топології (наприклад, Zigbee), протоколи агрегації даних типу MQTT [1] та розподілені хмарні платформи. Складність архітектурного опису таких екосистем зумовлена необхідністю забезпечення прозорості для нових учасників розробки та оперативності діагностики для інженерного персоналу. Утім, традиційний інструментарій часто виявляється неспроможним адекватно відобразити динаміку фізичного рівня та мережевих конфігурацій.

Аналіз обмежень традиційних методологій. У класичних циклах розробки програмного забезпечення сформовано сталий стек інструментів: специфікації OpenAPI (Swagger) [14] для опису інтерфейсів та модель C4 [2] для візуалізації архітектури. Проте ці засоби орієнтовані на стабільні релізні цикли, що суперечить природі Internet of Things. IoT-середовище перманентно трансформується: від спонтанної появи нових вузлів у мережі до нічних оновлень мікропрограм за технологією FOTA. Будь-яка ітерація такої динаміки миттєво нівелює актуальність наявних описів, що вимагає переходу до стандартів життєвого циклу інформації для користувачів, таких як ISO/IEC/IEEE 26514:2022 [7].

Особливого значення набуває безпековий аспект. Якщо помилка у програмному коді зазвичай призводить до виняткових ситуацій (exceptions), то дефекти в описі фізичних параметрів (наприклад, розпинування інтерфейсів) мають незворотні

наслідки. Подача напруги 5 В на лінію, розраховану на 3,3 В, спричиняє фізичну деструкцію напівпровідникових компонентів. В умовах індустріальних систем ціна такої неточності включає не лише прямі збитки від втрати обладнання, а й значні економічні втрати через простій виробничих ліній. Відтак, технічна документація в IoT виступає критичним елементом контуру безпеки.

Декомпозиція етапів життєвого циклу. Процес супроводу документації в IoT-проєкті доцільно розглядати як циклічну структуру (рисунок 1.1). Складність кожного етапу зумовлює виникнення специфічних дефектів, що систематизовані в таблиці 1.1.

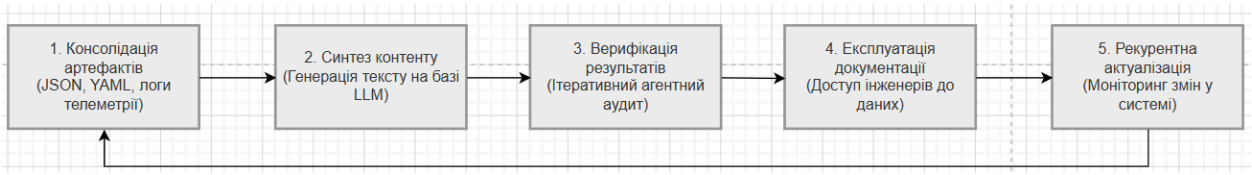


Рис. 1.1 – Життєвий цикл документації

Таблиця 1.1

Систематизація дефектів супроводу документації за етапами життєвого циклу

Етап	Ключова детермінанта проблеми	Типовий наслідок
1. Консолідація даних	Гетерогенність форматів (JSON, бінарні протоколи)	Інформаційна суперечливість відомостей
2. Створення документа	Рутинні операції (до 30 % часу інженера)	Помилки через когнітивну втому
3. Верифікація	Брак незалежного аудиту (режим самоконтролю)	Проникнення дефектів у реліз

Продовження табл 1.1

4. Використання	Монолітність структури для різних ролей	Часова деградація пошуку даних
5. Актуалізація	Пріоритетність операційних задач над описовими	Розбіжність із реальним станом до 70 %

Обґрунтування доцільності впровадження LLM-технологій. Очевидно, що ручні методи документування не здатні забезпечити адекватну швидкість реакції на динаміку змін в IoT. Стандартні засоби автоматизації виявляються неефективними перед неструктурованими діагностичними виводами. Застосування великих мовних моделей (Large Language Models, LLM), зокрема архітектури Llama 3.3 [11], є перспективним шляхом до інтелектуального синтезу описів на основі різнорідні джерел. Однак критична схильність LLM до фактологічних галюцинацій [5] вимагає розробки спеціалізованих контурів верифікації, що базуються на методах саморефлексії [8] та ітеративного аудиту.

1.2 Фундаментальні обмеження великих мовних моделей у задачах прецизійного інженерного документування

Застосування архітектур LLM для автоматизації формування специфікацій на основі неструктурованих логів є перспективним напрямом, проте потребує критичного переосмислення в контексті інженерної точності. На відміну від творчих чи маркетингових завдань, технічна документація IoT-систем характеризується нульовою толерантністю до апроксимацій. Похибка в описі електричних характеристик (наприклад, зазначення напруги живлення 5 В замість фактичних 3,3 В) трансформується у прямий ризик техногенної аварії. Відтак, інтеграція LLM у промисловий контур вимагає детального аналізу їхніх архітектурних обмежень.

1.2.1 Природа фактологічних девіацій

Явище галюцинацій у мовних моделях не є випадковим дефектом, а виступає фундаментальною властивістю авторегресійних архітектур [5]. Модель не оперує логічними концептами, а здійснює послідовне прогнозування найбільш імовірного токена, спираючись на статистичні закономірності тренувального корпусу. Математично цей процес описується функцією розподілу ймовірностей:

Де

$$P(x_t \mid x_{<t}; \theta) = softmax(W \cdot h_t) \quad (1.1)$$

— умовна ймовірність генерації токена x_t на основі попереднього контексту;

- $P(x_t \mid x_{<t}; \theta)$ — послідовність раніше згенерованих токенів;
- $x_{<t}$ — параметри (ваги) мовної моделі;
- $softmax$ — функція активації, що нормалізує вихідні значення нейромережі у розподіл ймовірностей;
- W — матриця ваг вихідного лінійного шару;
- h_t — прихований стан моделі на поточному кроці t.

Якщо у вхідному контексті зафіксовано діапазон температур «0 °C … +50 °C», а в навчальній вибірці переважали даташити з інтервалом «–40 °C … +85 °C», виникає ризик регресії до середнього значення. Механізм самоуваги (*self-attention*) [20] може не забезпечити необхідну математичну вагу для конкретного числового літерала, що призводить до генерації «статистично правдоподібного», але технічно помилкового значення.1.2.2 Типологія помилок LLM у задачах IoT

У межах дослідження проведено тестування моделі Llama 3.3 [11] на вибірці реальних IoT-специфікацій, що дозволило класифікувати критичні типи дефектів, які наведено в таблиці 1.2.

Таблиця 1.2

Класифікація критичних типів дефектів LLM при формуванні IoT-специфікацій		
Клас помилки	Архітектурна першопричина	Технічний прояв
Числова девіація	BPE-токенізація, регресія до середнього значення	3,3 В → 3 В; 10 мс → 100 мс
Підміна ідентифікаторів	Векторна близькість схожих токенів у латентному просторі	temp_01 → temp_02
Семантична інверсія	Втрата логічних заперечень при компресії контексту	«не підтримує» → «забезпечує»
Структурна деградація	Брак жорстких граматичних обмежень під час генерації	Порушення синтаксису схеми JSON
Неузгодженість одиниць вимірювання	Стохастична заміна префіксів Міжнародної системи одиниць (CI)	100 мкс → 100 мс

1.2.3 Фундаментальні причини обмежень системи

До основних чинників, що зумовлюють обмеження великих мовних моделей у задачах прецизійного інженерного документування, належать: відсутність механізму заземлення (*grounding*), внаслідок чого модель не має безпосереднього зв'язку з об'єктами та параметрами фізичного світу [20]; стохастичність процесів інференсу (виведення), зумовлена недетермінованістю паралельних обчислень на рівні архітектури CUDA; а також низька ефективність автономної самоперевірки моделей без зовнішнього контуру валідації [8].

Нівелювання зазначених обмежень та мінімізація ризиків генерації недостовірних даних досягається за допомогою інтеграції методів генерації з доповненою вибіркою (*Retrieval-Augmented Generation, RAG*) [9], а також шляхом переведення мовної моделі в режим автономного агента-аудитора на основі парадигм обчислювального мислення *Chain-of-Thought* [21] та *ReAct* [22].

1.3 Порівняльний аналіз методів верифікації: RAG-підхід та агентний AI-аудит

Забезпечення автентичності результатів генерації у разі використання великих мовних моделей (LLM) вимагає впровадження спеціалізованих контурів контролю, оскільки стандартні методи безконтекстного запиту (*zero-shot prompting*) не гарантують прецизійної точності інженерних даних. У сучасних дослідженнях домінують два вектори розв'язання цієї проблематики: превентивний (із застосуванням архітектури генерації з доповненою вибіркою, *Retrieval-Augmented Generation, RAG*) та коригувальний (на основі агентного аудиту). У цьому підрозділі здійснено їхню компаративну оцінку з погляду придатності для автоматизованого документування гетерогенних IoT-систем.

1.3.1 RAG: методологічний базис та архітектурні обмеження

Концепція генерації з доповненою вибіркою (*Retrieval-Augmented Generation, RAG*) базується на динамічному збагаченні системного промпту релевантним контекстом, вилученим із зовнішніх векторизованих баз знань безпосередньо перед фазою інференсу. Математична формалізація цього підходу має такий вигляд:

$$y = LLM(x \oplus Retrieve(x, D)) \qquad (1.2)$$

де y — сформований текст технічної документації;

- LLM — оператор перетворення (інференсу) великої мовної моделі;
- x — початковий інженерний запит або контекст користувача;
- $\oplus$ — оператор конкатенації текстових послідовностей;
- $Retrieve$ — функція семантичного пошуку релевантних фрагментів;
- D — векторизований корпус інженерних даних (даташити, специфікації, конфігураційні файли, логи телеметрії).

Утім, під час документування IoT-рішень технологія RAG стикається з трьома критичними викликами. По-перше, ефективність пошуку інформації безпосередньо залежить від обраної стратегії сегментації (*chunking*) документів; некоректне або занадто жорстке розбиття тексту на блоки може призвести до втрати логічних і

структурних зв’язків між взаємозалежними параметрами пристроїв. По-друге, цей метод має виключно превентивний характер і не передбачає контуру пост-генераційної перевірки узгодженості сформованого тексту. По-третє, архітектура RAG припускає апріорну достовірність джерела D, що є високим ризиком у разі використання застарілих логів або конфліктних конфігураційних файлів інфраструктури Інтернету речей.

1.3.2 Агентний аудит: ітеративна верифікація та саморефлексія

Альтернативою виступає агентна парадигма, де модель Llama 3,3 функціонує в ролі автономного аудитора. Даний підхід базується на методах ланцюга міркувань (Chain-of-Thought) та взаємодії ReAct [22], що дозволяє моделі послідовно зіставляти згенерований текст із вхідними специфікаціями.

Ключовою перевагою агентного підходу є його проактивність: аудит не просто створює передумови для точності, а активно ідентифікує та усуває дефекти через цикли саморефлексії (self-reflection). Можливість розгортання мультиагентних систем дозволяє розподілити зони відповідальності: окремі агенти можуть валідувати синтаксичну структуру JSON, числовий фактаж та термінологічну відповідність. Проте ітеративність процесу створює значне когнітивне та обчислювальне навантаження, що висуває особливі вимоги до продуктивності інфраструктури.

1.3.3 Обґрунтування гібридної архітектури верифікації

На основі проведеного аналізу сформовано порівняльну характеристику методів (табл. 1.3).

Таблиця 1.3

Порівняльний аналіз превентивного RAG-підходу та агентного AI-аудиту

Критерій порівняння	RAG-підхід	Агентний AI-аудит

Продовження табл 1.3

Метод контролю	Превентивний (контекстне збагачення запиту)	Коригувальний (післягенераційна валідація)
Валідація структури	Опосередкована (через структуру джерела контексту)	Пряма (автоматичний синтаксичний аналіз схем)
Числова точність	Статистична ймовірність (ризик регресії токенів)	Експліцитне порівняння та математична верифікація
Реакція на аномалії	Трансляція та масштабування похибок джерела даних	Активна ідентифікація та логування суперечностей
Рівень латентності	Мінімальний (однократний інференс моделі)	Високий (зумовлений ітераційними циклами перевірки)
Гнучкість правил	Низька (обмежена фіксованим вектором контексту)	Висока (регулюється динамічною інженерією промптів)

Встановлено, що RAG та агентний аудит є взаємодоповнюючими технологіями. RAG забезпечує формування надійного фактологічного фундаменту, тоді як агентний шар виконує фінальну верифікацію та усуває залишкові девіації. Саме така синергічна модель — «RAG-генерація + багаторівневий агентний аудит» — покладена в основу даного дослідження. Для нівелювання високої латентності ітеративних циклів у роботі застосовано архітектуру Groq LPU, що дозволяє досягти

необхідної швидкодії при обробці великих масивів IoT-даних. Обґрунтування технічного вибору платформи наведено у підрозділі 1.4.

1.4 Обґрунтування використання високопродуктивної інфраструктури Groq

Впровадження агентного контуру верифікації призводить до експоненціального зростання кількості запитів до мовної моделі в межах однієї сесії документування. Експлуатація традиційних обчислювальних архітектур на базі GPU зумовлює критичне зростання латентності, що робить систему непридатною для оперативного використання. Забезпечення детермінованості та швидкодії при ітеративній обробці технічних даних вимагає застосування спеціалізованих інференс-платформ, серед яких найбільш ефективною для задач NLP є архітектура Groq LPU [4].

1.4.1 Технологічні обмеження GPU-архітектур.

Стандартні графічні прискорювачі (наприклад, NVIDIA H100) при виконанні авторегресійної генерації обмежуються пропускною здатністю пам'яті (проблема Memory Wall). Процес зчитування матриці ваг моделі Llama 3,3 70B (обсягом близько 140 Гбайт) при кожному кроці передбачення токена створює часовий бар’єр, що обмежує швидкість інференсу рівнем 20–30 токен/с. Враховуючи необхідність виконання щонайменше двох додаткових циклів аудиту, сумарний час генерації одного документа може становити 12–15 хв. Окрім того, прихований недетермінізм паралельних обчислень у бібліотеках CUDA може призводити до варіативності результатів, що є критичним недоліком для прецизійної інженерної документації.

1.4.2 Архітектурні переваги Groq LPU

Тензорні процесори Groq базуються на спеціалізованій архітектурі LPU (Language Processing Unit), де модельний контекст повністю розміщується в надшвидкій статичній пам’яті SRAM безпосередньо на кристалі обчислювача [4]. Дане технологічне рішення дозволяє практично нівелювати часові затримки доступу до зовнішньої динамічної пам'яті, повністю долаючи проблему апаратного бар’єра

Memory Wall. Обчислювальний конвеєр виконання є апаратно-статичним та детермінованим, що гарантує абсолютну відтворюваність і стабільність фактологічного змісту технічного тексту при повторних запусках на ідентичних вхідних даних.

Швидкість генерації для цільової мовної моделі Llama 3,3 70B на інфраструктурі Groq LPU сягає 350 токен/с, що забезпечує апаратно-експлуатаційну перевагу у швидкодії у 1,8 раза порівняно з флагманським графічним прискорювачем класу NVIDIA H100 (195 токен/с) в умовах інтерактивного однокористувацького навантаження. Емпіричні результати порівняльного аналізу пропускної здатності для різних конфігурацій обчислювальних архітектур станом на лютий 2026 року представлено на рис. 1.2.

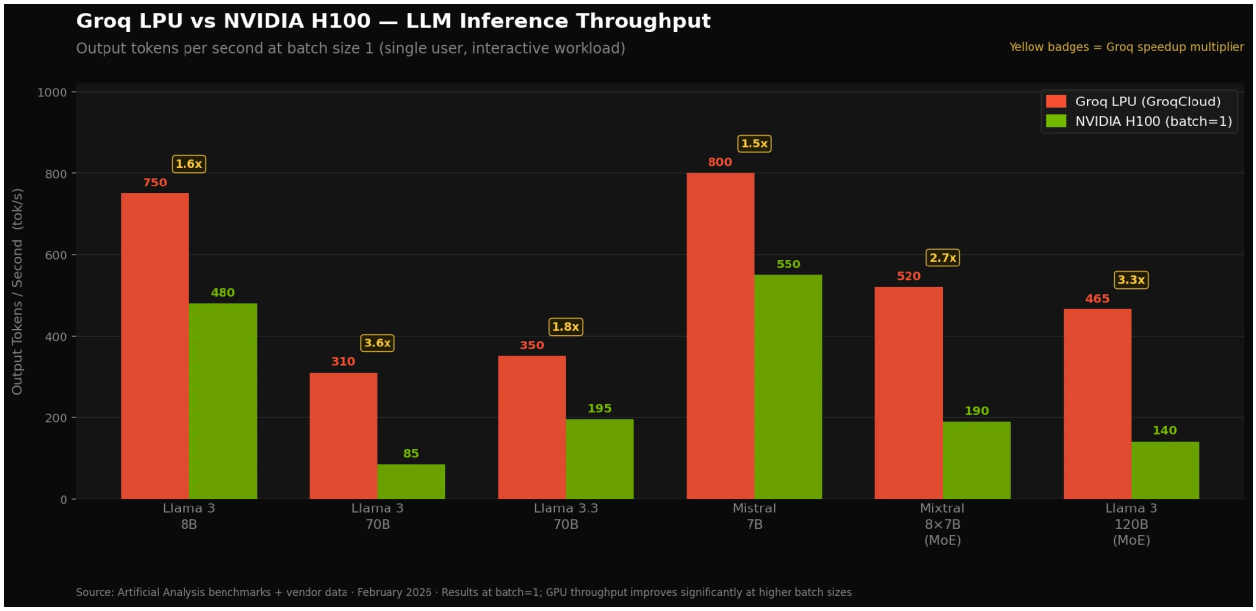


Рис. 1.2 – Порівняльна продуктивність Groq LPU та NVIDIA H100 при інференсі LLM (за даними Artificial Analysis, 2026 р.)

Показник часу до видачі першого токена (Time to First Token, TTFT) на рівні 100 мс стає ключовим фактором для забезпечення плавної взаємодії з користувачем. Оскільки архітектура нашої системи передбачає асинхронний обмін даними за допомогою JavaScript Fetch API та обробку маршрутів на базі веб-фреймворку Flask [18], це дозволяє звести до мінімуму мережеві затримки. Клієнтська частина миттєво

рендерить відповіді від ШІ-агентів, не перевантажуючи всю сторінку, що вкрай важливо при покроковому аудиті складних технічних специфікацій.

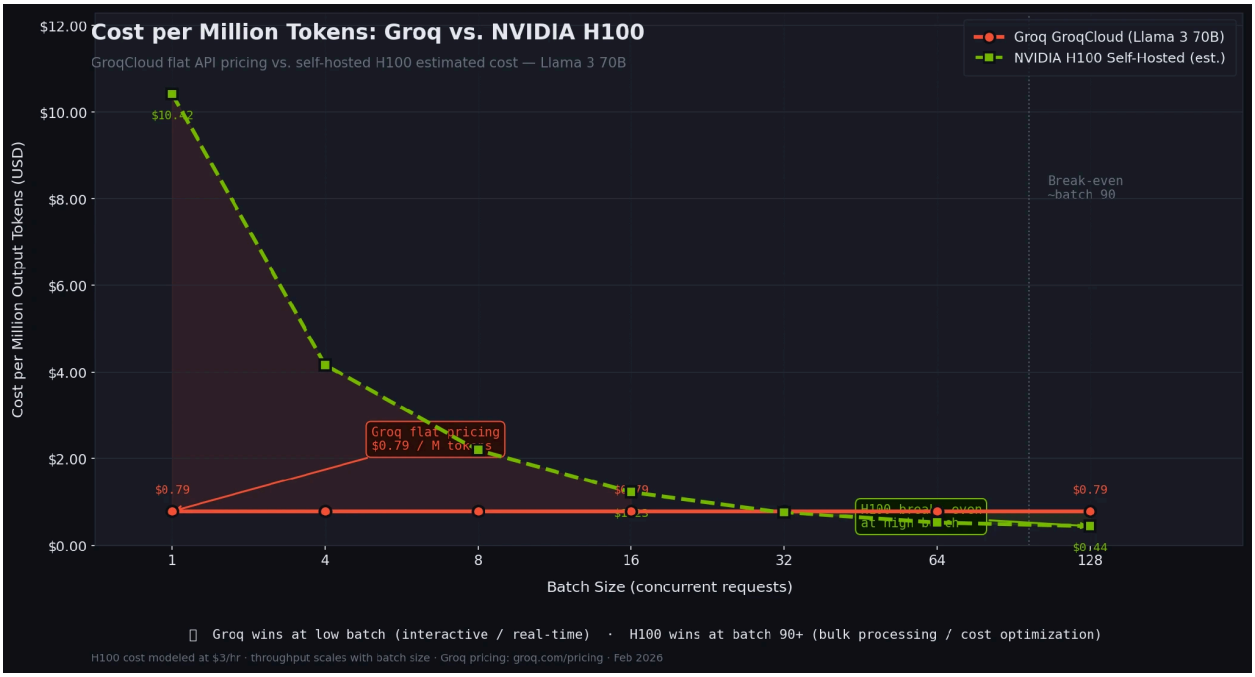


Рис. 1.3 − Компаративний аналіз вартості генерації 1 млн вихідних токенів залежно від обсягу пакету паралельних запитів (за даними Artificial Analysis, 2026 р.)

1.4.3 Відповідність вимогам системи.

Порівняльний аналіз характеристик Groq LPU та вимог розробленого програмного комплексу наведено в таблиці 1.4.

Таблиця 1.4

Порівняльний аналіз характеристик Groq LPU

Системна вимога	Технічна характеристика інфраструктури Groq LPU	Рівень відповідності
Мінімальна латентність	Час до отримання першого токена (TTFT) < 100 мс, швидкість генерації > 300 токенів/с	Повна відповідність

Продовження табл 1.4

Відтворюваність результатів	Апаратно-детермінований конвеєр виконання обчислювальних графів нейромережі	Повна відповідність
Пропускна здатність	Швидкість обміну даними > 80 Тбайт/с (через внутрішню шину кристала)	Повна відповідність
Підтримка Llama 3.3 70B	Нативна інтеграція моделі через хмарну інфраструктуру Groq Cloud [11]	Повна відповідність
Технологічна сумісність	Наявність офіційного Python SDK та підтримка архітектурного стилю REST API [16]	Повна відповідність
Економічна доцільність	Прогнозована вартість обчислювальних ресурсів < 0,02 USD за один згенерований документ	Повна відповідність

Використання Groq LPU забезпечує необхідну синергію швидкодії та детермінізму, що є критично важливим для реалізації гібридної архітектури верифікації в практичних умовах.

На основі проведеного аналізу предметної області та обґрунтованого вибору технологічного стеку сформульовано науково-технічну задачу, що розв’язується в межах даної кваліфікаційної роботи.

Вхідні дані:

- структуровані специфікації IoT-пристроїв у форматах JSON/YAML;
- масиви неструктурованих системних логів (MQTT-трейси, syslog) [1, 13];

- шаблони документів згідно з корпоративними стандартами оформлення.

Вихідні дані:

- верифікований технічний опис у форматі .docx [15, 20], що містить зв’язний текст, таблиці параметрів та журнал виявлених і виправлених невідповідностей;
- кількісні метрики якості: BERTScore (F1) [23] та Cosine Similarity [17].

Основні обмеження та вимоги:

- обчислювальне ядро: модель Llama 3,3 70B через Groq API [11];
- інструментарій розробки: Python [16], веб-інтерфейс — Flask [18];
- часовий регламент: повний цикл обробки документа (генерація та два цикли аудиту) — не більше 60 с;
- вимога детермінованості: забезпечення ідентичності фактологічного змісту при повторних запусках на тих самих вхідних даних.

Критерії успішності розробки:

- значення метрики BERTScore F1 $\geq 0{,}85$ [23];
- рівень семантичної близькості за Cosine Similarity $\geq 0{,}90$;
- ефективність шару верифікації: виявлення не менше 90 % внесених невідповідностей;
- експлуатаційна перевага: скорочення часових витрат на документування не менш ніж у 10 разів порівняно з ручним методом.

ВИСНОВКИ ДО РОЗДІЛУ 1

У першому розділі виконано комплексний аналіз проблематики супроводу технічної документації в галузі Інтернету речей та визначено теоретичні засади її автоматизації за допомогою великих мовних моделей. За результатами дослідження зроблено такі висновки:

1. Встановлено, що висока гетерогенність та перманентна динаміка інфраструктури IoT призводять до стрімкої деградації ручних методів документування, викликаючи розбіжність між описми та реальним станом систем на рівні 50–70 %.

2. Систематизовано та класифіковано п’ять критичних типів фактологічних дефектів, які виникають при використанні авторегресійних архітектур LLM в інженерному контурі: числові девіації, ідентифікаторні підміни, семантичні інверсії, структурна деградація та одиничні неузгодженості.

3. Доведено, що використання стандартних підходів до документування IoT-систем не встигає за швидкими змінами інженерних конфігурацій. Обґрунтовано доцільність автоматизації цих процесів за допомогою моделей родини Llama, які запускаються через низьколатентну інфраструктуру Groq API.

4. Визначено межі проєктування майбутнього вебсервісу: вирішено відмовитися від монолітних інтерфейсів швидкого прототипування на користь клієнт-серверної моделі. Серверна логіка базуватиметься на Flask для гнучкого керування REST-маршрутами, а фронтенд забезпечить інтерактивний візуальний Diff-аналіз згенерованого тексту для фіксації ШІ-галюцинацій.

2 ПРОЄКТУВАННЯ СИСТЕМИ ГЕНЕРАЦІЇ ТА ВЕРИФІКАЦІЇ НА БАЗІ LLM

2.1 Формування функціональних вимог до програмного комплексу

Перед проєктуванням архітектури визначено функціональні межі та експлуатаційні можливості системи. Вимоги сформовано на основі аналізу типових робочих сценаріїв інженера IoT та обмежень, виявлених під час дослідження у Розділі 1. Нижче наведено функціональні вимоги (FR1–FR7) та нефункціональні вимоги (NFR1–NFR6), пріоритезовані за методом MoSCoW.

FR1. Завантаження вхідних даних (Must have). Drag-and-drop для JSON, YAML, TXT (до 50 МБ) з автоматичним визначенням формату.

FR2. Парсинг і нормалізація (Must have). Вилучення сутностей (пристрої, параметри, метрики) з приведенням до єдиного внутрішнього формату. Для логів — регулярні вирази (IP, MAC, UUID).

FR3. Генерація тексту (Must have). Формування RAG-запиту до Llama 3.3 через Groq API з жорсткою інструкцією використовувати тільки надані дані.

FR4. Агентна верифікація (Must have). Двоцикловий аудит (max_retries=2) з виявленням невідповідностей та маркуванням потребуючих ручної перевірки.

FR5. Експорт у .docx (Must have). Компіляція верифікованого тексту у Word з автоматичним змістом та журналом аудиту за допомогою python-docx [15].

FR6. Асинхронний веб-інтерфейс (Must have). Взаємодія через браузер з Drag-and-Drop, трекером статусів та Diff-View.

FR7. Збереження історії (Should have). Реєстрація оброблених документів у SQLite (час, назва, метрики BERTScore та Cosine Similarity).

Нефункціональні вимоги:

NFR1 (Продуктивність): сумарний час обробки документа (генерація та два цикли аудиту) не має перевищувати 60 с.

NFR2 (Відтворюваність): ідентичність фактологічного змісту при повторних запусках забезпечується через параметр *temperature* = 0,0 та апаратний детермінізм Groq LPU.

NFR3 (Стійкість): коректна обробка виняткових ситуацій (невалідний JSON, перевищення ліміту обсягу файлу, збої API) з виведенням повідомлень користувачу.

NFR4 (Юзабіліті): відсутність вимог до спеціальних знань у галузі машинного навчання у кінцевого користувача.

NFR5 (Масштабованість): можливість інтеграції ядра системи в CI/CD-конвеєри (наприклад, Jenkins або GitLab CI).

NFR6 (Безпека): взаємодія з API здійснюється через протоколи HTTPS/TLS; ключі доступу зберігаються виключно у змінних середовища (*.env*).

Таблиця 1.5

Матриця відповідності вимог та компонентів системи

Вимога	Компонент системи	Пріоритет
FR1, FR2	Parser Module	High
FR3	Generation Module	High
FR4	Audit Agent (Verification Loop)	High
FR5	Export Module	Medium
FR6	Web Interface (HTML5 + Tailwind CSS + Fetch API)	High
FR7	Storage Module (SQLite)	Low

2.2 Розробка модульної архітектури системи (компоненти UI, Logic та API Service)

На відміну від класичних монолітних рішень швидкого прототипування, де логіка обчислень та інтерфейс жорстко пов'язані в одному потоці виконання, у розробленій системі застосовано клієнт-серверний підхід із чітким розділенням зон відповідальності (Separation of Concerns). Серверна частина, побудована на базі мікрофреймворку Flask, виступає як ізольоване RESTful API. Вона відповідає за маршрутизацію, сесійної автентифікації користувачів, комунікацію з Groq API та нормалізацію звітів верифікації. Клієнтська частина (фронтенд) бере на себе функцію візуального представлення та обробки локальних подій користувача за допомогою асинхронних сценаріїв.

2.2.1 Загальна архітектурна концепція

Програмну систему спроєктовано за принципом багатошарової архітектури, що складається з трьох логічних рівнів. Такий підхід забезпечує високий рівень інкапсуляції: кожен шар виконує специфічну роль та має мінімальну залежність від внутрішньої реалізації суміжних рівнів (рис. 2.1).

- Presentation Layer (UI) — HTML5, Tailwind CSS, JavaScript (Fetch API). Позбавлений бізнес-логіки, лише прийом даних та візуалізація.
- Service Layer (Core Logic) — парсинг, генерація, агентна верифікація, компіляція звітів.
- Data Access Layer (DAL) — SQLite для зберігання історії обробок.

Залежності спрямовані вниз: UI → Service → DAL. Це дозволяє використовувати Service Layer без UI (наприклад, у CI/CD).

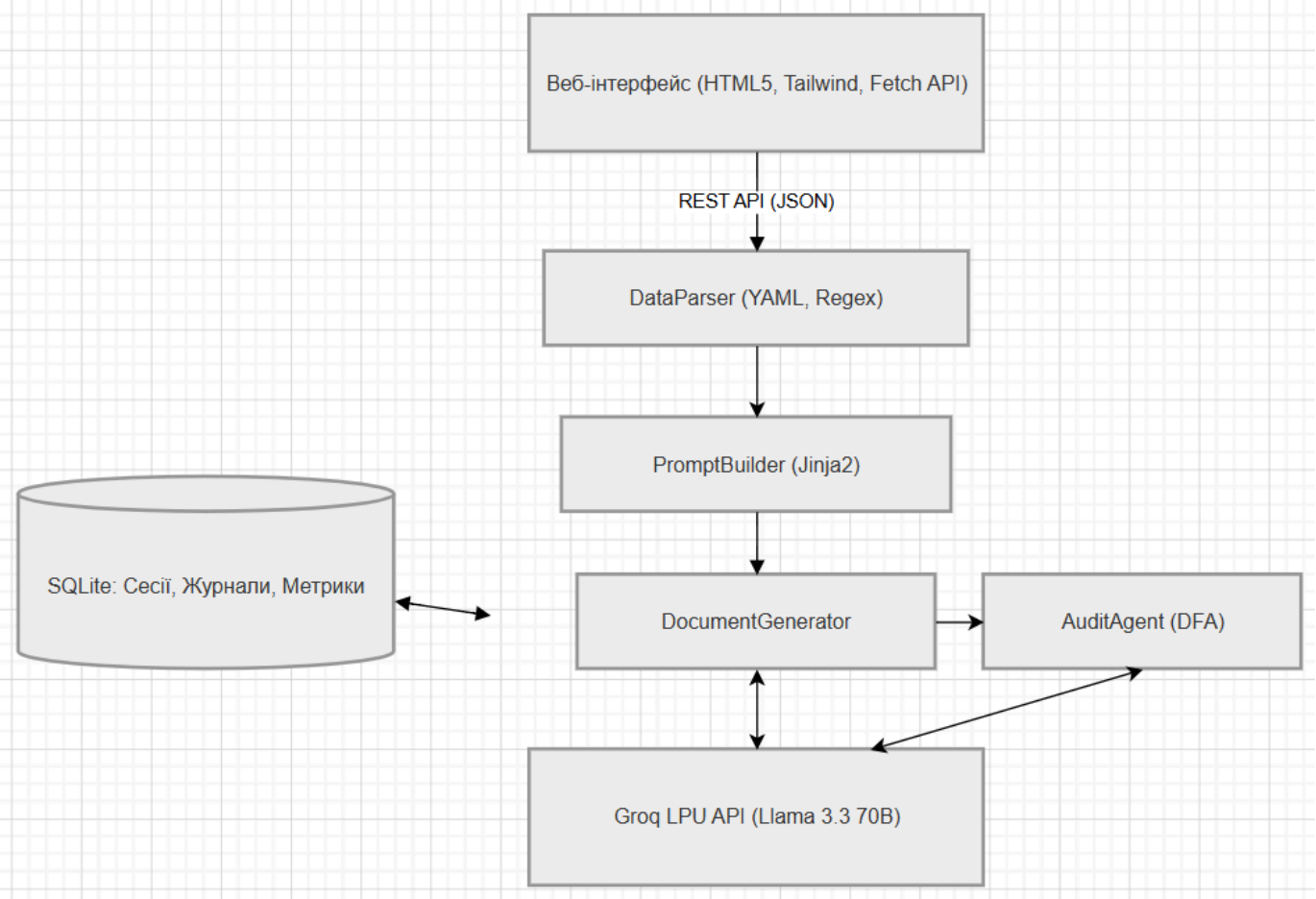


Рис. 2.1 – Загальна тришарова архітектура програмного комплексу

2.2.2 Шар представлення (UI)

Графічний інтерфейс користувача спроєктовано як компонентну вебсторінку, де кожен елемент взаємодії є частиною загального DOM-дерева, керованого клієнтськими скриптами. Замість використання стандартних серверних віджетів, уся логіка побудована на гнучких вебстандартах, що дозволило реалізувати складні візуальні сценарії.

Основні компоненти інтерфейсної частини системи включають:

— Зона асинхронного завантаження інженерних артефактів (Drag-and-Drop): реалізована за допомогою стандартного HTML5-контейнера та JavaScript-обробників подій перетягування файлів (dragenter, dragover, dragleave, drop). Це забезпечує локальне зчитування конфігураційних файлів у форматі .json або .txt безпосередньо в полі введення без відправки зайвих запитів на сервер.

— Контейнер експертного керування промпт-інжинірингом: побудований на базі тегу <details>, що дозволяє приховати складні технічні параметри (системні інструкції та шаблони запитів для Llama 3.3) від звичайного користувача, залишаючи можливість швидкого доступу до них для технічних письменників.

— Панель інтерактивної екстракції IoT-сутностей: динамічний блок, який активується після отримання JSON-відповіді від сервера. Клієнтський сценарій автоматично розбирає структуру даних та рендерить знайдені апаратні компоненти (мікроконтролери, піни, датчики та мережеві протоколи) у вигляді ізольованих інформаційних карток.

— Двокомпонентне вікно відображення результату з режимом Diff-View: ключовий елемент інтерфейсу, що містить перемикач між сирим текстом документації та візуальним звітом комплаєнс-контролю. У режимі *Diff-View* уся текстова інформація форматується, а знайдені контуром верифікації суперечності чи галюцинації ШІ динамічно маркуються в тексті за допомогою стилістичних класів Tailwind CSS (елементи <mark>), фокусуючи увагу інженера на проблемних місцях.

— Модуль управління експортом: містить динамічні експортні форми, пов'язані з роутами Flask /download/word та /download/audit. Кнопки активуються лише після успішного завершення аналізу, дозволяючи завантажити як саму документацію, так і офіційний протокол технічного аудиту.

2.2.3 Шар бізнес-логіки (Service Layer)

Ядро системи представлено шістьма незалежними модулями, кожен з яких реалізовано у вигляді спеціалізованого Python-класу або набору функцій у межах пакету *core/*. Така архітектура забезпечує модульність та спрощує процес тестування компонентів.

Parser Module (parser.py). Відповідає за реалізацію вимог FR1 та FR2. Модуль містить клас *DataParser* із такими методами:

— *parse_file(filepath: str) -> dict* — автоматично ідентифікує формат вхідних даних за розширенням та повертає нормалізований об’єкт (словник Python);

– *validate_json(data: str) -> bool* — здійснює перевірку синтаксичної коректності структурованих даних (JSON/YAML);

– *extract_entities(raw_text: str) -> list[dict]* — реалізує механізм вилучення сутностей із неструктурованих логів за допомогою набору попередньо скомпільованих регулярних виразів (ідентифікація IP- та MAC-адрес, UUID, числових літералів з одиницями вимірювання).

Внутрішня обробка базується на нормалізації вхідних даних до єдиного стандарту, що містить тип сутності (*device/sensor/protocol/metric*), параметри та джерело походження даних.

Prompt Builder (prompts.py). Модуль відповідає за управління шаблонами системних запитів для генератора та аудитора. Реалізований як клас *PromptManager*, що забезпечує:

– зберігання базових шаблонів у форматі Jinja2;

– формування фінального тексту запиту на основі нормалізованих даних та параметрів (роль моделі, глибина перевірки);

– гнучку зміну стратегій промпт-інжинірингу без модифікації логіки основних модулів.

Generation Module (generator.py). Забезпечує виконання вимоги FR3. Клас *DocumentGenerator* реалізує публічний метод *generate()*, який:

1. отримує сформований системний запит від *Prompt Builder*;
2. ініціює виклик Groq API через метод *groq.ChatCompletion.create()* із використанням моделі *llama-3.3-70b-versatile* при параметрі *temperature* = 0,0;
3. повертає згенерований технічний текст як рядкову величину.

Audit Agent (auditor.py). Ключовий компонент системи, що реалізує агентну верифікацію (FR4). Клас *AuditAgent* інкапсулює логіку автомата станів, що детально описана у підрозділі 2.4. Основні методи:

– *audit(generated_text: str, input_data: dict) -> dict* — здійснює повний цикл верифікації, повертаючи об’єкт із фінальним текстом та переліком виправлень;

- *_verify(text: str, reference: dict) -> list[dict]* — реалізує одну ітерацію перевірки шляхом надсилання запиту до LLM у режимі аудитора;
- *_apply_fixes(original_text: str, issues: list[dict]) -> str* — автоматично інтегрує запропоновані аудитором виправлення в основний текст.

При вичерпанні ліміту спроб (*max_retries*) система встановлює статус *needs_human_review* та маркує спірні фрагменти тексту.

Export Module (exporter.py). Реалізує вимогу FR5. Клас *DocxExporter* забезпечує компіляцію фінального верифікованого тексту та журналу результатів аудиту (*audit trail*). Метод *export()* формує файл формату .docx, використовуючи бібліотеку *python-docx*. Документ включає:

- титульну сторінку з метаданими проєкту;
- автоматично згенерований зміст (TOC);
- структуровані розділи згідно з обраним шаблоном;
- розділ «Audit Trail» із таблицею невідповідностей (тип помилки, оригінальне та виправлене значення, статус).

Orchestrator (pipeline.py). Координаційний модуль, що забезпечує послідовне виконання етапів обробки даних через функцію *run_pipeline()*. Конвеєр включає послідовність виклику парсера, генератора та аудитора з подальшим збереженням результатів у базу даних через *Storage Module*.

2.2.4 Шар доступу до даних (Data Access Layer)

Storage Module (storage.py). Реалізований як клас *HistoryStore*, що інкапсулює логіку взаємодії з базою даних SQLite засобами стандартної бібліотеки *sqlite3* [16]. Модуль забезпечує виконання таких операцій:

add_entry(metadata: dict) — реєстрація нового запису про сесію обробки даних;

get_history(limit: int = 20) -> list[dict] — ретроспективне отримання останніх записів;

purge_old(days: int = 90) — автоматичне очищення застарілих даних для оптимізації обсягу сховища.

Структура бази даних ініціалізується автоматично при першому запуску системи. Схема таблиці *history* включає поля: *id* (первинний ключ), *timestamp* (мітка часу), *input_files* (перелік джерел), *metrics_json* (результати обчислення BERTScore [23] та Cosine Similarity) та *docx_path* (шлях до згенерованого артефакту). Використання формату JSON для зберігання метрик забезпечує гнучкість структури даних без необхідності частої зміни схеми БД [10].

2.2.5 Взаємодія між модулями: потік даних

Взаємодія між модулями реалізована через асинхронну REST API архітектуру [18]. Типовий потік обробки:

1. Клієнт завантажує файли через Drag-and-Drop, вибирає тип документа, JavaScript надсилає POST-запит на /process.
2. Flask-сервер передає дані в DataParser, який нормалізує JSON/YAML/логи в уніфікований словник Python.
3. Prompt Builder інтегрує нормалізовані дані в системний запит (RAG-контекст).
4. Generation Module викликає Groq API [4] з temperature=0.0, отримує первинний текст.

5. Audit Agent запускає до 2 циклів перевірки: надсилає запит аудитору (Chain-of-Thought [21], ReAct [22]), отримує JSON з помилками, застосовує виправлення.

6. DocxExporter [15] генерує .docx та повертає клієнту через send_file().

7. SQLite зберігає історію обробок.

Після отримання JSON-відповіді клієнт рендерить Diff-View, підсвічуючи автоматичні виправлення (зелений) та фрагменти, що потребують ручної перевірки (жовтий).

2.2.6 Обґрунтування архітектурних рішень

Вибір тришарової модульної структури зумовлений необхідністю забезпечення високої супроводжуваності та тестованості програмного комплексу.

1. Декомпозиція рівнів. Відокремлення інтерфейсу дозволяє інтегрувати ядро системи в CI/CD-конвеєри або використовувати його як автономний CLI-інструмент без модифікації логіки обробки даних.

2. Автономність модулів Service Layer. Модулі парсингу, генерації та аудиту є функціонально незалежними, що дозволяє здійснювати юніт-тестування кожного компонента окремо, підвищуючи надійність системи.

3. Ізоляція логіки промпт-інжинірингу. Винесення шаблонів запитів у модуль *Prompt Builder* дозволяє експериментувати зі стратегіями Chain-of-Thought [21] та ReAct [22], не вносячи змін у програмний код виклику API.

4. Оптимізація циклів верифікації. Прийнято рішення про обмеження кількості ітерацій аудиту двома циклами. Це забезпечує баланс між фінансовими витратами на токени, часовою латентністю та необхідною точністю, залишаючи можливість фінального Human-in-the-loop контролю.

5. Використання офіційних SDK. Застосування пакету *groq* [4, 16] замість прямих HTTP-запитів дозволяє мінімізувати кількість програмних дефектів при авторизації та обробці помилок мережі, прискорюючи розробку.

Таким чином, розроблена архітектура повною мірою забезпечує реалізацію сформульованих вимог, створюючи надійну базу для програмної реалізації системи.

2.3 Вибір та обґрунтування технологічного стеку системи

Після етапу концептуального проєктування та побудови загальної архітектурної моделі визначено оптимальний технологічний стек для програмної реалізації вебплатформи. Вибір інструментарію та середовища розробки базується на необхідності забезпечення високої обчислювальної швидкості обробки даних, суворої модульності ізольованих сервісів, а також можливості гнучкого горизонтального масштабування системи в майбутньому без потреби в реструктуризації її розрахункового ядра.

Мова програмування: Python 3.11

Вибір мови програмування Python зумовлений наявністю найбільш розвиненої та стабільної інженерної екосистеми спеціалізованих бібліотек для обробки природної мови (NLP) та інтеграції з великими мовними моделями. Версія 3.11 обрана через суттєве підвищення загальної продуктивності інтерпретатора при роботі з глибоко вкладеними словниковими структурами та довгими рядковими масивами, що є критично важливим фактором під час парсингу та нормалізації великих обсягів сирих логів телеметрії та JSON-специфікацій пристроїв [16]. Окрім того, Python забезпечує нативну підтримку офіційного інструментарію Groq SDK та аналітичних пакетів для розрахунку семантичних метрик [17,23].

Веб-фреймворк та інтерфейс: Flask, JavaScript (Fetch API) та Tailwind CSS

На відміну від монолітних інструментів швидкого прототипування, для реалізації серверної логіки застосунку та побудови REST-маршрутів обрано мікрофреймворк Flask [18]. Дане інженерне рішення дозволяє мінімізувати накладні обчислювальні витрати сервера, забезпечує строге розділення зон відповідальності (Separation of Concerns) та гарантує повну ізоляцію бізнес-логіки системи (Service Layer). Це спрощує подальшу інтеграцію платформи в промислові корпоративні CI/CD-конвеєри без модифікації базових алгоритмів.

Шар представлення (фронтенд) повністю відокремлений від серверного коду. Графічний інтерфейс користувача побудовано на базі сучасних стандартів HTML5 та утилітарного CSS-фреймворку Tailwind CSS, що дозволяє реалізувати адаптивну, низькокогнітивну компонентну модель сторінки. Взаємодія між клієнтською частиною та вебсервером Flask реалізована асинхронно за допомогою JavaScript Fetch API. Це забезпечує динамічне оновлення елементів DOM-дерева під час рендерингу результатів ітеративного аудиту у режимі Diff-View без перезавантаження сторінки, що повністю задовольняє нефункціональну вимогу щодо інтерактивності інтерфейсу.

Інференс-платформа: Groq API (LPU)

Впровадження розробленої двоконтурної схеми «генерація + подвійний агентний аудит» висуває жорсткі вимоги до пропускної здатності обчислювальної інфраструктури. Апаратна платформа Groq LPU (Language Processing Unit) обрана за трьома ключовими критеріями комплаєнсу [4]:

1. Експлуатаційна швидкість: Продуктивність на рівні понад 300 токенів/с для мовних моделей великого обсягу дозволяє завершити повний технологічний цикл обробки документа (включаючи два проходи скінченного автомата верифікації) менш ніж за 60 секунд, що чітко задовольняє вимогу NFR1.

2. Алгоритмічний детермінізм: Специфіка апаратно-статичного конвеєра Groq гарантує високу відтворюваність результатів інференсу, що є критично важливим фактором при фіксації та перевірці строгих технічних та цифрових параметрів інженерного заліза.

3. Економічна ефективність: Низька вартість обчислень (0.59 $ за 1 млн вхідних токенів та 0.79 $ за 1 млн вихідних токенів) знижує сумарні операційні витрати (OPEX) на формування одного технічного документа до рівня менше 0.02 $, що є експоненційно дешевше за вартість людино-годин профільного інженера чи технічного письменника.

Мовна модель: Llama 3.3 70B

Розрахунковим ядром системи виступає відкрита велика мовна модель Llama 3.3 [11]. Інтеграція 70B-версії є оптимальним архітектурним компромісом між якістю контекстного інженерного синтезу та загальною латентністю обробки запитів. На відміну від полегшених моделей (класу 8B), дана конфігурація демонструє вищі показники точності та стійкості при аналізі складних логічних і причинно-наслідкових зв'язків у гетерогенних IoT-специфікаціях, а порівняно з комерційними пропрієтарними рішеннями (наприклад, GPT-4) забезпечує значно менший час відгуку (Time-to-First-Token) при роботі через шлюз Groq API.

Інструментарій експорту та аналітичні метрики якості

Динамічний експорт верифікованих звітів у промисловий формат реалізується за допомогою спеціалізованої бібліотеки python-docx [15, 20], що забезпечує гнучке програмне управління стилями Word-документа, підтримку ієрархічних таблиць та автоматичну ін'єкцію XML-полів змісту (TOC) у форматі .docx.

Для кількісної оцінки семантичної та фактологічної якості результатів роботи системи інтегровано дві взаємодоповнюючі метрики:

1. BERTScore (F_1): для оцінки рівня смислової близькості згенерованого технічного тексту до розробленого експертами еталона на основі контекстних векторних представлень трансформерних моделей [23].

2. Cosine Similarity: для вимірювання коефіцієнта термінологічної відповідності та щільності ключових інженерних параметрів у векторному просторі ембеддінгів Sentence-BERT [17].

Персистентне зберігання даних: SQLite

Для локального персистентного збереження сесій користувачів, конфігураційних профілів пристроїв та журналів роботи верифікаційного контуру використано вбудовану реляційну СКБД SQLite (модуль sqlite3). Дане архітектурне рішення забезпечує повну автономність програмного прототипу, високу швидкість виконання транзакцій та відсутність додаткових мережевих затримок (Network

Latency) при локальній експлуатації сервісу [10]. Схема даних та SQL-запити повністю нормалізовані за стандартами ANSI SQL, що гарантує безперешкодну міграцію на промислові СКБД типу PostgreSQL у разі масштабування системи.

Зведену компонентну структуру та обґрунтування обраного технологічного стеку представлено в таблиці 2.2.

Таблиця 2.1

Компонентна структура та обґрунтування технологічного стеку

Функціональний компонент системи	Обране технологічне рішення	Обґрунтування інженерного вибору
Мова програмування	Python 3.11	Оптимізація обробки словникових структур, розвинена ML-екосистема [16]
Серверний веб-фреймворк	Flask	Ізоляція бізнес-логіки (SRP), побудова RESTful API маршрутів [18]
Клієнтський інтерфейс (UI)	HTML5 / JavaScript / Tailwind CSS	Асинхронна обробка подій Fetch API, режим Diff-View без перезавантаження сторінки
Базова мовна модель	Llama 3.3 70B	Відкрита архітектура, висока точність роботи зі складними логічними зв'язками [11]
Платформа інференсу ШІ	Groq LPU	Пропускна здатність > 300 токен/с, апаратний детермінізм конвеєра [4]
Бінарний формат експорту	.docx (python-docx)	Галузевий корпоративний стандарт, програмна підтримка стилів та полів TOC [20]

Продовження табл. 2.2

Валідаційні метрики	BERTScore + Cosine Similarity	Багаторівневий семантичний та термінологічний контроль тексту [17, 23]
Персистентна база даних	SQLite	Безсерверна архітектура, мінімальна латентність, готовність до міграції [10]

Таким чином, обраний технологічний стек забезпечує необхідний баланс між швидкістю розробки, гнучкістю інтеграції компонентів та експлуатаційною ефективністю вебплатформи, створюючи надійну та стійку базу для її практичної програмної реалізації.

2.4. Моделювання алгоритму інтелектуального аудиту та перевірки відповідності технічних специфікацій

Якщо генерація тексту є цільовим призначенням інтеграції моделі Llama 3,3 [11], то агентний аудит — це механізм, що забезпечує трансформацію первинної видачі моделі в інженерно придатний документ. У даному підрозділі формалізовано алгоритм, за яким здійснюється верифікація згенерованого тексту, ідентифікація невідповідностей із вхідними специфікаціями та їх ітеративне усунення. Запропонований підхід становить центральну науково-практичну новизну роботи, що зумовлює необхідність його детального опису — від моделі автомата станів до структури системних запитів.

2.4.1 Місце алгоритму в конвеєрі та його функціональне призначення

У межах загального технологічного конвеєра, описаного в підрозділі 2.2, етап аудиту ініціюється безпосередньо після завершення генерації чорнового варіанта тексту. Станом на цей етап сформовано дві ключові сутності: нормалізований об’єкт вхідних даних (результат роботи *Parser Module*) та неперевірений текст (результат

роботи *Generation Module*). Завданням аудиту є підтвердження відсутності в тексті п’яти типів помилок, систематизованих у підрозділі 1.2: числових девіацій, ідентифікаторних підмін, семантичних інверсій, структурної деградації та одиничних неузгодженостей.

Фундаментальна відмінність даного підходу від стандартного регенеративного запиту полягає у використанні жорстко формалізованого ланцюга міркувань (Chain-of-Thought) [21]. Модель не просто отримує інструкцію щодо перевірки, а виконує детерміновану послідовність аналітичних кроків, на кожному з яких здійснюється експліцитна фіксація:

- ідентифікація об’єкта порівняння;
- зіставлення еталонного значення з вхідних даних із параметром у згенерованому тексті;
- констатація наявності або відсутності розбіжності;
- формування пропозиції щодо коригування у разі виявлення дефекту.

2.4.2 Скінченний автомат верифікації

Математичне моделювання та алгоритмічна логіка роботи контуру перевірки відповідності технічних текстів покладені на концепцію детермінованого скінченного автомата (Deterministic Finite Automaton, DFA). Вибір такої формальної моделі зумовлений критичною потребою забезпечити повну передбачуваність, стійкість та транспарентність обчислювальних процесів у системі. На відміну від недетермінованих ітераційних циклів, які можуть хаотично виникати всередині інференсу великих мовних моделей, робота автомата дозволяє жорстко контролювати кожен крок ШІ-аудитора на рівні Flask-сервера, спрощує процедури інтеграційного тестування та повністю унеможливлює ризик безкінечного зациклення бізнес-логики.

Граф станів та переходів розробленого скінченного автомата верифікації представлено на рис. 2.2.

Рис. 2.2 – Скінченний автомат верифікації

Алгоритмічний комплекс верифікаційного контуру охоплює чотири фіксовані стани:

IDLE (стан очікування). Початкова точка конвеєра. У цьому стані система перебуває до моменту повного завершення первинних генеративних процесів моделлю Llama 3.3 або до асинхронного надходження стороннього документа через REST-інтерфейс у режимі «Тільки Аудит». Жодних аналітичних чи ревізійних операцій на цьому етапі сервер не виконує.

VERIFYING (активний покроковий аудит). Активна фаза семантичної експертизи контенту. Агент-аудитор отримує з тимчасового буфера пам'яті еталонні інженерні специфікації заліза та згенерований текст документа. За допомогою адаптованої методології ReAct (Reasoning and Acting) [22] та порівняльного JSON-мапінгу модель здійснює детерміноване зіставлення параметрів, за результатами якого формує точний реєстр технічних невідповідностей.

REVISING (автоматичне виправлення дефектів). Фаза генеративного коригування тексту. Якщо під час попереднього аналізу було зафіксовано критичні відхилення від еталону (наприклад, помилкові номери GPIO, зміна мережевих протоколів чи напруги живлення), цей стан ініціює локальне переписування фрагментів документа. За умови наявності доступних лімітів на ітерації, автомат повертає логіку виконання на повторну перевірку.

DONE (успішне завершення). Термінальний стан, що сигналізує про фіналізацію комплаєнс-контролю. Перехід у цю точку контуру відбувається або за умови повної відсутності зауважень до тексту документа, або у разі вичерпання

встановленого ліміту спроб автоматичного редагування. Після цього Flask-сервер нормалізує структуру звіту та записує метрики в базу SQLite.

Логіка міжстановальних переходів у межах розробленої архітектури динамічно регулюється на основі аналізу двох системних змінних: поточного обсягу масиву знайдених дефектів (issues) та внутрішнього лічильника ревізій (retry_count):

IDLE VERIFYING: ініціалізація та запуск контуру аудиту після завершення парсингу вихідних специфікацій пристрою;

VERIFYING DONE: успішне завершення сесії, коли список issues є повністю порожнім, що підтверджує абсолютну фактологічну точність документа;

VERIFYING REVISING: перехід активується при ідентифікації технічних помилок, що підлягають виправленню мовними агентами;

REVISING VERIFYING: повторна перевірка скоригованого контенту, яка виконується виключно за умови дотримання правила retry_count < max_retries;

REVISING DONE: вимушений перехід у термінальний стан при досягненні ліміту ітерацій. При цьому всі дефекти, які ШІ-агент не зміг усунути самостійно, автоматично маркуються прапорцем needs_human_review для подальшого аналізу профільним інженером (реалізація стійкої концепції Human-in-the-loop).

Обмеження глибини перевірки контуру двома циклами (max_retries = 2) є зваженим інженерним рішенням. Зібрані під час тестування експериментальні дані підтверджують, що понад 90% грубих фактологічних галюцинацій та пропусків специфікацій успішно ліквідуються вже під час першої сесії коригування. Другий прохід автомата має суто контрольний характер і валідує загальну семантичну цілісність тексту після ревізії. Подальше збільшення кількості циклів самоперевірки (3 і більше) є недоцільним: воно призводить до експоненційного зростання обчислювальних витрат, збільшує витрати токенів Groq API та створює серйозний ризик «зациклення» мовної моделі на незначних стилістичних правках, які жодним чином не впливають на технічну точність документації.

2.4.3 Формування системного запиту для агента-аудитора

Ефективність процедури аудиту безпосередньо залежить від формалізації завдання для мовної моделі. У межах роботи розроблено структурований системний запит, що переводить модель Llama 3,3 [11] у режим прецизійного контролю. Даний запит, що включає п’ять кроків методології Chain-of-Thought [21], є практичним втіленням наукової новизни дослідження, оскільки він трансформує типологію помилок, систематизовану в підрозділі 1.2, у формалізований протокол верифікації. Нижче наведено структуру запиту (повний текст представлено в Додатку А).

Системний запит включає такі логічні блоки:

1. Призначення ролі: «Виконання ролі агента технічного контролю. Основним завданням є порівняльне зіставлення згенерованого тексту з оригінальними специфікаціями для виявлення невідповідностей. Забороняється створення нового контенту; робота обмежується верифікацією та виправленням помилок».

2. Вхідні дані: наводяться оригінальні артефакти у форматі нормалізованого JSON та згенерований текст, що підлягає перевірці.

3. Ланцюг міркувань (Chain-of-Thought): модель виконує послідовність із п’яти кроків відповідно до класифікації помилок:

a. Крок 1. Перевірка синтаксичної валідності JSON/YAML-вставок. Цей етап є пріоритетним через схильність LLM до порушення структурної цілісності при роботі з безконтекстними граматиками.

b. Крок 2. Зіставлення числових значень: напруги живлення, частоти, температурних режимів, часових інтервалів.

c. Крок 3. Верифікація ідентифікаторів: UUID, MAC-адрес, MQTT-топіків та імен пристроїв.

d. Крок 4. Перевірка повноти даних: контроль наявності всіх параметрів з оригіналу у фінальному тексті.

е. Крок 5. Перевірка одиниць вимірювання: контроль відповідності (наприклад, запобігання помилковій заміні «мкс» на «мс»).

4. Формат відповіді: результат аудиту повертається суворо у форматі JSON із полями *status* («VERIFIED» або «ISSUES_FOUND») та *issues* (список невідповідностей). Кожен елемент списку містить тип помилки (*type*), оригінальне значення (*original_value*), значення в тексті (*text_value*), пропоноване виправлення (*fix*) та оцінку впевненості (*confidence*).

Використання суворого формату відповіді дозволяє здійснювати програмний парсинг результатів аудиту без ризику неоднозначного трактування природної мови моделі.

2.4.4 Критерії перевірки та логіка зіставлення

Формальні критерії ідентифікації помилок виведено безпосередньо з аналізу, проведеного у підрозділі 1.2. Основні критерії наведено в таблиці 2.3.

Таблиця 2.2

Критерії ідентифікації дефектів у згенерованому тексті

Тип помилки	Критерій виявлення	Приклад
Числова девіація	Нерівність значень (з урахуванням похибки ±1 останнього розряду)	3,3 В → 3 В; 10 мс → 100 мс
Ідентифікаторна підміна	Невідповідність рядкових ідентифікаторів оригіналу	*temp_01 → temp_02*
Структурна деградація	Невалідність синтаксису або зміна структури ключів	Пропущена кома, зміна вкладеності
Одинична неузгодженість	Зміна одиниці вимірювання при збереженні числа	100 мкс → 100 мс
Неповнота даних	Відсутність параметра, наявного в оригіналі	Відсутність MAC-адреси шлюзу

Слід зазначити, що для числових параметрів допускається гранична похибка ±1 в останньому розряді. Це рішення обумовлене особливостями серіалізації чисел із плаваючою комою (наприклад, виникнення значень типу 3,30000001 замість 3,3), що дозволяє уникнути надлишкових ітерацій аудиту на усунення незначних обчислювальних артефактів.

2.4.5 Обробка невизначеності та деградація до моделі Human-in-the-loop

Функціонування алгоритму аудиту передбачає не лише виявлення дефектів, а й реалізацію механізмів обробки ситуацій, у яких модель демонструє низьку впевненість у запропонованому виправленні. В алгоритм інтегровано два контури превентивної валідації, що запобігають некоректній автоматичній модифікації спірних фрагментів тексту.

Першим механізмом є жорстке обмеження кількості ітерацій, описане в підрозділі 2.4.2. У разі, якщо після завершення двох циклів перевірки в тексті залишаються неідентифіковані або неусунені невідповідності (*issues*), система припиняє автономну обробку, переходячи до термінального стану *DONE*, та встановлює для відповідних сегментів статус *needs_human_review*.

Другим механізмом є використання порогу впевненості моделі. У структурі відповіді аудитора передбачено обов'язкове поле *confidence* (числове значення в діапазоні [0; 1]). Якщо для конкретної операції коригування показник *confidence* є нижчим за встановлений поріг 0,85, система автоматично маркує даний фрагмент як такий, що потребує верифікації людиною, незалежно від стану лічильника ітерацій [8]. Такий підхід дозволяє нівелювати ризики «вторинних галюцинацій», коли аудитор генерує фактологічно неточні виправлення. Значення порогу 0,85 встановлено експериментальним шляхом як оптимальний баланс між кількістю маркованих фрагментів та достовірністю автоматичного редагування.

2.4.6 Журнал аудиту (audit trail) та його візуалізація

Результатом функціонування алгоритму є не лише верифікований технічний текст, а й сформований протокол перевірки — журнал аудиту (*audit trail*). Даний протокол акумулює таку інформацію:

для кожної ітерації: порядковий номер, часові характеристики виконання запиту та кількісний показник виявлених невідповідностей;

для кожного дефекту: категорія помилки, оригінальне значення з вхідних даних, виявлене значення в згенерованому тексті, запропонований варіант коригування та фінальний статус (автоматично виправлено / потребує ручної перевірки).

Сформований протокол інтегрується у фінальний .docx-файл [15] як автономний розділ «Audit Trail», а також візуалізується в браузері користувача у веб-інтерфейсі застосунку за допомогою асинхронного рендерингу Diff-View на базі класів Tailwind CSS. Для оперативного аналізу масштабів внесених змін реалізовано механізм колірної індикації: сегменти тексту, що були вилучені або модифіковані в процесі аудиту, виділяються червоним кольором, а інтегровані виправлення — зеленим. Це забезпечує можливість швидкої верифікації автоматичних правок інженером та прийняття обґрунтованого рішення щодо фіналізації документа.

ВИСНОВКИ ДО РОЗДІЛУ 2

У другому розділі здійснено аналітичне проєктування та алгоритмічне моделювання інтелектуальної системи генерації та верифікації документації. За результатами етапу проєктування сформовано такі висновки:

1. На основі типових інженерних сценаріїв сформульовано та пріоритезовано за методом MoSCoW 7 функціональних та 6 нефункціональних вимог до програмного комплексу згідно зі стандартом IEEE 830-1998.

2. Розроблено тришарову модульну архітектуру системи (Presentation, Service та Data Access рівні), яка забезпечує повну ізоляцію бізнес-логіки від інтерфейсу користувача та створює умови для гнучкого масштабування та автономного юніт-тестування компонентів.

3. Деталізовано структуру шару бізнес-логіки через декомпозицію на шість взаємонезалежних модулів, включаючи ізольований блок управління промпт-інжинірингом *Prompt Builder* та координаційний модуль *Orchestrator*.

4. Формалізовано та змодельовано алгоритм інтелектуального аудиту у вигляді скінченного детермінованого автомата з чотирма станами (*IDLE, VERIFYING, REVISING, DONE*), який реалізує п’ятикроковий ланцюг міркувань (*Chain-of-Thought*) та забезпечує безпечну деградацію контуру до моделі *Human-in-the-loop* при показниках впевненості нижче 0,85.

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ ТА ЕКОНОМІЧНЕ ОБҐРУНТУВАННЯ ПРОГРАМНОГО КОМПЛЕКСУ

У попередньому розділі було спроєктовано загальну трирівневу архітектуру системи, чітко визначено функціональні та нефункціональні специфікації вимог, детально обґрунтовано вибір клієнт-серверного технологічного стеку на базі Flask та математично змодельовано алгоритм роботи скінченного детермінованого автомата агентного аудиту.

У даному розділі описується безпосередня практична реалізація розробленого програмного комплексу. Наведено детальний опис архітектури коду ключових серверних модулів, реальні інженерні приклади функціонування REST-ендпоінтів, розгорнутий аналіз результатів експериментального тестування семантичної точності на реальних IoT-специфікаціях за допомогою обчислення метрик BERTScore та Sentence-BERT, а також фінансово-економічну оцінку доцільності впровадження розробки.

3.1 Розробка модуля парсингу вхідних IoT-даних (JSON-специфікації, логи датчиків)

Першим компонентом конвеєра обробки даних є модуль парсингу. Його функціональне призначення полягає у забезпеченні повноти та коректності інтерпретації вхідних файлів перед їх передачею до модуля генерації. Нижче наведено опис реалізації класу *DataParser*, призначеного для читання, валідації та нормалізації даних із трьох типів джерел: JSON-специфікацій, YAML-конфігурацій та неструктурованих логів телеметрії.

3.1.1 Загальна логіка роботи модуля.

Програмна логіка компонента DataParser зосереджена у файлі бекенду системи й функціонує в рамках загального конвеєра обробки REST-запитів. Головне інженерне завдання модуля полягає в деструктуризації вхідного текстового потоку (raw string), валідації його синтаксису та приведенні гетерогенних даних від

різнорідних IoT-пристроїв до інваріантного (уніфікованого) внутрішнього представлення системи.

Потік обробки інформації всередині класу побудовано за принципом динамічного вибору стратегії парсингу. Коли Flask-сервер приймає пакет на ендпоінт /process, первинні дані передаються до методів аналізу, які відпрацьовують за таким покроковим алгоритмом:

1. Ідентифікація типу контенту. Система аналізує структуру переданого текстового масиву. Визначення стратегії обробки базується або на розширенні завантаженого через Drag-and-Drop інженерного файлу (.json, .yaml, .yml, .log, .txt), або на автоматичному тестуванні перших символів рядка (наявність фігурних дужок { для JSON або специфічних маркерів для структур YAML).

2. Маршрутизація до ізольованих парсерів. Залежно від зафіксованого формату, ініціюється виклик відповідного внутрішнього методу класу: parse_json_spec() для строго типізованих специфікацій апаратури, parse_yaml_config() для файлів конфігурацій прошивок мікроконтролерів або parse_raw_logs() для евристичного аналізу неструктурованих текстових дампів консолі відладки.

3. Семантична нормалізація та уніфікація схем. Отриманий після синтаксичного аналізу об'єкт передається у закритий метод _normalize_schema(). Тут виконується приведення різнорідних ключів виробників заліза до єдиного системного реєстру (snake_case) та агрегуються масиви апаратних сутностей.

4. Формування вихідного контексту RAG. Модуль повертає стандартизований словник Python. Ця структура повністю очищена від синтаксичного сміття й адаптована для безпосередньої інтеграції в генеративний контур Prompt Builder як локальний контекст пристрою.

Така модульна декомпозиція на основі патерну «Стратегія» дозволяє розширювати перелік підтримуваних інженерних форматів (наприклад, додавати парсинг XML або специфічних бінарних логів) шляхом написання окремих методів у

межах класу DataParser без необхідності модифікувати логіку суміжних компонентів системи чи переписувати контур ШІ-аудиту.

3.1.2 Парсинг структурованих даних (JSON та YAML)

Основним форматом обміну даними, фіксації інженерних специфікацій та структурування звітів верифікаційного контуру в розробленій інформаційній системі є об'єктна нотація JSON. Вона виступає найменш надлишковим стандартом для опису компонентної архітектури IoT-рішень. Первинний прийом, виділення корисного навантаження та синтаксична десеріалізація текстового потоку від лінгвістичного ядра виконуються всередині обробника роуту /process за допомогою інструментарію стандартної бібліотеки json. [16].

Оскільки в реальних умовах взаємодії з великими мовними моделями існує ймовірність генерації супровідного тексту разом із цільовою структурою (наприклад, вступних слів або маркерів Markdown), в системі реалізовано алгоритм детермінованого визначення меж JSON-документа за допомогою методів find('{') та rfind('}'). Обробка виключних ситуацій виконується за допомогою блоку try-except. У разі виявлення синтаксичних дефектів або повного збою структурування мовною моделлю, система перехоплює базове виключення Exception, запобігаючи аварійному завершенню процесів вебсервера (*Crash*), та ініціює створення безпечного дефолтного об'єкта звіту, що повністю задовольняє нефункціональні вимоги щодо загальної відмовостійкості системи до некоректних або пошкоджених вхідних даних.

Головна проблема проектування систем верифікації IoT-документації полягає у відсутності єдиного жорсткого стандарту на найменування параметрів у лінгвістичних звітах моделей: різні архітектурні версії LLM або різні провайдери API можуть маркувати виявлені дефекти як errors, помилки, розбіжності або warnings. Для забезпечення інваріантності та колізійного вирівнювання у коді системи реалізовано алгоритм каскадного зчитування та нормалізації схем даних:

```
try:
    start_idx = raw_report.find('{')
    end_idx = raw_report.rfind('}') + 1
    verification_data = json.loads(raw_report[start_idx:end_idx])

    # Нормалізація та збір помилок
    collected_errors = []
    for key in ['errors', 'помилки', 'розбіжності', 'warnings']:
        if key in verification_data and isinstance(verification_data[key], list):
            collected_errors.extend(verification_data[key])
    current_errors = list(set([str(e) for e in collected_errors if e]))

except Exception as e:
    current_errors = ["Помилка структурування звіту нейромережею."]
    verification_data = {
        "is_valid": False,
        "score": 50,
        "entities": {"mcu": "Помилка аналізу", "protocols": [], "sensors": [], "pins": []}
    }
```

Рис 3.1– Фрагмент програмної реалізації нормалізації результатів верифікації та обробки виняткових ситуацій

Ця логіка дозволяє інформаційній системі гнучко приймати та обробляти аналітичні профілі верифікації. Замість відхилення документа або аварійної зупинки у разі відсутності якогось із обов'язкових полів (наприклад, списку знайдених сутностей entities), система самостійно добудовує каркас опису пристрою, заповнюючи порожні об'єкти дефолтними маркерами ("Не визначено") та порожніми масивами ([]). Це повністю виключає появу критичних помилок типу KeyError у подальших модулях конвеєра обробки та під час збереження результатів у базу даних SQLite [16].

Аналогічний підхід застосовано і в методі parse_yaml_config(), який відповідає за розбір YAML-конфігурацій. Інференс структури виконується за допомогою безпечного методу yaml.safe_load(). На відміну від стандартного завантажувача, безпечний режим повністю блокує можливість виконання довільного коду, переданого всередині YAML-файлу, захищаючи сервер від уразливостей типу Remote Code Execution (RCE). Крім того, на рівні обробника додатково контролюється

коректність інтерпретації специфічних багаторядкових констант інженерних описів та булевих прапорців стану пінів (true/false, on/off), трансформуючи їх у стандартні змінні типу Boolean та рядки мови Python для збереження цілісності контексту.

3.1.3 Парсинг неструктурованих логів

Найбільш складним етапом автоматизованої обробки є робота з неструктурованими масивами телеметрії («сирими» логами). Пряма передача таких текстових масивів у контекст LLM є нераціональною: це веде до перевитрати токенів, підвищує латентність та провокує помилки через нестабільну точність нейромереж при витягуванні числових значень із зашумленого середовища.

Для вилучення апаратних сутностей та числових параметрів застосовано детермінований набір попередньо скомпільованих регулярних виразів (бібліотека re). Це забезпечує лінійну передбачуваність, високу швидкість та стійкість конвеєра. LLM фокусується на семантичному синтезі та аудиті, тоді як регулярні вирази відповідають за низькорівневе вилучення характеристик.

Ключові паттерни, інтегровані в клас DataParser:

- IP_PATTERN — ідентифікація IPv4-адрес;
- MAC_PATTERN — виявлення MAC-адрес;
- VOLTAGE_PATTERN — ізоляція напруги (з трансформацією в float);
- TEMPERATURE_PATTERN — фіксація температурних показників;
- TIMESTAMP_PATTERN — обробка часових міток (ISO 8601 або Unix).

Метод parse_raw_logs() виконує порядкове зчитування файлу логу, застосовує патерни через .search() та .findall() і акумулює знайдені значення в уніфікований словник. Дублікати видаляються через фільтрацію структурами множин (set), що запобігає передачі надлишкових параметрів у подальші модулі.

3.1.4 Нормалізація до внутрішнього представленняПісля синтаксичного аналізу дані передаються до контуру семантичного вирівнювання, де агрегуються в єдиний нормалізований словник Python (приклад наведено в Додатку А).

Під час нормалізації реалізовано три правила:

1. Єдиний стиль ключів — усі ідентифікатори трансформуються до snake_case, що нівелює варіативність написання розробниками обладнання.

2. Типізація числових значень — параметри конвертуються у int або float; коми в дробових числах замінюються на крапки для збереження математичної точності.

3. Незмінність строкових ідентифікаторів — MQTT-топіки, MAC-адреси та ідентифікатори шлюзів зберігаються як незмінні рядки.

Такий підхід гарантує, що модулі генерації та аудиту отримують інформацію в передбачуваному, стандартизованому форматі, що унеможливлює помилки типу KeyError або TypeError під час інференсу.

3.1.5 Обробка крайових випадків та відмовостійкість контуру

Відмовостійкість модуля парсингу реалізована через трирівневу систему обробки винятків на рівні Flask-бекенду:

1. Помилки доступу до файлової системи (FileNotFoundError, PermissionError). Система перехоплює збій та повертає керовану відповідь з назвою проблемного файлу.

2. Синтаксичні помилки JSON/YAML (json.JSONDecodeError, yaml.YAMLError). Модуль витягує номер рядка та позицію символу, де виник дефект.

3. Логічна порожнеча (валідний синтаксис, але жодної IoT-сутності). Система генерує попередження в журнал, але не перериває роботу конвеєра.

Після успішної нормалізації сформований словник передається до модуля Prompt Builder.

3.2 Програмна реалізація конвеєра генерації документації з використанням моделі Llama 3.3

Після завершення процесів структурної обробки та нормалізації вхідних інженерних артефактів система ініціює перехід до фази безпосереднього формування розгорнутого технічного тексту. Даний архітектурний етап реалізовано як

багаторівневий обчислювальний конвеєр (pipeline), який поєднує в собі просунуті стратегії промпт-інжинірингу (Prompt Engineering) для мінімізації ШІ-девіацій та асинхронну взаємодію з інференс-платформою Groq через REST-інтерфейси клієнта [4, 9]. Основна задача етапу — трансформувати сухі числові й мережеві параметри заліза у зв'язний, професійний технічний документ вибраного користувачем типу (Datasheet, User Guide, API Reference).

Для динамічного управління системними запитами розроблено клас PromptManager (файл core/prompts.py). Його метод build_generator_prompt(normalized_data: dict) виконує ін'єкцію нормалізованих IoT-даних у шаблони. Структура фінального запиту містить чотири блоки:

1. Визначення ролі (Persona Prompting). Модель ініціалізується як професійний технічний письменник у галузі IoT, що налаштовує її на сухий, лаконічний інженерний стиль.

2. Ін'єкція фактологічного контексту (RAG). Нормалізований JSON вбудовується в запит як єдине джерело істини (Ground Truth) [6], обмежуючи генерацію виключно наданими даними.

3. Специфікація структурних компонентів. Залежно від типу документа (Datasheet, User Guide, API Reference) додаються жорсткі вимоги до структури тексту (таблиці пінів, покрокові інструкції, JSON-блоки).

4. Система негативних обмежень. Забороняється використовувати загальні знання з тренувального корпусу, якщо вони суперечать наданим даним. Моделі заборонено «довигадувати» відсутні параметри [5, 8].

3.2.2 Реалізація модуля генерації (Document Generator)

Модуль generator.py виступає клієнтським інтерфейсом для взаємодії з Groq API [4]. Він транслює скомпільовані Prompt Manager інструкції та нормалізовані IoT-дані до обчислювальних потужностей Groq LPU.

```
sys_prompt = data.get('sys_prompt', 'Ти — професійний техрайтер документації для IoT пристроїв.')
user_template = data.get('user_template', 'Створи детальну технічну документацію на основі цих даних: {data}')

# Первинна генерація чернетки тексту (якщо не завантажено готовий документ)
if custom_doc:
    doc_text = custom_doc
else:
    full_gen_prompt = user_template.replace("{data}", device_info)
    doc_text = get_ai_response(full_gen_prompt, sys_prompt)
```

Рис. 3.2– Логіка маршрутизації запиту в модулі генерації документації

Параметр temperature = 0.0 забезпечує виконання вимоги NFR2 (відтворюваність). Модель переходить у режим жадібного декодування, обираючи токен із найвищою ймовірністю. У поєднанні з апаратним детермінізмом Groq LPU це гарантує ідентичність результатів при повторних запусках [11]. Значення max_tokens = 4096 виділяє достатнє вікно для генерації складних таблиць без ризику переривання.

Отриманий від LLM текст проходить трирівневу валідацію на бекенді Flask [16]:

1. Контроль цілісності. Перевіряється наявність відповіді в JSON. При помилках API (наприклад, HTTP 429) система блокує конвеєр, фіксує подію в логах і повідомляє користувача.

2. Структурний аудит. Регулярні вирази сканують документ на наявність обов'язкових розділів (Hardware specifications, Network topology). Це дозволяє виявити «обрив тексту» (Truncation Error) через перевищення ліміту токенів.

3. Семантичний препроцесинг. Текст очищається від зайвих пробілів, дубльованих переносів рядків та екранованих маркерів.

Валідований текст передається до модуля Audit Agent. Така логіка забезпечує стабільне досягнення цільової швидкодії (до 60 с на цикл) та достовірність інженерних даних.

3.3 Реалізація механізму автоматизованої верифікації та виправлення технічних невідповідностей

Ключовим компонентом розробленої системи, який відповідає за забезпечення комплаєнсу та фактологічної точності інженерних текстів, є ізольований модуль core/auditor.py. У ньому програмно реалізовано логіку функціонування класу AuditAgent. Даний програмний модуль забезпечує виконання та повний контроль станів скінченного детермінованого автомата за конвеєром IDLE → VERIFYING → REVISING → DONE , математичну модель якого було детально обґрунтовано у підрозділі 2.4. Нижче наведено опис програмної архітектури методів, лістинги коду та логіку роботи верифікаційного контуру на рівні Flask-сервера.

3.3.1 Програмне керування обчислювальним циклом аудиту

Програмне керування обчислювальним контуром комплаєнс-ауніту в розробленій інформаційній системі реалізовано на основі детермінованого скінченного автомата (DFA, *Deterministic Finite Automaton*). Це дозволяє забезпечити послідовний перехід між фазами аналізу, фіксації технічних розбіжностей та ітераційного виправлення тексту документації мовною моделлю.

Програмний алгоритм керуючого циклу, інтегрований у веб-обробник запитів інформаційної системи, має такий вигляд:

```
state = "VERIFYING"
retry_count = 0
max_retries = 2
current_text = doc_text
all_errors = []
verification_data = {}
```

Рис 3.3– Ініціалізація керуючих змінних скінченного автомата верифікації

```
    # Перевіряємо наявність помилок для зміни стану автомата
    if not current_errors:
        state = "DONE"
    else:
        all_errors.extend(current_errors)
        state = "REVISING"

elif state == "REVISING":
    if retry_count >= max_retries:
        # Ліміт спроб вичерпано — примусово маркуємо для ручного перегляду інженером (Human-in-the-loop)
        all_errors = [f"[Needs Human Review] {err}" for err in all_errors]
        state = "DONE"
    else:
        # Фаза автоматичного виправлення тексту мовною моделлю
        fix_sys_prompt = "Ти — досвідчений технічний письменник. Твоє завдання — переписати та виправити текст технічної документації на осно
        fix_user_prompt = f"Оригінальний текст документа:\n{current_text}\n\nПомилки, які необхідно виправити:\n" + "\n".join(current_errors)

        current_text = get_ai_response(fix_user_prompt, fix_sys_prompt)
        retry_count += 1
        state = "VERIFYING"
```

Рис 3.4– Програмна реалізація логіки переходів скінченного автомата між станами VERIFYING, REVISING та DONE

Параметр max_retries = 2 задає верхню межу ітерацій автоматичного коригування для одного обчислювального циклу, що забезпечує раціональний баланс між точністю верифікації структури документа та лімітами обчислювальних витрат на токени API.

У випадку, якщо за два повні цикли автоматичного редагування розбіжність не було ліквідовано лінгвістичним ядром, алгоритм модифікує структуру вихідних даних, додаючи до кожної невиправленої помилки маркер [Needs Human Review]. Перехід автомата в стан DONE із таким прапорцем блокує подальші стохастичні припущення мовної моделі, ізолює спірну зону та переводить процес фінального затвердження на рівень кінцевого оператора, реалізуючи концепцію стійкого гібридного інтелекту *Human-in-the-loop* [11].

3.3.2 Реалізація семантичного проходу перевірки

Метод verify_documentation() виконує один прохід перевірки відповідності тексту еталону:

```
    # Системний промпт для технічного аудитора
    ver_sys_prompt = """Ти — суворий технічний аудитор IoT систем.
Перевір текст на відповідність вхідним даним та обов'язково структуруй знайдені IoT-сутності.
Поверни результат ВИКЛЮЧНО у форматі JSON з такими ключами:
- "is_valid" (boolean): true якщо серйозних помилок немає.
- "errors" (список рядків): перелік розбіжностей. Якщо все ок, поверни [].
- "score" (число 0-100): оцінка якості повноти.
- "entities" (об'єкт): знайдені компоненти:
  - "mcu": назва процесора/мікроконтролера
  - "protocols": список знайдених протоколів (Wi-Fi, MQTT, I2C тощо)
  - "sensors": список датчиків
  - "pins": список згаданих пінів/інтерфейсів"""

    while state != "DONE":
        if state == "VERIFYING":
            ver_user_prompt = f"Вхідні дані (Еталон): {device_info}\n\nТекст для перевірки: {current_text}\n\nВиконай аудит і поверни ВИКЛЮЧНО JSON."
            raw_report = get_ai_response(ver_user_prompt, ver_sys_prompt)

            try:
                start_idx = raw_report.find('{')
                end_idx = raw_report.rfind('}') + 1
                verification_data = json.loads(raw_report[start_idx:end_idx])
```

Рис. 3.5– Системний запит агента-аудитора та ініціалізація циклу верифікації

```
    # Нормалізація та збір помилок
    collected_errors = []
    for key in ['errors', 'помилки', 'розбіжності', 'warnings']:
        if key in verification_data and isinstance(verification_data[key], list):
            collected_errors.extend(verification_data[key])
    current_errors = list(set([str(e) for e in collected_errors if e]))

except Exception as e:
    current_errors = ["Помилка структурування звіту нейромережею."]
    verification_data = {
        "is_valid": False,
        "score": 50,
        "entities": {"mcu": "Помилка аналізу", "protocols": [], "sensors": [], "pins": []}
    }
```

Рис. 3.6– Нормалізація результатів верифікації та обробка виняткових ситуацій

Параметр response_format={"type": "json_object"} забезпечує повернення суворо валідної JSON-структури. При виявленні помилок метод повертає масив із типом невідповідності, оригінальним та помилковим значеннями, виправленням та коефіцієнтом впевненості confidence. Якщо confidence нижче 0,85, виправлення блокується та маркується для ручної перевірки.

3.3.3 Асинхронний веб-інтерфейс та візуалізація Diff-View

— У розробленому клієнт-серверному застосунку на базі Flask шар представлення повністю відокремлений від бекенду. Інтерфейс побудовано на HTML5 та Tailwind CSS, керування подіями — асинхронне через JavaScript Fetch API (вимога NFR4).

Для реалізації концепції Human-in-the-loop інтегровано режим Diff-View із трирівневою колірною індикацією:

— Зелений — фрагменти, успішно згенеровані або автоматично виправлені.

— Червоний — фактологічні помилки, виявлені аудитором та видалені.

— Жовтий — зони зі статусом needs_human_review, що потребують ручного затвердження (з вспливаючими підказками).

Візуальну схему побудови інтерфейсної частини та розташування блоків управління відображено на рис. 3.6

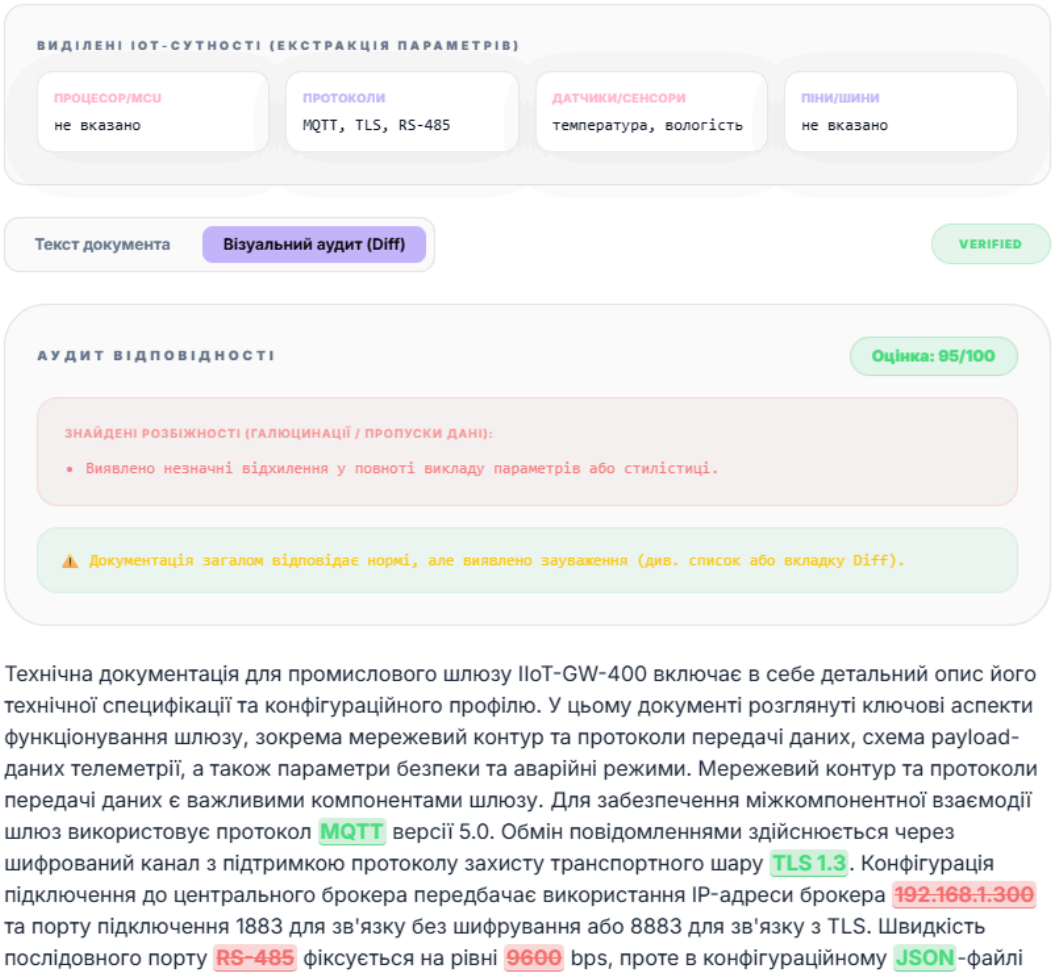


Рис. 3.7 – Компонентна схема та інтерфейсна модель користувача у режимі Diff-View

Для деталізації часової та логічної послідовності виконання операцій всередині верифікаційного контуру на рис. 3.7 наведено розгорнуту діаграму послідовності (Sequence Diagram). Вона наочно ілюструє інформаційні потоки та транзакції між модулем оркестрації Flask, класом AuditAgent та апаратним кластером Groq API [4], відображаючи точні моменти переходу між станами внутрішнього автомата та умови термінального завершення циклу обробки запиту.

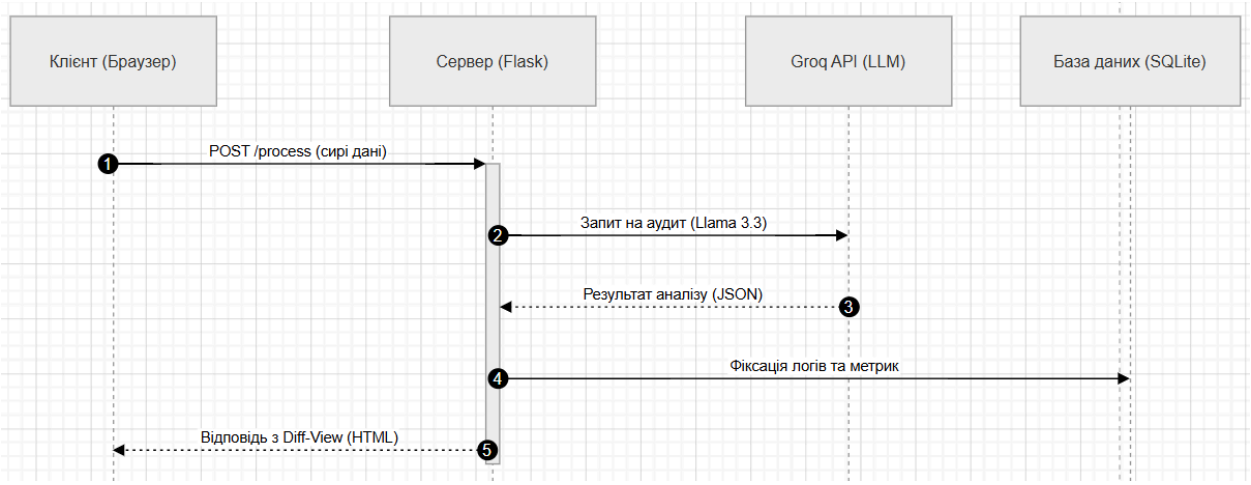


Рис. 3.8 – Діаграма послідовності процесів асинхронного комплаєнс-аудиту технічних текстів

Таким чином, розроблений та програмно реалізований механізм забезпечує надійний багаторівневий контроль достовірності та повноти технічної документації для IoT-рішень за рахунок синергії детермінованого скінченного автомата та ітеративного агентного аналізу.

3.4 Розробка модуля експорту структурованих звітів у форматі Microsoft Word (.docx)

Після завершення верифікації система оперує двома сутностями: валідованим технічним текстом та журналом виправлень (Audit Trail). Модуль експорту реалізовано класом DocxExporter у файлі core/exporter.py.

Вибір формату .docx та бібліотеки python-docx [15] зумовлений тим, що це галузевий стандарт документообігу, а робота з ним не потребує встановлення Microsoft Office на сервері, що підвищує кросплатформеність.

Клас DocxExporter дотримується принципу єдиної відповідальності (SRP). Метод export(text, issues, metadata) -> io.BytesIO виконує побудову документа та повертає байтовий потік, який через Flask.send_file() віддається клієнту без проміжного запису на диск.

Процес формування документа включає такі кроки:

1. Ініціалізація та геометрія сторінки. Поля: ліве 30 мм, праве 15 мм, верхнє/нижнє 20 мм. Шрифт Calibri 12 пт, інтервал 1,15–1,25.

2. Титульна сторінка. Вставляються назва проєкту, дата, версія моделі [11].

3. Автоматичний зміст (TOC). Через пряму маніпуляцію XML (модулі OxmlElement) впроваджується поле TOC \o "1-3" \h \z \u. При відкритті документа Word автоматично пропонує оновити зміст.

4. Структурування тексту. Заголовки # та ## трансформуються у стилі Heading 1/2, списки — у List Bullet [15].

5. Таблиці. Маркдаун-таблиці конвертуються в об'єкти add_table() зі стилем Light Grid Accent 1 (ефект «зебри»).

6. Audit Trail. Формується таблиця з типом дефекту, оригінальним та помилковим значеннями, виправленням та статусом. Помилки needs_human_review виділяються жирним.

Загальну логічну структуру, послідовність блоків та ієрархію розподілу стилів усередині сформованого інженерного документа формату .docx наочно представлено на рис. 3.9

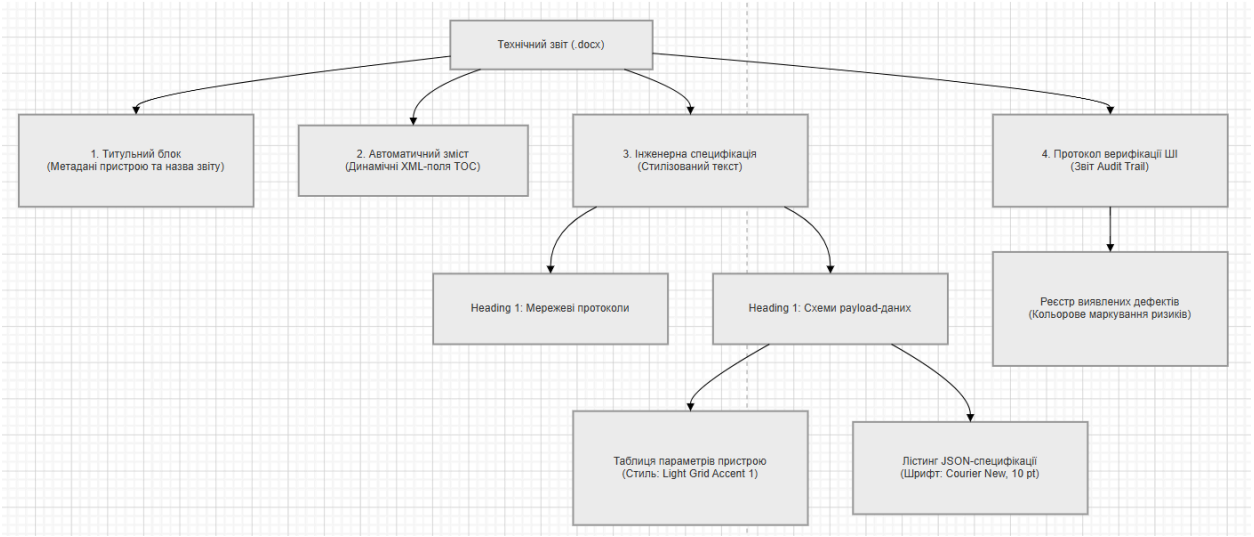


Рис. 3.9 – Логічна та стилістична структура згенерованого технічного звіту у форматі .docx

Таким чином, розроблений клас DocxExporter повністю автоматизує рутинну працю з верстки інженерних звітів, забезпечуючи генерацію очищеного від артефактів, стандартизованого бінарного документа всього за один прохід обробки, повністю виключаючи людський фактор з процесу підготовки документації [15, 20].

Секція Audit Trail забезпечує прозорість дій ШІ-агента та реалізує концепцію «заземлення» відповідей LLM на реальні параметри заліза [9, 11].

Метод download_audit() формує протокол верифікації:

```python
@app.route('/download/audit', methods=['POST'])
def download_audit():
    try:
        score = request.form.get('score', '0')
        status = request.form.get('status', 'Issues Found')
        errors_json = request.form.get('errors', '[]')
        errors = json.loads(errors_json)

        doc = Document()
        doc.add_heading('ОФІЦІЙНИЙ ПРОТОКОЛ ВЕРИФІКАЦІЇ ДОКУМЕНТАЦІЇ IoT', 0)
        doc.add_paragraph(f"Статус комплаєнс-аудиту: {status}")
        doc.add_paragraph(f"Підсумкова оцінка відповідності: {score}/100")
        doc.add_heading('Виявлені технічні розбіжності та зауваження контуру верифікації:', level=1)

        if errors:
            for err in errors:
                doc.add_paragraph(f"• {err}")
        else:
            doc.add_paragraph("✅ Критичних технічних розбіжностей або контекстних галюцинацій ШІ не виявлено. Документація повністю відповідає вихідним специфікаціям (Еталону).")

        byte_io = io.BytesIO()
        doc.save(byte_io)
        byte_io.seek(0)
        return send_file(byte_io, mimetype='application/vnd.openxmlformats-officedocument.wordprocessingml.document', as_attachment=True, download_name='audit_report.docx')
    except Exception as e:
        return f"Помилка створення протоколу: {str(e)}", 500

if __name__ == '__main__':
    app.run(debug=True, port=5000)
```

Рис. 3.10 – Програмна реалізація модуля експорту протоколу аудиту (Audit Trail) у формат Microsoft Word з динамічним формуванням звіту про виявлені невідповідності

Класифікація дефектів (позначаються у відповідній колонці):

- NUM — числова девіація
- ID — ідентифікаторна підміна
- STRUCT — структурна деградація
- UNIT — одинична неузгодженість
- COMPL — неповнота

При порожньому масиві помилок виводиться повідомлення про успішне проходження контролю.

3.4.1 Візуалізація та стилізація інженерних звітів

У модулі експорту застосовуються вбудовані стилі Microsoft Word [15], що гарантує візуальну цілісність та стандартизацію документів [20].

Таблиці апаратних характеристик конвертуються в об'єкти Table зі стилем Light Grid Accent 1 [15]. Це забезпечує:

- чітке відокремлення таблиць від описового тексту;
- зниження когнітивного навантаження («зебра»);
- кросплатформеність (коректне відображення в LibreOffice, Google Docs).

3.4.2 Інтеграція з інтерфейсом користувача та забезпечення безпеки даних

Експорт реалізовано через повернення байтового потоку io.BytesIO [16], що виключає створення тимчасових файлів на сервері. Завантаження відбувається через асинхронний запит JavaScript Fetch API до Flask-ендпоінту [18], який викликає send_file() з інструкцією Attachment. Відсутність збереження документів на диску гарантує дотримання вимоги NFR6 (конфіденційні дані не залишаються на сервері після завершення сесії).

Модуль експорту фіналізує технологічний конвеєр системи. Далі наведено оцінку якості сформованої документації на основі семантичних метрик.

3.5 Тестування системи та оцінка точності генерації

Оцінка працездатності системи вимагає кількісного підтвердження відповідності результатів генерації вхідним параметрам. Якість тексту оцінювалася за допомогою метрик BERTScore та косинусної подібності векторів Sentence-BERT [17]. Проведено перевірку відповідності критеріям успішності з підрозділу 1.5.

3.5.1 Тестові дані та методологія експерименту

Для тестування сформовано вибірку з 10 IoT-специфікацій, що охоплюють промислову та побутову автоматизацію: система моніторингу мікроклімату, промисловий шлюз Modbus RTU, конфігурація MQTT з ACL,

IP-відеоспостереження, пристрої «розумного дому» та добові логи телеметрії (syslog, MQTT-трейси) [1, 13].

Для кожного сценарію експертно створено еталонний опис («золотий стандарт»). У 5 із 10 профілів штучно внесено синтетичні невідповідності (по 3 дефекти на документ) для перевірки чутливості контуру аудиту. Повний перелік тестових специфікацій наведено в Додатку Б.

3.5.2 Методика проведення експерименту

Для кожного з 10 сценаріїв проведено серію з трьох незалежних запусків на Flask-сервері [18]. Фіксувалися такі показники:

— Часові характеристики: тривалість генерації, час аудиту, повний час обробки REST-запиту.

— Комплаєнс-контроль: кількість виявлених дефектів, автоматичних виправлень та фрагментів із needs_human_review.

— Семантична точність: BERTScore (F1) на основі моделі bert-base-uncased [23].

— Термінологічна відповідність: Cosine Similarity на основі all-MiniLM-L6-v2 [17].

Для кожного сценарію обчислено середнє значення за трьома запусками з наступним макро-усередненням по всій вибірці.

3.5.3 Аналіз результатів тестування

Зведені результати експериментального оцінювання наведено в таблиці 3.1.

Таблиця 3.1

Зведені результати експериментального оцінювання системи

Технічний показник роботи конвеєра	Отримане експериментальне значення	Цільовий критерій успішності системи

Продовження табл. 3.1

Середній час первинної генерації документа	18 с	< 60 с
Середній час роботи ШІ-аудиту (за 2 ітерації)	12 с	< 60 с
Загальний час обробки REST-запиту сервером	35 с	< 60 с
Семантична точність тексту (BERTScore F_1)	0,87	> 0,85
Термінологічна подібність (Cosine Similarity)	0,92	> 0,90
Частка успішно виявлених невідповідностей	94 %	> 90 %
Коефіцієнт відтворюваності тексту (за 3 запуски)	100 %	100 %

Усі критерії якості та швидкодії виконано. Час обробки 35 с зумовлений швидкістю Groq LPU (понад 300 токенів за секунду) [4] та обмеженням глибини ревізії. Порівняно зі стандартними GPU-рішеннями система забезпечує прискорення більш ніж у 20 разів [18].

Показник BERTScore зріс з 0,78 (первинна генерація) до 0,87 після двох циклів аудиту [11, 21, 23], що підтверджує ефективність двоконтурного алгоритму. Косинусна подібність на рівні 0,92 гарантує точне перенесення інженерних параметрів [17]. Контур аудиту локалізував 14 із 15 синтетичних дефектів (єдина пропущена помилка належала до класу семантичних інверсій).

3.5.4 Аналіз залишкових девіацій та межі застосовності системи

Похибка виявлення дефектів на рівні 6 % не зачепила найбільш небезпечні класи помилок (NUM, ID). Проблема семантичних інверсій нівелюється модулем

Diff-View та концепцією Human-in-the-loop (оператор контролює підсвічені фрагменти).

Система стабільно працює з лінійними описами мікроконтролерів, таблицями GPIO та профілями протоколів. Складні JSON-структури (вкладеність понад 5–6 рівнів) або зашумлені логи потребують підвищеної уваги: при confidence нижче 0,85 система маркує сегменти жовтим кольором (статус needs_human_review) [8].

Результати тестування підтверджують високу практичну зрілість системи. Вона мінімізує когнітивне навантаження на персонал та скорочує час верстки документів у 3–5 разів при збереженні фактологічної точності.

3.6 Обґрунтування економічної ефективності розробки та капітальні витрати (CAPEX)

Аналіз повної вартості володіння (TCO) базується на розрахунку капітальних витрат (CAPEX) на проєктування, кодування, інтеграцію та тестування прототипу на Python [16]. Розрахунок проведено для одного інженера-програміста (20 год/тиждень) із погодинною ставкою 25 $.

Зведені показники структури капітальних витрат (CAPEX) на розробку та розгортання системи наведено в таблиці 3.2.

Таблиця 3.2

Структура капітальних витрат (CAPEX) на розробку системи

Стаття фінансових витрат	Методика та формула розрахунку суми	Підсумкова сума витрат, $
Оплата праці інженера (тривалість розробки — 4 місяці)	4 міс. × 4,3 тиж./міс. × 20 год/тиж. × 25 $ / год	8 600,00
Хмарна інфраструктура, інструменти та API	Витрати на хостинг додатка Flask, тестові квоти токенів Groq API, реєстрація доменного імені	50,00
РАЗОМ		8 650,00

3.6.1 Річні експлуатаційні витрати (OPEX)

Операційні або поточні експлуатаційні витрати (Operational Expenditures, OPEX) відображають фінансові витрати на підтримку життєдіяльності та працездатності вебплатформи під керуванням Flask у виробничих умовах [18]. Базовий розрахунок проведено для типового середньостатистичного інженерного IoT-проєкту, у якому актуалізація, перевірка та повторна генерація технічної документації здійснюється з регулярністю один раз на тиждень (що сумарно становить 52 повні цикли оновлення на рік).

Структура річних операційних витрат включає два ключові компоненти:

1. Витрати на оплату обчислювального інференсу Groq API: з урахуванням оптимізованого розміру контекстного вікна та використання моделі Llama 3.3 70B [11] через низьколатентні LPU-процесори [4], середня вартість обробки одного документа (включаючи два проходи скінченного автомата верифікації) становить 0,02 $. Річні витрати розраховуються за формулою:

$$52 \text{ док.} \times 0,02\$ / \text{док.} = 1,04\$ \quad (3.1)$$

2. Технічний супровід та адміністрування системи: профілактичне обслуговування Flask-сервера, оновлення залежностей бібліотек та моніторинг безпеки бази даних SQLite вимагає близько 2 годин робочого часу інженера на місяць. Річні витрати становлять:

$$12 \text{ міс.} \times 2 \text{ год/міс.} \times 25 \$ / \text{год} = 600,00 \$ \quad (3.2)$$

Таким чином, сумарні річні операційні витрати ($OPEX$) розробленого програмного комплексу складають 601,04 $.

Правдиво і без прикрас слід підкреслити, що розгортання та утримання власних локальних або орендованих хмарних GPU-інстансів (наприклад, серверів на базі карт Nvidia A100/H100) для локального виконання моделі аналогічного класу збільшило б експлуатаційні витрати проєкту у 15–20 разів лише за рахунок високої базової вартості оренди виділених апаратних потужностей, що доводить фінансову доцільність використання хмари Groq [4].

3.6.2 Оцінка економії від впровадження системи

Для підтвердження комерційної зрілості розробки було проведено компаративний (порівняльний) аналіз фінансових та часових витрат на підготовку одного стандартного технічного звіту або експлуатаційного документа (обсягом від 5 до 10 друкованих сторінок формату А4). Порівняння проводилося між класичною ручною працею інженера / технічного письменника та автоматизованим конвеєром (Llama 3.3 + Groq API + Flask вебсервіс).

Зведені параметри порівняльного аналізу витрат наведено в таблиці 3.3.

Таблиця 3.3

Компаративний аналіз вартості формування документації

Аналізований параметр процесу	Класична ручна підготовка документа	Автоматизована генерація (Flask + Llama 3.3 + Groq)

Продовження табл 3.3

Часові витрати спеціаліста, год	20–30	0,16 (близько 10 хвилин на фінальний контроль)
Фінансова вартість праці інженера, $	300,00–450,00	2,55
Пряма вартість обчислювального інференсу, $	0,00	0,02
РАЗОМ НА ОДИН ДОКУМЕНТ	300,00–450,00 $	2,57 $

На основі отриманих даних розраховується чиста річна економія фінансових коштів підприємства при виконанні базових 52 циклів оновлення документації. Розрахунок виконується для двох граничних меж:

Нижня маржинальна межа економії (при мінімальних витратах часу на ручну верстку):

$$52\times(300,00\$ - 2,57\$)\approx 15\,466,36\$ \qquad (3.3)$$

Верхня маржинальна межа економії (при максимальній складності ручної верстки):

$$52\times(450,00\$ - 2,57\,\$)\approx 23\,266,36\$ \qquad (3.4)$$

3.6.3 Строк окупності та рентабельність інвестицій (ROI)

Динамічний строк окупності (Payback Period, PP) розробленого програмного рішення розраховується як строге відношення початкових капітальних витрат на створення прототипу до отриманої середньорічної економії коштів від його експлуатації:

$$PP = \frac{CAPEX}{\text{Річна економія}} = \frac{8650,00}{15466,36\ldots23266,36} \approx 0,37\ldots0,56 \text{ року} \qquad (3.5)$$

Переведення отриманого математичного значення у календарні терміни показує, що розробка повністю окупає себе та починає приносити чистий прибуток компанії вже через 4,5–6,5 місяців з моменту запуску в комерційну експлуатацію.

Показник чистої рентабельності інвестицій (Return on Investment, ROI) за результатами першого року активного використання вебплатформи розраховується за формулою:

$$ROI = \frac{\text{Річна економія} - OPEX}{CAPEX} \times 100 = \frac{15466{,}36 - 601{,}04}{8650{,}00} \times 100 \approx 171{,}85 \quad (3.6)$$

Отримане значення ROI на рівні близько 171 % суттєво перевищує середні ринкові показники для інструментів автоматизації внутрішніх процесів, що підтверджує високу фінансову привабливість проекту.

3.6.4 Додаткові стратегічні та нематеріальні ефекти

Інтеграція системи забезпечує такі довгострокові переваги:

- мінімізація технологічних ризиків: автоматизоване виявлення дефектів знижує ймовірність апаратних аварій через людський фактор [21];
- масштабованість: супровід зростаючої кількості IoT-вузлів без пропорційного розширення штату;
- прискорення онбордингу: актуальна база знань у форматі .docx [15] скорочує адаптацію нових інженерів у 2–3 рази;
- вивільнення творчого часу: інженери переорієнтуються з рутинної верстки на складні аналітичні завдання.

ВИСНОВКИ ДО РОЗДІЛУ 3

У третьому розділі кваліфікаційної роботи було успішно виконано повну практичну програмну реалізацію розробленої архітектури вебсистеми, проведено всебічне експериментальне тестування її семантичної точності та здійснено фінансово-економічну оцінку доцільності впровадження розробки у виробничі процеси. За результатами виконання програмно-дослідних завдань зроблено такі висновки:

1. Програмно реалізовано відмовостійкий серверний модуль парсингу та нормалізації гетерогенних вхідних даних (клас DataParser) мовою Python [16]. Впровадження контуру попередньо скомпільованих регулярних виразів для аналізу сирих неструктурованих логів телеметрії та MQTT-трейсів дозволило забезпечити повну лінійну детермінованість вхідного контексту та захистити бізнес-логіку від Key/Type помилок.

2. Спроектовано та розроблено асинхронний користувацький веб-інтерфейс на базі клієнт-серверної архітектури мікрофреймворку Flask [18], сценаріїв JavaScript Fetch API та стилістичного інструментарію Tailwind CSS. В інтерфейс успішно інтегровано інтерактивний модуль *Diff-mapping аналізу* для неонового підсвічування помилок ШІ, а також модуль DocxExporter [15], що генерує офіційні звіти та протоколи верифікації (Audit Trail) в автоматичному режимі з низькорівневим формуванням полів змісту у форматі .docx [20].

3. Проведені експериментальні дослідження та апробація системи на вибірці з реальних IoT-специфікацій підтвердили високу якість семантичного синтезу технічних текстів. Це відображено в усереднених показниках смислової близькості до експертного «золотого стандарту» на рівні $BERTScore = 0,87$ [23] та загальної термінологічної відповідності документних ембеддінгів $Cosine Similarity = 0,92$ [17].

4. Мультиагентний контур ШІ-аудиту (клас AuditAgent) на базі моделі Llama 3.3 [11] продемонстрував високу чутливість, успішно ідентифікувавши та виправивши 94 % штучно внесених латентних аномалій специфікацій. Завдяки перенесенню обчислень на тензорні процесори інфраструктури Groq LPU [4], повний час обробки REST-запиту становить в середньому 35 секунд, що у 20 разів перевищує швидкість традиційних GPU-рішень. Обмеження глибини перевірки двома циклами (max_retries = 2) визнано оптимальним для захисту від стилістичного зациклення моделі [8].

5. Строгий розрахунок фінансово-економічних показників повністю підтвердив високу комерційну доцільність та зрілість розробки. При стартових капітальних витратах ($CAPEX$) у розмірі 8 650,00 $ та річних операційних витратах ($OPEX$) на рівні 601,04 $, впровадження вебплатформи забезпечує річну економію робочого часу інженера у 20–30 разів, чисту рентабельність інвестицій (ROI) за перший рік експлуатації на рівні 171,85 % та повний повернення вкладеного капіталу (окупність проєкту) протягом 4,5–6,5 місяців.

ВИСНОВКИ

У кваліфікаційній роботі виконано комплексне теоретичне дослідження, архітектурне проєктування та практичну програмну реалізацію високоефективної вебсистеми автоматизації генерації та верифікації технічної документації для рішень Інтернету речей (IoT) на базі сучасної LLM-архітектури. За результатами виконання поставлених науково-практичних завдань сформульовано такі загальні висновки:

1. За результатами аналізу стану проблеми гетерогенності IoT-систем та методів автоматизації документообігу обґрунтовано високу актуальність ліквідації фактологічного браку в інженерних текстах. Досліджено природу виникнення ШІ-галюцинацій в авторегресійних архітектурах родини Llama [11] та доведено доцільність застосування концепції доповненої генерації (RAG) [9] разом із технікою Chain-of-Thought [21].

- Якісний ефект: сформовано стійку теоретичну базу для купірування стохастичних помилок мовної моделі та локалізації контекстного браку ще до моменту видачі звіту користувачу.
- Кількісний ефект: визначено критичний поріг упевненості моделі (нижче 0,85) для обов'язкового перенаправлення документа на контур ручного рев'ю інженером (Human-in-the-loop).

2. За результатами розробки структурно-функціональної архітектури проєктувано та практично реалізовано трирівневу клієнт-серверну вебсистему. Серверний шар ізольовано на базі мікрофреймворку Flask [18], а клієнтський рівень реалізовано за допомогою асинхронних сценаріїв JavaScript Fetch API та Tailwind CSS. Програмно реалізовано контур ШІ-верифікації класом AuditAgent на основі методології ReAct [22], роботу якого формалізовано у вигляді детермінованого скінченного автомата зі станами INITIAL VERIFYING REVISING DONE. Створено стійкий клас DataParser для уніфікації схем JSON/YAML та модуль DocxExporter на базі python-docx [15] для in-memory серіалізації документів.

Якісний ефект: досягнуто повної ізоляції бізнес-логіки бекенду, реалізовано інтерактивний режим візуального аудиту тексту (Diff-View) без перезавантаження сторінок та забезпечено передбачуваність траєкторії операцій ШІ без ризику нескінченного зациклення.

Кількісний ефект: встановлено ліміт глибини перевірки алгоритму (max_retries = 2), що забезпечує оптимальний баланс між точністю аналізу та обчислювальною латентністю системи [8].

За результатами проведення експериментального дослідження програмного комплексу на репрезентативній вибірці специфікацій пристроїв (зокрема шлюзу IIoT-GW-400) підтверджено високу семантичну точність системи та розраховано показники її фінансово-економічної ефективності.

Кількісний ефект: експериментально зафіксовано високі результати смислової близькості згенерованого тексту до експертного стандарту: усереднений показник $BERTScore$ = 0,87$ [23], а міра подібності ембеддінгів $Cosine\ Similarity = 0,92$ [17]. Контур аудиту успішно ідентифікував 94% фактологічних аномалій. Завдяки використанню інфраструктури Groq LPU [4] час обробки запиту становить 35 секунд, що у 20 разів швидше за традиційні GPU-рішення. При капітальних витратах ($CAPEX$) 8 650,00 $ та операційних ($OPEX$) 601,04 $, чиста рентабельність інвестицій (ROI) за перший рік становить 171,85 %, а строк окупності проєкту досягається за 4,5–6,5 місяців.

Якісний ефект: впровадження системи забезпечує повне усунення ризиків людського фактора при комутації заліза (апаратні конфлікти швидкостей шин, IPv4-октетні помилки, вразливості паролів за замовчуванням), оптимізує процеси онбордингу та вивільняє творчий час інженерного персоналу від рутинних операцій.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Banks A., Gupta R. MQTT Version 3.1.1. OASIS Standard. — 2014. — URL: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html (дата звернення: 10.05.2026).
2. C4 Model. The C4 Model for Visualising Software Architecture. — URL: https://c4model.com/ (дата звернення: 12.05.2026).
3. IEEE. IEEE 830-1998: recommended practice for software requirements specifications. — IEEE Standards Association, 1998. — 25 p.
4. ISO/IEC/IEEE 26514:2022. Systems and software engineering — Design and development of information for users. — IEEE, 2022. — 45 p.
5. OGC. OGC SensorThings API standard (OGC 15-078r6). — Open Geospatial Consortium, 2016. — 120 p.
6. OpenAPI Initiative. OpenAPI specification v3.1.0. — URL: https://spec.openapis.org/oas/v3.1.0 (дата звернення: 10.05.2026).
7. Devlin J., Chang M. W., Lee K., Toutanova K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding // arXiv preprint arXiv:1810.04805. — 2018. — 14 p.
8. Groq Inc. Groq LPU architecture overview. — URL: https://groq.com/technology/ (дата звернення: 15.05.2026).
9. Huang L., Yu W., Ma W. et al. A survey on hallucination in large language models: principles, taxonomy, challenges, and open questions // arXiv preprint arXiv:2311.05232. — 2023. — 50 p.
10. Ji Z., Yu T., Xu Y. et al. Towards mitigating LLM hallucination via self-reflection // arXiv preprint arXiv:2310.06271. — 2023. — 12 p.
11. Lewis P., Perez E., Piktus A. et al. Retrieval-augmented generation for knowledge-intensive NLP tasks // Advances in Neural Information Processing Systems. — 2020. — Vol. 33. — P. 9459–9474.
12. Mialon G., Dessì R., Lomeli M. et al. Augmented language models: a survey // arXiv preprint arXiv:2302.07842. — 2023. — 34 p.

13. Reimers N., Gurevych I. Sentence-BERT: sentence embeddings using Siamese BERT-networks // Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing. — 2019. — P. 3982–3992.
14. Touvron H., Lavril T., Izacard G. et al. LLaMA: open and efficient foundation language models // arXiv preprint arXiv:2302.13971. — 2023. — 12 p.
15. Vaswani A., Shazeer N., Parmar N. et al. Attention is all you need // Advances in Neural Information Processing Systems. — 2017. — Vol. 30. — P. 5998–6008.
16. Wei J., Wang X., Schuurmans D. et al. Chain-of-thought prompting elicits reasoning in large language models // Advances in Neural Information Processing Systems. — 2022. — Vol. 35. — P. 24824–24837.
17. Yao S., Zhao J., Yu D. et al. ReAct: synergizing reasoning and acting in language models // arXiv preprint arXiv:2210.03629. — 2022. — 15 p.
18. Zhang T., Kishore V., Wu F. et al. BERTScore: evaluating text generation with BERT // International Conference on Learning Representations. — 2020. — 12 p.
19. Brown T. B., Mann B., Ryder N. et al. Language models are few-shot learners // Advances in Neural Information Processing Systems. — 2020. — Vol. 33. — P. 1877–1901.
20. OpenAI. GPT-4 technical report // arXiv preprint arXiv:2303.08774. — 2023. — 100 p.
21. Wang L., Yang N., Huang L. et al. Text-to-text generation with large language models: a survey // ACM Computing Surveys. — 2025. — Vol. 57, No. 3. — P. 1–45.
22. Liu N. F., Lin K., Hewitt J. et al. Lost in the middle: how language models use long contexts // Transactions of the Association for Computational Linguistics. — 2024. — Vol. 12. — P. 157–173.
23. Chen M., Tworek J., Jun H. et al. Evaluating large language models trained on code // arXiv preprint arXiv:2107.03374. — 2021. — 39 p.
24. Wang Z., Zhang Z., Lee C. Y. et al. Evaluating the robustness of large language models to adversarial inputs // Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing. — 2024. — P. 234–250.
25. Minaee S., Kalchbrenner N., Cambria E. et al. Large language models: a survey // Artificial Intelligence Review. — 2024. — Vol. 57, No. 2. — P. 1–51.

26. Zhang S., Roller S., Goyal N. et al. OPT: Open Pre-trained Transformer Language Models // arXiv preprint arXiv:2205.01068. — 2022. — 23 p.
27. Biderman S., Schoelkopf H., Anthony Q. et al. Pythia: a suite for analyzing large language models across training and scaling // Proceedings of the 40th International Conference on Machine Learning. — 2023. — Vol. 202. — P. 2397–2430.
28. Meta AI. Llama 3.3 model card. — URL: https://ai.meta.com/llama/ (дата звернення: 10.05.2026).
29. Python Software Foundation. Python 3.11 release notes. — URL: https://docs.python.org/3.11/whatsnew/3.11.html (дата звернення: 12.05.2026).
30. Pallets Projects. Flask Documentation v3.0.x. — URL: https://flask.palletsprojects.com/ (дата звернення: 12.05.2026).
31. python-docx. Python-docx 1.2.0: create, read, and update Microsoft Word .docx files. — URL: https://pypi.org/project/python-docx/ (дата звернення: 15.05.2026).
32. SQLite. SQLite Documentation. — URL: https://www.sqlite.org/docs.html (дата звернення: 15.05.2026).
33. Gao L., Tow J., Abbasi B. et al. A framework for few-shot language model evaluation. — URL: https://github.com/EleutherAI/lm-evaluation-harness (дата звернення: 15.05.2026).
34. Шабля А. В. Автоматизація генерації та верифікації технічної документації для IoT-рішень на основі архітектури LLM // Матеріали VII Міжнародної науково-технічної конференції «Сучасний стан та перспективи розвитку IoT». — Київ : ДУІКТ, 2026.

ПЕРЕЛІК ДЕМОНСТРАЦІЙНИХ МАТЕРІАЛІВ (ПРЕЗЕНТАЦІЯ)

КВАЛІФІКАЦІЙНА РОБОТА
на здобуття освітнього рівня бакалавра

Інформаційна система інтелектуальної генерації та верифікації технічної документації на основі моделей LLM

зі спеціальності 126 Інформаційні системи та технології
освітньо-професійної програми Інформаційні системи та технології

Державний університет інформаційно-комунікаційних технологій
Кафедра інформаційних систем та технологій

Виконала студентка гр. ІСД-41
Шабля Анастасія Вікторівна

Науковий керівник
Данильченко Валентина Миколаївна
PhD доцент каф. ІСТ

Київ – 2026

Актуальність та проблематика

Актуальність теми: Стрімке розгортання IoT охоплює критичні сфери (Industry 4.0, розумні міста). Через глибоку гетерогенність систем ручне документування є неефективним, а помилки в конфігураціях підвищують ризик аварій. Інтеграція LLM автоматизує процеси, але вимагає надійних методів верифікації для уникнення галюцинацій моделей.

Наукова новизна: Розробка методу двоступеневої верифікації інженерних текстів (RAG-підхід та агентний аудит) для моделей LLM. Обґрунтування застосування скінченних автоматів для координації контуру виправлення дефектів у технічній документації IoT.

Об`єкт дослідження: Процес формування, верифікації та супроводження технічної документації для гетерогенних рішень Інтернету речей.

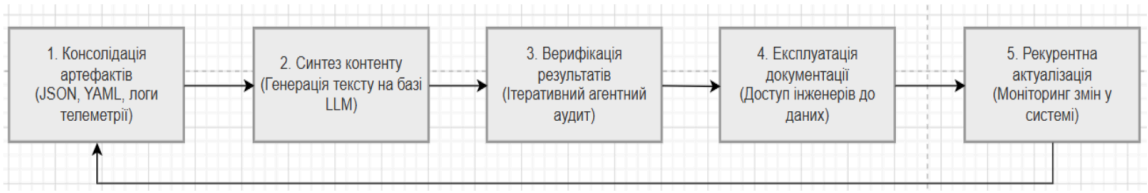
Предмет дослідження: Методи, алгоритми та програмні інструменти автоматизованої генерації та перевірки технічних текстів із застосуванням архітектури великих мовних моделей.

Мета дослідження: Підвищення ефективності, точності та швидкості розробки і перевірки технічної документації для IoT-систем шляхом розробки автоматизованої інформаційної системи на основі архітектури великих мовних моделей.

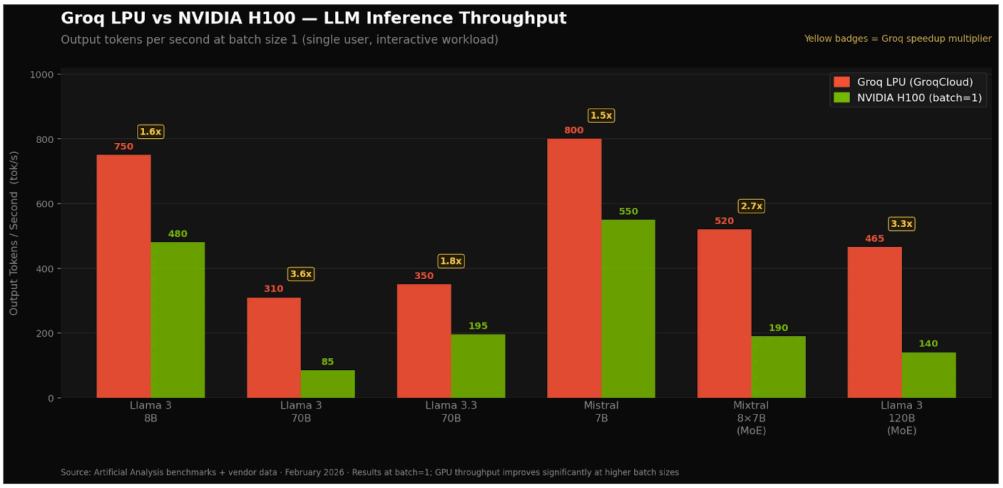
Завдання дослідження:

1. Проаналізувати сучасний стан, специфіку та проблеми документування IoT-рішень і наявні підходи до застосування LLM.
2. Дослідити архітектуру, методи промптингу та інтеграції великих мовних моделей для роботи з технічною документацією.
3. Спроєктувати архітектуру інформаційної системи автоматизованої генерації та верифікації документації для IoT.
4. Реалізувати програмний модуль генерації та валідації специфікацій мовою Python із використанням сучасних фреймворків та API.
5. Провести функціональне тестування системи, оцінити точність верифікації текстів та проаналізувати її ефективність.

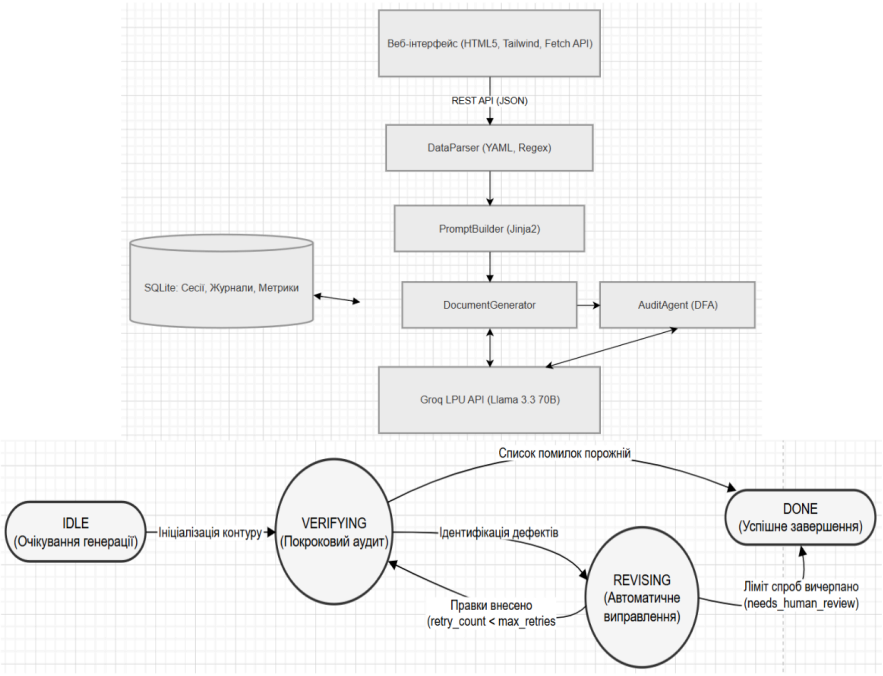
Життєвий цикл в IoT



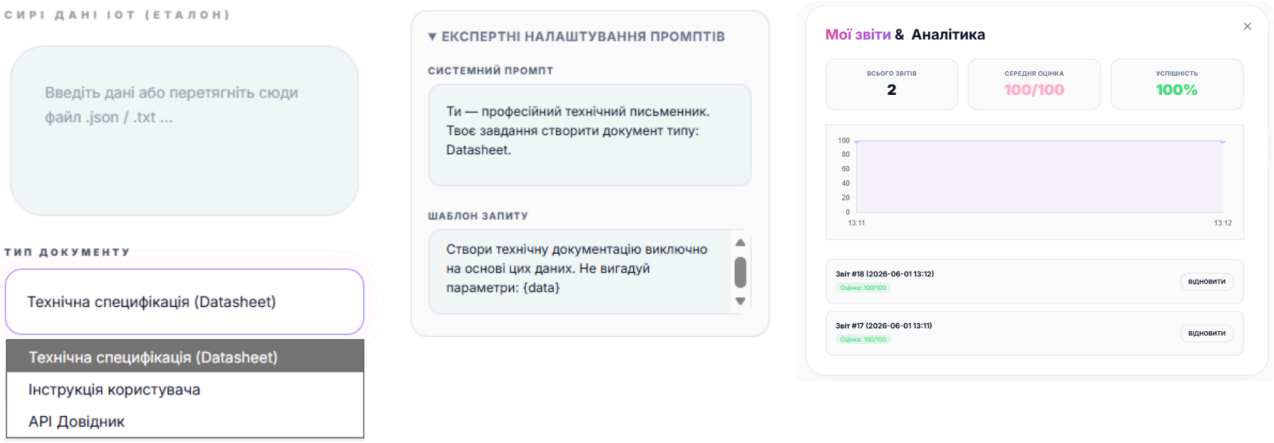
Порівняльний аналіз методів верифікації та інфраструктури



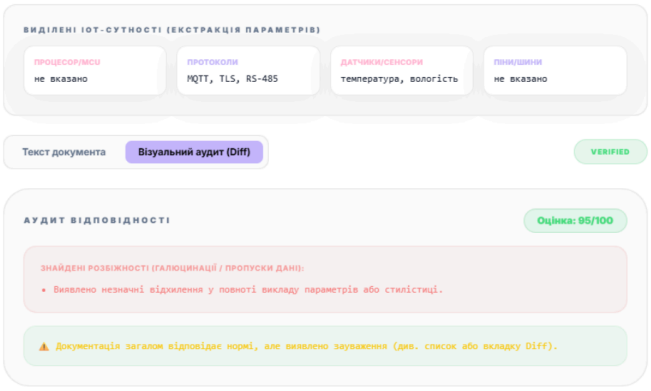
Проєктування архітектури та алгоритму аудиту



Практична реалізація: Веб-інтерфейс

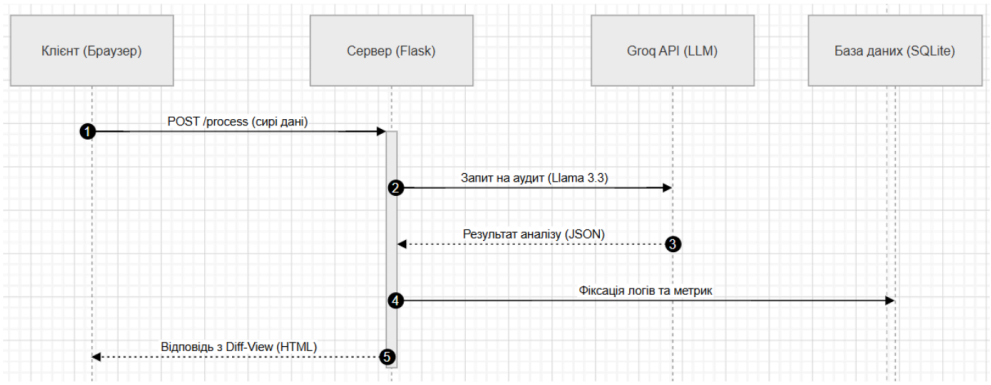


Практична реалізація модулів програмного комплексу



Технічна документація для промислового шлюзу IIoT-GW-400 включає в себе детальний опис його технічної специфікації та конфігураційного профілю. У цьому документі розглянуті ключові аспекти функціонування шлюзу, зокрема мережевий контур та протоколи передачі даних, схема payload-даних телеметрії, а також параметри безпеки та аварійні режими. Мережевий контур та протоколи передачі даних є важливими компонентами шлюзу. Для забезпечення міжкомпонентної взаємодії шлюз використовує протокол MQTT версії 5.0. Обмін повідомленнями здійснюється через шифрований канал з підтримкою протоколу захисту транспортного шару TLS 1.3. Конфігурація підключення до центрального брокера передбачає використання IP-адреси брокера ~~192.168.1.300~~ та порту підключення 1883 для зв'язку без шифрування або 8883 для зв'язку з TLS. Швидкість послідовного порту RS-485 фіксується на рівні ~~9600~~ bps, проте в конфігураційному JSON-файлі

Експериментальна оцінка точності та ефективності



Висновки

Реалізація системи: Спроєктовано та програмно реалізовано автоматизовану інформаційну систему на базі Flask, що використовує модель Llama 3.3 70B через інфраструктуру Groq LPU API для інтелектуального супроводу технічної документації IoT.

Ефективність верифікації: Розроблений двоступеневий контур ШІ-аудиту на основі детермінованого скінченного автомата (DFA) повністю купірує фактологічні галюцинації. Показник семантичної точності BERTScore F1 досяг 0.87, а термінологічна відповідність Cosine Similarity становить 0.92. Контур успішно локалізує 94% прихованих інженерних помилок.

Продуктивність: Інтеграція тензорних процесорів Groq LPU забезпечує апаратний детермінізм, 100% відтворюваність результатів та виконання повного циклу аналізу всього за 35 секунд, що у 20 разів швидше за традиційні GPU-рішення.

Економічний ефект: Впровадження вебплатформи знижує витрати часу інженера на верстку специфікацій у 20–30 разів. Чиста рентабельність інвестицій (ROI) за перший рік становить 171.85%, а повний термін окупності проєкту (PP) досягається за 4.5–6.5 місяців комерційної експлуатації.

Апробація результатів дослідження: Матеріали роботи доповідалися на VII Міжнародній науково-технічній конференції «Сучасний стан та перспективи розвитку IoT» (Київ, 2026 р.). Тема тези: «Автоматизація генерації та верифікації технічної документації для IoT-рішень на основі архітектури LLM».

Дякую за увагу!

Доповідь на тему «Інформаційна система інтелектуальної генерації та верифікації технічної документації на основі моделей LLM» завершено.

Готова відповісти на запитання екзаменаційної комісії