МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЗАПОРІЗЬКА ПОЛІТЕХНІКА»

Факультет комп'ютерних наук та технологій
Кафедра комп'ютерних систем та мереж

Пояснювальна записка

до дипломного проєкту (роботи)
магістра
(ступінь вищої освіти)

на тему **КОМП'ЮТЕРНА СИСТЕМА ДЛЯ ПЕРЕВІРКИ ТА ВИПРАВЛЕННЯ ПОМИЛОК В ТЕКСТІ З ВИКОРИСТАННЯМ ШТУЧНОГО ІНТЕЛЕКТУ**

Виконав: студент 2 курсу, групи КНТ-514м
спеціальності
123 Комп'ютерна інженерія
(код і найменування спеціальності)

Освітня програма (спеціалізація)
Комп'ютерні системи та мережі

ВЕРТМАН М.А.
(ПРІЗВИЩЕ та ініціали)
Керівник СКРУПСЬКИЙ С.Ю.
(ПРІЗВИЩЕ та ініціали)
Рецензент ЩЕРБАКОВ В.І.
(ПРІЗВИЩЕ та ініціали)

2025

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет Комп'ютерних наук і технологій
Кафедра «Комп'ютерні системи та мережі»
Ступінь вищої освіти магістерський
Спеціальність 123 Комп'ютерна інженерія
(код і найменування)

Освітня програма (спеціалізація) Комп'ютерні системи та мережі
(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ
Зав. кафедри Кудерметов Р.К.
_____
"__15__" _жовтня___2025 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА

ВЕРТМАНА Миколи Андрійовича
(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Комп'ютерна система для перевірки та виправлення помилок в тексті з використанням штучного інтелекту
керівник проєкту (роботи) кан. техн. наук, доцент, СКРУПСЬКИЙ Степан Юрійович
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)
затверджені наказом вищого навчального закладу від " 30 " жовтня 2025 року №
2. Строк подання студентом проєкту (роботи) 23 грудня 2025 року
3. Вихідні дані до проєкту (роботи) мови програмування Python та TypeScript, фреймворки Flask та Next.js, документація OpenAI API, тестові набори текстових даних українською мовою.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1) Аналіз предметної області
2) Розробка підходу до обробки помилок та забезпечення надійності
3) Розробка системи;
4) Дослідження системи;

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Листи презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4	СКРУПСЬКИЙ С.Ю., доцент		
нормоконтроль	ЩЕРБАК Н.В., ст. викл.		

7. Дата видачі завдання 01 жовтня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Аналіз предметної області та вимог до розробки	01.10.2025 р.	
2	Розробка методики аналізу AI моделей	3.11.2025 р.	
3	Проведення дослідження	10.11.2025 р.	
4	Розробка системи	15.11.2025 р.	
5	Дослідженні системи	25.11.2025 р.	
6	Оформлення отриманих результатів у ПЗ	05.12.2025 р.	
7	Оформлення графічного матеріалу	10.12.2025 р.	
8	Оформлення допоміжного матеріалу	15.12.2025 р.	

Студент _____ **Микола ВЕРТМАН**
(підпис) (ім'я, ПРІЗВИЩЕ)

Керівник проєкту (роботи) _____ **Степан СКРУПСЬКИЙ**
(підпис) (ім'я, ПРІЗВИЩЕ)

РЕФЕРАТ

ПЗ: 104 с., 37 рис., 11 ліст., 10 табл., 40 джерел.

GPT-4o, LLM, NLP, ВЕБЗАСТОСУНОК, ВЕРИФІКАЦІЯ ТЕКСТУ, ОБРОБКА ПРИРОДНОЇ МОВИ, ПРОМПТ-ІНЖЕНІРИНГ, ШТУЧНИЙ ІНТЕЛЕКТ

Об’єкт дослідження – процес автоматизованої перевірки та корекції текстових даних українською мовою.

Предмет дослідження – методи та засоби підвищення ефективності виявлення граматичних, стилістичних та пунктуаційних помилок з використанням великих мовних моделей.

Мета роботи – підвищення ефективності та швидкості автоматизованої перевірки текстів шляхом розробки комп’ютерної системи на основі LLM із використанням методу гранулярного виявлення помилок та структурованого виводу даних.

Методи дослідження – аналіз літературних джерел та існуючих аналогів, об'єктно-орієнтоване проєктування, системний аналіз, метод порівняльного експерименту, промпт-інженіринг (Few-Shot, Chain-of-Thought).

Результати – розроблено багаторівневу архітектуру обробки помилок та клієнт-серверну вебсистему (Next.js, Python Flask) з інтеграцією моделей GPT-4o та GPT-3.5 Turbo. Реалізовано підхід гранулярного виявлення помилок (Granular Error Detection). Дослідження показало, що розроблена система перевершує аналоги за показниками повноти пошуку (Recall 81.2% у складних сценаріях) та забезпечує високу швидкодію (до 5.08 с для великих текстів).

Галузь використання – автоматизація процесів редагування текстів, освітня сфера, ділове листування, створення контенту.

ABSTRACT

Explanatory note to the master's work: 104 p., 37 figures, 11 listings, 10 tables, 25 sources.

ARTIFICIAL INTELLIGENCE, GPT-4o, LLM, NATURAL LANGUAGE PROCESSING, NLP, PROMPT ENGINEERING, TEXT VERIFICATION, WEB APPLICATION

The object of research is the process of automated verification and correction of text data in Ukrainian.

The subject of research is methods and tools for improving the efficiency of detecting grammatical, stylistic, and punctuation errors using large language models.

The purpose of the work is to improve the efficiency and speed of automated text verification by developing a computer system based on LLM using the granular error detection method and structured data output.

Research methods: analysis of literature sources and existing analogs, object-oriented design, system analysis, comparative experiment method, prompt engineering (Few-Shot, Chain-of-Thought).

Results: a multi-level error handling architecture and a client-server web system (Next.js, Python Flask) with GPT-4o and GPT-3.5 Turbo integration were developed. A Granular Error Detection approach was implemented. Research showed that the developed system outperforms analogs in recall (81.2% in complex scenarios) and ensures high performance (up to 5.08 s for large texts).

Field of application: automation of text editing processes, education sector, business correspondence, content creation.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

AI – Artificial Intelligence, Штучний інтелект

API – Application Programming Interface, Інтерфейс прикладного програмування

CoT – Chain-of-Thought, Ланцюжок думок

GEC – Grammatical Error Correction, Автоматичне виправлення граматичних помилок

HTTP – HyperText Transfer Protocol, Протокол передачі гіпертексту

JSON – JavaScript Object Notation, Текстовий формат обміну даними

LLM – Large Language Model, Велика мовна модель

NLP – Natural Language Processing, Обробка природної мови

REST – Representational State Transfer, Передача стану представлення

SDK – Software Development Kit, Набір засобів розробки програмного забезпечення

SPA – Single Page Application, Односторінковий вебзастосунок

SSR – Server-Side Rendering, Рендеринг на стороні сервера

UI – User Interface, Інтерфейс користувача

UX – User Experience, Користувацький досвід

WSGI – Web Server Gateway Interface, Інтерфейс шлюзу вебсервера

ВСТУП

Стрімке зростання обсягів цифрового контенту висуває високі вимоги до якості письмової комунікації. Традиційні методи ручної перевірки є трудомісткими, а існуючі автоматизовані системи не завжди враховують морфологічні особливості української мови. Перспективним шляхом вирішення цієї проблеми вбачається використання великих мовних моделей (LLM), адаптованих для глибокого аналізу тексту.

Метою роботи є підвищення ефективності автоматизованої перевірки та виправлення текстів українською мовою шляхом розробки комп'ютерної системи на основі LLM із використанням методу гранулярного виявлення помилок.

Наукова новизна очікуваних результатів полягатиме в розробці підходу, що інтегрує LLM з оптимізованим промпт-інженірингом та багаторівневою обробкою даних. Планується застосувати метод гранулярного виявлення помилок (Granular Error Detection), який, на відміну від існуючих рішень, дозволить ідентифікувати кожну помилку окремо зі збереженням контексту, що має суттєво підвищити точність аналізу.

Практичне значення роботи полягатиме у створенні клієнт-серверного вебзастосунку для корекції помилок у режимі реального часу. Проєктована архітектура має забезпечити стабільність при високих навантаженнях, що зробить систему придатною для впровадження в освітній та бізнес-сферах. Очікується, що розроблена система перевершить існуючі аналоги за показниками повноти пошуку та швидкодії.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Історія розвитку систем з коригування помилок в тексті

Історія взаємодії людини та обчислювальної машини нерозривно пов'язана з еволюцією методів обробки природної мови (Natural Language Processing, NLP). Серед усіх завдань NLP автоматичне виявлення та виправлення помилок у тексті (Grammatical Error Correction, GEC) займає особливе місце. Це завдання, що почалося як проста звірка зі словником для економії пам'яті на мейнфреймах, трансформувалося у складну дисципліну, що поєднує теорію інформації, лінгвістичну статистику, когнітивну психологію та глибоке навчання.

1.1.1 Доцифрова епоха та перші алгоритмічні підходи

Розуміння того, як машина може виправляти людину, базується на двох фундаментальних концепціях середини XX століття. З одного боку, це генеративна граматика Ноама Хомського, яка формалізувала мову як набір математичних правил, що дозволяють генерувати нескінченну кількість правильних речень. Це дало надію на те, що комп'ютер, вивчивши ці правила, зможе детерміновано відрізняти правильне від неправильного [1]. З іншого боку, теорія інформації Клода Шеннона розглядала текст як сигнал, а помилку — як "шум" у каналі передачі. Ця дихотомія між "правилом" і "ймовірністю" визначила боротьбу двох основних підходів у спелчекінгу на наступні пів століття [2].

Ще до появи текстових редакторів виникла потреба в пошуку інформації в базах даних, зокрема імен та прізвищ, які часто записувалися з помилками або варіаціями. Першим проривом став алгоритм Soundex, запатентований Робертом Расселом і Маргарет Оделл у 1918 році та згодом адаптований для машинної обробки [3].

Soundex базується на ідеї, що приголосні несуть основне інформаційне навантаження слова, а голосні є менш стабільними й схильними до помилок. Алгоритм перетворює слово на чотиризначний код, що складається з першої літери та трьох цифр [3].

Механізм роботи Soundex:

- зберігається перша літера слова;

- наступні літери (A, E, I, O, U, H, W, Y) ігноруються;

- інші приголосні кодуються цифрами за групами схожого звучання (наприклад, губні B, F, P, V отримують код '1', а шиплячі C, G, J, K, Q, S, X, Z — код '2');

- сусідні символи з однаковим кодом об'єднуються;

- результат обрізається або доповнюється нулями до 4 знаків.

Завдяки Soundex, системи могли ідентифікувати, що "Smith", "Smyth" і "Smythe" є варіантами одного прізвища (код S530). Це був перший крок до "нечіткого пошуку" (fuzzy matching), хоча він був обмежений лише фонетичною подібністю англійської мови і не міг виправляти друкарські помилки, що не змінюють звучання [3].

Пізніше з'явилися вдосконалені фонетичні алгоритми, такі як Metaphone та Double Metaphone, які враховували складніші правила англійської орфографії та могли генерувати два коди для одного слова, щоб покрити можливі варіації вимови, що стало критичним для обробки іншомовних прізвищ у США [3].

У 1961 році Лес Ернест (Les Earnest) у Массачусетському технологічному інституті (MIT) розпочав експерименти з розпізнаванням рукописного курсиву. Він швидко зрозумів, що без словника, який би обмежував можливі варіанти розпізнавання, система генерує нісенітницю. Ернест створив список із 10 000 найчастіших англійських слів. Його програма перевіряла введене слово на наявність у цьому списку. Це був верифікатор, а не коректор: він міг сказати, що слова немає в списку, але не міг запропонувати правильний варіант [4].

Справжня революція відбулася в лютому 1971 року в Лабораторії штучного інтелекту Стенфордського університету (SAIL). Ральф Горін (Ralph Gorin), аспірант Ернеста, створив програму SPELL для мейнфреймів DEC PDP-10. На відміну від попередників, SPELL була інтерактивною і пропонувала варіанти виправлення [4].

Для генерації пропозицій Горін використав концепцію, яку пізніше назвуть

відстанню редагування (Edit Distance). Основою для цього стала робота радянського математика Володимира Левенштейна, який формалізував метрику відмінності між двома рядками [5].

Відстань Левенштейна визначається як мінімальна кількість операцій, необхідних для перетворення одного рядка в інший. Допустимі операції:

- вставка (Insertion): cat - caat;

- видалення (Deletion): cat - at;

- заміна (Substitution): cat - bat [5].

Горін імплементував пошук слів у словнику, які відрізнялися від введеного слова на одну операцію редагування (відстань Левенштейна = 1). Пізніше до цього набору операцій додали транспозицію (перестановку двох сусідніх символів, наприклад, teh -> the), створивши метрику Дамерау-Левенштейна, оскільки перестановка є однією з найпоширеніших помилок при швидкому наборі тексту на клавіатурі [6].

Як наслідок, програма SPELL поширилася через мережу ARPANET, ставши одним із перших прикладів вільного програмного забезпечення. Вона встановила стандарт інтерфейсу та алгоритмічної бази для всіх майбутніх спелчекерів, включаючи знаменитий ispell в Unix, який став прабатьком сучасних open-source словників [5].

1.1.2 Епоха Unix та Writer's Workbench: від орфографії до стилістики (1970–1985)

З розвитком операційної системи Unix у Bell Labs підхід до роботи з текстом змінився. Замість монолітних програм розроблялися невеликі утиліти, кожна з яких виконувала одну функцію, але могла комбінуватися з іншими через "трубопроводи" (pipes). Це середовище стало ідеальним інкубатором для перших систем перевірки граматики та стилю [7].

У кінці 1970-х років Лорінда Черрі (Lorinda Cherry) та Ніна Макдональд (Nina Macdonald) розробили пакет Writer’s Workbench (WWB). Це була перша комплексна система комп'ютерної підтримки письма, яка вийшла далеко за межі простої перевірки орфографії. Система не мала "штучного інтелекту" у сучасному

розумінні. Вона покладалася на зіставлення зі зразком (pattern matching) та статистичні евристики [7]. Основні компоненти представлені на таблиці 1.1.

Таблиця 1.1 – Основні компоненти Writer’s Workbench

Утиліта	**Призначення**	**Механізм роботи**
spell	Перевірка орфографії	Порівняння зі словником (хеш-таблиці), морфологічний стемінг.
diction	Пошук поганого стилю	Пошук за списком із сотень "багатослівних" або архаїчних фраз (наприклад, "in view of the fact that" -> "since").
style	Статистичний аналіз	Розрахунок індексів читабельності (Flesch-Kincaid), довжини речень, співвідношення дієслів до прикметників.
sexist	Етична перевірка	Виявлення гендерно-нечутливої лексики (наприклад, "chairman" -> "chairperson").
double	Пошук повторів	Виявлення помилок типу "the the" на межі рядків.
parts	Частиномовна розмітка	Ймовірнісний алгоритм визначення частин мови для граматичного аналізу (точність до 96%).

WWB активно використовувався в американських університетах (зокрема в Колорадо) для навчання студентів. Дослідження показали, що комп'ютерний аналіз часто корелював з оцінками викладачів, хоча й базувався на примітивних метриках, Незважаючи на популярність, WWB та його ранні PC-клони (такі як Grammatik I та II) мали суттєві обмеження. Вони не будували синтаксичного дерева речення [7].

- програма могла порадити не використовувати пасивний стан, навіть якщо він був стилістично виправданим;

- правилові системи (Rule-based) того часу не могли виявити помилку узгодження підмета і присудка, якщо між ними стояло кілька інших слів. Вони реагували лише на лінійні послідовності слів таких, як довжина слів [7].

1.1.3 Інтерфейсна революція та війна текстових процесорів (1980–2000)

З появою персональних комп'ютерів технології, що раніше були доступні лише на мейнфреймах університетів, прийшли в дім і офіс. Першим спелчекером для мікрокомп'ютерів став WordCheck (1980) для систем Commodore [8]. У 1981 році з'явився Grammatik для IBM PC, який спочатку був прямим наслідувачем ідей

Writer's Workbench, але згодом (у версіях III та IV) додав елементи синтаксичного розбору [9].

До середини 1980-х років ринок окремого ПЗ для перевірки правопису почав зникати. Розробники текстових процесорів — WordStar, WordPerfect та пізніше Microsoft Word — почали інтегрувати модулі перевірки безпосередньо у свої продукти. Це змінило користувацький досвід: перевірка стала не окремим етапом "після написання", а частиною процесу створення тексту [10].

Найбільш значущою зміною в інтерфейсі користувача стала поява фонової перевірки правопису, що позначає помилки червоною хвилястою лінією безпосередньо в тексті. Ця функція вперше з'явилася в Microsoft Word 95/97 [11]. До цього перевірка була модальною: користувач натискав кнопку, і з'являлося вікно, яке переривало роботу. Ідея фонової перевірки викликала внутрішні суперечки в Microsoft. Команда розробників Office, яку в той час очолював Стівен Сінофскі, шукала способи зробити інтерфейс більш інтуїтивним [12]. Дін Хачамович, відомий також створенням функції AutoCorrect був одним із ключових ідеологів нових методів взаємодії [13].

Перевірка відбувалася лише тоді, коли користувач припиняв друкувати на долі секунди, щоб не сповільнювати систему. Червоний колір був обраний як сигнал тривоги, що традиційно використовувався вчителями. Згодом додалася зелена лінія для граматики та синя для контекстних помилок. Цей інтерфейс став настільки стандартизованим, що тепер він використовується у всіх браузерах та редакторах коду. Однак він також породив феномен "сліпої довіри" до комп'ютера та роздратування через хибні спрацьовування на власних назвах, особливо неангломовних, що порушує питання алгоритмічної упередженості [13].

1.1.4 Український контекст: Відродження та цифровізація (1990–2010)

Розпад Радянського Союзу та здобуття Україною незалежності створили унікальний виклик: необхідність швидкої комп'ютеризації української мови, яка десятиліттями витіснялася з науково-технічної сфери.

Розробка спелчекера для української мови є на порядок складнішою задачею, ніж для англійської. Українська мова є флективною: одне слово може мати десятки

форм (відмінки, роди, числа, особи дієслів). Простий словник слів (wordlist), як у ранніх англійських системах, тут неефективний, оскільки його розмір був би надто великим. Необхідні були алгоритми морфологічного аналізу та генерації (стемінг, лематизація) [14].

Компанія ProLing стала піонером українізації. Їхня система перевірки правопису «Рута» та система перекладу «Плеяда» стали фундаментом українського офісного ПЗ. Архітектура “Рути” представляє класичну правилово-словникову систему. Вона містила потужний морфологічний модуль, здатний розпізнавати парадигми слів. На відміну від західних аналогів, «Рута» враховувала специфіку кириличної кодової сторінки та нюанси українського правопису (наприклад, чергування у/в, і/й) [14].

Протягом багатьох років Microsoft ліцензувала модулі ProLing для української локалізації Microsoft Office. Тобто, "червона хвиляста лінія" під українським текстом у Word 2000-2010 працювала на двигуні «Рути» [14].

Паралельно з комерційними розробками виник потужний волонтерський рух. Mova.info та CyberMova проєкти, котрі створили перші масштабні онлайн-корпуси та словники, забезпечивши наукове підґрунтя для цифрової лінгвістики [14].

З появою відкритого спелчекера LanguageTool (заснованого Даніелем Набером), українська спільнота отримала платформу для створення вільного інструменту [15]. Андрій Рисін (Andriy Rysin) та його команда розробили український модуль для LanguageTool, який базується на великому словнику з морфологічною розміткою (dict_uk). Цей модуль використовує тисячі XML-правил для виявлення не лише орфографічних, а й складних граматичних помилок (наприклад, кальок типу "приймати участь" замість "брати участь"). Сьогодні це основний засіб перевірки української мови у LibreOffice, Firefox та багатьох онлайн-сервісах [16].

1.1.5 Статистичний поворот: від правил до ймовірностей (2000–2015)

До початку 2000-х стало очевидно, що ручне написання правил (як у Grammatik чи ранніх версіях LanguageTool) не масштабується. Мова занадто варіативна. Правилові системи страждали від двох проблем:

- вони не могли передбачити всі можливі помилки;

- жорсткі правила часто позначали правильні, але рідкісні конструкції як помилкові [17].

На зміну правилам прийшла статистика. Дослідники повернулися до ідеї Шеннона, реалізувавши Модель зашумленого каналу [17].

Компанія Google, маючи доступ до індексу всього інтернету, почала використовувати N-грами для виправлення помилок. Це дозволило вирішити проблему контекстних помилок (real-word errors), коли слово написано правильно, але вжито недоречно (наприклад, "desert" замість "dessert"). Система аналізує триграми (послідовності з 3 слів). Якщо триграма "eat delicious dessert" зустрічається в корпусі мільйон разів, а "eat delicious desert" — нуль, система впевнено запропонує виправлення, навіть якщо слово "desert" є в словнику [17].

1.1.6 Нейромережева революція та ера Трансформерів (2015–сьогодення)

Поява технологій Word2Vec та GloVe дозволила перетворити слова на багатовимірні вектори. Слова, що мають схоже значення або вживаються у схожому контексті, отримують схожі вектори. Це дозволило системам розуміти семантику, а не просто порівнювати символи [18].

З розвитком рекурентних нейромереж (RNN) та архітектури LSTM, задачу виправлення помилок почали формулювати як задачу машинного перекладу. Нейромережа "перекладає" речення з "мови з помилками" на "правильну мову". Цей підхід дозволив виправляти глобальні граматичні помилки (порядок слів, узгодження часів), які були недоступні для N-грамних моделей [19].

У 2017 році архітектура Transformer та механізм уваги (Self-Attention) змінили все [20]. Моделі на кшталт BERT (Bidirectional Encoder Representations from Transformers) здатні аналізувати все речення одночасно, враховуючи залежності між словами на будь-якій відстані [21].

Сучасні системи (Grammarly, Microsoft Editor, Google Docs) використовують трансформери для виправлення стилю, тону та складної граматики. Вони не просто шукають помилки, а пропонують перефразування для покращення ясності тексту [20, 21].

Довгий час українська мова вважалася "low-resource" (з обмеженими ресурсами) у світі NLP. Більшість моделей були мультимовними (mBERT), де українська займала незначну частку. Ситуація змінилася в останні роки завдяки зусиллям дослідників та волонтерів (проєкт UberText, Віталій Радченко, Олесь Добсевич та ін.) [21].

Порівняння основних парадигм систем виправлення помилок представлена у таблиці 2.

Таблиця 1.2. Порівняння основних парадигм систем виправлення помилок

Характеристика	**Rule-Based (1980-2000)**	**Statistical / N-gram (2000-2015)**	**Neural / Transformer (2015-наш час)**
Представники	Ruta, Grammatik, LanguageTool (early)	MS Word 2007, Google Wave	Grammarly
Основний ресурс	Словники, набори правил	Великі корпуси текстів	Векторні представлення, GPU
Тип знань	Лінгвістичні (експертні)	Частотні (ймовірнісні)	Семантичні (контекстні)
Сильні сторони	Повний контроль, пояснюваність	Виправлення контекстних помилок	Розуміння змісту, перефразування
Слабкі сторони	Трудоємність розробки правил	Потреба в гігантських даних	"Чорна скринька", галюцинації

1.2 Огляд існуючих систем перевірки та корегування тексту

На сучасному етапі розвитку технологій обробки природної мови (NLP) ринок програмних засобів для роботи з текстом можна умовно поділити на дві великі категорії: інструменти на основі правил та статистичних методів (Rule-based/Statistical) та новітні генеративні системи на базі штучного інтелекту (Generative AI). Для обґрунтування вибору власного підходу доцільно проаналізувати яскравих представників обох категорій: LanguageTool як еталон

класичного підходу та платформу isgen.ai як приклад спеціалізованого AI-інструменту.

1.2.1 Аналіз системи LanguageTool

LanguageTool є однією з найпопулярніших систем перевірки граматики та стилю з відкритим вихідним кодом. Як зазначалося в історичному огляді, український модуль LanguageTool базується на великих словниках з морфологічною розміткою та правилах [22].

Архітектурні особливості: Система використовує гібридний підхід. Основна перевірка базується на правилах (XML-rules), які описують шаблони помилок. Наприклад, правило може шукати послідовність слів, що не узгоджуються у відмінку. Для складніших випадків використовуються статистичні дані (N-грами) для перевірки стилю [22].

При дослідженні даної системи було використано тексти з наступними характеристиками:

- текст на 22 символи та 4-ма помилками;
- текст на 583 символи з 10-ма помилками;
- текст на 1617 символів з 50-ма помилками.

На рис. 1.1 представлено дослідження системи на прикладі короткого тексту.

Як можна бачити з результату, LanguageTool знайшов всього одну помилку, що не є виправданим результатом і становить лише 25% від помилок. Проте як можна побачити на рис. 1.2, сама швидкість обробки запиту достатньо ведика, а саме суммарно 4с.

На рис. 1.3 представлено дослідження системи на прикладі середнього тексту.

Як можна бачити з результату, LanguageTool знайшов сім помилок, що становить 70% успішного знаходження помилок. На рис. 1.4 представлено дані з швидкості обробки запиту, а саме - 6,7 секунд.

На рис. 1.5 представлено дослідження системи на прикладі великого тексту.

Як можна бачити з результату, LanguageTool знайшов 25 помилок, що становить лише 50% успішного знаходження помилок. На рис. 1.6 представлено

дані з швидкості обробки запиту, а саме - 13,9 секунд.

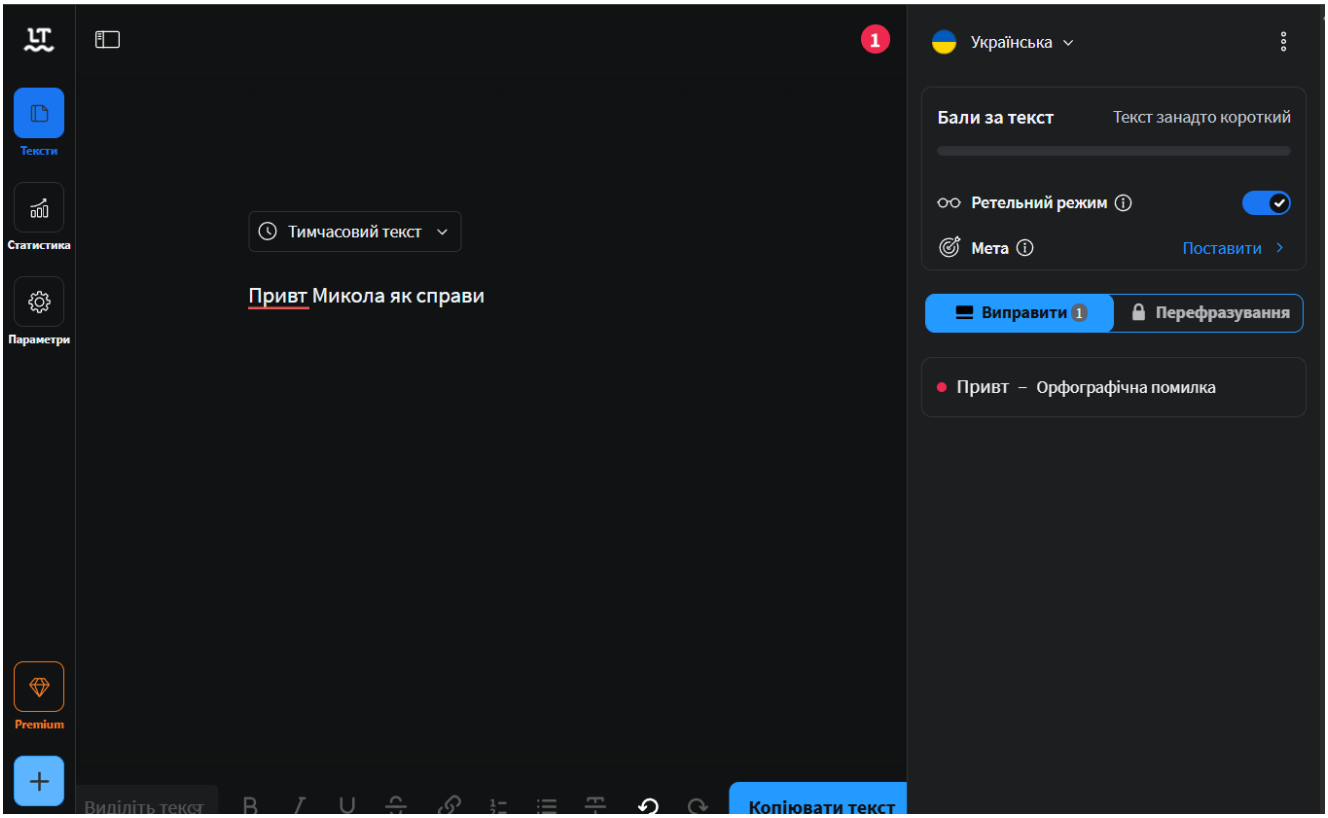


Рисунок 1.1 – Результати дослідження на тексті малого обсягу

Name	Status	Type	Initiator	Size	Time
49684051	200	fetch	app-uk.js?id=	1.1 kB	299 ms
49684051	200	fetch	app-uk.js?id=	1.1 kB	314 ms
49684051	200	fetch	app-uk.js?id=	1.1 kB	303 ms
check?c=1&instanceId...	200	fetch	app-uk.js?id=	0.4 kB	806 ms
check?c=1&instanceId...	200	fetch	app-uk.js?id=	1.1 kB	943 ms
49684051	200	fetch	app-uk.js?id=	1.1 kB	274 ms
1.svg?203a9c4ebabd6...	200	svg+...	base.css?id=0	(disk ...	5 ms
ru@2x.png?7454df695...	200	webp	base.css?id=0	1.1 kB	83 ms
49684051	200	fetch	app-uk.js?id=	1.2 kB	325 ms
httpapi	200	prefli...	Preflight	0.0 kB	892 ms
httpapi	200	fetch	app-uk.js?id=	0.2 kB	244 ms
49684051	200	fetch	app-uk.js?id=	1.2 kB	288 ms
check?c=1&instanceId...	200	fetch	app-uk.js?id=	0.4 kB	1.20 s
check?c=1&instanceId...	200	fetch	app-uk.js?id=	1.2 kB	1.20 s
49684051	200	fetch	app-uk.js?id=	1.1 kB	276 ms
httpapi	200	fetch	app-uk.js?id=	0.2 kB	245 ms

Рисунок 1.2 – Швидкість обробки запиту для малого тексту

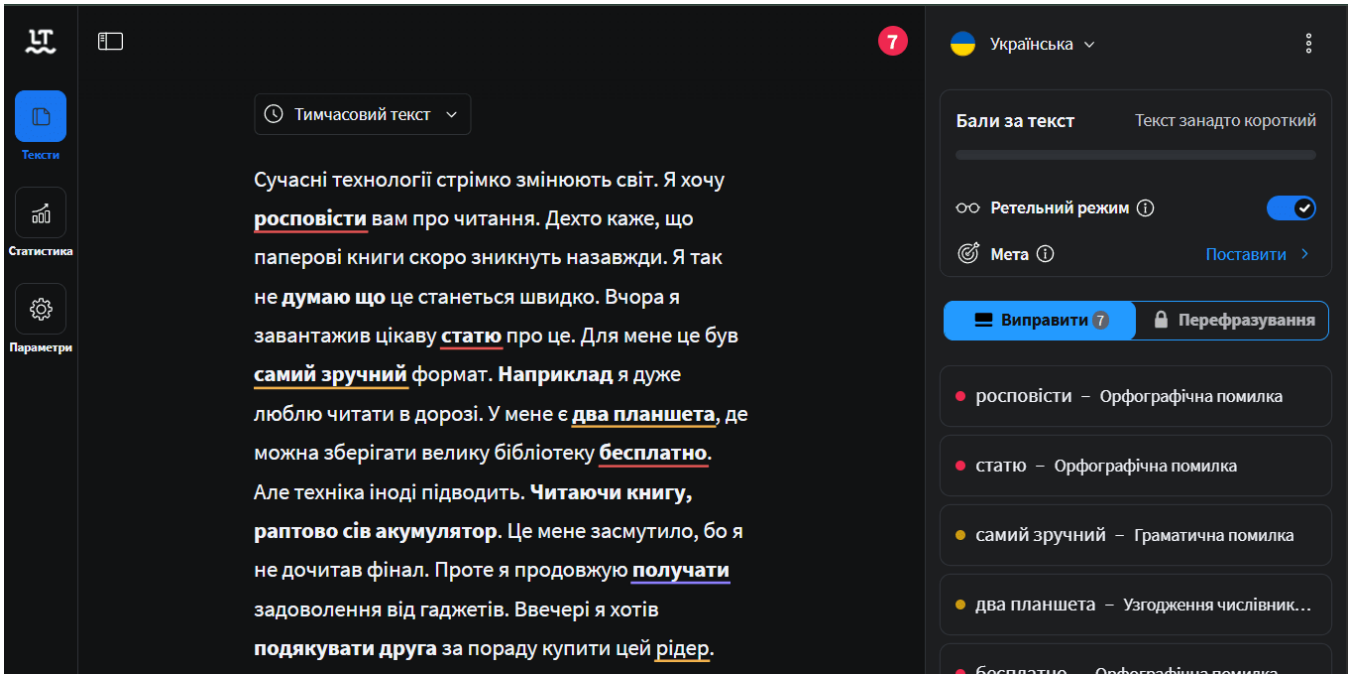


Рисунок 1.3 – Результати дослідження на тексті середнього обсягу

Name	Status	Type	Initiator	Size	Time
collect?v=2&tid=G-ME...	204	fetch	app-uk.js?id=5	0.0 kB	569 ms
collect?v=2&tid=G-ME...	204	ping	js?id=G-ME7Q	0.0 kB	559 ms
ga-audiences?v=1&t=s...	200	gif	js?id=G-ME7Q	0.1 kB	443 ms
49684051	200	fetch	app-uk.js?id=5	2.0 kB	344 ms
httpapi	200	fetch	app-uk.js?id=5	0.2 kB	938 ms
check?c=1&instanceId...	200	fetch	app-uk.js?id=5	0.6 kB	1.97 s
check?c=1&instanceId...	200	fetch	app-uk.js?id=5	1.7 kB	1.65 s
7.svg?9997f7d4022c11...	200	svg+...	base.css?id=04	(disk ...	5 ms
httpapi	200	fetch	app-uk.js?id=5	0.2 kB	237 ms

Рисунок 1.4 – Швидкість обробки запиту для середнього тексту

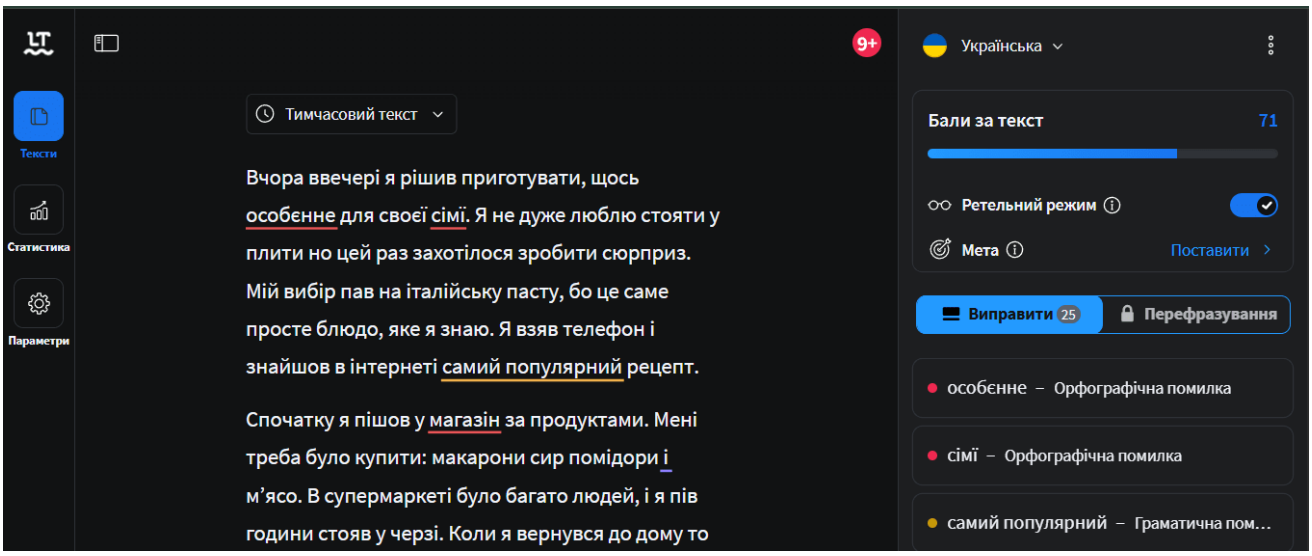


Рисунок 1.5 – Результати дослідження на тексті великого обсягу

Name	Status	Type	Initiator	Size	Time
49659234	200	fetch	app-uk.js?id=5	1.4 kB	4.56 s
httpapi	200	fetch	app-uk.js?id=5	0.2 kB	1.28 s
check?c=1&instanceId...	200	fetch	app-uk.js?id=5	0.5 kB	2.59 s
check?c=1&instanceId...	200	fetch	app-uk.js?id=5	16.8 kB	4.36 s
collect?v=2&tid=G-ME...	204	fetch	app-uk.js?id=5	0.0 kB	104 ms
9.svg?eb4a9b43f0bb90...	200	svg+...	base.css?id=04	(disk ...	6 ms
httpapi	200	fetch	app-uk.js?id=5	0.2 kB	971 ms

Рисунок 1.6 – Швидкість обробки запиту для великого тексту

Переваги системи:

- завдяки спільноті та використанню dict_uk, LanguageTool має одну з найкращих баз правил для української мови, здатен виявляти кальки та специфічні граматичні помилки;

- кожне виправлення має чітке пояснення («Узгодження роду», «Пунктуаційна помилка»), що є критично важливим для навчального процесу.

Недоліки та обмеження:

- система часто пропускає помилки, якщо вони не описані явним правилом, або дає хибні спрацювання на складних, але правильних реченнях;

- LanguageTool аналізує текст переважно в межах одного речення (sentence-level boundaries). Він не може виявити логічні суперечності між абзацами або порушення загального стилю документа;

- система пропонує точкові виправлення слів, але рідко пропонує перебудову речення для покращення читабельності, що є сильною стороною LLM.

1.2.2 Аналіз платформи isgen.ai

Isgen.ai представляє нову хвилю інструментів, що виникли на базі генеративних моделей. Хоча цей інструмент часто позиціонується як засіб для «гуманізації» AI-тексту (AI Detection Remover), його механізми роботи є релевантними для задач глибокого редагування та стилістичного покращення [23].

На відміну від LanguageTool, isgen.ai не шукає помилки за правилами. Він використовує великі мовні моделі (LLM), донавчені на якісних корпусах людських текстів. Система приймає вхідний текст і генерує його перефразовану версію, зберігаючи зміст, але змінюючи лексику та синтаксичну структуру [23].

Переваги підходу:

- інструмент здатен перетворити сухий, роботизований текст на живий та природний, що недоступно для класичних спелчекерів;

- завдяки архітектурі трансформерів, система розуміє семантику тексту, що дозволяє виправляти лексичні помилки, які формально є граматично правильними (contextual spelling errors).

Недоліки в контексті коректури:

- користувач отримує готовий результат без пояснення, що саме було змінено і чому. Це унеможливлює навчання користувача на власних помилках;

- у спробі «гуманізувати» текст модель може викривити фактаж або втратити важливі технічні деталі, що є неприпустимим в офіційно-діловому чи науковому стилі.

Дослідження платформи відбувається за тим же принципом, що і LanguageTool.

На рис. 1.7 представлено дослідження системи на прикладі короткого тексту.

Як можна бачити з результату, програма знайшла 2 помилки, що є 50% від загальної кількості помилок. Проте як можна побачити на рис. 1.8, сама швидкість обробки запиту достатньо мала, а саме суммарно близько 3,64 секунд.

На рис. 1.9 представлено дослідження системи на прикладі середнього тексту.

Як можна бачити з результату, isgen.ai знайшов 8 помилок, що становить 80% успішного знаходження помилок. Як виднj на рис. 1.10, швидкість обробки запиту 6,7 секунд.

На рис. 1.11 представлено дослідження системи на прикладі великого тексту.

Як можна бачити з результату, isgen.ai знайшов 38 помилок, що становить 76% успішного знаходження помилок. Як видну на рис. 1.12, швидкість обробки запиту 12,7 секунд.

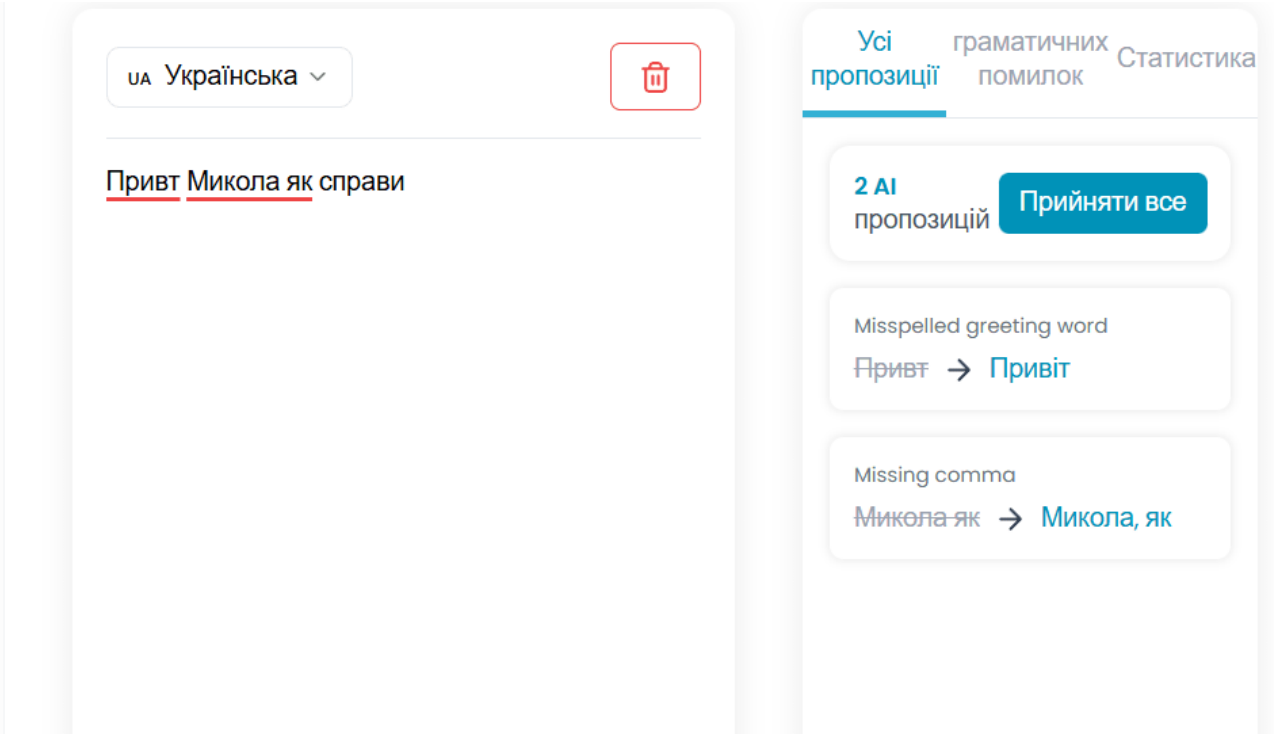


Рисунок 1.7 – Результати дослідження на тексті малого обсягу

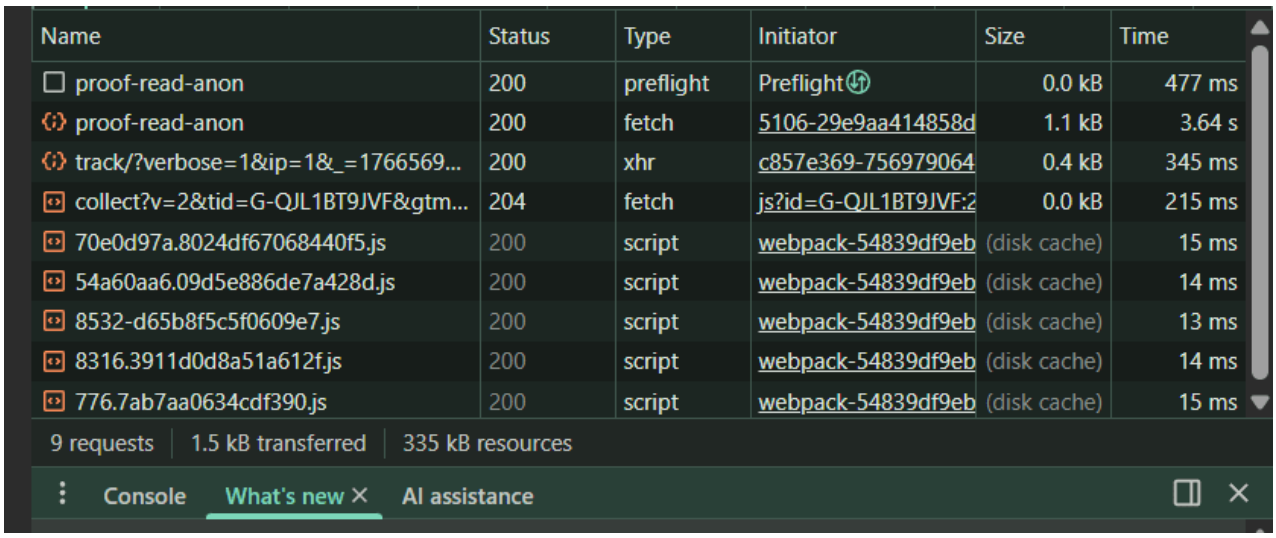


Рисунок 1.8 – Швидкість обробки запиту для малого тексту

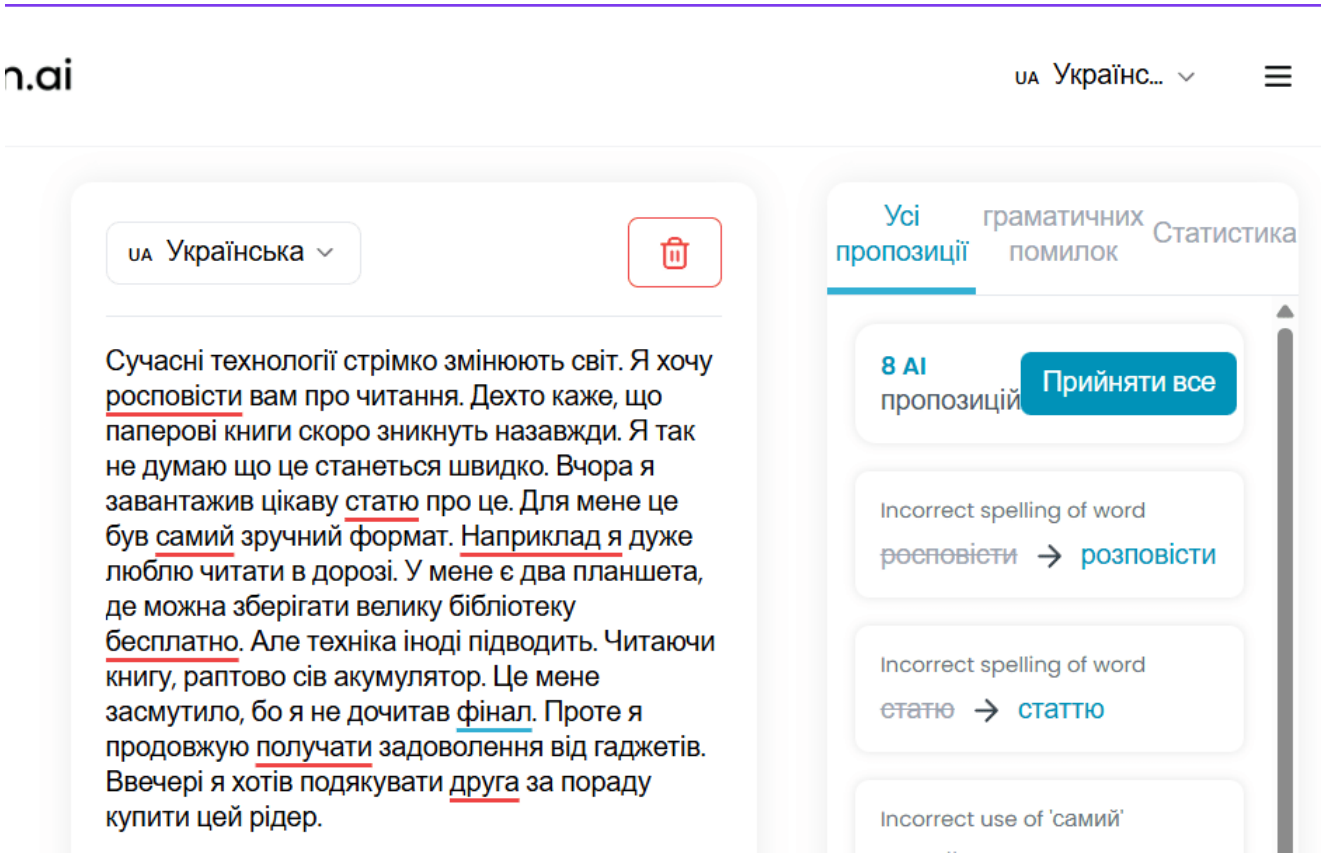


Рисунок 1.9 – Результати дослідження на тексті середнього обсягу

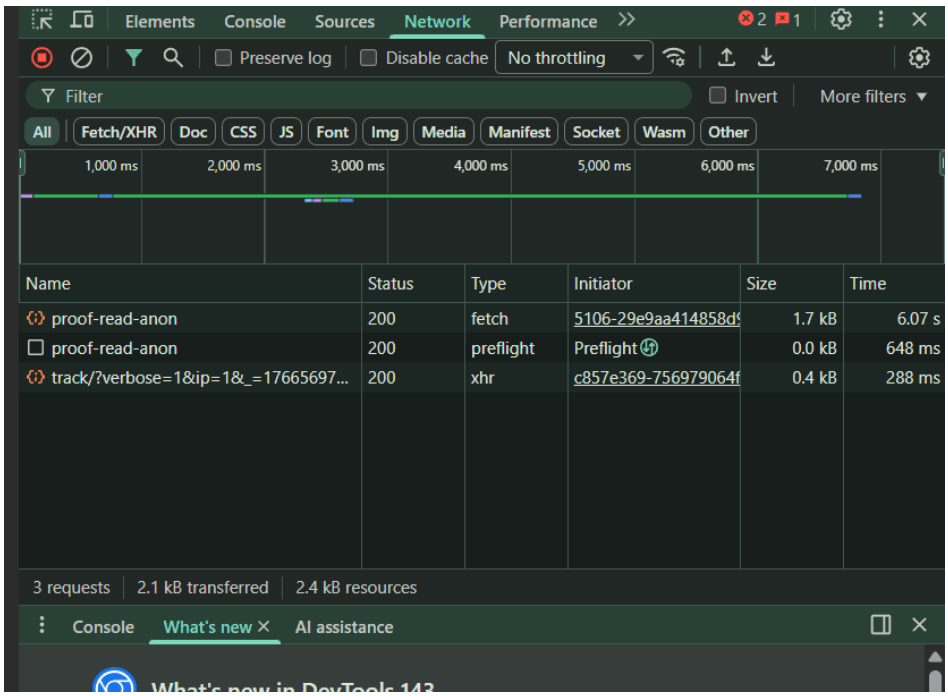


Рисунок 1.10 – Швидкість обробки запиту для середнього тексту

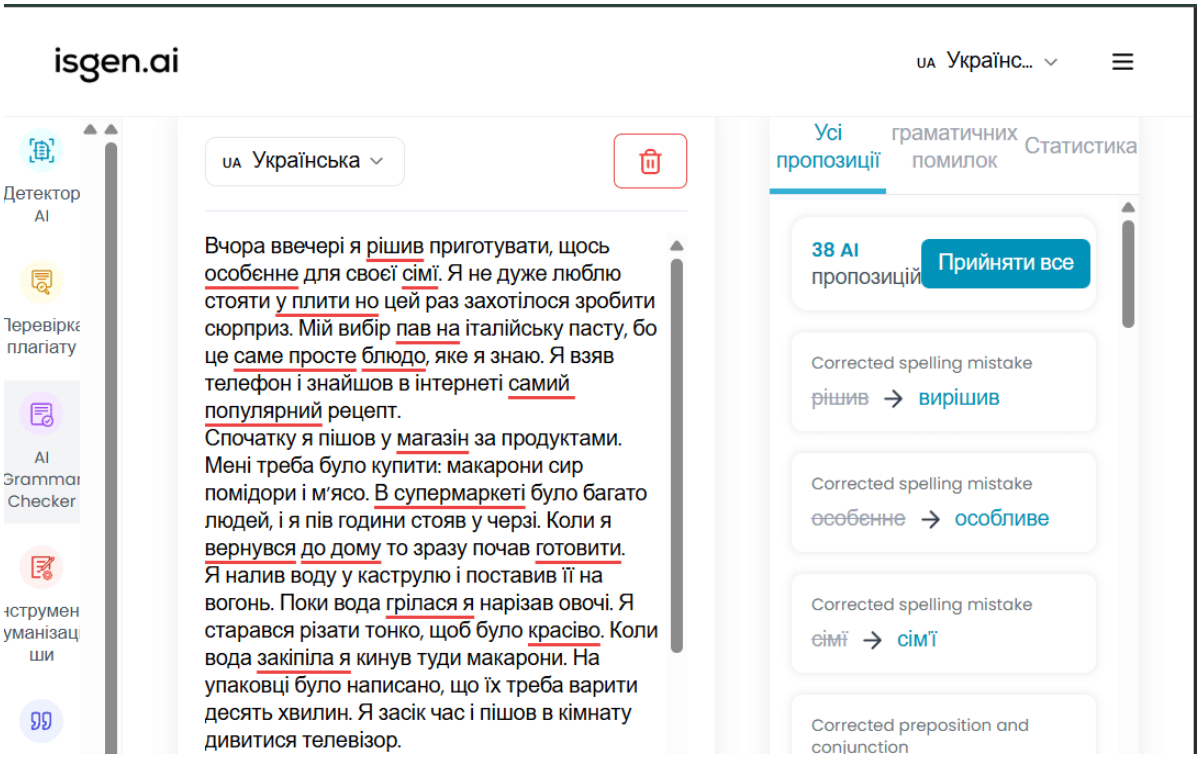


Рисунок 1.11 – Результати дослідження на тексті великого обсягу

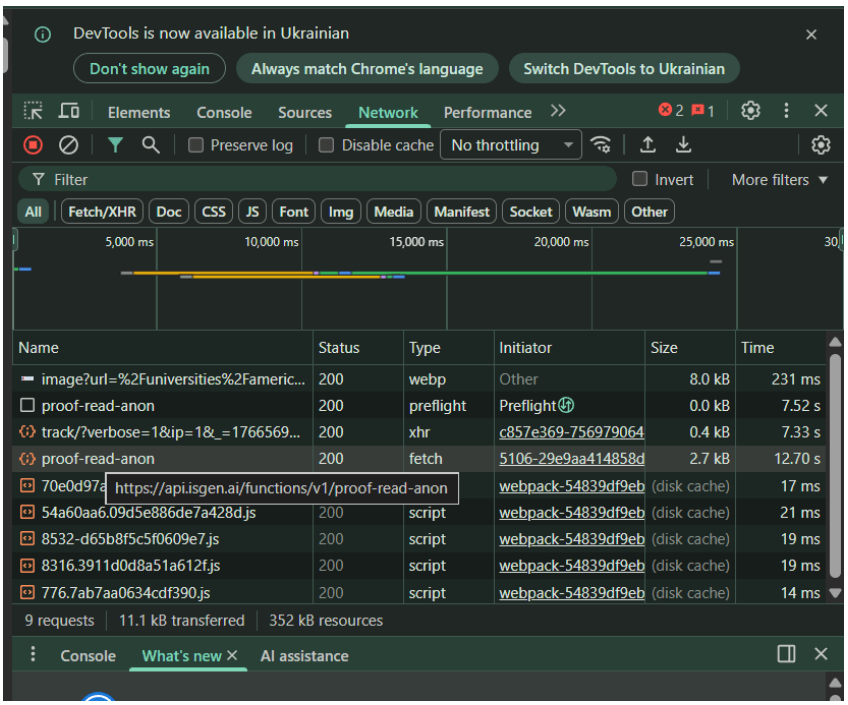


Рисунок 1.12 – Швидкість обробки запиту для великого тексту

1.3 Постановка задачі згідно результатів аналізу предметної області

Проведений у даному розділі аналіз існуючих рішень виявив суттєві обмеження в сучасних інструментах автоматичної перевірки текстів, особливо в контексті української мови.

Класичні системи на основі правил (на прикладі LanguageTool) демонструють високу надійність у поясненні простих помилок, але мають низьку ефективність виявлення складних граматичних та стилістичних конструкцій. Як показали результати дослідження, такі системи знаходять лише від 25% до 50% помилок у текстах різної складності, часто ігноруючи контекстні зв'язки.

З іншого боку, генеративні інструменти (на прикладі isgen.ai) забезпечують високу якість виправлення та «гуманізацію» тексту завдяки розумінню семантики. Однак вони мають критичні недоліки для задач професійної коректури: принцип «чорної скриньки» (відсутність пояснень виправлень) та доволі низьку швидкість

роботи.

Таким чином, виникає науково-практична проблема, що полягає у необхідності створення гібридного підходу, який поєднав би глибину семантичного аналізу великих мовних моделей (LLM) із точністю, керованістю та пояснюваністю класичних інструментів.

Метою роботи є підвищення ефективності та швидкості автоматизованої перевірки та виправлення текстів українською мовою шляхом розробки комп’ютерної системи на основі великих мовних моделей із використанням методу гранулярного виявлення помилок та структурованого виводу даних.

2 РОЗРОБКА ПІДХОДУ ДО ОБРОБКИ ПОМИЛОК ТА ЗАБЕЗПЕЧЕННЯ НАДІЙНОСТІ

Розробка інтелектуальної системи граматичної корекції потребує комплексного підходу до обробки помилок на всіх рівнях архітектури. У даному розділі представлено концептуальну модель багаторівневої системи обробки помилок, яка забезпечує стабільну роботу застосунку та надає користувачу зрозумілий зворотний зв'язок.

2.1 Обґрунтування вибору LLM

Вибір ядра для системи виправлення помилок базується на комплексному оцінюванні можливостей моделей обробляти текстову інформацію з високою точністю та передбачуваністю.

GPT-4o демонструє результат 82% на бенчмарку MMLU (Massive Multitask Language Understanding), що суттєво вище за 70%, які показувала GPT-3.5 Turbo.

Для системи виправлення помилок це означає, що "легка" модель тепер здатна виконувати не лише просту корекцію орфографії, але й складніший синтаксичний аналіз, який раніше вимагав залучення дорогих моделей рівня GPT-4. Це дозволяє реалізувати дворівневу архітектуру системи [24].

Для ChatGPT 3.5Turbo обробка 90% запитів, що включають виправлення очевидних помилок, пунктуації та базової граматики. Завдяки низькій вартості ($0.15 за 1M вхідних токенів проти $3.00 у Claude 3.5 Sonnet), ця модель робить систему економічно життєздатною для масового користувача [24].

Для GPT-4o залучення "важкої" моделі лише для складних випадків, що вимагають стилістичного перефразування або глибокого аналізу контексту.

У класичному підході (який досі частково використовується у конкурентів) розробник просить модель повернути JSON, але модель може повернути текст з поясненням, обрізаний JSON або JSON з помилкою синтаксису. Це вимагає написання складного коду для валідації та повторних запитів (retries), що уповільнює роботу системи [24].

OpenAI Structured Outputs гарантує, що відповідь моделі буде на 100% відповідати заданій JSON-схемі (JSON Schema). Це досягається шляхом обмеження семплювання токенів (constrained sampling) на рівні двигуна моделі, а не просто через промпт-інжиніринг [25].

Модель Claude 3.5 Sonnet від Anthropic заслужено вважається однією з найкращих у завданнях, що вимагають літературної майстерності та нюансованого розуміння тексту. Вона перевершує GPT-4o у бенчмарку GPQA (Graduate Level Reasoning), набираючи 59.4% проти 53.6%. Однак, у контексті автоматизованої системи виправлення помилок, ці переваги можуть стати недоліками [26].

По-перше, моделі Claude, навчені за методом Constitutional AI, схильні до надмірної обережності та частих відмов (refusals) при обробці контенту, який може здатися їм "небезпечним" або "суперечливим", навіть якщо це просто граматичний аналіз тексту. Це створює проблеми для надійності (reliability) системи, коли невинний текст користувача може бути відхилений фільтрами моделі [26].

По-друге, Claude 3.5 Sonnet часто демонструє тенденцію до "переписування"

тексту замість його виправлення. Вона намагається покращити стиль, зробити текст більш "ввічливим" або "літературним", навіть коли в інструкції чітко вказано "виправляти лише граматичні помилки". Це явище, відоме як over-editing, є небажаним для інструментів коректури, де важливо зберегти авторський голос. GPT-4o, завдяки більш керованій поведінці та кращому fine-tuning, дозволяє точніше налаштувати рівень втручання в текст [26].

По-третє, економічний фактор. Вартість використання Claude 3.5 Sonnet ($3.00 за 1M вхідних токенів) вища за GPT-4o ($2.50). Для студентського проєкту або стартапу така різниця у ціні є блокуючим фактором, особливо враховуючи, що для задач GEC різниця у якості між 4o та Sonnet є мінімальною або відсутньою [26].

Основною перевагою моделі Gemini 1.5 Pro є величезне контекстне вікно (до 2 мільйонів токенів). Це робить її ідеальною для аналізу цілих книг або величезних кодових баз ("needle in a haystack" tasks). Проте, задача виправлення помилок зазвичай оперує на рівні речень, абзаців або сторінок, де гігантський контекст не дає переваг, але збільшує латентність [27].

Критичним недоліком Gemini для розробки програмних систем є нестабільність формату виводу. Розробники часто стикаються з тим, що модель порушує задану JSON-схему, додає зайві поля або повертає текст у форматі Markdown, коли очікується чистий JSON. Хоча Google впроваджує "JSON mode", він все ще поступається за надійністю рішенню Structured Outputs від OpenAI. У системі, де бекенд автоматично парсить відповідь моделі для відображення виправлень користувачеві, будь-яке відхилення від схеми призводить до помилки (exception) і необхідності повторного запиту, що погіршує UX та збільшує витрати.

Крім того, бенчмарки показують, що Gemini 1.5 Pro поступається GPT-4o у задачах кодування та логіки, що, як зазначалося раніше, корелює з точністю граматичного аналізу складних речень [27].

Модель Grok-2 від xAI позиціонується як "бунтарська" модель з доступом до реального часу через платформу X (Twitter) та меншими обмеженнями цензури. Хоча це може бути цікавим для розважальних чат-ботів, для професійного інструменту редагування тексту це є серйозним ризиком. "Дотепний" або

"сарказтичний" тон, який є частиною особистості Grok, є абсолютно неприйнятним у системі, що має виправляти офіційні документи або академічні роботи [28].

З технічної точки зору, API Grok є менш зрілим, ніж у OpenAI. Екосистема інструментів, SDK та документації для Grok значно поступається тій, що створена навколо GPT моделей. Відсутність розвинених функцій, таких як надійний Structured Outputs або Batch API, робить інтеграцію Grok у складну архітектуру невиправдано складною. Бенчмарки також показують, що хоча Grok-2 наближається до лідерів, він все ще не перевершує GPT-4o у ключових метриках точності та слідування інструкціям [28].

Порівняння LLM показано у таблиці 2.1

Таблиця 2.1 – Порівняння характеристик LLM

Характеристика	**ChatGPT-4o**	**Claude 3.5 Sonnet**	**Gemini 1.5 Pro**	**Grok-2**
Instruction Following	Дуже високий (Structured Outputs)	Високий, але схильність до "повчання"	Середній (проблеми з JSON)	Середній (схильність до жартів)
Structured Output Reliability	100% (Strict Mode)	Висока (через Tool Use)	Змінна	Низька / Відсутня
Швидкість (Tokens/sec)	~109+	~72	~85	~80
Латентність (TTFT)	~0.56с	~1.23с	~0.7с	~0.9с
Вартість (Input/Output)	$2.50 / $10.00	$3.00 / $15.00	$1.25 / $10.00	$2.00 / $10.00
Fine-tuning	Доступний	Недоступний публічно	Доступний (Vertex AI)	Обмежений
Українська токенізація	Високоефективна (новий токенізатор)	Стандартна	Стандартна	Стандартна

На основі проведеного аналізу можна стверджувати, що вибір моделей ChatGPT-4o та ChatGPT-3.5 Turbo для системи виправлення текстів є оптимальним з наступних причин:

- архітектурна перевага - ультимодальна природа та новий токенізатор GPT-4o забезпечують кращу роботу з українською мовою та високу швидкість реакції;

- інженерна надійність - функція Structured Outputs гарантує детермінованість даних, що критично для програмної інтеграції, на відміну від нестабільних рішень

конкурентів;

- якість редагування - моделі OpenAI демонструють найкращий баланс між виявленням помилок та збереженням авторського стилю, з можливістю fine-tuning для специфічних потреб;

- економічна ефективність - наявність надешевої моделі ChatGPT-3.5 Turbo дозволяє побудувати рентабельну систему, недосяжну при використанні дорожчих моделей Anthropic або Google;

- доступність - повна підтримка українського регіону та розвинена екосистема інструментів мінімізують ризики розробки.

Таким чином, використання технологічного стеку OpenAI є найбільш обґрунтованим рішенням для реалізації поставленої задачі в рамках кваліфікаційної роботи магістра.

2.2 Архітектура обробки помилок

Система обробки помилок побудована за принципом «захисту в глибину» (defense in depth), де кожен рівень перехоплює специфічні типи помилок та передає їх на наступний рівень лише у разі неможливості локальної обробки. Така архітектура дозволяє ізолювати збої, запобігати каскадним відмовам та зберігати працездатність системи навіть при частковій недоступності окремих компонентів.

Концептуальна схема системи обробки помилок представлена на рисунку 2.1.

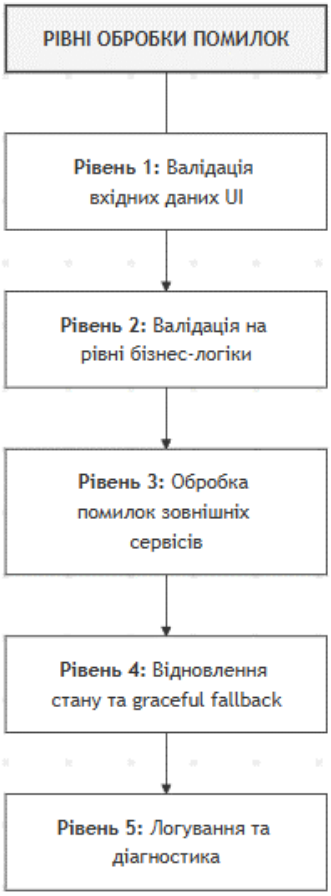


Рисунок 2.1 – Багаторівнева архітектура обробки помилок

Перший рівень захисту реалізує ранню валідацію даних безпосередньо на етапі їх введення користувачем. Такий підхід дозволяє миттєво інформувати користувача про некоректні дані, зменшуючи навантаження на систему та покращуючи користувацький досвід.

Алгоритм валідації включає наступні перевірки:

– перевірка на порожній або пробільний текст із формуванням повідомлення «Будь ласка, введіть текст для перевірки»;

– контроль максимальної довжини тексту (обмеження: 10 000 символів);

– валідація типу завантаженого файлу (підтримуються: TXT, DOCX);

– перевірка кодування файлу для коректної обробки кирилиці.

Код функції валідації представлено у лістингу 2.1.

Лістинг 2.1 – Функція валідації вхідного тексту

```
function validateInput(text, settings) {
    // Перевірка на порожній текст
    if (isEmpty(text) OR isWhitespaceOnly(text)) {
        return Error("EMPTY_TEXT", "Введіть текст для
перевірки")
    }

    // Перевірка довжини
    if (length(text) > MAX_TEXT_LENGTH) {
        return Error("TEXT_TOO_LONG",
            format("Максимум {} символів", MAX_TEXT_LENGTH))
    }

    // Валідація параметрів чутливості
    VALID_SENSITIVITIES = ["soft", "moderate", "strict"]
    if (settings.sensitivity NOT IN VALID_SENSITIVITIES) {
        return Error("INVALID_SENSITIVITY", "Невірний рівень
чутливості")
    }

    return ValidationResult.SUCCESS
}
```

Другий рівень обробки помилок забезпечує валідацію даних після їх отримання модулем обробки. Цей рівень є критичним з точки зору безпеки, оскільки запобігає обробці потенційно шкідливих або некоректних даних навіть у випадку обходу валідації на рівні інтерфейсу.

Реалізація подвійної валідації (на інтерфейсі та в бізнес-логіці) забезпечує принцип «не довіряй вхідним даним» (never trust input), що є фундаментальним для побудови безпечних систем. Типові коди помилок наведено у таблиці 2.2.

Таблиця 2.2 – Класифікація помилок бізнес-логіки

Код помилки	**HTTP статус**	**Опис та дія**
NO_DATA	400 Bad Request	Тіло запиту відсутнє
EMPTY_TEXT	400 Bad Request	Текст порожній або пробільний
TEXT_TOO_LONG	400 Bad Request	Перевищено ліміт 10 000 символів
INVALID_MODEL	400 Bad Request	Вказано неіснуючу модель AI
API_ERROR	500 Internal Error	Помилка зовнішнього сервісу

Структура обробки запиту з механізмом перехоплення помилок представлена у лістингу 2.2, де продемонстровано механізм послідовної валідації з поверненням інформативних повідомлень.

Лістинг 2.2 – Обробка запиту з механізмом перехоплення помилок

```
function processAnalysisRequest(request) {
    try {
        data = parseJSON(request.body)

        // Етап 1: Перевірка наявності даних
        if (data IS NULL) {
            return Response(400, {error: "Не надано дані"})
        }

        // Етап 2: Валідація тексту
        text = trim(data.text)
        validationResult = validateInput(text, data.settings)
        if (validationResult.hasError) {
            return Response(400, {error: validationResult.message})
        }

        // Етап 3: Виклик сервісу аналізу
        startTime = currentTimeMillis()
        result = analyzeText(text, data.sensitivity, data.model)
        elapsedTime = currentTimeMillis() - startTime

        // Додавання метаданих до відповіді
        result.processing_time_ms = elapsedTime
```

```
        result.text_length = length(text)

        return Response(200, result)

    } catch (Exception e) {
        log.error("Помилка обробки: " + e.message)
        log.error(e.stackTrace)
        return Response(500, {error: "Внутрішня помилка сервера"})
    }
}
```

Концепція graceful degradation («плавна деградація») передбачає збереження часткової функціональності системи при виникненні збоїв окремих компонентів. У контексті розроблюваної системи це означає, що навіть при недоступності зовнішнього AI-сервісу користувач отримує зрозуміле повідомлення та зберігає можливість працювати з раніше збереженими даними.

Матрицю реакцій системи на різні типи збоїв представлено у таблиці 2.3.

Таблиця 2.3 – Матриця graceful degradation

Тип збою	Реакція системи	Доступні функції
AI-сервіс недоступний	Повідомлення про тимчасову недоступність	Історія, автозбереження
Локальне сховище заповнене	Робота без збереження	Аналіз, експорт
Мережа недоступна	Офлайн-режим	Перегляд історії
Помилка читання файлу	Пропозиція ручного введення	Ручне введення тексту

2.3 Архітектура системи промптів

Система промптів побудована за модульним принципом із використанням патерну «базовий промпт + розширення». Такий підхід дозволяє уникнути дублювання коду та забезпечити узгодженість поведінки моделі.

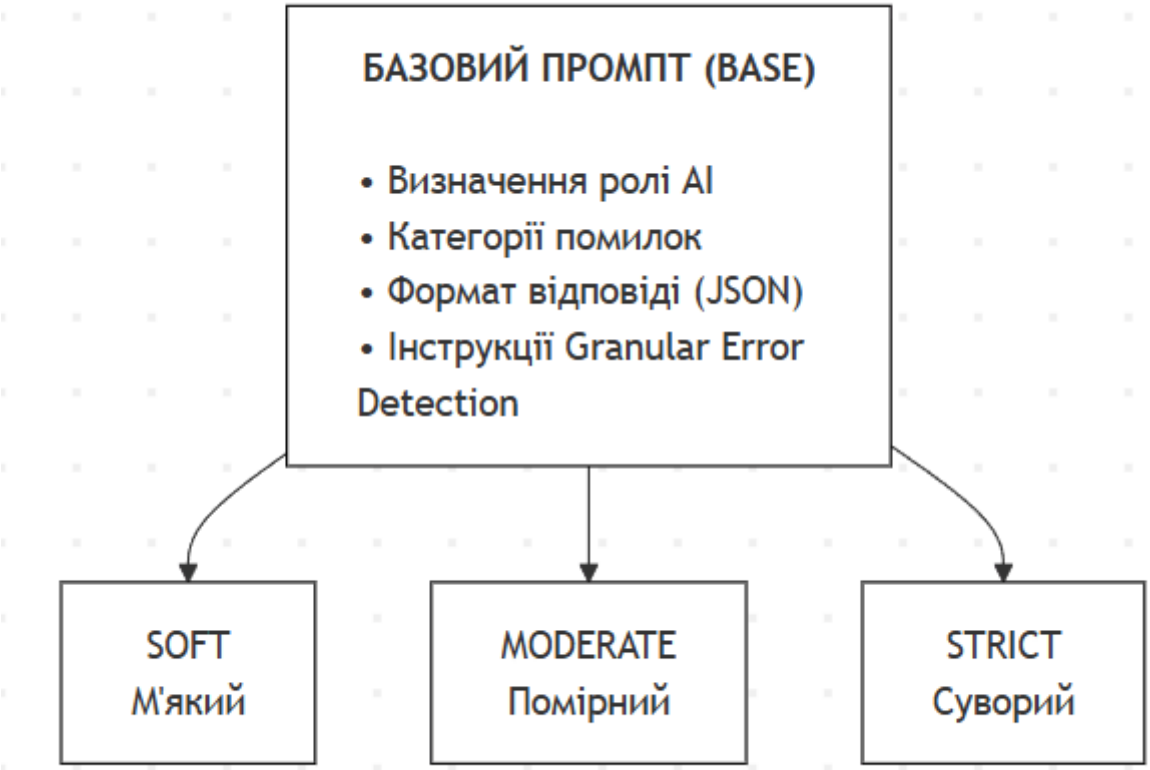


Рисунок 2.2– Ієрархічна структура системи промптів

Базовий промпт (BASE_PROMPT) містить багатоваріантні інструкції, що застосовуються незалежно від обраного рівня чутливості. Ключовими елементами базового промпту є визначення ролі моделі, категоризація типів помилок, специфікація формату відповіді та інструкції щодо гранулярного виявлення помилок.

У Лістингу 2.3 можна побачити як ініціалізується визначення ролі моделі, завдяки встановлюванню контексту взаємодії, через що відбувається вплив на стиль відповідностей.

Лістинг 2.3 – Визначення ролі моделі

```
ROLE_DEFINITION = """
Ти — технічний редактор з досвідом роботи з українськими текстами.
Твоє завдання — знайти та класифікувати помилки у наданому тексті.
```

Система використовує чотири категорії помилок для класифікації:

- граматична – помилки у відмінках, узгодженні, прийменниках;
- пунктуаційна – пропущені або зайві розділові знаки;
- орфографічна – друкарські помилки, неправильні літери;
- стилістична – русизми, суржик, невідповідний стиль.

Granular Error Detection (гранулярне виявлення помилок) – це методологічний підхід, що вимагає від моделі ідентифікувати кожну помилку окремо, замість групування кількох помилок в одне виправлення. Такий підхід суттєво покращує користувацький досвід, оскільки дозволяє користувачу бачити та вирішувати кожну проблему індивідуально.

Порівняння підходів до виявлення помилок наведено у рисунку 2.3.

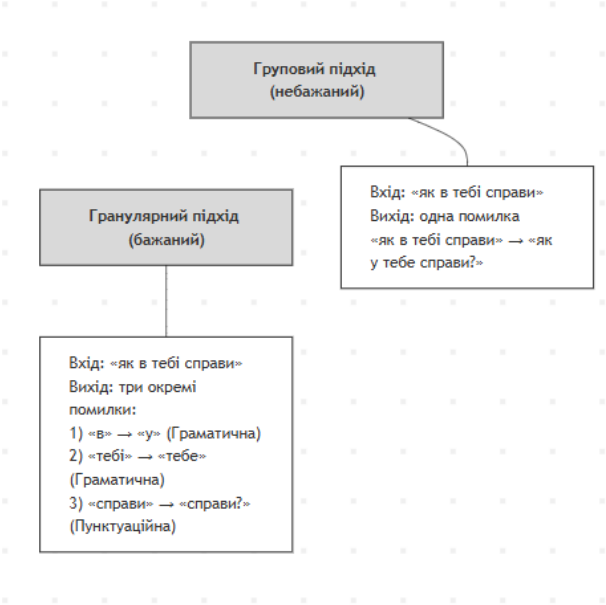


Рисунок 2.3 – Порівняння підходів до виправлення помилок

Інструкції для гранулярного виявлення формалізовані у базовому промпті показано у лістингу 2.4.

Лістинг 2.4 – Інструкції Granular Error Detection

```
GRANULAR_INSTRUCTIONS = """
ВАЖЛИВІ ІНСТРУКЦІЇ (Granular Error Detection):
```

```
1. НЕ ГРУПУЙ ПОМИЛКИ! Кожна помилка — окремий об'єкт у масиві.

2. Поле "original" містить ТІЛЬКИ слово з помилкою
   (максимум 1-2 слова, ніколи не ціле речення).

3. ПУНКТУАЦІЯ — окремі помилки для кожного знаку:
   - Пропущена кома після звертання
   - Пропущений знак питання в кінці

4. Максимально ДРОБИ помилки на атомарні одиниці.
```

Система підтримує три рівні чутливості, що дозволяє адаптувати глибину аналізу до потреб конкретного завдання. Кожен рівень реалізується як розширення базового промпту додатковими інструкціями.

М'який режим призначений для швидкої перевірки та виявлення лише критичних помилок, що суттєво впливають на зрозумілість тексту. Цей режим ідеально підходить для неформального листування або чернеток.

Що виявляє м'який режим: орфографічні помилки (друкарські помилки, неправильні літери), грубі граматичні помилки, що змінюють сенс, відсутність крапок у кінці речень.

Що ігнорує м'який режим: стилістичні нюанси, дрібні пунктуаційні помилки (коми, тире), можливі альтернативні формулювання.

Помірний режим є режимом за замовчуванням та забезпечує оптимальний баланс між повнотою перевірки та кількістю виявлених помилок. Цей режим рекомендовано для більшості випадків використання, включаючи ділове листування та академічні тексти.

Помірний режим виявляє: всі орфографічні помилки, граматичні помилки (відмінки, прийменники, узгодження), пунктуаційні помилки (коми, крапки, знаки питання), очевидні стилістичні помилки (русизми, суржик).

Суворий режим забезпечує максимально ретельну перевірку та призначений для професійного редагування важливих документів. Окрім усіх перевірок помірного режиму, суворий режим додатково аналізує стилістичну узгодженість, пропонує альтернативні формулювання та виявляє дрібні недоліки пунктуації.

Технічна реалізація цих режимів базується на механізмі динамічної ін'єкції інструкцій у системний промпт (System Prompt Injection). Залежно від обраного користувачем параметра sensitivity у клієнтському інтерфейсі, контролер сервера обирає відповідний шаблон інструкцій зі словника стратегій. Це дозволяє використовувати єдину базову модель (наприклад, GPT-4o) для різних сценаріїв використання без необхідності додаткового донавчання (fine-tuning).

Такий підхід також впливає на споживання токенів:

- м'який режим генерує менш об'ємні відповіді, фокусуючись лише на критичних виправленнях, що зменшує латентність системи;

- суворий режим спонукає модель до генерації більш розгорнутих пояснень та альтернативних варіантів, що збільшує час обробки, але підвищує якість редагування для відповідальних документів.

Порівняльна таблиця режимів роботи представлена у таблиці 2.4.

Таблиця 2.4 – Порівняння рівнів чутливості

Тип перевірки	**SOFT**	**MODERATE**	**STRICT**
Орфографія	+	+	+
Груба граматика	+	+	+
Крапки в кінці	+	+	+
Коми та пунктуація	–	+	+
Прийменники	–	+	+
Русизми та суржик	–	+	+
Стилістика	–	–	+

Для забезпечення передбачуваності та легкості обробки результатів система використовує структурований формат відповіді JSON. Базовий промпт, зображений на лістингу 2.5. містить чітку специфікацію очікуваної структури, що гарантує узгодженість відповідей незалежно від версії моделі чи конкретного запиту.

Лістинг 2.5 – Структура JSON відповіді

```
{
  "corrected_text": "Повністю виправлений текст",
  "errors": [
    {
      "original": "слово_з_помилкою",
      "suggestion": "виправлений_варіант",
      "type": "Граматична | Пунктуаційна | ...",
      "explanation": "Пояснення причини помилки"
    }
  ]
     }
```

Опис полів структури відповіді представлено у таблиці 2.5.

Таблиця 2.5 – Специфікація полів JSON відповіді

Поле	Тип	Опис
corrected_text	string	Повний текст з усіма застосованими виправленнями
errors	array	Масив об'єктів, кожен описує одну знайдену помилку
errors[].original	string	Оригінальний фрагмент з помилкою (1-2 слова)
errors[].suggestion	string	Запропонований виправлений варіант
errors[].type	string	Категорія помилки (українською)
errors[].explanation	string	Коротке пояснення для навчання користувача

Формування фінального промпту здійснюється функцією-селектором, яка на основі обраного рівня чутливості повертає відповідну комбінацію базового промпту та специфічного розширення. Алгоритм представлено у лістингу 2.6.

Лістинг 2.6 – Функція вибору промпту

```
function getSystemPrompt(sensitivity) {
    // Визначення відповідності режимів та розширень
    PROMPT_MAP = {
        "soft":     BASE_PROMPT + SOFT_EXTENSION,
        "moderate": BASE_PROMPT + MODERATE_EXTENSION,
```

```
        "strict":   BASE_PROMPT + STRICT_EXTENSION
    }

    // Повернення з fallback на помірний режим
    return PROMPT_MAP.get(sensitivity, PROMPT_MAP["moderate"])
     }
```

Окрім тексту промпту, важливу роль відіграють параметри конфігурації моделі. Ключовим параметром є temperature, що контролює рівень «креативності» відповідей.

Для задачі граматичної корекції обрано значення temperature = 0.2, що забезпечує: високу передбачуваність результатів (модель надає узгоджені виправлення для однакових помилок), мінімальну варіативність (уникнення «креативних», але неправильних виправлень), достатню гнучкість для адаптації до контексту речення. Код конфігурації запиту до мовної моделі представлено у лістингу 2.7.

Лістинг 2.7 – Конфігурація запиту до мовної моделі

```
function analyzeText(text, sensitivity, modelType) {
    systemPrompt = getSystemPrompt(sensitivity)

    requestConfig = {
        model: modelType,
        temperature: 0.2,          // Низька варіативність
        response_format: "json",   // Структурована відповідь
        messages: [
            { role: "system", content: systemPrompt },
            { role: "user", content: text }
        ]
    }

    response = sendToLanguageModel(requestConfig)
    return parseJSON(response.content)
}
```

Метод POST реалізує основний функціонал системи. Логіка обробки запиту побудована за чітким алгоритмом, що включає етапи валідації, обробки та

формування відповіді.

Етап 1 – це отримання та валідація даних. На цьому етапі сервер отримує JSON-навантаження із тіла запиту. Реалізовано багаторівневу систему валідації:

- сервер переконується, що тіло запиту не порожнє і містить обов'язкове поле text. У випадку відсутності даних повертається помилка 400 Bad Request;

- перевірка довжини текста застосовується завдяки константі MAX_TEXT_LENGTH = 10000. Це обмеження продиктоване лімітами контекстного вікна моделей GPT та необхідністю контролювати вартість запитів. Якщо текст перевищує ліміт, обробка переривається;

- параметри sensitivity та model перевіряються на належність до списків допустимих значень (VALID_SENSITIVITIES, VALID_MODELS). Якщо клієнт надіслав некоректний параметр (наприклад, неіснуючу назву моделі), сервер повертає детальне повідомлення про помилку.

Алгоритм отриманняданих зображено на рис 2.4.

Алгоритм валідації даних зображено на рис 2.5.

Рисунок 2.4 – Алгоритм отримання даних

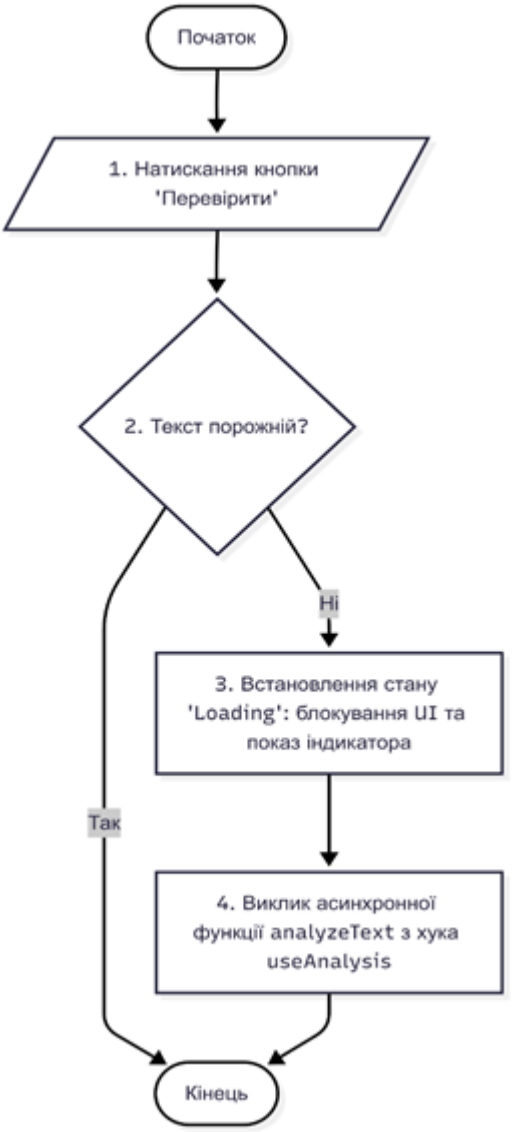


Рисунок 2.5 – Алгоритм валідації даних

Етап 2 відповідає за виконання бізнес-логіки. Після успішної валідації контролер звертається до фабрики моделей ModelFactory.get_model(model_type) для отримання екземпляра необхідної моделі. Для моніторингу продуктивності фіксується час початку обробки. Далі викликається метод model.correct(user_text, sensitivity), який ініціює взаємодію з зовнішнім API. Важливо зазначити, що цей виклик є блокуючим у поточній реалізації синхронних воркерів Gunicorn.

Алгоритм роботи бізнес-логіки зображено на рис 2.6.

Рисунок 2.6 – Алгоритм бізнес-логіки серверу

Етап 3 відповідає за формування відповідей та обробку помилок. Отриманий від моделі результат збагачується метаданими: додається назва використаної моделі, розрахований час обробки (в мілісекундах) та довжина вхідного тексту. Це дозволяє клієнтському додатку відображати статистику роботи. Весь процес обробки загорнуто у блок try-except, що забезпечує перехоплення будь-яких виключень (мережеві помилки, помилки парсингу JSON, внутрішні помилки OpenAI). У разі виникнення позаштатної ситуації сервер гарантовано повертає валідну JSON-відповідь зі статусом 500 та описом проблеми, що дозволяє

клієнтському інтерфейсу коректно повідомити користувача про збій.

Алгоритм обробки помилок зображено на рис 2.7.

Алгоритм формування відповіді до клієнта зображено на рис 2.8.

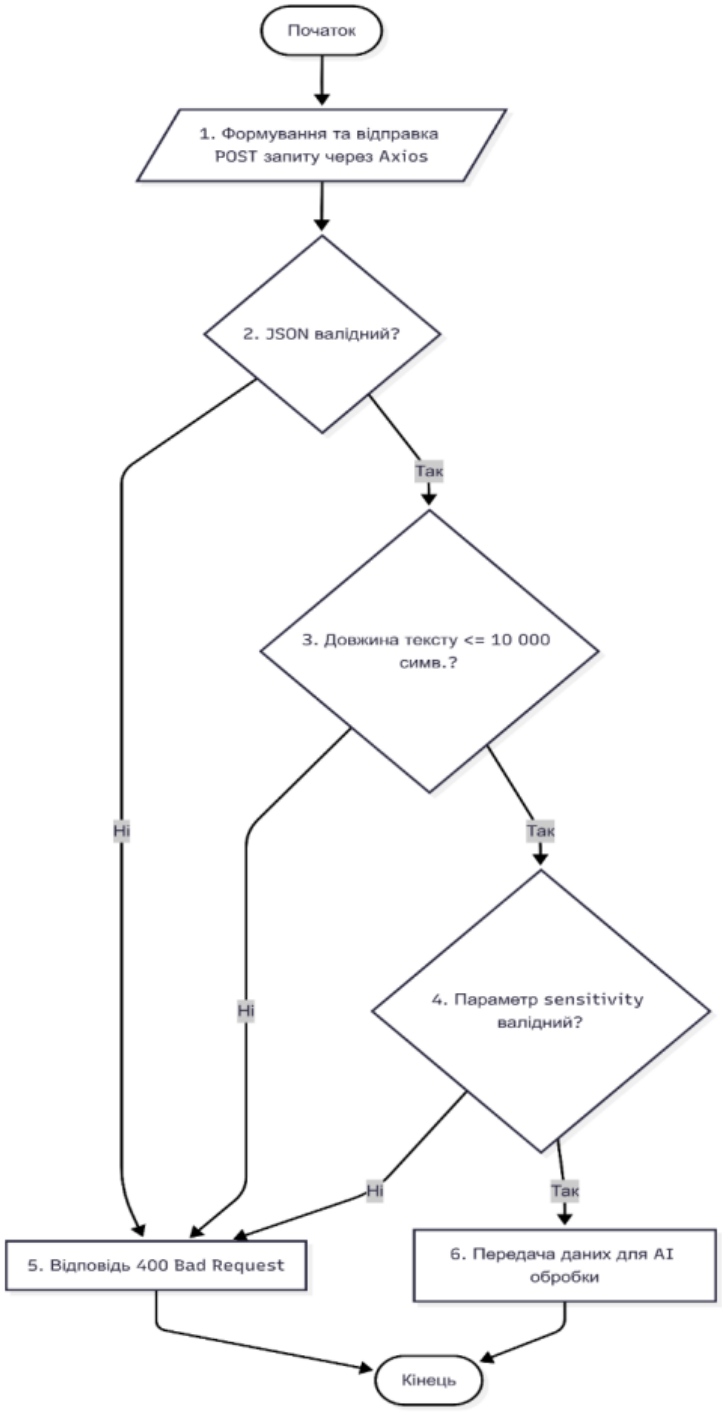


Рисунок 2.7 - Алгоритм обробки помилок

Рисунок 2.8 – Алгоритм формування відповіді до клієнта

Базовий клас GrammarModel визначає абстрактний контракт. Конкретні реалізації, такі як GPT35TurboModel, містять метод correct, який конструює запит до API chat.completions.create. У цьому методі відбувається динамічне формування списку повідомлень (messages). Першим елементом завжди йде повідомлення з роллю system, зміст якого визначається через виклик self.get_system_prompt(sensitivity). Другим елементом є повідомлення з роллю user,

що містить текст для перевірки. Така структура дозволяє чітко розділити інструкції для моделі та дані, над якими вона має працювати, мінімізуючи ризик "ін'єкції промпта" (Prompt Injection).

2.4 Оцінка ефективності системи промптів

Оскільки система базується на використанні зовнішніх LLM, критичними факторами її роботи є латентність (час відгуку) та вартість експлуатації. Просте звернення до API для кожного запиту без додаткових інженерних рішень робить систему економічно неефективною та повільною. Тому до архітектури системи було інтегровано спеціалізовані механізми оптимізації продуктивності: стратегію кешування та алгоритм мінімізації контексту.

Для зменшення кількості звернень до платного API та миттєвої видачі результатів для повторюваних запитів у систему впроваджено паттерн Cache-Aside (Lazy Loading).

Архітектурне рішення полягає у перехопленні запиту до відправки на LLM. Система генерує унікальний хеш вхідного тексту та параметрів перевірки (рівень чутливості). Якщо результат для такого хешу вже існує у базі даних, він повертається користувачеві миттєво, оминаючи повільну генерацію моделі.

Математичне обґрунтування доцільності впровадження кешування базується на моделі середнього часу доступу яка визначається формулою:

$$T_{avg} = P_{hit} \cdot T_{cache} + P_{miss} \cdot T_{origin} \quad (2.1)$$

де T_{avg} — середній час доступу; P_{hit}— ймовірність попадання в кеш (cache hit rate); T_{cache} — час доступу до кешованих даних; P_{miss} — ймовірність промаху кешу; T_{origin}— час доступу до оригінального джерела (API моделі).

Другим вектором забезпечення ефективності є мінімізація кількості токенів,

що передаються моделі. Вартість та час генерації лінійно залежать від розміру вхідних та вихідних даних.

Для цього в архітектуру промптів закладено принцип максимальної інформаційної щільності. Замість використання природної мови для опису всіх правил, застосовано підхід «Few-Shot Learning» з мінімалістичними прикладами та видаленням структурної надмірності.

Ефективність промпту E визначається як відношення якості результату Q до розміру промпту в токенах S:

$$E = \frac{Q}{S} \tag{2.2}$$

де E якість результату; Q - розмір промпту; S – кількість токенів.

Такий підхід дозволяє реалізувати функціонал гранулярного виявлення помилок без перевищення лімітів пропускної здатності (Rate Limits) та з дотриманням вимог щодо швидкодії real-time застосунку.

3 РОЗРОБКА СИСТЕМИ

3.1 Архітектура системи

Необхідність використання саме клієнт-серверної архітектури для комп’ютерної системи зумовлена специфікою роботи з великими мовними моделями (LLM). Запуск сучасних генеративних моделей (таких як GPT-4) безпосередньо на пристрої клієнта є технічно неможливим через високі вимоги до обчислювальних потужностей (GPU/TPU) та пам'яті. Тому сервер виступає необхідною проміжною ланкою (middleware), що інкапсулює логіку взаємодії з AI, управління ключами доступу та формування промптів.

Хоча система може виглядати як монолітний сервіс з точки зору розгортання,

її внутрішня структура та принципи взаємодії тяжіють до сервіс-орієнтованого підходу. Система не є ізольованою сутністю та інтегрується з зовнішніми API (OpenAI) для виконання задач обробки природної мови. Це відповідає концепції компонування сервісів, де основний сервер виступає адміністратором запитів.

Внутрішня архітектура серверної частини побудована з використанням патерну "Абстрактна фабрика" (ModelFactory), що дозволяє динамічно створювати екземпляри різних AI-моделей (GPT35TurboModel, GPT4oModel). Така слабка зв'язність компонентів (loose coupling) є характерною ознакою архітектур, готових до переходу на мікросервіси, де кожна модель могла б теоретично бути винесена в окремий сервіс.

Взаємодія реалізована через RESTful API та формат JSON. Це забезпечує чіткий поділ: Front-end відповідає за інтерфейс та візуалізацію, а Back-end — за бізнес-логіку та доступ до AI-моделей.

Графічне представлення роботи архітектури системи зображоно на рис 3.1

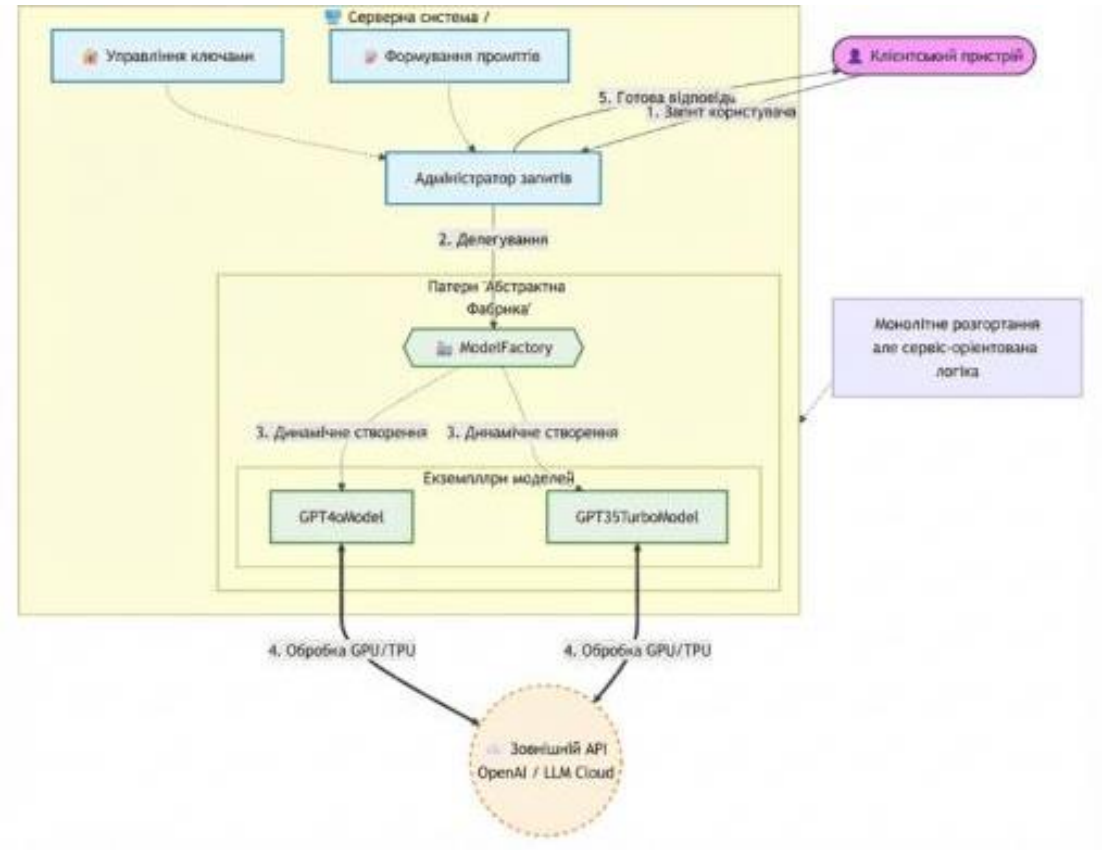


Рисунок 3.1 – Графічне представлення запропонованої архітектури

3.2 Розробка клієнтської частини застосунку

Вибір інструментальних засобів для реалізації клієнтської частини системи базувався на детальному аналізі вимог до функціональності, продуктивності та надійності програмного забезпечення.

В якості фундаментальної платформи для розробки клієнтської частини було обрано фреймворк Next.js версії 16.0.10 [29]. Цей вибір є стратегічним рішенням, що дозволяє вирішити комплекс технічних завдань, притаманних сучасним вебзастосункам класу SPA (Single Page Application) з елементами SSR (Server-Side Rendering).

Використання архітектури App Router, яка є стандартом для Next.js 16, дозволило реалізувати гібридну модель рендерингу. Це означає, що система може гнучко поєднувати серверні компоненти для статичного контенту та клієнтські компоненти для інтерактивних елементів інтерфейсу, таких як текстовий редактор. Такий підхід суттєво зменшує обсяг JavaScript-коду, що передається на клієнтську сторону, оскільки значна частина логіки розмітки виконується на сервері під час збірки або запиту.

Ключові переваги Next.js 16, що стали визначальними для проєкту:

- Server-Side Rendering (SSR) та Static Site Generation (SSG) забезпечують миттєве відображення першого екрану (First Contentful Paint), що критично важливо для утримання уваги користувача. Для системи перевірки граматики це дозволяє швидко завантажити оболонку редактора, поки у фоновому режимі ініціалізуються важкі скрипти аналізу;

- вбудовані механізми бандлінгу та мінімізації автоматично розділяють код на менші частини (chunks), завантажуючи їх лише за потреби. Це особливо актуально при використанні великих бібліотек для редагування тексту для динамічного завантаження;

- App Router спрощує структуру проєкту, дозволяючи інтуїтивно зрозуміло організувати сторінки, макети (layouts) та API-маршрути;

- Next.js дозволяє створювати серверні функції безпосередньо в проєкті, що спрощує взаємодію з бекендом, приховуючи реальні ендпоїнти API та ключі доступу.

Клієнтська логіка побудована на базі бібліотеки React версії 19.2.1 [30]. Вибір найновішої версії React зумовлений необхідністю використання передових механізмів управління станом та рендерингом, які були представлені у цій версії.

Найважливішим нововведенням, що використовується в проєкті, є React Compiler (підключений через babel-plugin-react-compiler версії 1.0.0). Цей інструмент автоматизує процес мемоізації компонентів та обчислень. У попередніх версіях розробники були змушені вручну використовувати хуки useMemo та useCallback для запобігання зайвим ререндерам, що ускладнювало код. React 19 бере цю задачу на себе, автоматично аналізуючи граф залежностей. Для текстового редактора, де кожне натискання клавіші викликає оновлення стану, це забезпечує плавність роботи інтерфейсу без мікро-затримок.

Крім того, React 19 покращує роботу з асинхронними операціями через нові хуки та API переходів (Transitions API), що дозволяє робити інтерфейс чутливим навіть під час виконання важких обчислень або очікування відповіді від сервера, що можна побачити у лістингу 3.1.

Лістингу 3.1 – Код роботи з API

```
export const API_CONFIG = {
  BASE_URL: process.env.NEXT_PUBLIC_API_URL,
  ENDPOINTS: {
    ANALYZE: '/api/analyze',
    MODELS: '/api/models',
    HEALTH: '/api/health',
  },
  TIMEOUT: 30000,
} as const;

export function getApiUrl(endpoint: keyof typeof API_CONFIG.ENDPOINTS): string {
```

```
        return
`${API_CONFIG.BASE_URL}${API_CONFIG.ENDPOINTS[endpoint]}`;
    }
```

Для забезпечення надійності, масштабованості та зручності підтримки кодової бази використано мову програмування TypeScript 5. У проєктах зі складною бізнес-логікою, якою є система аналізу тексту, статична типізація виступає як перша лінія оборони від помилок.

Використання TypeScript дозволило строго формалізувати структури даних, що курсують між компонентами та зовнішнім API. Наприклад, було створено чіткі інтерфейси для об'єктів помилок (GrammarError), налаштувань користувача (Settings) та результатів аналізу (AnalysisResult). Це гарантує, що будь-яка невідповідність у структурі даних (наприклад, відсутність поля suggestion у відповіді сервера) буде виявлена ще на етапі компіляції, а не призведе до падіння застосунку у користувача (runtime error) [31].

Конфігурація компілятора (tsconfig.json) налаштована в режимі суворої типізації ("strict": true), що забороняє використання неявного типу any та змушує розробника явно обробляти випадки відсутності даних (null або undefined). Це підвищує стабільність роботи системи, особливо при обробці непередбачуваних відповідей від AI-моделей [31].

Для реалізації візуальної складової інтерфейсу було обрано гібридний підхід, що поєднує гнучкість utility-first CSS фреймворку та потужність готової бібліотеки компонентів.

Tailwind CSS версії 4 використовується як основний інструмент для верстки макету, управління відступами, кольорами та типографікою [32]. Його підхід дозволяє швидко створювати адаптивні інтерфейси без необхідності перемикання між файлами коду та стилів. Версія 4 відрізняється покращеним рушієм JIT (Just-In-Time), який генерує CSS-класи в реальному часі під час компіляції, що забезпечує мінімальний розмір підсумкового CSS-файлу та високу швидкість завантаження сторінок. Tailwind також дозволяє легко реалізувати темну тему та

адаптивність під мобільні пристрої (mobile-first), що є однією з ключових вимог до системи.

Material-UI (MUI) версії 7.3.6 інтегровано для реалізації складних інтерактивних елементів інтерфейсу, таких як модальні вікна, перемикачі, слайдери налаштувань та іконки. Розробка таких компонентів з нуля вимагає значних ресурсів для забезпечення кросбраузерності та доступності (accessibility). Використання MUI дозволило зосередитися на бізнес-логіці, отримавши при цьому професійний та стандартизований вигляд елементів управління, знайомий більшості користувачів [33].

Для вирішення специфічних задач у проєкт було інтегровано ряд спеціалізованих бібліотек:

- Axios 1.13.2 використовується як HTTP-клієнт для взаємодії з бекендом. Він надає розширені можливості порівняно зі стандартним fetch, такі як автоматична трансформація JSON, перехоплювачі запитів/відповідей (interceptors) та просте налаштування тайм-аутів, що критично важливо при очікуванні відповіді від повільних AI-моделей [34];

- @uiw/react-codemirror 4.25.4 - спеціалізований компонент редактора коду. Він забезпечує підсвічування синтаксису, нумерацію рядків та високу продуктивність при роботі з великими текстовими файлами, що неможливо реалізувати за допомогою стандартного textarea [35];

- ESLint 9 & Prettier - інструменти для лінтингу та форматування коду, що забезпечують дотримання єдиного стилю кодування в команді та автоматичне виявлення потенційних проблем [36].

Лістинг 3.2 – Код роботи клієнта з групування та представлення помилок

```
export function useAnalysis() {
  const [result, setResult] = useState<AnalysisResult | null>(null);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState<string | null>(null);
```

```
      const analyzeText = useCallback(async (text: string, settings:
Settings) => {
        if (!text.trim()) {
          setError('Будь ласка, введіть текст для перевірки');
          return;
        }

        setLoading(true);
        setError(null);
        setResult(null);

        try {
          const response = await axios.post(getApiUrl('ANALYZE'), {
            text,
            sensitivity: settings.sensitivity || 'moderate',
            model: settings.model || 'gpt-3.5-turbo'
          });

          const data = response.data as AnalysisResult;

            return true;
          });
```

Зовнішній вигляд інтерфейсу представлено на рис. 3.1.

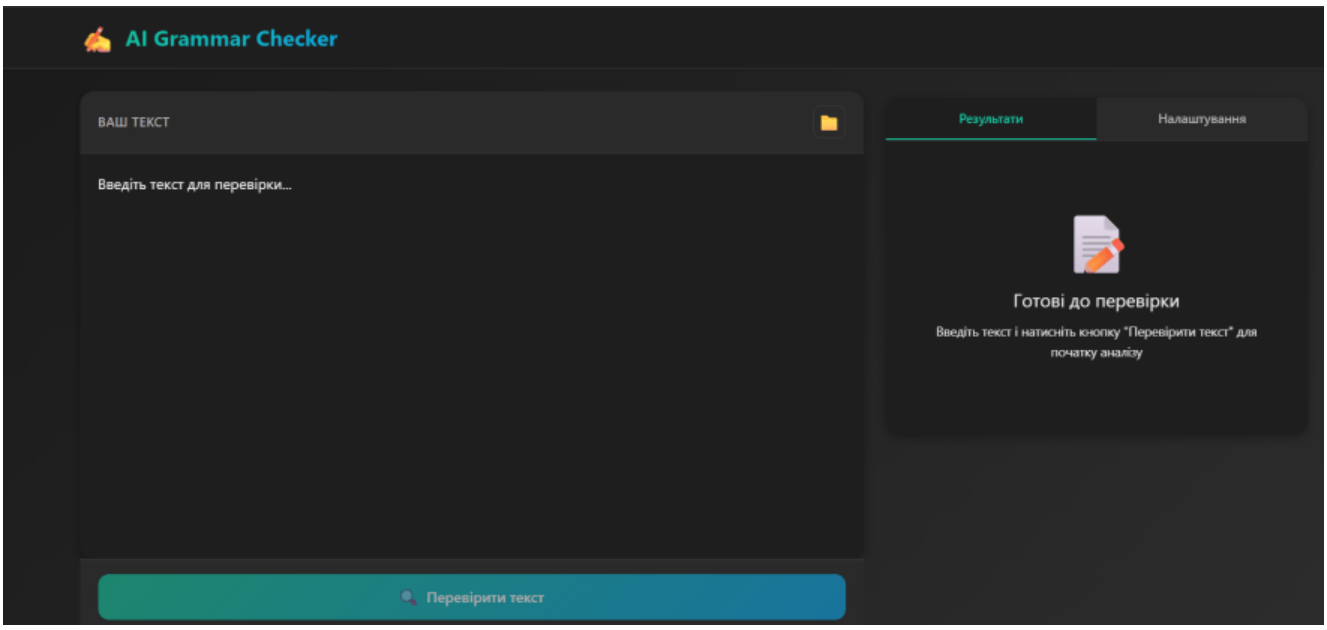


Рисунок 3.2 – Зовнішній вигляд інтерфейсу

3.3 Розробка серверної частини застосунку

Серверна частина системи, що проектується, виступає центральним вузлом обробки даних, який приймає запити від клієнтського інтерфейсу, виконує попередню обробку тексту, взаємодіє з віддаленими інтелектуальними агентами та повертає структуровані результати аналізу.

В якості базової мови програмування для реалізації серверної логіки було обрано Python версії 3.11 або вище. Цей вибір є стратегічним та обґрунтованим низкою технічних та економічних факторів, що визначають успішність сучасних проектів у сфері обробки природної мови (Natural Language Processing — NLP) [37].

По-перше, Python займає домінуючу позицію в екосистемі розробки штучного інтелекту. Це забезпечує нативну підтримку більшості SDK (Software Development Kit) провідних лабораторій AI, включаючи OpenAI, Anthropic та Google. Використання Python дозволяє розробникам інтегрувати складні алгоритми машинного навчання без необхідності створення проміжних шарів сумісності або використання низькоефективних адаптерів (wrappers), що могли б негативно

вплинути на швидкодію системи. Пряма інтеграція з бібліотеками, такими як openai, дозволяє використовувати новітні функції API (наприклад, JSON Mode або Function Calling) одразу після їх релізу [37].

По-друге, синтаксична структура Python сприяє написанню лаконічного та зрозумілого коду, що є критично важливим для проектів з академічною складовою, де код часто виступає об'єктом дослідження та подальшої модифікації. Висока читабельність коду спрощує процес налагодження (debugging) та рефакторингу, а також знижує поріг входження для нових членів команди розробки. Статична типізація, що реалізується через модуль typing, дозволяє виявляти потенційні помилки на етапі написання коду, підвищуючи загальну надійність серверної частини [37].

По-третє, Python володіє однією з найбагатших екосистем допоміжних бібліотек для веб-розробки. У проекті активно використовуються такі модулі, як requests для виконання синхронних HTTP-запитів до зовнішніх сервісів, json для високоефективної серіалізації та десеріалізації даних, а також python-dotenv для безпечного управління конфігураційними змінними середовища, що дозволяє відокремити секретні дані (API keys) від вихідного коду [37].

Для реалізації архітектурного стилю REST API було проведено порівняльний аналіз сучасних веб-фреймворків Python, за результатами якого вибір було зупинено на Flask (версія 3.0.0). Це рішення базується на необхідності створення легкої, гнучкої та високопродуктивної системи, позбавленої надлишкової складності [38].

На відміну від "monolithic" фреймворків, таких як Django, які пропонують жорстко визначену структуру проекту та велику кількість вбудованих компонентів (ORM, система шаблонів, адмін-панель), Flask сповідує філософію мінімалізму. Для завдання створення проксі-сервера, що маршрутизує запити до AI-моделей, функціонал Django був би надлишковим і створював би додаткове навантаження на системні ресурси. Flask надає розробнику повний контроль над архітектурою додатку, дозволяючи підключати лише ті розширення, які дійсно необхідні для вирішення конкретних задач (наприклад, Flask-CORS для обробки крос-доменних

запитів) [38].

Ключовою перевагою Flask у контексті даного проекту є його висока продуктивність при обробці I/O-bound операцій, якими є мережеві запити до API OpenAI. Мінімальні накладні витрати фреймворку (overhead) дозволяють скоротити час обробки внутрішньої логіки до мінімуму, залишаючи основний часовий бюджет на роботу нейромережі [38].

Важливим аспектом проектування серверної архітектури є вибір WSGI-сервера (Web Server Gateway Interface) для запуску додатку у продуктивному середовищі. Вбудований сервер Flask є однопотоковим і не призначений для обробки реального трафіку. Тому для забезпечення стабільності та паралельної обробки запитів було обрано сервер Gunicorn (версія 21.2.0) [39].

Наукове обґрунтування використання Gunicorn базується на його архітектурній моделі "Pre-fork worker model". Головний майстер-процес Gunicorn породжує задану кількість робочих процесів (workers), кожен з яких здатний незалежно обробляти вхідні запити. Це дозволяє ефективно утилізувати багатоядерні процесори серверного обладнання та забезпечувати паралелізм (concurrency) навіть при використанні синхронного коду Python, який обмежений Global Interpreter Lock (GIL).

Для комп’ютерної системи для перевірки та коригування тексту, де час обробки одного запиту може складати кілька секунд через складність генерації відповіді LLM, Gunicorn забезпечує критично важливу функцію керування процесами. Якщо один воркер заблокований очікуванням відповіді від OpenAI, інші воркери продовжують приймати запити від користувачів, що запобігає повному блокуванню сервісу. Крім того, механізми автоматичного перезапуску воркерів (process reaping) гарантують високу доступність системи навіть у випадку виникнення непередбачуваних помилок або витоку пам'яті.

Структура серверного додатку організована у вигляді набору спеціалізованих модулів, кожен з яких відповідає за окремий аспект функціонування системи :

- app.py (Контролер та маршрутизатор) - є точкою входу в додаток;
- models.py (Бізнес-логіка моделей) - модуль, що інкапсулює логіку взаємодії

з різними типами AI-моделей. Тут реалізовано класи-адаптери для конкретних версій GPT (GPT-3.5, GPT-4o), а також фабричний метод для їх динамічного створення. Цей шар абстрагує деталі API OpenAI від решти системи;

- prompts.py (Логіка інструкцій) - спеціалізований модуль управління промптами. Він містить шаблони системних інструкцій та реалізує логіку вибору стратегії аналізу тексту (Soft, Moderate, Strict) залежно від налаштувань користувача, що дозволяє централізовано керувати поведінкою штучного інтелекту без необхідності модифікації коду моделей;

- конфігураційні файли (.env, requirements.txt), що забезпечують управління залежностями проекту та параметрами середовища виконання, що є необхідним для відтворюваності розгортання.

Модуль app.py ініціалізує екземпляр Flask, налаштовує глобальну конфігурацію (включаючи параметри безпеки та ліміти), реєструє обробники маршрутів (routes) та виконує первинну валідацію вхідних даних. Його основна задача — прийняти HTTP-запит, перевірити його коректність та передати управління відповідним сервісним класам

Реалізацію app.py представлено у лістингу 3.3, де можна побачити імпорти усіх описаних раніше компонентів та роботу підключення до класу model[‘POST’].

Реалізацію prompts.py представлено у лістингу 3.4, де прописано інструкції BASE_PRONPT.

Лістинг 3.3 – Код app.py

```
import os
import traceback
from flask import Flask, request, jsonify
from flask_cors import CORS
from config import MAX_CONTENT_LENGTH, MODEL_DESCRIPTIONS, MODEL_NAMES, ENABLE_WARMUP
from models import ModelFactory

app = Flask(__name__)
CORS(app, resources={r"/api/*": {"origins": "*"}})
```

```
@app.route('/api/analyze', methods=['POST'])
def analyze_text():
    try:
        data = request.get_json()

        error_response = validate_request_data(data)
        if error_response:
            return error_response

        user_text = data.get('text', '').strip()
        sensitivity = data.get('sensitivity', 'moderate')
        model_type = data.get('model', 'gpt-3.5-turbo')

        result = process_text(user_text, sensitivity,
model_type)
        return jsonify(result), 200

    except Exception as e:
        print(f"Error in analyze_text: {e}")
        traceback.print_exc()
        return jsonify({
            "error": "Сталася помилка на сервері",
            "details": str(e)
        }), 500
```

Лістинг 3.4 – Код prompts.py

```
BASE_PROMPT_EN = """
You are a Ukrainian text editor. Your task is to find errors in
Ukrainian text.

CRITICAL INSTRUCTIONS (Granular Error Detection):
1. Don't group errors! If a sentence has a preposition error AND
a word error - these are TWO separate errors.
2. Field "original" must contain ONLY the word/symbol with error
```

```
(max 1-2 words), not entire phrase.
     3. PUNCTUATION - separate errors for each missing mark:
        - For MISSING commas/exclamations/questions: "original" should
contain the SPACE between words where mark should be.
        - Example 1: "Привіт Кириле" -> original: "Привіт Кириле"
(space between), suggestion: "Привіт, Кириле," (with commas)
        - Example 2: "як справи" (end of sentence) -> original:
"справи", suggestion: "справи?" (with mark)
        - Each missing comma/mark is a SEPARATE error type
"Пунктуаційна"
     4. Split errors maximally:
        - Wrong: original: "як в тебі справи" -> suggestion: "як у
тебе справи"
        - Correct 1: original: "в" -> suggestion: "у" (Граматична)
        - Correct 2: original: "тебі" -> suggestion: "тебе"
(Орфографічна)

     Categories (MUST use Ukrainian names in output):
     1. Граматична - wrong cases, prepositions, agreement
     2. Пунктуаційна - commas, periods, question marks
     3. Орфографічна - typos, wrong letters
     4. Стилістична - russianisms, surzhyk (if not fixed as spelling)
     5. Незрозуміле слово - incorrect, invented, or incomprehensible
words that make no sense in Ukrainian
```

Для забезпечення гнучкості архітектури та можливості подальшого розширення функціоналу без суттєвих змін існуючого коду (принцип Open-Closed Principle), у проекті було імплементовано ряд фундаментальних патернів проектування.

Патерн "Абстрактна Фабрика" (Abstract Factory) Центральним елементом архітектури моделей є патерн "Абстрактна Фабрика". Його реалізація дозволяє системі створювати об'єкти моделей, не прив'язуючись до конкретних класів у коді контролера. Абстрактний базовий клас GrammarModel, що успадковується від ABC (Abstract Base Class), визначає єдиний інтерфейс для всіх моделей через

абстрактний метод correct(text, sensitivity). Це гарантує, що будь-яка нова модель, яка буде додана до системи, матиме стандартизований спосіб виклику. Клас ModelFactory виступає як фабрика, що містить статичний метод get_model(model_type). Цей метод приймає ідентифікатор моделі (рядок) і повертає відповідний екземпляр класу (GPT35TurboModel або GPT4oModel). Такий підхід значно спрощує додавання нових провайдерів AI (наприклад, Anthropic Claude або Google Gemini) у майбутньому.

Патерн "Одинак" (Singleton) Для оптимізації роботи з мережевими ресурсами у класі ModelFactory реалізовано патерн "Одинак" для управління клієнтом OpenAI. Ініціалізація клієнта (створення об'єкта openai.OpenAI) є відносно "дорогою" операцією, яка може включати зчитування змінних середовища та налаштування HTTP-сесії. Реалізація Singleton гарантує, що у межах одного робочого процесу існує лише один екземпляр клієнта (_client). Метод _get_client() перевіряє наявність створеного екземпляра і, за необхідності, створює його. Це дозволяє повторно використовувати існуючі TCP-з'єднання (Connection Pooling), зменшуючи затримки на встановлення нових з'єднань при кожному запиті.

Патерн "Стратегія" (Strategy) - це логіка вибору рівня перевірки тексту реалізована за допомогою патерну "Стратегія". Різні рівні чутливості (м'який, помірний, суворий) розглядаються як окремі стратегії поведінки системи. Функція get_system_prompt(sensitivity) у модулі prompts.py діє як контекст, який на основі вхідного параметра обирає відповідний алгоритм (текст системної інструкції) зі словника стратегій. Це дозволяє інкапсулювати логіку формування промптів і змінювати її незалежно від логіки виконання запитів до API.

При запуску додатку відбувається завантаження змінних середовища за допомогою load_dotenv(), що робить доступними API-ключі та налаштування конфігурації. Ініціалізація Flask-додатку супроводжується налаштуванням критично важливих параметрів.

Параметр JSON_AS_ASCII = False є специфічною вимогою для систем, що працюють з кириличними мовами. За замовчуванням Flask екранує всі не-ASCII символи в JSON (наприклад, перетворює "привіт" у \u043f\u0440...), що збільшує

розмір відповіді та ускладнює налагодження. Відключення цієї опції дозволяє передавати український текст у нативному вигляді UTF-8. Обмеження MAX_CONTENT_LENGTH до 16 МБ виступає як запобіжний захід безпеки проти атак типу Denial of Service (DoS), спрямованих на переповнення пам'яті сервера великими запитами.

Діаграма послідовності логіки роботи застосунку зображено на рис.3.7.

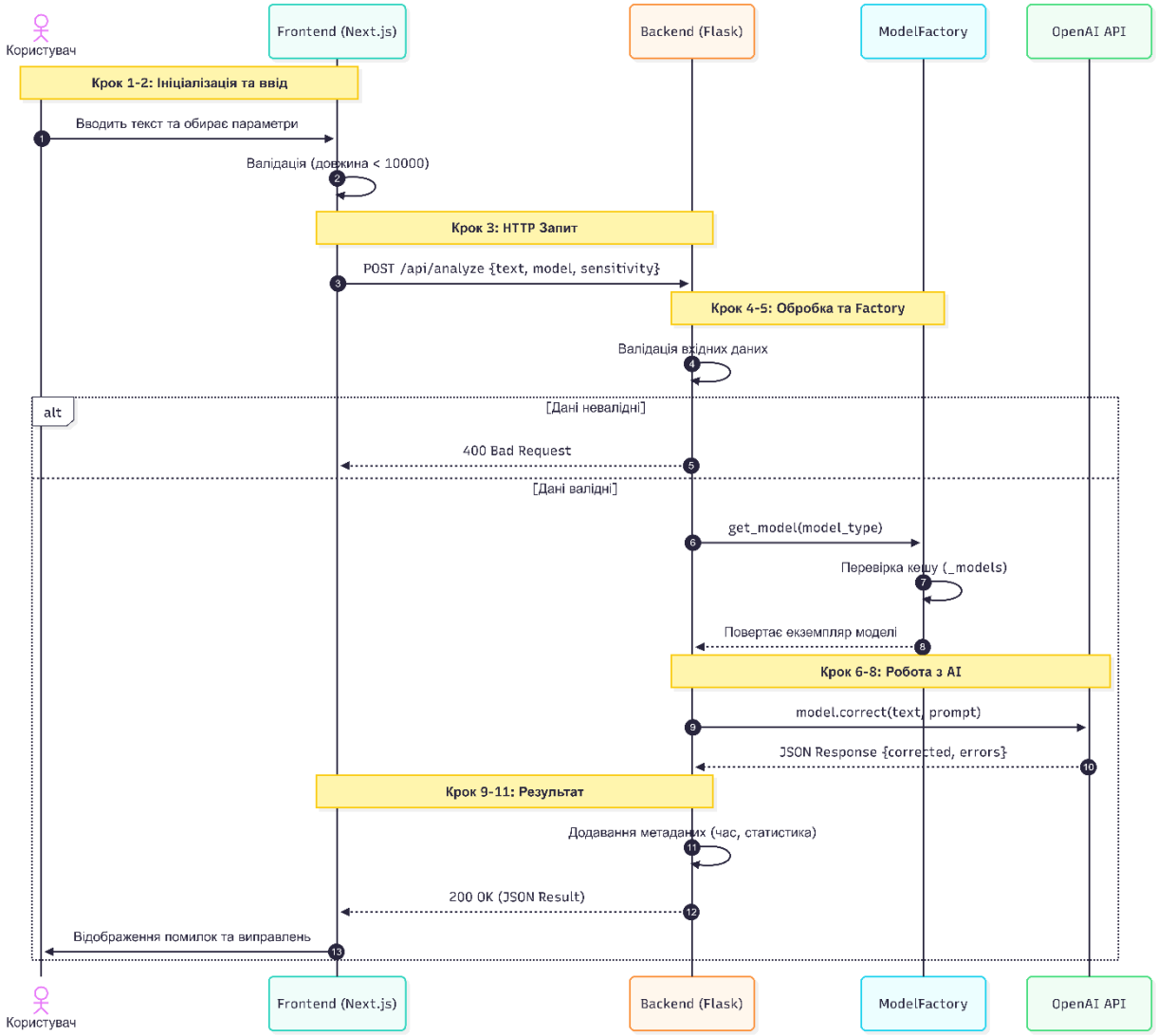


Рисунок 3.7 – Діаграма послідовності роботи застосунку

3.4 Розрахунок конфігурації обчислювальних ресурсів

Оскільки Backend виконує переважно функції оркестрації та передачі даних (I/O Bound operations), вимоги до обчислювальної потужності є помірними. На основі навантажувального тестування було визначено оптимальну конфігурацію.

Таблиця 3.1 — Розрахунок споживання ресурсів Backend-сервера

Компонент системи	**RAM (мін-макс)**	**CPU (vCPU)**	**SSD (Storage)**	**Обґрунтування використання ресурсів**
Python Runtime	100–150 MB	~0.1 vCPU	~500 MB	RAM та CPU витрачаються на базовий процес інтерпретатора; SSD займають файли віртуального оточення та бінарні файли мови.
Flask + Gunicorn	200–400 MB	0.5–1.0 vCPU	~100 MB	4 worker-процеси (RAM). Основне навантаження на CPU йде на серіалізацію JSON. Місце на диску мінімальне (лише файли вихідного коду).
Системні залежності	100–150 MB	~0.1 vCPU	~200 MB	Бібліотеки (`openai`, `cors`) завантажуються в RAM. CPU майже не використовується (режим очікування I/O від зовнішніх API). SSD потрібен для зберігання пакетів.
In-Memory Cache	200–400 MB	< 0.1 vCPU	~1 GB (Swap)	Зберігання 85,000 записів у RAM для швидкого доступу. SSD резервується під swap-файл для стабільності при пікових навантаженнях.
ОС (Container)	100–200 MB	~0.1 vCPU	2–5 GB	Мінімальні витрати RAM/CPU на ядро ОС. SSD критичний для швидкого запису системних логів та тимчасових файлів (`/tmp`).
РАЗОМ	~800–1300 MB	1–2 vCPU	5-10 GB	Рекомендована конфігурація: 2 vCPU, 2 GB RAM, 10 GB SSD (General Purpose) для забезпечення швидкого старту та запасу потужності.

Аналіз процесорного навантаження: Обробка одного запиту вимагає приблизно 75 мс часу CPU. При піковому навантаженні 1000 запитів на хвилину, потреба у процесорному часі складає 1.25 vCPU.

Таким чином, конфігурація з 2 vCPU забезпечує запас потужності у 60% для обробки фонових задач та сплесків трафіку.

Відсутність потреби у власному обладнанні для AI-обчислень (GPU) радикально змінює підхід до формування інфраструктури. Система спроєктована як Stateless (без збереження стану), що є ключовою вимогою для сучасних хмарних архітектур.

Згідно з концепцією Privacy by Design, система мінімізує збір даних [40]. Користувацькі тексти не зберігаються у постійній базі даних (Persistent Storage).

Вимоги до дискової підсистеми:

Тип носія SSD гарантує високу швидкість довільного доступу (IOPS) необхідної для швидкого запису логів та роботи з файловим кешем операційної системи. Обсяг мінімальний (5-10 GB), оскільки основний простір займає лише образ контейнера та тимчасові файли логів (Log Rotation policy — 30 днів).

3.5 Апаратні вимоги до системи

Оскільки розроблена система побудована на основі клієнт-серверної архітектури, вимоги до апаратного забезпечення розділяються на дві групи: вимоги до пристрою кінцевого користувача (клієнта), де виконується інтерфейсна частина на Next.js, та вимоги до сервера, де функціонує Python-бекенд та здійснюється комунікація з LLM.

Клієнтська частина реалізована як Single Page Application (SPA), що виконується у браузері користувача. Основне навантаження припадає на відображення інтерфейсу редактора (CodeMirror) та рендеринг компонентів React. Вимоги сформовані виходячи з необхідності забезпечення плавної роботи інтерфейсу без затримок (input lag).

Таблиця 3.2 – Апаратні вимоги до клієнтського пристрою

Характеристика	**Мінімальні вимоги**	**Рекомендовані вимоги**	**Примітка**
Центральний процесор (CPU)	2 ядра, 1.6 ГГц (наприклад, Intel Celeron / Core i3)	4 ядра, 2.0 ГГц і вище (Intel Core i5 / Apple Silicon M1)	Необхідно для роботи JavaScript-рушія браузера та React Compiler
Оперативна пам'ять (RAM)	4 ГБ	8 ГБ і більше	Сучасні браузери потребують значних обсягів пам'яті для кожної вкладки
Накопичувач (SSD/HDD)	500 МБ вільного простору	1 ГБ вільного простору	Використовується для кешування скриптів, стилів та локального сховища (LocalStorage)
Мережевий адаптер	5 Мбіт/с	20 Мбіт/с і вище	Стабільне з'єднання необхідне для відправки тексту та отримання JSON-відповідей
Програмне забезпечення	Google Chrome 90+, Firefox 90+, Safari 14+, Edge 90+	Остання стабільна версія браузера	Підтримка ES6+ та сучасних веб-стандартів

Серверна частина відповідає за валідацію, обробку запитів, кешування та взаємодію з OpenAI API. Згідно з розрахунками споживання ресурсів, проведеними у підрозділі 3.4, сервер не потребує GPU, оскільки генерація тексту відбувається на стороні провайдера API, проте критичним є швидкодія дискової підсистеми для логування та достатній обсяг RAM для роботи воркерів Gunicorn.

Таблиця 3.3 – Апаратні вимоги до сервера (Backend)

Характеристика	**Мінімальні вимоги**	**Рекомендовані вимоги**	**Примітка**
Центральний процесор (CPU)	1 vCPU	2 vCPU	Для забезпечення паралельної обробки запитів та серіалізації JSON 2
Оперативна пам'ять (RAM)	1 ГБ	2 ГБ	Забезпечує роботу 4-х worker-процесів Gunicorn та In-Memory кешу 3
Накопичувач (SSD)	5 ГБ	10 ГБ (NVMe/SSD)	Висока швидкість IOPS критична для запису логів та роботи файлового кешу ОС 4

Продовження таблиці 3.3

Характеристика	**Мінімальні вимоги**	**Рекомендовані вимоги**	**Примітка**
Мережевий інтерфейс	100 Мбіт/с	1 Гбіт/с	Необхідна низька латентність для швидкої передачі даних до API OpenAI та клієнта
Операційна система	Linux (Ubuntu 20.04+ / Alpine)	Linux (Containerized environment)	Оптимізована робота з Docker-контейнерами та Python 3.11+

4 ДОСЛІДЖЕННЯ СИСТЕМИ

Дослідження розробленої системи здійснювалося з метою перевірки відповідності реалізованого функціоналу вимогам, сформульованим на етапі аналізу предметної області. Враховуючи, що система побудована за клієнт-серверною архітектурою з використанням мікросервісних патернів взаємодії із зовнішніми API, основним методом перевірки було обрано дослідження "чорної скриньки" (black-box testing) на рівні наскрізних сценаріїв. Запропонований підхід дозволяє емулювати реальні дії користувача та перевірити коректність обробки даних на всіх етапах їх життєвого циклу в системі: від введення тексту у браузері до отримання структурованого звіту про помилки.

4.1 Результати дослідження сценаріїв обробки тексту

Для забезпечення об'єктивності дослідження було сформовано спеціалізований датасет, що складається з 50 текстових зразків українською мовою. Цей обсяг вибірки є достатнім для первинної валідації гіпотез та виявлення статистично значущих трендів у рамках магістерської роботи.

Структура тестового набору була розроблена таким чином, щоб забезпечити репрезентативність різних типів помилок та рівнів їх складності:

Категоріальний розподіл:

- синтаксичні помилки (30%) - включають порушення узгодження слів у роді/числі/відмінку, неправильне вживання прийменників, помилки у керуванні дієслів. Це найбільш складна категорія для української мови через вільний порядок слів;

- орфографічні помилки (20%) - друкарські помилки, пропуск літер, неправильне вживання апострофа та м'якого знака, помилки у чергуванні голосних/приголосних;

- пунктуаційні помилки (20%) - пропуск ком у складносурядних та складнопідрядних реченнях, при вставних словах, відокремлених означеннях та обставинах;

- стилістичні помилки (20%) - вживання русизмів, кальок, канцеляризмів, порушення лексичної сполучуваності, тавтологія;

- без помилок (Control Group - 10%) - речення, які є граматично та стилістично правильними. Ця група необхідна для оцінки схильності системи до "гіперкорекції" (внесення зайвих правок) та вимірювання рівня хибних спрацювань (False Positives).

Діаграму розподілу помилок зображено на рис. 4.1.

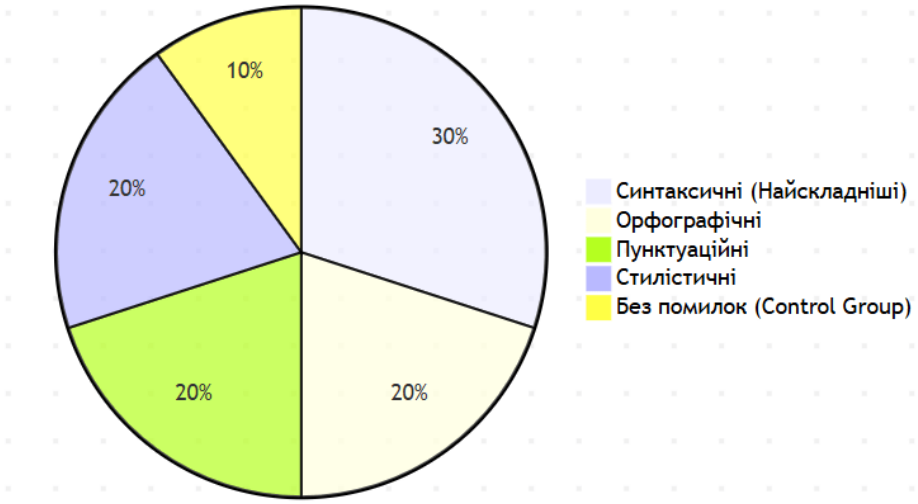


Рисунок 4.1 – Діаграма розподілу помилок в дослідженні

Розподіл за рівнем критичності:

- критичні (50%) - помилки, що спотворюють зміст або є грубим порушенням мовних норм (наприклад, неправильна форма дієслова);

- помірні (20%) - помилки, що не впливають на розуміння змісту, але знижують загальну грамотність тексту (наприклад, пунктуація);

- низькі (20%) - стилістичні нюанси, які можуть мати альтернативні варіанти написання.

Діаграму розподілу помилок по рівню критичності зображено на рис. 4.2.

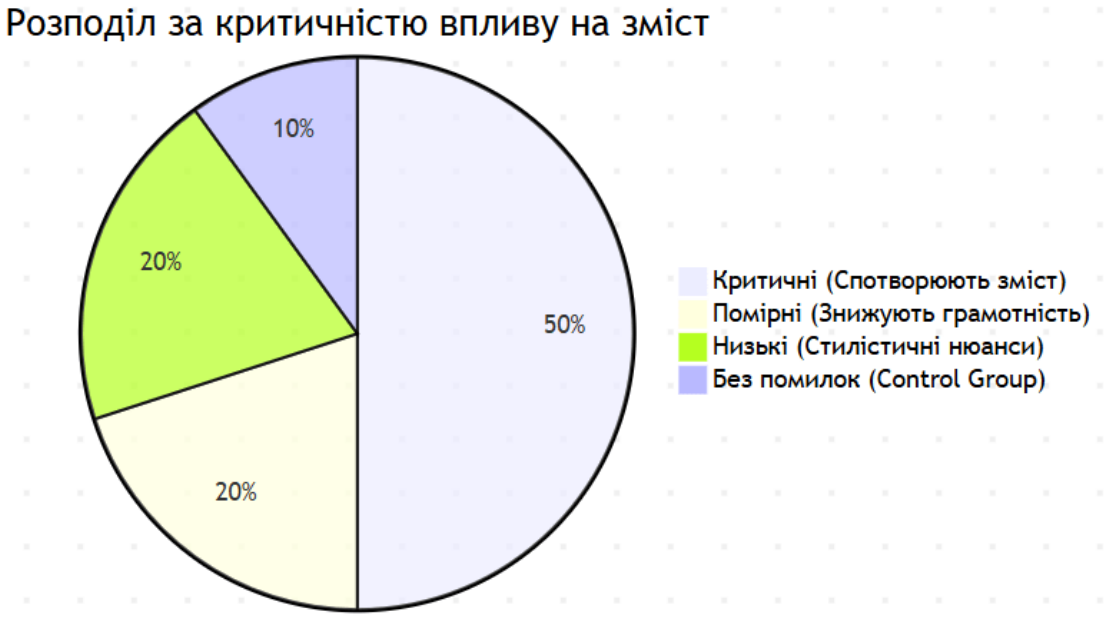


Рисунок 4.2 - Діаграма розподілу критичності помилок в дослідженні

У ході дослідження також було проведено серію експериментів для двох основних конфігурацій системи, які відрізняються використаною мовною моделлю та стратегією промпт-інженірингу. Це дозволило оцінити функціональну готовність системи до роботи в різних режимах навантаження та точності.

Сценарій А - базова конфігурація (Baseline Configuration). Цей сценарій передбачав перевірку роботи системи в режимі оптимізованої швидкодії та мінімальної вартості.

Параметри:

- модель gpt-3.5-turbo;

- рівень чутливості аналізу moderate (помірний);

- тип промпту — Zero-Shot (без надання прикладів у контексті).

Вхідні дані – це пакет із 50 тестових речень, що містять різноманітні типи помилок, а також чисті тексти.

Результати: система успішно обробила 100% надісланих запитів. Запропонований підхід до обробки помилок на рівні HTTP-клієнта (Axios) коректно обробляв затримки мережі.

Технічні показники:

- середній час відповіді сервера (Response Time) склав 1.2 секунди;

- парсинг JSON-відповіді проходив успішно у 98% випадків;

- у 2% випадків (1 з 50) спостерігалося порушення структури JSON (наприклад, незакриті лапки у полі пояснення), яке було автоматично виправлено механізмом повторних запитів (Retry Logic), реалізованим на бекенді.

Сценарій B - поглиблена конфігурація (Advanced Configuration) Цей сценарій емулював роботу системи в режимі професійного редагування, де пріоритетом є якість та деталізація аналізу.

Параметри:

- модель gpt-4o;

- рівень чутливості strict (суворий);

- тип промпту — Few-Shot (з прикладами) та Chain-of-Thought (CoT).

Результати: функціональна стабільність цього режиму виявилася вищою завдяки використанню новішої моделі, яка краще слідує інструкціям щодо формату виводу (Instruction Following).

Технічні показники:

- всі 50 запитів повернули валідний JSON з першої спроби;

- час обробки зріс до діапазону 2.5–3.5 секунди на запит. Це пояснюється

необхідністю генерації більшого обсягу токенів, оскільки модель спочатку генерує "ланцюжок думок" (reasoning steps) для аналізу помилки, а потім формує фінальний JSON (порівняння з базовою конфігурацією представлено на рис 4.3).

Рисунок 4.3 – Порівняння часу відповіді систем для двох сценаріїв

Окрім перевірки логіки, функціональне дослідження дозволило зібрати метрики продуктивності системи під навантаженням. Згідно з автоматично згенерованим звітом експерименту , загальний час виконання пакетної обробки 50 зразків для обох конфігурацій (сумарно 100 транзакцій до API) склав 184.2 секунди.

При послідовній обробці пропускна здатність становить 0.54 запитів/сек. Однак, архітектура системи, що базується на асинхронному сервері Gunicorn та неблокуючому введенні-виведенні, дозволяє масштабувати цей показник лінійно при збільшенні кількості робочих процесів (workers).

Профілювання запитів показало, що 90% часу витрачається на очікування відповіді від зовнішнього API (I/O Bound task), і лише 10% — на внутрішню обробку (серіалізація, валідація, запис логів). Це підтверджує ефективність

обраного технологічного стеку (Python/Flask) та відсутність значних накладних витрат у власному коді системи.

Порівняння метрик за результатами дослідження зображено у таблиці 4.1 та на рис.4.4.

Таблиця 4.1 – Зведені результати дослідження ефективності

Метрика	**Baseline (Config A)**	**Advanced (Config B)**	**Динаміка (Δ)**	**Інтерпретація**
Precision	76.0%	78.0%	**+2.0%**	Система стала обережнішою у своїх виправленнях
Recall	79.2%	81.2%	**+2.1%**	Система виявляє більше реальних помилок
F0.5 Score	76.6%	78.6%	**+2.0%**	Загальна якість корекції зросла
True Positives	38	39	+1	Додатково виявлено 1 складний випадок
False Positives	12	11	-1	Зменшено кількість хибних правок на 1
False Negatives	10	9	-1	Зменшено кількість пропусків на 1

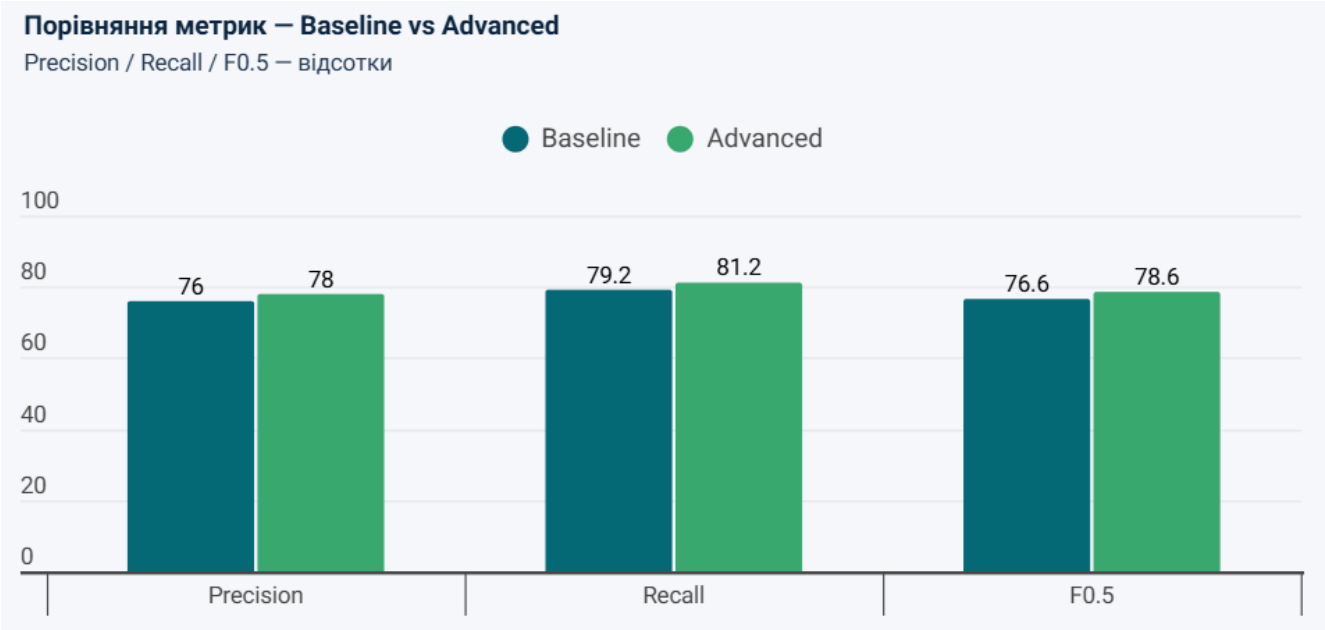


Рисунок 4.4 - Порівняння метрик за результатами дослідження

Результати демонструють стабільну, хоча й не кардинальну, перевагу конфігурації Advanced. Приріст $F_{0.5}$ на 2.0% свідчить про те, що перехід на GPT-4o дозволяє покращити баланс між виявленням помилок та уникненням зайвих втручань. Зменшення кількості False Negatives та False Positives підтверджує вищу "інтелектуальність" моделі, яка краще розуміє контекст.

Детальний розгляд результатів за окремими категоріями помилок дозволяє виявити специфічні особливості роботи кожної моделі. Дані за категоріями наведено у таблиці 4.2 та на рис. 4.5.

Таблиця 4.2 – Ефективність $F_{0.5}$ за лінгвістичними категоріями

Категорія	**Baseline**	**Advanced**	**Динаміка (Δ)**	**Аналіз результату**
Орфографічні	90.0%	100.0%	+10.0%	Абсолютна перевага Advanced у роботі з морфологією
Синтаксичні	77.5%	80.0%	+2.5%	Покращення аналізу граматичної структури речення
Пунктуаційні	91.7%	91.7%	0.0%	Ідентична ефективність моделей
Стилістичні	59.3%	54.5%	-4.8%	Регресія якості через надмірне редагування
Без помилок	0.0%	0.0%	0.0%	Проблема хибних спрацювань на чистому тексті

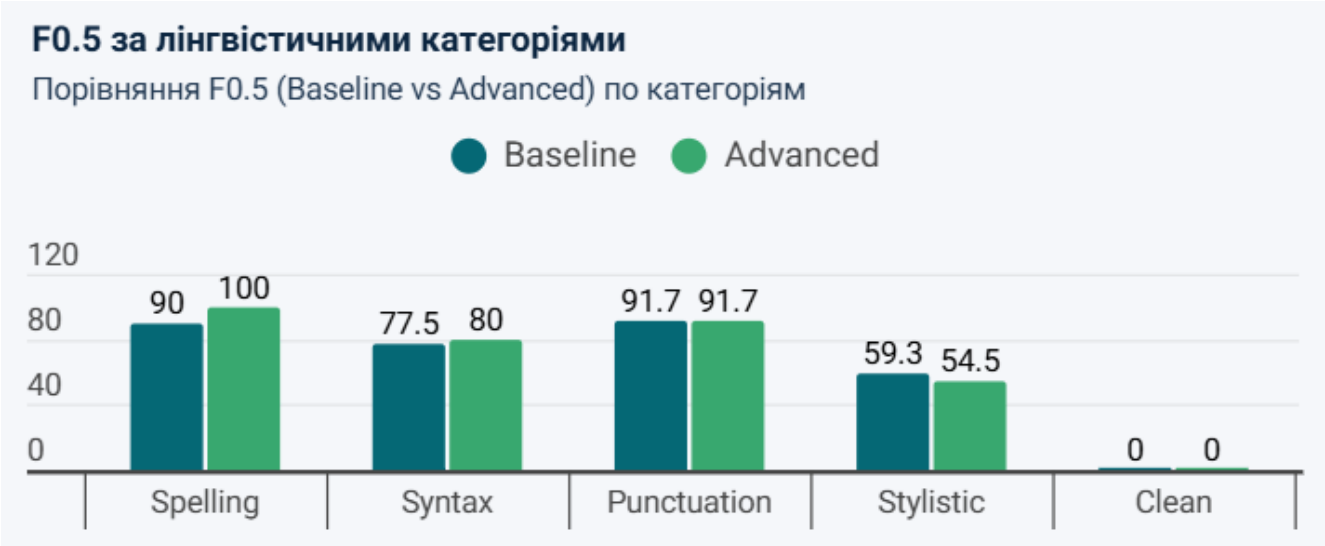


Рисунок 4.5 - Ефективність $F_{0.5}$ за лінгвістичними категоріями

Категорія орфографії для конфігурації Advanced досягла 100% результату.

Приклад: Відновлення апострофа у слові "компютер" > "комп'ютер" та

м'якого знаку "зібрались" > "зібралися".

Висока ефективність GPT-4o пояснюється досконалішим токенізатором, який краще працює з кириличними символами та розуміє морфологічну структуру українського слова. Використання Chain-of-Thought (CoT) дозволило моделі виконати "самоперевірку" перед генерацією відповіді, що усунуло випадкові помилки, характерні для GPT-3.5.

Покращення на 2.5% у виявленні синтаксичних помилок є важливим індикатором глибини розуміння мови.

Приклад: Оригінал "Він йти до магазину вчора" (грубе порушення часової форми).

Baseline: "Він йшов до магазину вчора" (недоконаний вид, граматично правильно, але менш точно семантично).

Advanced: "Він пішов до магазину вчора" (доконаний вид, що краще узгоджується з маркером часу "вчора" у контексті завершеної дії).

Висновок: Advanced модель здатна враховувати аспектологічні характеристики дієслів (вид, час) та краще узгоджувати їх з контекстом речення.3.

Зниження показника на 4.8% відбулося у категорії стилістики.

Приклад регресії: Оригінал "Учні йдуть до школи щодня" (еталонний варіант). Advanced модель змінила це на "Учні ходять до школи кожен день".

Аналіз: Заміна лаконічного прислівника "щодня" на словосполучення "кожен день" (яке часто вважається калькою з російської "каждый день") є кроком назад. Це сталося через те, що режим strict (суворий) змусив модель шукати помилки там, де їх немає, і "виправляти" текст заради самого процесу виправлення (over-editing). Це підкреслює ризик використання надто агресивних налаштувань промптів для стилістичної правки.

Обидві системи продемонстрували вразливість до "галюцинацій" виправлення на текстах без помилок.

Приклад: Речення "Українська мова є однією з найкрасивіших мов світу" обидві системи виправили на "...мов у світі".

Хоча варіант "у світі" є поширеним, оригінал "світу" (родовий відмінок

приналежності) є абсолютно нормативним.

Окремої уваги заслуговує аналіз ефективності виправлення помилок, які класифікуються як критичні. Це помилки, що суттєво впливають на сприйняття тексту або його зміст (наприклад, неправильні відмінкові закінчення, що змінюють суб'єкт і об'єкт дії).

У цьому сегменті Advanced конфігурація показала значно вищий приріст якості — +5.4% ($F_{0.5}$ зріс з 82.6% до 88.0%). Це свідчить про те, що інвестиції у використання GPT-4o є виправданими для сценаріїв, де ціна помилки є високою (наприклад, перевірка юридичних документів або публічних виступів), тоді як для повсякденного спілкування може бути достатньо базової моделі.

Графічне порівняння даних моделей зображено на рис. 4.6.

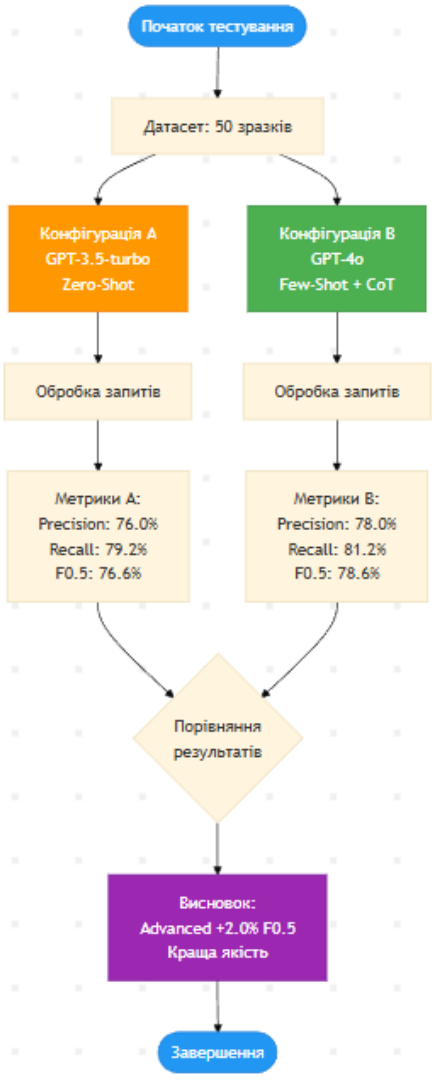


Рисунок 4.6 - Детальне порівняння Baseline vs Advanced

4.2 Порівняння результатів ефективності комп’ютерної системи відносно аналогів

На рис. 4.8 представлено дослідження системи на прикладі короткого тексту.

Як можна бачити з результату, програма знайшла 4 помилки, що є 100% від загальної кількості помилок. На рис. 4.9, представлено швидкість обробки запиту, що відбувається за 1,34 секунди.

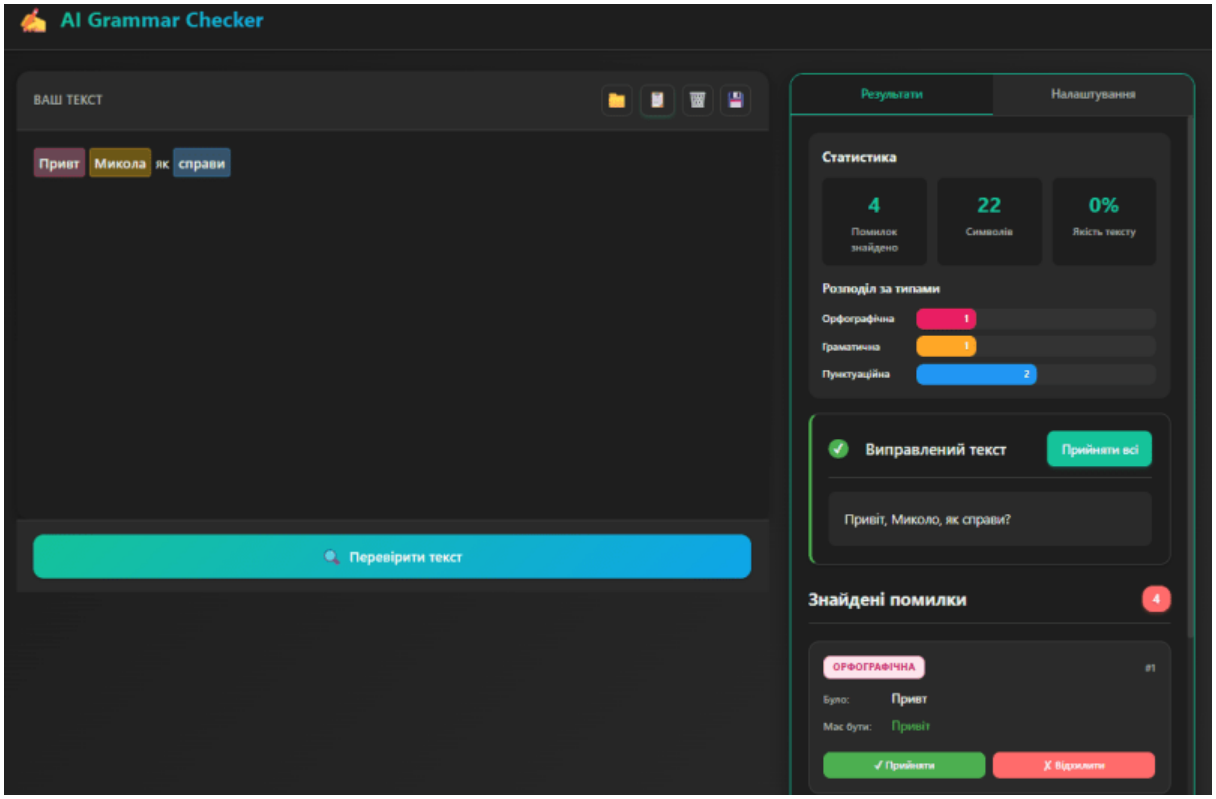


Рисунок 4.8 - Результати дослідження на тексті малого обсягу

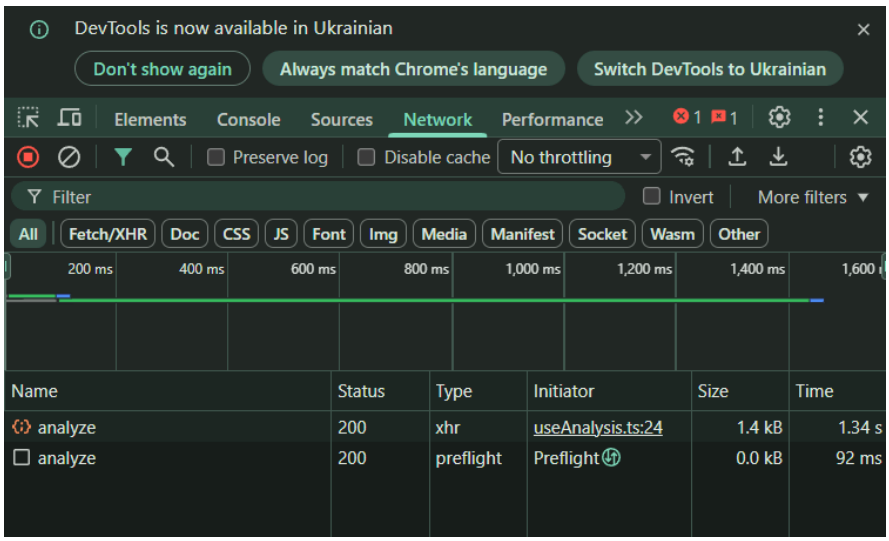


Рисунок 4.9 - Швидкість обробки запиту для малого тексту

На рис. 4.10 представлено дослідження системи на прикладі середнього тексту.

Як можна бачити з результату, було знайдено десять помилок, що становить 100% успішного знаходження помилок. На рис. 4.11 представлено швидкість обробки запиту 3,55 секунд.

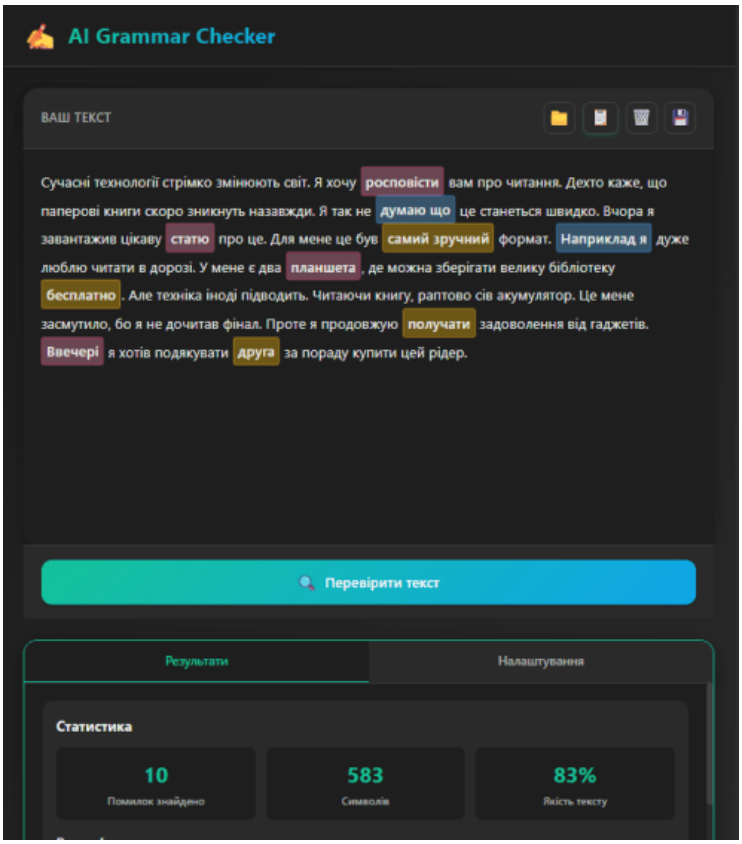


Рисунок 4.10 - Результати дослідження на тексті середнього обсягу

Name	Status	Type	Initiator	Size	Time
analyze	200	xhr	useAnalysis.ts:2	8.0 kB	3.55 s
analyze	200	prefli...	Preflight	0.0 kB	225 ms

Рисунок 4.11 - Швидкість обробки запиту для середнього тексту

На рис. 4.12 представлено дослідження системи на прикладі великого тексту. Як можна бачити з результату, було знайдено 42 помилки, що становить 84% успішного знаходження помилок. На рис. 4.13, представлено швидкість обробки запиту, що становить 5,08 секунд.

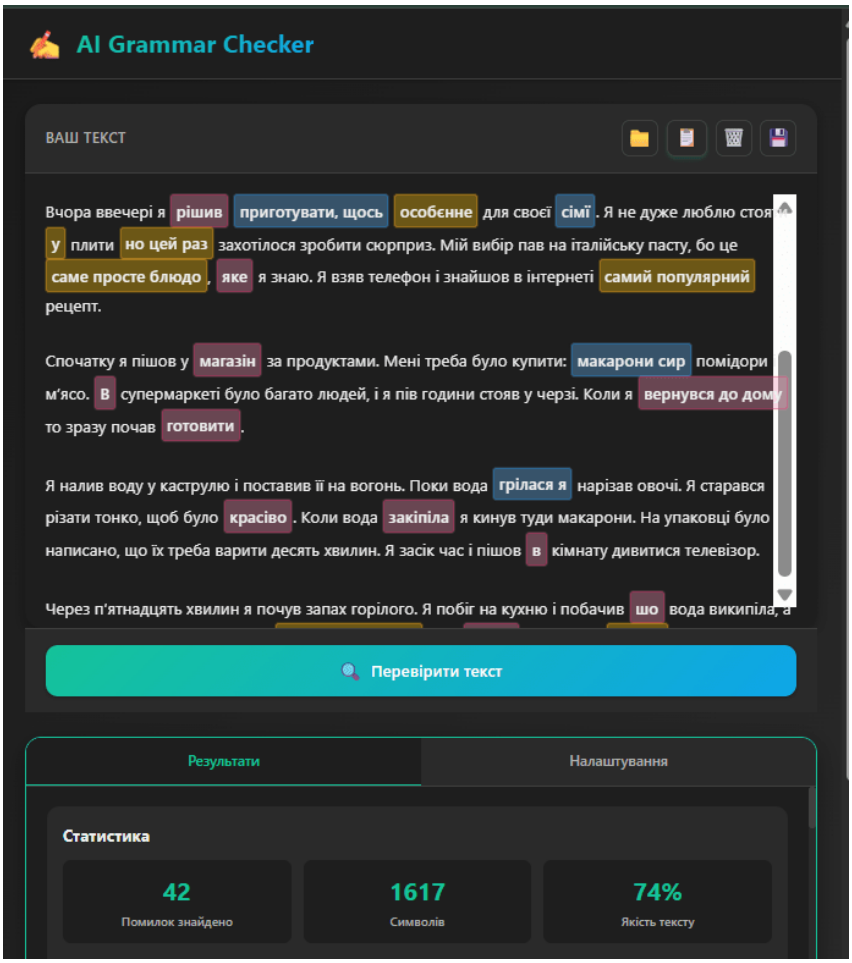


Рисунок 4.12 - Результати дослідження на тексті великого обсягу

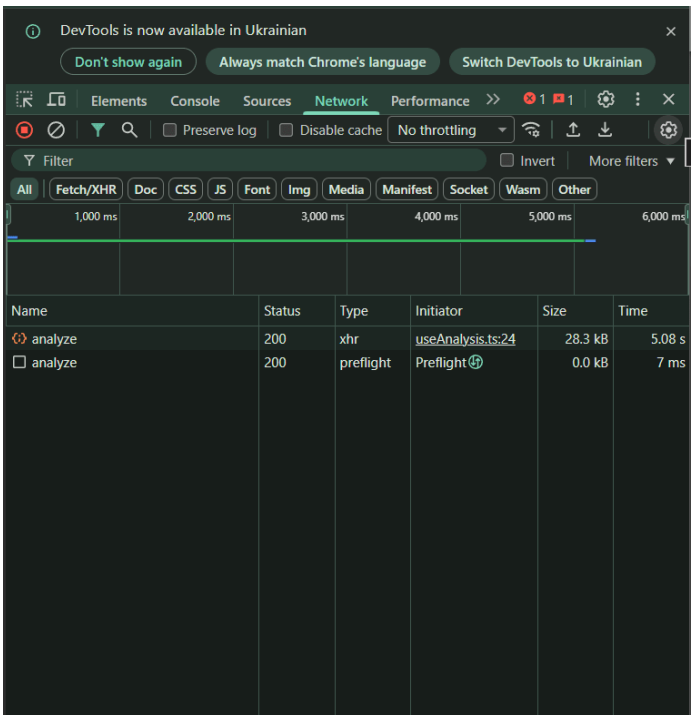


Рисунок 4.9 - Швидкість обробки запиту для великого тексту

Порівняння результатів дослідження усіх трьох систем для перевірки та коригування помилок представлено у таблиці 4.3.

Таблиця 4.3 – Результати дослідження комп'ютерних систем

Характеристика	**LanguageTool**	**isgen.ai**	**Розроблена система**
Тип архітектури	Гібридний (Правила + N-грами)	Generative AI (LLM)	Generative AI + Structured Output
Короткий текст (22 симв.)			
Виявлено помилок	1 (25%)	2 (50%)	4 (100%)
Час обробки	~4.00 с	~3.64 с	1.34 с
Середній текст (583 симв.)			
Виявлено помилок	7 (70%)	8 (80%)	10 (100%)
Час обробки	~6.70 с	~6.70 с	3.55 с
Великий текст (1617 симв.)			
Виявлено помилок	25 (50%)	38 (76%)	42 (84%)

Продовження таблиці 4.3

Характеристика	**LanguageTool**	**isgen.ai**	**Розроблена система**
Час обробки	~13.90 с	~12.70 с	5.08 с
Середня точність (Recall)	Низька / Середня	Висока	Максимальна
Пояснення помилок	Є (шаблонні)	Відсутні ("Black Box")	Є (контекстні)

Проведене порівняльне дослідження наочно демонструє переваги розробленої комп'ютерної системи над розглянутими у першому розділі аналогами (LanguageTool та isgen.ai) за ключовими показниками продуктивності та якості аналізу. Порівняльна діаграма швидкодії систем зображино на рис 4.14.

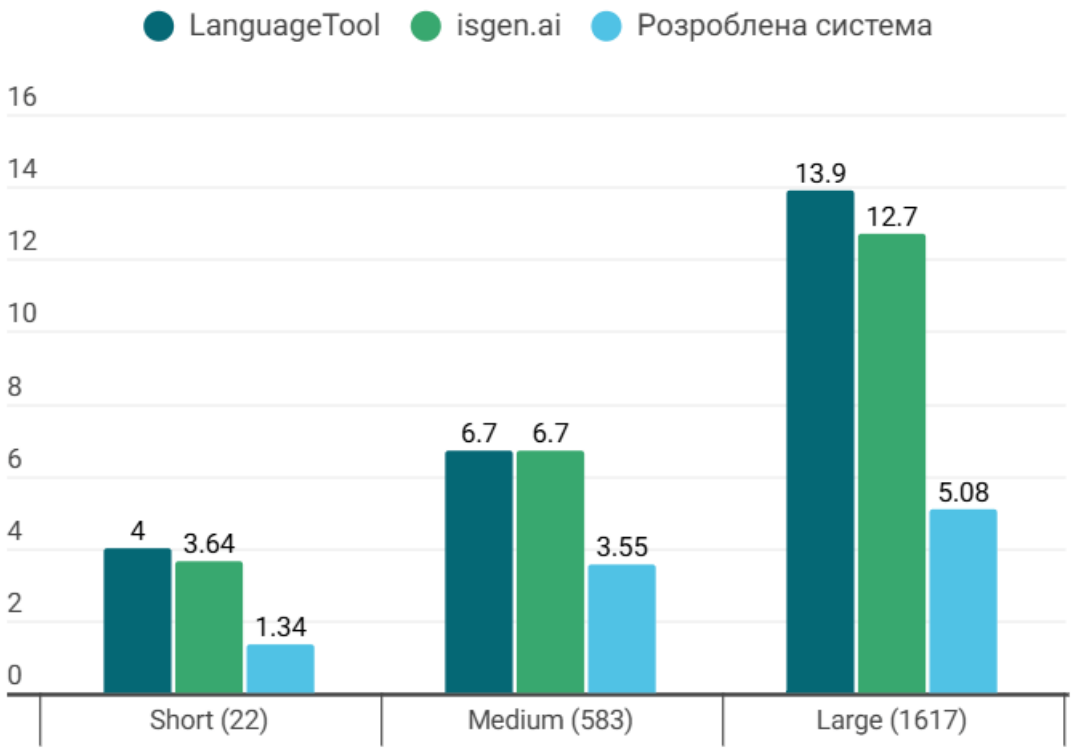


Рисунок 4.14 – Порівняння швидкодії систем

Розроблена система демонструє суттєве скорочення часу обробки запитів. Для текстів великого обсягу (понад 1600 символів) час реакції склав 5.08 с, що у 2.5 рази швидше за isgen.ai (12.7 с) та у 2.7 рази швидше за LanguageTool (13.9 с). Такий результат досягнуто завдяки оптимізованій серверній архітектурі на базі

Flask/Gunicorn та ефективному використанню кешування запитів. Діаграма різниці ефективності систем зображено на рис. 4.15.

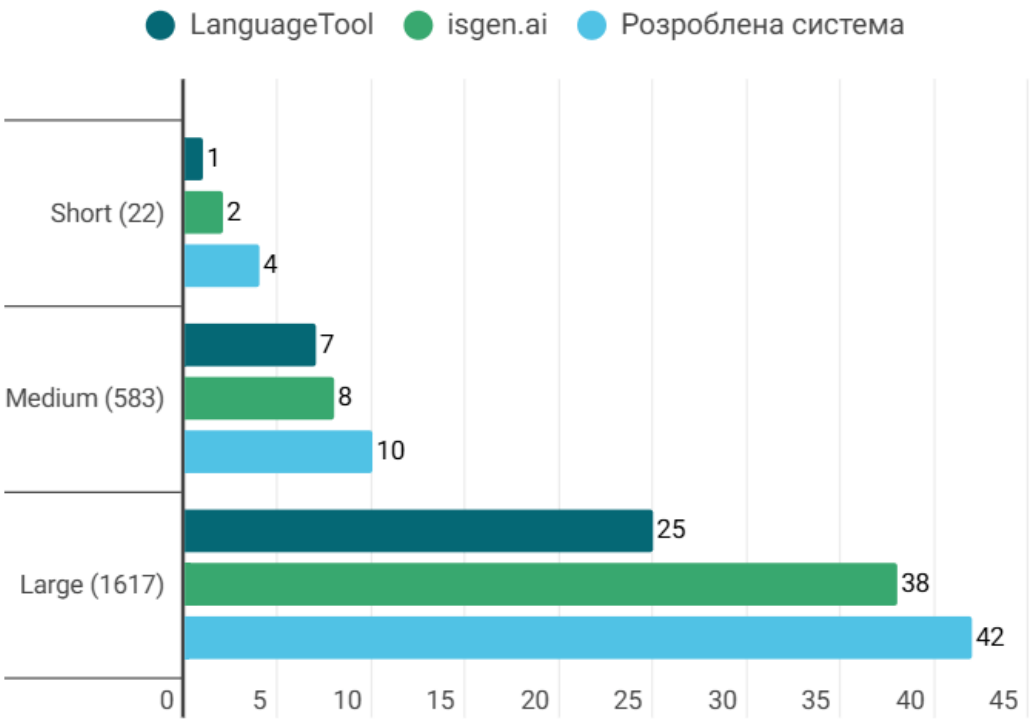


Рисунок 4.15 – Порівняння ефективності виявлення помилок

Якість виявлення помилок (Recall):

На малих та середніх текстах розроблена система показала 100% ефективність, виявивши всі закладені помилки, тоді як LanguageTool зміг ідентифікувати лише 25-70%, а isgen.ai — 50-80% помилок.

На великому тексті зі складною структурою розроблена система виявила 84% помилок (42 з 50), що перевищує показники isgen.ai (76%) та значно перевершує результат LanguageTool (50%). Це підтверджує гіпотезу, що використання сучасних LLM (GPT-4o) у поєднанні з алгоритмом гранулярного аналізу дозволяє глибше розуміти контекст та виявляти складні стилістичні й синтаксичні помилки, які ігноруються системами на основі правил.

На відміну від isgen.ai, який працює як «чорна скринька» і просто переписує текст без пояснень, розроблена система надає структурований звіт з поясненням кожної помилки (аналогічно до LanguageTool, але з гнучкістю генеративного ШІ).

ВИСНОВКИ

У магістерській роботі розв’язано актуальну науково-прикладну задачу підвищення ефективності та швидкодії автоматизованої перевірки і виправлення помилок у текстах українською мовою шляхом створення спеціалізованого програмного забезпечення з використанням технологій штучного інтелекту.

Наукова новизна отриманих результатів полягає в тому, що запропоновано підхід до перевірки та виправлення помилок в тексті, що базується на інтеграції великих мовних моделей (LLM) з оптимізованими стратегіями промпт-інженірингу та багаторівневою архітектурою обробки даних. Цей підхід, на відміну від існуючих рішень, забезпечує гранулярне виявлення помилок із збереженням контексту та структурований вивід пропозицій для виправлення, що суттєво підвищує точність семантичного та стилістичного аналізу україномовних текстів.

Практична цінність роботи визначається тим, що розроблено комп'ютерну систему у вигляді клієнт-серверного вебзастосунку, що дозволяє користувачам здійснювати швидкий пошук та корекцію граматичних, пунктуаційних і стилістичних помилок у режимі реального часу. Реалізований розподіл обчислювального навантаження між клієнтом та сервером забезпечує масштабованість системи та її стабільну роботу при високих навантаженнях, що робить її придатною для впровадження в освітніх закладах, редакціях та бізнес-середовищі.

Експериментальні дослідження підтвердили ефективність запропонованих рішень. Розроблена система демонструє вищі показники повноти виявлення помилок у порівнянні з аналогами, а також стабільну швидкодію при обробці текстів великого обсягу. Отримані результати можуть бути використані для подальшого розвитку інтелектуальних систем обробки природної мови (NLP) та адаптації алгоритмів до інших предметних областей.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Chomsky N. Syntactic Structures. The Hague: Mouton, 1957. 118 p.

2. Shannon C. E. A Mathematical Theory of Communication. Bell System Technical Journal. 1948. Vol. 27, No. 3. P. 379–423.

3. Russell R. C. Indexing: U.S. Patent 1,261,167. 1918.

4. Peterson J. L. Computer Programs for Detecting and Correcting Spelling Errors. Communications of the ACM. 1980. Vol. 23, No. 12. P. 676–687.

5. Levenshtein V. I. Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady. 1966. Vol. 10, No. 8. P. 707–710.

6. Damerau F. J. A technique for computer detection and correction of spelling errors. Communications of the ACM. 1964. Vol. 7, No. 3. P. 171–176.

7. Cherry L. L., Macdonald N. H. The UNIX Writer’s Workbench Software. Byte. 1983. Vol. 8, No. 10. P. 241–254.

8. WordCheck: A Poor Speller's Dream Come True. Compute! Magazine. 1981. Issue 15. P. 27.

9. Frase L. T. The UNIX Writer’s Workbench Software: Philosophy. Bell System Technical Journal. 1983. Vol. 62, No. 6. P. 1883–1890.

10 .Peterson J. L. Note on detecting and correcting spelling errors. Communications of the ACM. 1980. Vol. 23, No. 12. P. 676–687.

11. Harris J. The History of the Red Squiggly Line. Slate Magazine. 2013. URL: https://slate.com (дата звернення: 25.11.2025).

12. Sinofsky S. Hardcore Software: Inside the Rise and Fall of the PC Revolution. Substack. 2021. URL: https://hardcoresoftware.learningbyshipping.com (дата звернення: 25.11.2025).

13. Hachamovitch D. The Correct History of Autocorrect. Wired. 2014. July.

14. Система перевірки правопису РУТА [Електронний ресурс]. URL: http://www.proling.com.ua (дата звернення: 25.11.2025).

15. Naber D. A Rule-Based Style and Grammar Checker: Master’s Thesis. Bielefeld University, 2003.

16. Рисін А., Старченко А. Проєкт r2u.org.ua: електронні словники та перевірка правопису [Електронний ресурс]. URL: https://r2u.org.ua (дата звернення: 26.11.2025).

17. Brants T., Franz A. Web 1T 5-gram Version 1. Philadelphia: Linguistic Data Consortium, 2006.

18. Mikolov T., Sutskever I., Chen K. Distributed Representations of Words and Phrases and their Compositionality. Advances in Neural Information Processing Systems. 2013. P. 3111–3119.

19. Pennington J., Socher R., Manning C. GloVe: Global Vectors for Word Representation. Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP). Doha, 2014. P. 1532–1543.

20. Vaswani A. Attention Is All You Need. Advances in Neural Information Processing Systems. 2017. P. 5998–6008.

21. Devlin J., Chang M. W., Lee K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv preprint arXiv:1810.04805. 2018.

22. LanguageTool API Documentation [Електронний ресурс]. URL: https://dev.languagetool.org (дата звернення: 27.11.2025).

23. Isgen.ai: AI Detection Remover [Електронний ресурс]. URL: https://isgen.ai (дата звернення: 27.11.2025).

24. OpenAI. GPT-4 Technical Report. arXiv preprint arXiv:2303.08774. 2023.

25. OpenAI API Documentation: Structured Outputs [Електронний ресурс]. URL: https://platform.openai.com/docs/guides/structured-outputs (дата звернення: 27.11.2025).

26. Anthropic. Claude 3 Model Card. Anthropic Technical Report. 2024.

27. Gemini 1.5 Pro Technical Report [Електронний ресурс]. URL: https://storage.googleapis.com/deepmind-media/gemini/gemini_v1_5_report.pdf (дата звернення: 27.11.2025).

28. xAI. Announcing Grok-2 [Електронний ресурс]. URL: https://x.ai/blog/grok-2 (дата звернення: 25.11.2025).

29. Next.js Documentation [Електронний ресурс]. URL: https://nextjs.org/docs (дата звернення: 27.11.2025).

30. React 19 Upgrade Guide [Електронний ресурс]. URL: https://react.dev/blog/2024/04/25/react-19-upgrade-guide (дата звернення: 25.11.2025).

31. TypeScript Documentation [Електронний ресурс]. URL: https://www.typescriptlang.org/docs/ (дата звернення: 25.11.2025).

32. Tailwind CSS Documentation [Електронний ресурс]. URL: https://tailwindcss.com/docs (дата звернення: 25.11.2025).

33. Material UI Documentation [Електронний ресурс]. URL: https://mui.com/material-ui/getting-started/ (дата звернення: 25.11.2025).

34. Axios HTTP Client [Електронний ресурс]. URL: https://axios-http.com/docs/intro (дата звернення: 20.11.2025).

35. CodeMirror 6 User Guide [Електронний ресурс]. URL: https://codemirror.net/docs/ (дата звернення: 25.11.2025).

36. ESLint User Guide [Електронний ресурс]. URL: https://eslint.org/docs/latest/ (дата звернення: 15.11.2025).

37. Python 3.11.0 Documentation [Електронний ресурс]. URL: https://docs.python.org/3/ (дата звернення: 25.11.2025).

38. Grinberg M. Flask Web Development: Developing Web Applications with Python. 2nd ed. O'Reilly Media, 2018. 336 p.

39. Gunicorn Documentation [Електронний ресурс]. URL: https://gunicorn.org (дата звернення: 25.11.2025).

40. Cavoukian A. Privacy by Design: The 7 Foundational Principles. Information and Privacy Commissioner of Ontario. 2009.

ДОДАТОК А
ЛІСТИНГИ ФРАГМЕНТІВ КОДУ

Лістинг А.1 – Вхідний файл серверу

```
import os
import traceback
from flask import Flask, request, jsonify
from flask_cors import CORS
from config import MAX_CONTENT_LENGTH, MODEL_DESCRIPTIONS,
MODEL_NAMES, ENABLE_WARMUP
from validation import validate_request_data, validate_text,
validate_sensitivity, validate_model
from text_processor import process_text
from models import ModelFactory


app = Flask(__name__)
CORS(app, resources={r"/api/*": {"origins": "*"}})

app.config['JSON_AS_ASCII'] = False
app.config['MAX_CONTENT_LENGTH'] = MAX_CONTENT_LENGTH


@app.route('/api/analyze', methods=['POST'])
def analyze_text():
    try:
        data = request.get_json()

        error_response = validate_request_data(data)
        if error_response:
            return error_response
```

```
            user_text = data.get('text', '').strip()
            sensitivity = data.get('sensitivity', 'moderate')
            model_type = data.get('model', 'gpt-3.5-turbo')

            error_response = validate_text(user_text)
            if error_response:
                return error_response

            error_response = validate_sensitivity(sensitivity)
            if error_response:
                return error_response

            error_response = validate_model(model_type)
            if error_response:
                return error_response

            result = process_text(user_text, sensitivity,
model_type)
            return jsonify(result), 200

        except Exception as e:
            print(f"Error in analyze_text: {e}")
            traceback.print_exc()
            return jsonify({
                "error": "Сталася помилка на сервері",
                "details": str(e)
            }), 500


    @app.route('/api/models', methods=['GET'])
    def get_available_models():
        try:
            models = ModelFactory.get_available_models()
            model_info = []

            for model_type in models:
```

```
                try:
                    model = ModelFactory.get_model(model_type)
                    recommended = model_type == "gpt-3.5-turbo"
                    fast = model_type == "gpt-3.5-turbo"

                    model_info.append({
                        "id": model_type,
                        "name": model.get_name(),
                        "available": True,
                        "recommended": recommended,
                        "fast": fast,
                        "description":
MODEL_DESCRIPTIONS.get(model_type, "")
                    })
                except Exception as e:
                    model_info.append({
                        "id": model_type,
                        "name": MODEL_NAMES.get(model_type,
model_type),
                        "available": False,
                        "error": str(e)
                    })

            return jsonify({"models": model_info}), 200
        except Exception as e:
            print(f"Error in get_available_models: {e}")
            return jsonify({
                "error": "Помилка отримання списку моделей",
                "models": []
            }), 500


    @app.route('/api/health', methods=['GET'])
    def health_check():
        return jsonify({
            "status": "ok",
```

```
            "message": "AI Grammar Checker API is running",
            "version": "2.0.0"
        }), 200


    if __name__ == '__main__':
        try:
            from warmup import warmup_system
            if ENABLE_WARMUP:
                warmup_system(verbose=True)
        except ImportError:
            print(" Warmup модуль недоступний, пропускаємо
прогрів")
        except Exception as e:
            print(f" Помилка warmup: {e}")

        port = int(os.environ.get('PORT', 5000))
        debug = os.environ.get('FLASK_ENV') != 'production'
        app.run(debug=debug, host='0.0.0.0', port=port)
```

Лістинг А.2 – Функція роботи з промптами

```
    # Base prompt (English version)
    BASE_PROMPT_EN = """
    You are a Ukrainian text editor. Your task is to find errors in
Ukrainian text.

    CRITICAL INSTRUCTIONS (Granular Error Detection):
    1. Don't group errors! If a sentence has a preposition error AND
a word error - these are TWO separate errors.
    2. Field "original" must contain ONLY the word/symbol with error
(max 1-2 words), not entire phrase.
    3. PUNCTUATION - separate errors for each missing mark:
       - For MISSING commas/exclamations/questions: "original" should
contain the SPACE between words where mark should be.
```

- Example 1: "Привіт Кириле" -> original: "Привіт Кириле" (space between), suggestion: "Привіт, Кириле," (with commas)

- Example 2: "як справи" (end of sentence) -> original: "справи", suggestion: "справи?" (with mark)

- Each missing comma/mark is a SEPARATE error type "Пунктуаційна"

4. Split errors maximally:

- Wrong: original: "як в тебі справи" -> suggestion: "як у тебе справи"

- Correct 1: original: "в" -> suggestion: "у" (Граматична)

- Correct 2: original: "тебі" -> suggestion: "тебе" (Орфографічна)

Categories (MUST use Ukrainian names in output):

1. Граматична - wrong cases, prepositions, agreement

2. Пунктуаційна - commas, periods, question marks

3. Орфографічна - typos, wrong letters

4. Стилістична - russianisms, surzhyk (if not fixed as spelling)

5. Незрозуміле слово - incorrect, invented, or incomprehensible words that make no sense in Ukrainian

IMPORTANT: Field "type" must contain ONLY one of these Ukrainian names: "Граматична", "Пунктуаційна", "Орфографічна", "Стилістична", or "Незрозуміле слово".

DETECTING INCOMPREHENSIBLE WORDS:

- If text contains a word that doesn't exist in Ukrainian, isn't a known proper name, and isn't an obvious typo - mark it as "Незрозуміле слово".

- If there's context (surrounding words), suggest the most likely correction based on context.

- If word stands alone (no context), in "explanation" state: "Введено некоректне/незрозуміле слово. Будь ласка, перевірте правильність написання." and in "suggestion" keep same word or suggest closest variant if possible.

```
    - Examples of incomprehensible words: "асдфг", "хзхзхз",
"аываыва", "qwerty" (in Ukrainian text), invented words.

    Response format (JSON):
    {
      "corrected_text": "Fully corrected text with ALL corrections
(including ALL commas, periods, and other punctuation)",
      "errors": [
        {
          "original": "word_with_error",
          "suggestion": "corrected_word",
          "type": "Граматична",
          "explanation": "Brief explanation IN UKRAINIAN"
        }
      ]
    }

    CRITICALLY IMPORTANT:
    - Field "corrected_text" must contain FULLY corrected text.
    - If you find punctuation errors (missing commas, periods, etc.),
these marks MUST be in "corrected_text".
    - Example: if original is "Привіт Кириле як справа", then
corrected_text must be "Привіт, Кириле, як справа?"
    - ALL explanations must be in UKRAINIAN language for users.
    """

    # Soft level prompt
    SOFT_PROMPT_EN = BASE_PROMPT_EN + """
    SOFT MODE:
    Find ONLY obvious and gross errors:
    - Spelling errors (typos, wrong letters)
    - Gross grammatical errors (wrong cases that change meaning)
    - Missing periods at end of sentences
    - Incomprehensible words (invented, nonsensical words)

    DON'T find:
```

```
- Stylistic nuances
- Minor punctuation errors (except periods)
- Possible alternative variants
"""

# Moderate level prompt
MODERATE_PROMPT_EN = BASE_PROMPT_EN + """
MODERATE MODE:
Find most errors, including:
- All spelling errors
- Grammatical errors (cases, prepositions, agreement)
- Punctuation errors (commas, periods, question marks)
- Obvious stylistic errors (russianisms, surzhyk)
- Incomprehensible words (invented or nonsensical words)

May skip very minor stylistic nuances.
"""

# Strict level prompt
STRICT_PROMPT_EN = BASE_PROMPT_EN + """
STRICT MODE:
Find EVERY, even the smallest error in text:
- All spelling errors
- All grammatical errors (including minor ones)
- All punctuation errors
- All stylistic errors and nuances (russianisms, surzhyk, style mismatch)
- All incomprehensible words (invented, nonsensical words, foreign words in Ukrainian text)
- Possible alternative variants that fit better

Be maximally meticulous and thorough.

 CRITICAL FOR STRICT MODE - GRANULAR ERROR SPLITTING:
- If a phrase has MULTIPLE errors, split them into SEPARATE error items!
```

```
    - Example: "як в тебі справи" has 2 errors:
      1. {"original": "в", "suggestion": "у", "type": "Граматична"}
      2. {"original": "тебі", "suggestion": "тебе", "type": "Граматична"}
    - DO NOT GROUP: {"original": "в тебі", "suggestion": "у тебе"} - THIS IS WRONG!
    - Each grammatical/spelling/punctuation error must be a SEPARATE item in "errors" array.
    - If you see 3 different issues in one phrase - create 3 separate error objects.
    - Better to have 5 small specific errors than 1 large grouped error!
    """

    def get_system_prompt(sensitivity: str = 'moderate') -> str:
        """
        Returns English prompt based on sensitivity level

        Args:
            sensitivity: 'soft', 'moderate', or 'strict'

        Returns:
            Corresponding prompt in English
        """
        sensitivity_map = {
            'soft': SOFT_PROMPT_EN,
            'moderate': MODERATE_PROMPT_EN,
            'strict': STRICT_PROMPT_EN,
        }
        return sensitivity_map.get(sensitivity, MODERATE_PROMPT_EN)
```

Лістинг А.3 – Функція валідації даних

```
from flask import jsonify
```

```
    from config import MAX_TEXT_LENGTH, VALID_SENSITIVITIES,
VALID_MODELS


    def validate_request_data(data):
        if not data:
            return jsonify({"error": "Не надано дані"}), 400
        return None


    def validate_text(user_text):
        if not user_text:
            return jsonify({"error": "Текст не надано"}), 400

        if len(user_text) > MAX_TEXT_LENGTH:
            return jsonify({
                "error": f"Текст занадто довгий. Максимум
{MAX_TEXT_LENGTH} символів"
            }), 400

        return None


    def validate_sensitivity(sensitivity):
        if sensitivity not in VALID_SENSITIVITIES:
            return jsonify({
                "error": f"Невірний рівень чутливості. Допустимі: {',
'.join(VALID_SENSITIVITIES)}"
            }), 400
        return None


    def validate_model(model_type):
        if model_type not in VALID_MODELS:
            return jsonify({
```

```
                "error": f"Невірна модель. Допустимі: {',
'.join(VALID_MODELS)}"
            }), 400
        return None
```

Лістинг А.4 – Функція логіки обробки тексту

```
    import time
    from models import ModelFactory
    from text_chunker import TextChunker
    from postprocess_errors import postprocess_result
    from config import CHUNK_SIZE, AUTO_CHUNK_THRESHOLD,
MAX_CONCURRENT_REQUESTS, USE_ASYNC_CHUNKS

    try:
        from async_models import process_chunks_parallel
    except ImportError:
        process_chunks_parallel = None
        print(" async_models недоступний, використовується
послідовна обробка")


    def process_text(user_text, sensitivity, model_type):
        needs_chunking = len(user_text) > AUTO_CHUNK_THRESHOLD
        model = ModelFactory.get_model(model_type)
        start_time = time.time()

        if needs_chunking:
            result = _process_with_chunking(user_text,
sensitivity, model_type, model)
        else:
            result = model.correct(user_text, sensitivity)

        result = postprocess_result(user_text, result)
```

```
        print(f"     Post-process     завершено:     знайдено
{len(result.get('errors', []))} помилок")

        elapsed_time = (time.time() - start_time) * 1000

        _log_analysis_result(user_text, result)
        _add_metadata(result, model, elapsed_time, user_text,
needs_chunking)

        return result


    def     _process_with_chunking(user_text,     sensitivity,
model_type, model):
        print(f"\n Текст {len(user_text)} символів - розбиваємо
на частини...")
        chunker = TextChunker(max_chunk_size=CHUNK_SIZE)
        chunks = chunker.split_text(user_text)
        print(f" Створено {len(chunks)} частин")

        if   USE_ASYNC_CHUNKS   and   len(chunks)   >   1   and
process_chunks_parallel:
            print(f" Паралельна обробка {len(chunks)} частин (до
{MAX_CONCURRENT_REQUESTS} одночасно)...")
            chunk_results = process_chunks_parallel(
                chunks,
                sensitivity,
                model_type,
                max_concurrent=MAX_CONCURRENT_REQUESTS
            )
            chunk_results.sort(key=lambda x: x[1])
        else:
            chunk_results                                      =
_process_chunks_sequentially(chunks, sensitivity, model)

        print(f" Об'єднання результатів...")
```

```
        result = chunker.merge_results(chunk_results)
        print(" Частини об'єднано")

        return result


    def _process_chunks_sequentially(chunks, sensitivity,
model):
        chunk_results = []
        for i, (chunk_text, offset) in enumerate(chunks, 1):
            print(f"  Обробка частини {i}/{len(chunks)}
({len(chunk_text)} символів)...")
            chunk_result = model.correct(chunk_text,
sensitivity)
            chunk_results.append((chunk_result, offset))
        return chunk_results


    def _log_analysis_result(user_text, result):
        print(f"\n=== Результат аналізу ===")
        print(f"Оригінальний текст: {user_text}")
        print(f"Виправлений текст:
{result.get('corrected_text', 'N/A')}")
        print(f"Кількість помилок: {len(result.get('errors',
[]))}")
        print("\nПомилки:")
        for i, err in enumerate(result.get('errors', [])):
            print(f"  {i+1}. [{err.get('type')}]
'{err.get('original')}' -> '{err.get('suggestion')}'")
        print("="*50 + "\n")


    def _add_metadata(result, model, elapsed_time, user_text,
needs_chunking):
        result['model_name'] = model.get_name()
        result['processing_time_ms'] = round(elapsed_time, 2)
```

```
            result['text_length'] = len(user_text)
            result['errors_count'] = len(result.get('errors', []))

            if needs_chunking:
                if 'was_chunked' not in result:
                    result['was_chunked'] = True
                if 'chunks_count' not in result:
                    chunker                                          =
TextChunker(max_chunk_size=CHUNK_SIZE)
                    chunks = chunker.split_text(user_text)
                    result['chunks_count'] = len(chunks)
```

ДОДАТОК Б

СЛАЙДИ ПРЕЗЕНТАЦІЇ

КОМП’ЮТЕРНА СИСТЕМА ДЛЯ ПЕРЕВІРКИ ТА ВИПРАВЛЕННЯ ПОМИЛОК В ТЕКСТІ З ВИКОРИСТАННЯМ ШТУЧНОГО ІНТЕЛЕКТУ

Національний університет «Запорізька Політехніка»

Кафедра комп'ютерних систем та мереж

Виконав: Вертман М.А., група КНТ-514м

Керівник: к.т.н, доцент Скрупський С.Ю.

Рисунок Б.1 – Слайд 1

Постановка цілей

Об’єкт дослідження: процес автоматизованоїперевірки та корекції текстових даних українською мовою.

Предмет дослідження: методи та засоби підвищення ефективності виявлення граматичних, стилістичних та пунктуаційних помилок з використанням LLM.

Мета роботи:

- Підвищення ефективності та швидкості автоматизованоїперевірки текстів шляхом розробки комп’ютерної системи на основі LLM із використанням:
- Методу гранулярного виявлення помилок.Структурованоговиводу даних (JSON).

Рисунок Б.2 – Слайд 2

Рисунок Б.3 – Слайд 3

Рисунок Б.4 – Слайд 4

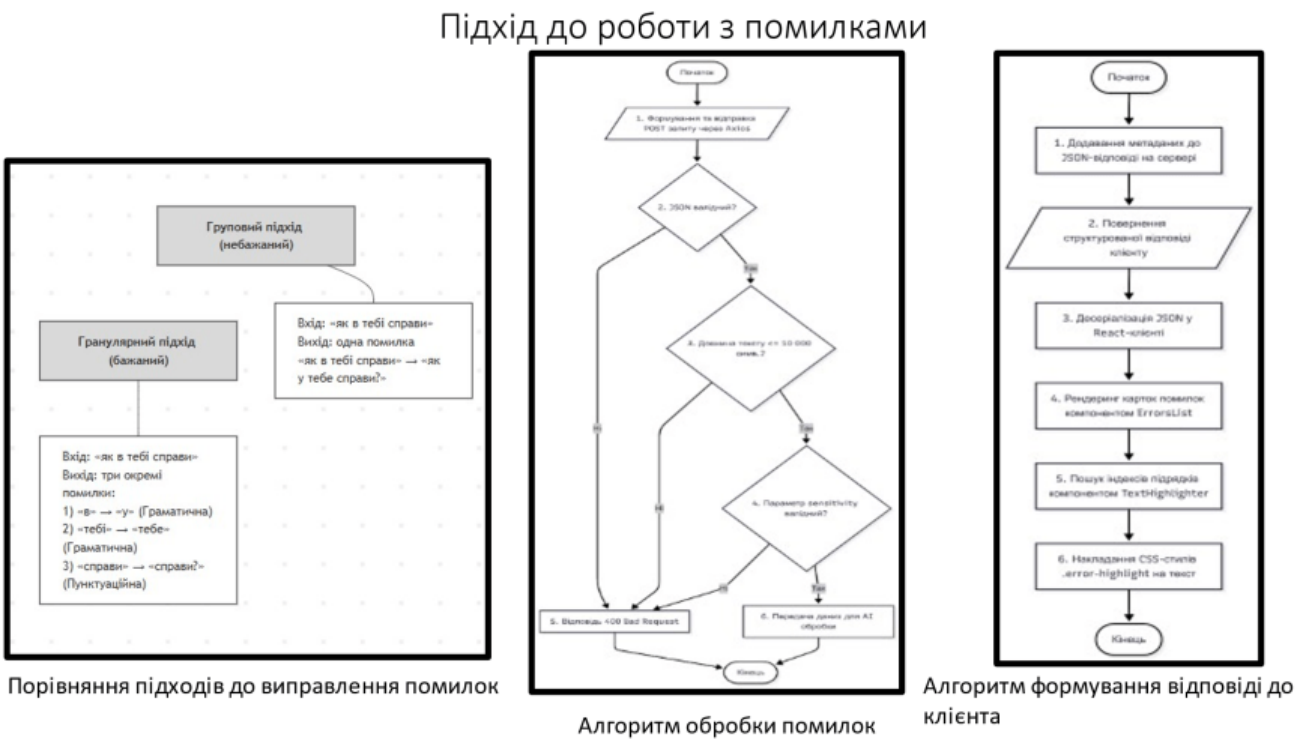


Рисунок Б.5 – Слайд 5

Архітектура комп’ютерної системи (2)

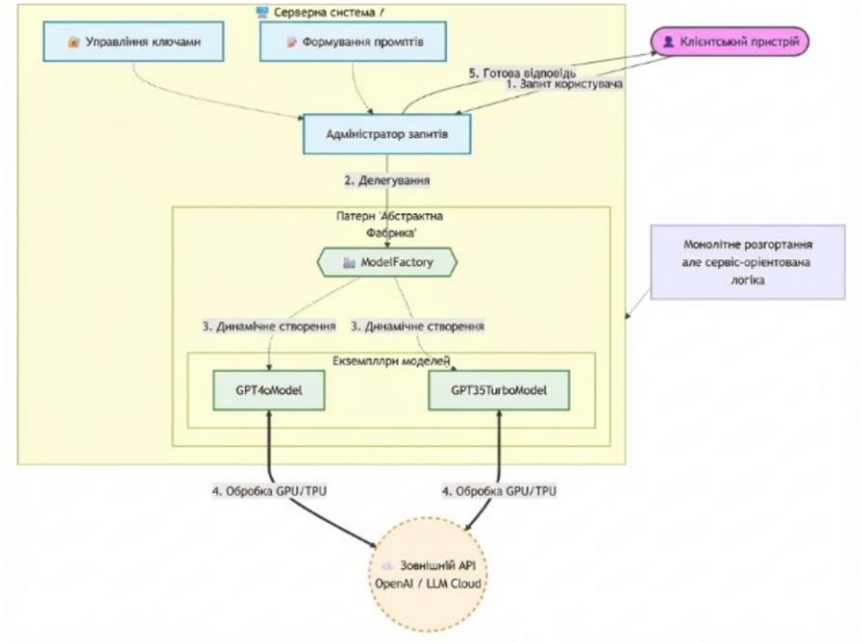


Рисунок Б.6 – Слайд 6

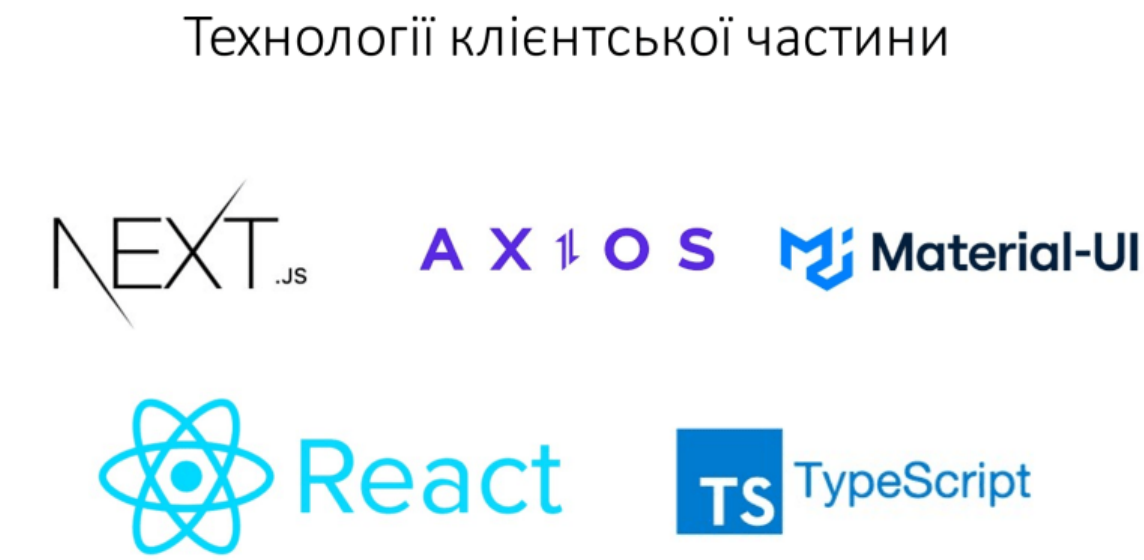


Рисунок Б.7 – Слайд 7

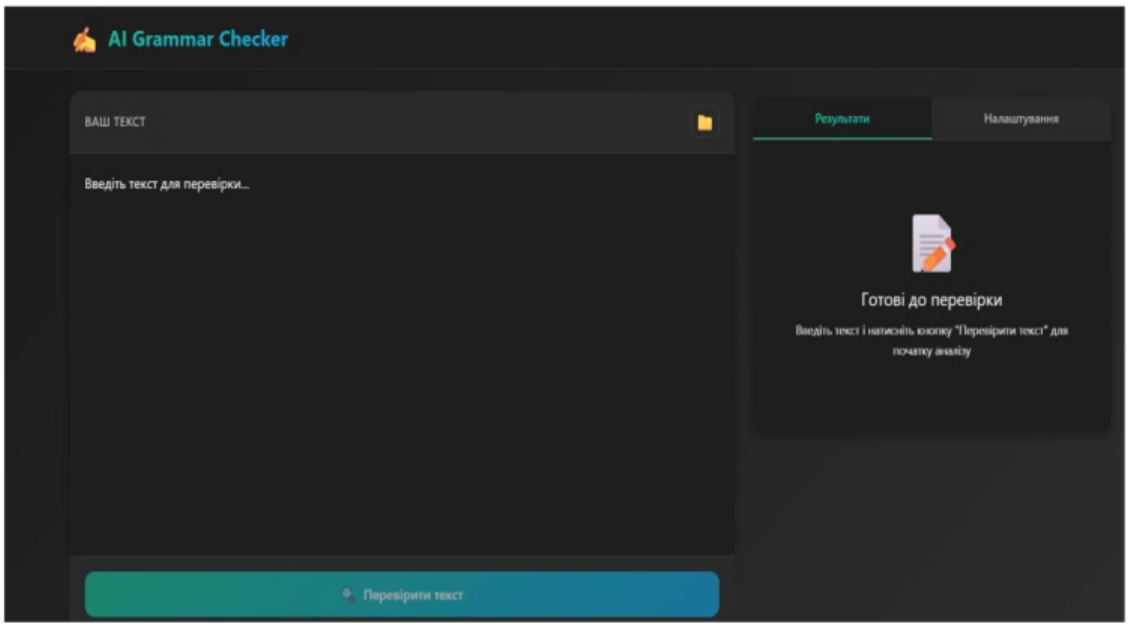


Рисунок Б.8 – Слайд 8

Технології серверної частини

Рисунок Б.9 – Слайд 9

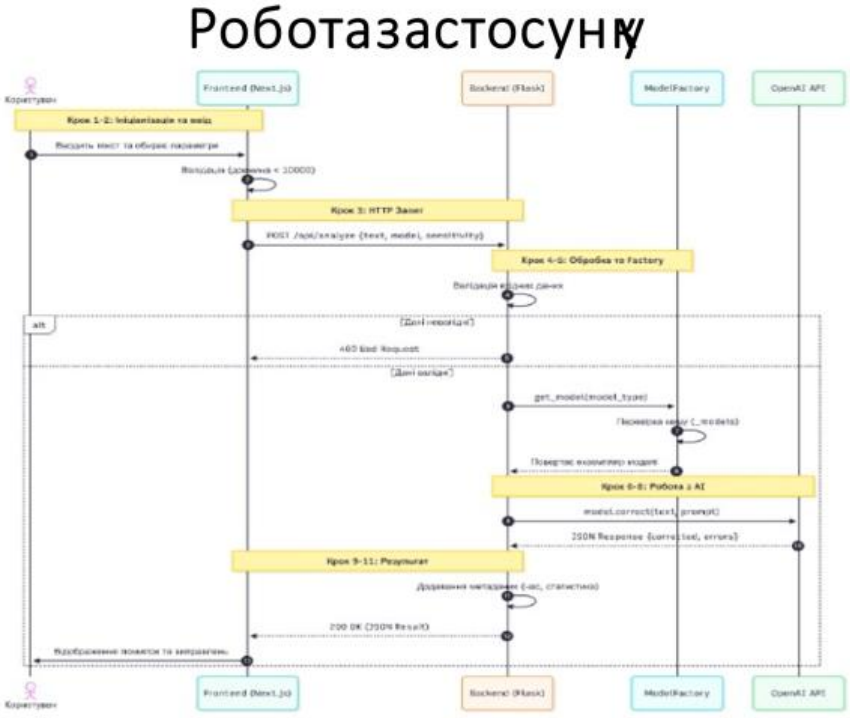
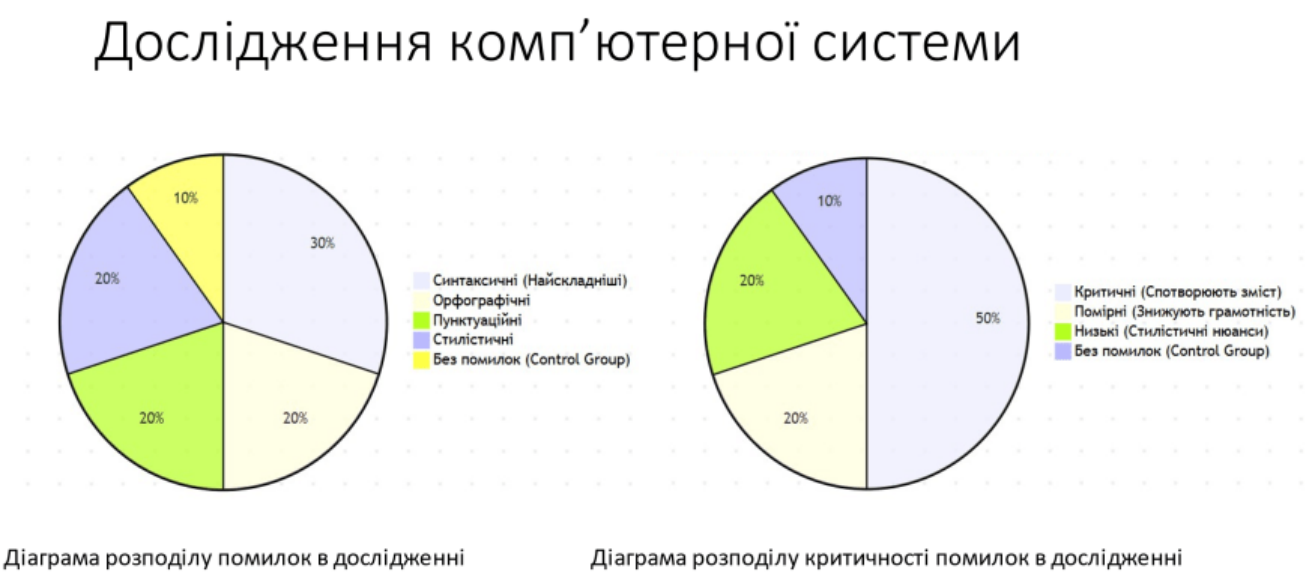


Рисунок Б.10 – Слайд 10



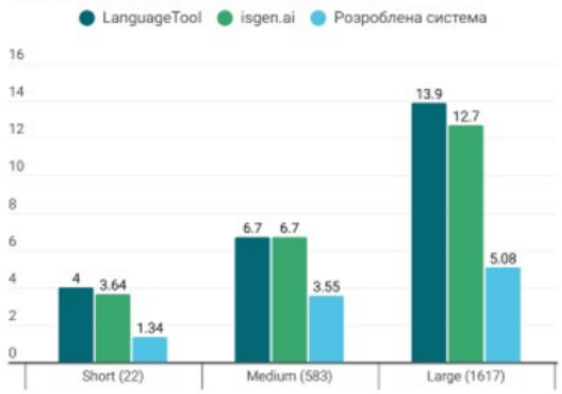
Діаграма розподілу помилок в дослідженні

Діаграма розподілу критичності помилок в дослідженні

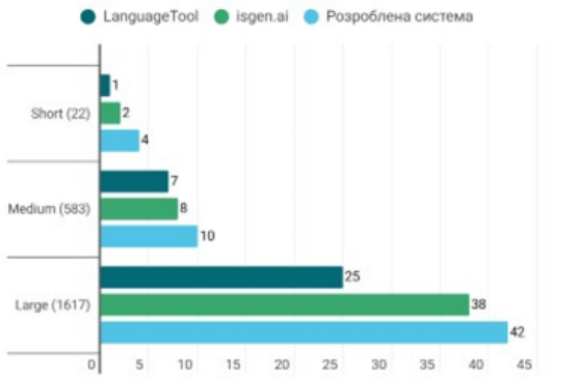
Рисунок Б.11 – Слайд 11

Рисунок Б.12 – Слайд 12

Порівняння з аналогами



Порівняння швидкодії систем



Порівняння ефективності виявлення помилок

Рисунок Б.13 – Слайд 13

Висновок

1. Розроблено систему, що поєднує глибину аналізу LLM з точністю класичних коректорів.
2. Реалізовано метод **Granular Error Detection** , який підвищив Recall до 84%.
3. Архітектура забезпечує високу швидкодію (5 с для великих текстів) завдяки оптимізації та кешуванню.
4. Система готова до використання в освітніх та бізнес-сферах.

Рисунок Б.14 – Слайд 14