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 **КОНСУЛЬТАТИВНА СИСТЕМА ДЛЯ ВЕРИФІКАЦІЇ КОДУ НА VHDL ІЗ ВИКОРИСТАННЯМ ЗАСОБІВ 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

КОТЕЛЕВСЬКИЙ Д.С.
(ПРІЗВИЩЕ та ініціали)
Керівник ЗЕЛЕНЬОВА І.Я.
(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

КОТЕЛЕВСЬКОГО Данила Сергійовича
(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Консультативна система для верифікації коду на VHDL із використанням засобів штучного інтелекту
керівник проєкту (роботи) кан. техн. наук, доцент, ЗЕЛЕНЬОВА Ірина Яківна
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)
затверджені наказом вищого навчального закладу від "_30_" _жовтня_ 2025 року №491
2. Строк подання студентом проєкту (роботи) 23 грудня 2025 року
3. Вихідні дані до проєкту (роботи) мова опису апаратури VHDL, схеми цифрових пристроїв, Altera Quartus, Visual Studio Code, OpenAI API, Antropic Claude API, Gemini API

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1) Аналіз предметної області та вимог до розробки
2) Розробка методики аналізу AI моделей та проведення дослідження
3) Розробка системи
4) Тестування системи
5) Практичні рекомендації щодо використання розробленої системи

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

6. Консультанти розділів проєкту (роботи)

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

7. Дата видачі завдання 01.10.2025 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	03.11.2025 р.	
3	Проведення дослідження	10.11.2025 р.	
4	Розробка системи	15.11.2025 р.	
5	Тестування 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 _____ **Данило КОТЕЛЕВСЬКИЙ**
(підпис) (ім'я, ПРІЗВИЩЕ)

Керівник проєкту (роботи) _____ **Ірина ЗЕЛЕНЬОВА**
(підпис) (ім'я, ПРІЗВИЩЕ)

РЕФЕРАТ

ПЗ: 95 с., 27 рис., 5 ліст., 3 табл., 22 джерела.

VHDL, FPGA, ВЕРИФІКАЦІЯ, ШТУЧНИЙ ІНТЕЛЕКТ, LLM, ПРОМПТ-ІНЖЕНІРИНГ, STATIC ANALYSIS, TESTBENCH

Об’єкт дослідження – процес автоматизованої верифікації коду на мові опису апаратури VHDL.

Предмет дослідження – методи та засоби підвищення ефективності аналізу VHDL-коду з використанням великих мовних моделей.

Мета роботи – підвищення якості та прискорення процесу верифікації FPGA-проєктів шляхом розробки консультативної системи на базі AI.

Методи дослідження – теоретичний аналіз архітектур нейронних мереж, порівняльний експеримент ефективності LLM (Claude, GPT, Gemini), об'єктно-орієнтоване проєктування вебсистем.

Робота містить наукову новизну, яка полягає в розробці методики обрання адекватної моделі LLM для тестування програм на мові опису апаратури.

Результати – проведено порівняльний аналіз 9 сучасних AI моделей, виявлено, що Claude Haiku 4.5 є оптимальною для інтерактивного аналізу. Розроблено вебсистему, що виконує статичний аналіз коду та генерацію тестбенчів.

Галузь використання – навчальний процес підготовки фахівців з комп'ютерної інженерії, автоматизація розробки FPGA.

ABSTRACT

Explanatory note to the master's work: 95 pages, 27 figures, 5 listings, 3 tables, 22 sources.

VHDL, FPGA, VERIFICATION, ARTIFICIAL INTELLIGENCE, LLM, PROMPT ENGINEERING, STATIC ANALYSIS, TESTBENCH

The object of research is the process of automated code verification in the VHDL hardware description language.

The subject of research is methods and means of improving the efficiency of VHDL code analysis using large language models.

The purpose of the work is to improve the quality and speed up the verification process of FPGA projects by developing an AI-based advisory system.

Research methods: theoretical analysis of neural network architectures, comparative experiment on the effectiveness of LLM (Claude, GPT, Gemini), object-oriented design of web systems.

The work contains scientific novelty, which consists in developing a methodology for selecting an adequate LLM model for testing programs in a hardware description language.

Results: a comparative analysis of nine modern AI models was conducted, revealing that Claude Haiku 4.5 is optimal for interactive analysis. A web system was developed that performs static code analysis and testbench generation.

Field of application: training process for computer engineering specialists, automation of FPGA development.

ЗМІСТ

Скорочення та умовні познаки 8
Вступ 9
1 Аналіз предметної області та вимог до розробки 10
1.1 Актуальність верифікації проєктів на FPGA 10
1.2 Основи функціонування великих мовних моделей та промпт-інжиніринг 11
1.3 Класифікація моделей за рівнем продуктивності та архітектурною складністю 12
1.4 Аналіз вимог до консультативної системи 13
1.5 Вибір стеку технологій для розробки 16
1.6 Постановка задачі розробки 17
2 Розробка методики аналізу AI моделей та проведення дослідження 18
2.1 Розробка методики порівняльного аналізу 19
2.2 Обґрунтування вибору моделей для дослідження 20
2.3 Формування тестового набору даних 21
2.4 Процедура та автоматизація проведення експерименту 24
2.4.1 Етап 1: Автоматизований збір даних 24
2.4.2 Проблема аналізу та нормалізація результатів 25
2.4.3 Етап 2: Технічна реалізація нормалізації 27
2.5 Результати експерименту та їх аналіз 29
2.5.1 Загальні показники ефективності моделей 29
2.5.2 Детальний аналіз якості за категоріями помилок 32
2.5.3 Аналіз надійності 36
2.5.4 Аналіз продуктивності та вартості 38
2.6 Підсумки аналізу AI моделей 40
3 Розробка системи 41
3.1 Архітектура системи 41
3.2 Загальна структура програмного забезпечення 43
3.3 Реалізація модулю аналізу коду 45

3.4 Реалізація модулю генерації верифікаційного оточення 49
3.5 Особливості інтеграції розробленої системи з LLM та використання промпт-інжинірингу 52
3.5.1 Використані стратегії промпт-інжинірингу 52
3.5.2 Обґрунтування вибору мови взаємодії з LLM 52
3.5.3 Технічна реалізація архітектури інтеграції 56
4 Тестування системи 58
4.1 Функціональне тестування наскрізного потоку даних 58
4.2 Тестування надійності та валідація даних 60
4.3 Тестування інтерфейсу користувача та адаптивності 62
4.4 Розгортання системи та заходи безпеки 63
4.5 Підсумки тестування 65
5 Практичні рекомендації щодо використання розробленої системи 65
5.1 Вимоги до технічного забезпечення та кваліфікації користувача 66
5.2 Методика проведення верифікації коду 67
5.3 Рекомендації щодо автоматичної генерації тестбенчів 68
5.4 Перспективи розвитку та вдосконалення системи 71
Висновки 73
Перелік джерел посилання 75
Додаток А 77
Додаток Б 87

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

БД	–	База даних
ПЗ	–	Програмне забезпечення
VHDL	–	VHSIC Hardware Description Language, Мова опису апаратури
ПЛІС	–	Програмована логічна інтегральна схема
AI	–	Artificial Intelligence, штучний інтелект
API	–	Aplication Programming Interface, Інтерфейс прикладного програмування
LLM	–	Large Language Models, Великі мовні моделі
SoC	–	System on a Chip, Система на кристалі
SPA	–	Single Page Application, Односторінковий вебзастосунок
UI	–	User Interface, Інтерфейс користувача
UX	–	User Experience, Користувацький досвід
FSM	–	Finite State Machine, Скінченний автомат

ВСТУП

Стрімкий розвиток цифрової електроніки підвищує вимоги до проєктування апаратних систем на базі FPGA. Ручний пошук помилок стає надмірно трудомістким процесом, що вимагає від інженера високої кваліфікації та значних часових ресурсів.

Ситуація ускладнюється паралельною природою VHDL, яка відрізняє її від класичного програмування. Специфічні помилки, такі як гонки сигналів, некоректна робота з тактовими доменами чи створення небажаних засувок (latches), часто залишаються непоміченими на етапі компіляції, але призводять до збоїв обладнання. Це створює високий поріг входження для початківців та знижує ефективність навчального процесу.

Вирішенням цієї проблеми є створення автоматизованих інструментів, здатних не лише виявляти дефекти, а й надавати інтелектуальні пояснення. Найбільш перспективним підходом є використання великих мовних моделей (LLM), інтеграція яких у процес розробки дозволяє суттєво прискорити верифікацію та підвищити якість коду.

Однак, ринок сучасних AI моделей пропонує широкий спектр рішень, що відрізняються точністю та вартістю. Сліпе використання будь-якої моделі без попереднього аналізу може призвести до створення ненадійної системи, схильної до генерації хибних порад.

Метою даної роботи є підвищення ефективності процесу верифікації та навчання VHDL-коду шляхом розробки спеціалізованої консультативної вебсистеми на основі штучного інтелекту.

Для досягнення поставленої мети необхідно вирішити задачу обґрунтованого вибору оптимальної великої мовної моделі. У зв'язку з цим, важливою складовою роботи є розробка методики та проведення порівняльного аналізу ефективності сучасних LLM, результати якого стануть фундаментом для програмної реалізації та архітектури проєктованої системи.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ВИМОГ ДО РОЗРОБКИ

1.1 Актуальність верифікації проєктів на FPGA

Сучасна еволюція програмованих логічних інтегральних схем характеризується переходом до архітектур типу система на кристалі (SoC), що інтегрують мільйони логічних елементів. Таке масштабування апаратних ресурсів створило фундаментальну проблему індустрії - так званий "розрив у верифікації" (verification gap). Суть цього явища полягає у диспропорції: логічна ємність мікросхем зростає значно швидше, ніж продуктивність праці розробників та ефективність традиційних засобів перевірки.

Проблема забезпечення якості коду та відсутності помилок є однією з найгостріших в індустрії. Згідно з дослідженням Wilson Research Group Functional Verification Study, проведеним компанією Siemens EDA у 2022 році, лише 16% усіх FPGA-проєктів змогли досягти стадії виробництва без жодної пропущеної помилки [1]. Це свідчить про те, що традиційні методи ручної перевірки та написання тестбенчів (testbenches) часто виявляються недостатньо ефективними для покриття всіх можливих сценаріїв роботи пристрою. При цьому, статистика вказує, що саме логічні та функціональні помилки залишаються домінуючою причиною некоректної роботи апаратури.

Особливої актуальності ця проблема набуває в контексті використання мови опису апаратури VHDL. Будучи мовою із суворою типізацією та складним синтаксисом, VHDL вимагає від інженера високої уважності до деталей. Для початківців та студентів це створює високий поріг входження: значна частина часу витрачається не на реалізацію алгоритмів, а на боротьбу із синтаксичними конструкціями, типами даних та сигналами, що призводить до неефективного використання ресурсів розробки.

Ціна виправлення помилки зростає експоненціально на кожному наступному етапі життєвого циклу виробу. Помилка, виявлена на етапі написання RTL-коду (Register Transfer Level), коштує мінімальних зусиль для виправлення.

Та ж помилка, пропущена на етап синтезу або, що гірше, на етап імплементації в фізичну схему, потребує значних часових та фінансових витрат на повторну ітерацію проєктування.

Таким чином, існує об'єктивна необхідність у впровадженні нових інструментів автоматизованого аналізу, які здатні виявляти помилки на ранніх стадіях проєктування. Використання технологій штучного інтелекту для статичного аналізу коду та генерації верифікаційного оточення може стати ключовим фактором, що дозволить зменшити кількість помилок, знизити бар'єр входу для нових спеціалістів та скоротити загальний час розробки проєктів на FPGA.

1.2 Основи функціонування великих мовних моделей та промпт-інжиніринг

Сучасний етап розвитку штучного інтелекту характеризується домінуванням великих мовних моделей (Large Language Models, LLM), які демонструють здатність не лише генерувати зв'язний текст, а й вирішувати складні аналітичні та творчі завдання. В основі більшості сучасних LLM лежить архітектура Трансформер (Transformer), вперше представлена дослідниками Google у 2017 році [2]. Ключовою особливістю цієї архітектури є механізм "уваги" (self-attention), який дозволяє моделі аналізувати залежності між усіма елементами вхідних даних одночасно, незалежно від відстані між ними у тексті. Це забезпечує глибоке розуміння контексту, що було недосяжним для попередніх поколінь рекурентних нейронних мереж.

Функціонування LLM базується на імовірнісному принципі: модель не "знає" фактів у людському розумінні, а передбачає наступний токен (фрагмент слова) у послідовності на основі попереднього контексту. Навчання таких моделей відбувається на надвеликих масивах текстових даних, які включають не

лише літературні твори та статті, а й мільйони репозиторіїв програмного коду різними мовами. Завдяки цьому моделі (зокрема серії GPT, Claude, Gemini) засвоїли синтаксичні правила та семантичні патерни мов програмування, включаючи VHDL, що дозволяє використовувати їх як інструмент для статичного аналізу коду (Static Analysis) та рефакторингу.

Однак, ефективність роботи моделі критично залежить від якості вхідних даних - запиту. Це призвело до виникнення нової прикладної дисципліни - промпт-інжинірингу. Промпт-інжиніринг визначається як техніка програмної інженерії, спрямована на розробку та оптимізацію вхідних інструкцій для отримання від моделі результатів, що відповідають заданим критеріям точності та релевантності [3].

У контексті верифікації VHDL-коду, промпт-інженіринг дозволяє нівелювати такі недоліки LLM, як схильність до «галюцинацій» (генерації правдоподібної, але хибної інформації). Застосування таких патернів проєктування запитів, як "Persona Pattern" (надання моделі ролі експерта-верифікатора) та "Few-Shot Prompting" (надання прикладів правильного аналізу), дозволяє значно підвищити точність виявлення логічних та синтаксичних помилок у коді.

1.3 Класифікація моделей за рівнем продуктивності та архітектурною складністю

Ринок сучасних великих мовних моделей характеризується значною гетерогенністю архітектур, які суттєво відрізняються за кількістю параметрів та обчислювальною складністю. У науковій літературі та технічній документації для опису характеристик моделей часто використовують якісні терміни, що потребують чіткої формалізації у контексті даного дослідження. Під терміном "потужність" (capability) у цій роботі розуміється не швидкість генерації токенів, а

здатність моделі до побудови складних ланцюжків логічних міркувань (reasoning) та глибокого розуміння контексту VHDL-коду. Швидкість реакції системи (latency) та вартість інференсу зазвичай знаходяться в оберненій залежності до показників інтелектуальної потужності.

Для систематизації порівняльного аналізу в рамках цієї роботи вводиться класифікація моделей за трьома умовними категоріями продуктивності:

– флагманські моделі ("важкі") - рішення з максимальною кількістю параметрів, пріоритетом яких є найвища якість міркувань ціною великих обчислювальних витрат і затримок;

– збалансовані моделі - оптимізовані версії, що пропонують компромісне співвідношення між точністю аналізу, швидкістю роботи та вартістю використання;

– високоефективні моделі ("легкі") - рішення із мінімальною кількістю параметрів або квантовані версії більш "важких" моделей, розроблені для забезпечення миттєвої генерації відповідей із мінімальними фінансовими витратами.

Такий поділ дозволяє оцінити доцільність використання кожного класу моделей для специфічних задач верифікації, де вимоги до глибини аналізу можуть варіюватися. Зокрема, виявлення неочевидних логічних помилок у VHDL-коді вимагає застосування "важких" моделей, здатних утримувати в контексті складні причинно-наслідкові зв'язки апаратної логіки. Натомість задачі генерації шаблонного коду або перевірки синтаксису можуть бути ефективно вирішені "легкими" моделями без втрати якості.

1.4 Аналіз вимог до консультативної системи

Головною проблемою, яку має вирішувати проєкт, є відсутність доступного, інтерактивного та навчально-орієнтованого інструменту для аналізу якості VHDL-

коду. Існуючі професійні засоби (симулятори ModelSim, Vivado тощо) орієнтовані на досвідчених фахівців, мають складний інтерфейс і надають лише формальні повідомлення про помилки, які часто є незрозумілими для початківців.

Виходячи з цього, головною задачею розробки є створення вебсистеми, яка виконує роль інтелектуального асистента. Система повинна не лише вказувати на місце помилки, а й пояснювати її природу та пропонувати шляхи виправлення, сприяючи глибшому розумінню принципів проєктування апаратури.

Система позиціюється насамперед як асистент для навчання аналізу VHDL-коду. Її другорядна роль - бути зручним інструментом для швидкої верифікації коду досвідченими розробниками.

Функціональні вимоги до системи:

– введення коду - система повинна надавати текстове поле з підтримкою підсвітки синтаксису VHDL, куди користувач може вставити або написати код;

– ініціалізація аналізу - після введення коду користувачу стає доступною кнопка "Аналізувати", яка запускає процес перевірки;

– візуалізація результатів - за результатами аналізу система повинна динамічно підсвічувати рядки або блоки коду, що містять потенційні проблеми;

– деталізація проблеми - при взаємодії (клік або наведення) з підсвіченою ділянкою користувачу має показуватись додаткова інформація про помилку.

У свою чергу деталізація проблеми має містити:

– тип помилки (наприклад "синтаксична", "логічна", "стилістична");

– ступінь критичності помилки;

– пояснення суті проблеми;

– рекомендоване рішення.

На основі цих вимог було визначено два основні функціональні модулі системи:

– аналіз та оптимізація коду - відповідає за інтерактивний пошук помилок, надання інтелектуальних пояснень та пропозицій з рефакторингу;

– генерація тестового оточення - відповідає за автоматичний парсинг інтерфейсу VHDL-коду та створення повноцінних тестбенчів на основі текстових

запитів користувача.

Узагальнена схема, що ілюструє взаємодію цих модулів в єдиному інтерфейсі користувача, наведена на рисунку 1.1.

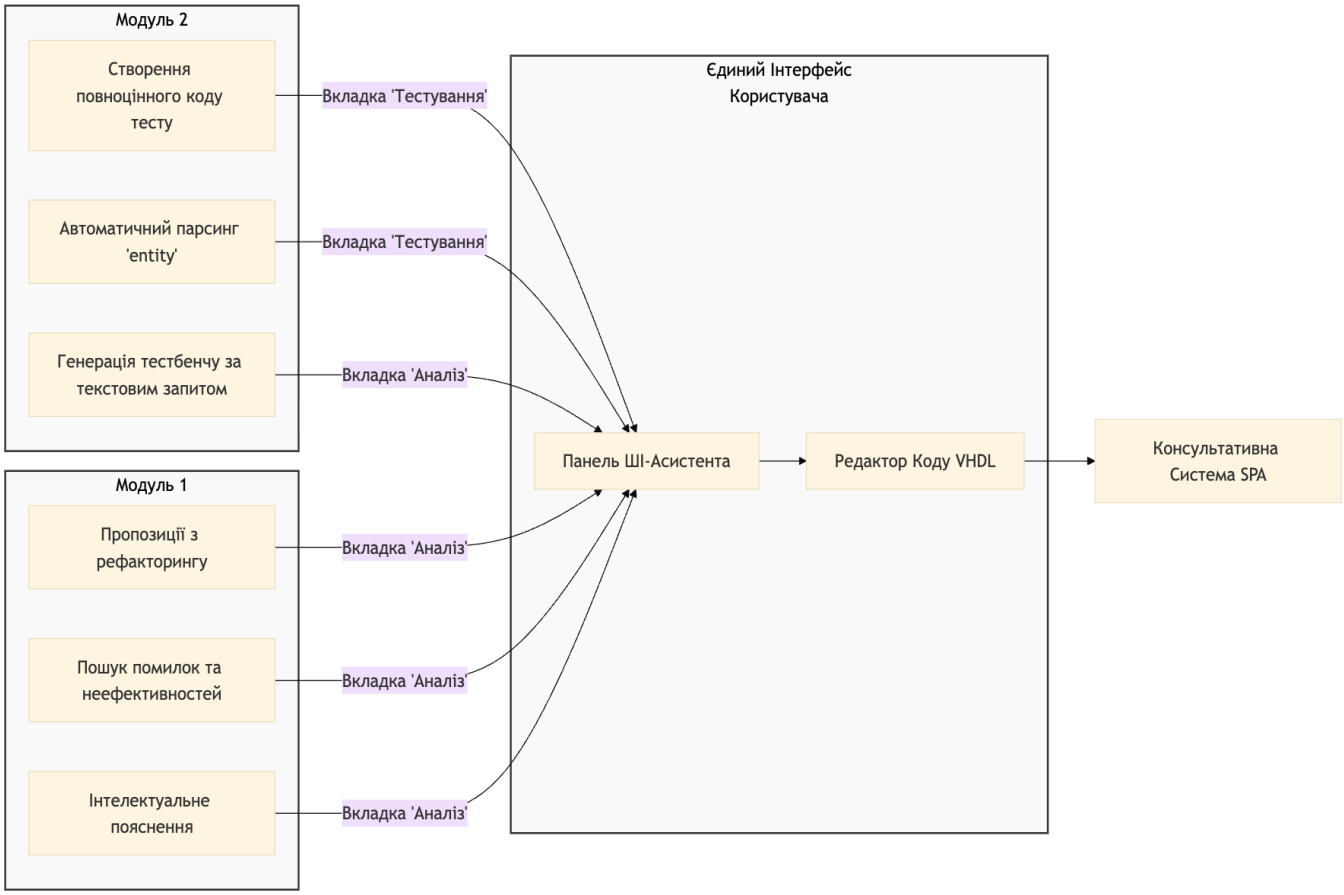


Рисунок 1.1 – Функціональні модулі системи

Окрім функціональних, до системи висувається ряд нефункціональних вимог:

– швидкодія - час від моменту натискання кнопки "Аналізувати" до отримання результату не повинен викликати дискомфорту у користувача;

– інтуїтивність - система має бути легкою для розуміння її використання а також реагувати на дії користувача;

– надійність - результати аналізу мають бути стабільними, точними та відтворюваними;

– масштабованість - архітектура системи повинна бути розроблена таким чином, щоб у майбутньому можна було легко додати нові AI моделі для аналізу

або замінити існуючу без зміни основної логіки застосунку;

– доступність - реалізація у вигляді вебзастосунку (SPA) для забезпечення доступу з будь-якого пристрою без необхідності встановлення локального ПЗ.

1.5 Вибір стеку технологій для розробки

Вибір стека технологій для розробки є стратегічним рішенням, яке безпосередньо впливає на швидкість, якість та подальшу підтримку проєкту. У цьому випадку вибір ґрунтувався на двох ключових принципах: повне покриття функціональних та нефункціональних вимог системи, а також мінімізація ризиків розробки шляхом використання перевірених та добре знайомих інструментів.

В основі клієнтської частини системи лежить бібліотека React. Її компонентна архітектура ідеально підходить для створення динамічних та інтерактивних користувацьких інтерфейсів, що є ключовою вимогою для даного проєкту. Для надання додаткової структури, оптимізації продуктивності та спрощення маршрутизації було обрано фреймворк Next.js, що є надбудовою над React. Ця комбінація дозволяє поєднати гнучкість React з потужними можливостями сучасного фреймворку [4].

Для забезпечення надійності та масштабованості коду було прийнято рішення використовувати TypeScript [5]. Його система статичної типізації дозволяє виявляти значну частину помилок ще на етапі написання коду, що значно підвищує його якість та спрощує довгострокову підтримку.

Щоб прискорити розробку користувацького інтерфейсу, було обрано бібліотеку компонентів Material UI (MUI) [6]. Вона надає широкий набір готових, стилізованих та доступних UI-елементів, що дозволяє зосередитись на реалізації основної бізнес-логіки системи. Для управління станом застосунку було вирішено використати бібліотеку Zustand. Вона є простою та ефективною альтернативою складнішим рішенням і дозволяє керувати даними в додатку з мінімальною

кількістю шаблонного коду [7].

Таким чином, обраний стек (Next.js, TypeScript, MUI, Zustand) є сучасною, потужною та гнучкою комбінацією. Найголовніше, наявність практичного досвіду роботи з усіма цими інструментами гарантує високу якість та своєчасне виконання проєкту, повністю покриваючи всі поставлені вимоги.

1.6 Постановка задачі розробки

Проведений аналіз предметної області засвідчив, що верифікація FPGA-проєктів є критично важливим, але часомістким етапом розробки. Існуючі інструменти часто мають високий поріг входження, тоді як сучасні великі мовні моделі (LLM) демонструють значний потенціал у ролі інтелектуальних асистентів. Однак, наразі відсутні чіткі дані щодо того, яка саме модель є найбільш ефективною та економічно доцільною для специфічної задачі аналізу VHDL-коду.

З огляду на визначену проблему та вимоги, цю роботу присвячено підвищенню ефективності процесу верифікації VHDL-коду через розробку спеціалізованої консультативної вебсистеми на основі штучного інтелекту.

Досягнення зазначеного результату потребує вирішення таких завдань:

– розробити комплексну методику порівняльного аналізу ефективності сучасних LLM у задачах виявлення синтаксичних, логічних та стилістичних помилок у VHDL-коді;

– провести експериментальне дослідження обраних AI моделей (зокрема сімейств GPT, Claude, Gemini), проаналізувати отримані результати за критеріями точності, швидкодії та вартості, та визначити оптимальну модель для інтеграції;

– спроєктувати та програмно реалізувати клієнт-серверну архітектуру консультативної системи, що забезпечує взаємодію користувача з обраною AI моделлю;

– реалізувати функціональні модулі системи: модуль статичного аналізу коду з поясненнями помилок та модуль автоматичної генерації верифікаційного оточення (testbenches);

– провести тестування розробленої системи та надати практичні рекомендації щодо її використання в навчальному та виробничому процесах.

2 РОЗРОБКА МЕТОДИКИ АНАЛІЗУ AI МОДЕЛЕЙ ТА ПРОВЕДЕННЯ ДОСЛІДЖЕННЯ

Головним завданням дослідження є проведення порівняльного аналізу ефективності різних AI-моделей у виявленні помилок та проблем продуктивності в коді на мові VHDL для визначення оптимального кандидата для інтеграції в консультативну систему.

Для успішного виконання поставленого завдання необхідно виконати наступні кроки:

– сформувати репрезентативний набір тестових даних, що покриває різні типи помилок VHDL;

– розробити автоматизовану методику тестування моделей у рівних умовах;

– провести кількісну оцінку точності (Accuracy, Recall, F1-Score) та надійності моделей;

– оцінити експлуатаційні характеристики моделей (швидкість, вартість).

Об'єктом дослідження є процес автоматизованого аналізу коду на мові VHDL за допомогою засобів штучного інтелекту.

Предметом дослідження є показники точності, швидкості обробки, вартості та надійності, які демонструють сучасні великі мовні моделі (LLM) при вирішенні задач верифікації апаратної логіки.

2.1 Розробка методики порівняльного аналізу

Для виконання поставленого завдання та об'єктивної оцінки предмета дослідження - показників точності, швидкості, вартості та надійності - необхідна чітка та відтворювана наукова методика. Вона повинна забезпечити рівні, стандартизовані умови для тестування всіх моделей, що є критично важливим для отримання неупереджених та порівнянних результатів. Запропонована методика порівняльного аналізу складається з п'яти послідовних етапів, графічна схема яких зображена на рисунку 2.1.

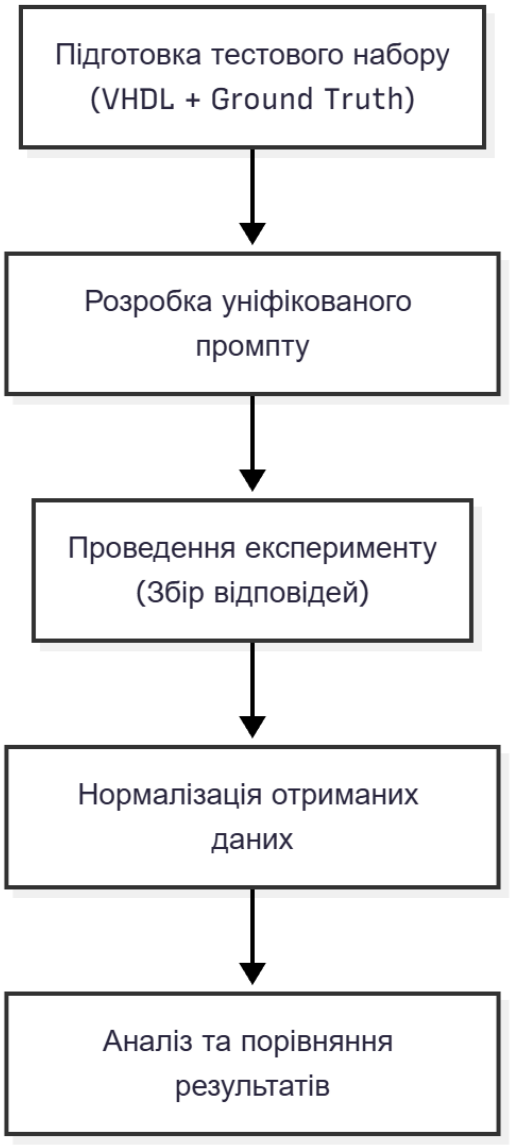


Рисунок 2.1 – Блок-схема методики порівняльного аналізу

Алгоритм методики включає наступні кроки:

– підготовка тестового набору - формування колекції VHDL-файлів для верифікації та створення для кожного з них "ground truth" файлу з еталонним описом помилок, які мають бути виявлені;

– розробка уніфікованого промпту - створення комплексного запиту (промпту) з чіткими інструкціями для AI моделей щодо аналізу коду, пошуку помилок та форматування вихідної відповіді;

– проведення експерименту - послідовна подача кожного тестового VHDL-файлу та уніфікованого промпту на вхід кожній AI моделі з обраного набору (збір та фіксація отриманих відповідей);

– нормалізація даних - приведення отриманих відповідей від різних моделей до єдиного, стандартизованого формату для подальшої обробки;

– аналіз результатів - аналіз нормалізованих даних, порівняння знайдених моделями помилок з "ground truth" файлами та розрахунок показників ефективності.

Такий підхід дозволяє перетворити суб'єктивний аналіз текстових відповідей чат-ботів на сувору кількісну оцінку, що відповідає вимогам до магістерського дослідження.

2.2 Обґрунтування вибору моделей для дослідження

Для забезпечення всебічного та об'єктивного порівняльного аналізу було обрано 9 ключових моделей від трьох провідних розробників (OpenAI, Anthropic та Google). Вибір ґрунтується на необхідності охопити різні "сімейства" та "вагові категорії" моделей, що дозволяє оцінити компроміси між продуктивністю, швидкістю та вартістю.

Список досліджуваних моделей включає:

Сімейство GPT (OpenAI) [9]: GPT-5; GPT-5 mini; GPT-5 nano.

Сімейство Claude (Anthropic) [10, 11]: Claude Opus 4.1; Claude Sonnet 4.5; Claude Haiku 4.5.

Сімейство Gemini (Google) [12, 13]: Gemini 2.5 Pro; Gemini 2.5 Flash; Gemini 2.5 Flash Lite.

Цей набір дозволяє порівняти три основні категорії:

– флагманські моделі (GPT-5, Claude Opus, Gemini Pro), що орієнтовані на максимальну точність аналізу;

– збалансовані моделі (GPT-5 mini, Claude Sonnet, Gemini Flash), що пропонують компроміс між швидкістю та якістю;

– швидкі та легкі моделі (GPT-5 nano, Claude Haiku, Gemini Flash Lite), для яких пріоритетом є миттєва відповідь та низька вартість обробки запиту.

2.3 Формування тестового набору даних

Основою для об'єктивного дослідження є якісний та ретельно структурований набір тестових даних. Для всебічної оцінки можливостей AI моделей буде сформовано спеціалізований датасет, що складається з 25 тестових випадків. Такий підхід дозволить перевірити здатність моделей виявляти широкий спектр проблем різної складності та критичності.

Кожен тестовий випадок у наборі складатиметься з двох файлів:

– VHDL-файл (case_N.vhd) - містить код із заздалегідь закладеною помилкою або є еталонно правильним;

– файл "ground truth" (case_N_gt.json) - файл у форматі JSON, що описує очікуваний результат аналізу, він слугуватиме еталоном для порівняння з відповіддю, згенерованою AI моделлю.

Для систематизації тестування всі помилки, що вносяться у файли, поділяються на категорії за їхнім типом. Основні категорії наведені в таблиці 2.1.

Для об'єктивної оцінки впливу кожної проблеми на кінцевий проєкт, усі

помилки в тестовому наборі також класифікуються за ступенем їх критичності.

Таблиця 2.1 – Категорії помилок VHDL-коду

Категорія помилки	**Опис**	**Приклад**
Синтаксичні	Порушення граматичних правил мови VHDL	Пропущена крапка з комою, помилка в ключовому слові
Стилістичні	Порушення стандартів кодування, що ускладнюють читабельність та підтримку	Непослідовне іменування сигналів, відсутність коментарів
Логічні	Код синтаксично правильний, але його поведінка не відповідає очікуваній	Неповний список чутливості (`sensitivity list`), умови гонок (`race conditions`)
Проблеми синтезу	Конструкції, що призводять до неоптимальної реалізації або не можуть бути синтезовані	Створення небажаних засувок (`latches`), використання `wait for`.
Змішані	Комбінація з 2-3 помилок різних типів в одному файлі	Логічна помилка та стилістичні проблеми.

Класифікація за ступенем критичності, що базується на впливі дефекту на компіляцію та коректність роботи апаратури, наведена в таблиці 2.2.

Таблиця 2.2 – Класифікація рівнів критичності

Рівень критичності	**Опис**
Критичні (Critical)	Помилки, які унеможливлюють компіляцію коду або призводять до створення функціонально непрацездатної апаратної логіки.
Помірні (Moderate)	Проблеми, що не зупиняють компіляцію, але призводять до небажаних наслідків: неефективних апаратних структур (напр., latches), погіршення продуктивності або прихованих логічних дефектів.
Низька (Low)	Проблеми, що не впливають на функціональність або продуктивність, але погіршують якість коду (напр., стилістичні зауваження).

Тестовий набір з 25 файлів буде розподілено наступним чином:

– три файли із синтаксичними помилками (різної складності);

– шість файлів із логічними помилками (середньої та високої складності);

– два файли із проблемами ефективності та синтезу;

– два файли зі стилістичними проблемами;

– сім файлів зі змішаними проблемами;

– п’ять файлів будуть повністю коректними.

Для візуалізації складу тестового набору даних, його структуру наведено на рисунку 2.2.

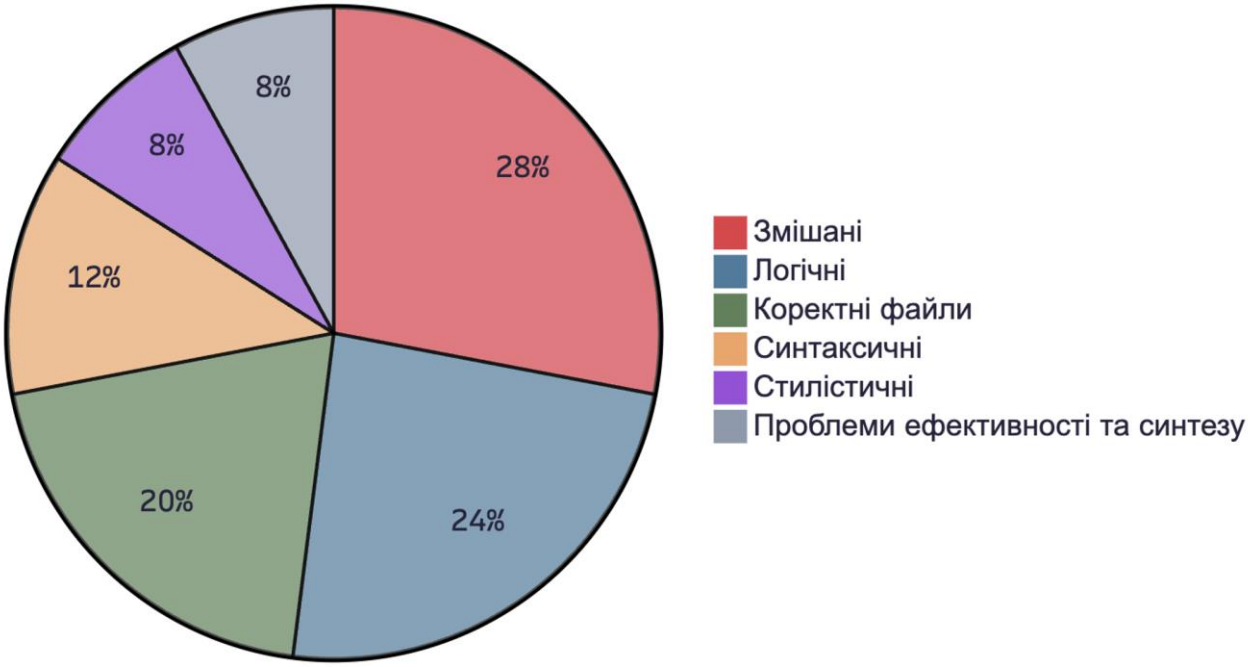


Рисунок 2.2 – Структура тестового набору даних

Важливим буде зазначити, що п’ять повністю коректних файлів ("чистих") потрібні для перевірки моделей на хибні спрацювання (false positives) - ситуації, коли модель повідомляє про неіснуючу помилку.

Для кожного файлу в наборі буде створено відповідний "_gt.json" файл, що містить опис очікуваного результату. Для "чистих" файлів "ground truth" буде порожнім масивом.

2.4 Процедура та автоматизація проведення експерименту

Для забезпечення точності, об'єктивності та відтворюваності результатів, процес дослідження був повністю автоматизований за допомогою спеціалізованого репозиторію на базі Node.js та TypeScript. Весь експеримент поділено на два послідовні етапи (скрипти): збір сирих даних та їх подальша нормалізація.

2.4.1 Етап 1: Автоматизований збір даних

На першому етапі запускається скрипт, що збирає відповіді від усіх дев’яти досліджуваних моделей.

Підготовка середовища:

– тестовий набір - VHDL-файли та файли з "ground truth" (.json), що описують очікувані проблеми, розміщуються у папці test-files;

– промпт - формується єдиний головний промпт для постановки задачі аналізу VHDL-файлу (використання однакового промпту для всіх моделей забезпечує рівні умови експерименту);

– конфігурація - створюється конфігураційний файл, що перелічує всі 9 AI моделей та їх додаткові параметри.

Для забезпечення відтворюваності та зменшення варіативності, для кожної моделі жорстко задаються параметри "temperature" або "seed". Також записуються ціни на використання кожної з моделей (кількість американських доларів за 1 мільйон токенів).

Алгоритм роботи скрипту (рис. 2.3):

– скрипт зчитує всі тестові файли з папки test-files;

– запускається ітераційний процес - кожен VHDL-файл послідовно надається на опрацювання кожній з дев’яти AI моделей;

– відповіді від усіх моделей, що стосуються одного тестового файлу, агрегуються та зберігаються у відповідний json файл у папці "results/responses/".

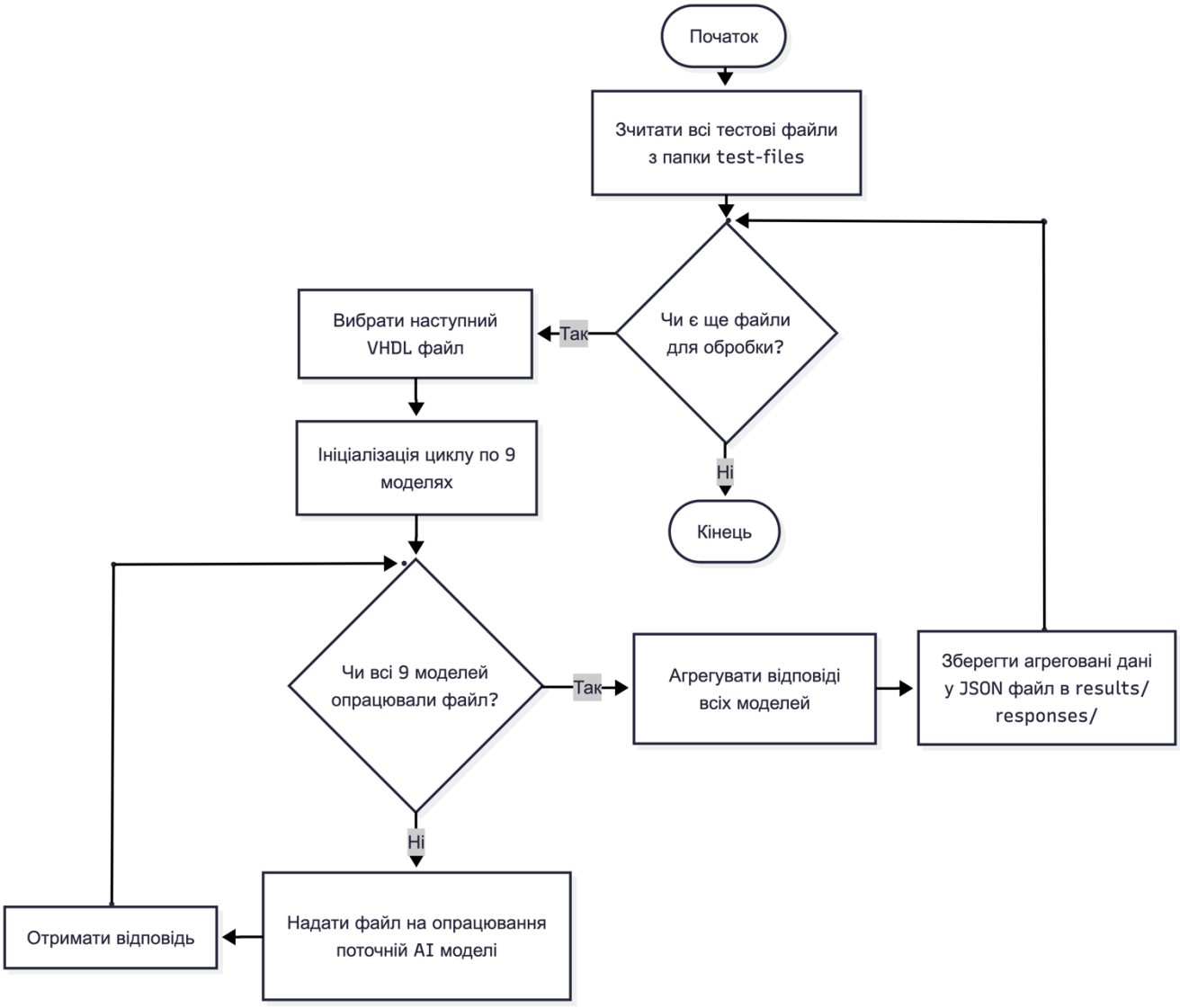


Рисунок 2.3 – Алгоритм збору даних

Таким чином, програмна реалізація алгоритму дозволила повністю автоматизувати процес збору даних. Результатом роботи цього скрипту є набір json-відповідей усіх моделей.

2.4.2 Проблема аналізу та нормалізація результатів

Після завершення першого етапу ми отримуємо масив первинних даних. Однак пряме автоматичне порівняння цих даних із заздалегідь підготовленими еталонними файлами "ground truth" є неможливим. Різні AI моделі, навіть якщо вони коректно ідентифікують одну й ту саму проблему, можуть описувати її різними словами, з різним рівнем деталізації та у різному форматі. Спроба порівняти такі відповіді за допомогою простого зіставлення тексту призведе до

невірних висновків.

Для вирішення цієї проблеми вводиться етап нормалізації даних. Його задача - привести всі отримані відповіді до єдиного, стандартизованого формату, що повністю відповідає структурі "ground truth" файлів. Враховуючи складність семантичного аналізу тексту, цей етап також буде автоматизовано, але з використанням іншого підходу.

Використання AI моделі для аналізу та нормалізації відповідей є обґрунтованим підходом, який базується на концепції семантичної стандартизації за допомогою LLM. Ідея використання великих мовних моделей для приведення неструктурованих або різнорідних даних до єдиного стандарту є активною сферою досліджень. Наукові роботи в цій галузі демонструють, як LLM можуть ефективно вирішувати проблему неузгодженості даних, зіставляючи їх із контрольованим словником чи форматом. Аналогічно до цього, у даному випадку "модель-арбітр" буде стандартизувати відповіді для забезпечення об'єктивного аналізу [5].

Ключова перевага такого підходу полягає у здатності моделі до семантичного зіставлення, а не простого синтаксичного порівняння. Вона розуміє зміст написаного і може визначити, що функціонально ідентичні помилки, описані різними словами, є одним і тим же, що неможливо при простому порівнянні тексту. Крім того, цей метод забезпечує високу масштабованість та ефективність, оскільки ручний аналіз сотень відповідей є надзвичайно трудомістким, а автоматизований підхід дозволяє обробити весь набір даних за значно коротший час, роблячи дослідження відтворюваним.

Надійність методу додатково підкріплюється тим, що завдання "моделі-арбітра" є вузькоспеціалізованим. Їй не потрібно шукати помилку з нуля, а лише порівняти вже наданий опис проблеми з еталоном та, у разі збігу семантики, виконати структурне перетворення у формат JSON. Ймовірність правильного виконання такої чітко обмеженої задачі у передових моделей є надзвичайно високою. Таким чином, залучення "моделі-арбітра" для семантичної стандартизації дозволяє перетворити складну задачу суб'єктивного аналізу тексту

на стандартизований, об'єктивний та науково обґрунтований процес.

2.4.3 Етап 2: Технічна реалізація нормалізації

На другому етапі запускається скрипт, який використовує "модель-арбітра" для семантичного аналізу та приведення всіх даних до єдиного формату csv.

У якості "моделі-арбітра" було обрано Claude Sonnet 4.5. Цей вибір обґрунтований тим, що завдання семантичного зіставлення вимагає потужної моделі з глибоким розумінням контексту, а Sonnet 4.5 пропонує оптимальний баланс між високою інтелектуальною спроможністю та надійністю у генерації чіткого JSON-виводу, що є критично важливим для автоматизації.

Алгоритм роботи скрипту:

– завантажуються очікувані проблеми з "test-files/*.json";

– зчитуються всі відповіді моделей з "results/responses/*.json";

– для кожного тестового файлу генерується спеціальний промпт для "моделі-арбітра", цей промпт містить перелік фактичних проблем (ground truth) та перелік повідомлених моделлю проблем;

– "модель-арбітр" аналізує обидва списки та класифікує відповідь моделі, що тестується, за трьома категоріями: збіги (True Positives), хибні спрацювання (False Positives) та пропуски (False Negatives);

– модель повертає результат у структурованому форматі JSON;

– скрипт парсить JSON-відповідь від арбітра та розраховує фінальні метрики для кожної моделі по кожному тестовому файлу (справжні/хибні позитивні/негативні результати, точність, повнота, F1-score, використання токенів, вартість запиту, час обробки);

– усі розраховані метрики записуються у фінальний файл "results/normalyzed/arbiter_evaluation.csv".

Узагальнена схема, що ілюструє алгоритм перетворення необроблених текстових відповідей на стандартизовані дані за допомогою "моделі-арбітра", наведена на рисунку 2.4.

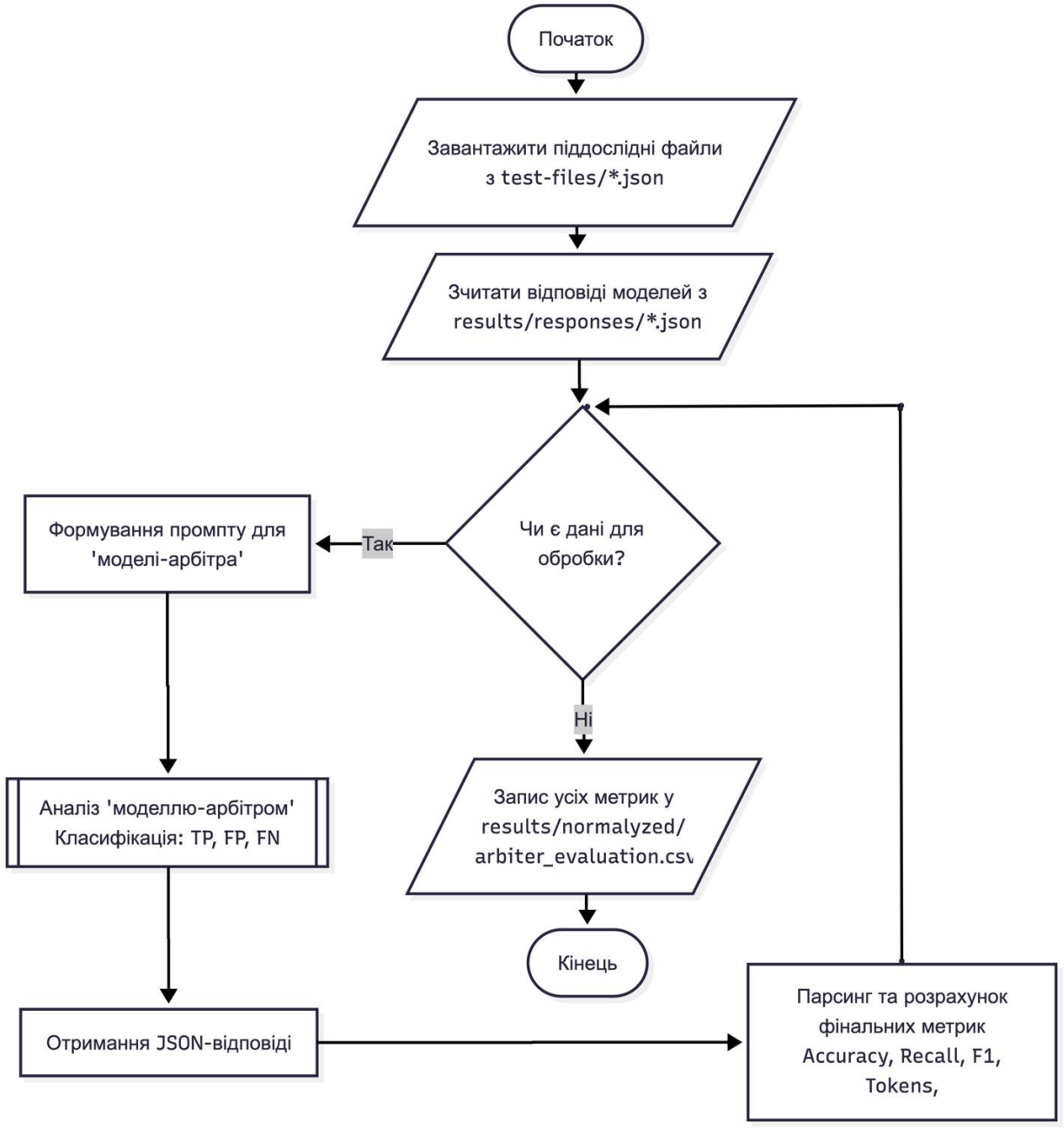


Рисунок 2.4 –Алгоритм процесу нормалізації за допомогою "моделі-арбітра"

Після завершення етапу нормалізації ми отримуємо стандартизований набір даних у форматі csv, який вже можна об'єктивно та автоматично аналізувати для розрахунку фінальних метрик точності та надійності кожної моделі.

2.5 Результати експерименту та їх аналіз

2.5.1 Загальні показники ефективності моделей

Після проведення серії експериментів згідно з розробленою методикою, отримані результати були агреговані для порівняльного аналізу. Ключовим інтегральним показником, що використовується для ранжування моделей, є F1-Score. Він забезпечує гармонійний баланс між точністю (Precision) та повнотою (Recall).

Загальні результати ефективності досліджуваних моделей наведені на рисунку 2.5.

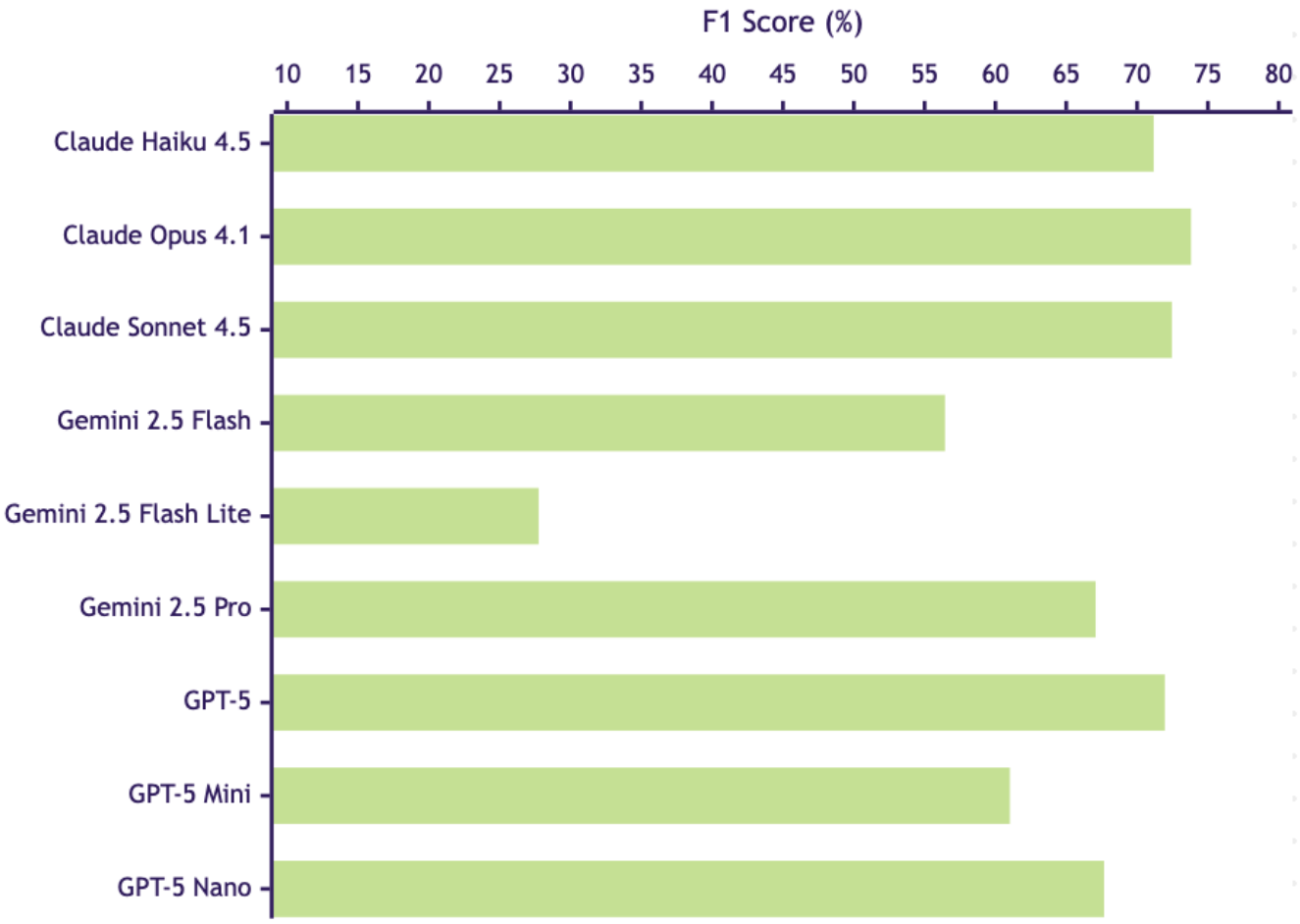


Рисунок 2.5 – Порівняння моделей за F1-Score

Аналіз діаграми демонструє, що лідером за загальною ефективністю стала модель Claude Opus 4.1 з показником F1-Score 73.81%. Щільну конкуренцію їй складають Claude Sonnet 4.5 (72.46%) та GPT-5 (71.96%), що свідчить про високий рівень флагманських моделей усіх виробників.

Варто відзначити неочікувано високий результат "легкої" моделі Claude Haiku 4.5, яка досягла показника 71.18%, випередивши значно потужнішу Gemini 2.5 Pro (67.08%). Це вказує на ефективну оптимізацію архітектури Haiku для задач аналізу коду.

Аутсайдером тестування виявилася модель Gemini 2.5 Flash Lite з критично низьким показником 27.75%, що ставить під сумнів доцільність її використання для задач верифікації VHDL.

Для глибшого розуміння природи помилок та поведінки моделей, доцільно розглянути складові метрики F1-Score окремо (рис. 2.6).

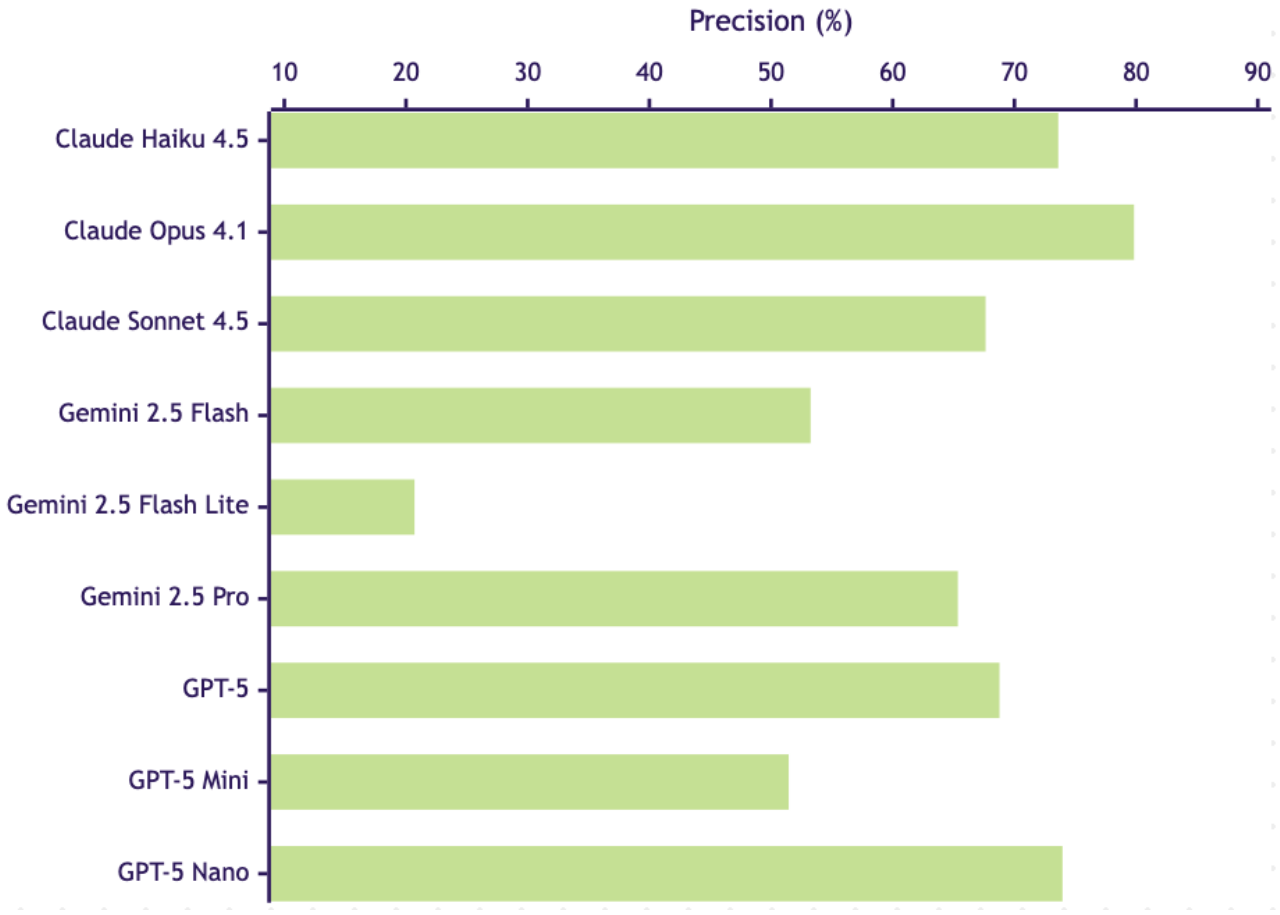


Рисунок 2.6 – Порівняння моделей за Precision

На рисунку 2.6 наведено порівняння моделей за показником точності (Precision), що відображає частку релевантних знахідок серед усіх повідомлень про помилки.

Найвищу точність продемонструвала модель Claude Opus 4.1 (79.83%), що свідчить про її схильність мінімізувати кількість хибних спрацювань. Також високий результат показала GPT-5 Nano (73.95%). Натомість Gemini 2.5 Flash Lite має вкрай низьку точність (20.70%), що свідчить про генерацію великої кількості "шуму" у відповідях.

На рисунку 2.7 представлено порівняння за показником повноти (Recall), який характеризує здатність моделі знайти всі існуючі помилки у коді.

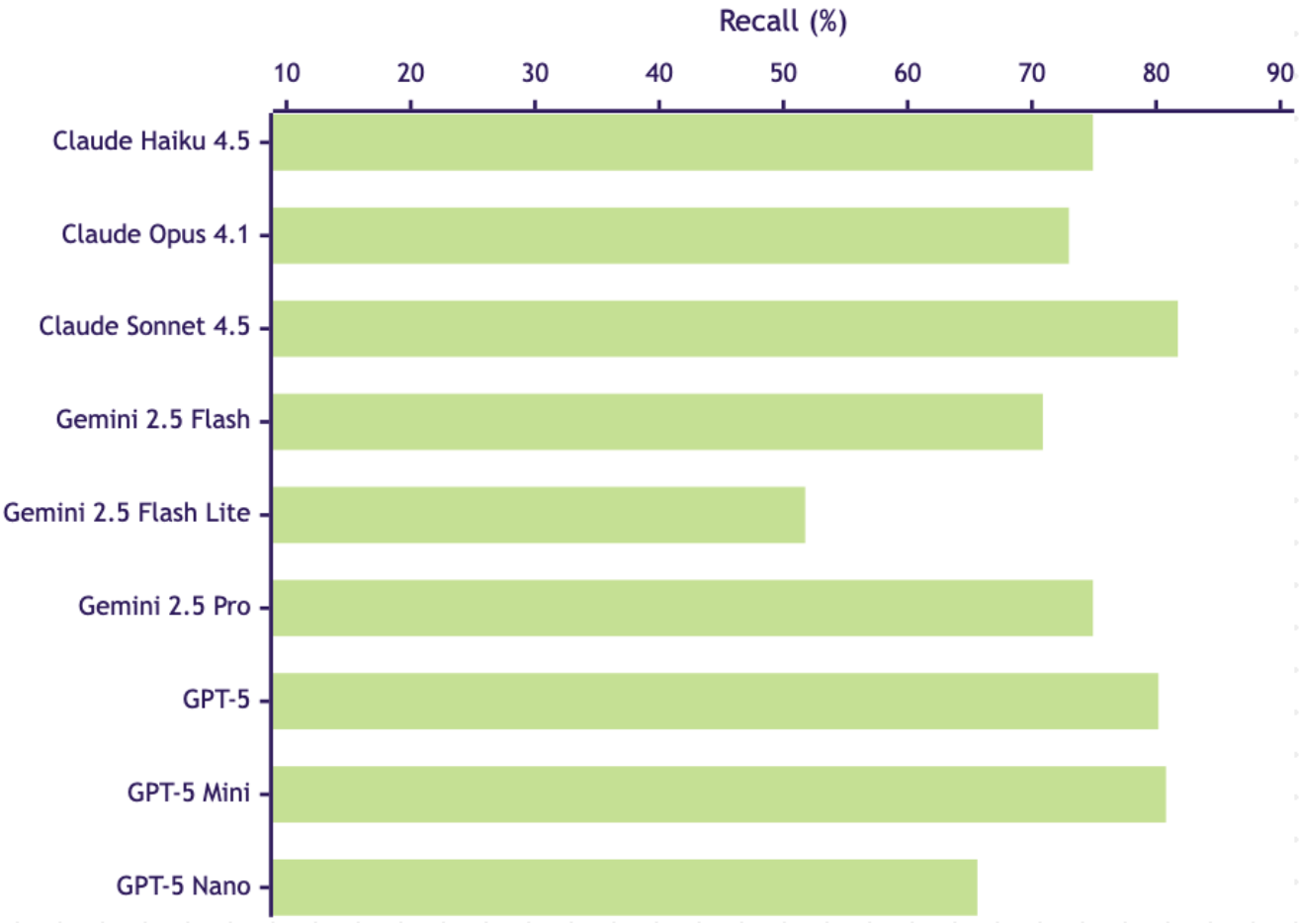


Рисунок 2.7 – Порівняння моделей за Recall

Лідером за пошуковими здібностями стала модель Claude Sonnet 4.5 (81.75%). Також високі результати показали моделі сімейства GPT: GPT-5 Mini

(80.79%) та GPT-5 (80.18%).

Зіставлення метрик дозволяє зробити висновок про різний «характер» моделей: Claude Opus 4.1 працює більш консервативно та точно, тоді як Claude Sonnet 4.5 та GPT-5 Mini налаштовані більш агресивно, знаходячи більше потенційних проблем, але ціною збільшення кількості хибних повідомлень.

2.5.2 Детальний аналіз якості за категоріями помилок

Здатність знаходити приховані логічні помилки є критично важливою характеристикою для інтелектуального асистента, оскільки саме такі дефекти найважче виявити під час компіляції. Результати тестування моделей на цій категорії наведені на рисунку 2.8.

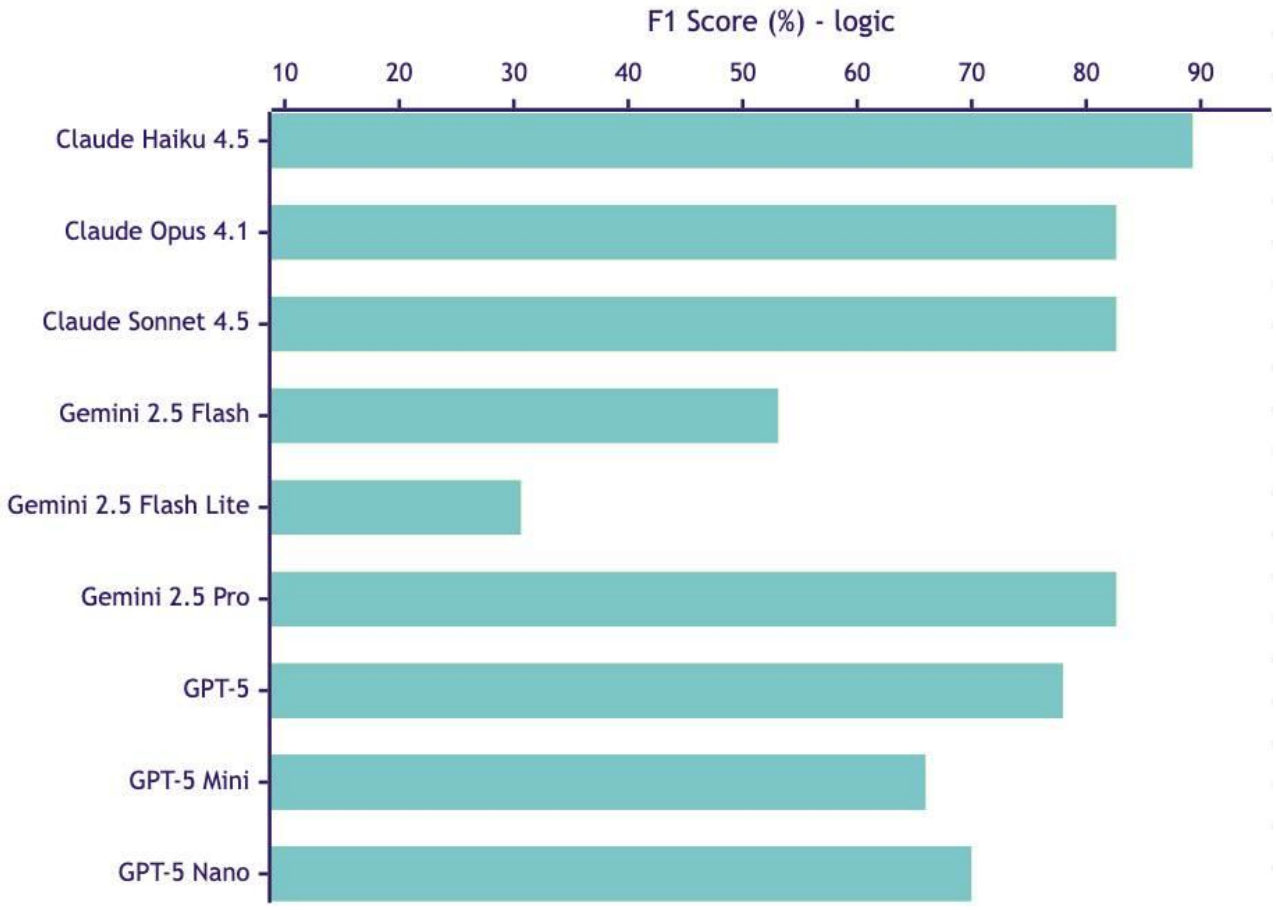


Рисунок 2.8 – Порівняння F1-Score на логічних помилках

Аналіз даних демонструє цікавий феномен: лідером у цій складній категорії стала "легка" модель Claude Haiku 4.5 із показником F1-Score 89.33%. Вона випередила значно потужніші флагманські моделі Claude Opus 4.1, Claude Sonnet

4.5 та Gemini 2.5 Pro, які показали ідентичний результат - 82.67%. Це свідчить про високу ефективність архітектури Haiku саме для розуміння логічних зв'язків у коді без зайвого «ускладнення» контексту. Gemini 2.5 Flash Lite знову підтвердила свою неспроможність у складних задачах (30.67%).

Наступним етапом став аналіз синтаксичних помилок. Хоча сучасні IDE здатні автоматично знаходити більшість таких проблем, для навчальної системи важливо, щоб AI-модель могла чітко пояснити суть синтаксичного порушення. Результати наведені на рисунку 2.9.

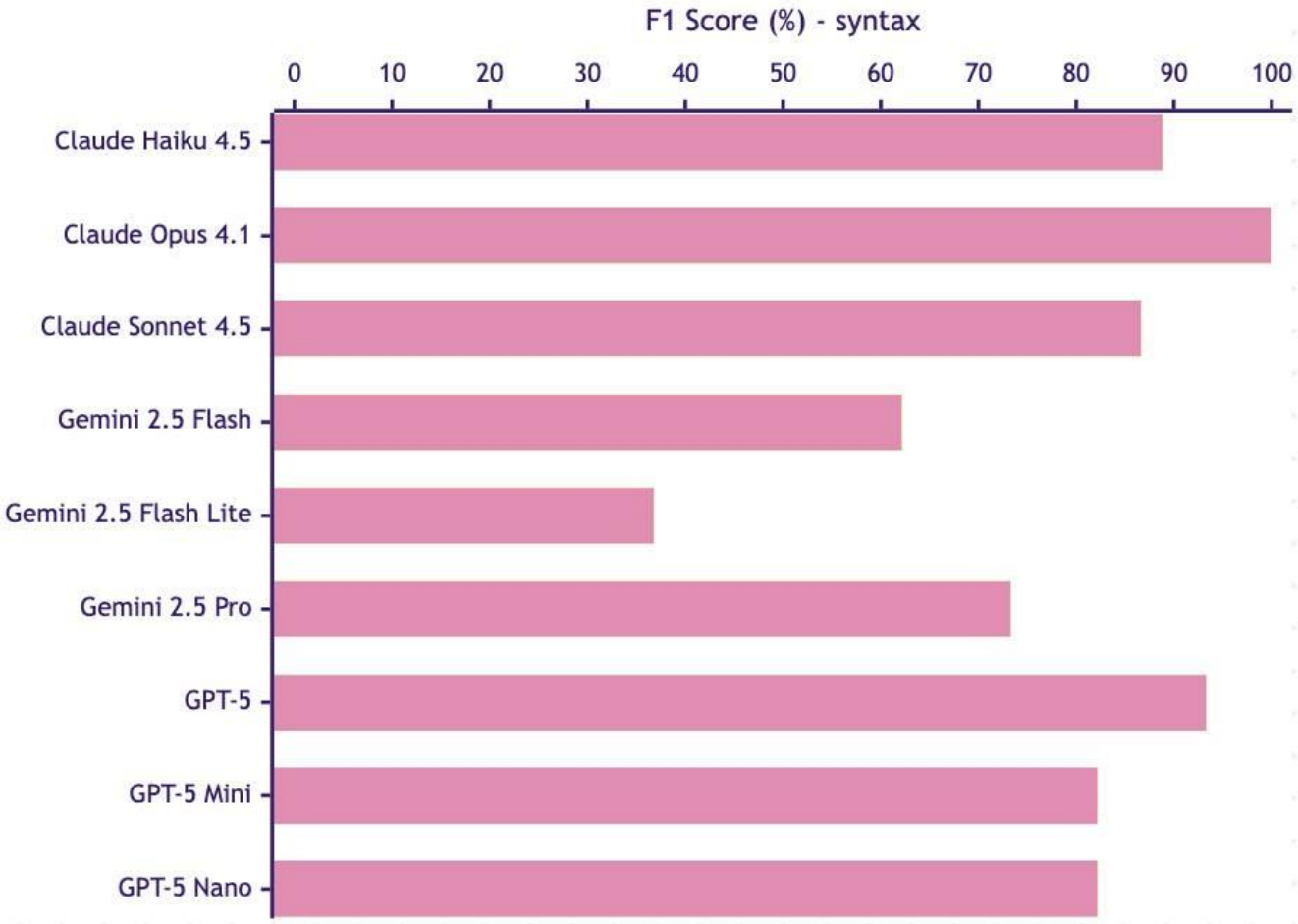


Рисунок 2.9 – Порівняння F1-Score на синтаксичних помилках

У цій категорії модель Claude Opus 4.1 продемонструвала абсолютну точність, досягнувши показника 100%. Це означає, що модель безпомилково ідентифікувала всі синтаксичні дефекти без жодного хибного спрацювання. Високий результат також показала GPT-5 (93.33%). Навіть Claude Haiku 4.5

впоралася дуже добре (88.89%), що робить її надійним інструментом для перевірки синтаксису.

Окрему увагу було приділено проблемам синтезу - конструкціям, які є синтаксично правильними, але призводять до створення неефективної апаратної логіки (наприклад, небажаних засувок). Результати представлені на рисунку 2.10.

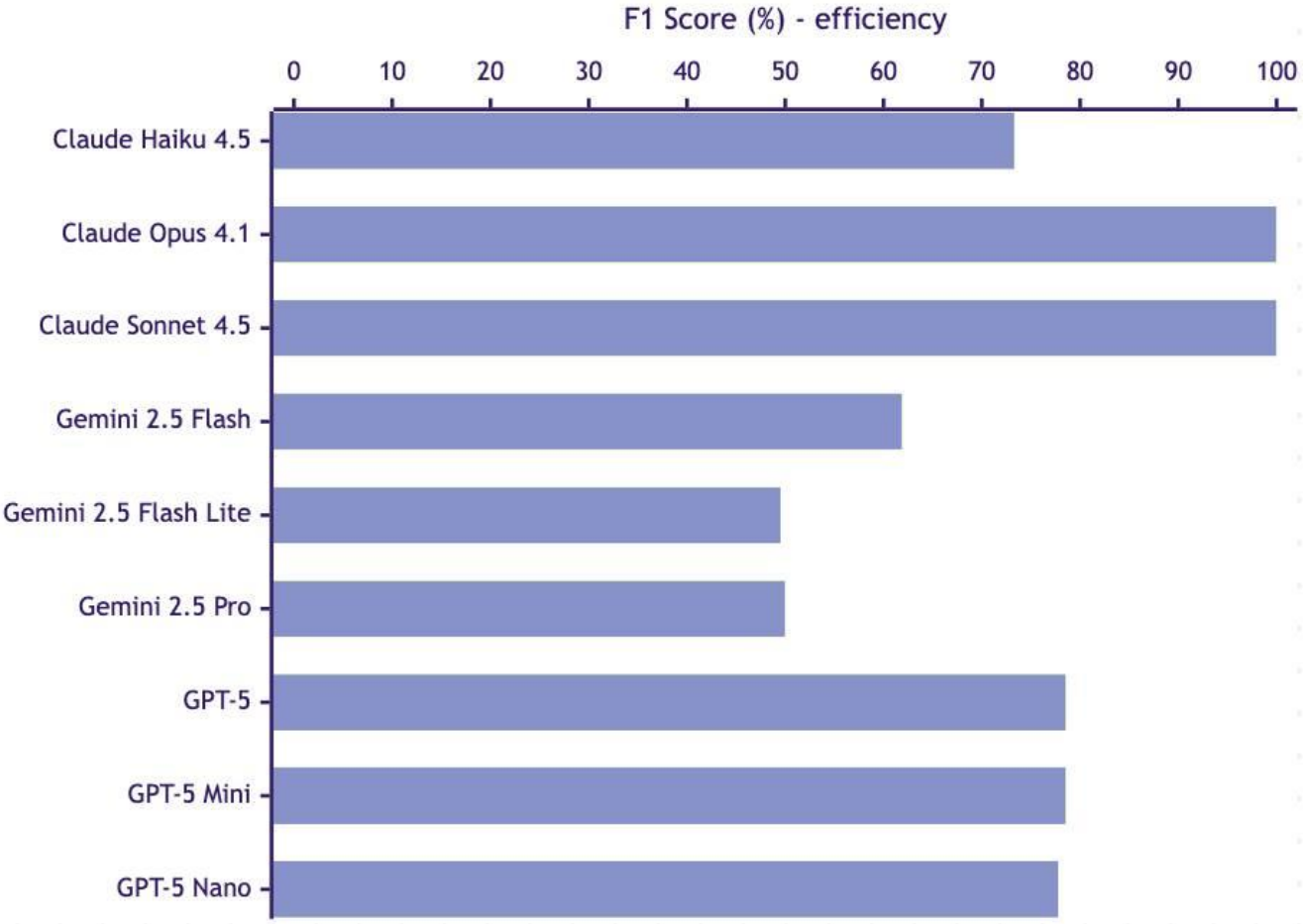


Рисунок 2.10 – Порівняння F1-Score на проблемах синтезу

Тут беззаперечними лідерами стали моделі від Anthropic: Claude Opus 4.1 та Claude Sonnet 4.5 обидві досягли ідеального результату 100%. Це вказує на глибоке «розуміння» цими моделями специфіки проєктування апаратури, де код описує фізичні схеми, а не просто алгоритм. Моделі сімейства GPT також показали гідний результат (~78%), тоді як Gemini 2.5 Pro виявилася менш компетентною у питаннях апаратного синтезу (50.00%).

Четверта категорія - стилістичні помилки (іменування змінних,

форматування, коментарі). Результати зображено на рисунку 2.11.

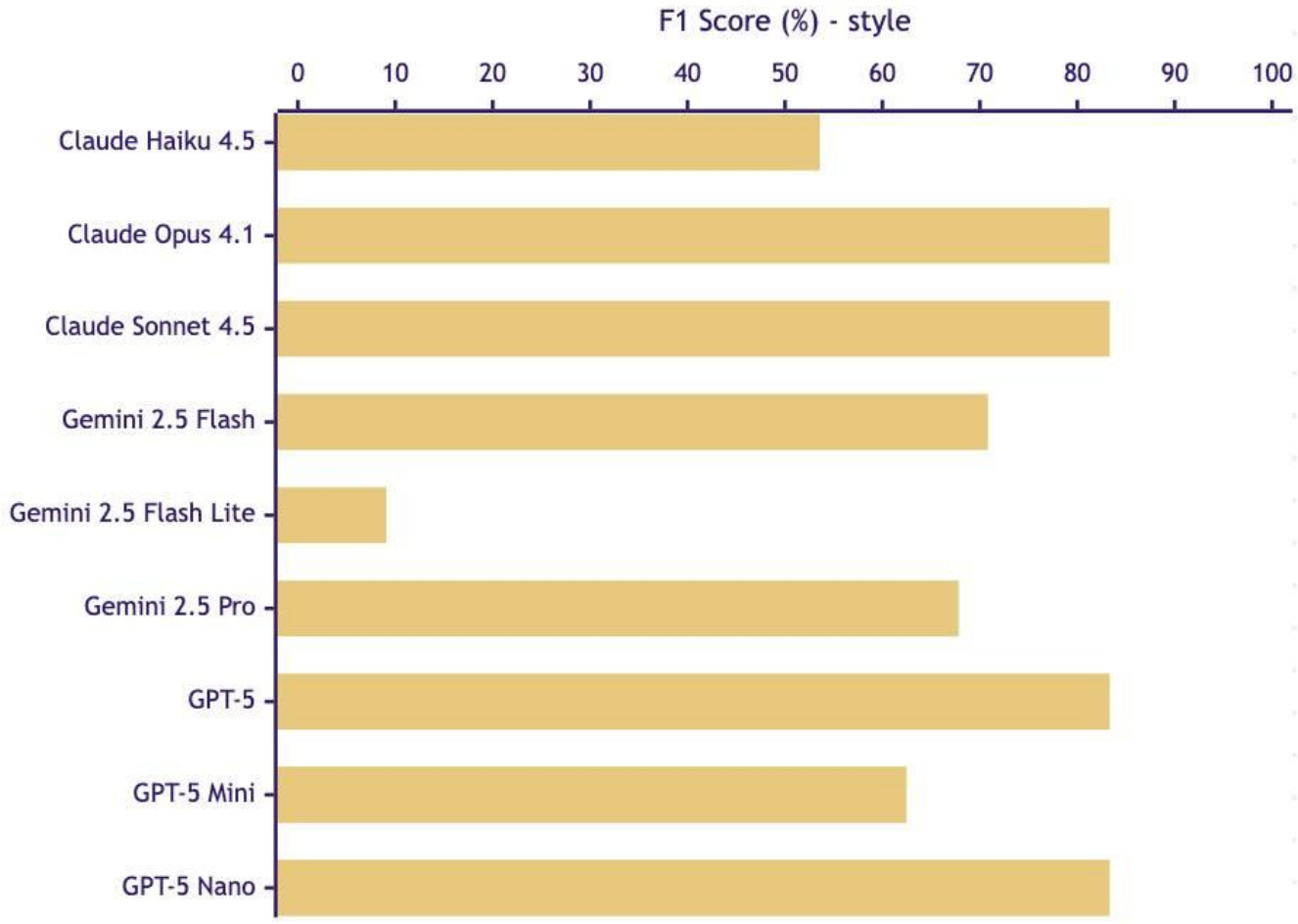


Рисунок 2.11 – Порівняння F1-Score на стилістичних проблемах

Найвищий бал (83.34%) розділили між собою чотири моделі: Claude Opus 4.1, Claude Sonnet 4.5, GPT-5 та GPT-5 Nano. Це свідчить про те, що дотримання стандартів кодування є добре формалізованою задачею, з якою ефективно справляються як великі, так і малі моделі (окрім Gemini 2.5 Flash Lite, яка набрала лише 9.09%).

Найскладнішим випробуванням стали файли зі змішаними помилками, де в одному фрагменті коду зустрічалися проблеми різних типів. Результати тестування наведені на рисунку 2.12.

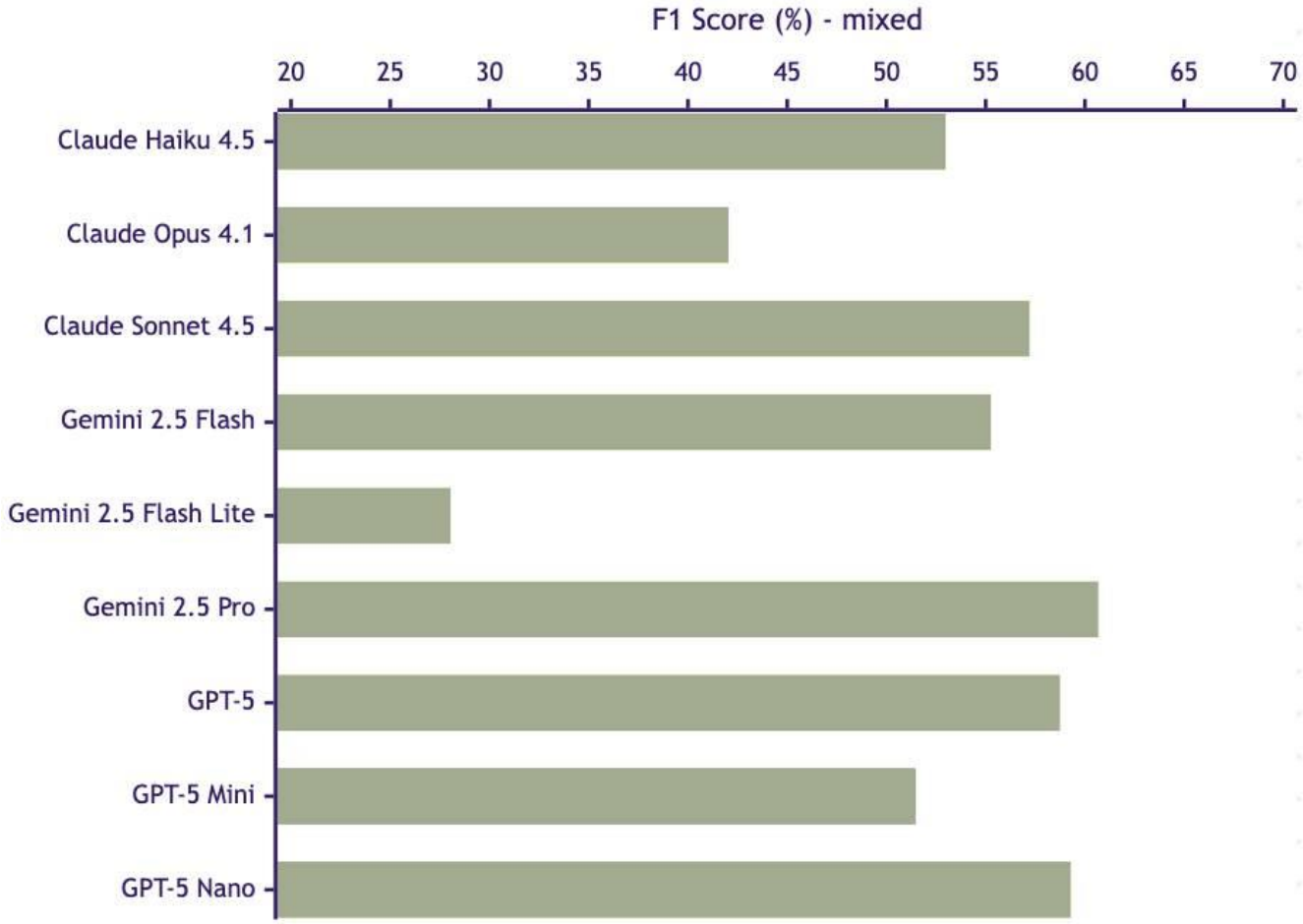


Рисунок 2.12 – Порівняння F1-Score на змішаних проблемах

Результати підтвердили складність цієї категорії, оскільки жодна модель не перетнула позначку в 65%. Несподіваним лідером стала Gemini 2.5 Pro з результатом 60.68%, продемонструвавши найкращу здатність розплутувати клубок різнорідних помилок. На другому місці опинилася GPT-5 Nano (59.31%), що є відмінним показником для моделі такого класу. Варто відзначити значне падіння ефективності Claude Opus 4.1 (42.04%) у цій категорії, що може свідчити про труднощі моделі з фокусуванням уваги при наявності великої кількості різнопланових дефектів одночасно.

2.5.3 Аналіз надійності

Окрім здатності знаходити помилки, критично важливою характеристикою для «AI-асистента» є його надійність - здатність не генерувати помилки там, де їх немає (False Positives). Це особливо важливо для навчальної системи, оскільки хибні зауваження можуть дезорієнтувати студента.

Для оцінки цього параметру було проведено тестування на 5 "чистих" (еталонно коректних) файлах. Ідеальним результатом у цьому тесті є 0 хибних спрацювань. Результати наведені на рисунку 2.13.

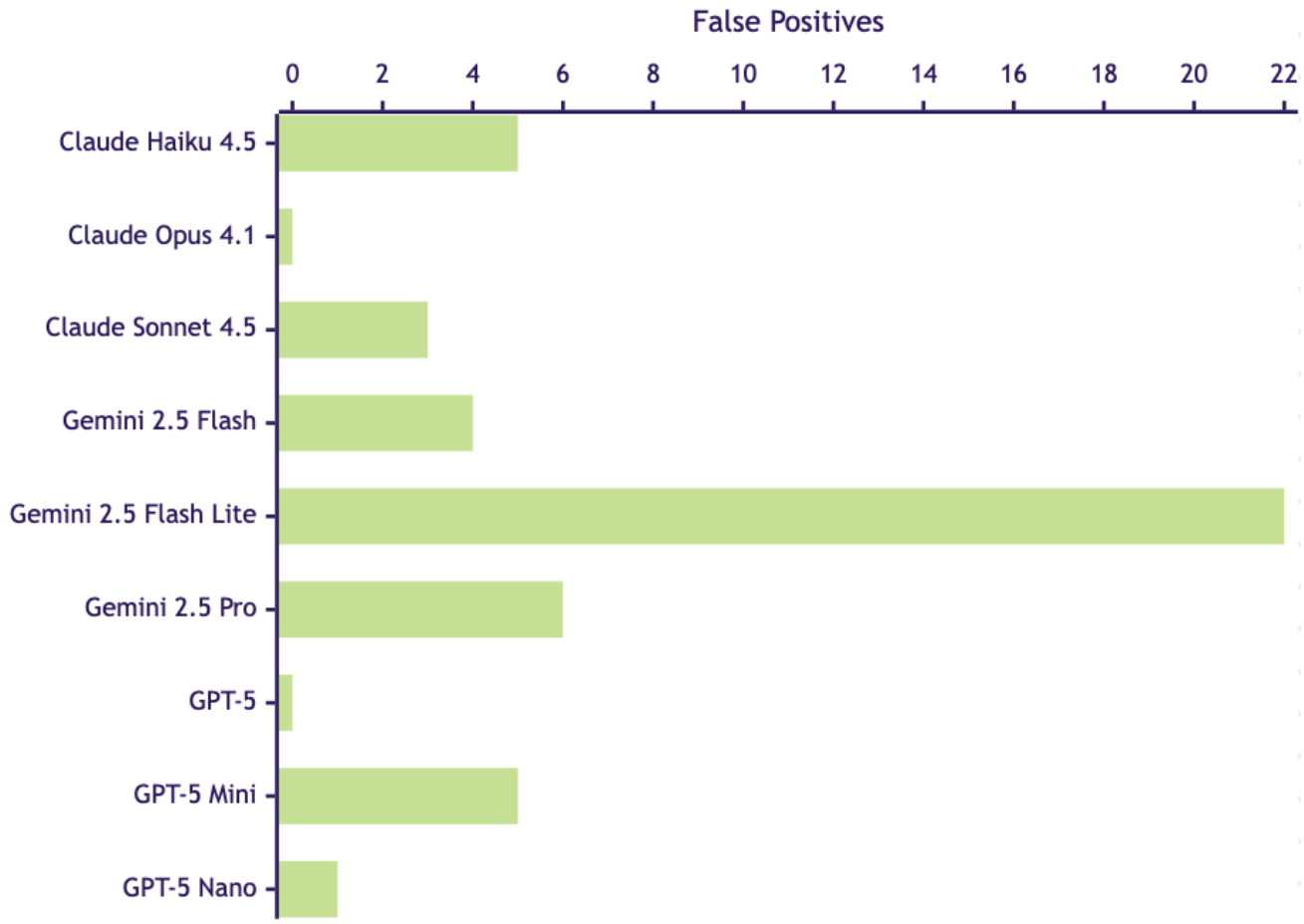


Рисунок 2.13 – Порівняння кількості хибних спрацювань

Результати експерименту виділили двох абсолютних лідерів: Claude Opus 4.1 та GPT-5 продемонстрували еталонну надійність, не згенерувавши жодного хибного повідомлення (0 False Positives). Це підтверджує їхній статус найбільш інтелектуально розвинених моделей, здатних чітко відрізняти коректний код від помилкового.

Окремої уваги заслуговує результат моделі Claude Sonnet 4.5, у якої автоматизована система зафіксувала 3 хибні спрацювання. Детальний ручний аналіз цих випадків показав, що їх не можна кваліфікувати як «галюцинації» чи фактичні помилки. У 100% випадків модель надавала корисні рекомендації зі

стилю ("severity: low"), пропонуючи перейменувати змінні для кращої читабельності (наприклад, замінити input_level на traceback_index) або оптимізувати математичні операції в циклах. Така «надмірна пильність» моделі може бути корисною для консультативної системи, оскільки спрямована на покращення якості коду, а не на введення користувача в оману.

Натомість Gemini 2.5 Flash Lite продемонструвала критично низький рівень надійності, згенерувавши 22 повідомлення. На відміну від Sonnet 4.5, ці спрацювання часто мали характер вигаданих проблем, що робить використання цієї моделі ризикованим без додаткової верифікації.

2.5.4 Аналіз продуктивності та вартості

Для обґрунтування вибору моделі, яка буде інтегрована у вебсистему, необхідно врахувати не лише якість аналізу, але й нефункціональні вимоги: час очікування відповіді (рис. 2.14) та вартість експлуатації (рис. 2.15).

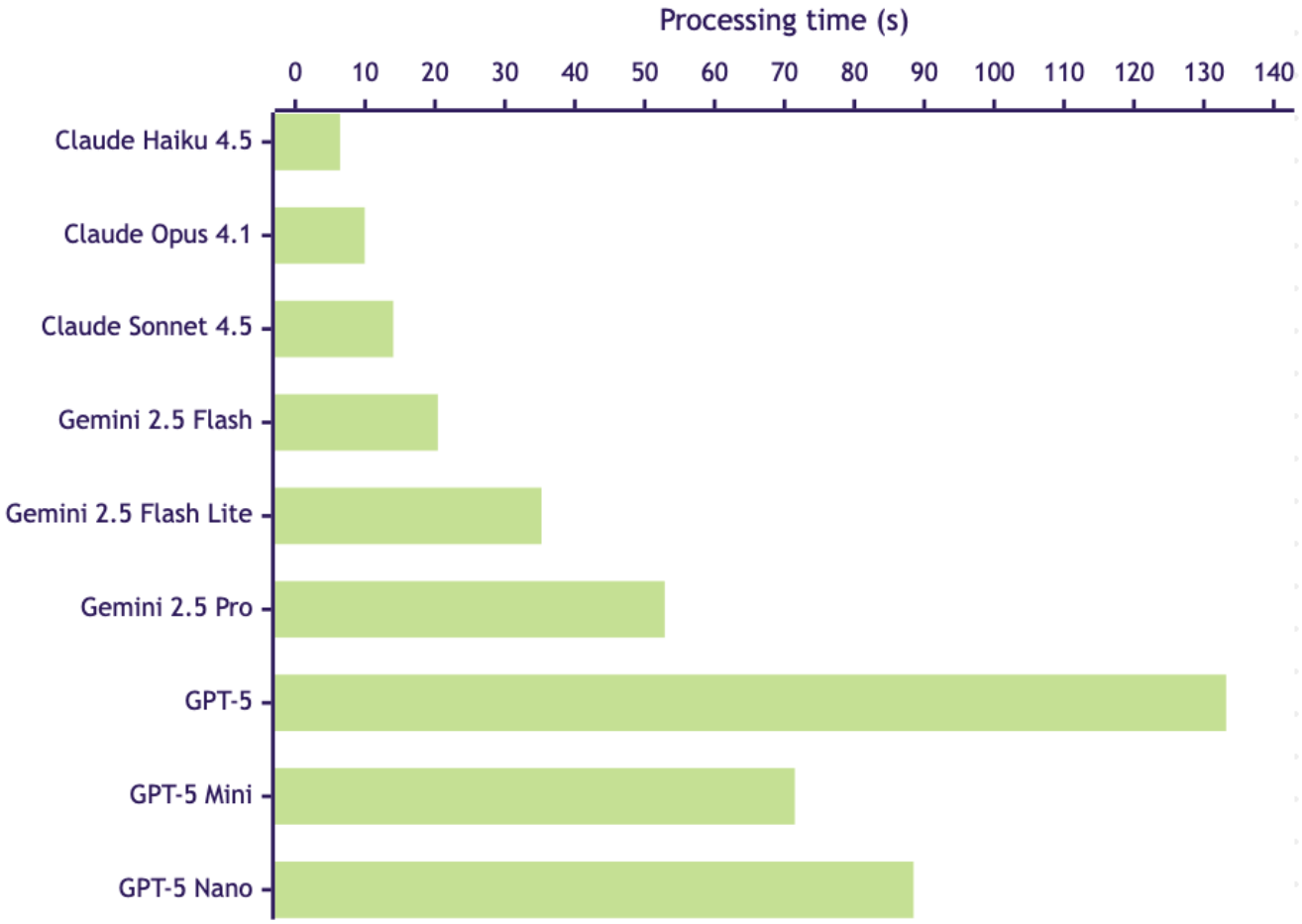


Рисунок 2.14 – Порівняння моделей за середнім часом обробки (с)

Аналіз швидкодії виявив беззаперечного лідера Claude Haiku 4.5, яка обробляє запит у середньому за 6.40 с. Це єдиний результат, що дозволяє реалізувати справді інтерактивний інтерфейс, наближений до реального часу.

Несподівано високу швидкість для свого класу показала флагманська модель Claude Opus 4.1 (9.94 с), випередивши навіть "середню" Claude Sonnet 4.5 (14.02 с).

Натомість модель GPT-5 виявилася найповільнішою у тесті із середнім часом 133.19 с (понад 2 хвилини). Така затримка є критичною для користувацького інтерфейсу (UI/UX) і робить цю модель малопридатною для ролі інтерактивного асистента, попри її високі показники якості. Користувач не готовий чекати так довго на результат перевірки невеликого фрагмента коду.

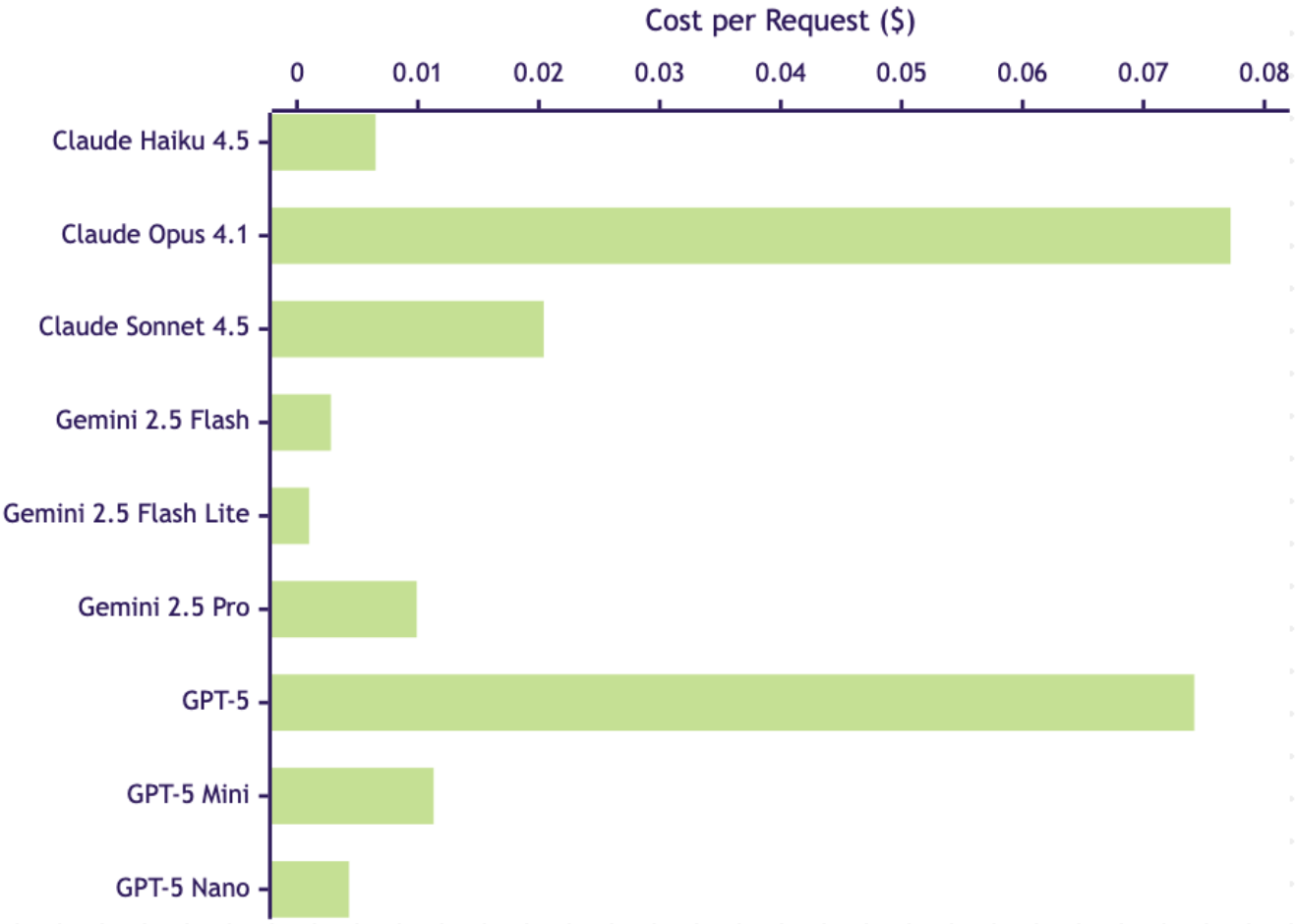


Рисунок 2.15 – Порівняння моделей за середньою вартістю ($)

Економічний аналіз чітко розділяє моделі на три цінові категорії:

– преміум - Claude Opus 4.1 ($0.0772) та GPT-5 ($0.0742), використання їх для рутинної перевірки коду студентами є економічно недоцільним у рамках навчального проєкту;

– середній сегмент - Claude Sonnet 4.5 ($0.0204);

– бюджетний - Claude Haiku 4.5 ($0.0065) та моделі Gemini Flash.

Найкраще співвідношення демонструє Claude Haiku 4.5: вона майже в 12 разів дешевша за флагманів, при цьому, як показав аналіз у попередніх пунктах (зокрема у категорії логічних помилок), не поступається їм за якістю. Найдешевшою є Gemini 2.5 Flash Lite ($0.0010), але враховуючи її низьку надійність (велика кількість помилок), ця економія нівелюється необхідністю постійного перевіряння результатів.

Варто окремо відзначити унікальну конкурентну перевагу сімейства Gemini - наявність повністю безкоштовного рівня доступу до API (Free Tier), який надає Google. Жоден інший провайдер із протестованих (OpenAI, Anthropic) не пропонує подібних умов для розробників. Це робить моделі Gemini (зокрема Flash та Pro) безальтернативним варіантом для реалізації проєктів з нульовим бюджетом, дозволяючи студентам користуватися системою безкоштовно, навіть попри певні компроміси в якості аналізу, виявлені під час тестування.

2.6 Підсумки аналізу AI моделей

Проведене комплексне дослідження дозволило оцінити 9 сучасних AI-моделей за чотирма групами критеріїв: точність виявлення помилок (F1-Score), надійність, швидкодія та вартість.

Підсумовуючи результати аналізу, було визначено найбільш ефективні моделі для різних аспектів роботи системи:

– модель Claude Haiku 4.5 є оптимальним вибором для режиму «інтерактивного асистента», оскільки показала найкращу точність у категорії

логічних помилок (89,33%) при найвищій швидкості обробки запиту;

– модель Claude Sonnet 4.5 доцільно використовувати для сценарію «поглибленого навчання», оскільки вона забезпечує найвищу повноту пошуку та надає корисні рекомендації щодо стилю коду;

– модель Claude Opus 4.1 демонструє еталонну надійність (0 хибних спрацювань) і рекомендується для сценарію "фінальної верифікації", де критичною є відсутність помилок, а не швидкість;

– моделі сімейства Gemini (зокрема Flash та 2.5 Pro) є стратегічно важливими для забезпечення масової доступності системи завдяки наявності повністю безкоштовного рівня доступу до API.

Базуючись на отриманих результатах, для інтеграції в консультативну систему обрано гібридний підхід. Основною моделлю визначено Claude Haiku 4.5 (баланс ціни/якості/швидкості), з передбаченою можливістю перемикання на Gemini 2.5 Flash (для безкоштовного використання). Для більш глибокого аналізу користувачі можуть застосувати Claude Sonnet 4.5, Claude Opus 4.1 або Gemini 2.5 Pro.

3 РОЗРОБКА СИСТЕМИ

3.1 Архітектура системи

Для реалізації поставлених завдань обрано класичну клієнт-серверну архітектуру, яка забезпечує гнучкість, масштабованість та чіткий розподіл відповідальності між компонентами.

Клієнт - це односторінковий застосунок (SPA), з яким безпосередньо взаємодіє користувач. Його відповідальність:

– надання зручного інтерфейсу для введення коду;

– відправка VHDL-коду на сервер для аналізу;

– отримання структурованих даних з результатами від сервера;

– візуалізація отриманих даних (підсвічування коду та відображення інформаційних вікон).

Сервер - це проміжна ланка, що виконує основну логіку. Його відповідальність:

– приймати запити від клієнта через API;

– формувати на основі отриманого коду та заздалегідь визначених інструкцій запит (промпт) до зовнішнього сервісу штучного інтелекту;

– взаємодіяти з API обраної великої мовної моделі;

– отримувати відповідь, обробляти та структурувати її у формат (наприклад JSON), що є зручним для клієнтської частини;

– повертати оброблені дані клієнту.

Діаграма системи зображена на рисунку 3.1.

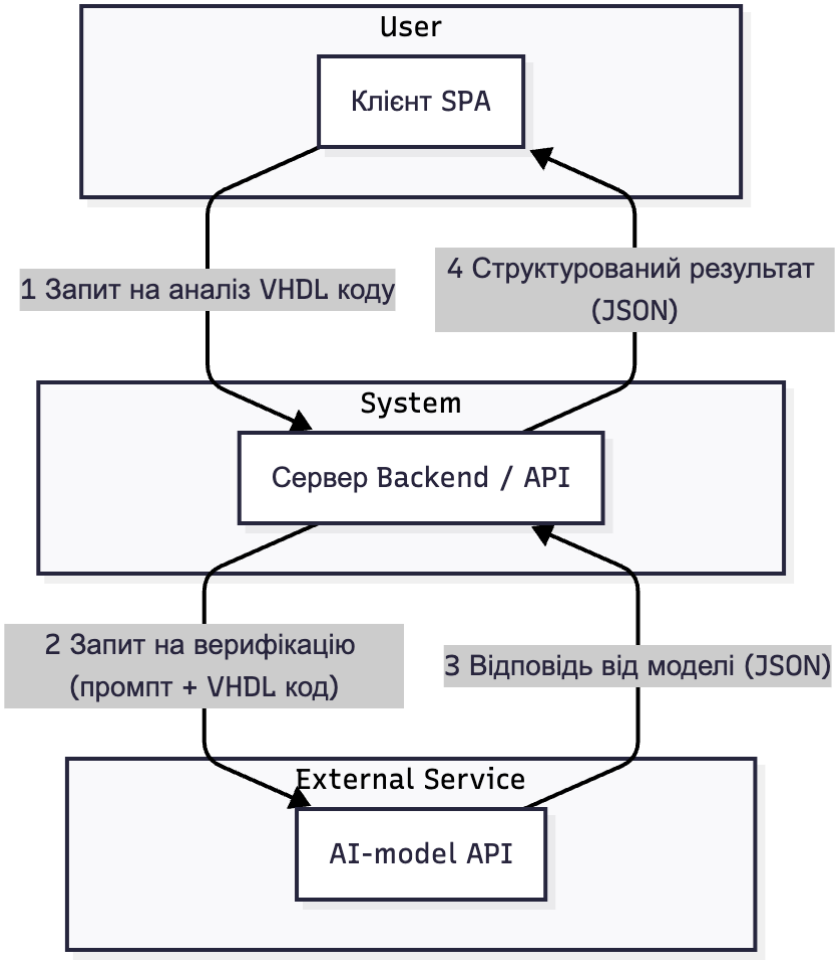


Рисунок 3.1 – Архітектура розроблюваної консультативної системи

Така архітектура дозволяє в майбутньому легко змінювати AI модель на сервері, не вносячи жодних змін у клієнтську частину застосунку.

3.2 Загальна структура програмного забезпечення

Розроблена консультативна система реалізована як сучасний вебзастосунок на базі фреймворку Next.js 15 із використанням архітектури App Router. Програмне забезпечення побудовано за компонентним принципом, що забезпечує гнучкість, масштабованість та простоту підтримки коду. Для гарантування надійності та типізації даних на всіх рівнях застосунку використано мову TypeScript.

Організація файлової структури проєкту виконана згідно зі стандартними конвенціями Next.js, де вихідний код відокремлено у директорію src. Це дозволяє чітко розмежувати конфігураційні файли середовища та логіку застосунку.

Основні директорії проєкту та їх призначення:

– /app - містить логіку маршрутизації застосунку та серверні API-маршрути (api/analyze, api/generate-testbench), що виступають захищеним шлюзом між клієнтом та зовнішніми сервісами штучного інтелекту;

– /components - містить бібліотеку перевикористовуваних компонентів інтерфейсу (наприклад, CodeEditor, ResultsPanel), які відповідають виключно за візуалізацію даних та взаємодію з користувачем;

– /store - відповідає за логіку керування глобальним станом застосунку на клієнтській стороні, містить визначення сховищ даних;

– /services - інкапсулює бізнес-логіку та адаптери для інтеграції із зовнішніми API (сервіси ClaudeService, GeminiService), а також реалізує патерн "Factory" для вибору моделі;

– /types - централізоване сховище TypeScript-інтерфейсів та типів даних, що забезпечує узгодженість структур даних між клієнтом та сервером;

– /config - містить статичні константи, налаштування доступних моделей штучного інтелекту та параметри застосунку за замовчуванням.

Важливою складовою архітектури є система керування станом (State Management). Для реалізації клієнтського стану обрано бібліотеку Zustand, яка забезпечує високу продуктивність завдяки мінімізації зайвих рендерів компонентів та є значно простішим у порівняні з аналогами (Redux, Context API). Глобальний стан застосунку визначається через хук useStore і містить дані, доступні для різних модулів системи.

До складу глобального стану входять такі елементи:

– вихідний VHDL-код, що редагується користувачем у реальному часі;

– результати аналізу, отримані від моделі штучного інтелекту, включаючи список виявлених помилок, їх опис та рівень критичності;

– згенерований код верифікаційного оточення (testbench) та сценарії тестування;

– статус застосунку, що включає індикатори процесу завантаження (isAnalyzing) та повідомлення про системні помилки.

Архітектура інтерфейсу користувача побудована на основі бібліотеки компонентів Material UI (MUI). Це дозволило створити уніфікований дизайн та забезпечити адаптивність системи. Структура інтерфейсу розділена на компоненти макета (Layout) та функціональні компоненти.

Основні функціональні компоненти системи:

– CodeEditor - компонент редактора коду, що забезпечує підсвітку синтаксису, нумерацію рядків та візуальне відображення місць помилок;

– ResultsPanel - динамічна панель, що відображає результати аналізу або згенерований тестбенч залежно від обраної вкладки інтерфейсу;

– ModelSelector - компонент керування, що дозволяє користувачу перемикати активну AI модель, взаємодіючи безпосередньо з глобальним сховищем даних;

– TestbenchDialog - модальне вікно для налаштування параметрів симуляції перед генерацією тестового оточення.

Зовнішній вигляд розробленого інтерфейсу та розташування основних функціональних компонентів наведено на рисунку 3.2.

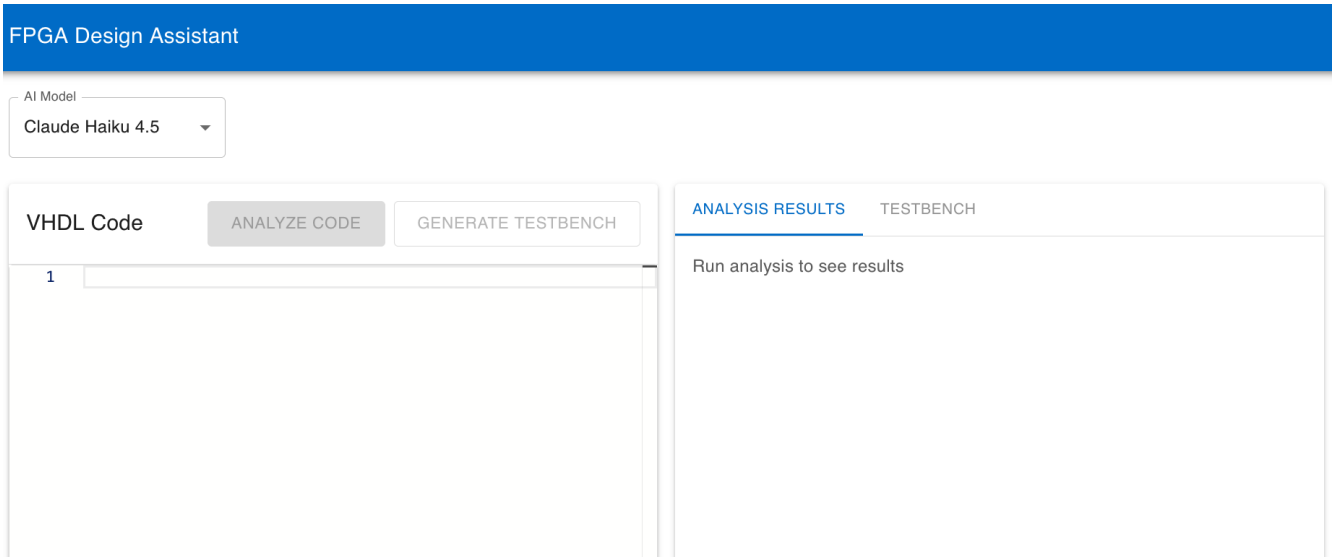


Рисунок 3.2 – Інтерфейс користувача та основні функціональні компоненти

Така модульна архітектура забезпечує чіткий розподіл відповідальності: компоненти відповідають лише за відображення, сервіси - за бізнес-логіку, а сховище - за цілісність даних.

3.3 Реалізація модулю аналізу коду

Реалізація функціоналу аналізу коду базується на ланцюжку асинхронних операцій, що забезпечують передачу вихідного VHDL-коду з клієнтського інтерфейсу до обраної моделі штучного інтелекту та зворотну обробку результатів.

Процес взаємодії з системою розпочинається у компоненті CodeEditor, який використовує бібліотеку Monaco Editor для забезпечення повноцінного середовища редагування. Введені користувачем дані синхронізуються з глобальним сховищем Zustand за допомогою дії setVhdlCode. Ініціалізація аналізу

відбувається за подією натискання кнопки "Аналізувати", яка викликає асинхронну функцію-обробник. Дана функція виконує підготовку JSON-навантаження (payload), що містить поточний код та ідентифікатор обраної моделі, і відправляє POST-запит до внутрішнього API.

Серверна частина реалізована через API-маршрут Next.js (src/app/api/analyze/route.ts), який виконує роль захищеного шлюзу. Це дозволяє приховати ключі доступу до зовнішніх сервісів від клієнтського браузера.

Алгоритм обробки запиту на сервері включає такі етапи:

– валідація вхідних даних - перевірка наявності коду та коректності ідентифікатора моделі;

– вибір провайдера послуг - використання патерну проєктування «Фабрика» (AIServiceFactory) для створення екземпляра сервісу;

– виконання запиту - виклик методу analyzeVHDL відповідного сервісу (ClaudeService або GeminiService);

– повернення результату - відправка обробленої відповіді клієнту у форматі JSON.

Ключовим архітектурним рішенням є використання патерну «Фабрика» для організації сервісного шару. Клас AIServiceFactory аналізує конфігурацію обраної моделі та повертає конкретну реалізацію інтерфейсу AIService. Такий поліморфізм дозволяє системі уніфіковано працювати з різними провайдерами (Anthropic, Google) без зміни основної бізнес-логіки контролера.

Окремої уваги заслуговує етап парсингу та нормалізації даних, оскільки великі мовні моделі повертають відповідь у текстовому форматі.

Процедура перетворення відповіді у структуровані дані реалізована наступним чином:

– промпт-інженіринг - формування запиту з чіткою вимогою повертати результат виключно у форматі JSON згідно із заданою схемою;

– екстракція даних - застосування регулярних виразів для виокремлення JSON-об'єкта з текстової відповіді моделі, відсіюючи можливий супровідний текст;

– валідація структури - перевірка отриманого об'єкта на відповідність TypeScript-інтерфейсу AnalysisResult (наявність полів issuesFound, reasoning);

– мапінг даних - приведення виявлених помилок до внутрішнього формату системи, присвоєння унікальних ідентифікаторів та нормалізація рівнів критичності.

Для візуалізації описаного алгоритму руху та обробки даних розроблено діаграму послідовності, яка наведена на рисунку 3.3.

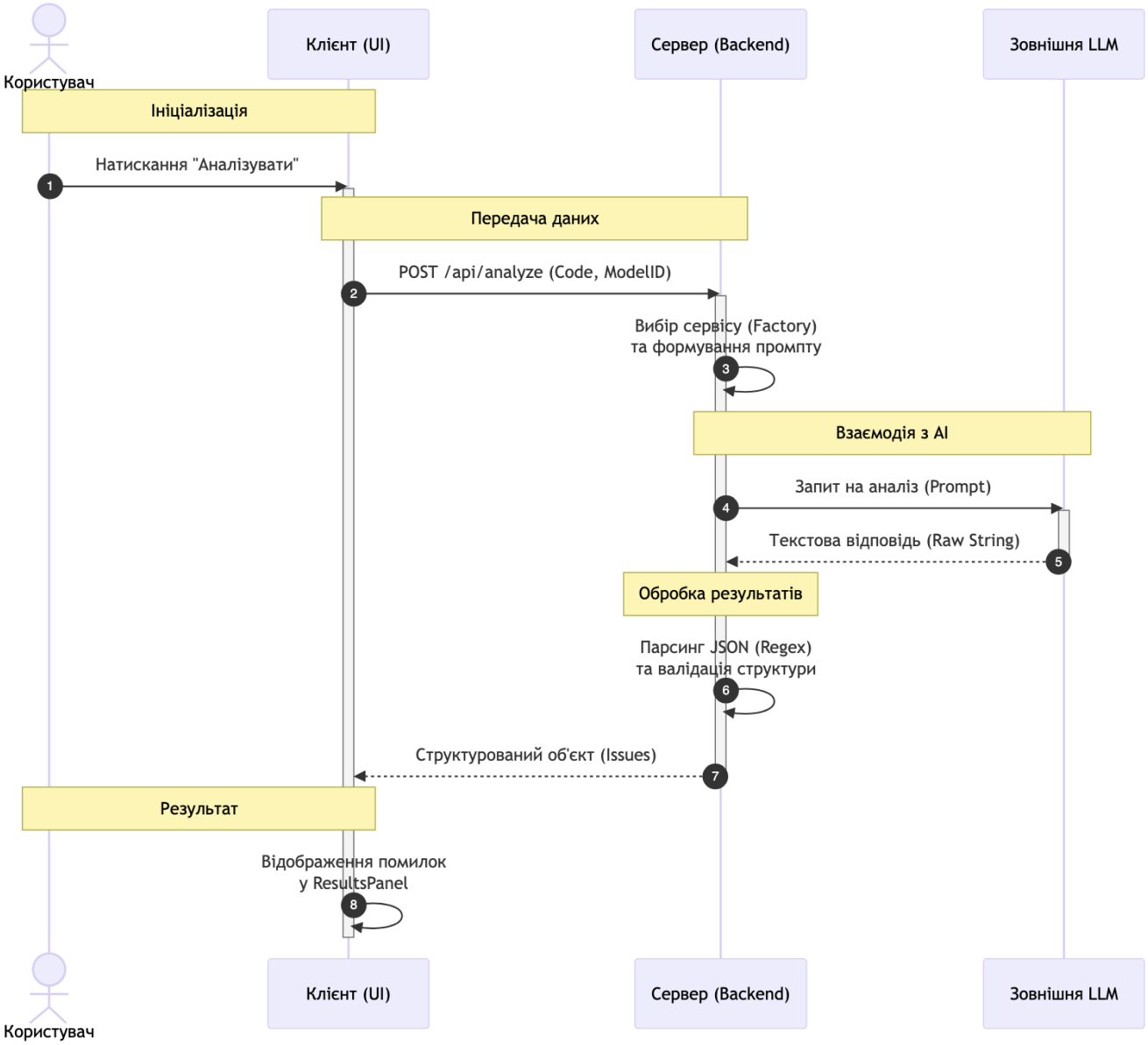


Рисунок 3.3 – Діаграма послідовності процесу аналізу коду

Візуалізація результатів здійснюється компонентом ResultsPanel. Він підписується на оновлення сховища даних та автоматично відображає список знайдених проблем. Помилки програмно групуються за рівнями критичності (Critical, High, Medium, Low). Кожна проблема відображається у вигляді інтерактивної картки, що містить опис, категорію помилки та рекомендації щодо виправлення, а також вказує на відповідні рядки коду в редакторі.

Візуалізація роботи модулю аналізу, включаючи відображення відповіді від AI моделі на панелі результатів, наведена на рисунку 3.4.

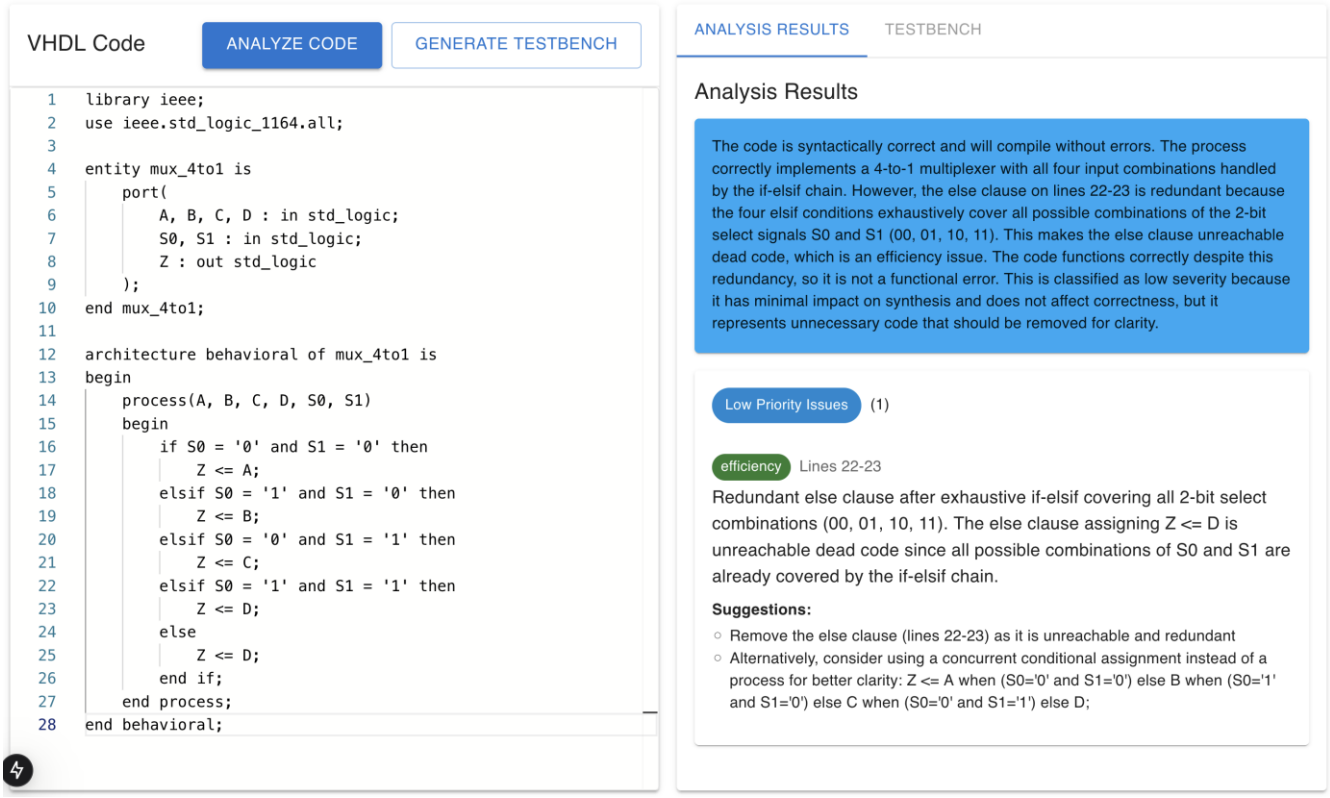


Рисунок 3.4 – Відображення результатів аналізу коду VHDL

Як видно на рисунку модель успішно визначила незначну проблему синтезу, яку віднесла до категорії "low", про що свідчить зелений індикатор поля де вказано категорію помилки.

3.4 Реалізація модулю генерації верифікаційного оточення

Модуль генерації верифікаційного оточення призначений для автоматизованого створення VHDL-коду тестбенчів (testbenches), що дозволяє перевірити коректність роботи спроєктованої апаратури у симуляторах. Реалізація цього модулю базується на інтерактивній взаємодії користувача з системою та використанні генеративних можливостей великих мовних моделей.

Важливою особливістю клієнтської реалізації є механізм попередньої валідації. Кнопка "Generate Testbench" стає активною для користувача виключно за умови наявності введеного коду та, що критично важливо, відсутності критичних помилок після етапу аналізу. Такий підхід запобігає витрачанню ресурсів API на генерацію тестів для, знаючи наперед, непрацездатного коду.

У процесі розробки було прийнято архітектурне рішення відмовитися від використання регулярних виразів для парсингу сутностей (Entity) на клієнті. Замість цього система делегує задачу семантичного розбору структури портів та сигналів безпосередньо AI моделі. Це спрощує логіку клієнтського застосунку та підвищує стійкість системи до складних синтаксичних конструкцій VHDL.

Конфігурація параметрів майбутнього тесту здійснюється через компонент TestbenchDialog. Користувач має можливість задати текстовий опис сценаріїв тестування, період тактового сигналу та тривалість симуляції. Дані параметри об'єднуються в об'єкт TestbenchScenario і передаються на серверну частину через API-маршрут /api/generate-testbench.

Ключовим елементом підсистеми є алгоритм конструювання промпту (createTestbenchGenerationPrompt), реалізований на стороні сервера, лістинг якого наведено у додатку А.1. Для побудови інструкцій обрано англійську мову, оскільки це забезпечує вищу якість генерації коду моделями, що навчалися переважно на англомовній технічній документації.

Структура промпту для генерації включає наступні компоненти:

– контекст - повний вихідний код модуля (Design Under Test), для якого

створюється тест;

– параметри симуляції - конкретні значення часових параметрів (clock period, simulation time), задані користувачем або встановлені за замовчуванням;

– структурні вимоги - чіткий перелік інструкцій для AI моделі, що включає вимогу оголосити компонент, створити процес генерації тактового сигналу (clock process) та реалізувати процес стимулів (stimulus process) згідно з описом користувача;

– формат виводу - директива повертати виключно чистий VHDL-код без Markdown-форматування або пояснювального тексту.

Отриманий від моделі код автоматично зберігається у глобальному стані застосунку та відображається у відповідній вкладці інтерфейсу в режимі "тільки для читання". Для інтеграції з зовнішніми середовищами розробки реалізовано функції швидкого копіювання коду та завантаження файлу у форматі .vhd.

Інтерфейс налаштування параметрів симуляції (період тактового сигналу, опис сценаріїв) реалізовано у вигляді модального вікна (рис. 3.5).

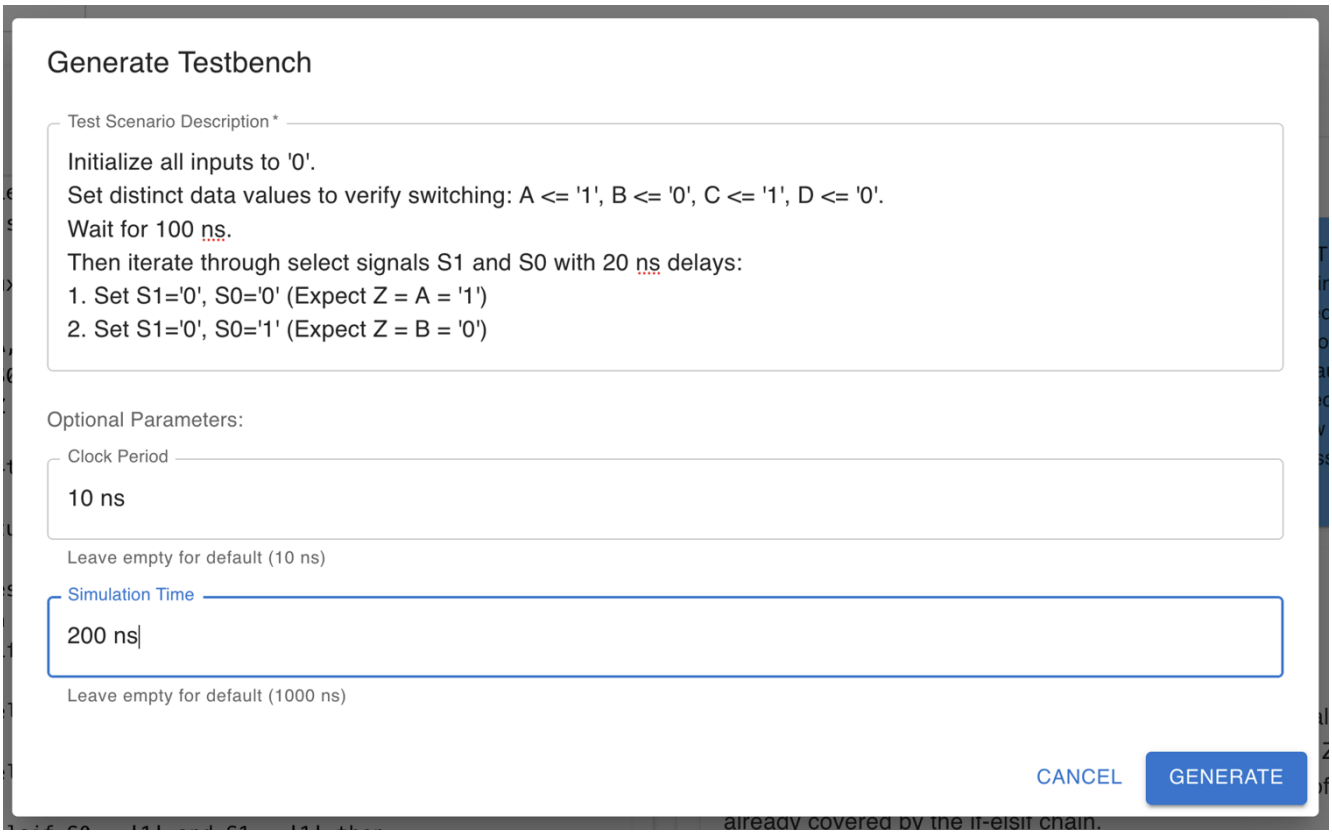


Рисунок 3.5 – Діалогове вікно генерації тестового оточення

Результат роботи генеративного модуля - готовий до компіляції VHDL-код, що відображається у відповідній вкладці панелі результатів, представлено на рисунку 3.6.

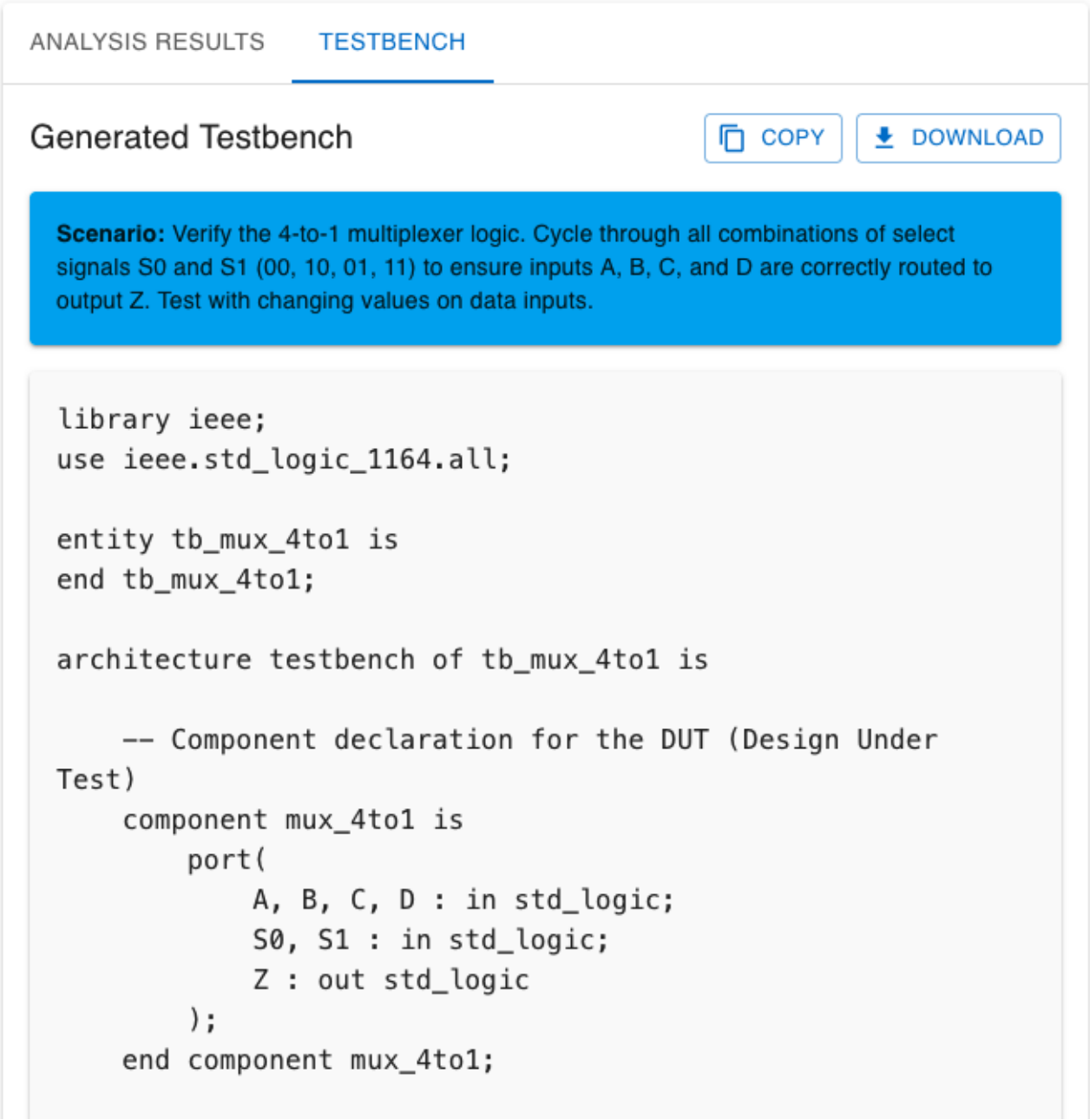


Рисунок 3.6 – Результат роботи генеративного модуля

Згенерований код тестового оточення користувач може зручно копіювати у свій буфер обміну або завантажити отриманий vhd файл на свій персональний комп’ютер.

3.5 Особливості інтеграції розробленої системи з LLM та використання промпт-інженірингу

Ефективність роботи консультативної системи критично залежить від якості інтеграції з великими мовними моделями та стратегії формування запитів. У розробленій системі застосовано комплексний підхід, що поєднує структурні обмеження виводу та семантичне налаштування контексту.

3.5.1 Використані стратегії промпт-інженірингу

Для забезпечення високої точності та машинної читабельності відповідей використано наступні стратегії промпт-інженірингу:

– шаблон "Персона" (Persona Pattern) - хоча інструкції подаються у вигляді системних повідомлень, вони сформульовані таким чином, щоб модель імітувала поведінку старшого інженера з верифікації FPGA, що дозволяє виявляти специфічні нюанси VHDL (наприклад, застарілі конструкції типу CLK'event);

– ін'єкція контексту (Context Injection) - вихідний код передається у промпт повністю з використанням розмітки Markdown, що дозволяє моделі аналізувати перехресні зв'язки між сигналами та портами, які неможливо виявити при порядковому аналізі;

– обмеження формату виводу (JSON Enforcement) - для модуля аналізу встановлено сувору вимогу повертати відповідь виключно у форматі JSON згідно із заданою схемою, що забезпечує стабільний парсинг на стороні сервера;

– ланцюжок думок (Chain-of-Thought) - схема відповіді включає обов'язкове поле reasoning, що змушує модель спочатку сформулювати логічне обґрунтування аналізу, і лише потім деталізувати конкретні помилки, що значно підвищує якість детекції.

3.5.2 Обґрунтування вибору мови взаємодії з LLM

Архітектурним рішенням проєкту було визначення англійської мови як єдиного стандарту як для інтерфейсу користувача, так і для системних промптів та внутрішньої генерації коду. Такий підхід забезпечує цілісність системи та

базується на комплексі трьох ключових техніко-економічних факторів:

– розрив у представленні знань (Knowledge Representation Gap) - масштабне домінування англомовних текстів у навчальних датасетах змушує моделі використовувати англійську як опорну мову, що спричиняє ризик семантичних викривлень при обробці запитів іншими мовами;

– економічна ефективність та "податок на токени" (Token Tax) - неефективна токенізація кириличних символів призводить до значного зростання вартості використання API та нераціонального витрачання лімітів контексту порівняно з латиницею;

– продуктивність та точність відповідей - результати спеціалізованих тестувань підтверджують зниження якості виконання завдань на логічні міркування при відмові від англійської мови, яка є рідною для синтаксису VHDL.

Деталізуючи перший фактор, варто зазначити, що згідно з технічними звітами розробників моделі Llama 2, частка англомовних текстів у навчальному датасеті складає 89,7%, тоді як частка української мови є меншою за 0,1% [14]. Використання української мови для опису складних технічних завдань створює додатковий шар "латентного перекладу" всередині моделі, що підвищує ймовірність втрати контексту [15]. Аналогічна ситуація спостерігається у загальнодоступних наборах даних Common Crawl, де англійська мова займає близько 45-50% обсягу, значно випереджаючи будь-яку іншу мову [16].

Стосовно другого фактору, практичні дослідження показують, що передача ідентичного за змістом повідомлення українською мовою коштує значно дорожче, ніж англійською. Це пов'язано з особливостями токенізації, де кириличні слова розбиваються на більшу кількість фрагментів [17, 18].

Порівняльна статистика вартості токенізації для різних мов відносно англійської наведена на рисунку 3.7.

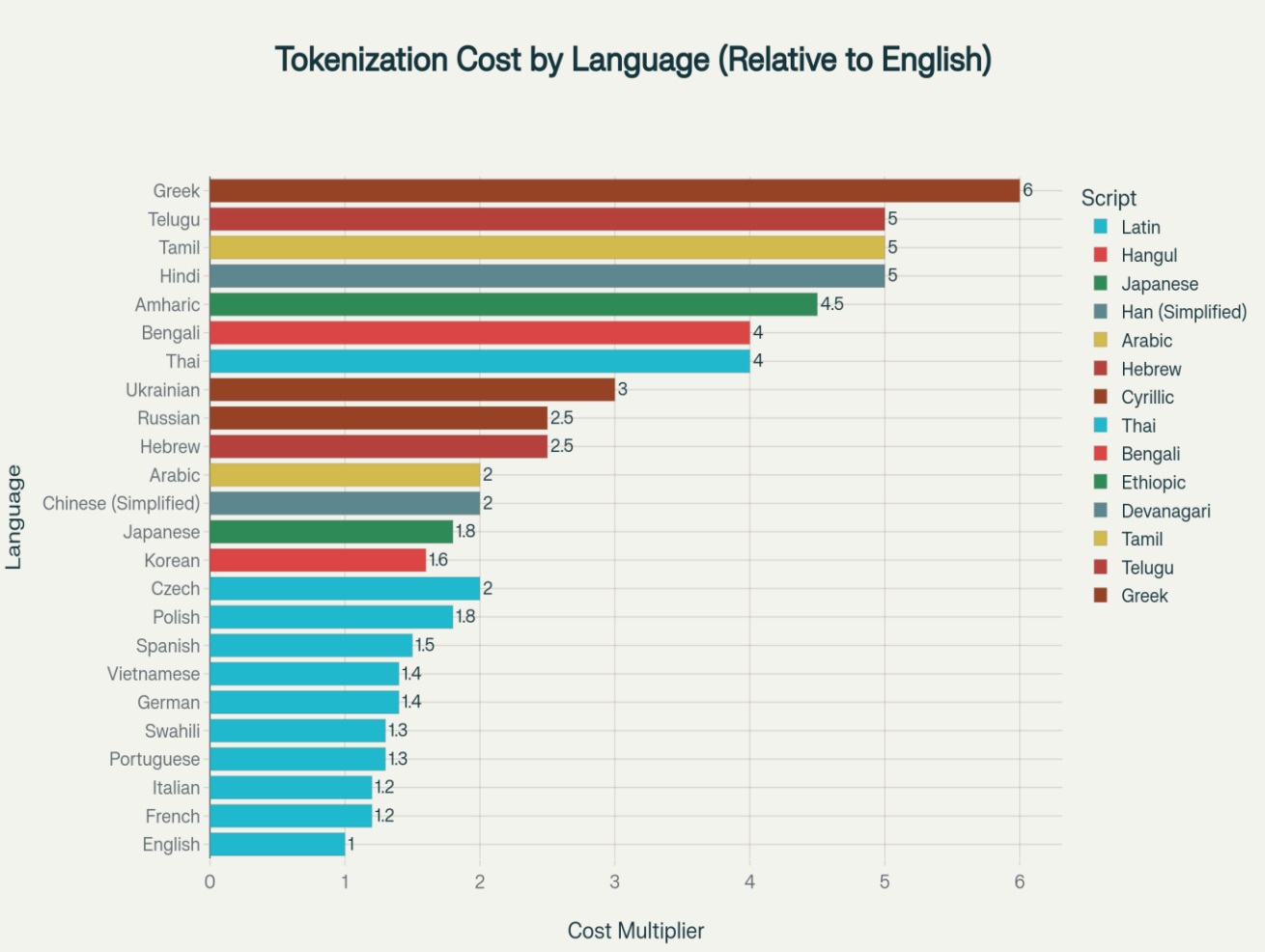


Рисунок 3.7 – Вартість токенізації за мовами [17]

Як видно з діаграми, коефіцієнт витрат для української мови становить 3.0, що робить україномовні запити втричі дорожчими за аналогічні англомовні. Детальніший аналіз ефективності різних систем письма наведено на рисунку 3.8.

Графік демонструє, що кириличні системи письма (Cyrillic) мають значно вищий множник вартості (Cost Multiplier) порівняно з латиницею (Latin). Це підтверджує доцільність використання англійської мови для оптимізації витрат.

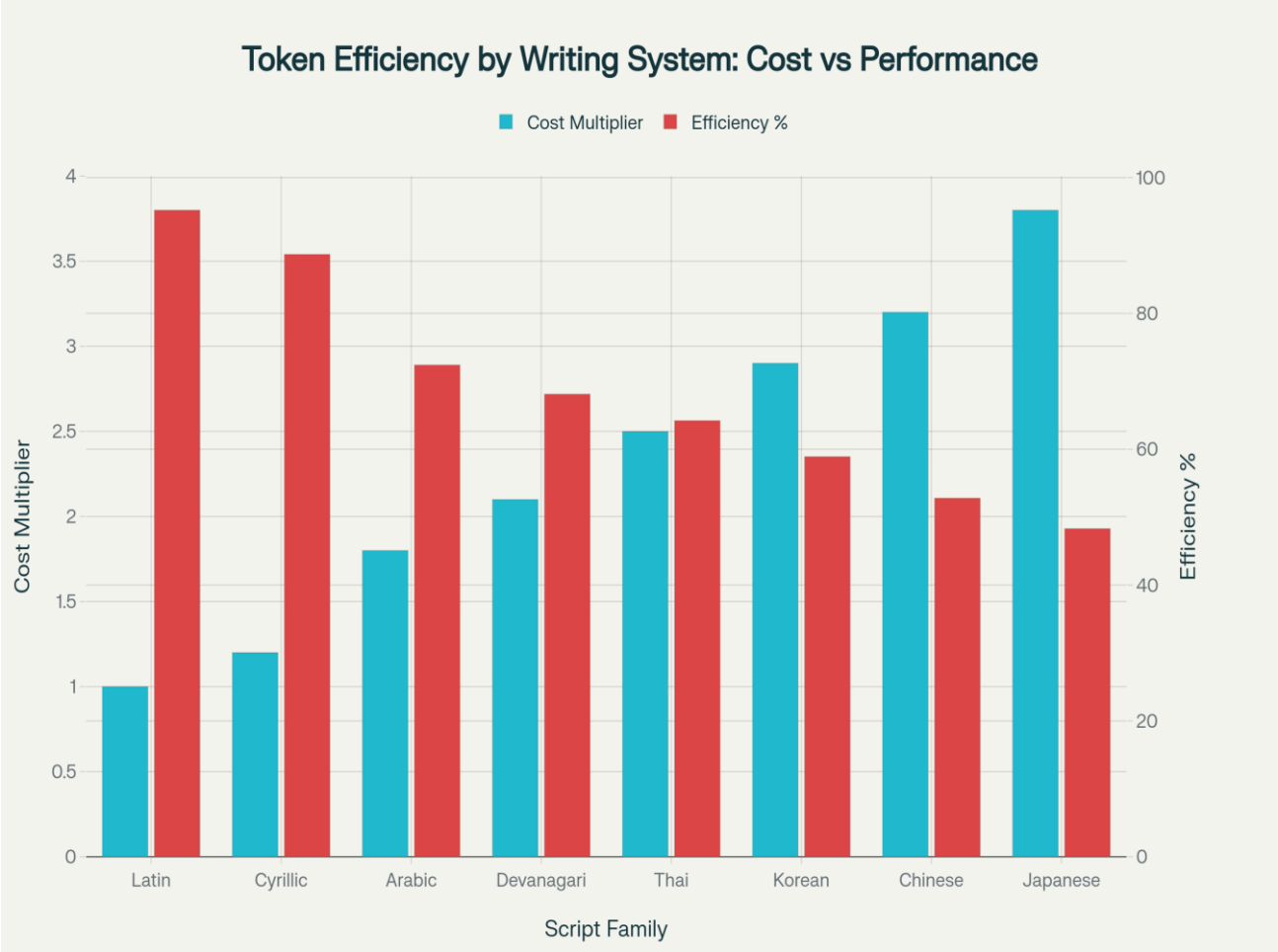


Рисунок 3.8 – Ефективність токенізації за системами письма [17]

Щодо третього фактора, спеціалізовані бенчмарки, такі як MMLU-ProX, демонструють, що загальна продуктивність моделей може падати при переході з англійської на інші мови, а розрив у точності (accuracy gap) для низькоресурсних мов сягає 20–25% [19]. Проте, при звуженні вибірки до української та основних європейських мов, дослідження показують значно меншу деградацію - в межах 5%. Враховуючи це, можна стверджувати, що втрата точності при використанні української мови є мінімальною і не є критичною для загальної логіки роботи системи.

Втім, оскільки VHDL є формалізованою англоподібною мовою, використання англійської в системних промптах забезпечує пряму термінологічну відповідність та усуває навіть мінімальні ризики неоднозначного трактування специфічних конструкцій (наприклад, sensitivity list, process, entity).

Підсумовуючи, вибір англійської мови як основного інструменту взаємодії з

LLM є стратегічним інженерним рішенням, продиктованим трьома факторами: домінуванням англійської у навчальних даних (що гарантує найкраще розуміння контексту), суттєвою економією ресурсів завдяки ефективній токенізації (зниження вартості до 3 разів) та забезпеченням максимальної термінологічної точності при роботі з кодом VHDL.

3.5.3 Технічна реалізація архітектури інтеграції

Програмна реалізація підсистеми взаємодії з великими мовними моделями побудована на принципах модульності та слабкої зв'язності (Low Coupling), що забезпечується використанням патерну проєктування «Абстрактна Фабрика» (Abstract Factory). Такий підхід дозволяє відокремити бізнес-логіку застосунку від специфіки реалізації конкретних API провайдерів (Anthropic та Google). Ключові компоненти архітектури включають:

– інтерфейс AIService - визначає уніфікований контракт методів analyzeVHDL та generateTestbench, гарантуючи типізацію та взаємозамінність різних AI-адаптерів;

– клас AIServiceFactory - відповідає за інстанціювання конкретного класу-сервісу (ClaudeService або GeminiService) на основі ідентифікатора моделі, отриманого від клієнта;

– централізована конфігурація - файл models.ts містить масив об'єктів AIModelConfig, що визначає параметри доступних моделей (ID, назва, провайдер) та дозволяє легко масштабувати систему додаванням нових записів.

Для забезпечення високої якості аналізу застосовано стратегію динамічного конструювання промпту. Функція-генератор createVHDLAnalysisPrompt формує комплексний запит, який поєднує рольову модель, суворі вимоги до формату виводу (JSON Enforcement) та безпечну ін'єкцію коду користувача. Фрагмент реалізації системного промпту наведено у лістингу 3.1, а повний програмний код функції - у додатку А.2.

Лістинг 3.1 – Фрагмент реалізації промпту для аналізу VHDL коду

```
export function createVHDLAnalysisPrompt(vhdlCode: string): string {
  return `Analyze the following VHDL code for errors and issues. Report all issues you can identify with appropriate confidence levels. Critical and high severity issues should be reported if 90%+ confident. Medium and low severity issues should be reported if 70%+ confident.
Provide your analysis in JSON format with the following EXACT structure:
{
  "issuesFound": [
    {
      "description": "specific issue description",
      "lines": [{"start": 21, "end": 21}],
      "category": "syntax|logic|style|efficiency",
      "severity": "critical|high|medium|low",
      "suggestions": ["specific suggestion to fix this issue"]
    }
  ],
  "reasoning": "brief explanation of your analysis"
}
CATEGORY DEFINITIONS (use exactly these):
- "syntax": Compilation errors that prevent code from compiling...
- "logic": Functional errors that cause incorrect behavior...
[... Detailed Analysis Rules & Patterns Omitted for Brevity ...]
VHDL Code:
\`\`\`vhdl
${vhdlCode}
\`\`\`
Respond ONLY with valid JSON. Do not include any text outside the JSON structure.`;
}
```

Як видно з лістингу, система використовує маркерні блоки Markdown для чіткого відокремлення коду користувача від інструкцій, що запобігає атакам типу "Prompt Injection".

Обробка запитів здійснюється через захищений API-маршрут (/api/analyze), який виконує роль контролера. Він реалізує механізм глобальної обробки помилок (Global Error Handling), перехоплюючи виключення від зовнішніх сервісів (тайм-аути, ліміти запитів, помилки валідації) та повертаючи клієнту стандартизовану відповідь із кодом стану та описом проблеми.

4 ТЕСТУВАННЯ СИСТЕМИ

Етап тестування програмного продукту мав верифікувати відповідності розробленої системи сформульованим функціональним та нефункціональним вимогам. Оскільки система базується на взаємодії із зовнішніми недетермінованими сервісами (LLM), особливу увагу було приділено перевірці стабільності роботи (Robustness) та обробці виключних ситуацій. Процес тестування був розділений на три ключові етапи.

4.1 Функціональне тестування наскрізного потоку даних

Головною задачею цього етапу була перевірка коректності проходження даних повним ланцюжком системи (End-to-End flow) згідно з "позитивним сценарієм". Тестування підтвердило цілісність передачі даних між клієнтською частиною, сервером та зовнішніми API.

Алгоритм успішної транзакції, реалізований у функції handleAnalyze (src/app/page.tsx), проходить наступні кроки перевірки:

– ініціалізація запиту - користувач вводить код, натискає кнопку «Analyze», після чого статус застосунку переходить у стан завантаження;

– передача даних - сформоване навантаження (payload) із VHDL-кодом та ідентифікатором моделі надсилається на захищений маршрут /api/analyze методом POST.

– типізація даних - на всіх етапах (від клієнта до серверного контролера) дані валідуються через суворі TypeScript-інтерфейси AnalyzeRequest та AnalyzeResponse, що виключає помилки розбіжності типів;

– отримання результату - після обробки запиту сервісом AI, отриманий JSON-об'єкт десеріалізується, перевіряється на наявність поля success: true та зберігається у глобальному сховищі стану для відображення у компоненті ResultsPanel.

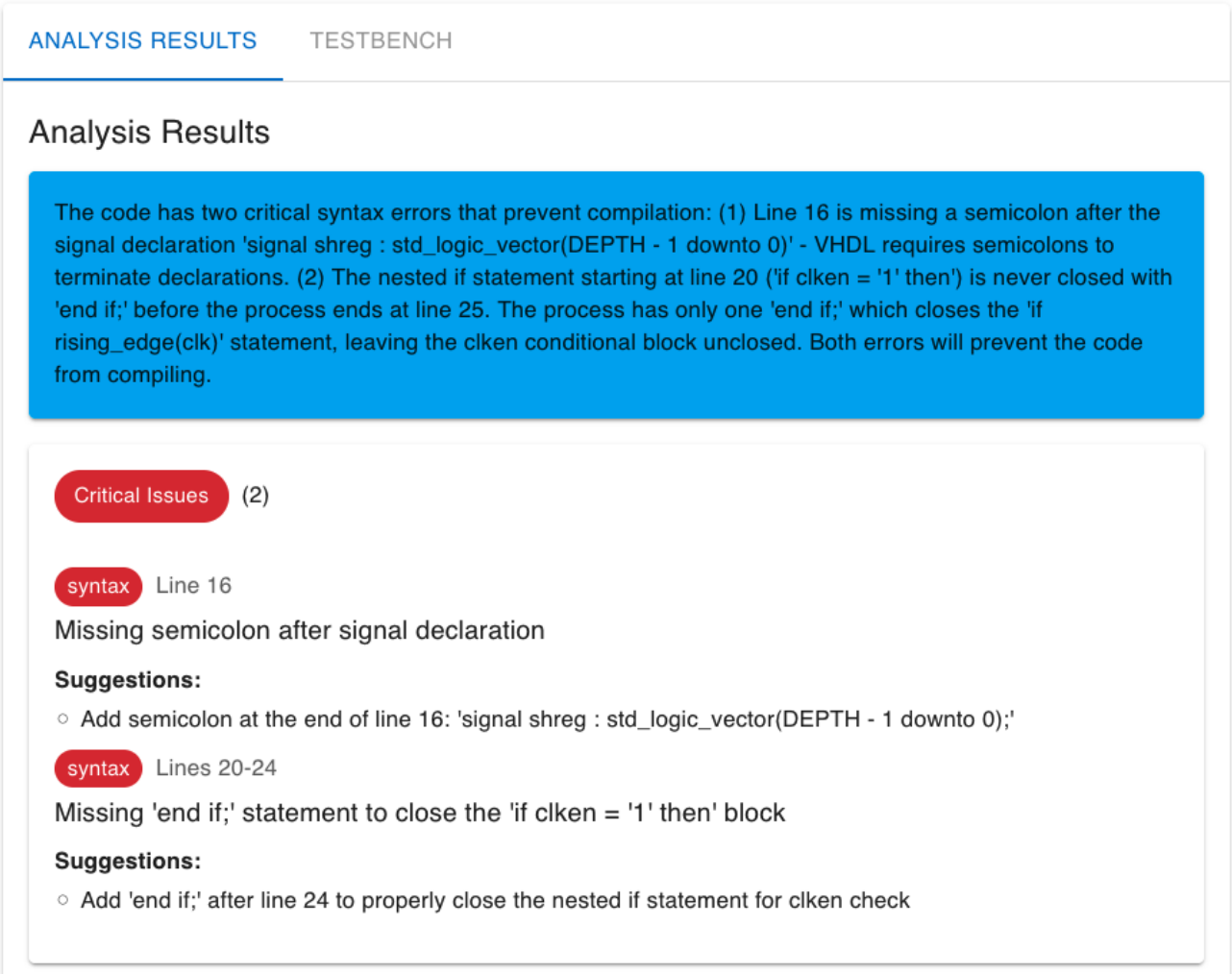


Рисунок 4.1 – Результат аналізу коду з синтаксичною помилкою

Для наочної демонстрації роботи системи було проведено модельний експеримент із виявлення синтаксичної помилки (рис. 4.1). У редактор було введено код сутності reg_multifunc із навмисно пропущеним символом ";" та конструкцією "end if".

Як видно з рисунка, система коректно ідентифікувала проблему, класифікувала її як синтаксичну та надала точні координати помилки, що підтверджує успішне проходження функціонального тесту.

4.2 Тестування надійності та валідація даних

Для забезпечення відмовостійкості системи реалізовано стратегію «глибокого захисту», яка передбачає багаторівневу валідацію даних. Тестування підтвердило коректність спрацювання захисних механізмів на двох рівнях:

– клієнтська валідація - інтерфейс блокує кнопку відправки форми, якщо поле введення коду порожнє, а додаткова перевірка !vhdlCode.trim() в обробнику подій запобігає надсиланню пустих запитів;

– серверна валідація - API-маршрут (route.ts) виконує незалежну перевірку вхідних параметрів і повертає статус 400 Bad Request у випадку відсутності обов'язкових полів code або model, захищаючи систему від маніпуляцій із запитами в обхід інтерфейсу.

Візуалізацію роботи механізму клієнтської валідації представлено на рисунку 4.2. При спробі ініціювати аналіз без введення коду, система блокує дію користувача, запобігаючи відправленню "порожнього" запиту на сервер, що економить ресурси API. Індикатором недоступної функціональності для користувача виступає заблокована ("disabled") кнопка "Analyze code".

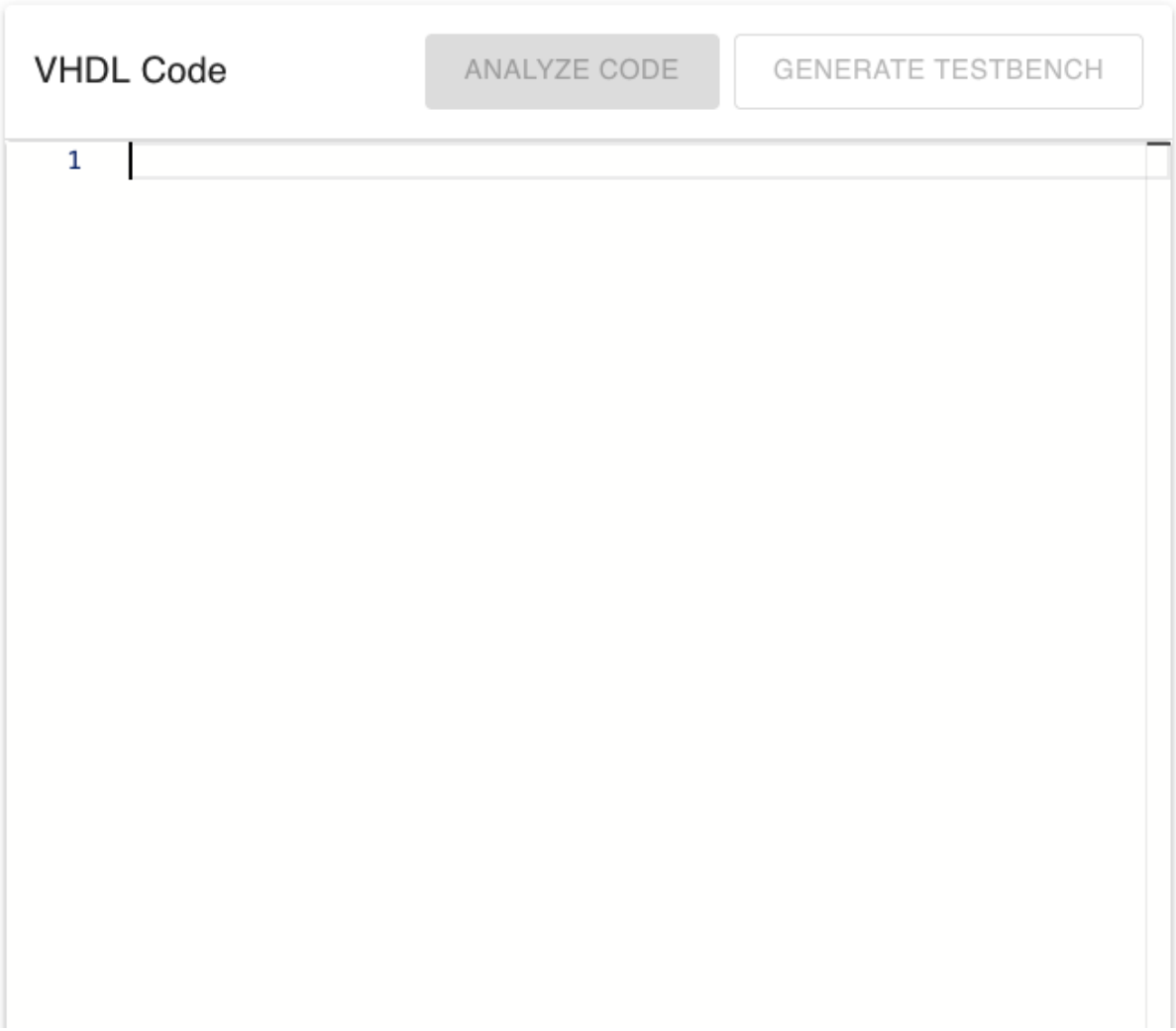


Рисунок 4.2 – Спрацювання валідації при спробі відправки порожнього запиту

Окрім базової валідації вхідних даних, в системі реалізовано механізм логічної валідації залежностей. Доступ до модуля генерації верифікаційного оточення (кнопка "Generate Testbench") надається користувачу виключно після успішного проходження етапу аналізу і лише за умови, що у коді відсутні помилки рівнів Critical або Syntax.

Важливо зазначити що після активації кнопки "Generate Testbench" якщо користувач вирішить модифікувати VHDL код у редакторі, то дана кнопка деактивується знову і користувач буде змушений виконати аналіз коду повторно.

Такий підхід гарантує, що система не витрачатиме ресурси токенів на спроби згенерувати тестбенч для завідомо непрацездатного коду. Активація функціоналу генерації після успішного аналізу продемонстрована на рисунку 4.3.

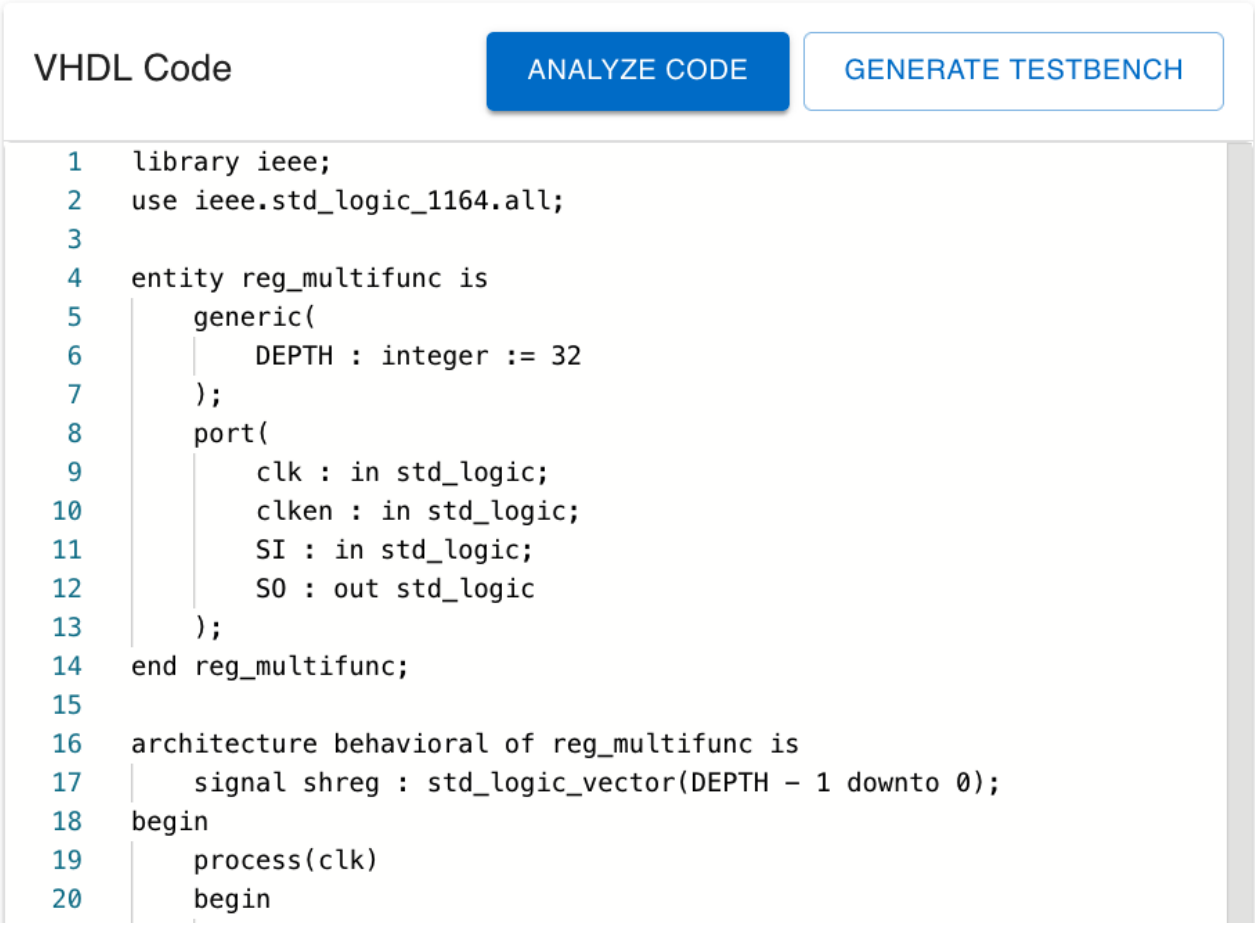


Рисунок 4.3 – Активація функції генерації тестбенчу після успішного аналізу коду

Окрему увагу приділено обробці помилок зовнішніх AI-провайдерів. Реалізовані блоки try-catch та перевірки статусів відповідей гарантують, що у випадку збою API (наприклад, помилка 500 або тайм-аут) або повернення некоректного JSON (hallucinated JSON format), застосунок не припиняє роботу аварійно. Натомість користувачу відображається інформативне повідомлення через механізм setError, реалізований у панелі результатів.

4.3 Тестування інтерфейсу користувача та адаптивності

Тестування UI/UX було спрямоване на перевірку зручності взаємодії та коректності відображення інтерфейсу на різних пристроях.

Перевірка механізмів зворотного зв'язку (User Feedback) показала коректну роботу станів завантаження. Під час виконання асинхронного запиту до LLM, який може тривати від 5 до 15 секунд, система візуально інформує користувача: кнопка дії переходить у стан disabled, змінюється текст на індикатор процесу, а в панелі результатів відображається відповідний Alert-компонент. Це запобігає повторному відправленню форми та покращує досвід користування.

Адаптивність верстки протестовано за допомогою інструментів розробника браузера на емуляторах мобільних та десктопних пристроїв. Використання системи сіток Grid2 бібліотеки Material UI із налаштованими точками розриву (breakpoints xs, md) забезпечує коректне масштабування робочої області: на великих екранах редактор та панель результатів розташовуються поруч, а на мобільних пристроях - трансформуються у вертикальну структуру зі збереженням повної функціональності.

Варто окремо зазначити, що попри реалізовану повну технічну адаптивність під мобільні пристрої, специфіка предметної області накладає певні обмеження на зручність використання. Написання та аналіз VHDL-коду вимагає роботи з великими обсягами тексту, одночасного перегляду вихідного коду та результатів аналізу, що фізично складно розмістити на екрані смартфона без втрати контексту.

Тому, хоча застосунок зберігає повну функціональність на пристроях будь-якого розміру, рекомендованим сценарієм використання залишається робота на десктопних ПК або ноутбуках.

4.4 Розгортання системи та заходи безпеки

Останнім етапом розробки програмного продукту є організація процесу розгортання (deployment) та забезпечення заходів безпеки для захисту API-шлюзів від перевантаження.

Враховуючи обраний технологічний стек (Next.js), як платформу для хостингу було обрано хмарний сервіс Vercel, який забезпечує нативну підтримку фреймворку та інтегрований CI/CD конвеєр. Процес розгортання налаштовано за моделлю GitOps: оновлення коду в репозиторію автоматично ініціює процес збірки та оновлення продуктивного середовища.

Для конфігурації системи використовуються змінні середовища, що дозволяє гнучко керувати налаштуваннями без зміни вихідного коду. У конфігураційному файлі на сервері визначено ключі доступу до зовнішніх сервісів (ANTHROPIC_API_KEY, GOOGLE_API_KEY) та параметри обмеження навантаження (RATE_LIMIT_WINDOW_MS, RATE_LIMIT_MAX_REQUESTS).

Ключовим компонентом архітектури безпеки є система обмеження частоти запитів (Rate Limiting), реалізована на рівні проміжного програмного забезпечення Middleware. Це дозволяє перехоплювати запити до маршрутів /api/analyze та /api/generate-testbench ще до їх обробки основною логікою застосунку.

Реалізований механізм захисту працює за наступним алгоритмом:

– ідентифікація клієнта здійснюється за IP-адресою, отриманою із заголовків x-forwarded-for або x-real-ip;

– облік кількості запитів ведеться в оперативній пам'яті з використанням змінного часового вікна;

– при перевищенні ліміту система миттєво повертає відповідь зі статусом 429 (Too Many Requests) та заголовком Retry-After;

– для інформування клієнта додаються стандартизовані заголовки X-RateLimit-Limit, X-RateLimit-Remaining та X-RateLimit-Reset.

Поточна реалізація є оптимальною для односерверного розгортання, проте архітектура модуля передбачає можливість масштабування до використання Redis (наприклад, Upstash) у разі переходу на кластерну інфраструктуру.

4.5 Підсумки тестування

У третьому розділі виконано програмну реалізацію та тестування консультативної системи для верифікації VHDL-коду. У ході роботи було отримано наступні результати:

– розроблено клієнт-серверну архітектуру вебзастосунку на базі Next.js, що забезпечує гнучку взаємодію з різними моделями штучного інтелекту через патерн "Абстрактна Фабрика";

– реалізовано ключові функціональні модулі: модуль статичного аналізу коду з підтримкою потокового відображення помилок та модуль генерації верифікаційного оточення;

– впроваджено комплексний підхід до промпт-інженірингу, який включає використання рольових моделей, ін'єкцію контексту та сувору валідацію JSON-структури відповідей;

– проведено функціональне тестування, яке підтвердило коректність роботи алгоритмів валідації даних, стабільність обробки помилок від зовнішніх API та адаптивність інтерфейсу користувача;

– налаштовано процес автоматизованого розгортання на платформі Vercel із впровадженням Middleware для захисту API від зловживань та перевантажень.

Створена система повністю відповідає вимогам технічного завдання, демонструє високу швидкодію та готова до впровадження у навчальний процес.

5 ПРАКТИЧНІ РЕКОМЕНДАЦІЇ ЩОДО ВИКОРИСТАННЯ РОЗРОБЛЕНОЇ СИСТЕМИ

У даному розділі наведено вимоги до технічного забезпечення робочого місця користувача, сформульовано методику ефективного використання системи

для верифікації VHDL-коду та окреслено перспективи подальшого розвитку програмного продукту.

5.1 Вимоги до технічного забезпечення та кваліфікації користувача

Розроблена система функціонує як вебзастосунок, що значно знижує вимоги до апаратної частини клієнтського пристрою, оскільки основні обчислювальні навантаження виконуються на хмарних серверах AI-провайдерів.

Для забезпечення коректної роботи з консультативною системою, робоче місце користувача повинно відповідати наступним технічним вимогам:

– наявність персонального комп'ютера або ноутбука з діагоналлю екрана не менше 13 дюймів для комфортної роботи з редактором коду;

– наявність стабільного інтернет-з'єднання з пропускною здатністю не менше 5 Мбіт/с для забезпечення швидкого обміну даними з API;

– встановлений сучасний веббраузер із підтримкою стандарту ES6+ (Google Chrome, Mozilla Firefox, Microsoft Edge або Safari).

Система не вимагає встановлення локального спеціалізованого програмного забезпечення, компіляторів або інтерпретаторів мов програмування на комп'ютер користувача. Для ефективного використання інструменту користувач повинен володіти певним рівнем кваліфікації:

– базове розуміння синтаксису та семантики мови опису апаратури VHDL;

– знання принципів цифрової схемотехніки для коректної інтерпретації знайдених логічних помилок;

– вміння формулювати технічні завдання англійською мовою для ефективної роботи з модулем генерації тестів.

5.2 Методика проведення верифікації коду

Для досягнення максимальної ефективності аналізу та мінімізації витрат на використання токенів, рекомендується дотримуватись визначеного алгоритму взаємодії з системою. Процес верифікації поділяється на етапи залежно від стадії готовності проєкту та задачі перевірки.

Рекомендації щодо вибору AI моделі залежно від сценарію використання наведені у таблиці 5.1.

Таблиця 5.1 – Рекомендації щодо вибору моделі для аналізу

Сценарій використання	**Рекомендована модель**	**Обґрунтування вибору**
Поточна перевірка під час написання коду	Claude Haiku 4.5	Найвища швидкість реакції та мінімальна вартість при достатній точності виявлення синтаксичних помилок
Пошук складних логічних помилок та оптимізація	Claude Sonnet 4.5	Збалансований варіант, що надає детальні пояснення та рекомендації щодо стилю коду
Фінальна верифікація перед синтезом	Claude Opus 4.1	Максимальна точність та відсутність хибних спрацювань, що критично важливо перед етапом імплементації
Робота в умовах обмеженого бюджету	Gemini 2.5 Flash	Доступність безкоштовного рівня використання API для навчальних цілей
Поглиблений аналіз в умовах обмеженого бюджету	Gemini 2.5 Pro	Оптимальний вибір для складних студентських проєктів, що забезпечує глибокий аналіз логіки без значних фінансових витрат

Користувачам рекомендується критично ставитися до результатів аналізу, особливо щодо повідомлень рівня «Low», які часто мають рекомендаційний характер і можуть не враховувати специфічні обмеження конкретного апаратного забезпечення.

Окремої уваги потребує аналіз логічних попереджень системи. Важливо

розуміти, що AI моделі базуються на найбільш поширених практиках кодування, тому вони схильні класифікувати будь-які нестандартні інженерні рішення як дефекти.

Тобто, система може помилково сприйняти свідомий архітектурний задум розробника за помилку. Тому користувач повинен критично оцінювати зауваження, особливо у випадках реалізації специфічної апаратної поведінки:

– реалізація нестандартної асинхронної логіки або навмисне використання засувок (latches) для специфічних задач;

– створення схем із навмисними часовими перегонами (race conditions), наприклад, у генераторах випадкових чисел;

– використання оптимізованих, але неочевидних для моделі математичних перетворень на рівні бітових операцій.

У таких ситуаціях користувач виступає фінальним арбітром, оскільки AI модель може хибно класифікувати свідоме інженерне рішення як логічну помилку через нерозуміння специфічного контексту роботи пристрою.

5.3 Рекомендації щодо автоматичної генерації тестбенчів

Модуль генерації верифікаційного оточення працює за принципом "text-to-code". Якість та повнота згенерованого VHDL-коду тесту (testbench) прямо залежить від того, наскільки зрозуміло користувач сформулює завдання у полі "Test Scenario Description".

Для студентів та інженерів-початківців, які ще не мають значного досвіду у верифікації, рекомендується використовувати структуру мінімально достатнього промпту. Цей підхід дозволяє отримати робочий тест без необхідності писати довгі специфікації.

Розглянемо еталонний приклад мінімального запиту для тестування мультиплексора 4-в-1 який наведено у лістингу 5.1.

Лістинг 5.1 – Приклад мінімального запиту для генерації тестового оточення

```
Verify the 4-to-1 multiplexer logic. Cycle through all
combinations of select signals S0 and S1 (00, 10, 01, 11) to ensure
inputs A, B, C, and D are correctly routed to output Z. Test with
changing values on data inputs.
```

Щоб система гарантовано зрозуміла завдання, промпт повинен містити три обов'язкові складові:

– об'єкт тестування та завдання - це вступне речення, яке задає загальний контекст генерації та пояснює штучному інтелекту сутність перевірки;

– сценарій впливів - це опис алгоритму зміни вхідних сигналів, який має містити конкретні імена портів із VHDL-сутності;

– критерій успіху - це пояснення очікуваного стану виходів схеми, що дозволяє згенерувати код для автоматичної звірки результатів.

Детальний опис кожної складової з прикладами та порадами наведено нижче.

Об’єкт тестування та завдання - ця частина пояснює AI моделі, що саме ми перевіряємо. У наведеному прикладі це фраза: "Verify the 4-to-1 multiplexer logic". Для кращого розуміння завдання рекомендується використовувати дієслова дії, такі як "Verify", "Test", "Check", та вказувати повну назву вузла, що тестується.

Сценарій впливів - ця частина описує, як повинні змінюватися вхідні сигнали у часі. У прикладі це фрагмент: "Cycle through all combinations of select signals S0 and S1 (00, 10, 01, 11)... Test with changing values on data inputs". Для початківців критично важливо вказувати конкретні імена сигналів так, як вони названі в entity. Також перерахування конкретних значень (наприклад, "00, 01, 10, 11") працює ефективніше, ніж абстрактні фрази типу "test all cases", оскільки це усуває неоднозначність.

Критерій успіху - ця частина пояснює, який результат ми очікуємо побачити на виході. У прикладі це фраза: "...to ensure inputs A, B, C, and D are correctly

routed to output Z". Чітке визначення залежності "Вхід -> Вихід" (наприклад, "Output Y should equal Input A when Sel is '0'") дозволяє моделі згенерувати автоматичні перевірки, які самостійно повідомлять про помилку в симуляторі.

Використання такої трикомпонентної структури дозволяє навіть користувачам з мінімальним досвідом отримувати коректні VHDL-тестбенчі.

Для закріплення розуміння принципу побудови запитів, наведемо конкретні приклади мінімальних промптів для типових навчальних задач цифрової схемотехніки.

У лістингу 5.2 наведено приклад запиту для перевірки комбінаційної логіки на прикладі дешифратора.

Лістинг 5.2 – Промпт для верифікації дешифратора 2-в-4

```
Verify 2-to-4 Decoder logic; drive input vector 'Sel' through values 00 to 11 and check if the corresponding bit in output 'Y' becomes active.
```

Для верифікації послідовнісних схем, де стан виходів залежить від історії вхідних впливів, запит має бути детальнішим. Приклад наведено у лістингу 5.3.

Лістинг 5.3 – Промпт для верифікації FSM

```
Test Sequence Detector FSM for pattern '101'; reset the circuit for 20ns, then feed serial sequence '0110101' to 'D_in' port and verify that 'Found' flag goes high correctly.
```

Приклад формування запиту для тестування вузлів із синхронним керуванням та завантаженням даних наведено у лістингу 5.4.

Лістинг 5.4 – Промпт для верифікації синхронного лічильника

```
Verify 8-bit Up-Counter with Load enable; apply Reset, then assert 'Load' with value '10', release Load and check if output increments on every rising edge of 'Clk'.
```

Наведені лістинги демонструють, як, змінюючи лише назви сигналів та опис логіки, можна адаптувати універсальний шаблон під будь-який цифровий компонент.

5.4 Перспективи розвитку та вдосконалення системи

Сфера штучного інтелекту перебуває у стані перманентної технологічної еволюції, де цикл оновлення фундаментальних моделей скоротився до кількох місяців. Така динаміка призводить до того, що емпіричні дані щодо ефективності конкретних інструментів можуть втрачати актуальність ще до моменту завершення розробки програмного продукту.

Яскравим прикладом цього процесу є вихід на ринок моделі Claude Opus 4.5, яка, згідно з офіційним релізом розробників, демонструє значно вищі показники у задачах складного логічного міркування та кодування порівняно з попереднім поколінням [20]. При цьому аналіз оновленої цінової політики вказує на підвищення економічної ефективності використання флагманських моделей: співвідношення "ціна/якість" для Opus 4.5 дозволяє отримати еталонну точність верифікації зі значно меншими витратами, ніж це було можливо з Opus 4.1 [21].

Паралельно компанія Google представила модель Gemini 3, яка позиціюється як найбільш інтелектуальне рішення платформи, здатне обробляти надвеликі контексти з високою швидкістю [22]. Інтеграція цих рішень у розроблену систему дозволить суттєво зменшити відсоток «галюцинацій» при генерації складних тестбенчів без необхідності внесення змін у програмну архітектуру застосунку.

Окремим стратегічним вектором розвитку є імплементація відкритих моделей (Open Source), таких як моделі сімейств DeepSeek або Qwen. Використання локальних версій цих моделей дозволяє мінімізувати фінансові

витрати на токени до вартості оренди сервера та гарантує повну конфіденційність коду, що є критичним фактором для комерційних розробок.

Окрім оновлення генеративного ядра, існують широкі можливості для функціонального вдосконалення самої системи:

– впровадження технології RAG (Retrieval-Augmented Generation) дозволить системі аналізувати код з урахуванням завантаженої технічної документації та специфікацій конкретного FPGA-кристала;

– розробка спеціалізованих плагінів для середовищ розробки (наприклад, VS Code Extension) забезпечить безшовну інтеграцію асистента безпосередньо у робочий процес інженера;

– створення механізму зворотного зв'язку дозволить накопичувати базу вдалих рішень для подальшого донавчання (Fine-tuning) спеціалізованої моделі під вузькі задачі проєктування.

Реалізація цих кроків дозволить трансформувати розроблений прототип у повноцінну екосистему для автоматизованої верифікації апаратних описів.

ВИСНОВКИ

У магістерській кваліфікаційній роботі вирішено актуальну науково-прикладну задачу підвищення ефективності та доступності процесу верифікації VHDL-коду шляхом розробки спеціалізованої консультативної системи на базі технологій штучного інтелекту. Отримані результати підтверджують виконання поставленого технічного завдання та досягнення мети роботи, що засвідчують наступні висновки:

– на основі аналізу предметної області встановлено, що традиційні засоби верифікації FPGA-проєктів мають високий поріг входження, тому обґрунтовано доцільність інтеграції великих мовних моделей як інтелектуальних асистентів, здатних забезпечити пояснення помилок та автоматизацію рутинних процесів;

– у ході проведення експериментального дослідження 9 сучасних AI-моделей визначено, що модель Claude Haiku 4.5 є оптимальним вибором для режиму інтерактивного асистента, забезпечуючи найвищу точність виявлення складних логічних помилок із показником F1-Score 89,33% при середньому часі реакції 6,40 с, тоді як модель Claude Opus 4.1 продемонструвала еталонну надійність із повною відсутністю хибних спрацювань;

– спроєктовано та програмно реалізовано гнучку клієнт-серверну архітектуру системи на базі стека технологій Next.js та TypeScript із використанням патерну проєктування "Абстрактна Фабрика", що дозволило забезпечити уніфіковану взаємодію з різними провайдерами штучного інтелекту (Anthropic, Google) та легку масштабованість програмного комплексу;

– розроблено функціональний модуль статичного аналізу коду, який завдяки застосуванню стратегій промпт-інженірингу та суворій валідації JSON-структури забезпечує детальну класифікацію синтаксичних, логічних та стилістичних помилок із наданням рекомендацій щодо їх виправлення;

– реалізовано підсистему автоматичної генерації верифікаційного оточення, яка дозволяє створювати синтаксично коректні VHDL-тестбенчі на основі

текстового опису сценаріїв тестування природною мовою, що значно спрощує процес перевірки працездатності розроблених цифрових вузлів;

– підтверджено надійність та безпеку розробленої системи шляхом впровадження багаторівневої валідації вхідних даних, захисту від ін'єкцій у промпти та реалізації механізму обмеження частоти запитів.

Наукова новизна одержаних результатів полягає у розробці методики автоматизованого порівняльного аналізу ефективності великих мовних моделей для задач верифікації апаратної логіки, яка, на відміну від існуючих підходів, застосовує семантичну нормалізацію відповідей за допомогою моделі-арбітра. Це дозволило вперше систематизувати кількісні показники точності та надійності сучасних LLM при роботі з VHDL-кодом та експериментально обґрунтувати доцільність використання гібридного підходу в консультативних системах.

Практична цінність отриманих результатів полягає у створенні повнофункціонального інструменту, який дозволяє суттєво скоротити часові витрати на налагодження VHDL-коду та рекомендований до впровадження у навчальний процес підготовки фахівців з комп'ютерної інженерії.

Таким чином, результати проведеного комплексного тестування підтверджують працездатність та ефективність запропонованих рішень, що доводить виконання всіх поставлених завдань та досягнення мети роботи у повному обсязі.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1 Foster H. The 2022 Wilson Research Group Functional Verification Study / Harry Foster. – Siemens Digital Industries Software, 2022. URL: https://blogs.sw.siemens.com/verificationhorizons/2022/10/24/part-2-the-2022-wilson-research-group-functional-verification-study/ (date of access: 23.11.2025);

2 Vaswani A. et al. Attention Is All You Need / A. Vaswani, N. Shazeer, N. Parmar et al. URL: https://arxiv.org/abs/1706.03762 (date of access: 24.11.2025);

3 White J. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT / J. White, Q. Fu, S. Hays et al. URL: https://arxiv.org/abs/2302.11382 (date of access: 24.11.2025);

4 The React Framework for the Web. URL: https://nextjs.org (date of access: 14.10.2025);

5 Benefits of Implementing TypeScript. URL: https://strapi.io/blog/benefits-of-typescript (date of access: 14.10.2025);

6 Material UI – Overview. URL: https://mui.com/material-ui/getting-started (date of access: 14.10.2025);

7 Zustand vs Redux: A Comprehensive Comparison. URL: https://betterstack.com/community/guides/scaling-nodejs/zustand-vs-redux (date of access: 14.10.2025);

8 Toward Total Recall: Enhancing FAIRness through AI-Driven Metadata Standardization. URL: https://arxiv.org/html/2504.05307v1 (date of access: 14.10.2025).

9 OpenAI's Next Generation: GPT-5 Architecture and Capabilities. URL: https://openai.com/index/introducing-gpt-5 (date of access: 28.10.2025).

10 Introducing the Claude 4.1 Family: Opus and Sonnet. URL: https://www.anthropic.com/news/claude-opus-4-1 (date of access: 28.10.2025).

11 Claude 4.5 Haiku: The Fastest Model in its Class. URL: https://www.anthropic.com/news/claude-haiku-4-5 (date of access: 28.10.2025).

12 Google AI Unveils Gemini 2.5: Pro and Flash. URL: https://cloud.google.com/blog/products/ai-machine-learning/gemini-2-5-flash-lite-flash-pro-ga-vertex-ai (date of access: 28.10.2025).

13 Gemini 2.5 Flash Lite: High-Efficiency AI for Mobile. URL: https://ai.google.dev/gemini-api/docs/models#gemini-2.5-flash-lite (date of access: 28.10.2025).

14 Touvron, H. et al. (Meta AI). "Llama 2: Open Foundation and Fine-Tuned Chat Models". arXiv, 2023 URL: https://arxiv.org/abs/2307.09288 (date of access: 28.11.2025);

15 Common Crawl Foundation. "Common Crawl Statistics". URL: https://commoncrawl.github.io/cc-crawl-statistics/ (date of access: 28.11.2025);

16 Wendler, C. et al. "Do Large Language Models Have an English 'Accent'? Evaluating and Improving the Naturalness of Multilingual LLMs". ACL Anthology, 2025. URL: https://aclanthology.org/2025.acl-long.193/ (date of access: 28.11.2025);

17 Boopathi, K. "Why Your LLM Costs 5X More If You Don't Speak English". 2025. URL: https://programmerraja.is-a.dev/post/2025/Why-Your-LLM-Costs-5X-More-If-You-Don't-Speak-English (date of access: 28.11.2025);

18 Ahia, O. et al. "Do All Languages Cost the Same? Tokenization in the Era of Commercial Language Models". arXiv, 2023.. URL: https://arxiv.org/abs/2305.13707 (date of access: 28.11.2025);

19 Wang, Y. et al. "MMLU-ProX: A Multilingual Benchmark for Advanced Large Language Model Evaluation". arXiv, 2025. URL: https://arxiv.org/abs/2503.10497 (date of access: 28.11.2025);

20 Introducing Claude Opus 4.5. Anthropic. URL: https://www.anthropic.com/news/claude-opus-4-5 (date of access: 30.11.2025);

21 Claude Docs. Model Pricing. URL: https://platform.claude.com/docs/en/about-claude/pricing (date of access: 30.11.2025);

22 Introducing Gemini 3: our most intelligent model that helps you bring any idea to life. Blog Google. URL: https://blog.google/products/gemini/gemini-3/#gemini-3 (date of access: 30.11.2025);

ДОДАТОК А
ЛІСТИНГИ ФРАГМЕНТІВ КОДУ

Лістинг А.1 – Функція формування промпту для генерації тестового оточення

```
export function createTestbenchGenerationPrompt(
  vhdlCode: string,
  scenario: string,
  clockPeriod?: string,
  simulationTime?: string
): string {
  const clockInfo = clockPeriod ? `Clock Period: ${clockPeriod}` : 'Clock Period: 10 ns (default)';
  const simInfo = simulationTime ? `Simulation Time: ${simulationTime}` : 'Simulation Time: 1000 ns (default)';

  return `Generate a complete, synthesizable VHDL testbench for the provided design. The testbench must be production-ready and follow industry best practices.

DESIGN CODE:
\`\`\`vhdl
${vhdlCode}
\`\`\`

TEST SCENARIO:
${scenario}

TESTBENCH PARAMETERS:
- ${clockInfo}
- ${simInfo}

TESTBENCH REQUIREMENTS:
```

```
1. **Entity Declaration**: Create testbench entity with no ports
2. **Component Declaration**: Declare the DUT (Design Under Test) as a component
3. **Signal Declarations**: Create all necessary signals matching DUT ports
4. **Clock Generation**: If DUT has clock input, generate clock with specified period
5. **Reset Logic**: If DUT has reset, apply proper reset sequence at start
6. **Stimulus Process**: Implement test scenarios as described by user
7. **Assertions**: Include basic checks for expected outputs where applicable
8. **File Headers**: Add proper comments explaining testbench purpose

CODING STANDARDS:
- Use IEEE standard libraries (ieee.std_logic_1164, ieee.numeric_std)
- Follow consistent naming: tb_<entity_name> for testbench entity
- Use wait statements properly for timing control
- Add comments explaining each test phase
- Use report statements for simulation feedback
- Ensure proper signal initialization

OUTPUT FORMAT:
Return ONLY the complete VHDL testbench code. Do not include explanations, JSON wrappers, or markdown formatting. Start directly with the library declarations.`;
}
```

Лістинг А.2 – Функція формування промпту для аналізу VHDL коду

```
export function createVHDLAnalysisPrompt(vhdlCode: string): string {
```

return `Analyze the following VHDL code for errors and issues. Report all issues you can identify with appropriate confidence levels. Critical and high severity issues should be reported if 90%+ confident. Medium and low severity issues should be reported if 70%+ confident.

Provide your analysis in JSON format with the following EXACT structure:

```
{
  "issuesFound": [
    {
      "description": "specific issue description",
      "lines": [{"start": 21, "end": 21}],
      "category": "syntax|logic|style|efficiency",
      "severity": "critical|high|medium|low",
      "suggestions": ["specific suggestion to fix this issue"]
    }
  ],
  "reasoning": "brief explanation of your analysis"
}
```

CATEGORY DEFINITIONS (use exactly these):

- "syntax": Compilation errors that prevent code from compiling (missing semicolons, parentheses, undeclared identifiers, type mismatches)

- "logic": Functional errors that cause incorrect behavior (wrong assignments, missing assignments, incorrect logic expressions, race conditions, unreachable states)

- "style": Code style violations following VHDL conventions. MUST check for: (1) Inconsistent capitalization (library names like "IEEE" vs "ieee", signal names like "CLK" vs "clk", package names like "STD_LOGIC_1164" vs "std_logic_1164"), (2) Missing whitespaces (no spaces around assignment operators like "<=", no spaces in sensitivity lists like "process(x0,x1)" vs "process(x0, x1)"), (3) Old-style constructs (CLK'event instead of

rising_edge/falling_edge), (4) Naming convention violations (intentional uppercase acronyms such as "SI"/"SO" alongside lowercase names are acceptable and should not be flagged)

- "efficiency": Suboptimal implementations that waste resources (redundant logic, unnecessary processes for combinational logic, inefficient algorithms)

EFFICIENCY - CONCRETE PATTERNS TO CHECK:

1. Process with only combinational assignments → suggest concurrent assignment or with-select statement

Example: process(sel, a, b) with only if-elsif-else assigning output should use "output <= a when sel='0' else b;"

2. For loop with if checking loop variable = constant → direct indexed access

Example: "for i in 0 to counter loop if i = counter then array(i) <= value; end if; end loop;" should be "array(counter) <= value;"

3. Else clause after exhaustive if-elsif covering all bit combinations

Example: 2-bit select has 4 cases (00,01,10,11); if all covered by if-elsif, else is unreachable redundant code

4. While loop with manual counter for fixed iterations → use for loop

Example: "i := 0; while i < 8 loop ... i := i + 1; end loop;" should be "for i in 0 to 7 loop ... end loop;"

For loops are more synthesizer-friendly and eliminate manual counter management

5. Loop to initialize/reset array → use aggregate assignment

Example: "for i in array'range loop array(i) := 0; end loop;" should be "array := (others => 0);"

6. Multiple intermediate signals for simple expressions that could be computed directly

7. Unnecessary type conversions in a chain

LOGIC - SEMANTIC REASONING (check naming vs behavior):

1. Signal names ending in "_up", "_inc", "counter_up" suggest increment → should use + operator

2. Signal names ending in "_down", "_dec" suggest decrement → should use - operator

3. Generic name "counter" without qualifier typically means up-counter → should increment

4. Entity/signal name contradicts actual operation (e.g., "comparator" but performs arithmetic)

SEVERITY GUIDELINES (use these exact mappings):

- "critical": Code will NOT compile or will cause immediate simulation errors. Examples: missing semicolons, undeclared identifiers, syntax errors that prevent parsing

- "high": Code compiles but has functional errors causing incorrect behavior. Examples: wrong logic expressions, missing assignments to output ports, duplicate assignments to same signal, unassigned outputs

- "medium": Code works correctly but has style violations or moderate efficiency issues. Examples: old-style constructs (CLK'event), redundant operations, style violations (missing spaces around operators, missing spaces in lists, inconsistent capitalization of library/package/signal names)

- "low": Minor style issues with minimal impact. Examples: spacing inconsistencies, minor naming variations

IMPORTANT: Multiple assignments to the same signal = "high" (not critical, code may compile but behave incorrectly). Unassigned output ports = "high" (not critical, code compiles but output is undefined).

LINE NUMBER ACCURACY REQUIREMENTS:

1. Line numbers are 1-indexed (first line of VHDL code is line 1)

2. Count ALL lines including blank lines in the code block

3. For multi-line issues, use the exact start and end line numbers

4. For single-line issues, start and end should be the same number

5. Before reporting, verify line numbers by counting from the first line of the VHDL code block

6. Double-check line numbers match the actual code structure

ISSUE DETECTION CHECKLIST (verify these systematically):

1. Output ports: Check every "out" port has at least one assignment statement

2. Signal assignments: Check for duplicate assignments to the same signal in the same scope

3. Syntax: Check for missing semicolons, parentheses, proper statement termination

4. Declarations: Verify all signals/variables used are declared

5. Style: Check capitalization, whitespaces, and old-style constructs (see STYLE ISSUE DETECTION section below)

6. Logic: Verify assignments make logical sense (no obvious contradictions)

ISSUE GROUPING RULES (IMPORTANT - applies to ALL categories):

- ALWAYS group similar issues of the same type together into a single issue entry with multiple line references

- This rule applies to ALL categories: syntax, logic, style, and efficiency

- Examples of issues that should be grouped (across all categories):

  * Syntax: Multiple missing semicolons → one issue with all affected lines

  * Logic: Multiple unassigned output ports → one issue with all affected lines

  * Logic: Multiple duplicate assignments to different signals → separate issues (different signals), but if same signal has multiple duplicate assignments → one issue

  * Style: Multiple missing spaces of the same type (e.g., all missing spaces before "<=" operators) → one issue with all affected lines

* Style: Multiple capitalization inconsistencies of the same type (e.g., all uppercase library names) → one issue with all affected lines

* Efficiency: Multiple redundant operations of the same type → one issue with all affected lines

- Use the "lines" array to include all line numbers where the same issue occurs

- For consecutive lines with the same issue, use a single range object: {"start": 41, "end": 45} instead of multiple individual entries

- For non-consecutive lines, use separate entries: [{"start": 10, "end": 10}, {"start": 25, "end": 25}]

- Only create separate issues if they are fundamentally different problems (e.g., missing spaces vs capitalization vs old-style constructs, or syntax errors vs logic errors)

- Examples:

* If lines 41, 42, 43, 44, 45 (consecutive) all have "missing space before <=", report ONE issue with lines: [{"start": 41, "end": 45}]

* If lines 10, 15, 20 (non-consecutive) all have "missing space before <=", report ONE issue with lines: [{"start": 10, "end": 10}, {"start": 15, "end": 15}, {"start": 20, "end": 20}]

* If lines 5-8 and 12-15 (two consecutive ranges) have the same issue, report ONE issue with lines: [{"start": 5, "end": 8}, {"start": 12, "end": 15}]

ISSUE REPORTING GUIDELINES:

1. Report clear compilation errors, functional bugs, and efficiency issues with appropriate confidence

2. For semantic logic issues (e.g., counter_up using - instead of +), explain the naming-behavior mismatch

3. Report efficiency issues if they have measurable synthesis/resource impact (redundant logic, unnecessary processes)

4. For style issues, only report inconsistencies within the same file or old-style constructs

5. Do NOT report issues that require simulation to verify (timing, race conditions without clear evidence)

6. Do NOT report valid style choices (uppercase keywords, uppercase port names, spacing variations if consistent)

7. If code appears correct with no identifiable issues, return empty array: "issuesFound": []

STYLE ISSUE DETECTION (MANDATORY CHECKS - verify ALL of these):

1. CAPITALIZATION CONSISTENCY (ONLY report if INCONSISTENT within same file):

- Library names: If code mixes "library IEEE;" and "library ieee;" in same file → report inconsistency
- Package names: If code mixes "IEEE.STD_LOGIC_1164" and "ieee.std_logic_1164" in same file → report inconsistency
- Signal/port names: If code mixes "CLK" and "clk" for similar signals in same file → report inconsistency
- DO NOT report if file consistently uses one style (all uppercase keywords OR all lowercase is both valid)
- Report as "style" category, "medium" severity ONLY if mixed/inconsistent within same file

2. WHITESPACE REQUIREMENTS:

- Sensitivity lists: Check for missing spaces after commas (e.g., "process(a,b,c)" should be "process(a, b, c)")
- Report as "style" category, "medium" severity if whitespaces are missing in comma-separated lists

3. OLD-STYLE CONSTRUCTS:

- Clock edge detection: Check for "CLK'event and CLK='0'" or "CLK'event and CLK='1'" instead of "falling_edge(CLK)" or "rising_edge(CLK)"
- Report as "style" category, "medium" severity for old-style clock edge detection

STYLE - DO NOT REPORT THESE (both are valid VHDL styles):

- Consistent uppercase keywords (ENTITY, PORT, END) - this is a valid style choice

- Consistent uppercase port/signal names - this is IEEE convention, perfectly acceptable

- Space before "<=" operator - both "signal<=value" and "signal <= value" are acceptable

- Spaces around logical operators (and, or, not) - spacing variations are acceptable

- Consistent all-uppercase or all-lowercase library names (if not mixed in same file)

AMBIGUOUS CASES (DO NOT REPORT):

- Uppercase acronyms intentionally preserved for ports or signals (e.g., "SI", "SO") mixed with lowercase names

- Spacing around colons in port declarations (e.g., "a:in" vs "a : in")

STRUCTURED VALIDATION (before finalizing response, verify):

1. Each issue category matches the issue type exactly (syntax/logic/style/efficiency)

2. Line numbers are accurate and correspond to actual code lines

3. Severity level matches the impact (critical = won't compile, high = functional error, medium = style/efficiency, low = minor style)

4. Similar issues are grouped together (e.g., all missing spaces in sensitivity lists in one issue with multiple line references)

5. Each reported issue can be verified by reading the code at the specified line numbers

6. For efficiency issues: Have you checked for redundant else, unnecessary process, inefficient loops, redundant logic?

7. For logic issues: Have you checked naming vs behavior (counter_up should increment), semantic correctness?

8. For style issues: Have you checked ONLY for inconsistencies within same file and old-style constructs?

```
9. If no clear issues exist, issuesFound array is empty

BALANCED REPORTING GUIDELINES:
- Report all identifiable issues with appropriate confidence levels
- Critical/high severity: Report if 90%+ confident (compilation/functional errors)
- Medium/low severity: Report if 70%+ confident (efficiency/style issues)
- For semantic issues (naming vs behavior), explain reasoning clearly
- For efficiency issues, explain why the alternative is better (resource usage, synthesis impact)
- Avoid reporting style preferences that are both valid in VHDL standards

For each issue:
- Use precise, verified line numbers (count carefully from line 1)
- Choose category that matches the issue type exactly
- Assign severity using the exact guidelines above
- Provide clear, specific description of what is wrong
- Include actionable suggestions to fix the issue

VHDL Code:
\`\`\`vhdl
${vhdlCode}
\`\`\`

Respond ONLY with valid JSON. Do not include any text outside the JSON structure.`;
}
```

ДОДАТОК Б
СЛАЙДИ ПРЕЗЕНТАЦІЇ

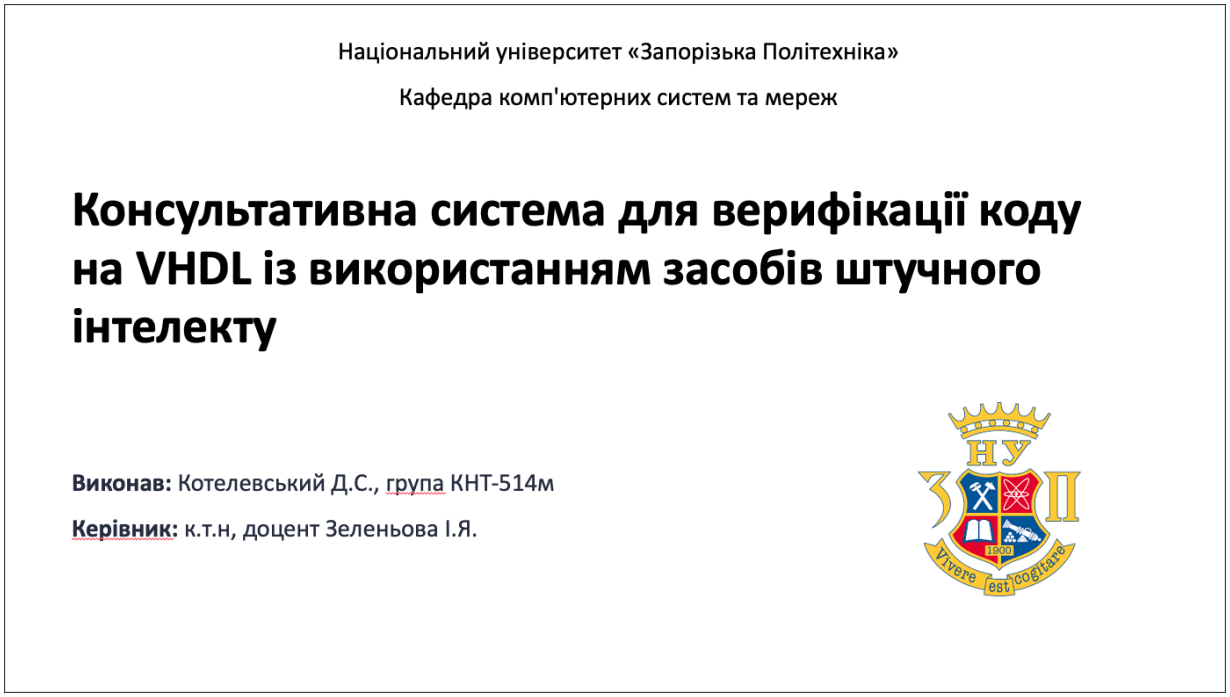


Рисунок Б.1 – Слайд презентації 1

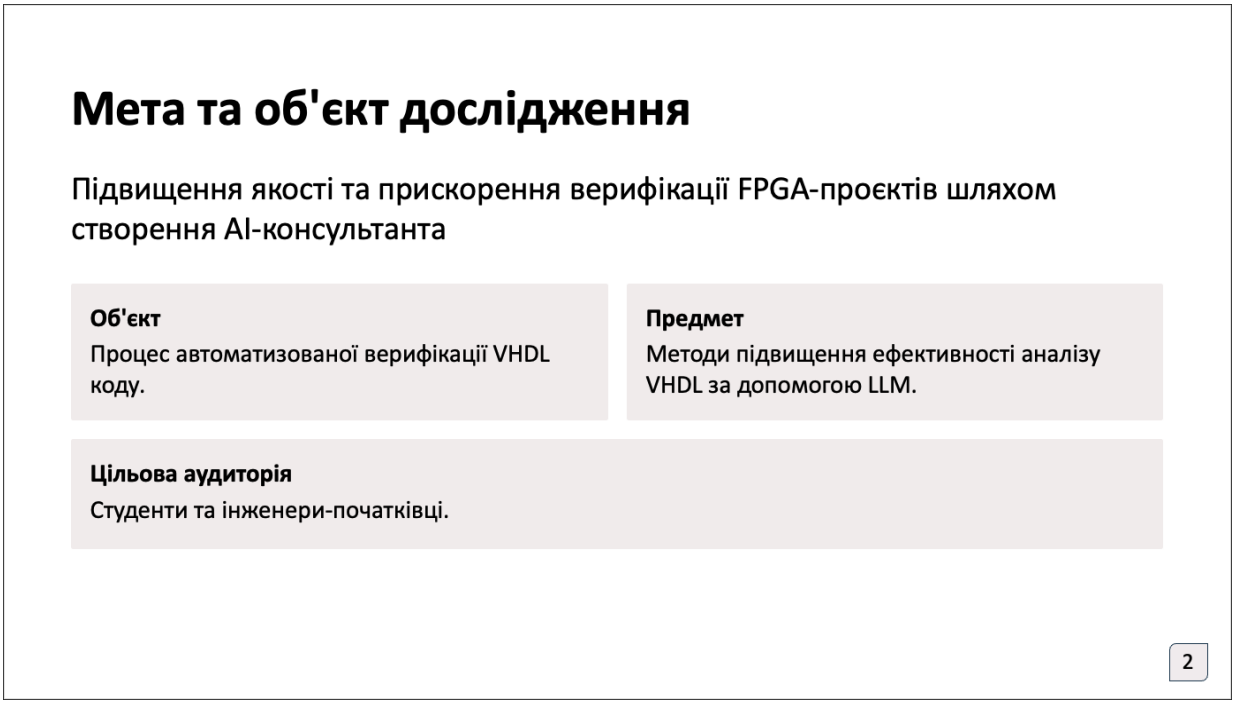


Рисунок Б.2 – Слайд презентації 2

Актуальність проблеми

1. **Verification Gap**
   Логічна ємність FPGA росте швидше, ніж продуктивність верифікації
2. **Вартість помилки**
   Лише 16% проєктів виходять без помилок з першого разу
3. **Проблема навчання**
   VHDL має високий поріг входу через складний синтаксис та паралельну природу

3

Рисунок Б.3 – Слайд презентації 3

Алгоритм проведення дослідження

1. **Підготовка тестового набору**
   25 пар файлів: з кодом (vhd) та “ground truth” (json)
2. **Розробка уніфікованого промпту**
   Використання технік Chain of Thought, Instruction Prompting, Context Injection та ін.
3. **Проведення експерименту**
   Збір даних відповідей від AI моделей
4. **Нормалізація даних**
   Приведення даних до одного формату для подальшого аналізу
5. **Аналіз нормалізованих даних**

4

Рисунок Б.4 – Слайд презентації 4

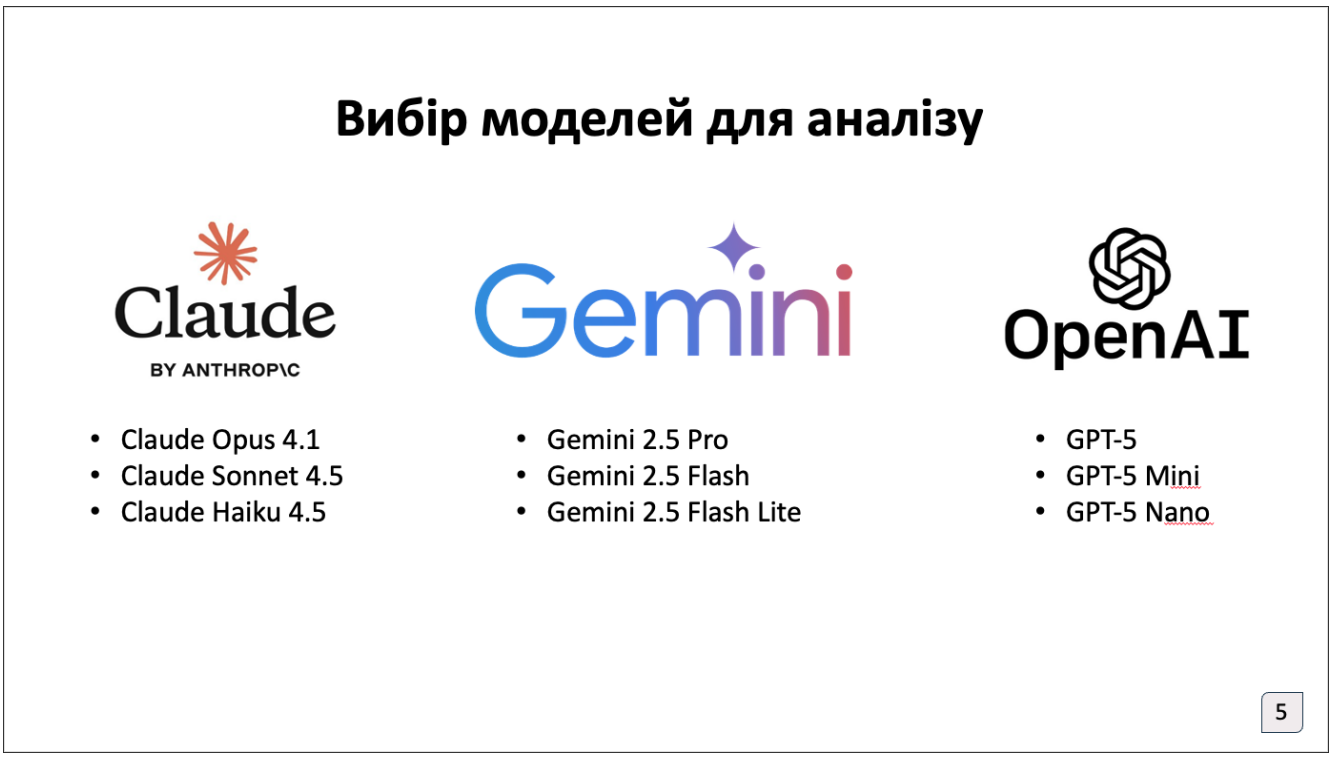


Рисунок Б.5 – Слайд презентації 5

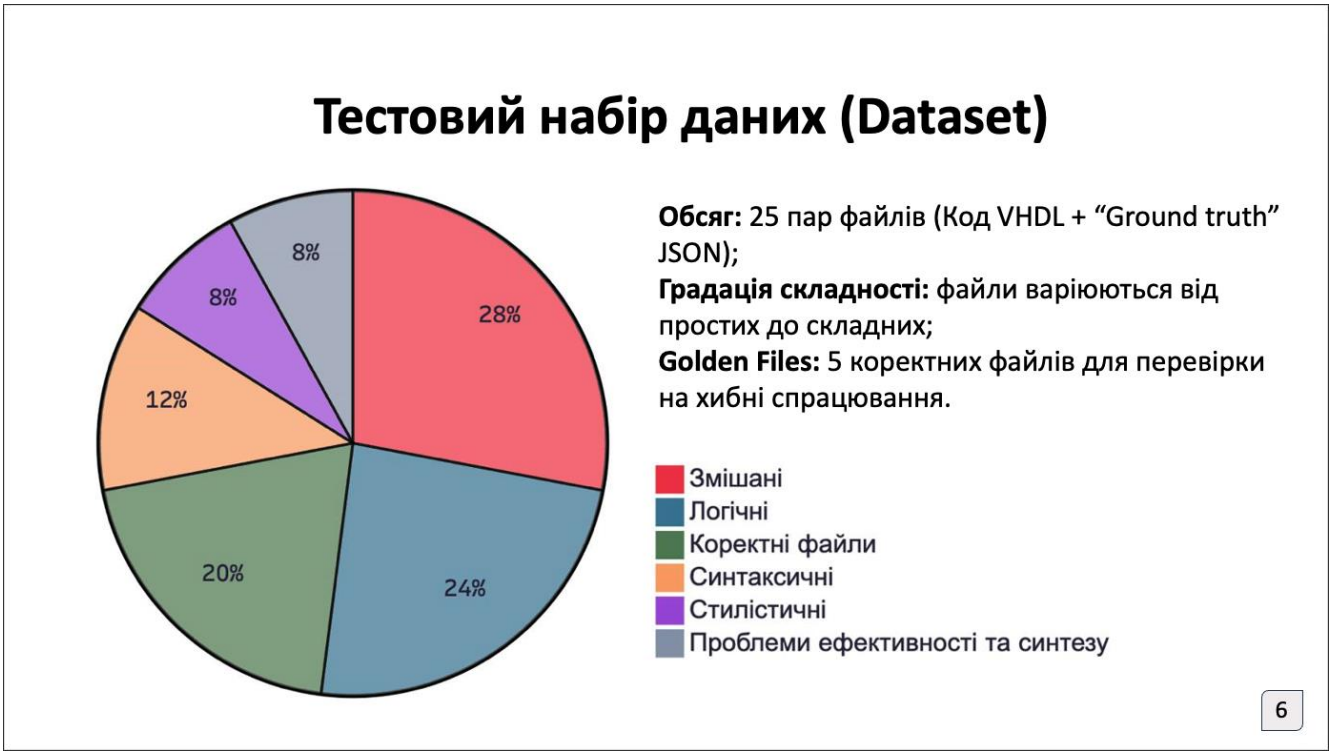


Рисунок Б.6 – Слайд презентації 6

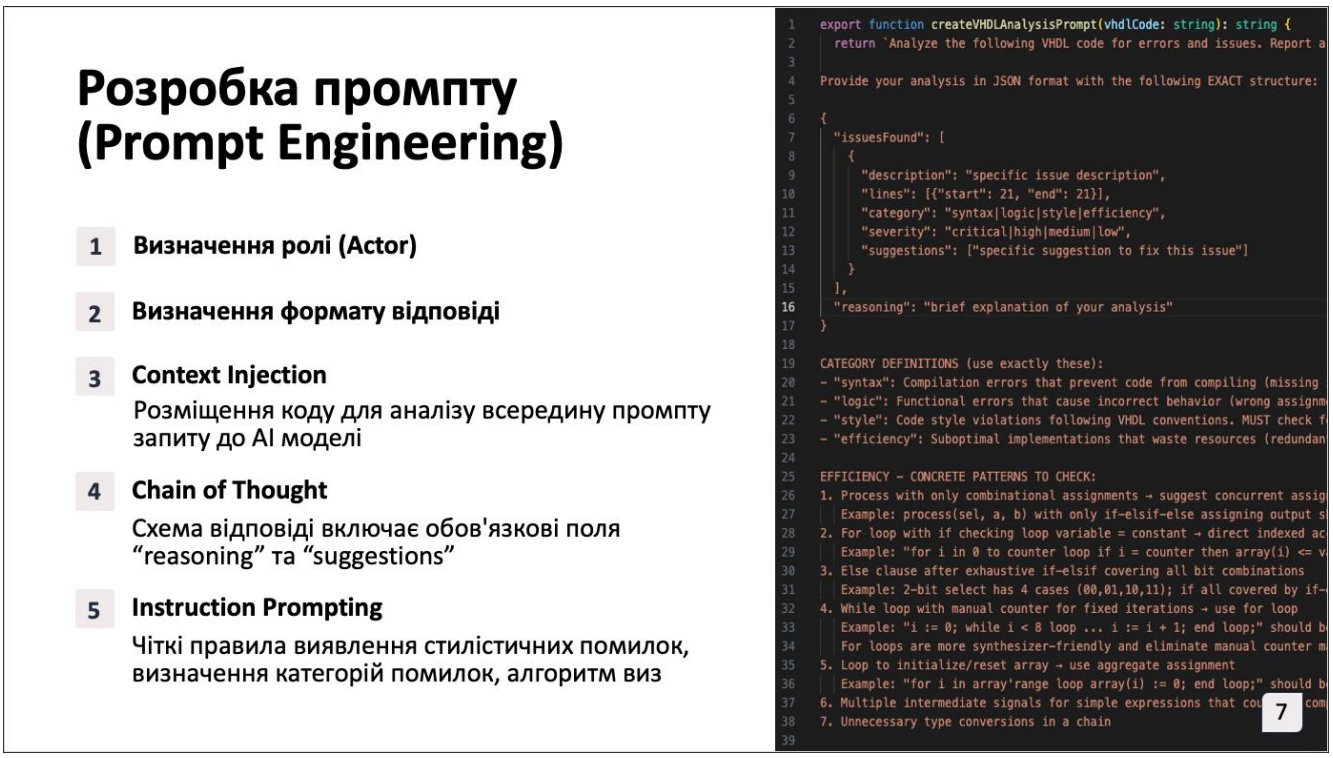


Рисунок Б.7 – Слайд презентації 7

Рисунок Б.8 – Слайд презентації 8

Рисунок Б.9 – Слайд презентації 9

Рисунок Б.10 – Слайд презентації 10

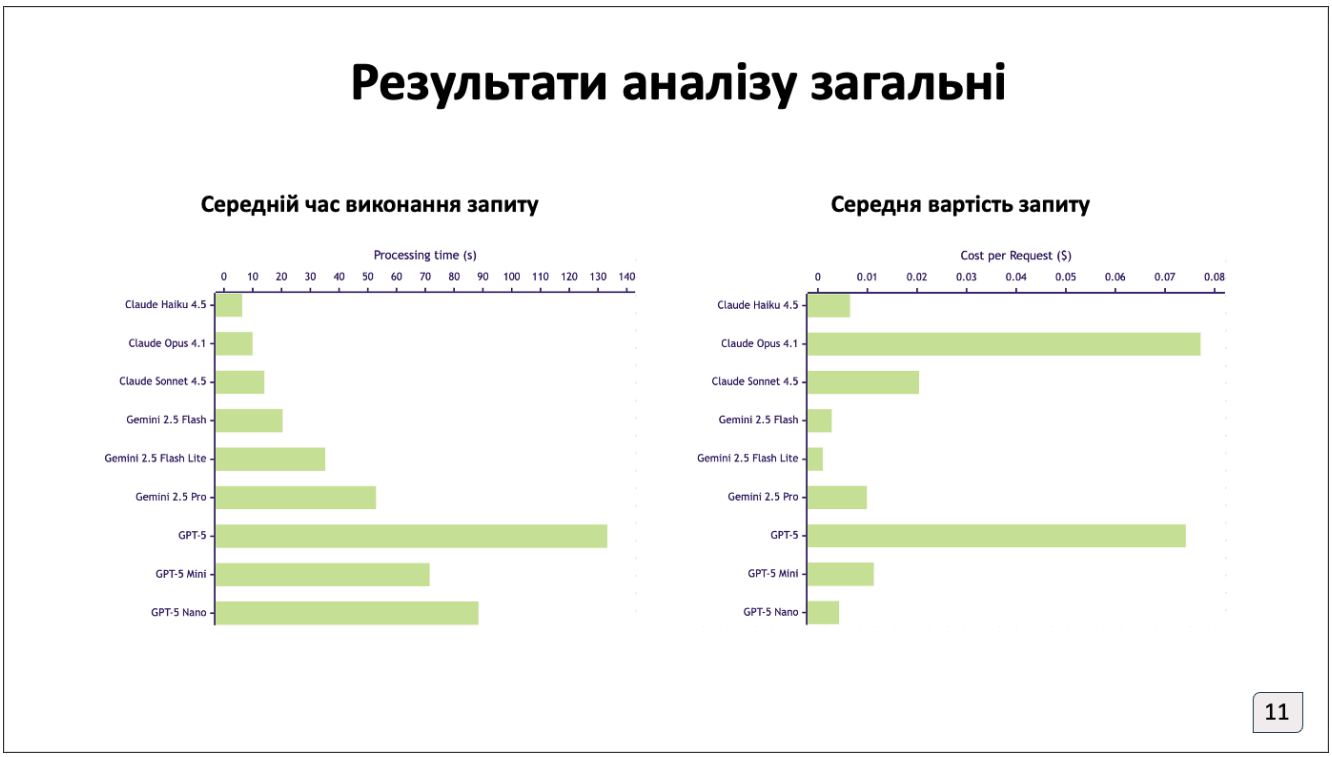


Рисунок Б.11 – Слайд презентації 11

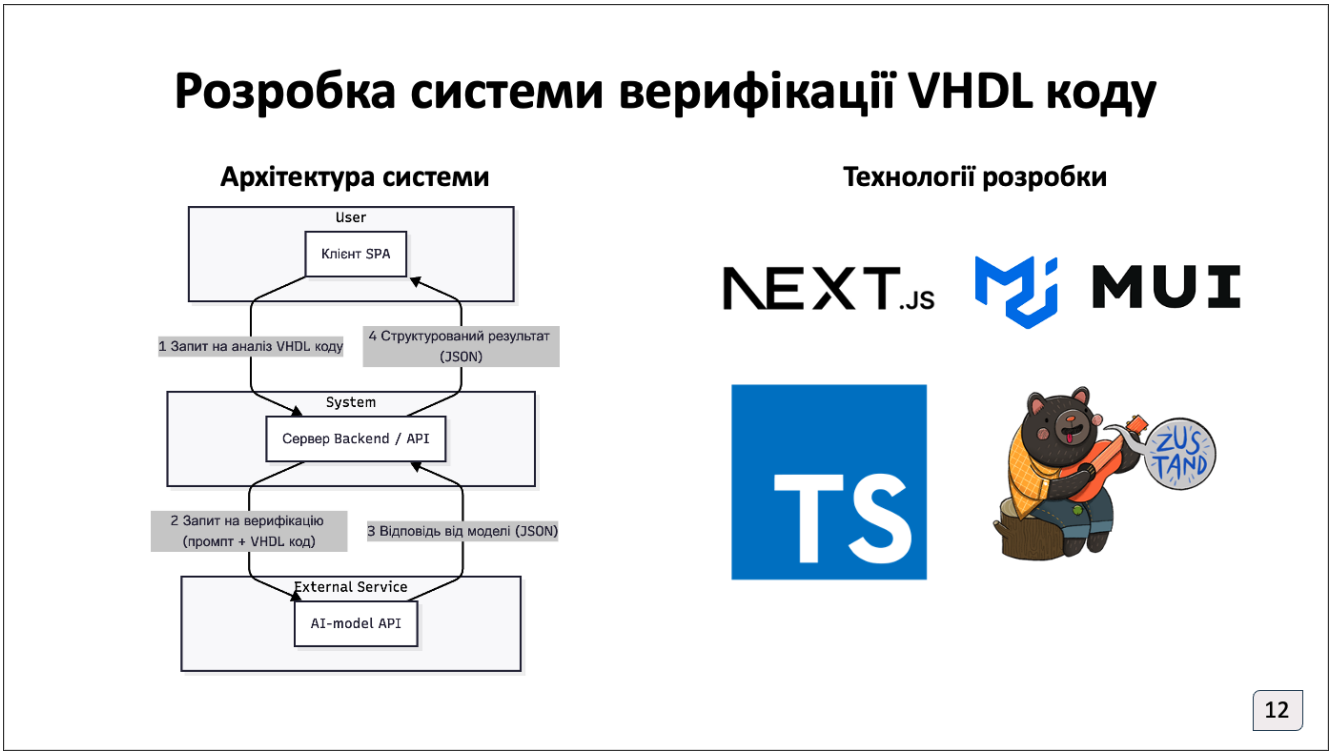


Рисунок Б.12 – Слайд презентації 12

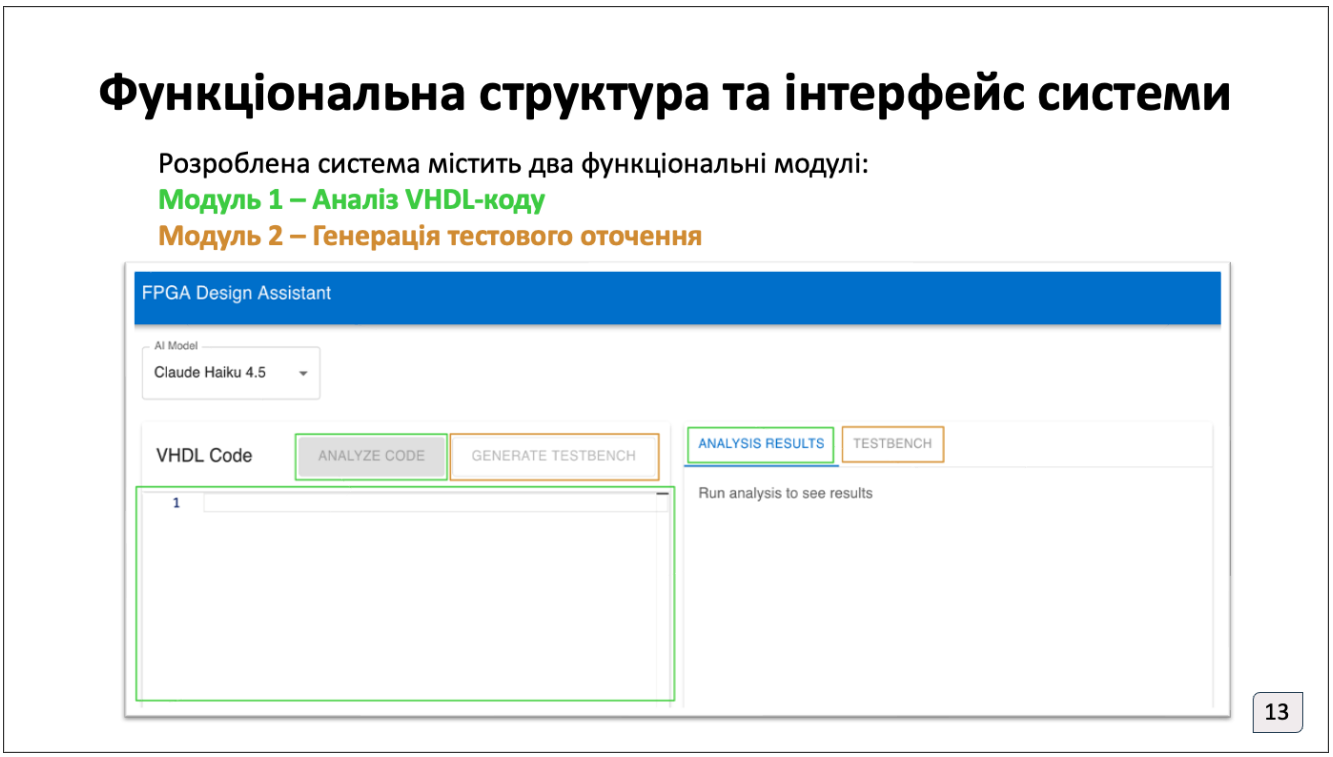


Рисунок Б.13 – Слайд презентації 13

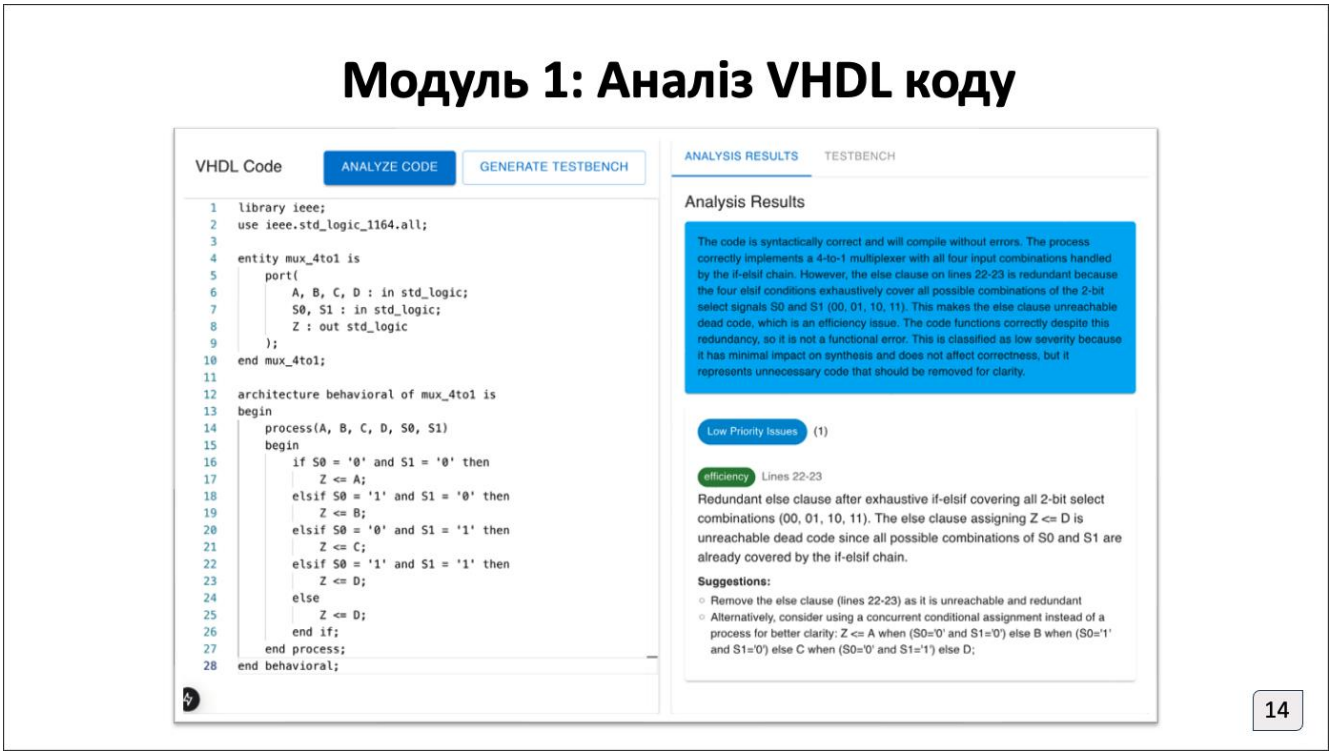
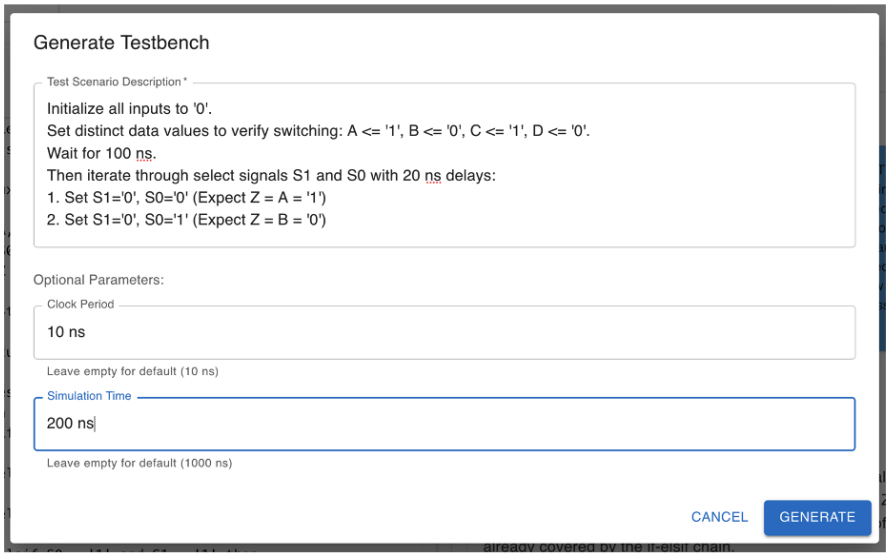


Рисунок Б.14 – Слайд презентації 14

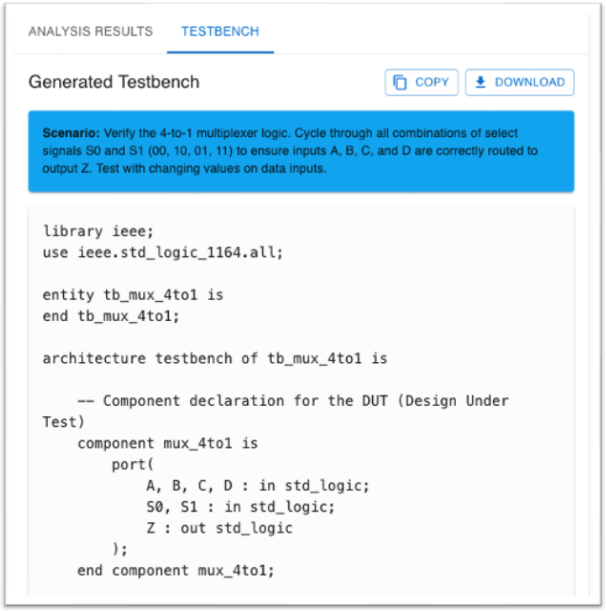
Модуль 2: Генерація тестового оточення



15

Рисунок Б.15 – Слайд презентації 15

Результат генерації коду тестового оточення



16

Рисунок Б.16 – Слайд презентації 16

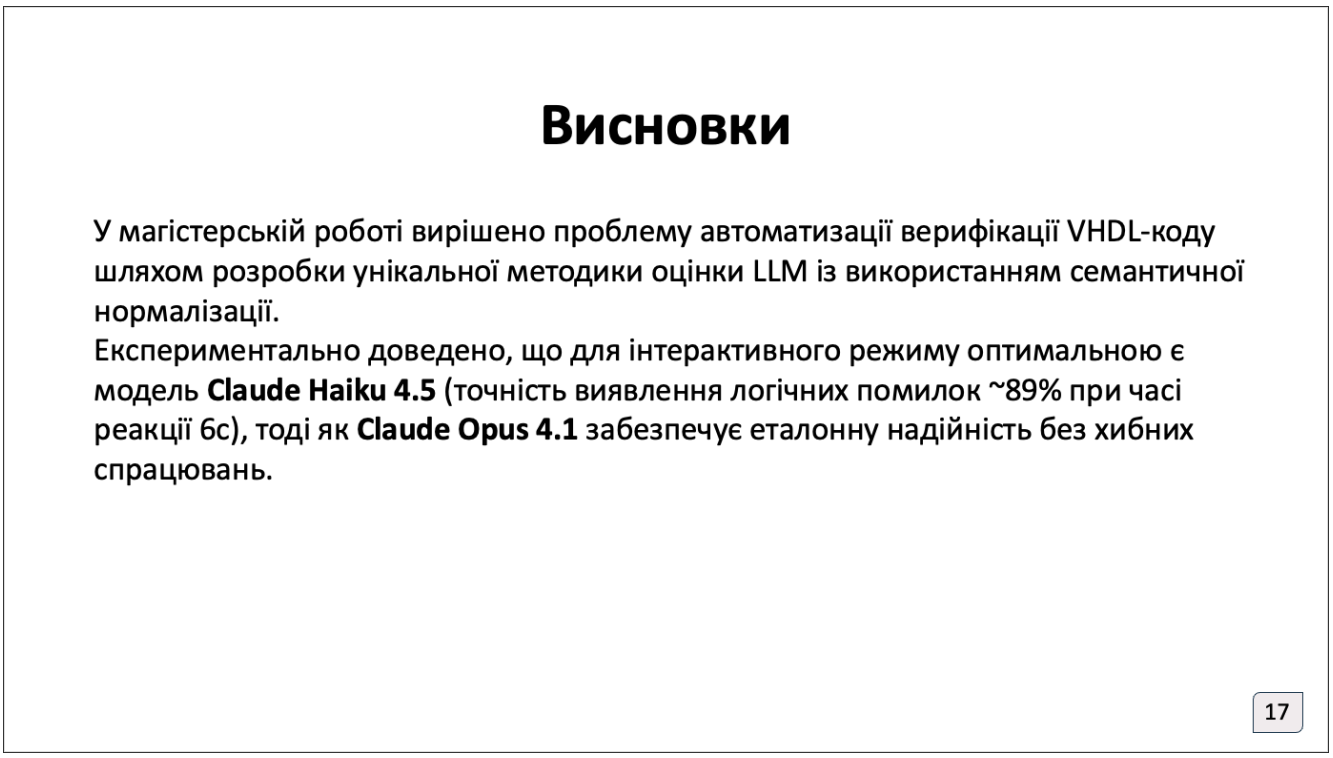


Рисунок Б.17 – Слайд презентації 17