

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ**

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Методика підбору комплектуючих для комп'ютерних систем з урахуванням бюджетних обмежень та наявної апаратної системи»

на здобуття освітнього ступеня магістра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

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

_____ Олексій КОЧЕТКОВ
(підпис)

Виконав: здобувач вищої освіти групи ПДМ-62
Олексій КОЧЕТКОВ

Керівник: _____ Світлана ШЕВЧЕНКО
канд. пед. наук, доц.

Рецензент: _____ Ім'я, ПРІЗВИЩЕ
науковий ступінь,
вчене звання

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ**
Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Магістр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

Інженерії програмного забезпечення

_____ Ірина ЗАМРІЙ

« _____ » _____ 2025 р.

**ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ**

Кочеткову Олексію Володимировичу

1. Тема кваліфікаційної роботи: «Методика підбору комплектуючих для комп'ютерних систем з урахуванням бюджетних обмежень та наявної апаратної конфігурації»

керівник кваліфікаційної роботи Світлана ШЕВЧЕНКО, канд. пед. наук, доц., доцент кафедри ПЗ

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «30» жовтня 2025 р. № 467.

2. Строк подання кваліфікаційної роботи «19» грудня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, параметри, метод підбору комплектуючих, вимоги до точності та ефективності підбору комплектуючих.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Дослідження методів підбору комплектуючих для ПК.

2. Аналіз технологій LLM-чатів для взаємодії з користувачем та алгоритмів для

- точного підбору комплектуючих для ПК.
3. Розробка методики для підбору комплектуючих для комп'ютерних систем з урахуванням бюджетних обмежень та наявної апаратної конфігурації.
 4. Програмна реалізація системи та проведення експериментальних досліджень
5. Перелік ілюстративного матеріалу: *презентація*
1. Мета, об'єкт та предмет дослідження.
 2. Порівняльна характеристика існуючих методів підбору комплектуючих для ПК.
 3. Математична модель автоматизованого підбору комплектуючих для комп'ютерних систем.
 4. Деталізація складових моделі.
 5. Система обмежень.
 6. Складові методики підбору комплектуючих.
 7. Діаграма послідовності роботи методики підбору комплектуючих.
 8. Практичний результат.
 9. Порівняльний аналіз ефективності розробленої методики.
6. Дата видачі завдання «31» жовтня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	31.10-02.11.2025	
2	Вивчення матеріалів для аналізу сучасних методів підбору комплектуючих для ПК	03.11-10.11.2025	
3	Дослідження методів підбору комплектуючих для ПК	11.11-13.11.2025	
4	Аналіз впливу методів підбору комплектуючих для ПК на якість та ефективність отриманої збірки	14.11-17.11.2025	
5	Розробка методики для підбору комплектуючих для ПК	18.11-20.11.2025	
6	Застосування методики для підбору комплектуючих для ПК	21.11-22.11.2025	
7	Оформлення роботи: вступ, висновки, реферат	22.11-23.11.2025	
8	Розробка демонстраційних матеріалів	23.11-24.11.2025	
9	Попередній захист роботи	24.10-19.12.2025	

Здобувач вищої освіти

(підпис)

Олексій КОЧЕТКОВ

Керівник

кваліфікаційної роботи

(підпис)

Світлана ШЕВЧЕНКО

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 111 стор., 1 табл., 29 рис., 24 джерел.

Мета роботи – підвищення ефективності та якості конфігурування комп'ютерних систем за допомогою методики автоматизованого підбору комплектуючих з урахуванням бюджетних обмежень та наявної апаратної конфігурації.

Об'єкт дослідження – процес конфігурування та модернізації апаратного забезпечення персональних комп'ютерних систем.

Предмет дослідження – методи та засоби підбору комплектуючих для комп'ютерних систем з урахуванням бюджетних обмежень та наявної апаратної конфігурації.

У роботі було розроблено методику підбору комплектуючих для ПК з урахуванням бюджетних обмежень та наявної апаратної конфігурації. Розробка враховує основні проблеми функціонування таких систем, зокрема, їхні суттєві обмеження, зокрема: орієнтація лише на нові збірки, відсутність механізмів модернізації та недостатня гнучкість при роботі з обмеженим бюджетом.

Здійснена формалізація задачі підбору конфігурації як задачі багатокритеріальної оптимізації на основі гібридного алгоритму з використанням LLM.

Розроблено методику автоматизованого підбору компонентів, яка базується на поєднанні гібридного алгоритму (для точної математичної оптимізації) та великої мовної моделі (для взаємодії з користувачем).

Реалізовано методику у вигляді веб-додатку, що дозволяє отримувати рекомендації у форматі діалогу та візуалізувати запропоновані конфігурації.

Проведено експерименти для розробленого методу та порівняння його ефективності з існуючими підходами.

Оцінено ефективність та точність системи підбору комплектуючих для ПК. Результати дослідження підтверджують успішність та перспективність використання гібридного методу для підбору комплектуючих для збірки ПК.

КЛЮЧОВІ СЛОВА: ПІДБОР КОМПЛЕКТУЮЧИХ ДЛЯ ПК, LLM, ГІБРИДНИЙ АЛГОРИТМ, БАГАТОКРИТЕРІАЛЬНА ОПТИМІЗАЦІЯ, КОНФІГУРАЦІЯ.

ABSTRACT

The text part of the qualification work for obtaining the master's degree: 111 pages, 1 table, 29 figures, 24 sources.

The aim of the work is to increase the efficiency and quality of computer systems configuration through an automated component selection methodology, taking into account budget constraints and the existing hardware configuration.

The object of research is the process of configuration and modernization of personal computer system hardware.

The subject of research includes methods and tools for selecting components for computer systems considering budget constraints and the existing hardware configuration.

The thesis develops a methodology for selecting PC components considering budget constraints and the existing hardware configuration. The development addresses the main operational problems of such systems, specifically their significant limitations: orientation solely towards new builds, lack of modernization mechanisms, and insufficient flexibility when working with a limited budget.

The task of configuration selection has been formalized as a multi-criteria optimization problem based on a hybrid algorithm using Large Language Models (LLM).

An automated component selection methodology has been developed, based on a combination of a genetic algorithm (for precise mathematical optimization) and a large language model (for interaction with the user).

The methodology has been implemented in the form of a web application that allows users to receive recommendations in a dialogue format and visualize the proposed configurations.

Experiments were conducted for the developed method, comparing its effectiveness with existing approaches. The efficiency and accuracy of the PC component selection system were evaluated. The research results confirm the success and potential of using the hybrid method for selecting components for PC assembly.

KEYWORDS: PC COMPONENT SELECTION, LLM, HYBRID ALGORITHM, MULTI-CRITERIA OPTIMIZATION, CONFIGURATION.

ЗМІСТ

ВСТУП	12
1 АНАЛІЗ ПРАКТИЧНИХ АСПЕКТІВ ВИКОРИСТАННЯ СУЧАСНИХ КОНФІГУРАТОРІВ ДЛЯ ЗБИРАННЯ ПК	15
1.1 Поняття та призначення конфігуратора ПК.....	15
1.1.1 Сутність конфігуратора та його роль у підборі комплектуючих.....	16
1.1.2 Класифікація сучасних конфігураторів комп'ютерних систем	18
1.1.2.1 Класифікація за ступенем інтеграції та бізнес-моделлю	18
1.1.2.2 Класифікація за рівнем автоматизації прийняття рішень	18
1.1.2.3 Технічна класифікація програмних засобів конфігуратора.....	19
1.1.3 Вимоги до функціональності та якості програмних засобів підбору комплектуючих.....	20
1.1.3.1 Функціональні вимоги	20
1.1.3.2 Вимоги якості	21
1.2 Порівняльний аналіз існуючих рішень.....	22
1.2.1 Огляд наявних конфігураторів і їхніх можливостей.....	22
1.2.1.1 Telemart.ua.....	22
1.2.1.2 be quiet! PSU Calculator	24
1.2.1.3 E-Katalog	26
1.2.1.4 ChatGPT.....	29
1.2.2 Аналіз метод підбору комплектуючих, що використовуються на практиці...31	
1.2.2.1 Метод евристичного підбору.....	31
1.2.2.2 Метод параметричної фільтрації.....	32
1.2.2.3 Статистичний метод.....	34
1.2.2.4 Метод комбінаторної оптимізації	36
1.2.3 Аналіз сучасних підходів ШІ в конфігуруванні.....	37
1.2.3.1 Рекомендаційні системи	38
1.2.3.2 Генеративні нейронні мережі	40
1.3 Аналітичні дані для точного підбору комплектуючих для збірки персонального комп'ютера	42
1.3.1 Технічні специфікації та параметри сумісності	42
1.3.2 Цінові дані та ринкові позиціонування.....	44
1.3.3 Дані продуктивності (Бенчмарки)	45

1.3.4 Емпіричні дані та якісні показники.....	46
2 МЕТОДИКА ПІДБОРУ КОМПЛЕКТУЮЧИХ ДЛЯ КОМП'ЮТЕРНИХ СИСТЕМ 3 УРАХУВАННЯМ БЮДЖЕТНИХ ОБМЕЖЕНЬ ТА НАЯВНОЇ АПАРАТНОЇ КОНФІГУРАЦІЇ.....	48
2.1 Математична постановка задачі та формалізація критеріїв оптимізації.....	48
2.1.1 Формалізація простору рішень.....	48
2.1.2 Критерії оптимізації та цільова функція.....	50
2.1.3 Система обмежень та умови сумісності.....	52
2.1.3.1 Апаратна (Логічна) несумісність (<i>Isocket, Ichipset</i>).....	53
2.1.3.2 Сумісність підсистеми пам'яті (<i>Iram</i>).....	53
2.1.3.3 Енергетична сумісність (<i>Ipower</i>).....	53
2.1.3.4 Фізична сумісність (<i>Iphys</i>).....	54
2.1.3.5 Деталізація змінних та атрибутів моделі сумісності.....	54
2.1.4 Модифікація цільової функції методом штрафів.....	55
2.2 Обґрунтування та опис застосованих алгоритмів.....	57
2.2.1 Генетичний алгоритм як метод пошуку оптимальної конфігурації.....	57
2.2.2 Механізм фіксації генів для задач модернізації.....	60
2.2.3 Інтеграція великих мовних моделей (LLM) для аналізу вимог та генерації пояснень.....	63
2.2.3.1 Процес обробки запиту.....	63
2.2.3.2 Генерація пояснень та аргументація рішень.....	65
2.3 Архітектура розробленої методики.....	67
2.4 Діаграма послідовності.....	71
2.5 Алгоритм роботи конфігуратора.....	74
3 ПРОГРАМНА РЕАЛІЗАЦІЯ КОНФІГУРАТОРА ДЛЯ ПІДБОРУ КОМПЛЕКТУЮЧИХ ДЛЯ КОМП'ЮТЕРНИХ СИСТЕМ З УРАХУВАННЯМ БЮДЖЕТНИХ ОБМЕЖЕНЬ ТА НАЯВНОЇ АПАРАТНОЇ КОНФІГУРАЦІЇ.....	78
3.1 Опис розробки методу.....	78
3.2 Опис використання програмних засобів.....	82
3.2.1 FastAPI.....	82
3.2.2 Інтеграція з Google Gemini API.....	84
3.2.3 Pydantic V2.....	88
3.2.4 Uvicorn.....	91
3.2.5 SQLite як СКБД для каталогу компонентів.....	92

3.3 Загальна архітектура та організація проекту	93
3.3.1 Сервіс оркестрації чату (chat_orchestrator_service)	95
3.3.2 Сервіс компонентів (components_service).....	98
3.3.3 Фронтенд (frontend)	102
3.3.4 Сервіс генетичного конфігуратора (ga_configurator_service).....	104
3.3.5 Сервіс Gemini (gemini_service)	110
3.4 Опис інтерфейсу користувача	116
3.5 Результати експериментального дослідження ефективності методики	119
ВИСНОВКИ.....	122
ПЕРЕЛІК ПОСИЛАНЬ	124
ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ	126
ДОДАТОК Б. ЛІСТИНГ ОСНОВНИХ МОДУЛІВ	132

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

БД — база даних

БЖ — блок живлення

ГА — генетичний алгоритм

ОЗП — оперативний запам'ятовуючий пристрій (оперативна пам'ять)

ПК — персональний комп'ютер

ПЗ — програмне забезпечення

САПР — система автоматизованого проєктування

СУБД — система управління базами даних

ШІ — штучний інтелект

API — Application Programming Interface (прикладний інтерфейс програмування)

BIOS — Basic Input/Output System (базова система введення-виведення)

CPU — Central Processing Unit (центральний процесор)

DTO — Central Processing Unit (об'єкт передачі даних (патерн проєктування))

GPU — Graphics Processing Unit (графічний процесор (відеокарта))

HDD — Hard Disk Drive (накопичувач на жорстких магнітних дисках)

LLM — Large Language Model (велика мовна модель)

NER — Named Entity Recognition (технологія розпізнавання іменованих сутностей у тексті)

NLP — Natural Language Processing (обробка природної мови)

PSU — Power Supply Unit (блок живлення)

RAM — Random Access Memory (оперативна пам'ять)

SSD — Solid State Drive (твердотільний накопичувач)

TDP — Thermal Design Power (розрахункова теплова потужність)

XAI — Explainable AI (пояснюваний штучний інтелект)

ВСТУП

Сучасний ринок комп'ютерних комплектуючих характеризується стрімким оновленням асортименту, високою варіативністю технічних стандартів та значною ціновою динамікою. Для пересічного користувача процес самостійного підбору персонального комп'ютера (ПК) перетворився на складну інженерну задачу, що вимагає глибоких знань про апаратну сумісність (сумісність сокетів, типів пам'яті, енергоспоживання тощо).

Особливої гостроти набуває проблема модернізації (апгрейду) наявних систем. Більшість існуючих інструментів (онлайн-конфігуратори інтернет-магазинів) орієнтовані виключно на збірку нових ПК і не надають інструментарію для ефективного використання наявних компонентів користувача. Водночас, використання популярних чат-ботів на базі штучного інтелекту (LLM) для цієї задачі несе ризики отримання недостовірної інформації («галюцинацій») та технічно несумісних рекомендацій.

В умовах обмеженого бюджету та необхідності економії часу виникає нагальна потреба у створенні автоматизованої методики, яка б поєднувала точність математичної оптимізації з гнучкістю природно-мовної взаємодії. Розробка такої системи дозволить мінімізувати помилки при підборі, скоротити час прийняття рішень та оптимізувати витрати користувачів, що зумовлює актуальність теми магістерської роботи.

Мета роботи - підвищення ефективності та якості конфігурування комп'ютерних систем за допомогою методики автоматизованого підбору комплектуючих з урахуванням бюджетних обмежень та наявної апаратної конфігурації.

Об'єкт дослідження - процес конфігурування та модернізації апаратного забезпечення персональних комп'ютерних систем.

Предмет дослідження - методи та засоби підбору комплектуючих для комп'ютерних систем з урахуванням бюджетних обмежень та наявної апаратної конфігурації.

Для досягнення мети потрібно вирішити наступні **завдання**:

- 1) провести дослідження та аналіз існуючих методів конфігурування комп'ютерних систем (онлайн-конфігуратори, прайс-агрегатори, LLM-чати), щоб виявити їхні обмеження, зокрема, у вирішенні задач модернізації (апгрейду) та гарантування апаратної сумісності;
- 2) формалізувати задачу підбору конфігурації як задачу багатокритеріальної оптимізації: визначити цільову функцію, систему обмежень та розробити механізм для відсіювання технічно несумісних рішень;
- 3) розробити гібридну методику, що поєднує генетичний алгоритм (для оптимізації) та великі мовні моделі (для інтерпретації вимог);
- 4) виконати програмну реалізацію системи у вигляді веб-додатку з використанням мікросервісної архітектури;
- 5) провести експерименти на реальних сценаріях (збірка з нуля, бюджетний апгрейд) та оцінити ефективність алгоритму за показниками: час генерації, точність влучання в бюджет та ступінь збалансованості системи;
- 6) провести порівняльний аналіз ефективності та якості різних методів підбору комплектуючих на реальних даних, використовуючи обрану метрику.

У дослідженні використано наступні **методи**: порівняльний, системно-структурний, методи математичного моделювання, методи багатокритеріальної оптимізації, методи програмної інженерії.

Наукова новизна одержаних результатів Розроблено методику автоматизованого підбору компонентів, яка базується на поєднанні гібридного алгоритму (для точної математичної оптимізації) та великої мовної моделі (для взаємодії з користувачем).

Практичне значення одержаних результатів. Запропоновано реалізацію автоматизованого підбору комплектуючих для комп'ютерних систем з урахуванням бюджетних обмежень та наявної апаратної конфігурації, що дозволить удосконалити процес збірки ПК самостійним чином.

Апробація результатів магістерської роботи.

1. Кочетков О.В., Шевченко С.М. Багатокритеріальна оптимізація для вибору найкращої конфігурації комп'ютерних систем // Вісник Херсонського національного технічного університету» № 4, 2025.

2. Кочетков О.В., Шевченко С.М. Розробка алгоритму підбору комплектуючих для комп'ютерних систем з урахуванням бюджетних обмежень та наявної апаратної конфігурації // VI Всеукраїнська науково-технічна конференція «Сучасний стан та перспективи розвитку IoT», 15 квітня 2025 року. Збірник тез. – К.: ДУІКТ, 2025. – С. 158-160.

3. Кочетков О.В., Шевченко С.М. Визначення вимог до алгоритму підбору комплектуючих для комп'ютерних систем з урахуванням бюджетних обмежень та наявної апаратної конфігурації // VI Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 24 квітня 2025 року. Збірник тез. – К.: ДУІКТ, 2025. – С.558-560

1 АНАЛІЗ ПРАКТИЧНИХ АСПЕКТІВ ВИКОРИСТАННЯ СУЧАСНИХ КОНФІГУРАТОРІВ ДЛЯ ЗБИРАННЯ ПК

1.1 Поняття та призначення конфігуратора ПК

В епоху стрімкого розвитку інформаційних технологій персональний комп'ютер (ПК) залишається ключовим інструментом для вирішення широкого спектра задач: від офісного документообігу до складних наукових обчислень та рендерингу тривимірної графіки. Архітектура сучасного ПК є відкритою та модульною, що дозволяє користувачам самостійно формувати систему з окремих вузлів (комплектуючих). Однак, така гнучкість породжує проблему надлишкового вибору: на ринку одночасно присутні тисячі найменувань компонентів різних поколінь, стандартів та цінових категорій.

У цьому контексті конфігуратор персонального комп'ютера визначається як спеціалізований програмний засіб або вебсервіс, призначений для автоматизації процесу проектування комп'ютерних систем. Це інтерактивне середовище, яке забезпечує підбір, перевірку сумісності та валідацію набору апаратних компонентів відповідно до заданих критеріїв користувача.

Основне призначення конфігуратора полягає у вирішенні проблеми інформаційної ентропії при виборі апаратного забезпечення. Його функціональне навантаження можна розділити на три вектори:

1. Запобігання замовленню несумісного обладнання, що є найпоширенішою проблемою при ручному підборі.

2. Допомога у раціональному розподілі бюджету для досягнення максимальної продуктивності системи (уникнення дисбалансу компонентів).

3. Надання можливості користувачам без глибоких інженерних знань створити працездатну та ефективну комп'ютерну систему.

Таким чином, сучасний конфігуратор трансформувався з простого

електронного каталогу в складну експертну систему, що акумулює знання про стандарти індустрії та логічні зв'язки між ними.

1.1.1 Сутність конфігуратора та його роль у підборі комплектуючих

Сутність конфігуратора як об'єкта дослідження полягає в його здатності виступати інтелектуальним посередником між користувачем та базою даних апаратного забезпечення. Він формалізує нечіткі вимоги користувача (наприклад, «комп'ютер для ігор») у чіткі технічні специфікації[1].

Роль конфігуратора у підборі комплектуючих є багатогранною і реалізується через наступні аспекти:

1. Гарантування багаторівневої сумісності - це фундаментальна роль системи. Конфігуратор виконує автоматичну верифікацію сумісності на трьох рівнях:

- логічна та інтерфейсна сумісність - перевірка відповідності сокетів центральних процесорів (CPU) та материнських плат, підтримки чіпсетами версій BIOS, сумісності типів оперативної пам'яті (DDR4/DDR5) та інтерфейсів накопичувачів (SATA/NVMe);
- електрична сумісність - розрахунок енергетичного профілю системи (Total Design Power — TDP). Система аналізує пікове споживання всіх компонентів та перевіряє, чи здатен обраний блок живлення забезпечити стабільну роботу, враховуючи необхідний запас потужності та наявність відповідних конекторів живлення (8-pin CPU, PCIe 6+2 pin тощо);
- фізична сумісність - аналіз геометричних параметрів. Конфігуратор перевіряє, чи вміститься відеокарта у вибраний корпус за довжиною, чи дозволить висота процесорного кулера закрити бічну панель корпусу, та чи не перекриватимуть модулі пам'яті з високими радіаторами встановлення системи охолодження.

2. Економічна оптимізація та балансування є важливою роллю конфігуратора для запобігання створенню незбалансованих систем (так званих «bottleneck»

ефектив)[2]. Система допомагає користувачеві уникнути ситуацій, коли надпотужний процесор працює в парі зі слабкою відеокартою, або коли дорогі компоненти встановлюються в дешевий корпус з поганою вентиляцією. Конфігуратор виступає інструментом бюджетного менеджменту, дозволяючи бачити динаміку зміни загальної вартості при заміні окремих вузлів.

3. Освітня та консультативна функція, тобто сучасні конфігуратори виконують роль навчальної платформи. У процесі підбору користувач отримує контекстні підказки, бачить альтернативні варіанти та вчиться розуміти залежності між компонентами. Це підвищує технічну грамотність споживача та дозволяє йому робити усвідомлений вибір, що базується на об'єктивних характеристиках, а не лише на маркетингових назвах.

Підсумовуючи, можна стверджувати, що роль конфігуратора полягає не лише в автоматизації процесу покупки, а й у забезпеченні інженерної цілісності майбутньої комп'ютерної системи. У контексті даної магістерської роботи особливу увагу буде приділено розширенню цієї ролі на задачі модернізації (апгрейду), що є слабким місцем існуючих рішень

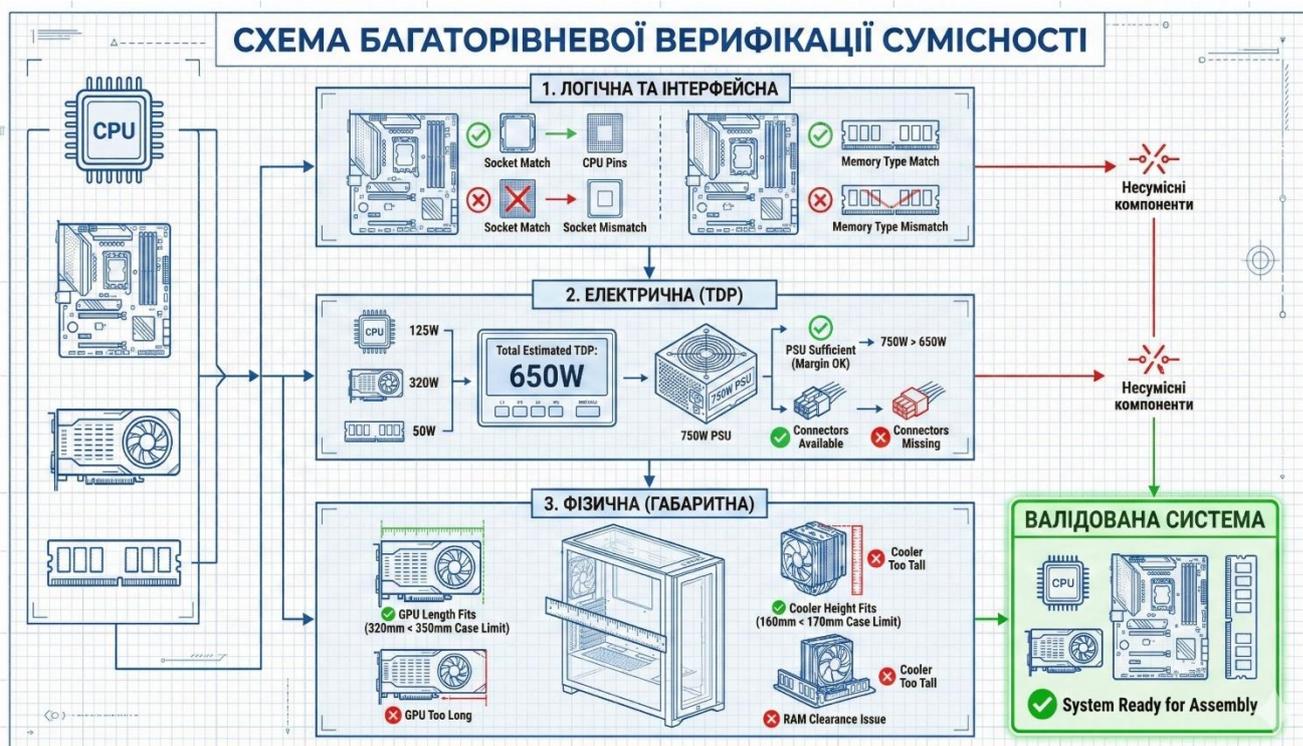


Рис. 1.1 Схема багаторівневої верифікації сумісності

1.1.2 Класифікація сучасних конфігураторів комп'ютерних систем

Сфера програмного забезпечення для підбору комп'ютерних комплектуючих є неоднорідною. Існуючі рішення суттєво відрізняються за функціоналом, цільовою аудиторією та алгоритмічною базою. Для системного аналізу доцільно провести класифікацію конфігураторів за трьома ключовими ознаками: за ступенем інтеграції, за рівнем автоматизації прийняття рішень та за сферою застосування.

1.1.2.1 Класифікація за ступенем інтеграції та бізнес-моделлю

За цим критерієм конфігуратори поділяються на інтегровані E-commerce системи, незалежні аналітичні агрегатори та OEM-рішення (рис.1.2). Перші (наприклад, на базі Telemart або Rozetka) функціонують як інструмент стимулювання продажів конкретного ритейлера, через що вибір компонентів жорстко обмежується наявними складськими залишками, часто змушуючи користувача обирати субоптимальні альтернативи. Натомість незалежні платформи (на кшталт PCPartPicker) агрегують пропозиції різних продавців, забезпечуючи об'єктивність вибору та пошук найнижчої ринкової ціни, проте вимагають від користувача вищої технічної кваліфікації. Третій тип — OEM-конфігуратори виробників (Dell, Lenovo) — пропонує лише кастомізацію серійних моделей у межах закритої екосистеми без можливості вибору брендів окремих вузлів.

1.1.2.2 Класифікація за рівнем автоматизації прийняття рішень

Цей критерій визначає «інтелект» системи та поділяє інструменти на пасивні, вузькоспеціалізовані та активні генеративні. Пасивні системи, що наразі домінують на ринку, працюють за принципом фільтрації: користувач обирає деталі самостійно, а система лише приховує несумісні варіанти, не пропонуючи оптимізації. Вузькоспеціалізовані калькулятори вирішують локальні інженерні задачі (розрахунок потужності БЖ або оцінка «пляшкового горла»), але не формують цілісну збірку. Найвищим рівнем розвитку є активні генеративні

системи, до яких належить і методика, що розробляється в даній роботі; вони здатні автоматично генерувати готову специфікацію на основі абстрактних параметрів (бюджет, задачі), виступаючи в ролі експертної системи.



Рис. 1.2 Класифікація конфігураторів з ступенем інтеграції та бізнес-моделлю

1.1.2.3 Технічна класифікація програмних засобів конфігуратора

З технічної точки зору програмні засоби поділяються на хмарні (Web-based) та локальні (Standalone) рішення (рис. 1.3). Хмарні сервіси, що працюють через веб-браузер, є сучасним стандартом де-факто, оскільки забезпечують централізоване оновлення бази даних компонентів та цін у реальному часі, не вимагаючи від користувача встановлення додаткового ПЗ. Локальні додатки використовуються вкрай рідко, переважно у професійній діяльності сервісних центрів або в закритих корпоративних мережах для роботи в умовах обмеженого доступу до Інтернету.



Рис. 1.3 Технічна класифікація програмних засобів конфігураторів

1.1.3 Вимоги до функціональності та якості програмних засобів підбору комплектуючих

Проектування ефективної системи автоматизованого підбору компонентів вимагає чіткої формалізації вимог, які можна розділити на дві основні групи: функціональні вимоги (що система повинна робити) та вимоги до якості (як система повинна працювати). Аналіз недоліків існуючих аналогів дозволяє сформулювати наступний перелік критичних вимог.

1.1.3.1 Функціональні вимоги

Ця група вимог визначає набір операцій, які система зобов'язана виконувати для вирішення задачі конфігурування:

1. Мультирівнева валідація сумісності, де система повинна автоматично перевіряти сумісність компонентів на трьох рівнях:

- Відповідність інтерфейсів (Socket, Slot, Port) та стандартів (DDR, PCIe).
- Розрахунок сумарного TDP та перевірка достатності потужності блоку живлення з урахуванням пікових навантажень.
- Перевірка геометричних обмежень (довжина GPU vs корпус, висота CPU-кулера vs корпус).

2. На відміну від більшості існуючих рішень, система повинна мати функціонал для введення переліку наявних компонентів користувача. Алгоритм має враховувати ці компоненти як незмінні та підбирати нові деталі так, щоб вони були сумісні зі старими, мінімізуючи загальні витрати.

3. Система не повинна просто підсумовувати вартість, вона має реалізовувати алгоритм пошуку глобального екстремуму цільової функції (максимальна продуктивність при обмеженому бюджеті). Це вимагає відходу від простої фільтрації до комбінаторної оптимізації.

4. Для зниження порогу входження система повинна вміти інтерпретувати неструктуровані текстові запити (наприклад, «Хочу комп'ютер для монтажу відео до 30 000 грн»), автоматично виділяючи з них ключові параметри (ціль, бюджет, обмеження).

1.1.3.2 Вимоги якості

Ці вимоги визначають атрибути якості системи, критичні для користувацького досвіду та надійності результатів:

1. Система повинна мінімізувати ймовірність надання помилкових рекомендацій («галюцинацій»), що є характерним для генеративних нейромереж. Це досягається шляхом використання детермінованих алгоритмів перевірки сумісності поверх рекомендацій ШІ.

2. База даних компонентів має підтримуватися в актуальному стані, відображаючи реальні ринкові ціни та наявність товарів. Застарілі ціни роблять будь-яку оптимізацію бюджету нерелевантною.

3. Час генерації оптимальної конфігурації не повинен перевищувати психологічного порогу очікування користувача (10–15 секунд), навіть при переборі тисяч можливих комбінацій.

4. Система не повинна бути «чорною скринькою». Користувач має отримати не лише список товарів, а й аргументоване пояснення: чому обрано саме цей процесор, який приріст продуктивності це дасть, та чому ця збірка є збалансованою.

1.2 Порівняльний аналіз існуючих рішень

1.2.1 Огляд наявних конфігураторів і їхніх можливостей

Для виявлення недоліків існуючого програмного забезпечення було проведено аналіз чотирьох типів конфігураторів, що використовуються на практиці.

1.2.1.1 Telemart.ua

Як яскравого представника класу інтегрованих E-commerce систем доцільно розглянути онлайн-конфігуратор українського ритейлера Telemart.ua (рис. 1.4). Цей сервіс є класичним прикладом інструменту, головною метою якого є не стільки інженерне проектування, скільки стимулювання продажів та спрощення шляху клієнта до оформлення замовлення. Функціонально він є інтерактивною надбудовою над каталогом магазину, яка дозволяє користувачеві формувати кошик товарів у вигляді цілісної системної збірки, оперуючи виключно актуальними складськими залишками.

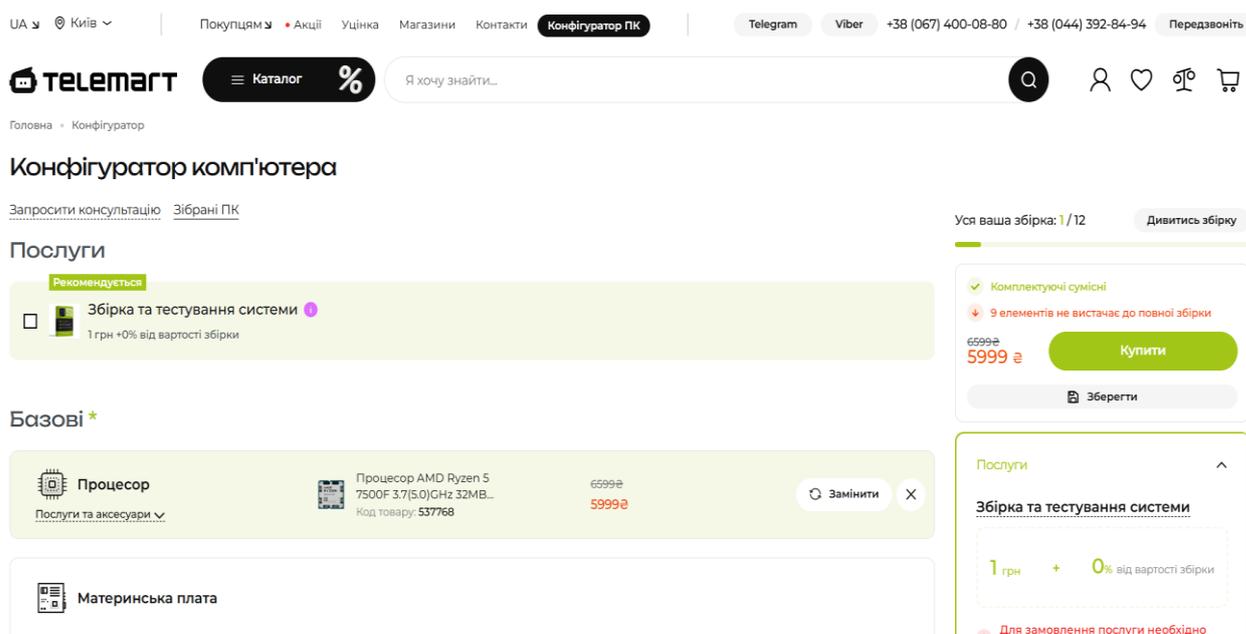


Рис. 1.4 Конфігуратор Telemart.ua

Принцип роботи конфігуратора базується на алгоритмі послідовної фільтрації (рис. 1.5). Процес підбору реалізовано у вигляді покрокового майстра: користувач починає з вибору ключового компонента (зазвичай процесора), після чого система автоматично накладає фільтри на наступні категорії. Наприклад, при виборі процесора Intel Core i5-12400F, з переліку доступних материнських плат автоматично виключаються моделі на сокетах AMD або застарілих платформах Intel, залишаючи лише сумісні варіанти (LGA1700). Далі, залежно від обраної плати (підтримка DDR4 або DDR5), фільтрується категорія оперативної пам'яті. Така логіка дозволяє реалізувати базову перевірку сумісності на рівні інтерфейсів та стандартів без залучення складних обчислень.

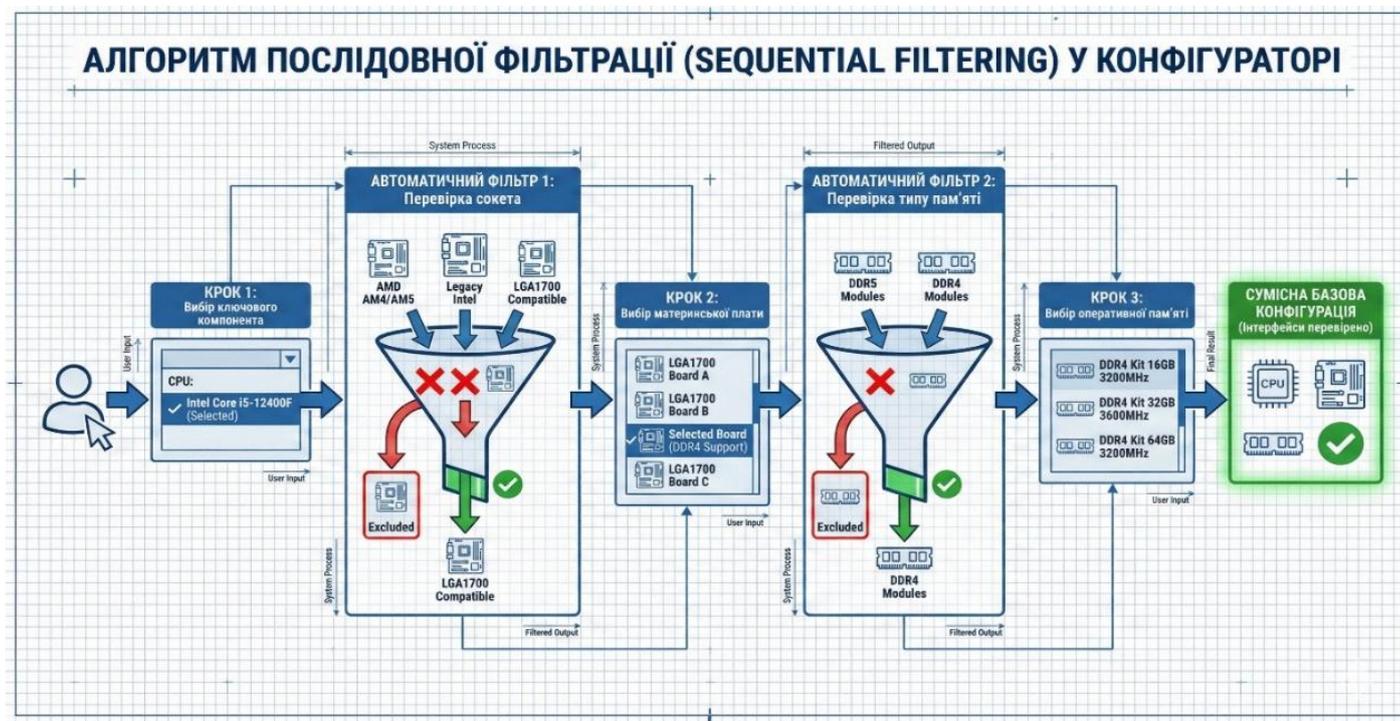


Рис. 1.5 Алгоритм послідовної фільтрації у Telemart.ua

Особливістю та головною перевагою такого підходу є максимальна зручність для пересічного споживача та актуальність даних. Користувач працює з реальними цінами та бачить підсумкову вартість збірки в режимі реального часу. Система інтегрована з сервісними послугами магазину, що дозволяє одразу замовити професійну збірку та тестування обраних компонентів, перекладаючи відповідальність за фізичну сумісність на співробітників магазину. Крім того,

інтерфейс часто містить візуальні підказки та спрощені характеристики, що знижує поріг входження для новачків, які не володіють глибокими технічними знаннями.

Втім, як інструмент для оптимального підбору, дана система має суттєві недоліки. Критичним обмеженням є «асортиментна пастка»: алгоритм здатний запропонувати лише те, що є в наявності на складі продавця. Це унеможливорює створення об'єктивно найкращої конфігурації, якщо ключовий компонент (наприклад, оптимальна за ціною відеокарта) відсутній у продажу саме в цьому магазині. Також у подібних системах повністю відсутній режим модернізації (апгрейду): користувач не може вказати наявні у нього компоненти (наприклад, старий корпус чи блок живлення) та перевірити їх сумісність з новими деталями. Нарешті, перевірка сумісності часто є поверхневою і не враховує фізичних габаритів (висота кулера, довжина відеокарти) або складних параметрів живлення (VRM, пікові струми), що робить інструмент недостатнім для професійного використання.

1.2.1.2 be quiet! PSU Calculator

Представником класу спеціалізованих інженерних інструментів є онлайн-калькулятор від відомого німецького виробника компонентів be quiet! (рис. 1.6). На відміну від повноцінних конфігураторів, цей сервіс не ставить за мету зібрати комп'ютер "під ключ". Його основне призначення — вирішення однієї конкретної, але критично важливої технічної задачі: точного розрахунку енергетичного бюджету системи та підбору відповідного блоку живлення (БЖ). Такий інструмент орієнтований на користувачів, які вже визначилися з основними компонентами (процесором та відеокартою) і потребують професійної верифікації стабільності електроживлення.

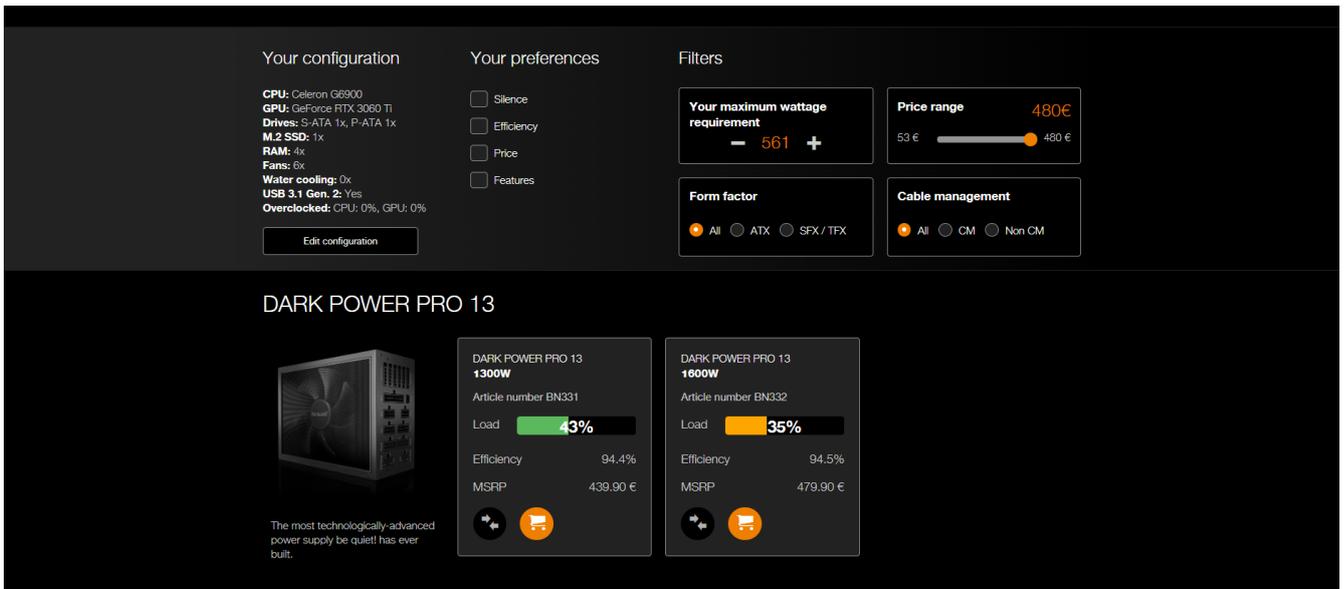


Рис. 1.6 Конфігуратор be quiet! PSU Calculator



Рис. 1.7 Алгоритм роботи конфігуратора be quiet! PSU Calculator

Алгоритм роботи калькулятора базується на детальному адитивному розрахунку пікового енергоспоживання (Total Design Power — TDP) кожного компонента (рис. 1.7). Інтерфейс пропонує користувачеві заповнити детальну анкету технічних параметрів майбутньої системи: від моделі центрального процесора та графічного прискорювача до кількості жорстких дисків, модулів оперативної пам'яті та навіть корпусних вентиляторів.

На основі введених даних система обчислює максимальну теоретичну потужність, яку може споживати комп'ютер під навантаженням, і пропонує список рекомендованих блоків живлення власного виробництва, ранжуючи їх за рівнем ефективності (сертифікати 80 PLUS) та акустичного комфорту.

Ключовою особливістю та перевагою цього інструменту є глибока інженерна точність, недосяжна для більшості універсальних магазинних конфігураторів. Алгоритм враховує не лише номінальні паспортні дані комплектуючих, а й приховані коефіцієнти: пікові стрибки напруги (transient spikes), характерні для сучасних відеокарт, та можливість ручного розгону (overclocking) процесора. Користувач може вказати, що планує "розганяти" систему, і калькулятор автоматично додасть необхідний запас потужності, гарантуючи стабільність роботи в екстремальних режимах. Це робить інструмент незамінним для ентузіастів, які прагнуть уникнути проблем з нестачею живлення.

Проте, у контексті комплексного проектування ПК, даний сервіс має суттєві обмеження концептуального характеру. По-перше, він є маркетинговим інструментом одного вендора (Vendor-lock), тому рекомендує виключно продукцію бренду be quiet!, ігноруючи альтернативи від інших виробників. По-друге, його функціонал є фрагментарним: інструмент вирішує проблему електричної сумісності, але повністю ігнорує фізичну та логічну. Він не попередить користувача, якщо обраний потужний блок живлення фізично не вміститься у компактний корпус, або якщо у блока не вистачить довжини кабелів для конкретної материнської плати. Таким чином, цей калькулятор є потужним допоміжним засобом, але не може слугувати єдиною точкою входу для підбору комп'ютера.

1.2.1.3 E-Katalog

В окрему категорію інструментів підбору слід виділити незалежні прайс-агрегатори, еталонним прикладом яких на ринку Східної Європи є сервіс E-Katalog (рис. 1.8). Ця платформа не є інтернет-магазином у класичному

розумінні; вона виступає глобальним інформаційним посередником, що консолідує пропозиції від сотень різних ритейлерів. Головна функція сервісу полягає не в продажу товару, а в наданні користувачеві вичерпної аналітичної інформації для прийняття зваженого рішення про покупку. Сервіс дозволяє порівнювати ціни на ідентичні товари в різних магазинах, відстежувати динаміку вартості та детально вивчати технічні характеристики.

Каталог / Комп'ютерна техніка / Комплектуючі

Комплектуючі



Материнські плати ¹⁹⁰⁹

micro-ATX • ATX • AMD AM4 • AMD AM5 • Intel 1200 • Intel 1700 • Intel 1851 • Asus • Gigabyte • MSI

Оперативна пам'ять ⁷²⁸³

16 ГБ (2x8) • 32 ГБ (2x16) • 64 ГБ (2x32) • DDR4 • DDR5 • 3200 МГц • 4800 МГц • 5600 МГц

Блоки живлення ⁵⁰²³

650 Вт • 750 Вт • 850 Вт • 1 кВт • 80+ Gold • ATX 3.0 • ATX 3.1 • модульні • Chieftec • Gamemax • MSI



Процесори ¹⁵²³

AMD AM4 • AMD AM5 • Intel 1200 • Intel 1700 • Intel 1851 • Intel Core i5 / Core 5 • Intel Core i7 • AMD Ryzen 5

Системи охолодження ⁸⁸²⁴

вентилятор • вежевий кулер • водяне охолодження • AMD AM5 • Intel 1700 / 1851 • ID-COOLING

Акcesуари для систем охолодження ⁸¹¹

Термопасти та



Відеокарти ¹⁶⁰⁶

GeForce RTX 4060 • GeForce RTX 4070 • GeForce RTX 5070 • Radeon RX 7600 • Radeon RX 7900 XT • Radeon RX 9070 XT

Звукові карти ³⁵⁷

зовнішня • внутрішня • звичайна • геймерська • ЦАП • Creative • Ugreen • Focusrite • Vention • ATCOM

PCI-контролери ¹⁹⁷⁰

PCI-E 1x • PCI-E 2x • PCI-E 4x • PCI-E 8x • PCI • Prime • Dynamode • Gembird • Maiwo • ATCOM



SSD-накопичувачі ⁹¹⁶¹

≈ 1 ТБ • ≈ 2 ТБ • M.2 • SATA 3 • PCI-E 3.0 4x • PCI-E 4.0 4x • USB C 3.2 gen2 • USB C 3.2 gen2x2 • Samsung • Kingston

Жорсткі диски ⁸⁷⁰⁹

1 ТБ • 2 ТБ • 3 ТБ • 4 ТБ • 6 ТБ • 8 ТБ • WD • Seagate • Toshiba • Apacer • Transcend

Кишені для накопичувачів ⁴⁰⁰⁵

зовнішній • вбудований • док-станція • 2.5" • 3.5" • M.2 SSD • Ugreen • Maiwo • Gembird • Dynamode

✓ ПОРІВНЯННЯ ♥ ЗАКЛАДКИ ⌚

Рис. 1.8 Метод підбору в E-Katalog

Принцип роботи агрегатора базується на системі глибокого параметричного пошуку (рис.1.9). База даних E-Katalog містить уніфіковані картки товарів із деталізованим описом специфікацій. Користувач має доступ до розвиненої системи фільтрів, що дозволяє здійснювати надзвичайно точний відбір компонентів за десятками технічних критеріїв (наприклад, шукати відеокарту не просто за моделлю чіпа, а за довжиною плати, кількістю вентиляторів, частотою ядра чи наявністю підсвічування). Окрім цього, сервіс надає інструментарій для прямого порівняння кількох моделей у вигляді зведеної таблиці, де наочно підсвічуються відмінності в характеристиках, що значно спрощує вибір між схожими альтернативами.



Рис. 1.9 Принцип роботи агрегатора в E-Katalog

Беззаперечною перевагою такого підходу є інформаційна нейтральність та економічна ефективність. Оскільки агрегатор не зацікавлений у продажу залишків конкретного складу, він надає об'єктивну картину ринку, дозволяючи користувачеві знайти найнижчу ціну на бажаний компонент. Наявність графіків динаміки цін дозволяє оцінити, чи є поточна вартість вигідною, чи краще зачекати. Додаткову цінність створює активна спільнота користувачів: розділ відгуків та обговорень часто містить інформацію про реальні нюанси експлуатації (наприклад, про свист дроселів або реальні температури), яку виробники зазвичай не вказують у специфікаціях.

Однак, як засіб для комплексного конфігурування ПК, прайс-агрегатори мають критичний недолік — фрагментарність процесу підбору. Користувач змушений обирати кожен компонент ізольовано, без системної перевірки їх взаємної сумісності. Сервіс не попередить, якщо обрана материнська плата не підтримує частоту обраної оперативної пам'яті, або якщо процесорний кулер перекриває слоти розширення. Відсутність наскрізної валідації перекладає всю відповідальність за технічну цілісність збірки на користувача, перетворюючи агрегатор на потужний інструмент для професіоналів, але роблячи його

ризикованим для новачків, які намагаються зібрати систему «з нуля» без глибоких знань.

1.2.1.4 ChatGPT

Останнім і найбільш інноваційним класом інструментів, що все частіше використовуються для підбору техніки, є універсальні чат-боти на базі великих мовних моделей, серед яких безумовним лідером є ChatGPT від OpenAI (рис. 1.10). На відміну від попередніх розглянутих сервісів, це не спеціалізоване інженерне програмне забезпечення, а генеративний штучний інтелект загального призначення. Він не оперує жорстко заданими базами даних товарів, а генерує відповіді на основі величезного масиву текстової інформації, на якій був навчений.

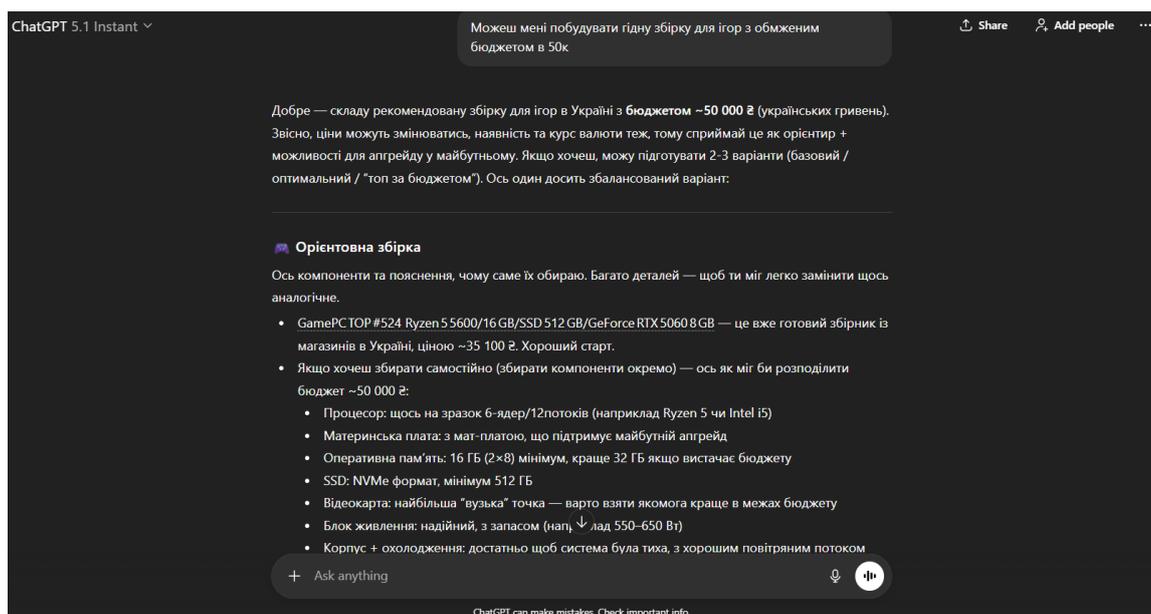


Рис. 1.10 Метод підбору в ChatGpt

Принцип роботи такого "конфігуратора" докорінно відрізняється від традиційних алгоритмів (рис. 1.11). Замість вибору фільтрів та кнопок, взаємодія відбувається у форматі природного діалогу. Користувач формулює запит у довільній формі (наприклад, «Порадь ігровий комп'ютер за 30 тисяч гривень, щоб йшов Cyberpunk 2077»). Модель аналізує семантичний зміст запиту, виокремлює ключові наміри (бюджет, призначення) і, використовуючи

ймовірнісні механізми передбачення тексту, генерує список рекомендованих комплектуючих разом з аргументацією вибору.

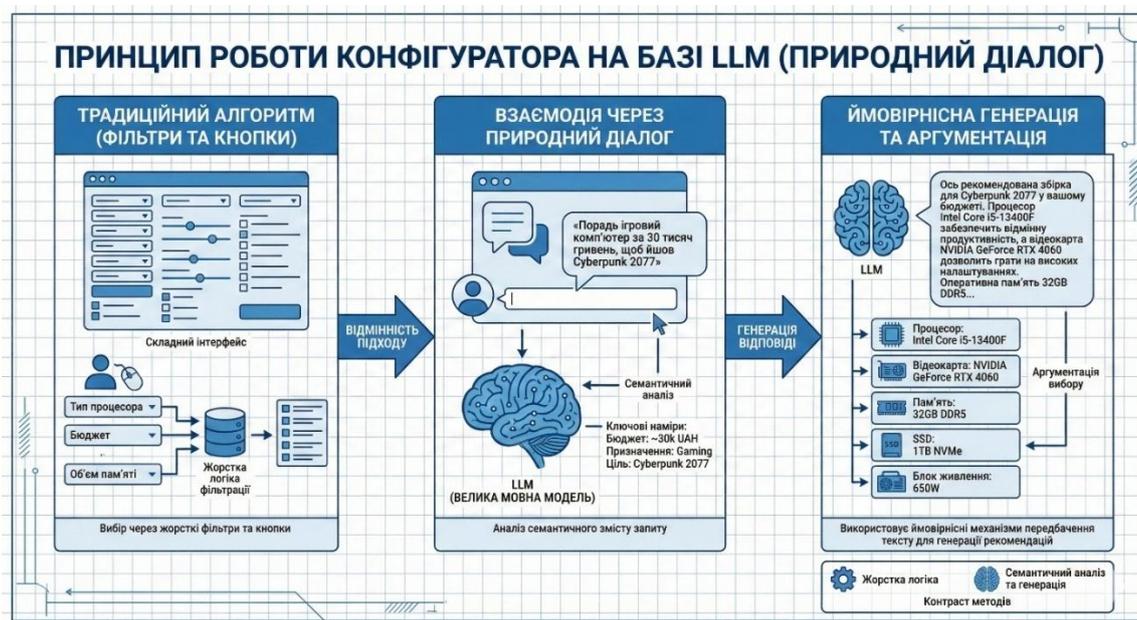


Рис. 1.11 Принцип роботи конфігуратора на базі LLM в ChatGpt

Головною особливістю та перевагою цього підходу є радикальне зниження порогу входження. Користувачеві не потрібно розумітися на сокетах, чіпсетах чи типах пам'яті — він може спілкуватися з системою як з живим консультантом. LLM здатна пояснити складні технічні нюанси простою мовою, запропонувати альтернативи та навіть надати поради щодо процесу фізичної збірки. Це забезпечує неперевершену гнучкість, недоступну для жорстких алгоритмічних систем.

Однак, використання LLM для точних інженерних задач несе в собі критичні ризики. Найбільшим недоліком є феномен «галюцинацій»: модель може впевнено рекомендувати неіснуючі моделі компонентів, плутати характеристики (наприклад, приписати підтримку DDR5 платі, що працює тільки з DDR4) або вказувати ціни, які були актуальні три роки тому (через обмеження дати навчання Knowledge Cutoff). Крім того, чат-бот не проводить реальних математичних розрахунків сумісності — він лише імітує експертну думку. Відсутність гарантії фізичної та електричної сумісності робить такі поради

небезпечними для недосвідченого користувача, який ризикує придбати неробочий набір деталей.

1.2.2 Аналіз метод підбору комплектуючих, що використовуються на практиці

При розробці системи автоматизованого конфігурування критично важливо обрати методологічну базу, яка б дозволяла ефективно вирішувати задачу багатокритеріальної оптимізації (ціна/продуктивність/сумісність). На практиці застосовуються чотири основні групи методів, кожна з яких має специфічні переваги та обмеження.

1.2.2.1 Метод евристичного підбору

Історично першим і досі найбільш поширеним підходом до конфігурування комп'ютерних систем є евристичний метод (рис. 1.12), який базується на когнітивних здібностях людини-експерта. Сутність цього методу полягає у прийнятті рішень на основі неформалізованих емпіричних правил (евристик), сформованих на базі попереднього досвіду та інтуїції. У контексті підбору ПК цей процес не передбачає виконання точних математичних розрахунків для кожної комбінації. Натомість, експерт (або досвідчений користувач) оперує узагальненими патернами, наприклад: «для ігрової системи відеокарта повинна складати близько 40-50% бюджету», «для стабільної роботи краще обирати материнські плати на чіпсетах середнього рівня» або «блоків живлення певного бренду слід уникати через високий відсоток браку».

Беззаперечною перевагою даного підходу є його висока гнучкість та здатність враховувати нечислові, суб'єктивні фактори, які важко піддаються математичній формалізації. Людина здатна оцінити естетичну сумісність компонентів (дизайн, колірна гама, наявність підсвічування), акустичний комфорт, лояльність замовника до певних брендів або специфічні вимоги до ергономіки збірки. Евристичний метод дозволяє знаходити нестандартні рішення в умовах невизначеності, коли вимоги користувача є розмитими або

суперечливими, адаптуючи конфігурацію під унікальні сценарії використання, що виходять за рамки стандартних шаблонів.



Рис. 1.12 Діаграма діяльності для 'Евристичного методу'

Однак, як базис для автоматизованої системи масового обслуговування, цей метод має критичні недоліки. Головним з них є високий рівень суб'єктивізму та неможливість масштабування: якість підбору напряму залежить від кваліфікації конкретного експерта, а його пропускна здатність фізично обмежена. Людина не здатна за прийнятний час проаналізувати весь простір можливих рішень (який може складати мільйони комбінацій), тому вибір завжди обмежується вузьким колом знайомих моделей ("local optimum"). Крім того, евристичний підхід не гарантує математичної оптимальності використання бюджету і схильний до когнітивних викривлень, коли перевага надається не об'єктивно кращим, а більш розрекламованим або звичним компонентам.

1.2.2.2 Метод параметричної фільтрації

Альтернативою суб'єктивному евристичному підходу став метод параметричної фільтрації (рис. 1.13), який є детермінованим алгоритмом і на сьогоднішній день лежить в основі абсолютної більшості комерційних онлайн-

конфігураторів (таких як інструменти на сайтах Rozetka, Telemart тощо). Сутність цього методу полягає у послідовному звуженні простору пошуку шляхом накладання набору жорстких логічних правил типу «ЯКЩО-ТО». Процес конфігурування реалізується як ланцюжок залежних кроків: вибір компонента на поточному етапі автоматично формує обмежувальний фільтр для наступного етапу. Наприклад, якщо користувач обирає процесор Intel Core i5-12400F, система активує правило WHERE Socket == 'LGA1700', приховуючи з каталогу всі материнські плати, що не відповідають цьому критерію. Аналогічно, вибір корпусу форм-фактору Mini-ITX активує фільтр, що приховує повнорозмірні плати ATX.

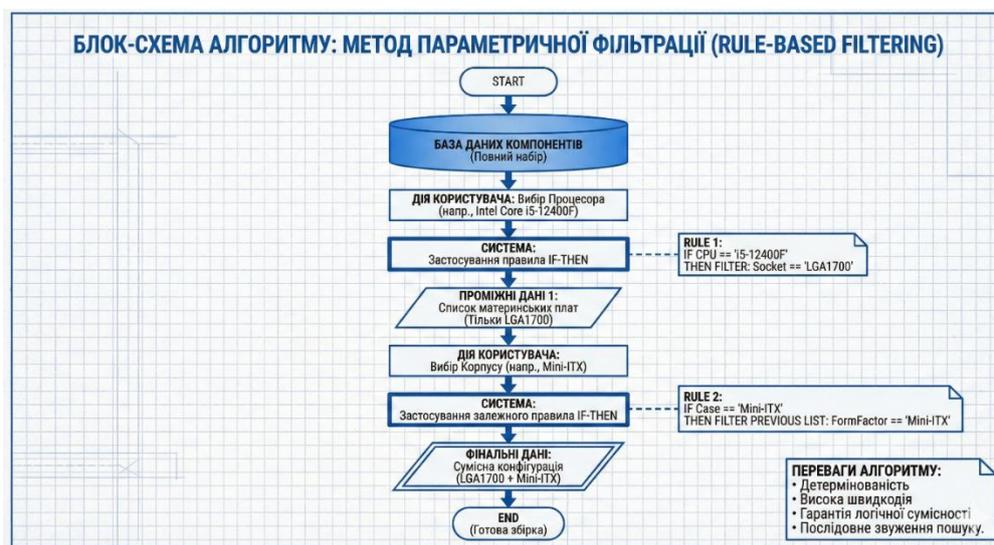


Рис. 1.13 Блок схема для методу ‘Параметричної Фільтрації’

Головною перевагою методу фільтрації є його простота в реалізації та гарантія базової логічної сумісності компонентів. Він не вимагає складних обчислень і працює миттєво навіть на великих базах даних, оскільки операція фільтрації за ключовим полем (наприклад, ID сокета) є вкрай швидкою з точки зору баз даних. Для користувача цей метод забезпечує чіткий і зрозумілий шлях: він захищає від найгрубіших помилок, фізично не дозволяючи додати до кошика несумісні деталі. Це робить процес покупки безпечнішим для новачків порівняно з повністю ручним підбором.

Втім, суттєвим недоліком методу є його пасивна природа. Система працює лише як сито, відсіюючи явно неправильні варіанти, але вона не здатна запропонувати найкраще рішення. Метод не вирішує задачу багатокритеріальної оптимізації: він не аналізує співвідношення ціна/продуктивність і не вміє балансувати бюджет. Якщо користувач помилково вибере невиправдано дорогий процесор, система слухняно покаже йому сумісні (і ймовірно дорогі) материнські плати, навіть якщо це призведе до того, що на відеокарту не залишиться коштів. Тобто, метод гарантує, що комп'ютер увімкнеться, але не гарантує, що він буде ефективним за витрачені кошти. Крім того, жорсткі правила часто не враховують граничних випадків (наприклад, фізичних габаритів відеокарти у міліметрах), обмежуючись лише перевіркою стандартів.

1.2.2.3 Статистичний метод

Суттєво відмінним від попередніх підходів є статистичний метод (рис. 1.14), який набув широкого поширення з розвитком технологій Big Data та лежить в основі рекомендаційних систем великих маркетплейсів (таких як Amazon або AliExpress або E-katalog). На відміну від інженерних методів, які аналізують технічні характеристики компонентів, цей підхід базується на аналізі поведінкових патернів користувачів. Сутність методу полягає у виявленні прихованих закономірностей у масивах даних про покупки. Система не знає, що процесор Intel Core i5 підходить до плати на чіпсеті B760 через сумісність сокетів; вона знає це тому, що тисячі інших користувачів придбали ці два товари разом. Алгоритм колаборативної фільтрації формує рекомендації за принципом соціального доказу: «Користувачі, які цікавилися обраною вами відеокартою, також додали до кошика цей блок живлення».

Перевагою статистичного методу є його здатність пропонувати перевірені ринком, народні рішення без необхідності глибокого моделювання технічних зв'язків між компонентами. Він автоматично підсвічує найбільш вдалі за співвідношенням ціна/якість комбінації, які стали популярними серед спільноти. Крім того, такий підхід є самонавчальним. Чим більше користувачів взаємодіє з

системою, тим точнішими стають рекомендації. Це дозволяє виявляти неявні тренди, наприклад, масовий перехід користувачів на певний бренд оперативної пам'яті через його високу надійність, про що неможливо дізнатися із сухих технічних специфікацій.

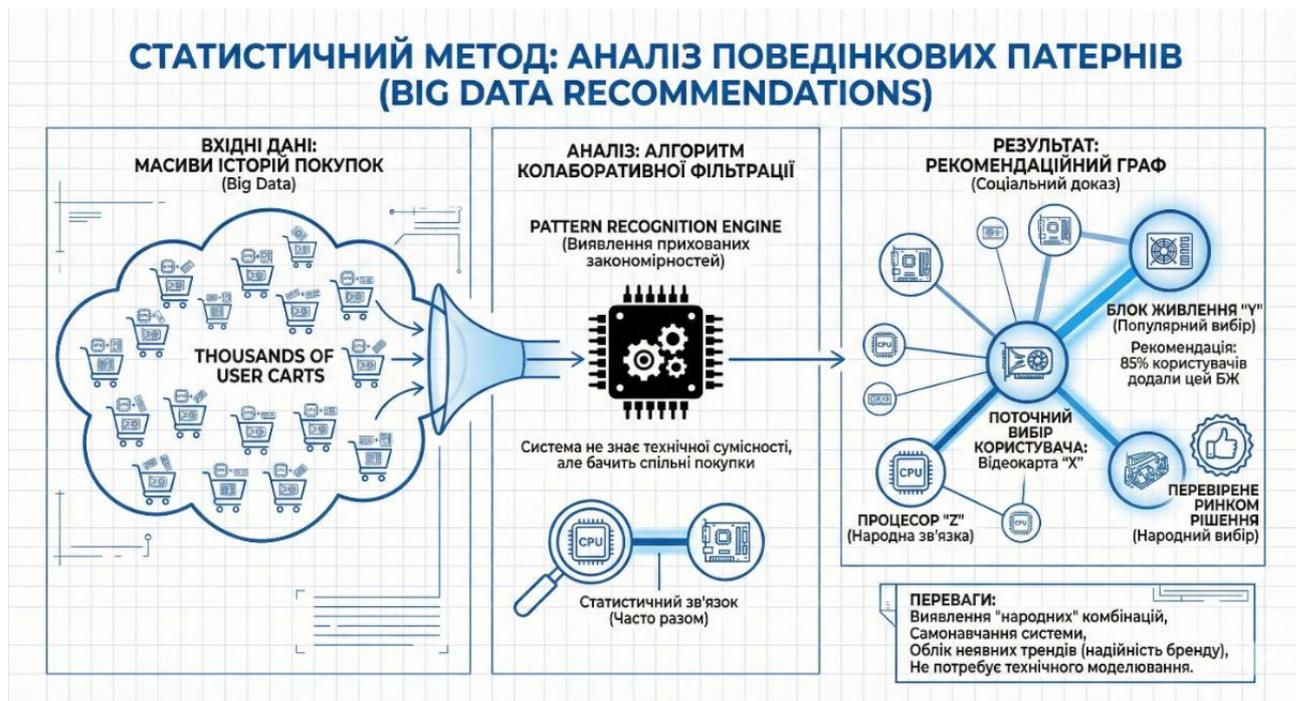


Рис. 1.14 Статистичний метод

Однак, застосування цього методу як основного для інженерного конфігурування несе в собі значні ризики, пов'язані з ефектом «помилки натовпу». Популярність товару не завжди корелює з його технічною досконалістю чи сумісністю з конкретними вимогами замовника. Статистичний алгоритм схильний тиражувати поширені помилки: наприклад, якщо більшість недосвідчених користувачів купує надлишкові блоки живлення або несумісні за габаритами кулери, система почне рекомендувати ці помилкові рішення як норму. Крім того, метод повністю ігнорує індивідуальні обмеження бюджету та специфічні задачі (наприклад, професійний рендеринг), пропонуючи усереднені варіанти, які можуть бути субоптимальними для конкретного випадку. Також він безсилий при появі на ринку нових компонентів, оскільки для них ще не накопичена статистика покупок.

1.2.2.4 Метод комбінаторної оптимізації

Найбільш досконалим та наукоємним підходом до вирішення задачі конфігурування є метод комбінаторної оптимізації (рис. 1.15), реалізований на базі еволюційних алгоритмів, зокрема Генетичного Алгоритму (Genetic Algorithm — GA). Цей підхід розглядає процес підбору комплектуючих не як послідовний вибір товарів, а як складну математичну задачу пошуку глобального екстремуму цільової функції в багатовимірному просторі рішень. Формально це є варіацією класичної задачі про рюкзак, де необхідно сформуувати набір компонентів (хромосому), який максимізує сумарну корисність (продуктивність системи), при цьому задовольняючи систему жорстких обмежень (сумісність, габарити) та не перевищуючи заданий ліміт ресурсів (бюджет). Оскільки простір можливих комбінацій сучасних комплектуючих є астрономічно великим (мільярди варіантів), повний перебір є обчислювально неможливим. Генетичний алгоритм вирішує цю проблему, імітуючи природний процес еволюції, де він створює популяцію випадкових збірок, оцінює їх пристосованість (fitness function), відбирає найкращі та схрещує їх, поступово наближаючись до оптимального результату.

Ключовою перевагою цього методу є його здатність знаходити не просто сумісне, а математично оптимальне рішення. Це єдиний підхід, який дозволяє автоматизувати балансування системи. Алгоритм самостійно розуміє, на чому можна заощадити без втрати якості (наприклад, взяти дешевший корпус або менший SSD), щоб інвестувати вивільнені кошти у критично важливий вузол (потужнішу відеокарту). Він працює як активна система прийняття рішень, здатна знаходити неочевидні, але ефективні комбінації, які людина або простий фільтр могли б пропустити. Крім того, метод дозволяє легко інтегрувати складні сценарії, такі як модернізація, розглядаючи наявні деталі користувача як незмінну частину геному майбутньої системи.

Головним викликом при впровадженні цього методу є висока складність реалізації. Він вимагає розробки детальної математичної моделі предметної області, точного налаштування вагових коефіцієнтів фітнес-функції та значних

обчислювальних ресурсів порівняно з простими фільтрами. Також результат роботи алгоритму є ймовірнісним, що гарантує знаходження непоганого рішення за прийнятний час, але теоретично може пропустити абсолютний ідеал. Втім, в умовах невизначеності ринку та суб'єктивності вимог користувача, саме цей метод забезпечує найкращий баланс між швидкістю пошуку та якістю результату, що робить його безальтернативним вибором для створення інтелектуальної системи підбору ПК.



Рис. 1.15 Діаграма діяльності генетичного алгоритму

1.2.3 Аналіз сучасних підходів ШІ в конфігуруванні

Інтеграція технологій штучного інтелекту (ШІ) у сферу проєктування комп'ютерних систем є одним із найбільш актуальних трендів останніх років. Еволюція підходів у цьому напрямку відбувається від простих статистичних моделей до складних генеративних систем. Аналіз науково-технічної літератури та ринкових рішень дозволяє виділити два домінуючі вектори застосування ШІ:

- 1) рекомендаційні системи на основі машинного навчання;
- 2) інтелектуальні асистенти на базі великих мовних моделей.

1.2.3.1 Рекомендаційні системи

Першим історичним етапом інтеграції штучного інтелекту в процеси електронної комерції стала поява рекомендаційних систем, побудованих на методах класичного машинного навчання. Цей підхід докорінно відрізняється від інженерного проектування, оскільки він працює не з фізичними законами чи специфікаціями, а з соціологією даних[3]. Найбільш поширеним алгоритмом у цій сфері є колаборативна фільтрація (Collaborative Filtering), яка лежить в основі механізмів рекомендацій гігантів на кшталт Amazon, Netflix чи Spotify (рис. 1.16).

Сутність методу ("Магія натовпу") Рекомендаційна система не розуміє, що таке комп'ютер, сокет або вольтаж. Для неї і процесор, і пачка чіпсів — це просто абстрактні об'єкти в базі даних. Алгоритм будує гігантську матрицю взаємодій «Користувач — Товар» і шукає в ній приховані математичні патерни. Він працює за двома основними векторами:

1. Система аналізує кошики мільйонів покупців і знаходить кореляції ("Якщо користувач поклав у кошик процесор AMD Ryzen 5 5600, то з ймовірністю 85% він також придбає материнську плату на чіпсеті B550"). Система не знає, чому вони підходять (сумісність сокетів), вона просто бачить, що їх купують разом.

2. Система знаходить цифрових двійників користувача. ("Користувач А поводить себе так само, як Користувач Б, тому йому, ймовірно, сподобається той самий товар той, що придбав Користувач А").

Для бізнесу цей метод є надзвичайно ефективним інструментом крос-продажів, а саме:

- ШІ може помітити тренди, які неочевидні для інженерів. Наприклад, що певна модель оперативної пам'яті має аномально низький відсоток повернень (тобто є дуже надійною), і почати пропонувати саме її;
- метод чудово працює там, де немає чітких формул, наприклад, у виборі дизайну корпусу, кольору підсвічування або бренду периферії.

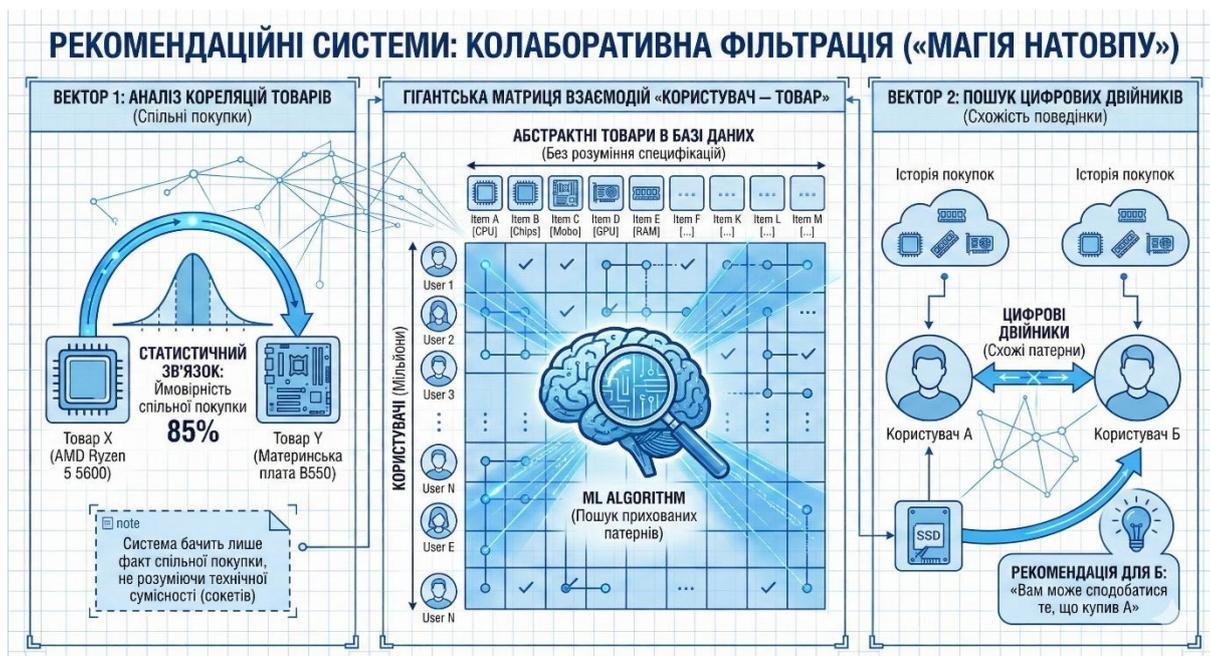


Рис. 1.16 Рекомендаційні системи

Попри успіх у продажу книг чи фільмів, використання колаборативної фільтрації як основного методу для конфігурування ПК є помилковим з ряду критичних причин:

1. Технічна сліпота та ризик несумісності, яка являється головним недоліком, оскільки система спирається лише на статистику покупок, вона може рекомендувати технічно абсурдні речі, якщо тисячі недосвідчених користувачів через незнання купуватимуть дешеві та вибухонебезпечні блоки живлення до дорогих відеокарт, система сприйме це як нормальний патерн і почне рекомендувати цю небезпечну зв'язку іншим. Вона тиражує помилки натовпу замість того, щоб виправляти їх.

2. Індустрія заліза оновлюється дуже швидко, коли виходить нове покоління відеокарт (наприклад, RTX 50-ї серії), для них ще немає накопиченої історії покупок, через що рекомендаційна система буде сліпою до новинок і продовжуватиме пропонувати застарілі моделі, для яких є статистика, що робить її безкорисною для збірки актуального ПК.

3. Такий ШІ не вміє рахувати, тобто він не зможе відповісти на такі прості питання: "Чи вистачить 650 Вт для цієї системи?". Він не зможе перерозподілити бюджет, щоб заощадити на корпусі заради процесора, оскільки

це вимагає розуміння цінності кожного компонента, а не просто його популярності.

Рекомендаційні системи — це чудовий маркетинговий інструмент для магазину ("З цим товаром також купують..."), але вони абсолютно непридатні як інженерний інструмент для проектування працездатної системи. Для задачі точного підбору необхідний детермінований алгоритм (як ГА), який оперує жорсткими правилами фізики та математики, а не думкою натовпу.

1.2.3.2 Генеративні нейронні мережі

Другим, революційним вектором розвитку інтелектуальних систем стало застосування генеративних нейронних мереж (рис. 1.17) на базі архітектури трансформерів (Transformer), представниками яких є моделі GPT-4 (OpenAI), Gemini (Google) та Claude (Anthropic). Цей підхід кардинально змінює парадигму взаємодії «людина-машина»: замість вибору опцій у меню користувач отримує можливість спілкуватися з системою природною мовою, формулюючи запити так, як він робив би це у розмові з живим експертом.

Сутність методу та переваги LLM навчені на колосальних обсягах текстової інформації з інтернету, включаючи технічну документацію, форуми ентузіастів, огляди та прайс-листи[4]. Завдяки цьому вони володіють не лише фактологічними знаннями, а й розумінням контексту:

- користувачеві більше не потрібно знати термінологію, а замість цього він може написати просте повідомлення: "Хочу комп'ютер, щоб грати в Відьмака на високих налаштуваннях і іноді монтувати відео, бюджет 30 тисяч"; модель самостійно перекладе це на мову технічних вимог (GPU рівня RTX 3060, 32 ГБ RAM);
- на відміну від "мовчазних" фільтрів, LLM може аргументувати вибір: "Я обрав цей кулер, тому що процесор гарячий, а цей корпус має погану продувність". Це створює відчуття довіри та навчання користувача;
- модель може легко адаптуватися до нестандартних запитів, наприклад, "підбери тільки білі комплектуючі" або "перевір, чи вистачить мого старого БЖ".



Рис. 1.17 Генеративні нейронні мережі

Попри вражаючі комунікативні здібності, використання "чистих" LLM для інженерних розрахунків несе серйозні ризики:

1. Феномен «Галюцинацій» - це є критичним недоліком, де генеративна модель працює як ймовірнісний механізм продовження тексту, а не як база даних фактів[5]. Вона може впевнено згенерувати назву неіснуючої материнської плати або приписати процесору Intel підтримку сокета AMD, якщо це семантично виглядає правдоподібно.

2. LLM погано справляються з арифметикою, вона може помилитися при підсумовуванні цін компонентів або при розрахунку сумарного TDP системи, що призведе до перевищення бюджету або нестабільної роботи ПК.

3. Знання моделі обмежені датою її останнього навчання, що в результаті може не знати про вихід нових відеокарт або оперувати цінами дворічної давнини, що робить її рекомендації економічно неактуальними.

Проведений аналіз дозволяє стверджувати, що жоден із сучасних підходів ШІ у чистому вигляді не є достатнім для створення надійної системи автоматизованого проектування ПК. Статистичні методи ігнорують інженерну логіку, а генеративні моделі не гарантують точності даних.

Це обґрунтовує необхідність розробки гібридної архітектури, яка б поєднувала комунікативні здібності LLM (для інтерпретації намірів користувача) з детермінованими алгоритмами оптимізації (наприклад, Генетичним алгоритмом) для гарантування технічної валідності та економічної ефективності рішень. Саме такий підхід покладено в основу даної магістерської роботи.

1.3 Аналітичні дані для точного підбору комплектуючих для збірки персонального комп'ютера

В епоху стрімкого розвитку інформаційних технологій персональний комп'ютер (ПК) залишається ключовим інструментом для вирішення широкого спектра задач: від офісного документообігу.

1.3.1 Технічні специфікації та параметри сумісності

Основою бази даних будь-якої системи автоматизованого конфігурування є набір технічних специфікацій компонентів [6]. У контексті розроблюваної методики ці параметри відіграють роль жорстких обмежень. Їх порушення робить функціонування комп'ютерної системи фізично неможливим або небезпечним. Для забезпечення комплексної валідації конфігурації необхідно оперувати трьома рівнями технічних даних.

1. Логічна та інтерфейсна сумісність (цей рівень визначає принципову можливість комунікації компонентів між собою на рівні стандартів та протоколів):

- Процесорний сокет (Socket) - це критичний параметр, що визначає механічну та електричну сумісність центрального процесора (CPU) та материнської плати. Система повинна розрізняти не лише типи роз'ємів (наприклад, Intel LGA1700, AMD AM5), але й покоління підтримуваних архітектур (наприклад, плата на чіпсеті B660 може

вимагати оновлення BIOS для роботи з процесором Intel 14-го покоління).

- Стандарт оперативної пам'яті – сучасні платформи знаходяться на етапі переходу від DDR4 до DDR5. Оскільки ці стандарти є електрично та механічно несумісними (різне розташування ключа-вирізу), алгоритм повинен суворо контролювати відповідність типу обраної пам'яті слотам на материнській платі.
- Інтерфейси накопичувачів - перевірка наявності відповідних портів (M.2 Key M, SATA III) та підтримки протоколів (NVMe, AHCI). Важливим нюансом є врахування кількості доступних ліній PCIe, щоб встановлення кількох накопичувачів не призвело до відключення інших портів.

2. Електрична сумісність (забезпечення стабільного живлення є запорукою надійності системи):

- Тепловий пакет (TDP/TBP) - база даних повинна містити інформацію про номінальне та пікове енергоспоживання (Total Board Power) для процесорів і відеокарт. Просте сумування номінальних значень часто є недостатнім, тому методика передбачає використання коефіцієнтів запасу для врахування транз'єнтних сплесків навантаження.
- Окрім загальної потужності (Вт), критично важливими є параметри сили струму на лінії +12В та наявність необхідного набору конекторів (наприклад, кількість роз'ємів 8-pin CPU для живлення процесора та наявність нового стандарту 12VHPWR для відеокарт NVIDIA RTX 40-ї серії).

3. Геометрична сумісність (цей аспект часто ігнорується в простих конфігураторах, що призводить до проблем на етапі фізичної збірки):

- Габарити відеокарти - перевірка довжини відеокарти (у мм) відносно максимального допуску корпусу, а також її товщини (кількість займаних слотів розширення).
- Висота системи охолодження - перевірка відповідності висоти

процесорного кулера ширині корпусу.

- Форм-фактор - контроль відповідності стандарту материнської плати (E-ATX, ATX, mATX, Mini-ITX) посадковим місцям у корпусі.
- Сумісність радіаторів СРО - для систем рідинного охолодження необхідна інформація про підтримку корпусом радіаторів певного розміру (240мм, 360мм) та можливість їх розміщення (спереду, зверху).

1.3.2 Цінові дані та ринкові позиціонування

Економічна ефективність спроектованої системи напряму залежить від якості та структури цінових даних, якими оперує алгоритм. У контексті задачі оптимізації бюджету вартість компонента розглядається не як статична величина, а як динамічний атрибут, що вимагає багаторівневого аналізу. База даних повинна містити комплекс цінової інформації, починаючи з даних про динаміку та актуальність вартості. Оскільки ринок комп'ютерних комплектуючих характеризується високою волатильністю, ціни можуть змінюватися щоденно під впливом курсових коливань, дефіциту складських запасів або маркетингових акцій. Для коректної роботи функції бюджетних обмежень система повинна оперувати цінами, актуальними на момент запиту (Real-time моніторинг). Використання застарілих прайс-листів призведе до того, що згенерована оптимальна збірка в реальності виявиться недоступною або значно дорожчою за ліміт користувача. Наявність історичних даних дозволяє алгоритму виявляти штучні завищення вартості перед розпродажами або прогнозувати вигідні моменти для покупки.

Критично важливим аспектом є здатність системи розрізняти вартість компонентів у межах лінійок одного виробника. Вендори часто випускають десятки варіацій одного й того ж продукту (наприклад, відеокарти на чіпі RTX 4060) з різним позиціонуванням: від бюджетних моделей з простою системою охолодження до преміальних версій з покращеним VRM, масивними радіаторами та RGB-підсвічуванням, ціна яких може бути на 20–30% вищою.

Методика повинна вміти визначати, чи є доцільною переплата за «преміальність» у рамках обмеженого бюджету. У більшості випадків для економії коштів раціональніше обрати базову версію потужнішого чіпа, ніж топову версію слабшого.

Також необхідно враховувати міжбрендову диференціацію, яка відображає різницю в ціні на аналогічні за характеристиками товари від різних виробників (наприклад, оперативна пам'ять від відомого бренду проти бюджетного аналога). Часто ціна відомих брендів включає значну маркетингову націнку. Алгоритм повинен порівнювати компоненти не за назвою, а за технічними специфікаціями, що дозволяє знаходити «золоту середину» — компоненти, які забезпечують необхідну надійність без переплати. На основі цих даних розраховується похідний інтегральний показник — коефіцієнт (ціна/продуктивність), який є основним критерієм для ранжування кандидатів у генетичному алгоритмі. Система автоматично визначає, скільки балів продуктивності купує користувач за кожен витрачений гривню, що дозволяє об'єктивно порівнювати компоненти різних поколінь і обирати математично вигідніший варіант.

1.3.3 Дані продуктивності (Бенчмарки)

Ключовим елементом для розрахунку фітнес-функції в Генетичному алгоритмі є об'єктивна кількісна оцінка швидкодії кожного компонента. Традиційний підхід, що базується на порівнянні номінальних технічних характеристик (тактової частоти, кількості ядер, обсягу кешу), в сучасних умовах є недостатнім і часто вводить в оману. Через відмінності в мікроархітектурі процесор із меншою частотою може бути значно продуктивнішим за високочастотний аналог попереднього покоління. Аналогічна ситуація спостерігається у відеокартах, де кількість шейдерних блоків (CUDA-ядер) не можна напряму порівнювати між архітектурами NVIDIA та AMD.

Для вирішення цієї проблеми методика спирається на результати стандартизованих синтетичних тестів — бенчмарків. База даних повинна містити уніфіковані бали продуктивності для ключових вузлів системи, а саме:

1. Для CPU. Використовуються результати тестів, що моделюють реальне навантаження, таких як Cinebench R23 або Geekbench. Важливо розділяти оцінку на два вектори: однопотокова продуктивність (критична для ігор та офісних завдань) та багатопотокова продуктивність (визначальна для рендерингу відео, 3D-моделювання та компіляції коду). Це дозволяє алгоритму адаптувати вибір під конкретний профіль користувача.

2. Для GPU. Базовим показником є бали в 3DMark Time Spy (для оцінки продуктивності в DirectX 12) або усереднений показник FPS (кадрів за секунду) у наборі референсних ігор класу AAA при роздільній здатності 1080p/1440p. Такий підхід дозволяє звести відеокарти різних виробників та поколінь до єдиної шкали ефективності.

Використання нормалізованих балів бенчмарків є єдиним способом математично коректно порівнювати компоненти, що дозволяє реалізувати у фітнес-функції механізм зваженої суми, де загальна продуктивність системи визначається не як сума гігагерц, а як сукупний обчислювальний потенціал, виражений у безрозмірних одиницях відносно еталонної системи.

1.3.4 Емпіричні дані та якісні показники

Окрім формалізованих технічних специфікацій та результатів синтетичних тестів, які слугують основою для жорсткої фільтрації та математичної оптимізації, існує клас параметрів, що визначають суб'єктивну якість досвіду користувача. Ці параметри, які в рамках методики класифікуються як м'які обмеження, формуються на основі емпіричних даних - інформації, отриманої дослідним шляхом у процесі реальної експлуатації обладнання.

Для забезпечення високої точності рекомендацій база даних повинна агрегувати наступні категорії емпіричної інформації:

1. Рівень шуму (у децибелах), що генерується системами охолодження під навантаженням. Технічний паспорт зазвичай містить лише максимальні значення обертів вентилятора, що не відображає реальної акустичної картини. Емпіричні дані дозволяють виявити моделі зі специфічними акустичними

дефектами, такими як високочастотний шум дроселів у відеокарт або гудіння помпи рідинного охолодження, і знизити їхній рейтинг для користувачів, що пріоритизують тишу.

2. Дані про ефективність розсіювання тепла в реальних умовах, де вони дозволяють оцінити схильність компонентів до перегріву та тротлінгу (автоматичного зниження частоти для захисту від перегріву). Наприклад, два процесорні кулери можуть мати однаковий заявлений TDP (250 Вт), але на практиці один утримує температуру процесора на рівні 70°C, а інший допускає нагрів до 95°C при однаковому навантаженні.

2 МЕТОДИКА ПІДБОРУ КОМПЛЕКТУЮЧИХ ДЛЯ КОМП'ЮТЕРНИХ СИСТЕМ З УРАХУВАННЯМ БЮДЖЕТНИХ ОБМЕЖЕНЬ ТА НАЯВНОЇ АПАРАТНОЇ КОНФІГУРАЦІЇ

2.1 Математична постановка задачі та формалізація критеріїв оптимізації

Задача автоматизованого підбору комплектуючих для комп'ютерної системи належить до класу задач дискретної комбінаторної оптимізації. Її метою є знаходження такої комбінації апаратних компонентів, яка забезпечує максимальну ефективність (продуктивність) при дотриманні заданих технічних та економічних обмежень.

2.1.1 Формалізація простору рішень

Для застосування методів математичної оптимізації необхідно представити предметну область (набір комп'ютерних компонентів та правила їх поєднання) у вигляді формальної алгебраїчної моделі.

Нехай C — множина всіх доступних на ринку компонентів, які зберігаються в базі даних системи. Ця множина є об'єднанням підмножин, що відповідають категоріям (типам) комплектуючих:

$$C = \bigcup_{i=1}^n C_i = \{C_{cpu}, C_{mb}, C_{gpu}, C_{ram}, C_{ssd}, C_{psu}, C_{case}, C_{cool}\} \quad (2.1)$$

де n — кількість категорій (у базовій моделі $n = 8$), а C_i — множина доступних моделей i -го типу.

Кожна конкретна конфігурація комп'ютера (варіант рішення задачі) формалізується як вектор x , компонентами якого є індекси обраних деталей з кожної категорії:

$$x = (c_1, c_2, \dots, c_n), \quad \text{де } c_i \in C_i \quad (2.2)$$

Тобто, збірка — це впорядкований набір, де на першому місці стоїть ідентифікатор процесора, на другому — материнської плати, і так далі.

Простір всіх можливих рішень X (простір пошуку) визначається як декартовий добуток множин компонентів:

$$X = C_1 \times C_2 \times \dots \times C_n \quad (2.3)$$

Потужність цього простору $|X|$ (загальна кількість можливих комбінацій) дорівнює добутку потужностей кожної категорії:

$$|X| = |C_1| \cdot |C_2| \cdot \dots \cdot |C_n| \quad (2.4)$$

Якщо припустити, що в кожній категорії доступно в середньому по 100 моделей (що є заниженою оцінкою для сучасного ринку), то розмір простору пошуку складатиме:

$$|X| \approx 100^8 = 10^{16} \quad (2.5)$$

Це астрономічно велике число варіантів. Навіть якщо перевірка однієї комбінації на сумісність займає 1 мікросекунду, повний перебір (метод Brute Force) займе сотні років. Це математично доводить неможливість знаходження оптимального рішення шляхом простого перебору і обґрунтовує необхідність використання евристичних методів, зокрема генетичного алгоритму, який дозволяє знайти субоптимальне рішення, дослідивши лише мізерну частку простору X . У рамках задачі модернізації простір пошуку звужується. Якщо користувач фіксує певні компоненти (наприклад, залишає старий корпус та блок живлення), відповідні множини C_i вироджуються до одного елемента, що значно зменшує розмірність задачі, але накладає додаткові жорсткі обмеження на вибір інших компонентів (наприклад, нова відеокарта повинна фізично поміститися у зафіксований старий корпус).

2.1.2 Критерії оптимізації та цільова функція

Задача конфігурування комп'ютерної системи за своєю природою є багатокритеріальною, оскільки користувач прагне одночасно досягти двох конфлікуючих цілей:

- максимізація продуктивності (обчислювальної потужності, швидкості графіки);
- мінімізація витрат (або максимізація ефективності використання фіксованого бюджету).

Для застосування методів багатокритеріальної оптимізації (зокрема класичного генетичного алгоритму) необхідно звести векторний критерій якості до скалярного вигляду.

Глобальна цільова функція (Fitness Function) $F(x)$, значення якої необхідно максимізувати, визначається як зважена сума індексів продуктивності та економічної ефективності [7][8]:

$$F(x) = w_p \cdot P(x) + w_e \cdot E(x) \rightarrow \max \quad (2.6)$$

w_p та w_e — вагові коефіцієнти значущості критеріїв продуктивності та ціни відповідно, причому $w_p + w_e = 1$. Їх значення можуть задаватися користувачем вручну (наприклад, профіль (максимальна економія) або (максимальна потужність)) або визначатися автоматично на основі аналізу запиту.

Індекс продуктивності $P(x)$ є безрозмірною величиною в діапазоні $[0; 1]$, що відображає сукупну обчислювальну потужність системи відносно теоретично можливого максимуму. Він розраховується на основі нормалізованих результатів синтетичних тестів.

Функція (2.7) оцінки продуктивності:

$$P(x) = \alpha_{cpu} \cdot \frac{Score_{cpu}(x)}{S_{cpu}^{max}} + \alpha_{gpu} \cdot \frac{Score_{gpu}(x)}{S_{gpu}^{max}} + \delta_{cool}(x) \quad (2.7)$$

Де:

- 1) $Score_{cpu}(x), Score_{gpu}(x)$ — бали для компонентів, що входять у збірку x .
- 2) $S_{cpu}^{max}, S_{gpu}^{max}$ максимальні значення бенчмарків (наприклад, показники найпотужніших компонентів на ринку), що використовуються для нормалізації. Це дозволяє привести різномірні бали до єдиної шкали.
- 3) $\alpha_{cpu}, \alpha_{gpu}$ — коефіцієнти важливості компонентів залежно від цільового призначення ПК. Приклад:
 - для ігрової системи: $\alpha_{gpu} > \alpha_{cpu}$ (наприклад, 0.6 та 0.4), оскільки відеокарта є пріоритетною.
 - для робочої станції: $\alpha_{cpu} > \alpha_{gpu}$, оскільки важливіша обчислювальна потужність процесора.
- 4) $\delta_{cool}(x)$ — малий бонусний коефіцієнт, що враховує ефективність системи охолодження (стимулює алгоритм обирати кращі кулери для забезпечення стабільності частот).

Тепер розглянемо наступну функція економічної ефективності. Цей критерій оцінює раціональність використання бюджету. Унікальність розробленої моделі полягає в тому, що вона оперує не повною вартістю ПК, а вартістю оновлення, що дозволяє ефективно вирішувати задачі модернізації.

Вартість оновлення $C_{new}(x)$ розраховується як:

$$C_{new}(x) = \sum_{i=1}^n Cost(c_i) - \sum_{j \in Locked} Cost(c_j) \quad (2.8)$$

Тобто, це сума вартості всіх компонентів мінус вартість тих деталей, які користувач вирішив залишити від старої системи.

Функція(2.9) економічної ефективності:

$$E(x) = \begin{cases} 1 - \frac{C_{new}(x)}{B} & \text{якщо } C_{new}(x) \leq B \\ k * \frac{C_{new}(x) - B}{B} & \text{якщо } C_{new}(x) > B \end{cases} \quad (2.9)$$

Де:

B — заданий користувачем бюджет,

k — коефіцієнт жорсткості штрафу за перевищення бюджету (зазвичай $k > 1$).

Фізичний зміст функції $E(x)$:

- якщо вартість збірки дорівнює 0 (теоретично) або значно менша за бюджет, $E(x) \rightarrow 1$.
- якщо вартість дорівнює бюджету ($C_{new} = B$), $E(x) = 0$.
- якщо вартість перевищує бюджет, функція стає стрімко від'ємною, знижуючи загальний рейтинг $F(x)$.

Запропонована структура цільової функції дозволяє алгоритму автоматично балансувати систему. Наприклад, якщо заміна процесора на більш потужний дасть приріст продуктивності $\Delta P = 0.1$, але призведе до перевищення бюджету, що зменшить економічну ефективність на $\Delta E = -0.5$, така заміна буде відхилена. І навпаки, якщо незначне перевищення бюджету дасть колосальний приріст продуктивності, така збірка може бути рекомендована як альтернатива.

2.1.3 Система обмежень та умови сумісності

Пошук оптимального рішення відбувається не на всьому просторі варіантів X , а лише на його допустимій підмножині $D \subset X$. Ця підмножина формується шляхом накладання системи жорстких технічних обмежень:

У розробленій математичній моделі обмеження формалізовані через набір бінарних індикаторів порушення $I_k(x)$.

$$I_k(x) = \begin{cases} 1, & \text{якщо обмеження } k \text{ порушено} \\ 0, & \text{якщо обмеження } k \text{ виконано} \end{cases} \quad (2.10)$$

Якщо хоча б один індикатор $I_k(x) = 1$, конфігурація вважається невалідною, і до неї застосовується штрафна санкція M_k .

2.1.3.1 Апаратна (Логічна) несумісність ($I_{socket}, I_{chipset}$)

Це базовий рівень перевірки, що визначає принципову можливість спільної роботи компонентів.

1. Сумісність процесора та материнської плати (Алгоритм перевіряє відповідність фізичного роз'єму (наприклад, LGA1700) та покоління підтримуваних процесорів):

$$I_{socket}(x) = (Socket_{cpu} \neq Socket_{mb}) \quad (2.11)$$

2. Для деяких комбінацій (наприклад, нові CPU на старих чіпсетах) перевіряється атрибут необхідності оновлення прошивки.

2.1.3.2 Сумісність підсистеми пам'яті (I_{ram})

Перевірка відповідності стандарту оперативної пам'яті слотам на материнській платі. Оскільки стандарти DDR4 та DDR5 є електрично та механічно несумісними (мають різні ключі-вирізи), їх невідповідність є критичною помилкою.

$$I_{ram}(x) = (Type_{ram} \neq Type_{mb}) \quad (2.12)$$

Також перевіряється кількість модулів пам'яті відносно кількості слотів (DIMM slots) на платі (наприклад, неможливість встановити 4 планки у плату з 2 слотами).

2.1.3.3 Енергетична сумісність (I_{power})

Забезпечення стабільності живлення системи. Алгоритм розраховує сумарне пікове енергоспоживання (TDP) всіх активних компонентів.

$$I_{power}(x) = (Power_{psu} < (TDP_{cpu} + TDP_{gpu} + P_{other}) \cdot K_{safe}) \quad (2.13)$$

Де:

P_{other} — константа споживання периферії (накопичувачі, вентилятори, плата), зазвичай приймається $\approx 50-75$ Вт;

K_{safe} — коефіцієнт запасу надійності (зазвичай 1.2 ... 1.3), який гарантує, що БЖ не працюватиме на 100% навантаження, що забезпечує тишу та довговічність.

2.1.3.4 Фізична сумісність (I_{phys})

Перевірка геометричних параметрів для запобігання конфліктам при фізичному складанні.

$$I_{phys}(x) = I_{gpu_len} \vee I_{cool_h} \vee I_{ff} \quad (2.14)$$

Де:

довжина відеокарти: $Length_{gpu} > MaxGPU_{case}$ (відеокарта впирається у передню стінку корпусу або вентилятори);

висота кулера: $Height_{cooler} > axCpuHeight_{case}$ (кришка корпусу не закриється);

форм-фактор: $Size_{mb} > SupportedSize_{case}$ (наприклад, спроба встановити плату ATX у корпус Mini-ITX).

2.1.3.5 Деталізація змінних та атрибутів моделі сумісності

Для реалізації логічних перевірок $I_k(x)$ у математичній моделі використовуються наступні змінні, що відповідають технічним характеристикам компонентів:

1. Атрибути логічної сумісності:

- $Socket_{cpu}$ — тип роз'єму центрального процесора (наприклад, 'LGA1700', 'AM5'). Визначає фізичний інтерфейс підключення.
- $Socket_{mb}$ — тип процесорного роз'єму на материнській платі.
- $Type_{ram}$ — стандарт (покоління) оперативної пам'яті, підтримуваний контролером процесора та модулем пам'яті (наприклад, 'DDR4', 'DDR5').
- $Type_{mb}$ — стандарт слотів оперативної пам'яті на материнській

платі.

2. Атрибути енергетичної сумісності:

- TDP_{cpu} (Thermal Design Power) — номінальна величина тепловідводу процесора (Вт), що використовується як базовий показник енергоспоживання.
- TDP_{gpu} — енергоспоживання відеокарти (Вт). Часто використовується значення TBP (Total Board Power).
- $Power_{psu}$ — номінальна потужність блоку живлення (Вт).
- P_{other} — константа, що апроксимує споживання інших компонентів системи (материнська плата, накопичувачі, вентилятори), зазвичай приймається рівною 50–75 Вт.
- K_{safe} — коефіцієнт запасу надійності (наприклад, \$1.25\$), який гарантує, що блок живлення не працюватиме на межі можливостей.

3. Атрибути фізичної/габаритної сумісності:

- $Length_{gpu}$ — фізична довжина відеокарти.
- $MaxGPU_{case}$ — максимальна довжина відеокарти, яку вміщує корпус (вказана в специфікації корпусу).
- $Height_{cooler}$ — висота процесорного кулера.
- $MaxCpuHeight_{case}$ — максимальна висота кулера, допустима корпусом (відстань від процесора до бічної стінки).
- $Size_{mb}$ — форм-фактор материнської плати, представлений як порядкове значення (наприклад: $MiniITX = 1, mATX = 2, ATX = 3, EATX = 4$).
- $SupportedSize_{case}$ — максимальний форм-фактор, підтримуваний корпусом.

2.1.4 Модифікація цільової функції методом штрафів

Класичний генетичний алгоритм призначений для розв'язання задач безумовної оптимізації. Однак задача підбору комплектуючих містить ряд

критичних умов (сумісність, бюджет), порушення яких є неприпустимим. Для адаптації алгоритму до роботи з обмеженнями в даній методиці застосовано метод штрафних функцій.

Сутність методу полягає в перетворенні вихідної задачі умовної оптимізації на задачу безумовної оптимізації шляхом введення до складу цільової функції спеціального від'ємного доданка — штрафу.

Модифікована фітнес-функція $F^*(x)$, яка використовується для оцінки пристосованості кожної хромосоми, має вигляд:

$$F^*(x) = F_{base}(x) - P_{total}(x) \quad (2.15)$$

Де:

$F_{base}(x) = w_p \cdot P(x) + w_e \cdot E(x)$ — базова оцінка якості конфігурації (без урахування технічних помилок).

$P_{total}(x)$ — сумарна величина штрафу.

У розробленій методиці реалізовано дворівневу систему штрафів, що відповідає поділу обмежень на м'які та жорсткі.

Динамічні штрафи для м'яких обмежень. Використовуються для бюджетних обмежень. Якщо вартість збірки перевищує бюджет, штраф зростає пропорційно величині переплати (як було описано у функції $E(x)$). Мета її, це дозволити алгоритму тимчасово зберігати в популяції трохи дорожчі, але перспективні варіанти, які в процесі еволюції можуть стати дешевшими (наприклад, після заміни корпусу).

Заборонні штрафи для жорстких обмежень. Використовуються для технічних обмежень сумісності (I_k). Тут застосовано метод «смертельного штрафу».

Формула (2.16) розрахунку штрафу:

$$P_{hard}(x) = \sum_k M_k \cdot I_k(x) \quad (2.16)$$

де M_k — константа штрафу, яка обирається з умови:

$$M_k \gg \max_{x \in X} F_{base}(x) \quad (2.17)$$

Якщо максимальна теоретична оцінка ідеальної збірки дорівнює, наприклад, 1.0, то величина штрафу M_k встановлюється на рівні 10.0 або 100.0. У результаті, якщо збірка має хоча б одну несумісність (наприклад, $I_{socket} = 1$), її фінальний рейтинг стає від'ємним.

Особини з від'ємним рейтингом мають нульову ймовірність виживання при турнірній селекції. Вони не передають свої дефектні гени наступному поколінню. Таким чином, популяція автоматично очищується від нежиттєздатних рішень вже на ранніх етапах еволюції, що гарантує 100% валідність результату, який отримує користувач.

Запропонована математична модель, що включає формалізацію простору рішень, адитивну цільову функцію та систему заборонних штрафів, створює необхідний фундамент для програмної реалізації алгоритму. Вона дозволяє автоматизувати процес прийняття рішень, перекладаючи інженерну логіку перевірки сумісності на мову математичних операцій.

2.2 Обґрунтування та опис застосованих алгоритмів

Для реалізації розробленої методики обрано гібридну алгоритмічну базу, що поєднує еволюційні обчислення для вирішення задачі комбінаторної оптимізації та технології обробки природної мови для забезпечення людино-машинної взаємодії [9][10].

2.2.1 Генетичний алгоритм як метод пошуку оптимальної конфігурації

Оскільки простір пошуку можливих комбінацій комп'ютерних компонентів є дискретним, багатовимірним та нелінійним, а загальна кількість

варіантів перевищує 10^{12} , використання класичних градієнтних методів або повного перебору є обчислювально неефективним. Для вирішення цієї задачі в роботі застосовано генетичний алгоритм — стохастичний метод пошуку, що базується на принципах природного відбору та еволюційної біології (рис. 2.1) [11]. В рамках розробленої методики кожна потенційна конфігурація персонального комп'ютера кодується у вигляді хромосоми — вектора цілих чисел фіксованої довжини ($N = 8$), де кожен ген відповідає унікальному ідентифікатору компонента в базі даних (CPU, Motherboard, GPU, RAM, SSD, PSU, Case, Cooler). Таке кодування дозволяє звести задачу підбору до операцій над числовими масивами.

Процес роботи алгоритму починається з етапу ініціалізації популяції [12]. Замість повністю випадкової генерації, яка в умовах жорстких обмежень сумісності призводила б до створення великої кількості нежиттєздатних особин, у роботі застосовано метод евристичної ініціалізації. Алгоритм формує стартові хромосоми послідовно: спочатку обирається процесор, а материнська плата генерується не з повного списку, а лише з підмножини моделей, що мають відповідний сокет. Аналогічно, блок живлення обирається з умовою перевищення мінімального порогу потужності. Такий підхід дозволяє стартувати з популяції, яка вже є частково валідною, що значно прискорює збіжність алгоритму до оптимального рішення.

Оцінка якості кожної особини здійснюється за допомогою фітнес-функції $F^*(x)$, яка розраховує рейтинг збірки на основі її продуктивності та економічної ефективності. На цьому етапі застосовується механізм заборонних штрафів, якщо конфігурація порушує будь-яке з критичних технічних обмежень (наприклад, невідповідність типу оперативної пам'яті), вона отримує значний від'ємний рейтинг і фактично виключається з подальшого репродуктивного процесу. Для відбору батьківських особин використовується метод турнірної селекції з розміром турніру $k = 3$.

Цей метод, на відміну від пропорційного відбору, не вимагає сортування всієї популяції та дозволяє гнучко регулювати тиск відбору, надаючи шанс на

розмноження не лише абсолютним лідерам, а й перспективним локальним рішенням [13].

Генерація нащадків відбувається шляхом застосування операторів кросинговеру та мутації [14]. Для комбінування вдалих рішень використовується одноточковий кросинговер: дві батьківські хромосоми розрізаються у випадковій точці, і нащадок отримує, наприклад, потужну зв'язку процесор-материнська плата від одного батька та якісну підсистему живлення від іншого. Для запобігання передчасній збіжності алгоритму (стагнації в локальному оптимумі) застосовується оператор мутації, який з малою ймовірністю замінює випадковий ген у хромосомі на інший компонент із бази даних. Щоб гарантувати збереження найкращих знайдених рішень, реалізовано стратегію елітаризму, за якої кілька найкращих особин копіюються в наступне покоління без змін. Така архітектура алгоритму забезпечує ефективний баланс між дослідженням простору рішень та покращенням існуючих варіантів.



Рис. 2.1 Діаграма діяльності алгоритму ГА

2.2.2 Механізм фіксації генів для задач модернізації

Однією з ключових проблем сучасних систем автоматизованого конфігурування є їхня орієнтація виключно на сценарій — створення системи з нуля, коли простір рішень обмежений лише бюджетом. Однак на практиці значна частина користувачів стикається із задачею модернізації, коли необхідно покращити характеристики наявного ПК, зберігши частину компонентів (наприклад, якісний корпус, блок живлення або накопичувачі) для економії коштів.

Стандартний генетичний алгоритм не пристосований до таких задач, оскільки його оператори (мутація, кросинговер) прагнуть змінювати всі гени хромосоми для пошуку глобального оптимуму [15]. Для вирішення цієї проблеми в рамках дипломної роботи розроблено та програмно реалізовано модифікацію ГА з механізмом фіксації генів.

Структурна модифікація хромосоми. У класичному ГА хромосома Ch є вектором змінних x_i . У запропонованій методиці кожному гену x_i (що відповідає компоненту типу i) додається мета-атрибут стану L_i (Lock Status):

$$Gene_i = \{ID_i, L_i\} \quad (2.18)$$

Де:

ID_i — ідентифікатор компонента в базі даних.

$L_i \in \{0,1\}$ — булевий прапорець фіксації.

1) Якщо $L_i = 1$, ген вважається константою в межах еволюційного процесу.

2) Якщо $L_i = 0$, ген підлягає оптимізації.

Таким чином, простір пошуку X динамічно звужується до підпростору X' , розмірність якого залежить від кількості незаблокованих компонентів.

Адаптація еволюційних операторів. Впровадження вимагає зміни логіки роботи базових операторів ГА, щоб запобігти руйнуванню зафіксованої конфігурації:

1. Ініціалізація популяції (P_0) відбувається на етапі створення першого покоління алгоритм отримує вхідний вектор U_{exist} (user existing parts). Для кожного типу компонента k , присутнього в U_{exist} , відповідний ген у всіх особинах популяції примусово встановлюється в значення ID_k з атрибутом $L_k = 1$. Варіабельні гени ($L_j = 0$) генеруються випадковим чином.

2. Модифікований оператор мутації, яка є стандартною мутацією, що обирає випадковий ген для заміни. У розробленій методиці введено перевірку стану, що гарантує, що алгоритм ніколи випадково не замінить, наприклад, наявний у користувача корпус на новий, навіть якщо новий кращий за характеристиками. Алгоритм змушений шукати оптимум навколо старого корпусу.

3. Кросинговер, оскільки зафіксовані гени є ідентичними у всіх батьківських особин, кросинговер на цих позиціях не створює генетичного різноманіття, але коректно транслює обмеження нащадкам. Це забезпечує стабільність ознак, які користувач вирішив залишити.

Контекстно-залежна валідація сумісності. Зафіксовані гени виступають джерелом жорстких обмежень для варіабельних генів. Фітнес-функція оцінює нові деталі не у вакуумі, а в контексті старих. Наведемо приклади:

- **Коли габаритний конфлікт.** Користувач зафіксував старий корпус ($L_{case} = 1$). Алгоритм підбирає нову відеокарту ($L_{gpu} = 0$). Якщо ГА обере потужну трикулерну карту довжиною 330 мм, а старий корпус підтримує лише 300 мм, спрацює штрафна функція I_{phys} . В результаті, алгоритм еволюційно навчиться обирати лише компактні відеокарти, що підходять до старого корпусу.
- **Енергетичний ліміт.** Зафіксовано старий блок живлення на 500 Вт ($L_{psu} = 1$). Алгоритм намагається підібрати пару CPU + GPU. Будь-яка комбінація, сумарний TDP якої перевищує поріг надійності (наприклад, 400-450 Вт), отримає штраф. В результаті, система автоматично знайде найбільш продуктивну комбінацію в межах енергетичного бюджету старого БЖ, не пропонуючи користувачеві

купувати новий блок, якщо це не критично.

Оптимізація маржинальних витрат. Використання механізму фіксацію генів змінює сутність цільової функції економічної ефективності $E(x)$. Замість мінімізації повної вартості ПК (C_{total}), алгоритм мінімізує вартість апгрейду ($C_{upgrade}$):

$$C_{upgrade} = \sum (Price_j \cdot (1 - L_j)) \quad (2.19)$$

Тобто ціна зафіксованих компонентів ($L_j = 1$) не враховується у витратах. Це дозволяє системі знаходити рішення з максимальним приростом продуктивності на кожен додатково витрачену гривню, що є математичним вираженням суті ефективної модернізації.



Рис 2.2 Механізм фіксації генів у ГА

Запропонований механізм (механізм фіксації генів) трансформує задачу безумовної оптимізації в задачу оптимізації з динамічними обмеженнями, дозволяючи автоматизувати складний інженерний процес підбору сумісних компонентів для часткового оновлення системи (рис.2.2).

2.2.3 Інтеграція великих мовних моделей (LLM) для аналізу вимог та генерації пояснень

Традиційні інтерфейси систем конфігурування, що базуються на розгалужених системах фільтрів та випадючих списків, створюють високий когнітивний бар'єр для невідготовленого користувача. Для вирішення цієї проблеми в розробленій методиці застосовано гібридну нейро-символьну архітектуру, яка інтегрує можливості генеративного штучного інтелекту (на базі Google Gemini API) у контур прийняття рішень.

Роль LLM у системі суворо регламентована і обмежується задачами обробки природної мови (NLP), тоді як задачі верифікації та оптимізації делеговані детермінованим алгоритмам. Процес інтеграції реалізується на двох етапах:

2.2.3.1 Процес обробки запиту

Традиційні інтерфейси систем підбору комплектуючих базуються на жорстко структурованих формах введення даних (checkboxes, sliders, dropdown lists). Такий підхід вимагає від користувача високого рівня компетенції: він повинен розуміти, що таке чіпсет, форм-фактор або тип пам'яті, щоб коректно заповнити форму. Це створює значний когнітивний бар'єр.

Для вирішення цієї проблеми в розробленій методиці застосовано підхід Natural Language Processing (NLP) на базі великої мовної моделі (LLM). Система дозволяє користувачеві формулювати запит у вільній формі, імітуючи діалог з живим консультантом.

Процес обробки запиту складається з трьох етапів (рис. 2.3) :

1. Розпізнавання іменованих сутностей - на цьому етапі модель сканує текст користувача для виявлення ключових параметрів, необхідних для роботи алгоритму оптимізації, а саме:

- **Бюджет.** Вилучення числових значень та валюти, де модель здатна розпізнавати непрямі вказівки (наприклад, "в межах 40 тисяч" або "до 1000 доларів").

- **Цілі.** Класифікація намірів користувача, к прикладу у нас такий запит: "хочу грати в Кіберпанк". Який класифікується як ігрова збірка, а "для монтажу відео" — як робоча збірка для даної сфери. Це є критично важливо, оскільки від цього залежать вагові коефіцієнти фітнес-функції (пріоритет GPU або CPU).
- **Компоненти.** Ідентифікація згадок про конкретні бренди або моделі (наприклад, "хочу процесор Intel" або "відеокарта тільки Nvidia").

2. Аналіз контексту модернізації - це є унікальною особливістю методики. Система аналізує запит на наявність згадок про компоненти, які користувач вже має. Наприклад:

- вхідний текст: "...відеокарту залишаю свою, RTX 3060, і корпус теж старий".
- дія: модель ідентифікує сутності "RTX 3060" (GPU) та "старий корпус" (Case) і позначає їх як заблоковані для механізму.

3. Формалізація специфікації є результатом роботи NLP-модуля є перетворення неструктурованого тексту в суворий JSON-об'єкт, який передається в генетичний алгоритм. Приклад трансформації:

- запит: "Збери ігровий комп за 30к, щоб було тихо."
- результат: { "budget": 30000, "type": "gaming", "preferences": ["silent"] }.

Така інтеграція дозволяє системі розуміти нечіткі, сленгові або неповні запити, автоматично доповнюючи їх значеннями за замовчуванням (наприклад, якщо бюджет не вказано, система може запитати уточнення або запропонувати варіанти в різних цінових категоріях). Це робить складний інженерний інструмент доступним для масового користувача.



Рис 2.3 Процес обробки запиту

2.2.3.2 Генерація пояснень та аргументація рішень

Однією з головних проблем класичних алгоритмів оптимізації є їхня непрозорість для кінцевого користувача (ефект чорної скриньки). Система видає математично ідеальний результат, але не пояснює, чому саме цей набір компонентів є найкращим. Для підвищення довіри до системи та покращення користувацького досвіду в методику інтегровано модуль пояснюваного штучного інтелекту на базі LLM.

Цей етап активується після завершення роботи генетичного алгоритму. На вхід модуля подається структурований набір даних:

1. Результат оптимізації: список компонентів переможної збірки (назви, ціни, характеристики).
2. Контекст запиту: початкові вимоги користувача (бюджет, мета, побажання).
3. Технічні метрики: розрахункова продуктивність системи та залишок бюджету.

Функціональні задачі модуля генерації (рис. 2.4):

1. Модель генерує нарративний опис, що пов'язує технічні характеристики

з потребами користувача. Замість сухого переліку параметрів, система формує причинно-наслідкові зв'язки. Приклад: «Оскільки вашою метою є ігри в 1440p, алгоритм виділив 45% бюджету на відеокарту RTX 4070. Щоб компенсувати витрати і вкластися в ліміт, було обрано процесор i5-12400F, який є достатнім для цієї карти і не створює ефекту "пляшкового горла"».

2. Будь-яка оптимізація в умовах обмеженого бюджету — це пошук компромісів. Система має чесно повідомити користувачеві, на чому довелося заощадити. Приклад: «Для досягнення максимальної продуктивності було обрано корпус із базовою вентиляцією без RGB-підсвічування. Це дозволило перенаправити кошти на швидший SSD-накопичувач».

3. Рекомендації щодо подальшого апгрейду, аналізуючи вузькі місця поточної збірки, LLM може надати поради на майбутнє. Приклад: «Материнська плата має 4 слоти для ОЗП, з яких зайнято лише 2. У майбутньому ви зможете легко подвоїти обсяг пам'яті до 32 ГБ без заміни існуючих модулів».

Також є критично важливим аспектом, що варто врахувати в розробленій методиці, де є архітектурне розмежування відповідальності, що запобігає виникненню галюцинацій (фактологічних помилок), властивих генеративним моделям, а саме:

- LLM не обирає компоненти, бо вона не має права змінити склад збірки, підбраної генетичним алгоритмом;
- LLM працює як технічний письменник, тобто її задача — лише літературно оформити вже верифіковані, технічно сумісні дані. Такий підхід гарантує, що користувач отримує 100% валідну з технічної точки зору конфігурацію, супроводжену зрозумілим та переконливим людським поясненням.

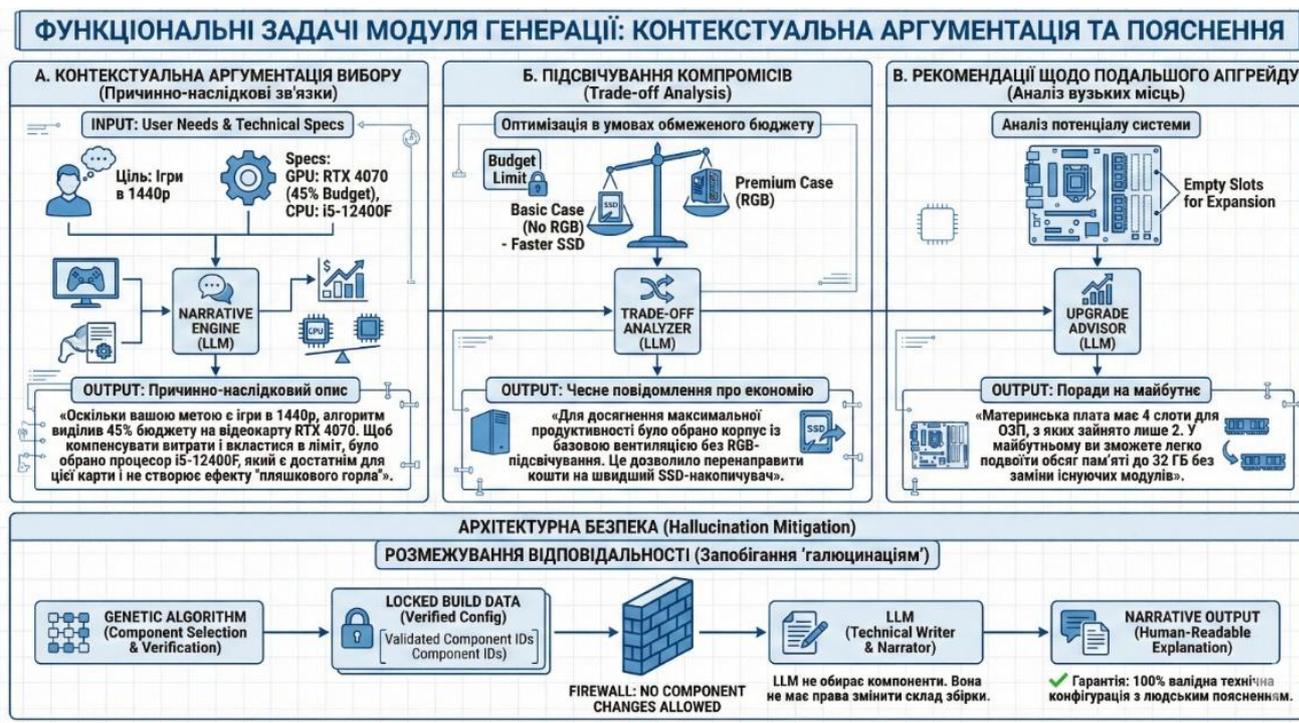


Рис 2.4 Функціональні задачі модуля генерації

2.3 Архітектура розробленої методики

Архітектура програмного комплексу побудована за модульним принципом, що забезпечує гнучкість, масштабованість та чітке розмежування логіки [16]. Система складається з п'яти взаємопов'язаних рівнів, взаємодія між якими зображена на структурній схемі (рис. 2.5).

Рівень 'Клієнта' (Client Layer) - це точка входу в систему.

- UI / API - інтерфейс (веб-додаток або чат-бот), через який користувач взаємодіє з системою. Він відправляє запит природною мовою та візуалізує отримані результати (картки товарів, пояснення).

Рівень 'Інтерпретації вимог' (Interpretation Layer) - відповідає за перетворення неструктурованого тексту в машиночитаний формат

- LLM/NLP Parser - модуль на базі великої мовної моделі (Gemini). Він отримує "сирий" текст від клієнта і виконує семантичний аналіз. Його суть: виділення сутностей (бюджет, тип задач, наявні компоненти) та формування об'єкта специфікації Spec (цілі, обмеження, ваги), який є інструкцією для оптимізатора.

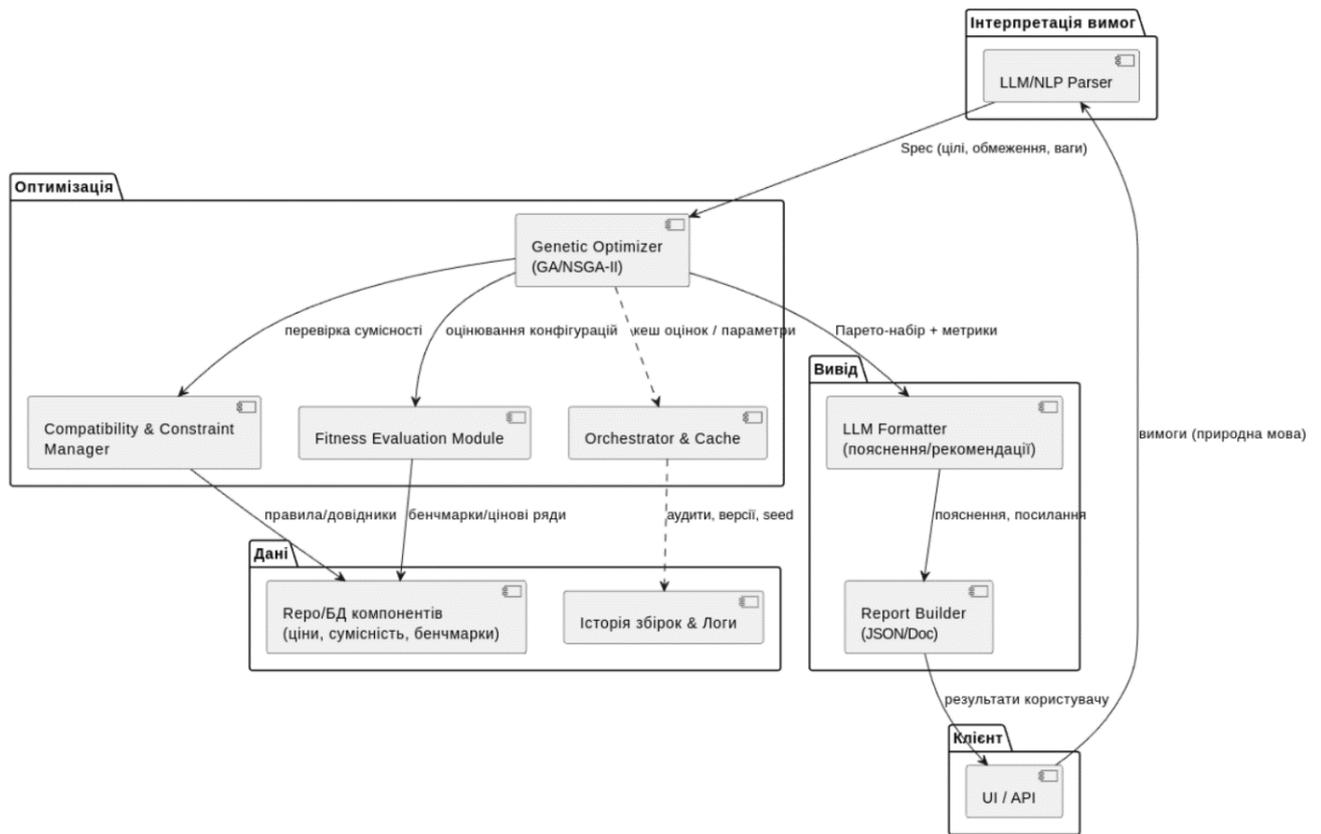


Рис 2.5 Складові розробленої методики

Рівень ‘Оптимізації’ (Optimization Layer) - це мозок системи, де відбувається основна обчислювальна робота.

- Genetic Optimizer (GA/NSGA-II) - це центральний компонент, що реалізує еволюційний пошук, де він керує популяцією збірок, виконує оператори кросинговеру та мутації.
- Compatibility & Constraint Manager - це модуль верифікації, який отримує правила сумісності з бази даних і перевіряє кожну збірку на валідність (сумісність сокетів, габаритів, живлення). Саме тут реалізовано механізм "заборонних штрафів".
- Fitness Evaluation Module – це модуль оцінювання, який розраховує кількісний показник якості (фітнес-функцію) для кожної конфігурації, спираючись на дані бенчмарків та цін.
- Orchestrator & Cache - це допоміжний компонент, що керує потоком виконання та кешує результати оцінки, щоб уникнути повторних

обчислень для однакових збірок.

Рівень ‘Даних’ (Data Layer) – це фундамент, на якому будуються розрахунки.

- Repo / БД компонентів - це локальне сховище (SQLite), що містить структуровану інформацію про всі доступні комплектуючі: технічні специфікації, актуальні ціни, результати бенчмарків та правила сумісності.
- Історія збірок & Логи - зберігає результати роботи системи для подальшого аналізу та аудиту.

Рівень ‘Виводу’ (Output Layer) - відповідає за гуманізацію математичного результату.

- LLM Formatter - отримує від оптимізатора сухий набір найкращих рішень (Парето-набір), використовуючи можливості генеративного ШІ, він створює зрозумілі текстові пояснення та рекомендації, аргументуючи вибір компонентів.
- Report Builder - компонує фінальну відповідь у форматі JSON або документа, поєднуючи технічні дані, ціни та згенерований текст, і передає це назад клієнту.

Дані рухаються по замкненому циклу, а саме від клієнта через парсер до оптимізатора, який постійно звертається до даних, і далі через форматер назад до клієнта. Така архітектура гарантує, що ШІ використовується лише для комунікації, а вся інженерна логіка виконується детермінованими модулями оптимізації. Але далі розглянемо більш деталізовано інформаційних потоків та зв'язків між модулями.

Взаємодія між компонентами системи реалізована через передачу структурованих даних. Розглянемо ключові зв'язки, зображені на схемі:

1. Вхідний потік (від користувача до алгоритму).

1) Клієнт (UI/API) → Інтерпретація вимог (LLM/NLP Parser):

- дані: вимоги (природна мова).
- опис: це сирий текст запиту (наприклад, "Збери комп'ютер за

30 тисяч"). Інтерфейс передає його на обробку в модуль парсингу.

2) LLM/NLP Parser → Оптимізація (Genetic Optimizer):

- дані: цілі, обмеження, ваги.
- опис: результат роботи парсера — нормалізований об'єкт специфікації (JSON), який містить числове значення бюджету (\$B\$), обрані вагові коефіцієнти (w_p, w_e) та перелік заблокованих компонентів для апгрейду. Це технічне завдання для генетичного алгоритму.

2. Цикл оптимізації (внутрішні зв'язки ядра).

1) Genetic Optimizer ↔ Compatibility & Constraint Manager:

- дані: перевірка сумісності.
- опис: генетичний оптимізатор генерує нову популяцію і передає її менеджеру обмежень, після перевіряє кожну збірку на порушення правил (сокети, габарити) і повертає статус валідності або величину штрафу (M_k).

2) Genetic Optimizer → Fitness Evaluation Module:

- дані: оцінювання конфігурацій.
- опис: валідні збірки передаються в модуль оцінки для розрахунку цільової функції $F(x)$. Модуль повертає числовий рейтинг, який визначає валідність збірки.

3) Genetic Optimizer ↔ Orchestrator & Cache:

- дані: кеш оцінок / параметри.
- опис: щоб не перераховувати фітнес для однакових збірок (які можуть з'являтися в різних поколіннях), оптимізатор звертається до кешу. Оркестратор також керує параметрами зупинки алгоритму.

3. Зв'язки з базою даних.

1) Compatibility Manager / Fitness Module → Repo/БД компонентів:

- дані: правила/довідники, бенчмарки/цінові ряди.

- опис: обчислювальні модулі не зберігають дані всередині себе, а роблять запити до БД для отримання технічних характеристик (для перевірки сумісності) та результатів бенчмарків (для оцінки продуктивності).

2) Orchestrator → Історія збірок & Логи:

- дані: аудити, версії, seed.
- опис: система зберігає метадані про процес пошуку (seed генератора випадкових чисел, кількість ітерацій) для можливості відтворення результатів та аналізу помилок.

4. Вихідний потік (від алгоритму до користувача).

1) Genetic Optimizer → Вивід (LLM Formatter):

- дані: парето-набір + метрики.
- опис: після завершення еволюції оптимізатор передає не одну, а набір найкращих збірок разом з їхніми технічними показниками. Це сирі дані (ID компонентів), які ще незрозумілі користувачеві.

2) LLM Formatter → Report Builder:

- дані: пояснення, посилання.
- опис: форматер генерує текстовий опис (чому обрано саме це), додає маркетингові назви та посилання на магазини.

3) Report Builder → клієнт (UI/API):

- дані: результати користувачу.
- опис: фінальний етап, який сформований JSON-документ (з картинками, цінами та текстом від ШІ) відправляється на фронтенд для відображення.

2.4 Діаграма послідовності

На діаграмі (рис. 2.6) відображено послідовність обміну даними між клієнтським інтерфейсом та мікросервісами бекенду в процесі обробки запиту

користувача.

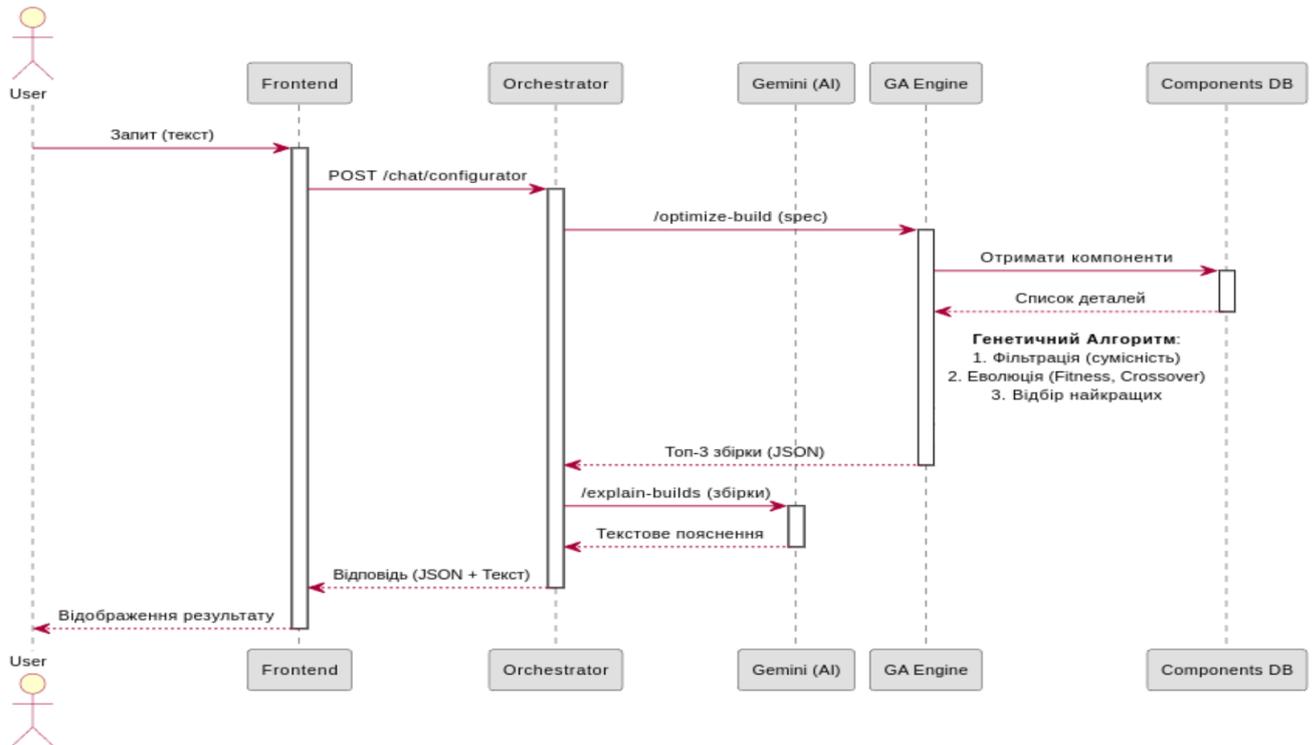


Рис 2.6 Діаграма послідовності розробленої методики

Життєвий цикл запиту складається з семи етапів:

1. Користувач вводить текстовий опис бажаної конфігурації через веб-інтерфейс. Клієнтська частина (Frontend) формує HTTP POST-запит на ендпоінт /chat/configurator і передає його до центрального шлюзу системи — сервісу Orchestrator.

2. Отримавши та формалізувавши вимоги користувача, Orchestrator ініціює процес підбору, надсилаючи внутрішній запит /optimize-build до обчислювального ядра системи — сервісу GA Engine.

3. Сервіс GA Engine звертається до репозиторію компонентів (Components DB) для завантаження актуального каталогу комплектуючих із технічними характеристиками та цінами.

4. Усередині сервісу GA Engine відбувається ізольований ітеративний процес оптимізації, який включає:

- Фільтрацію компонентів.
- Виконання циклу генетичного алгоритму (оцінка Fitness, селекція,

кросинговер, мутація).

- Ранжування та відбір фінальних варіантів.

5. По завершенню обчислень GA Engine повертає Оркестратору список із топ-3 оптимальних конфігурацій у форматі JSON.

6. Оркестратор передає технічні дані знайдених збірок до сервісу штучного інтелекту Gemini (AI) через запит /explain-builds. На цьому етапі модель трансформує сухі специфікації у зрозуміле текстове обґрунтування вибору.

7. Оркестратор об'єднує отриманий від Gemini текст із технічним JSON-об'єктом і відправляє сформовану відповідь на Frontend для візуалізації результату користувачеві.

Опис компонентів системи:

1. User (користувач) - кінцева особа, яка взаємодіє з системою. Користувач не має спеціальних технічних знань, тому формулює свої потреби у вигляді неструктурованого текстового запиту (наприклад, "хочу ігровий комп'ютер за 40 тисяч").

2. Frontend (клієнтський інтерфейс) - веб-додаток (реалізований на HTML/CSS/JS), що забезпечує графічний інтерфейс для спілкування. Він відповідає за відправку запитів користувача на сервер та відображення отриманих результатів (карток товарів, текстових пояснень) у зручному для сприйняття вигляді.

3. Orchestrator (сервіс-оркестратор) - головний мікросервіс (chat_orchestrator_service), який керує бізнес-логікою додатка. Він не виконує важких обчислень, але знає, в якій послідовності викликати інші сервіси: спочатку відправити текст до AI, потім отримані параметри — до генетичного алгоритму, і врешті — згенерувати фінальну відповідь. Він зв'язує всі компоненти в єдину систему.

4. Gemini AI (сервіс штучного інтелекту) - мікросервіс-обгортка (gemini_service) навколо API Google Gemini. Виконує дві інтелектуальні функції:

5. Розуміння намірів користувача та вилучення параметрів (бюджет, цілі) із тексту.

6. Генерація зрозумілих текстових пояснень та аргументації вибору компонентів.

7. GA Engine (рушій генетичного алгоритму) - сервіс (ga_configurator_service), де реалізовано математичну модель та сам генетичний алгоритм. Він отримує формалізовані вимоги і виконує ресурсоємні операції: створення популяції, розрахунок фітнес-функцій, перевірку сумісності та еволюційний пошук найкращої конфігурації.

8. Components DB (база даних компонентів) - репозиторій (components_service + SQLite), що зберігає структуровану інформацію про всі доступні на ринку комплектуючі. База містить технічні специфікації (сокети, частоти, TDP), результати бенчмарків та актуальні ціни, які необхідні GA Engine для розрахунків.

2.5 Алгоритм роботи конфігуратора

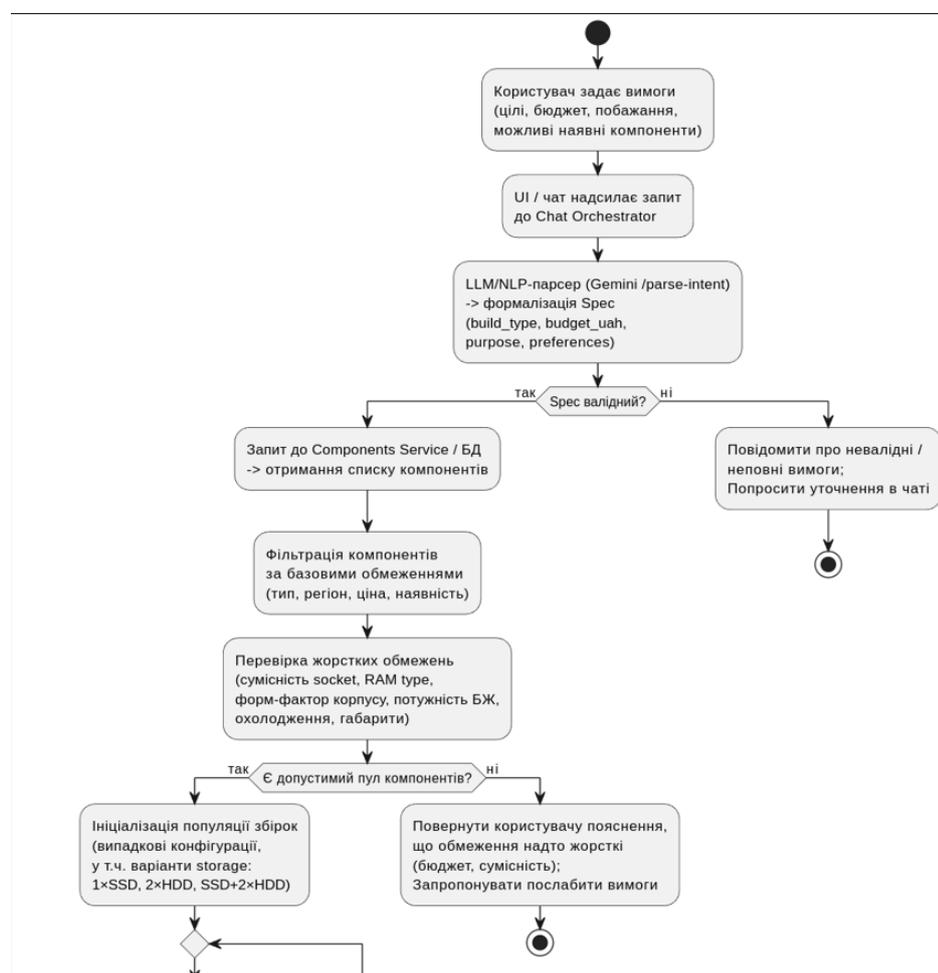


Рис 2.7 Блок схема алгоритму роботи конфігуратора (Частина 1)



Рис 2.8 Блок схема алгоритму роботи конфігуратора (Частина 2)

Процес функціонування розробленої системи ініціюється взаємодією користувача з веб-інтерфейсом, де він у довільній формі формулює свої вимоги, бюджет та побажання щодо майбутньої комп'ютерної системи. Отриманий запит передається через клієнтський інтерфейс до центрального сервісу Orchestrator, який перенаправляє його на модуль обробки природної мови (LLM/NLP-парсер) на базі Gemini. На цьому етапі відбувається семантичний аналіз тексту та його трансформація у формалізовану специфікацію, що містить структуровані дані

про бюджет, цільове призначення та наявні компоненти для модернізації. Система автоматично перевіряє валідність сформованої специфікації: у разі виявлення критичних помилок або неповноти даних користувачеві надсилається запит на уточнення, інакше — алгоритм переходить до роботи з базою даних.

Наступним кроком є підготовка простору пошуку. Система звертається до сервісу компонентів для отримання актуального переліку деталей, після чого застосовується багаторівнева фільтрація. Спочатку відсіюються компоненти, що не відповідають базовим критеріям (регіон, наявність, ціновий діапазон), а потім спрацьовує блок перевірки жорстких обмежень. На цьому етапі перевіряється фізична та логічна сумісність: відповідність сокетів, типів пам'яті, форм-факторів корпусу та достатності потужності блоку живлення. Якщо після фільтрації не залишається допустимого пулу компонентів, система повідомляє користувача про надто жорсткі обмеження; якщо ж пул достатній — ініціюється стартова популяція генетичного алгоритму з використанням евристик для підсистеми зберігання даних.

Основна обчислювальна робота відбувається в межах еволюційного циклу. Для кожної збірки в популяції розраховується значення фітнес-функції, яка враховує ігрову продуктивність, відповідність бюджету та якість конфігурації. Далі застосовуються генетичні оператори: селекція кращих рішень (з елементами елітаризму), кросовер (обмін блоками компонентів між збірками) та мутація (випадкова заміна окремих деталей). Важливою особливістю алгоритму є наявність Repair-операції, яка намагається виправити або штрафує технічно несумісні збірки, що виникли в процесі мутацій. Цей цикл повторюється доти, доки не буде досягнуто критерію зупинки (кількість поколінь або стабілізація результату).

На фінальному етапі отримана популяція сортується за значенням фітнес-функції, і система відбирає кілька найкращих варіантів (основний та альтернативні). Ці технічні дані передаються до модуля LLM-Formatter, який генерує зрозуміле для людини пояснення: аргументує вибір конкретних компонентів, підсвічує плюси та мінуси збірок і акцентує увагу на виконанні початкового запиту. Сформована відповідь, що складається зі структурованого

JSON-об'єкта (для карток товарів) та Markdown-тексту (для чату), відправляється на фронтенд для візуалізації кінцевому користувачеві.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ КОНФІГУРАТОРА ДЛЯ ПІДБОРУ КОМПЛЕКТУЮЧИХ ДЛЯ КОМП'ЮТЕРНИХ СИСТЕМ З УРАХУВАННЯМ БЮДЖЕТНИХ ОБМЕЖЕНЬ ТА НАЯВНОЇ АПАРТНОЇ КОНФІГУРАЦІЇ

3.1 Опис розробки методу

Програмна реалізація розробленої методики спрямована на створення автоматизованої системи, здатної вирішувати задачу багатокритеріального підбору комплектуючих в умовах невизначеності вхідних вимог. Розробка методу базувалася на принципах модульності та відокремлення обчислювальної логіки від інтерфейсу користувача (рис. 3.1) [17].

Головне завдання розроблюваного методу — виступити «інтелектуальним посередником» між користувачем, який має абстрактні потреби, та ринком компонентів, що має суворі технічні обмеження. Методика повинна автоматизувати рутинні інженерні розрахунки, які зазвичай виконує консультант.

Уявімо типовий сценарій модернізації. Користувач має старий ПК з якісним корпусом та блоком живлення на 600 Вт, але хоче оновити платформу для сучасних ігор, маючи бюджет 25 000 грн. Задача методики не просто підібрати процесор і відеокарту на цю суму, а знайти таку комбінацію, яка фізично вміститься в старий корпус, не перевантажить старий блок живлення і дасть максимальний FPS. Якщо методика просто запропонує потужну відеокарту, яка вимагає 750 Вт, це буде провал. Розроблений метод повинен автоматично врахувати ці обмеження ("заблокувати" старі компоненти) і знайти оптимум у звуженому просторі рішень.



Рис 3.1 Процес роботи розробленої методики

Для реалізації цього сценарію програмний метод повинен забезпечувати виконання наступних функцій:

- 1) здатність перебирати тисячі комбінацій для знаходження GPU конфігурації з максимальним сумарним показником продуктивності (за бенчмарками), не перевищуючи заданий бюджет;
- 2) автоматичне відсіювання варіантів, що порушують логічні (сокет, тип пам'яті), електричні (потужність БЖ) або фізичні (габарити корпусу) обмеження;
- 3) врахування наявних у користувача компонентів як незмінної частини хромосоми, адаптуючи нові деталі під них;
- 4) здатність приймати параметри, вилучені з природної мови через NLP-модуль, а не лише через жорсткі фільтри.

Вхідними даними для роботи обчислювального ядра є формалізована специфікація запиту, що містить:

- 1) числове обмеження вартості (у гривнях);
- 2) цільовий профіль використання (Gaming, Workstation), що визначає вагові коефіцієнти для фітнес-функції;

3) список ідентифікаторів компонентів, які необхідно зафіксувати (для сценарію апгрейду).

4) актуальна база даних із технічними характеристиками, цінами та результатами синтетичних тестів.

Процес створення програмного методу складався з п'яти послідовних кроків (рис. 3.2):

1. Проектування структур даних та простору пошуку. На першому етапі було розроблено об'єктну модель системи. Створено клас для уніфікованого представлення будь-якої деталі (процесора, відеокарти тощо) та клас (Хромосома), що описує конкретну збірку як набір посилок на компоненти. Для оптимізації доступу до даних реалізовано кешування бази даних у оперативній пам'яті у вигляді хеш-таблиць, згрупованих за типами комплектуючих.

2. Наступний етап це реалізація ядра Генетичного алгоритму. Розроблено основний клас, що керує життєвим циклом популяції. Реалізовано функцію ініціалізації, яка використовує евристику розумного посіву: замість повністю випадкового вибору, алгоритм на старті підбирає сумісні пари «процесор – материнська плата», що значно підвищує якість початкової популяції та прискорює збіжність методу.

3. Етап імплементація механізму перевірки обмежень. Це є ключовим етапом, яка стала розробка функції оцінки пристосованості. У ній запрограмовано математичну модель цільової функції, також реалізовано клас, який містить методи перевірки сумісності. Впроваджено логіку заборонних штрафів, якщо метод перевірки повертає False, фітнес-функція присвоює збірці критично низький рейтинг, виключаючи її з подальшої селекції.

4. Наступний етап розробка механізму для апгрейду. Для реалізації унікальної функції модернізації було модифіковано стандартні оператори еволюції. У функції мутації додано умовну перевірку, де перед зміною гена алгоритм перевіряє, чи входить цей компонент до списку. Якщо так, то зміна блокується, що дозволило трансформувати алгоритм глобального пошуку в алгоритм пошуку умовного оптимуму навколо зафіксованих вузлів.

5. Та останній етап інтеграція з сервісами ШІ. На фінальному етапі було розроблено модуль взаємодії з API Gemini, який налаштовано промпт-інжиніринг для двох задач:

- парсингу вхідного тексту в JSON-структуру специфікації.
- генерації текстового пояснення на основі технічних характеристик знайденої збірки.



Рис 3.2 Послідовні кроки створення методики

У результаті реалізації створено гібридну методику, що працює за конвеєрним принципом. Отримавши неструктурований запит від користувача, система за допомогою LLM перетворює його на чіткі математичні обмеження. Далі оптимізаційне ядро запускає еволюційний процес, де популяція збірок проходить через "сито" перевірок на сумісність та бюджетну ефективність. Завдяки механізму система успішно вирішує задачі апгрейду, зберігаючи старі компоненти користувача. На виході формується набір гарантовано сумісних, економічно збалансованих конфігурацій, доповнених аргументованим поясненням вибору.

3.2 Опис використання програмних засобів

3.2.1 FastAPI

FastAPI – це сучасний високопродуктивний веб-фреймворк Python для створення REST API. Маршрути (endpoints) у FastAPI визначаються декораторами над функціями-обробниками [18]. Наприклад, декоратор `@app.get("/")` вказує фреймворку, що наступна функція обробляє HTTP-запити, які приходять за шляхом. Розробник оголошує функцію (як правило, асинхронну `async def`) з параметрами, які відповідають частинам URL, запиту або тіла. FastAPI автоматично зв'язує параметри функції з даними запиту: шляхові параметри, запити та JSON-тіло прив'язуються до аргументів за іменем та типом. Для тіла запиту зазвичай визначаються моделі даних за допомогою Pydantic – це дозволяє FastAPI автоматично виконувати парсинг і валідацію входу. У типового обробника після отримання та валідації даних реалізується бізнес-логіка (наприклад, розрахунок конфігурації ПК чи звернення до бази даних), а результат повертається у вигляді об'єкта (словника або Pydantic-моделі), який FastAPI серіалізує у JSON-відповідь клієнту.

У мікросервісній архітектурі кожен сервіс розгорнуто окремо і спілкується з іншими через чітко визначений інтерфейс. Найчастіше використовуються HTTP-запити і REST API для синхронної взаємодії, або ж більш швидкі двійкові протоколи на кшталт gRPC для зменшення затримок. У контексті ПК-конфігуратора, один сервіс (наприклад, сервіс рекомендацій конфігурації) може викликати API іншого сервісу (наприклад, сервісу бази даних компонентів) через HTTP. Дані при цьому передаються у форматі JSON, що є зручним стандартом обміну між мовами та платформами. Кожен мікросервіс визначає контракт – структуру запиту і відповіді – і дотримується його, щоб інші компоненти могли коректно викликати цей сервіс. FastAPI спрощує такий обмін, оскільки вбудовано підтримує JSON: отриманий JSON автоматично конвертується у Python-об'єкти (Pydantic-моделі), а повернутий об'єкт – у JSON-відповідь. Для підвищення ефективності обробки паралельних викликів між сервісами FastAPI

використовує асинхронність – коли один мікросервіс очікує відповіді від іншого, він не блокує потік і може обробляти інші запити. Це особливо важливо в системах з великою кількістю взаємних звернень.

Обробники у FastAPI зазвичай визначаються як функції з анотаціями типів для всіх вхідних даних. Наприклад, можна визначити POST-маршрут для створення об'єкта Item: декоратор `@app.post("/items/")` прив'язує шлях, а функція `async def create_item(item: Item)` приймає JSON-об'єкт, що автоматично парситься у Pydantic-модель Item. У середині функції-обробника можна звернутися до полів item як до звичайного об'єкта Python (наприклад, `item.name` або `item.price`). Після виконання потрібних операцій (збереження до БД, виклику іншого сервісу тощо) обробник повертає результат – зазвичай словник або об'єкт, що можна конвертувати у JSON. FastAPI подбає про те, щоб серіалізувати його у правильний формат і надіслати клієнту з відповідним статусом HTTP. Завдяки інтеграції з типізацією Python, розробник отримує автодоповнення і перевірку типів у редакторі, а FastAPI – інформацію для валідації вхідних даних.

FastAPI надає ряд “з коробки” можливостей, які роблять його дуже зручним для створення та деплою мікросервісних API. Зокрема, він забезпечує автоматичну валідацію вхідних даних, асинхронну (неблокуючу) обробку запитів та автоматично генерує документацію API (OpenAPI/Swagger UI) без додаткових зусиль з боку розробника. Це означає, що щойно ви визначили маршрути і моделі, FastAPI одразу надає інтерактивну документацію, де можна тестувати запити, що прискорює інтеграцію та тестування сервісів. Крім того, за результатами незалежних тестів і завдяки використанню асинхронного сервера Uvicorn, FastAPI працює надзвичайно швидко – він “блискавичний і сучасний фреймворк, який автоматично генерує документацію API і підтримує асинхронні запити”, будучи “ідеальним для мікросервісів з великою кількістю паралельних запитів”. Легкість розгортання також забезпечується сумісністю з Docker: застосунок на FastAPI можна упакувати в компактний контейнер і масштабувати горизонтально при збільшенні навантаження. У підсумку, використання FastAPI дозволяє швидко створити

легковагий сервіс з чітко визначеним REST API, який легко підтримувати, документувати і інтегрувати в архітектуру мікросервісів.

3.2.2 Інтеграція з Google Gemini API

Google Gemini API – це хмарний сервіс від Google для доступу до потужних моделей штучного інтелекту нового покоління (LLM – Large Language Model). Він надає розробникам можливість виконувати завдання з обробки природної мови, генерації тексту та інші AI-функції через простий інтерфейс API [19]. Фактично, Gemini API виступає як зовнішній «мозок» для застосунку, замість того щоб розгортати велику мовну модель локально, мікросервіс може надсилати запити до API Google і отримувати від нього згенеровану відповідь. У нашому проєкті ПК-конфігуратора інтеграція з Google Gemini дозволяє реалізувати інтелектуальні можливості, такі як розуміння запиту користувача та генерація пояснень до зібраних конфігурацій ПК, використовуючи зовнішній AI-сервіс.

Щоб скористатися моделлю Gemini, мікросервіс автентифікується за допомогою API-ключа і формує запит, як правило у вигляді JSON з параметрами. Виклик може здійснюватись через офіційну бібліотеку (SDK) – Google Generative AI SDK для Python – або через HTTP REST запит. У випадку SDK, достатньо сконфігурувати клієнт API ключем і викликати метод генерації тексту. Наприклад, офіційний приклад показує виклик `client.models.generate_content` із зазначенням моделі (наприклад, "gemini-2.5-flash") та передачі рядка із підказкою (prompt) для генерації контенту. В результаті API повертає об'єкт відповіді, з якого можна отримати згенерований текст (через поле `response.text` у SDK або поле JSON у REST-відповіді).

Важливою частиною є формування prompt – текстової підказки, яка задає моделі завдання. Prompt може містити інструкцію (наприклад, "Поясни, як працює штучний інтелект простими словами"), а також контекст чи дані, на основі яких модель повинна згенерувати результат. Розробники можуть налаштовувати параметри генерації – максимальну довжину відповіді, температуру (ступінь творчості моделі), тощо. Після відправки запиту модель

опрацьовує його і повертає результат у вигляді згенерованого тексту або структурованих даних залежно від налаштувань запити.

Для досягнення потрібного результату від LLM, застосунок формує спеціальні підказки. У нашому випадку є дві ключові функції, що використовують LLM: `parse-intent` і `explain-builds`.

1. Функція `parse-intent` відповідає за інтерпретацію запити користувача. Вона бере текстове питання або вимогу користувача (наприклад, «Хочу ігровий ПК з бюджетом до \$1000») і надсилає його до моделі Gemini з проханням визначити намір користувача та витягти структуровані параметри. У реалізації це може бути зроблено через `prompt`, який інструктує модель: «Ти – експерт з визначення намірів. Проаналізуй наведений текст користувача і виділи всі наміри у форматі JSON за заданою схемою». Для прикладу, розробники можуть задати схемою, що відповідь має містити список об'єктів з полями `"intent"` (коротка назва наміру) та `"statement"` (оригінальне твердження користувача). Google Gemini API підтримує режим структурованого виводу: можна передати разом із `prompt` JSON-схему (наприклад, у форматі JSON Schema або через Pydantic-модель) і вимагати від моделі чітко дотримуватися цієї структури. Це гарантує передбачуваний і синтаксично правильний формат відповіді, що спрощує подальший розбір результату. Таким чином, `parse-intent` надсилає до LLM запит із контентом користувацького вводу та вказівкою повернути JSON. Модель у відповіді генерує текст у форматі JSON (наприклад, масив `intents` з об'єктами). Мікросервіс отримує цю відповідь і обробляє її – парсить JSON (використовуючи, наприклад, Pydantic-модель для намірів) і отримує структуровані дані про те, що хоче користувач. Завдяки цьому етапу система «розуміє» запит: наприклад, виявляє, що користувач хоче ігровий комп'ютер і має бюджет \$1000.

2. Функція `explain-builds` призначена для генерації пояснення зібраної конфігурації ПК зрозумілою мовою. Після того, як ядро системи (зокрема, генетичний алгоритм або інший модуль оптимізації) підбирає оптимальний набір компонентів під потреби та бюджет, постає задача пояснити користувачу, чому

саме ці компоненти були обрані і як вони відповідають вимогам. Замість ручного складання шаблонів тексту, мікросервіс використовує LLM для генерації обґрунтування. В реалізації це означає: формується `prompt`, що містить деталі зібраної конфігурації (список компонентів: модель процесора, обсяг пам'яті, тощо) та інструкцію на кшталт “Поясни користувачеві цю конфігурацію, наголоси, як вона відповідає його вимогам (ігри, бюджет) і переваги вибраних компонентів”. Модель Gemini отримує такий запит і створює розгорнутий опис, який потім повертається користувачу. Ця відповідь може, наприклад, пояснювати, що «Обраний процесор Intel Core i5 забезпечить високу продуктивність в іграх, а 16 ГБ ОЗП достатньо для стрімінгу і багатозадачності. Враховано ваш бюджет – використано відеокарту, яка дає максимальну продуктивність за свою ціну...». Важливо, що LLM здатна генерувати природні описи з урахуванням контексту; вона може навіть додати поради щодо майбутнього апгрейду або альтернатив, якщо це було задано в інструкції. З технічного боку, `explain-builds` надсилає запит до того самого API Gemini, але з іншим акцентом: тут не потрібен строгий JSON, натомість потрібен зв'язний текст. Тому цей запит може використовувати стандартний текстовий режим генерації. Модель поверне фрагмент тексту (можливо, кілька абзаців). Мікросервіс отримує цей текст і просто пересилає його як пояснення кінцевому користувачу через API. Таким чином, завдяки інтеграції з LLM, користувач отримує людино-зрозуміле пояснення складного технічного рішення, згенероване автоматично.

Отримавши відповідь від Gemini API, мікросервіс її обробляє відповідно до очікуваного формату. Якщо це текст (для пояснення конфігурації) – він може бути відданий клієнту без змін або з мінімальним постпроцесингом (наприклад, додати розмітку). Якщо це JSON (для намірів) – сервіс парсить його. Тут знову стає у пригоді `Pydantic`: можна визначити модель, що описує формат відповіді LLM, та викликати метод на кшталт `Model.model_validate_json(...)` для автоматичної десеріалізації JSON у об'єкт. Це забезпечує додаткову перевірку – якщо модель повернула щось не те, парсер викине помилку. В цілому,

використання зовнішнього LLM-сервісу надає кілька переваг для проєкту:

1) Google Gemini – це передова мовна модель від Google, яка володіє глибоким розумінням природної мови. Використовуючи її, ми отримуємо сучасний рівень якості в розумінні запитів і генерації тексту, не займаючись тренуванням власних моделей. Це особливо важливо, оскільки розробка й навчання великої моделі потребує великих даних, експертизи та обчислювальних ресурсів.

2) сервіс спроектовано дружнім для розробників, з простими REST/SDK викликами. Достатньо отримати API-ключ і зробити кілька викликів SDK, як описано вище, щоб інтегрувати AI-функціональність. В нашому випадку, це значно спростило реалізацію “розумних” можливостей: буквально кілька функцій, що формують запити, замінили потенційно складну логіку парсингу запитів і генерації описів вручну.

3) оскільки Gemini API працює на інфраструктурі Google, він автоматично масштабується під навантаження і забезпечує надійну роботу навіть при великій кількості запитів. Для нашої системи це означає, що якщо раптово багато користувачів почнуть одночасно запитувати рекомендації, тягар генерації відповідей лягає на потужності Google, а не на наш сервер. Ми лише відправляємо запити і отримуємо відповіді, не турбуючись про виділення GPU або пам'яті для моделі.

4) зовнішній сервіс постійно вдосконалюється провайдером. Google може випускати нові версії моделі (наприклад, Gemini 3) або покращувати якість відповідей, і наш застосунок автоматично виграє від цього, варто лише змінити параметр моделі. Не потрібно вручну оновлювати свій код для покращення якості мовної обробки – достатньо задати ім'я нової моделі або нові параметри API. Це майбутнє-захищає систему: через рік наш сервіс все ще зможе використовувати актуальний AI.

Звичайно, інтеграція з зовнішнім API потребує зважати на затримки мережі та можливі додаткові витрати (Gemini API є платним вище певних лімітів). Проте у контексті магістерського проєкту такі витрати виправдані функціональністю,

яку ми отримуємо.

В підсумку, parse-intent і explain-builds демонструють успішний випадок використання хмарного AI-сервісу: перший забезпечує NLU (Natural Language Understanding) – розуміння намірів користувача, другий – NLG (Natural Language Generation) – генерацію зрозумілого пояснення, і все це з мінімальним кодом з боку розробника, оскільки важку роботу виконує модель Gemini.

3.2.3 Pydantic V2

Для підтримки чіткої структури даних у взаємодії мікросервісів проєкт активно використовує Pydantic. Pydantic – це бібліотека Python, що дозволяє визначати моделі даних з використанням анотацій типів і автоматично виконувати валідацію та серіалізацію цих даних [20]. У мікросервісному ПК-конфігураторі моделі Pydantic слугують своєрідним “мовним контрактом” між сервісами: вони задають, які поля і якого типу очікуються в запитах і відповідях. Це особливо важливо, коли сервіси обмінюються JSON-повідомленнями – Pydantic дозволяє перевести неструктуровані JSON-дані у строго типізовані Python-об’єкти і назад.

Основні можливості Pydantic, корисні в нашій системі, можна виділити так:

1. Розробник визначає класи-моделі, що успадковуються від BaseModel Pydantic, перелічуючи атрибути і їх типи (наприклад, name: str, price: float, components: list[Component] тощо). Ці моделі використовуються як для вхідних, так і для вихідних даних мікросервісів. Завдяки використанню стандартних підказок типів Python, код стає більш наочним і інструменти можуть перевіряти коректність ще до виконання програми. Роль моделей між мікросервісами полягає в тому, що всі сервіси дотримуються однієї структури даних. Наприклад, якщо визначено модель PCBuild з полями для компонентів, цією ж моделлю можуть користуватися різні сервіси – один щоб сформувати об’єкт конфігурації, інший – щоб його розібрати. Це уникає розбіжностей у форматі: усі говорять “однією мовою моделей”.

2. Коли JSON надходить до сервісу (від користувача або від іншого сервісу), Pydantic-модель спробує побудувати об'єкт на основі цих даних. На цьому етапі автоматично перевіряється тип кожного поля і виконуються задані обмеження. Якщо дані не відповідають очікуванням, викидається виняток `ValidationError` з детальним описом всіх помилок по полях. Pydantic приводить (конвертує) типи, якщо це можливо, або повідомляє про невідповідність. Наприклад, якщо модель очікує `int`, а отримано строку "5", Pydantic спробує перетворити її на 5; але якщо отримано нечислову строку, буде помилка. Деталізація помилок – один з сильних боків Pydantic: для кожного поля повідомляється, що пішло не так (наприклад, "field X: expected positive int, got negative"). Як показано у прикладі, Pydantic надає докладне повідомлення про помилки для кожного поля, вказуючи, яке значення отримано і чому воно не валідне. Це дозволяє перехопити будь-які проблеми із даними на самому вході в сервіс, до виконання бізнес-логіки. У FastAPI всі такі помилки автоматично перетворюються на відповіді 422 `Unprocessable Entity` з описом, тому клієнт або інший сервіс відразу дізнається, які поля некоректні. В результаті система ніколи не працює з невалідними або неочікуваними даними – якщо модель створена успішно, можна бути впевненим у коректності типів і присутності необхідних значень. Це підвищує надійність: об'єкти, створені без помилок, гарантовано відповідають схемі і можуть використовуватися у подальших розрахунках або пересилатися іншим мікросервісам без додаткових перевірок.

3. Pydantic суттєво спрощує перетворення об'єктів у формат для передачі (JSON) і назад. Будь-яку модель можна легко перетворити на словник (`model.model_dump()`) або JSON-рядок (`model.model_dump_json()`), і навпаки – створити об'єкт з словника (`Model.model_validate(...)`) чи з JSON-рядка (`Model.model_validate_json(...)`). Ці методи враховують вкладені моделі і складні типи, автоматично серіалізуючи їх (наприклад, `datetime` в строку ISO, `Enum` в його значення тощо). Для нашого проекту це критично, адже мікросервіси обмінюються даними у JSON: Pydantic дозволяє просто взяти модель і отримати з неї JSON для передачі по мережі, або отримати JSON і розпарсити у модель –

без написання ручного коду перетворення. Бібліотека також може генерувати JSON Schema для моделі (використовується, наприклад, при інтеграції з LLM Gemini, як згадано вище, або для автоматичної документації FastAPI). Причина, чому FastAPI обрав Pydantic як основу для роботи з JSON, саме в тому, що Pydantic легко перетворює моделі до JSON і гарантує відповідність схемі. Це робить API само-документованим і сумісним практично з будь-яким клієнтом, що працює з JSON.

4. Pydantic дозволяє централізовано обробляти помилки валідації. У контексті FastAPI помилки автоматично перетворюються на HTTP-відповідь із кодом 422 і списком помилок, але також можна явно ловити `ValidationError` у власному коді (наприклад, якщо модель використовується поза FastAPI) і реагувати на нього. Типові помилки, які перехоплює Pydantic: відсутність обов'язкового поля, невідповідність типу (`type_error`, коли, скажімо, замість списку прийшов словник), порушення валідаторів (як от невірний формат email, занадто короткий рядок, число поза допустимого діапазону тощо). Pydantic v2 також надає більш структуровану інформацію про помилки, де кожна має код та посилання на документацію, що полегшує налагодження. Наприклад, якщо поле не пройшло перевірку `Regex` або не відповідає схемі вкладеного об'єкта – це все буде відображено в списку помилок. Таким чином, Pydantic перехоплює практично всі можливі аномалії в даних на рівні моделі. У результаті решта логіки системи може бути впевнена, що працює з чистими даними.

У проєкті використано Pydantic версії 2, що привносить ряд покращень порівняно з v1. Найсуттєвіше – це переписаний “движок” валідації на мові Rust (пакет `pydantic-core`), який робить валідацію значно швидшою та ефективнішою. Кожна модель тепер створюється з меншими накладними витратами, що важливо у мікросервісі, який може валідувати сотні запитів на секунду. Також v2 спростила API: замість методів `parse_obj`, `parse_raw` тощо з'явилися уніфіковані `model_validate` та `model_validate_json`, метод `dict()` замінено на `model_dump()`, що чіткіше відображає намір (`dump` даних). Ці зміни зробили код зрозумілішим і уникнули багатьох неоднозначностей v1. Pydantic v2 краще

підтримує строгий режим (Strict mode), де можна вимагати, щоб не відбувалось неявних приведень типів – це корисно, коли потрібна максимальна надійність. В нашому випадку це гарантує, що, наприклад, поле, яке мало бути цілим, не прийме дробове значення, а поле з перерахуванням (Enum) не прийме невідоме значення. Окрім того, інтеграція з сторонніми інструментами (як-от IDE, туру) залишилась на високому рівні: розробник отримує підказки і перевірку типів ще до запуску програми.

Dydantic відіграє критичну роль у побудові мікросервісної системи: він зв'язує компоненти єдиною схемою даних і забезпечує, щоб на кожному етапі дані відповідали очікуванням. Це зменшує кількість помилок, прискорює розробку (бо багато шаблонного коду з парсингу або перевірки відпадає) та підвищує впевненість у стабільності сервісів.

Саме завдяки Dydantic наші мікросервіси можуть легко обмінюватися складними структурами (наприклад, списками компонентів з вкладеними характеристиками) – ми просто визначаємо одну модель і використовуємо її у різних модулях. Якщо в цю модель внести зміни (додати поле, перейменувати тип), Dydantic і засоби типізації дозволяють швидко знайти місця, де потрібні правки, і гарантують, що нічого не пропущено.

Отже, Dydantic v2 забезпечує типізованість, валідованість та сумісність даних в межах усього ПК-конфігуратора, що є основою надійної роботи мікросервісної архітектури.

3.2.4 Uvicorn

Для запуску веб-сервісів, реалізованих на FastAPI, у проєкті використовується Uvicorn — високопродуктивний ASGI-сервер, написаний на Python із використанням бібліотеки uvloop. Його основне призначення — приймати HTTP-запити від клієнтів, передавати їх застосунку FastAPI та повертати сформовані відповіді.

Використання Uvicorn має декілька важливих переваг для заданої задачі:

1. Uvicorn реалізує протокол ASGI (Asynchronous Server Gateway

Interface), завдяки чому сервер може ефективно обробляти велику кількість одночасних запитів, не блокуючи потоки при зверненні до інших сервісів (наприклад, до бази даних або Gemini API). Це особливо актуально для мікросервісної архітектури, де оркестратор і сервіс генетичного алгоритму часто виконують зовнішні HTTP-запити.

2. Uvicorn оптимізовано для роботи з асинхронними фреймворками (FastAPI, Starlette тощо) і демонструє високу пропускну здатність навіть на обмежених ресурсах. Для магістерського прототипу це дозволяє отримати “реактивну” систему без необхідності в потужному серверному обладнанні.

3. Запуск сервісу зводиться до команди на кшталт `uvicorn app:app --host 0.0.0.0 --port 8000`, що спрощує локальний запуск, тестування та подальше розгортання. Uvicorn легко інтегрується з Docker та системами оркестрації (Kubernetes), що є перспективним для подальшого масштабування системи.

4. У режимі розробника Uvicorn може автоматично перезапустити застосунок при зміні коду, що прискорює цикл розробки та налагодження.

Таким чином, Uvicorn виступає легковаговою, але продуктивною “оболонкою” для FastAPI-сервісів, забезпечуючи стабільну та ефективну роботу усіх мікросервісів ПК-конфігуратора.

3.2.5 SQLite як СКБД для каталогу компонентів

Для зберігання каталогу комп'ютерних комплектуючих у проєкті використано вбудовану реляційну систему керування базами даних SQLite. Уся база даних розміщується в одному файлі (наприклад, `components.db`), який зберігається разом із програмним кодом. Вибір саме SQLite обумовлено такими міркуваннями:

На відміну від серверних СКБД (MySQL, PostgreSQL), SQLite не потребує розгортання окремого СУБД-сервера. Достатньо стандартної бібліотеки Python, щоб створити файл бази та виконувати SQL-запити [22]. Це значно спрощує запуск системи на будь-якій машині розробника чи під час демонстрації магістерської роботи.

Усі дані (перелік компонентів, їх характеристики, ціни, показники продуктивності) зберігаються в одному файлі. Для перенесення проєкту достатньо скопіювати програму разом із цим файлом, що робить розгортання прототипу максимально простим.

SQLite реалізує підмножину стандарту SQL-92, чого достатньо для типових операцій: створення таблиць, вставки/оновлення записів, виконання вибірок з фільтрацією та сортуванням. Це дозволяє реалізувати доволі гнучкі запити до каталогу компонентів (наприклад, вибірка процесорів певного сокета, відеокарт із мінімальним обсягом відеопам'яті, компонентів у заданому ціновому діапазоні).

Обсяги даних у магістерському проєкті (десятки чи сотні моделей компонентів) є відносно невеликими, тому продуктивності SQLite більш ніж достатньо для швидкого виконання запитів. Час відповіді вимірюється мілісекундами, що не створює вузьких місць у роботі системи.

Обрана архітектура не прив'язує застосунок “назавжди” до SQLite. За потреби на наступних етапах розвитку проєкту базу можна перенести на більш потужну СКБД (PostgreSQL, MySQL), зберігши ту саму логіку доступу до даних. Це робить SQLite оптимальним вибором для етапу розробки та досліджень.

У результаті SQLite виконує роль компактного й надійного сховища даних про комплектуючі, яке легко інтегрується з Python-кодом та не ускладнює розгортання системи на етапі магістерського захисту.

3.3 Загальна архітектура та організація проєкту

Система побудована за мікросервісною архітектурою: її функціональність розподілена між кількома незалежними сервісами, кожен з яких відповідає за чітко окреслену підзадачу.

Такий підхід дозволяє зробити компоненти системи невеликими, ізольованими та слабко пов'язаними між собою. Кожен сервіс розробляється і розгортається окремо, взаємодіючи з іншими виключно через стандартизовані API.

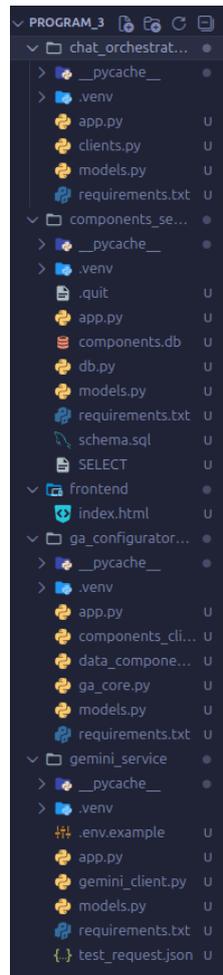


Рис 3.3 Структура проекту

Це забезпечує незалежність розгортання (можливість оновлювати чи масштабувати один сервіс, не зачіпаючи інші) та інкапсуляцію реалізації (внутрішня логіка сервісу прихована від інших, які бачать лише контракт взаємодії через API). Більше того, згідно з принципом «один сервіс — одна відповідальність», кожен сервіс зосереджується на власній бізнес-функції.

За необхідності сервіси можуть мати власні сховища даних, що усуває конкуренцію за ресурси та полегшує еволюцію окремих модулів системи. Нижче детально описано структуру кожного сервісу в проекті ПК-конфігуратора, включно з його роллю, причинами виокремлення, ізоляцією від інших, оглядом файлів у його репозиторії та прикладом використання.

Нижче подано детальніший опис кожної основної папки-сервісу проекту та її вмісту.

3.3.1 Сервіс оркестрації чату (`chat_orchestrator_service`)

`chat_orchestrator_service` виконує роль центрального координатора (оркестратора) в системі. Він отримує запити від користувача (наприклад, через інтерфейс чату) та керує виконанням цих запитів, взаємодіючи з іншими мікросервісами. Фактично цей компонент реалізує патерн оркестрації сервісів: центральний оркестратор послідовно викликає декілька інших сервісів для досягнення цілі запиту. Замість того, щоб кожен мікросервіс самостійно координував складний процес, `chat_orchestrator_service` бере на себе загальну бізнес-логіку робочого потоку і агрегує результати від інших компонентів.

Виділення оркестрації в окремий сервіс зумовлене прагненням розділити логіку діалогу або робочого процесу і спеціалізовані функції. Таким чином, цей сервіс займається тільки координацією (послідовністю викликів, аналізом намірів користувача, складанням підсумкової відповіді), а доменні обчислення (наприклад, підбір конфігурації чи отримання даних компонентів) делегуються спеціалізованим службам. Це підвищує модульність: алгоритми конфігурації ПК або звернення до моделей ШІ можуть змінюватися незалежно від логіки ведення діалогу. Команда, відповідальна за чат-інтерфейс, може розвивати його автономно, не заглиблюючись у деталі підбору комплектуючих. Крім того, такий розподіл відповідає принципу розподілу відповідальності і спрощує підтримку: зміни у бізнес-логіці діалогу не впливають на код інших сервісів і навпаки.

`chat_orchestrator_service` ізольований від інших частин системи тим, що не має прямої залежності від їхніх внутрішніх реалізацій або баз даних. Він взаємодіє з ними через клієнтські модулі та HTTP-запити, дотримуючись контрактів API. Це означає, що оркестратор оперує лише тими даними, які отримує у відповідях (наприклад, списком компонентів чи готовою конфігурацією), не знаючи деталей того, як інші сервіси їх обчислюють чи зберігають. Така слабка зв'язаність сприяє стійкості системи: у разі збою одного зі сервісів оркестратор може отримати повідомлення про помилку і обробити його (наприклад, повідомити користувача про недоступність певної функції), не падаючи вслід за ним.

Також даний сервіс не використовує спільних з іншими сервісами ресурсів - у нього немає власної бази даних, і всі залежності ізольовано інкапсульовано у файлі клієнтів. Це узгоджується з принципом, що кожен мікросервіс сам керує своїми даними та логікою, обмінюючись інформацією лише через чітко визначені інтерфейси. Таким чином досягається мінімальна залежність на рівні розробки: зміна у внутрішній структурі, скажімо, сервісу компонентів або GA-сервісу, не вимагає переписування оркестратора, якщо інтерфейс взаємодії залишається незмінним.

Вміст папки `chat_orchestrator_service`:

- `app.py` – основний виконуваний модуль сервісу оркестрації. Містить налаштування веб-застосунку (наприклад, створення Flask/FastAPI застосунку) та визначення маршрутів (`endpoint-ів`) API цього сервісу. Саме тут визначено, які HTTP-запити (наприклад, POST запит на отримання рекомендації конфігурації) обробляє оркестратор і як саме: зазвичай, шляхом виклику відповідних методів з модуля клієнтів та об'єднання їхніх результатів. `app.py` запускає сервіс на певному порту, очікуючи на вхідні запити від фронтенду або інших клієнтів.
- `clients.py` – модуль клієнтів для доступу до інших сервісів. Він інкапсулює логіку віддалених викликів: наприклад, може містити класи або функції на кшталт `ComponentsServiceClient` (для звернення до API сервісу компонентів), `GAServiceClient` (для звернення до генетичного алгоритму) чи `GeminiServiceClient` (для звернення до сервісу Gemini). У цьому файлі налаштовано URL-адреси інших мікросервісів, методи виклику (HTTP GET/POST) та обробка відповідей. Таким чином, `clients.py` дозволяє оркестратору викликати віддалені сервіси простими викликами методів, абстрагуючи деталі протоколу та формату запитів/відповідей.
- `models.py` – визначає структури даних (моделі), з якими працює оркестратор. Тут можуть бути описані класи або типи для запитів і

відповідей сервісу оркестрації. Наприклад, модель Запиту конфігурації (що містить побажання користувача: бюджет, призначення ПК тощо) або модель Відповіді (що містить рекомендовану конфігурацію у вигляді списку компонентів). Використання таких моделей забезпечує чіткість і валідність даних: при отриманні JSON-запиту з фронтенду `app.py` може розпарсити його в об'єкт із `models.py`, перевірити коректність, а потім передати до бізнес-логіки. Аналогічно, при формуванні відповіді оркестратор може створити об'єкт-модель і серіалізувати його у JSON для відправки клієнту.

Розглянемо сценарій, як оркестратор взаємодіє з іншими сервісами. Користувач через веб-інтерфейс (фронтенд) вводить запит, наприклад: «Хочу зібрати ігровий ПК з бюджетом до \$1000». При натисканні кнопки на сайті фронтенд викликає відповідний HTTP-запит до оркестратора (припустимо, `POST /configure` з параметрами бюджету та призначення). `app.py` в `chat_orchestrator_service` приймає цей запит і звертається до клієнтського модуля. Спершу, наприклад, він може викликати `ComponentsServiceClient.get_all_components()` з `clients.py`, щоб отримати актуальний перелік всіх доступних компонентів із сервісу компонентів. Потім викликається метод генетичного сервісу: скажімо, `GAServiceClient.generate_configuration(request)` – де `request` містить побажання користувача. Цей метод, реалізований у `clients.py`, відправляє HTTP-запит до відповідного endpoint `ga_configurator_service` (наприклад, `POST /generate_config`) і отримує згенеровану оптимальну конфігурацію (набір компонентів). Можливо, оркестратор також залучає `GeminiServiceClient` – наприклад, для уточнення вимог користувача через модель III або для генерації текстового пояснення до конфігурації.

Отримавши всі необхідні дані, оркестратор агрегує їх: формує підсумкову відповідь, що містить список рекомендованих комплектуючих (і, за потреби, коментарі чи пояснення). Ця відповідь (у форматі JSON) надсилається назад

фронтенду, який відображає користувачу результат – наприклад, перелік конкретних моделей компонентів з цінами та причинами вибору. Якщо викликати оркестратор окремо (наприклад, через інструмент типу curl або Postman), надіславши коректний запит, сервіс автономно виконає ту ж саму послідовність дій і поверне сформовану конфігурацію.

Така автономність дозволяє тестувати оркестратор як окремий модуль, впевнившись, що він правильно координує інші частини системи.

3.3.2 Сервіс компонентів (components_service)

Сервіс компонентів відповідає за зберігання та надання даних про апаратні компоненти ПК. Це довідковий сервіс, який містить каталог усіх доступних комплектуючих: процесори, відеокарти, оперативна пам'ять, накопичувачі тощо, разом з їх характеристиками (ціна, технічні параметри, сумісність з іншими компонентами, виробник та ін.). Функціонально цей сервіс забезпечує CRUD-операції або принаймні читання даних: інші частини системи можуть запитувати список компонентів певного типу, деталі про конкретний компонент, фільтрувати компоненти за критеріями (наприклад, показати всі відеокарти до певної ціни) тощо. Таким чином, сервіс виступає централізованим сховищем інформації про комплектуючі, гарантує її актуальність і цілісність.

Виділення сервісу компонентів в окремий мікросервіс мотивоване принципами поділу за бізнес-контекстами та перевикористання. Дані про апаратні компоненти є відносно статичними і добре відокремлюються від логіки підбору конфігурацій чи від діалогової логіки. Маючи окремий сервіс, команда може оновлювати базу компонентів (додавати нові моделі, змінювати ціни, підтримувати інформацію про сумісність) незалежно від інших сервісів. Інші підсистеми (наприклад, сервіс генетичного алгоритму або потенційно будь-який інший додаток) можуть звертатися до components_service за даними, не дбаючи про те, як і де ці дані зберігаються. Це сприяє узгодженості: вся система використовує єдине «джерело істини» щодо характеристик компонентів. Окрім того, цей сервіс можна масштабувати або оптимізувати окремо – наприклад,

якщо кількість даних чи запитів зростатиме, можна перемістити базу на потужніший сервер або оптимізувати запити, не торкаючись інших сервісів. В цілому, такий підхід відповідає принципам мікросервісів, за якими кожен сервіс реалізує одну ключову функцію і може еволюціонувати самостійно.

Даний сервіс має власне сховище даних та повністю автономний шар доступу до нього. Згідно з кращими практиками, кожен мікросервіс володіє своїми даними і схемою БД. У даному випадку, це реалізовано файлом `components.db` (локальна база, ймовірно SQLite) та схемою `schema.sql`. Жоден інший сервіс безпосередньо не звертається до цієї бази; натомість усі звернення відбуваються через API `components_service`. Це ізолює дані компонентів: навіть якщо інший сервіс потребує інформації, він здійснює HTTP-запит, і `components_service` повертає необхідне у вигляді відповіді. Таким чином, інші сервіси не знають деталей реалізації – наприклад, їм байдуже, реляційна це БД чи ні, як сформовані таблиці тощо. Важливо, що `components_service` не залежить від модулів інших сервісів; його можна запускати і тестувати окремо, використовуючи внутрішню БД.

Така ізоляція гарантує, що помилки або зміни в інших модулях (наприклад, в логіці GA-сервісу) не вплинуть на збережені дані компонентів. Навпаки, якщо знадобиться змінити структуру бази чи додати нові поля (наприклад, додати параметр енергоспоживання для компонентів), це можна зробити, не залучаючи інші сервіси – достатньо оновити `components_service` та його API. Ізоляція підтримується і на рівні залежностей: у коді `components_service` немає викликів сторонніх сервісів (він самодостатній у рамках своєї задачі), тож при тестуванні чи розгортанні можна оперувати тільки цим компонентом.

Вміст папки `components_service`:

- `app.py` – основний файл застосунку сервісу. Містить запуск веб-сервера та визначення REST-інтерфейсу для доступу до даних компонентів. Наприклад, тут можуть бути визначені маршрути: `GET /components` (повертає список усіх компонентів або фільтрованих за параметрами), `GET /components/<id>` (детальна інформація про

компонент за ідентифікатором), POST /components (додавання нового компонента, якщо передбачено) тощо. app.py обробляє вхідні HTTP-запити: викликає необхідні функції з db.py для отримання чи модифікації даних, серіалізує результати у формат JSON і відправляє у відповіді. Цей файл також може містити ініціалізацію з'єднання з базою даних (використовуючи db.py) та запуск сервера (наприклад, app.run() у Flask).

- components.db – файл бази даних, що містить інформацію про всі комп'ютерні комплектуючі, якими оперує система. Судячи з розширення, це SQLite база даних, яка зберігається локально у файлової системі. У ній, найімовірніше, є одна або кілька таблиць: наприклад, таблиця components із полями id, type (тип компонента – CPU, GPU, RAM тощо), name (назва моделі), specs (JSON або окремі колонки для характеристик), price, compatibility (можливо, посилання на сумісні компоненти чи сокет/форм-фактор) та ін. components.db створена і оновлюється на основі SQL-скрипту schema.sql і, можливо, містить початкові дані, додані при ініціалізації.
- db.py – модуль доступу до бази даних. Інкапсулює всі операції читання/запису до components.db. Тут відкривається з'єднання до SQLite, виконується ініціалізація схеми (можливо, при першому запуску, шляхом виконання команд із schema.sql), а також визначено функції для отримання даних. Наприклад, get_all_components(), get_components_by_type(type), get_component_by_id(id), add_component(component) тощо. Ці функції використовуються в app.py при обробці відповідних запитів. Таким чином, db.py відокремлює логіку роботи з SQL від решти застосунку: якщо завтра формат даних чи СУБД зміняться, достатньо відредагувати цей файл, не чіпаючи API або моделі.
- models.py – містить визначення моделей даних, пов'язаних з

компонентами. Може включати класи для представлення об'єктів компонентів у пам'яті. Наприклад, клас `Component` з атрибутами, що відповідають колонкам таблиці (ідентифікатор, назва, тип, технічні характеристики, ціна тощо). Ці моделі можуть використовуватися для зручності: `db.py` може повертати об'єкти `Component` замість сирих кортежів з БД, або `app.py` може створювати такі об'єкти перед записом. Крім того, `models.py` може визначати схеми (наприклад, Pydantic-моделі, якщо використовується FastAPI) для валідації вхідних/вихідних даних. У такому випадку, при приході JSON-запиту на створення нового компонента, дані переводяться в об'єкт-модель для перевірки, і тільки тоді записуються до бази.

- `schema.sql` – SQL-скрипт, що описує структуру бази даних `components.db`. У цьому файлі явно перелічуються команди створення таблиць, індексів, можливо, початкове наповнення. Наприклад, може бути `CREATE TABLE components (id INTEGER PRIMARY KEY, type TEXT, name TEXT, specs TEXT, price REAL, ...)`; та інші таблиці або налаштування (можливо, таблиці для сумісностей чи категорій). `schema.sql` використовується при першому запуску сервісу або при його розгортанні для ініціалізації бази. Наявність цього файлу спрощує перенесення системи на інше середовище: можна швидко розгорнути нову чисту базу, виконавши цей скрипт. Також, він служить документацією структури даних.

Продемонструємо, як інші сервіси звертаються до `components_service`. Припустимо, `ga_configurator_service` потребує отримати всі доступні процесори, щоб проаналізувати їх для підбору оптимального варіанту. У цьому випадку генетичний сервіс (через свій клієнт `components_client.py`) надішле HTTP-запит до API сервісу компонентів: наприклад, `GET /components?type=CPU`. В `app.py` сервісу компонентів цей запит буде зіставлений з відповідним обробником, який викличе функцію з `db.py` – припустимо, `get_components_by_type("CPU")`. Ця функція зчитає з `components.db` усі записи, де поле типу = "CPU", і поверне

список об'єктів або записів. Потім `app.py` серіалізує цей список у JSON (масив об'єктів з полями назв, характеристик тощо) і відправить у відповідь. `ga_configurator_service` отримає цю відповідь і зможе працювати з цими даними далі (наприклад, оцінювати продуктивність чи ціну процесорів). Сам `components_service` можна протестувати окремо: якщо розробник або тестувальник звернеться напряму до нього (наприклад, виконає локально `curl http://localhost:5001/components`), сервіс поверне повний каталог компонентів у форматі JSON.

Це підтверджує, що сервіс самодостатній і коректно надає дані без участі інших модулів. Така автономність дозволяє оновлювати і перевіряти базу знань компонентів незалежно: наприклад, додавши новий компонент у БД, достатньо викликати `GET /components/<новий_тип>` і переконатися, що він з'являється у відповіді.

3.3.3 Фронтенд (frontend)

Фронтенд відповідає за взаємодію з користувачем – це клієнтська частина системи ПК-конфігуратора. У найпростішому випадку, це веб-сторінка (або набір сторінок) з інтерфейсом, де користувач може ввести свої вимоги до комп'ютера, натиснути на певні кнопки для отримання результату і переглянути запропоновану конфігурацію. Фронтенд є "обличчям" системи: він відображає дані, отримані від бекенд-сервісів, у зручному для сприйняття вигляді (список компонентів, характеристики, ціни, поради тощо). Також він може виконувати базову валідацію вводу (наприклад, не дозволяти неправильні значення бюджету) перед тим, як відправити запит на сервер.

Розділення фронтенду і бекенду на окремі компоненти є стандартною практикою веб-розробки, особливо в контексті мікросервісів. Фронтенд виділено окремо, оскільки він реалізує зовсім інший шар системи – презентаційний. Це дозволяє розробляти інтерфейс незалежно від серверної логіки: дизайнери та фронтенд-інженери можуть змінювати вигляд сторінки, не втручаючись у роботу бекенд-сервісів, і навпаки, бекенд може змінювати API або алгоритми, не

вимагаючи змін у інтерфейсі (якщо зберігається сумісність API). Окремий фронтенд також спрощує розгортання: його можна розмістити на CDN чи окремому веб-сервері, оптимізувати для швидкого завантаження, тоді як бекенд-сервіси можуть працювати в хмарі або на сервері з API. У випадку з мікросервісною архітектурою нерідко використовують патерн Backend-for-Frontend (BFF) – спеціальний шар, який адаптує дані для потреб конкретного клієнтського інтерфейсу. У нашому випадку `chat_orchestrator_service` фактично виконує роль такого BFF, агрегуюючи дані для фронтенду. Таким чином, фронтенд залишається легким клієнтом, не обтяженим бізнес-логікою: вся логіка вичислень та обробки даних знаходиться на бекенді.

Фронтенд повністю ізольований від внутрішніх деталей бекенд-сервісів. Він спілкується з системою тільки через публічні API endpoints, які надають інші сервіси (передусім `chat_orchestrator_service`). Це означає, що фронтенд оперує лише HTTP-запитами (або WebSocket повідомленнями, якщо чат у реальному часі) до бекенду і отримує від нього вже опрацьовані результати. Будь-які залежності фронтенду – це, в основному, статичні файли (HTML, CSS, JavaScript) та, можливо, зовнішні бібліотеки для UI, але не бізнес-логіка застосунку. З точки зору структури проекту, фронтенд ізольований у власній папці і, ймовірно, може бути розгорнутий окремо. Його можна відкрити в браузері і він буде працювати (за умови доступності API) незалежно від деталей реалізації сервісів. Ізоляція також проявляється в тому, що фронтенд не має прямого доступу до баз даних або до файлової системи бекенду – всі дані проходять через контрольовані запити. Це підвищує безпеку і стабільність: помилки на UI-рівні не можуть напряму порушити роботу серверів, а збої на бекенді можуть бути відображені фронтендом як повідомлення, не ламаючи сам інтерфейс.

Вміст папки `frontend`:

- `index.html` – основна веб-сторінка застосунку. Це HTML-файл, що містить структуру інтерфейсу користувача. У ньому, ймовірно, присутні поля вводу для параметрів конфігурації (наприклад,

текстові поля для бюджету, перемикачі для вибору типу використання — ігри, робота, графічний дизайн тощо), елементи для відображення результату (секція, куди буде виведено список компонентів рекомендованої конфігурації), та елементи управління (кнопка "Підібрати конфігурацію"). Також `index.html` може містити підключення до файлів стилів (CSS) і скриптів (JavaScript). Скоріш за все, у ньому або в окремому JS-файлі прописана логіка здійснення запиту до бекенду: наприклад, при натисканні кнопки викликається JavaScript-функція, яка зчитає введені користувачем дані та відправить AJAX-запит (`fetch` або `XHR`) на адресу API оркестратора (`/configure` чи подібний `endpoint`). Потім отриманий результат (JSON з конфігурацією) ця функція обробить і відобразить у відповідних елементах сторінки (наприклад, заповнить таблицю компонентів). `index.html` виконує роль контейнера і базової розмітки, а динамічна взаємодія реалізована через скрипти, вбудовані або підключені в ньому. В академічному або курсовому проекті цей файл міг бути написаний вручну для демонстрації роботи системи: достатньо відкрити його в браузері, і, якщо бекенд сервіси запуснені, можна протестувати повний цикл роботи системи з точки зору користувача.

3.3.4 Сервіс генетичного конфігуратора (`ga_configurator_service`)

Сервіс генетичного конфігуратора реалізує механізм автоматичного підбору конфігурації ПК на основі заданих критеріїв, використовуючи генетичний алгоритм (GA). Функціонально цей сервіс отримує від користувача (через оркестратор) вимоги до системи – такі як бюджет, бажаний рівень продуктивності, сценарії використання (ігри, рендеринг відео, офісна робота тощо), а також, можливо, певні фіксовані вибори (наприклад, користувач хоче саме процесор Intel i7). У відповідь сервіс генерує оптимальний або близький до оптимального набір компонентів, який задовольняє ці вимоги. Генетичний алгоритм підбирає конфігурацію, еволюційно «еволюціонуючи» потенційні

рішення: він може оперувати популяцією можливих конфігурацій, оцінювати їх (функція пристосованості може враховувати сумарну ціну, продуктивність, сумісність компонентів, енергоспоживання та інші фактори), ітеративно покращувати рішення шляхом відбору, схрещування та мутацій. Результатом роботи є рекомендована конфігурація – наприклад, конкретна модель CPU, материнської плати, GPU, RAM і т.д., яка максимально відповідає критеріям. Таким чином, `ga_configurator_service` виступає як обчислювальний двигун системи: у той час як інші сервіси зберігають чи передають дані, цей сервіс виконує складний алгоритмічний пошук найкращого варіанту.

Генетичний конфігуратор винесено в окремий сервіс з кількох причин.

По-перше, це складна обчислювальна задача, яка може потребувати значних ресурсів CPU і пам'яті, особливо якщо набір компонентів великий і критерії комплексні. Запустивши її в окремому сервісі, ми можемо масштабувати його окремо (наприклад, виділити йому більше обчислювальних ресурсів або запустити декілька екземплярів для обробки паралельних запитів) без впливу на роботу інших сервісів.

По-друге, генетичний алгоритм – це спеціалізована логіка, яка може інтенсивно змінюватися під час досліджень та оптимізації (наприклад, коригування параметрів алгоритму, додавання нових критеріїв відбору). Винесення його з оркестратора дотримується принципу єдиної відповідальності: оркестратор займається лише координацією, а вся логіка оптимізації конфігурації зосереджена в одному місці.

По-третє, ізольований GA-сервіс дозволяє перевикористання: теоретично, інші клієнти або проєкти могли б запитувати оптимальні конфігурації (скажімо, інший інтерфейс або сценарій автоматичного тестування конфігурацій) без необхідності взаємодіяти з чат-інтерфейсом. Також, якщо згодом вирішено буде змінити метод оптимізації (наприклад, замінити генетичний алгоритм на повний перебір чи машинне навчання), це можна зробити, модифікуючи або замінюючи цей сервіс, не впливаючи на решту системи, доки зберігається той самий API контракт.

Генетичний конфігуратор винесено ізольований від інших частин системи таким чином, що він не напряму залежить від їх реалізації або баз даних, а отримує всю потрібну інформацію через чітко означені інтерфейси. Зокрема, для отримання даних про доступні компоненти він використовує клієнт до сервісу компонентів (`components_client.py`), замість прямого доступу до `components.db`. Це означає, що генетичний сервіс не знає деталей зберігання чи структури даних про компоненти — для нього це просто зовнішній ресурс, який надає перелік компонентів та їх властивості у формі, придатній для алгоритму. Аналогічно, `ga_configurator_service` не займається форматуванням кінцевої відповіді для користувача – він лише повертає сирий результат (наприклад, набір ID компонентів або їх назви), а вже оркестратор або фронтенд можуть представити це у зручному вигляді.

Таким чином, GA-сервіс сфокусований виключно на своїй задачі і його внутрішні залежності обмежені лише власними модулями (генетичний алгоритм, моделі) та зовнішнім API компонентів. Він, судячи з усього, не потребує власної бази даних: всі дані для розрахунку беруться з сервісу компонентів та, можливо, з вхідних параметрів користувача. Це спрощує ізоляцію – немає стану, який треба синхронізувати з іншими сервісами. Якщо GA-сервіс вимкнути або перезапустити, він не втратить важливих даних (бо всі знання приходять ззовні на запит). Комунікація з оркестратором (і потенційно прямо з фронтендом, якщо б хтось напряму звернувся) відбувається через HTTP; ні оркестратор, ні GA-сервіс не імпортують коду один одного. Така слабка зв'язаність означає, що зміни у генетичному алгоритмі (наприклад, додання нового критерію оптимізації) не зачіпають інші сервіси – лише, можливо, змінять дані, які повертаються, але формат і контракт залишаються сталі, тож інші сервіси цього навіть "не помітять".

Вміст папки `ga_configurator_service`:

- `app.py` – основний застосунок сервісу GA-конфігуратора. Тут відбувається запуск веб-сервера та визначення API ендпойнтів, через які зовнішні сервіси (як-от оркестратор) можуть звернутися до

генетичного алгоритму. Типовий сценарій – визначено маршрут, наприклад, `POST /generate_config`, який приймає JSON із вимогами до конфігурації. В тілі запиту очікуються поля: бюджет, бажані параметри, можливо вже обрані користувачем компоненти чи фіксовані вимоги. `app.py` обробляє цей запит: валідує дані (можливо, за допомогою моделей), викликає функцію або метод із `ga_core.py` для запуску генетичного алгоритму, та отримує результат (найкраща знайдена конфігурація). Результат серіалізується у зручний формат (JSON) і повертається у відповіді. Крім того, `app.py` може обробляти й інші маршрути, як-от отримання параметрів алгоритму чи тестові виклики. Він також, ймовірно, при старті ініціалізує клієнт до компонентів (щоб мати змогу викликати їх API).

- `data_components.py` - модуль-клієнт для взаємодії з `components_service`. Містить функції або класи, що відправляють HTTP-запити до сервісу компонентів і отримують потрібні дані. Наприклад, функція `get_all_components()` може робити запит до `GET /components` і повертати список усіх компонентів. Інша функція, можливо, `get_components_by_type(type)` – запит до `GET /components?type=...` і отримання конкретного типу компонентів. Цей клієнт слугує шаром абстракції: решта коду GA-сервісу (зокрема, `ga_core.py`) може викликати методи `components_client` для отримання даних, не турбуючись про деталі HTTP. Це відповідає принципу інверсії залежностей – генетичний алгоритм залежить від інтерфейсу (методів клієнта), а не від конкретної реалізації чи URL. Якщо, наприклад, зміниться адреса сервісу компонентів або формат запиту, правити треба буде тільки `components_client.py`.
- `ga_core.py` – ядро генетичного алгоритму. У цьому файлі зосереджена вся логіка оптимізаційного пошуку конфігурації. Очікується, що тут реалізовано: представлення рішення (кодування конфігурації у вигляді хромосоми, наприклад, як список з N позицій,

де кожна позиція відповідає категорії компонентів і зберігає ID або індекс вибраного компонента з цієї категорії), функція фітнесу (обчислення оцінки конфігурації – наприклад, негативна оцінка за перевищення бюджету і позитивна за продуктивність), а також оператори генетичного відбору, схрещування та мутації. `ga_core.py` може містити клас `GAConfigurator` або подібний, який при створенні завантажує список всіх компонентів (через `components_client.py` або `data_components.py`), фільтрує їх за категоріями, потім у методі `generate_config(requirements)` здійснює кілька етапів: генерацію початкової популяції випадкових конфігурацій, ітеративне застосування генетичних операторів, та повернення найкращого знайденого рішення. Також, якщо алгоритм вимагає значного часу, можуть бути реалізовані критерії зупинки (наприклад, число поколінь або відсутність покращення). У цьому ж файлі або в `models.py` визначені структури для представлення конфігурації, але є ймовірність, що все локально. Код у `ga_core.py` ретельно працює з даними: перевіряє сумісність кожної згенерованої конфігурації (щоб, скажімо, не поєднати процесор AMD з материнкою для Intel), рахує сумарну вартість, порівнює з бюджетом. Закінчуючи роботу, алгоритм вибирає конфігурацію, що максимально відповідає вимогам, та віддає її у вигляді структури даних (список компонентів або об'єкт). Цей модуль є «мозком» сервісу, і основна складність системи закладена саме тут.

- `models.py` – містить моделі даних, специфічні для GA-сервісу. По-перше, модель вимог користувача (щоб зручно передавати їх всередині системи): клас, який містить поля `budget`, `purpose`, `perhaps constraints` тощо, і допоміжні методи (наприклад, парсинг з JSON запиту). По-друге, модель конфігурації – можливо, клас `Configuration` з полями: список компонентів або окремі поля для ключових категорій (`cpu`, `gpu`, `motherboard`, `ram`,...), загальна ціна,

оцінка (фітнес). Цей клас полегшує передачу конфігурацій між функціями, а також форматування результату. Наприклад, `app.py` може отримати від `ga_core.py` об'єкт `Configuration` і викликати метод цього об'єкта `to_json()` для серіалізації. Крім того, `models.py` може містити допоміжні типи, як-от перерахування (`Enum`) для типів компонентів, або структури для проміжних даних алгоритму. Моделі в цьому контексті не обов'язково пов'язані з базою даних (як у сервісі компонентів), а радше з доменною моделлю завдання оптимізації.

Розглянемо послідовність викликів, пов'язану з GA-сервісом. Коли користувач запитує конфігурацію (через оркестратор), `chat_orchestrator_service` надсилає до `ga_configurator_service` запит, що містить необхідні параметри. Припустимо, це POST-запит на `/generate_config` з тілом: `{ "budget": 1000, "usage": "gaming", "pref_cpu_brand": "Intel" }`. В `app.py` GA-сервісу цей JSON розбирається: можливо, за допомогою моделі з `models.py` створюється об'єкт `UserRequirements(budget=1000, usage="gaming", pref_cpu_brand="Intel")`.

Далі `app.py` викликає основний метод генетичного алгоритму – ймовірно, `GAConfigurator.generate_configuration(requirements)`, визначений в `ga_core.py`. Ця функція спочатку звернеться до `components_client.py`, щоби отримати дані про всі доступні компоненти. Наприклад, викликається `components_client.get_all_components()` і завантажує список всіх компонентів з сервісу компонентів.

Потім `ga_core.py` відфільтрує їх по категоріях: виділить список всіх процесорів, всіх GPU, матерінських плат тощо. Генетичний алгоритм створить початкову популяцію випадкових конфігурацій: кожна конфігурація – це вибір випадкового CPU, випадкової материнки сумісної з цим CPU, випадкової відеокарти, RAM і т.д., що задовольняють базові обмеження сумісності. Кожна конфігурація оцінюється функцією фітнесу – наприклад, нараховується штраф, якщо ціна перевищує 1000, і додаються бали за потужність GPU, обсяг RAM тощо, залежно від `usage "gaming"`. Алгоритм кілька поколінь модифікує

популяцію: комбінує успішні рішення (можливо, бере половину компонентів від однієї конфігурації, половину від іншої – схрещування), випадково змінює деякі компоненти (мутація), і відбирає кращі. Припустимо, після 50 поколінь він знаходить конфігурацію, яка в межах бюджету 1000 USD дає максимальну продуктивність у іграх (наприклад, має потужну відеокарту при прийнятному процесорі, достатньо RAM). Ця конфігурація формується як об'єкт Configuration і повертається з `ga_core.py` назад у `app.py`. `app.py` перетворює її у відповідь (можливо, як JSON: `{"cpu": "Intel i5-12400", "gpu": "NVIDIA RTX 3060", "motherboard": "...", "total_price": 980, ...}`) і надсилає оркестратору. Оркестратор, у свою чергу, може доповнити відповідь додатковими даними чи передати прямо фронтенду. Пряма робота:

Ми також можемо викликати `ga_configurator_service` без оркестратора, напряму, для тестування. Наприклад, відправивши запит через `curl` на локальний адресу цього сервісу з відповідними параметрами, ми отримаємо від нього конфігурацію. Це дає змогу перевірити якість генетичного алгоритму незалежно: змінюючи критерії чи обмеження, ми одразу бачимо, як зміниться вихід. Таким чином, GA-сервіс демонструє свою працездатність як окремий модуль, який можна налаштовувати і вдосконалювати ізольовано (наприклад, оптимізувати його параметри, не впливаючи на інші частини системи).

3.3.5 Сервіс Gemini (`gemini_service`)

Сервіс Gemini інтегрує в систему можливості, пов'язані з передовими моделями штучного інтелекту (ШІ), зокрема з моделлю (або API) під кодовою назвою Gemini. Судячи з назви, можна припустити, що йдеться про використання когнітивних або мовних можливостей моделі Google Gemini чи подібної великої моделі для покращення роботи конфігуратора. Функціонально цей сервіс може виконувати декілька можливих ролей:

- Аналіз запитів у природній мові, якщо користувач формулює свої потреби розмовною мовою (наприклад, "Мені потрібен тихий комп'ютер для відеомонтажу"), сервіс Gemini може відповісти за

обробку такого тексту. Він здатний витягти структуровані вимоги (тихий = потреба в хорошому охолодженні, відеомонтаж = потрібен потужний процесор та багато пам'яті) і передати їх оркестратору або GA-сервісу.

- Генерація рекомендацій та пояснень, де модель ШІ може згенерувати людинозрозумілі пояснення до зібраної конфігурації ("Цей ПК оснащено відеокартою RTX 3060, оскільки для ігор вона забезпечує високу частоту кадрів в зазначеному бюджеті...") або альтернативні поради. Це підвищує цінність відповіді для користувача.
- Перевірка сумісності та логіки на основі знань, де модель може виступати додатковим валідатором, використовуючи знання з великого корпусу даних. Наприклад, якщо GA-сервіс запропонує конфігурацію, Gemini може "проверити" її на предмет сумісності чи потенційних проблем (хоча це залежить від здатності моделі і навчання).
- Діалогова взаємодія у чат-сценарії Gemini може підтримувати діалог з користувачем, відповідаючи на запитання чи уточнення щодо конфігурацій, працюючи як інтелектуальний співрозмовник.

Таким чином, `gemini_service` забезпечує інтелектуальний шар системи, використовуючи можливості AI для розуміння або генерації тексту, які виходять за межі жорстко запрограмованої логіки.

Винесення інтеграції з моделлю ШІ в окремий сервіс має низку причин. По-перше, робота з великою мовною моделлю (LLM) зазвичай вимагає спеціалізованих залежностей і може бути ресурсоємною. В окремому сервісі можна ізолювати усі необхідні бібліотеки для виклику зовнішнього AI API (наприклад, авторизація до Google API, формування промптів, обробка великих текстів) і забезпечити, що інші частини системи не залежать від цих бібліотек. По-друге, модель ШІ може використовуватися і поза контекстом конфігуратора – скажімо, для загального чат-бота чи інших завдань. Тому логічно мати її окремо, щоб повторно використовувати у разі потреби. По-третє, з міркувань

безпеки та стабільності: виклики до зовнішнього AI-сервісу можуть бути ненадійними (затримки, ліміти використання, зміни API). Якщо винести їх у окремий мікросервіс, то збої чи повільна робота моделі не будуть прямо блокувати, наприклад, роботу оркестратора – той може отримати таймаут і, наприклад, продовжити без AI-даних або видати повідомлення. Це ізолює ризики, пов'язані з інтеграцією ШІ. Нарешті, командна організація: можливо, інтеграцією ШІ займається окрема група, і їм зручніше працювати в межах самостійного мікросервісу. З врахуванням усього, `gemini_service` реалізує окрему субдоменну функціональність – AI-підказки – яка доповнює основні алгоритми системи, але не переплітається з ними на рівні коду.

Прикладом мікросервісу, ізолюваного як за даними, так і за залежностями є `gemini_service`. Він, найімовірніше, не має власної постійної бази даних (йому це не потрібно, адже він переважно виконує обчислення/запити на льоту). Натомість, його головна залежність – зовнішній сервіс (API моделі Gemini) або бібліотека машинного навчання – інкапсульована всередині. Ніякий інший сервіс системи безпосередньо не викликає сторонній AI API; вони звертаються лише до `gemini_service`.

Таким чином, деталі взаємодії з ШІ (такі як формати запитів, токени автентифікації, параметри моделі) заховані всередині цього сервісу. Він надає іншим частинам системи простий інтерфейс, можливо щось на кшталт: "отримати інтерпретацію вимог" або "згенерувати опис конфігурації". Інші сервіси, наприклад оркестратор, не знають, чи `gemini_service` сам реалізує ці функції, чи звертається до Google API – для них це чорний ящик. Це повністю відповідає принципу приховування реалізації за інтерфейсом.

Окрім того, ізоляція проявляється у конфігурації середовища: якщо для Gemini API потрібні певні ключі API або моделі, вони налаштовуються тільки в оточенні `gemini_service`. Можна оновлювати версію моделі чи навіть переключитися з однієї моделі на іншу, змінивши код цього сервісу, без необхідності модифікувати щось у інших компонентах. З точки зору збоїв: якщо `gemini_service` недоступний або відповідає з помилкою (скажімо, через ліміт

запитів до AI), решта системи продовжить роботу – оркестратор може або використати запасний шлях (наприклад, обійтися без пояснень ШІ), або повідомити користувача, але інші функції (генетичний алгоритм, база компонентів) залишаються працездатними. Така стійкість до помилок досягається саме ізоляцією сервісу Gemini.

Вміст папки `gemini_service`:

- `app.py` – головний файл сервісу Gemini. У ньому налаштовано веб-застосунок (як правило, Flask або FastAPI), що приймає HTTP-запити, пов'язані з функціональністю ШІ. Залежно від реалізації проекту, у `app.py` можуть бути визначені такі маршрути:
 - 1) `POST /analyze_requirements` – приймає текст користувацького запиту (або вже структуровані дані) і повертає структуровану інтерпретацію. Наприклад, на вхід надсилається: "Мені потрібен ПК для дизайну та ігор, до \$1500", а на виході модель повертає: `{ "budget": 1500, "primary_use": "gaming", "secondary_use": "design", "priority": "balance" }` – тобто витягнуті суттєві характеристики.
 - 2) `POST /explain_configuration` – приймає конфігурацію (список компонентів) і повертає згенерований мовний опис або обґрунтування цієї конфігурації.
 - 3) `POST /chat` – якщо передбачена діалогова взаємодія, цей endpoint міг би надсилати репліку користувача моделі і отримувати відповідь (як чат-бот).
- `gemini_client.py` – модуль-клієнт для безпосередньої роботи з зовнішньою моделлю Gemini (чи API). Його призначення – інкапсулювати всю взаємодію з AI-сервісом. Наприклад, тут може бути клас `GeminiAPI` з методами `analyze(text)` або `generate_response(prompt)`. У середині цих методів здійснюється виклик до зовнішнього API: формування HTTP-запиту або використання SDK, відправка запиту з потрібними параметрами

моделі (наприклад, вибір конкретної версії моделі, температура генерації тощо) і отримання відповіді. `gemini_client.py` також обробляє можливі помилки (наприклад, якщо API повернуло помилку, перетворює це на виняток або спеціальний результат). Важлива частина – автентифікація: цей модуль може завантажувати ключ API або токен доступу з конфігурацій (можливо, з змінних середовища) та підставляти його в запити, щоб авторизуватися на боці AI-провайдера. До того ж, `gemini_client.py` може займатися конструкцією промптів: наприклад, якщо треба отримати структуровану відповідь, клієнт може додати до користувачького тексту спеціальну інструкцію, типу "витягни ключові параметри і поверни JSON". Такі деталі залежать від конкретного API. В ідеалі, `gemini_client.py` на виході повертає зручні для `app.py` дані – або готовий текст, або вже розібраний JSON від моделі (якщо модель це підтримує). Таким чином, основний код сервісу не містить жодних URL або секретних ключів – все зосереджено тут. Якщо завтра модель Gemini зміниться, або постане задача перемкнутися на інший AI (наприклад, локальну модель), зміни торкнуться лише цього клієнтського модуля.

- `models.py` – моделі даних для сервісу Gemini. Схоже на попередні служби, тут можуть бути визначені структури, які використовуються при взаємодії з користувачами чи іншими сервісами. Наприклад, клас `RequirementsAnalysis` з полями (бюджет, призначення тощо), який репрезентує результат аналізу запиту користувача. Або модель `ConfigurationExplanation` з полями (текст пояснення, список посилань чи рекомендацій). Також `models.py` може містити моделі для запитів: скажімо, якщо `app.py` приймає JSON для `/chat` з полями `message` і `context`, може бути клас `MessageRequest` для валідації. Іншими словами, цей файл визначає контракти даних, з якими оперує `gemini_service`, що допомагає підтримувати строгість типізації і

форматів. Особливо у разі FastAPI, можна мати Pydantic-моделі для автоматичної генерації документації і перевірки вхідних даних.

Покажемо, як оркестратор може скористатися сервісом Gemini. Уявімо, що користувач задає складне запитання в чаті: "Мені потрібна дуже тиха збірка для 3D-моделювання, але щоб і в ігри можна було грати, доларів до 2500. Що порадиш?". Оркестратор може вирішити спершу звернутися до gemini_service для інтерпретації цього запиту, оскільки він містить неявні вимоги ("дуже тиха" – значить, бажано водяне охолодження або безшумні компоненти; "3D-моделювання і ігри" – потрібен баланс між CPU та GPU, високий об'єм RAM тощо). Оркестратор робить HTTP-запит до маршруту, припустімо, /analyze_requirements сервісу Gemini, передаючи текст запиту. gemini_service отримує його, і gemini_client.py відправляє підготовлений промпт до моделі Gemini (наприклад, з інструкцією: "Extract the budget, primary_use, secondary_use, and key requirements from this user request and output in JSON"). Модель повертає структуру, і app.py надсилає оркестратору щось таке: { "budget": 2500, "primary_use": "3D modeling", "secondary_use": "gaming", "requirements": ["very quiet"] }. Оркестратор, озброївшись цією інформацією, може передати її в GA-сервіс, щоб той підбирав конфігурацію з урахуванням primary_use і requirements.

Після того, як GA-сервіс поверне згенеровану конфігурацію, оркестратор може знов звернутися до gemini_service, цього разу до endpoint, який генерує пояснення. Він надсилає, наприклад, { "configuration": {...} } на /explain_configuration. gemini_service через gemini_client.py просить модель згенерувати пояснювальний текст. Модель Gemini створює, скажімо, абзац тексту: "Обрана відеокарта RTX 4070 забезпечить високий FPS в іграх і швидкий рендеринг у 3D-програмах. Процесор Intel i7 надасть достатню обчислювальну потужність для моделювання, а система водяного охолодження Kraken X забезпечить безшумність...". Цей текст повертається оркестратору, який далі передає його фронтенду. Користувач у веб-інтерфейсі отримує не лише список компонентів, але й дружнє пояснення, написане мовою, завдяки gemini_service.

Розробник системи може окремо протестувати можливості AI-сервісу,

використовуючи `test_request.json`. Наприклад, він запускає локально `gemini_service` і виконує: `curl -X POST -H "Content-Type: application/json" -d @test_request.json http://localhost:5003/analyze_requirements`. У відповідь отримує структурований JSON із виділеними параметрами.

Це підтверджує, що модель правильно розуміє запит. За необхідності, `test_request.json` можна відредагувати, змінюючи формулювання, і таким чином покращувати промпт або обробку у `gemini_client.py`. Подібно, можна напямку викликати `/chat endpoint` для інтерактивного тесту: відправити запит, отримати відповідь моделі (наприклад, уточнення чи додаткові питання, якщо модель налаштована на діалог). Ці можливості показують, що `gemini_service` повністю функціонує як незалежний AI-модуль, який можна розвивати чи масштабувати окремо від основного конфігуратора.

3.4 Опис інтерфейсу користувача

Графічний інтерфейс користувача розробленого веб-застосунку реалізовано у вигляді односторінкового веб-інтерфейсу, що працює за принципом динамічного оновлення вмісту без перезавантаження сторінки. Основна увага приділена простоті, зрозумілості та мінімальній кількості дій з боку користувача. Робочий простір поділено на дві ключові зони: панель взаємодії з користувачем (ліва частина) та панель візуалізації результатів конфігурування (права частина), що проілюстровано на рис. 3.4.

Ліва частина екрану слугує основною зоною взаємодії користувача із системою. Вона об'єднує засоби формалізації запиту та чатоподібний інтерфейс для обміну повідомленнями з конфігуратором, а саме:

1. Блок параметризації запиту (верхня частина). У верхній частині панелі розміщено елементи керування, які дозволяють явно задати основні параметри задачі конфігурування:

- 1) поле вводу текстового запиту - основний засіб взаємодії, де користувач у довільній формі описує свої потреби (наприклад: «Хочу зібрати ПК для ігор до 50 000 грн з невеликим запасом на

майбутнє»). Це повідомлення надсилається до серверної частини і використовується для аналізу намірів та генерації рекомендацій.

2) поля параметрів (фільтри) - для підвищення керованості процесу конфігурування інтерфейс передбачає окремі поля для явного задання. У разі заповнення ці параметри мають пріоритет над значеннями, які можуть бути автоматично витягнуті з текстового запиту:

- бюджетних обмежень (поле «Бюджет» у гривнях);
- цільового сценарію використання (наприклад, Gaming, Work, Mixed).

3) перемикач режиму роботи (режим збірки) - окремий елемент інтерфейсу дає змогу обрати між:

- режимом побудови нової конфігурації (new).
- режимом модернізації наявної системи (upgrade).

Обраний режим впливає на логіку роботи генетичного алгоритму: у випадку апгрейду частина компонентів фіксується як уже наявна, а оптимізація виконується лише щодо решти складових.

4) поле для опису наявної конфігурації - для режиму модернізації передбачено спеціальне поле, у яке користувач може ввести перелік уже встановлених компонентів у структурованому форматі (наприклад, JSON-опис). Це дозволяє системі коректно враховувати поточний стан ПК і розраховувати саме вартість апгрейду, а не повної збірки з нуля.

Такий поділ дає змогу поєднати два підходи до взаємодії: інтуїтивний (через текстовий опис) і формалізований (через явні параметри). Навіть за відсутності заповнених полів система може інтерпретувати намір користувача на основі тексту, але наявність окремих елементів керування забезпечує більший ступінь контролю.

2. Чат-зона (нижня частина). Нижню частину лівої панелі займає чат-

область, у якій відображається історія діалогу між користувачем та системою.

Після обробки запиту застосунок виводить:

- 1) текстову відповідь у форматі, зручному для читання (із використанням заголовків, списків та виділень);
- 2) узагальнену рекомендацію щодо однієї або кількох конфігурацій;
- 3) пояснення, чому саме ці варіанти є доцільними з огляду на бюджет, ціль використання та обмеження користувача.

Відповіді формуються модулем мовної моделі та подаються у вигляді зв'язного пояснювального тексту, що дозволяє користувачеві не лише бачити «сухі» параметри конфігурації, а й розуміти логіку вибору.

Права частина інтерфейсу призначена для наочного представлення результатів роботи системи. Тут відображаються варіанти зібраних конфігурацій, сформовані генетичним алгоритмом, у вигляді окремих карток.

Кожна картка конфігурації містить:

- перелік ключових компонентів (процесор, відеокарта, оперативна пам'ять, накопичувачі, блок живлення, корпус, система охолодження);
- вартість кожного компонента та загальну ціну збірки;
- у режимі модернізації — додатковий показник вартості апгрейду (вартість лише тих компонентів, які необхідно докупити);
- службові теги (наприклад, `best_overall`, `fits_budget`, `over_budget_total`), які коротко характеризують тип конфігурації.

Серед усіх варіантів одна збірка візуально виділяється як рекомендована (наприклад, за допомогою підсвічування або позначки *Recommended*). Вона відповідає найвищому значенню цільової функції (фітнес-функції) з урахуванням обмежень користувача. Інші картки можуть відображати альтернативні стратегії, наприклад:

- більш бюджетний варіант із дещо нижчою продуктивністю;
- варіант із вищою продуктивністю за рахунок незначного перевищення бюджету.

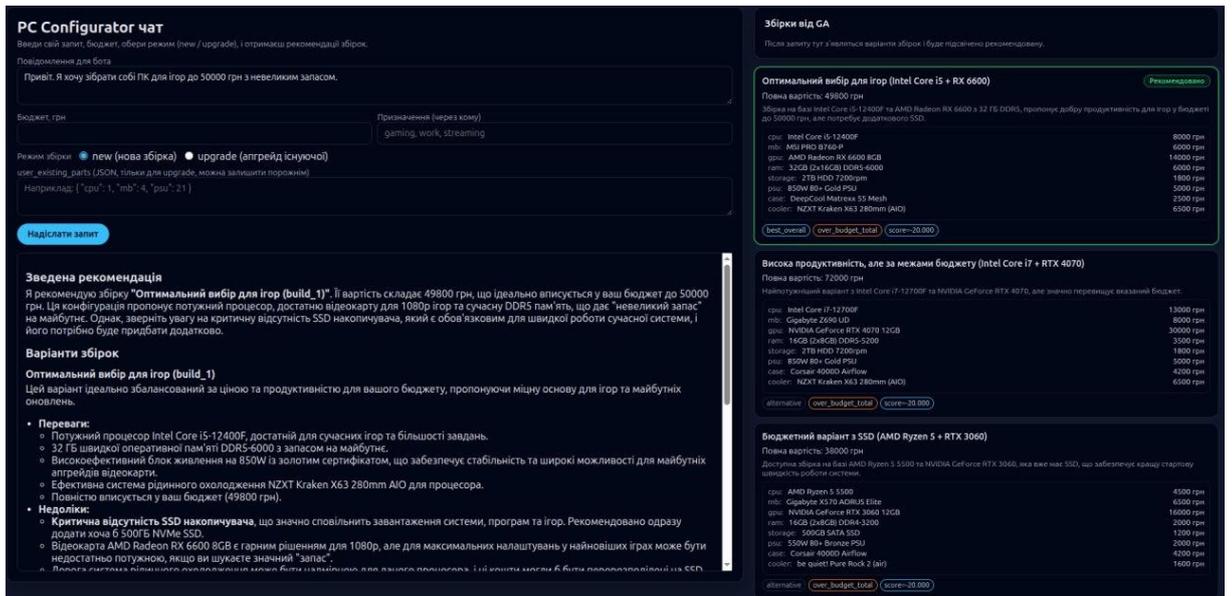


Рис 3.4 Інтерфейс користувача

Такий спосіб візуалізації дає можливість швидко порівняти кілька конфігурацій за основними характеристиками та прийняти обґрунтоване рішення.

Запропонована структура інтерфейсу забезпечує чітке розділення відповідальностей між зонами: ліва панель зосереджена на формуванні й уточненні запиту, тоді як права панель демонструє результати обчислень і дає змогу наочно порівняти альтернативи. Це підвищує зручність роботи з системою й робить процес підбору конфігурації максимально прозорим для користувача.

3.5 Результати експериментального дослідження ефективності методики

Для оцінки якості роботи розробленого програмного комплексу було проведено серію порівняльних тестів із існуючими аналогами (ручний підбір, магазинні конфігуратори, LLM-чати). Аналіз проводився за ключовими метриками продуктивності та якості. Отримані результати демонструють суттєві переваги запропонованої гібридної методики в наведеній таблиці 3.1.

Порівняльний аналіз ефективності розробленої методики

Метод	Швидкість підбору	Точність та сумісність збірки	Кількість запропонованих збірок	Принцип формування	Автоматизація підбору	Тип звітності
Ручний підбір	Дуже низька (хв/години/дні)	30-80% (залежить від знань)	1	Суб'єктивний ручний вибір	-	Відсутня (власні нотатки)
Конфігуратори магазинів (Telemart UA)	Середня (хвилини/години)	75-90%	1	Послідовний вибір з каталогу	-	Список товарів (Кошик)
LLM-чат (ChatGPT)	Висока (~20-25 сек.)	60-85%	1-3	Ймовірнісна генерація тексту	+	Неструктурований текст
Розроблена методика	Висока (~10-15 сек.)	96-100%	3	Гібридного алгоритму за допомогою ГА та LLM	+	Структурована (JSON + Пояснення)

Вдалося досягти високої швидкодії системи. Час генерації повної відповіді (включаючи парсинг, роботу Генетичного алгоритму та генерацію пояснень) становить 10–15 секунд.

- Це у 1.5-2 рази швидше, ніж отримання текстової відповіді від звичайного ChatGPT (20-25 с), і на порядки швидше за ручний пошук, який може займати години.

Завдяки впровадженню системи «заборонних штрафів» у математичну модель, точність підбору становить 96-100%.

- Порівняння: Система гарантує фізичну та логічну сумісність

компонентів, усуваючи ризик помилок. Для порівняння, точність «чистих» LLM (ChatGPT) варіюється в межах 60–85%, оскільки вони схильні до генерації неіснуючих характеристик («галюцинацій»).

На відміну від магазинних конфігураторів, які зазвичай пропонують один варіант кошика, розроблена система автоматично генерує 3 альтернативні збірки (наприклад: «Рекомендована», «Бюджетна», «Продуктивна»). Це надає користувачеві простір для вибору без необхідності повторних запитів.

Результат роботи системи надається у структурованому форматі: це комбінація технічних даних (JSON для карток товарів) та аргументованого текстового пояснення. Це вирішує проблему неструктурованості відповідей звичайних чат-ботів та «сухості» списків у інтернет-магазинах.

Реалізація гібридного алгоритму (GA + LLM) дозволила створити повністю автоматизований інструмент, який поєднує швидкість машинної обробки з якістю експертної оцінки, перевершуючи існуючі аналоги за сукупністю показників точності та зручності.

ВИСНОВКИ

У межах дослідження було здійснено глибокий аналіз існуючих підходів до підбору комп'ютерних комплектуючих. Виявлено їхні ключові недоліки, зокрема орієнтацію виключно на нові збірки, відсутність механізмів адаптації під модернізацію вже наявної системи та обмежену ефективність при жорстких бюджетних обмеженнях.

На основі виявлених проблем було здійснено формалізацію задачі конфігурування персонального комп'ютера як задачі багатокритеріальної оптимізації. Для її розв'язання запропоновано гібридний підхід, який поєднує алгоритмічну складову у вигляді генетичного алгоритму (ГА) та інтелектуальну мовну модель (LLM) для обробки запитів користувача та генерації пояснень.

Розроблено методику автоматизованого підбору конфігурацій, яка базується на взаємодії між декількома мікросервісами. Генетичний алгоритм виконує математичну оптимізацію на основі обмежень сумісності, бюджету та продуктивності, тоді як LLM-модуль (на основі Gemini API) виконує парсинг намірів користувача і надає пояснення до згенерованих збірок.

У межах реалізації було використано такі інструменти та бібліотеки: мова програмування Python як основа бекенду; SQLite — для зберігання локальної бази даних компонентів; FastAPI + Uvicorn — для реалізації мікросервісної архітектури; Pydantic — для валідації даних та обміну у форматі JSON; Gemini API — для реалізації мовної взаємодії з користувачем.

Методика впроваджена у вигляді інтерактивного веб-додатку, що дозволяє в режимі діалогу формувати запити, отримувати рекомендації та візуалізувати результати. Інтерфейс реалізовано у вигляді SPA-додатку з логічним поділом на панель введення та область результатів.

Проведене експериментальне дослідження підтвердило ефективність запропонованої системи. Методика забезпечує високу швидкодію (10–15 секунд для повного запиту), технічну коректність (96–100% сумісних збірок), адаптивність до різних сценаріїв (завдяки поєднанню ГА та LLM) та якісну

звітність у структурованому форматі (JSON + текстові пояснення).

Результати дослідження апробовано та опубліковано у наступних статті та тезах:

1. Кочетков О.В., Шевченко С.М. Багатокритеріальна оптимізація для вибору найкращої конфігурації комп'ютерних систем // Вісник Херсонського національного технічного університету» № 4, 2025. Прийнято до друку.

2. Кочетков О.В., Шевченко С.М. Розробка алгоритму підбору комплектуючих для комп'ютерних систем з урахуванням бюджетних обмежень та наявної апаратної конфігурації. VI Всеукраїнська науково-технічна конференція «Сучасний стан та перспективи розвитку IoT», 15 квітня 2025 року. Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. – К.: ДУІКТ, 2025. С. 158-160.

3. Кочетков О.В., Шевченко С.М. Визначення вимог до алгоритму підбору комплектуючих для комп'ютерних систем з урахуванням бюджетних обмежень та наявної апаратної конфігурації. VI Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 24 квітня 2025 року. Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. – К.: ДУІКТ, 2025. С.558-560.

ПЕРЕЛІК ПОСИЛАНЬ

1. Kaur L., Singh H. Software Component Selection Techniques – A Review. International Journal of Computer Science and Information Technologies. – 2015. Vol. 6, No. 2. С. 1568–1574.
2. Michael M., Winarno W. Design and Development of Computer Specification Recommendation System Based on User Budget With Genetic Algorithm // International Journal of New Media Technology. 2018. Vol. 5, No. 1. С. 25–29.
3. Hagan M.T. Demuth H.B., Beale M.H., De Jesús O. Neural Network Design. 2nd Edition. – Martin Hagan. 2014. 800 p.
4. Russell S., Norvig P. Artificial Intelligence: A Modern Approach. 2010. 1132p.
5. Russell S., Norvig P. Artificial Intelligence: A Modern Approach. 2020. 1152p.
6. Holland J.H. Adaptation in Natural and Artificial Systems. 1975. 228 p.
7. Mitchell M. An Introduction to Genetic Algorithms. 1996. 209 p.
8. Deb K., Pratap A., Agarwal S., Meyarivan T. A fast and elitist multiobjective genetic algorithm: NSGA-II // IEEE Trans. Evol. Comput. 2002. Vol. 6, No. 2. С. 182-197. [Електронний ресурс] URL: <https://www.jmlr.org/papers/volume13/fortin12a/fortin12a.pdf#:~:text=algorithm%20for%20multi,adaptation%20in%20evolution%20strategies>.
9. Кононюк А.Ю. Нейронні мережі і генетичні алгоритми. 2010. 256 с.
10. Goldberg D.E. Genetic Algorithms in Search, Optimization and Machine Learning. 1989. 412 p.
11. Gerges F., Zouein G., Azar D. Genetic Algorithms with Local Optima Handling to Solve Sudoku Puzzles. 2018. С. 19–22.
12. Бажан С.М. Застосування генетичних алгоритмів із випадковими процесами для оптимізації багатоекстремальних функцій. Міжнародний науковий журнал ‘Грань науки’. 2023. №33. С. 253–261.
13. Deb K. Multi-Objective Optimization using Evolutionary Algorithms. –

John Wiley & Sons, 2001. 518 p.

14. Eiben A.E., Smith J.E. Introduction to Evolutionary Computing. 2015. 287 p.

15. Grefenstette J.J., Gopal R., Rosmaita B., Van Gucht D. Genetic Algorithms for the Traveling Salesman Problem. 1985. С. 160–168.

16. Jurafsky D. Martin J.H. Speech and Language Processing. 2022. 1024 p.

17. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. – Addison-Wesley, 1994. 395 p.

18. FastAPI Documentation [Электронный доступ] // FastAPI. - 2025. – Режим доступа до ресурсу: <https://fastapi.tiangolo.com/>

19. Gemini API Documentation [Электронный доступ] // Google AI for Developers. - 2025. – Режим доступа до ресурсу: <https://ai.google.dev/gemini-api/docs>

20. Pydantic Documentation [Электронный доступ] // Pydantic. - 2025. – Режим доступа до ресурсу: <https://docs.pydantic.dev/>

21. Uvicorn: An ASGI web server, for Python [Электронный доступ] // Uvicorn. - 2025. – Режим доступа до ресурсу: <https://www.uvicorn.org/>

22. SQLite Documentation [Электронный доступ] // SQLite. - 2025. – Режим доступа до ресурсу: <https://www.sqlite.org/docs.html>

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ



ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ
ТЕХНОЛОГІЙ



НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ
ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Магістерська робота

«Методика підбору комплектуючих для комп'ютерних систем з урахуванням
бюджетних обмежень та наявної апаратної конфігурації»

Виконав: студент групи ПДМ-62 Олексій КОЧЕТКОВ

Керівник: канд., пед., наук., доц., доцент кафедри ІІЗ Світлана ШЕВЧЕНКО

Київ - 2025

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: підвищення ефективності та якості конфігурування комп'ютерних систем за допомогою методики автоматизованого підбору комплектуючих з урахуванням бюджетних обмежень та наявної апаратної конфігурації.

Об'єкт дослідження: процес конфігурування та модернізації апаратного забезпечення персональних комп'ютерних систем.

Предмет дослідження: методи та засоби підбору комплектуючих для комп'ютерних систем з урахуванням бюджетних обмежень та наявної апаратної конфігурації.

ПОРІВНЯЛЬНА ХАРАКТЕРИСТИКА ІСНУЮЧИХ МЕТОДІВ ПІДБОРУ КОМПЛЕКТУЮЧИХ ДЛЯ ПК

Метод	Принцип роботи	Ключові недоліки
Ручний підбір	Самостійний підбір комплектуючих за оглядами, форумами, порадами знайомих, порівнює характеристики та ціни на різних сайтах.	<ul style="list-style-type: none"> Високі часові витрати Потреба у глибоких знаннях Високий ризик помилок сумісності
Конфігуратори магазинів (Rozetka, Telemart)	Поетапний вибір з каталогу магазину з базовою перевіркою сумісності.	<ul style="list-style-type: none"> Немає оптимізації «ціна/якість»
Вузькопрофільні конфігуратор (be quiet! PSU calc)	Розрахунок окремих параметрів (напр., потужності БЖ) для заданої системи.	<ul style="list-style-type: none"> Вирішують лише локальну задачу Не підбирають основні компоненти
Прайс-агрегатори (Hotline, E-Katalog)	Пошук за фільтрами цін та параметрів у різних продавців.	<ul style="list-style-type: none"> Компоненти підбираються ізольовано Сумісність перевіряється вручну Складно збалансувати збірку
LLM-чати (ChatGPT, Gemini)	Поради на основі текстового запиту	<ul style="list-style-type: none"> Ризик галюцинацій Неактуальні ціни Технічні помилки сумісності
Розроблена методика (Гібридна система GA+LLM)	Поєднує зручність чату з точністю математики. Розуміє ваші бажання та рахує сумісність та бюджет з інженерною точністю.	-

3

МАТЕМАТИЧНА МОДЕЛЬ АВТОМАТИЗОВАНОГО ПІДБОРУ КОМПЛЕКТУЮЧИХ ДЛЯ КОМП'ЮТЕРНИХ СИСТЕМ

Критерій якості для конфігурації x :

$$F(x) = w_p \cdot P(x) + w_e \cdot E(x) - \sum_k M_k \cdot I_k(x)$$

Де:

x — вектор компонентів збірки.

$P(x)$ — функція оцінки продуктивності.

$E(x)$ — функція економічної ефективності (відповідності бюджету).

$I_k(x)$ — індикатори порушення технічних обмежень (сумісності).

w_p, w_e — вагові коефіцієнти продуктивності та ціни.

M_k — заборонні штрафні коефіцієнти для жорстких обмежень.

Математична умова для M_k :

$$M_k \gg \max(w_p \cdot P(x) + w_e \cdot E(x))$$

4

ДЕТАЛІЗАЦІЯ СКЛАДОВИХ МОДЕЛІ

Функція продуктивності, де сумуються бали CPU та GPU з нормалізацією та бонусом за охолодження:

$$P(x) = \frac{Perf_{cpu}(x) + Perf_{gpu}(x)}{N} + \delta_{cooler}(x)$$

$Perf_{...}$ — бали бенчмарків компонентів; N — коефіцієнт нормалізації;

δ_{cooler} — бонусний коефіцієнт за ефективність охолодження.

Функція вартості оновлення:

$$C_{new}(x) = \max(0, C_{total}(x) - C_{locked})$$

$C_{total}(x)$ — повна вартість компонентів;

C_{locked} — сумарна вартість компонентів, які користувач вирішив залишити від старого ПК.

Функція економічної ефективності:

$$E(x) = \begin{cases} 1 - \frac{C_{new}(x)}{B}, & \text{якщо } C_{new}(x) \leq B \\ -\frac{C_{new}(x) - B}{B}, & \text{якщо } C_{new}(x) > B \end{cases}$$

Якщо збірка дешевша за бюджет $E(x) \rightarrow 1$

Якщо дорожча, $E(x)$ стає від'ємним.

5

СИСТЕМА ОБМЕЖЕНЬ

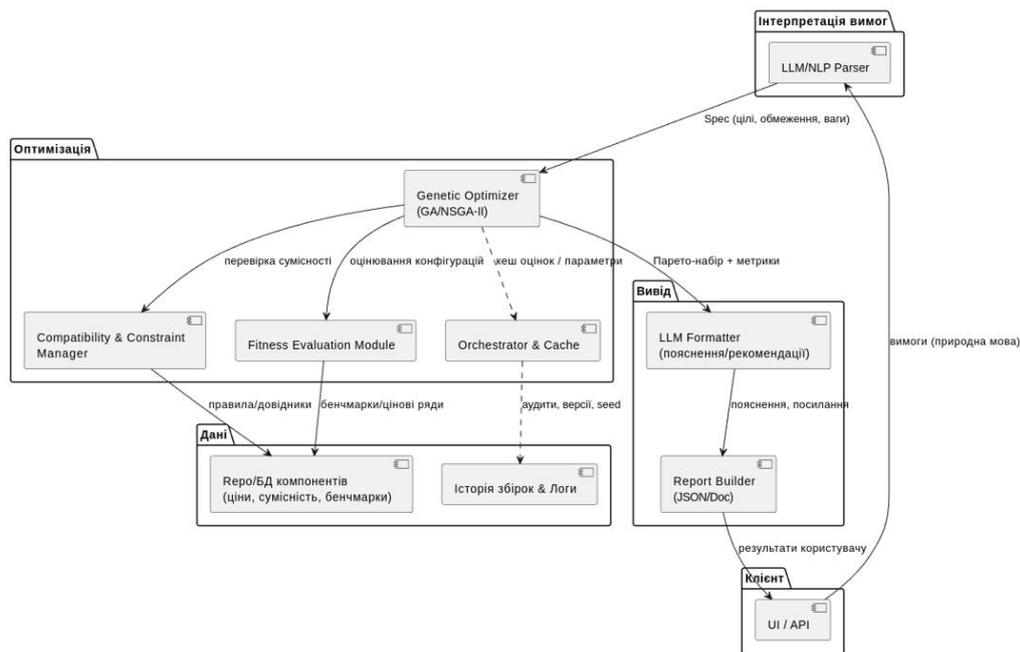
У моделі використовуються бінарні індикатори $I_k(x)$, які дорівнюють 1, якщо обмеження порушено, і 0, якщо все гаразд.

Перелік критичних обмежень (I_k):

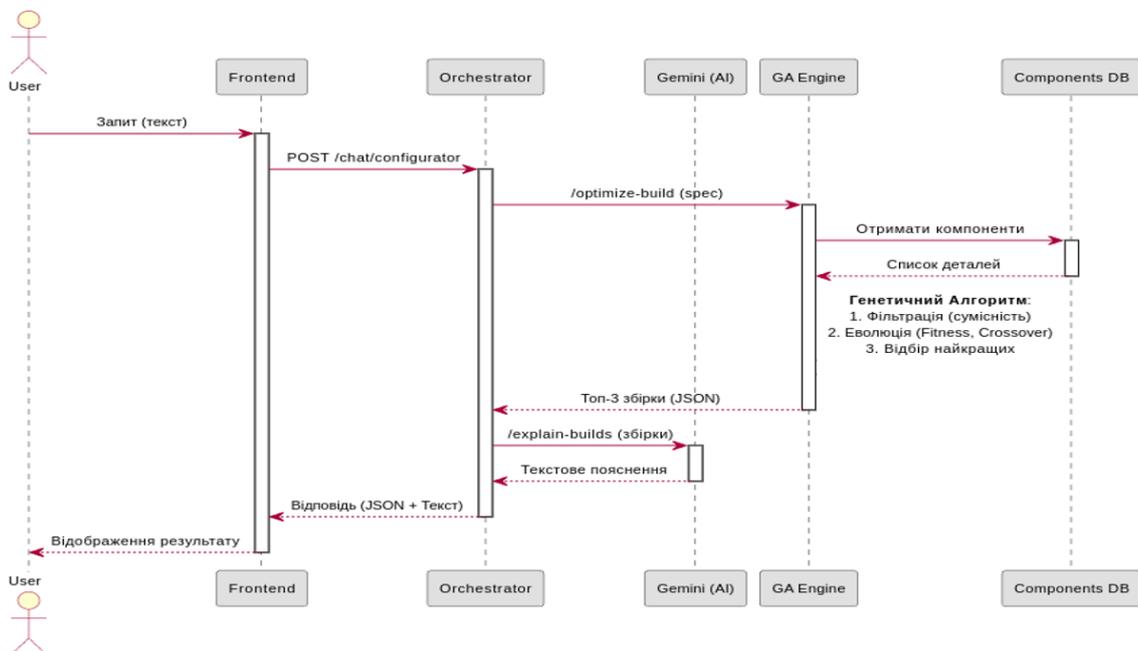
1. Апаратна несумісність (I_{socket}): $Socket_{cpu} \neq Socket_{mb}$;
2. Енергетична несумісність (I_{power}): $Wattage_{psu} < (TDP_{cpu} + TDP_{gpu} + \Delta)$;
3. Тип пам'яті несумісність (I_{ram}): $Type_{ram} \neq Type_{mb}$
4. Фізична несумісність (I_{phys}): $(Length_{gpu} > MaxLength_{case}) \vee (Height_{cooler} > MaxHeight_{case})$

6

СКЛАДОВІ МЕТОДИКИ ПІДБОРУ КОМПЛЕКТУЮЧИХ



ДІАГРАМА ПОСЛІДОВНОСТІ РОБОТИ МЕТОДИКИ ПІДБОРУ КОМПЛЕКТУЮЧИХ



ПРАКТИЧНИЙ РЕЗУЛЬТАТ

PC Configurator чат

Введіть свій запит, бюджет, обери режим (new / upgrade), і отримаєш рекомендації збірок.

Подіумнення для бота

Привіт. Я хочу зібрати собі ПК для ігор до 50000 грн з невеликим запасом.

Бюджет, грн:

Призначення (через кому):

Режим збірки: new (нова збірка) upgrade (апгрейд існуючої)

ipnet_existing_parts (JSON, тільки для upgrade, можна залишити порожнім)

Наприклад: {"cpu": "i", "mb": "4", "psu": "2"}

[Надіслати запит](#)

Зведена рекомендація

Я рекомендую збірку "Оптимальний вибір для ігор (build_1)". Її вартість складає 49800 грн, що ідеально вписується у ваш бюджет до 50000 грн. Ця конфігурація пропонує потужний процесор, достатню відеокарту для 1080p ігор та сучасну DDR5 пам'ять, що дає "невеликий запас" на майбутнє. Однак, зверніть увагу на критичну відсутність SSD накопичувача, який є обов'язковим для швидкої роботи сучасної системи, і його потрібно буде придати додатково.

Варіанти збірок

Оптимальний вибір для ігор (build_1)

Цей варіант ідеально збалансований за ціною та продуктивністю для вашого бюджету, пропонує міцну основу для ігор та майбутніх оновлень.

- Переваги:**
 - Потужний процесор Intel Core i5-12400F, достатній для сучасних ігор та більшості завдань.
 - 32 GB швидкої оперативної пам'яті DDR5-6000 з запасом на майбутнє.
 - Високоєфективний блок живлення на 850W із золотим сертифікатом, що забезпечує стабільність та широкі можливості для майбутніх апгрейдів відеокарти.
 - Ефективна система рідинного охолодження NZXT Kraken X63 280mm AIO для процесора.
 - Повністю вписується у ваш бюджет (49800 грн).
- Недоліки:**
 - Критична відсутність SSD накопичувача, що значно сповільнить завантаження системи, програм та ігор. Рекомендовано одразу додати хоча б 500GB NVMe SSD.
 - Відеокарта AMD Radeon RX 6600 8GB є гарним рішенням для 1080p, але для максимальних налаштувань у найновіших іграх може бути недостатньо потужною, якщо ви шукаєте значний "запас".
 - Висока система рідинного охолодження може бути незручною або великою опцією. Її краще міг би бути розроблений на SSD.

Збірки від ГА

Після запуску тут з'являться варіанти збірок і буде підсичено рекомендовано.

Оптимальний вибір для ігор (Intel Core i5 + RX 6600)

Повна вартість: 49800 грн

Збірка на базі Intel Core i5-12400F та AMD Radeon RX 6600 з 32 GB DDR5, пропонує добру продуктивність для ігор у бюджеті до 50000 грн, але потребує придати додаткового SSD.

cpu: Intel Core i5-12400F	8000 грн
mb: MSI PRO B760-P	6000 грн
gpu: AMD Radeon RX 6600 8GB	14000 грн
ram: 32GB (2x16GB) DDR5-6000	6000 грн
storage: 2TB HDD 7200rpm	1800 грн
psu: 850W 80+ Gold PSU	5000 грн
case: DeepCool Matrexx S5 Mesh	2500 грн
cooling: NZXT Kraken X63 280mm (AIO)	6500 грн

[View details](#) [Over budget total](#) [Score=20,000](#)

Висока продуктивність, але за межами бюджету (Intel Core i7 + RTX 4070)

Повна вартість: 72000 грн

Найпотужніший варіант з Intel Core i7-12700F та NVIDIA GeForce RTX 4070, але значно перевищує вказаний бюджет.

cpu: Intel Core i7-12700F	13000 грн
mb: Gigabyte Z790 UD	8000 грн
gpu: NVIDIA GeForce RTX 4070 12GB	30000 грн
ram: 16GB (2x8GB) DDR5-5200	3500 грн
storage: 2TB HDD 7200rpm	1800 грн
psu: 850W 80+ Gold PSU	5000 грн
case: Corsair 4000D AirFlow	4200 грн
cooling: NZXT Kraken X63 280mm (AIO)	6500 грн

[View details](#) [Over budget total](#) [Score=30,000](#)

Бюджетний варіант з SSD (AMD Ryzen 5 + RTX 3060)

Повна вартість: 38000 грн

Доступна збірка на базі AMD Ryzen 5 5500 та NVIDIA GeForce RTX 3060, яка має менш SSD, що забезпечує кращу стартову швидкість роботи системи.

cpu: AMD Ryzen 5 5500	4500 грн
mb: Gigabyte X570 AORUS Elite	6500 грн
gpu: NVIDIA GeForce RTX 3060 12GB	16000 грн
ram: 16GB (2x8GB) DDR4-3200	3000 грн
storage: 500GB SATA SSD	1200 грн
psu: 500W 80+ Bronze PSU	2000 грн
case: Corsair 4000D AirFlow	4200 грн
cooling: be quiet! Pure Rock 2 (air)	1600 грн

[View details](#) [Over budget total](#) [Score=20,000](#)

9

ПОРІВНЯЛЬНИЙ АНАЛІЗ ЕФЕКТИВНОСТІ РОЗРОБЛЕНОЇ МЕТОДИКИ

Метод	Швидкість підбору	Точність та сумісність збірки	Кількість запропонованих збірок	Принцип формування	Автоматизація підбору	Тип звітності
Ручний підбір	Дуже низька (хв/години/дні)	30-80% (залежить від знань)	1	Суб'єктивний ручний вибір	-	Відсутня (власні нотатки)
Конфігуратори магазинів (Telemart UA)	Середня (хвилини/години)	75-90%	1	Послідовний вибір з каталогу	-	Список товарів (Кошик)
LLM-чат (ChatGPT)	Висока (~20-25 сек.)	60-85%	1-3	Ймовірна генерація тексту	+	Неструктурований текст
Розроблена методика	Висока (~10-15 сек.)	96-100%	3	Гібридного алгоритму за допомогою ГА та LLM	+	Структурована (JSON + Пояснення)

10

ВИСНОВОК

1. Здійснено аналіз існуючих методів та засобів підбору комплектуючих. Виявлено їхні суттєві обмеження, зокрема: орієнтація лише на нові збірки, відсутність механізмів модернізації та недостатня гнучкість при роботі з обмеженим бюджетом.
2. Здійснена формалізація задачі підбору конфігурації як задачі багатокритеріальної оптимізації на основі гібридного алгоритму з використанням LLM.
3. Розроблено методику автоматизованого підбору компонентів, яка базується на поєднанні гібридного алгоритму (для точної математичної оптимізації) та великої мовної моделі (для взаємодії з користувачем).
4. Вибрані програмні засоби:
 - Python - основа бекенду та логіки ГА.
 - Gemini API - NLP-парсинг запитів та генерація звітів.
 - SQLite - Локальна база знань компонентів.,
 - Pydantic - Валідація даних та обмін JSON.
 - FastApi + Uvicorn - реалізація швидкого API мікросервісів.
5. Реалізовано методику у вигляді веб-додатку, що дозволяє отримувати рекомендації у форматі діалогу та візуалізувати запропоновані конфігурації.
6. Проведено експериментальні дослідження ефективності системи. Результати показали, що методика дозволяє: отримувати миттєвий результат (~10-15 сек.), технічну надійну збірку (96-100%), гнучкість вибору (Гібридний алгоритм за допомогою ГА та LLM) та якісну звітність (Структурована (JSON + Пояснення)).

11

ПУБЛІКАЦІ ТА АПРОБАЦІЯ РОБОТИ

Статті:

1. Кочетков О.В., Шевченко С.М. Багатокритеріальна оптимізація для вибору найкращої конфігурації комп'ютерних систем // Вісник Херсонського національного технічного університету» № 4, 2025

Тези доповідей:

1. Кочетков О.В., Шевченко С.М. Розробка алгоритму підбору комплектуючих для комп'ютерних систем з урахуванням бюджетних обмежень та наявної апаратної конфігурації // VI Всеукраїнська науково-технічна конференція «Сучасний стан та перспективи розвитку IoT», 15 квітня 2025 року. Збірник тез. – К.: ДУІКТ, 2025. – С. 158-160.
2. Кочетков О.В., Шевченко С.М. Визначення вимог до алгоритму підбору комплектуючих для комп'ютерних систем з урахуванням бюджетних обмежень та наявної апаратної конфігурації // VI Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 24 квітня 2025 року. Збірник тез. – К.: ДУІКТ, 2025. – С.558-560

12

ДОДАТОК Б. ЛІСТИНГ ОСНОВНИХ МОДУЛІВ

```
// фітнес функція
def fitness(
    individual: Dict[str, int],
    req: OptimizeRequest,
    id_map: Dict[int, Component],
    locked_price: int,
) -> float:
    """
    Чим більше значення – тим краща збірка.
    Для upgrade-режиму бюджет застосовується тільки до
    "нових" компонентів:
    new_cost = total_price_all - locked_price.
    """
    # Перевірка, що всі типи є
    if not all(t in individual for t in REQUIRED_TYPES):
        return -10.0
    try:
        cpu = id_map[individual["cpu"]]
        mb = id_map[individual["mb"]]
        gpu = id_map[individual["gpu"]]
        ram = id_map[individual["ram"]]
        storage = id_map[individual["storage"]]
        psu = id_map[individual["psu"]]
        case = id_map[individual["case"]]
        cooler = id_map[individual["cooler"]]
    except KeyError:
        return -10.0

    total_price = (
        cpu.price
        + mb.price
        + gpu.price
        + ram.price
        + storage.price
        + psu.price
        + case.price
        + cooler.price
    )
    # Вартість "оновлення" = усі компоненти мінус ті, що
    юзер уже має
    new_cost = max(total_price - locked_price, 0)

    # 1) Жорстке обмеження по бюджету (якщо сильно
    вище)
    if new_cost > req.budget_uah * 1.3:
        # Дуже сильний штраф – далеко за бюджетом
        return -20.0
    # 2) Перевірка сумісності
    # CPU ↔ MB socket
    if cpu.socket and mb.socket and cpu.socket != mb.socket:
        return -15.0

    # PSU wattage достатній?
    if psu.wattage is not None:
        needed_power = (cpu.tdp or 65) + (gpu.tdp or 150) + 100
        if psu.wattage < needed_power:
            return -15.0

    # Cooler достатньо потужний?
    if cooler.tdp is not None and cpu.tdp is not None:
        if cooler.tdp < cpu.tdp:
            return -12.0
    # MB ↔ RAM: тип пам'яті (DDR4/DDR5) має збігатися
    if mb.ram_type and ram.ram_type and mb.ram_type !=
    ram.ram_type:
        return -12.0

    # MB ↔ Case: форм-фактор
    if not is_mb_case_compatible(mb, case):
        return -12.0
    # GPU ↔ Case: довжина відеокарти
    if gpu.gpu_length_mm is not None and
    case.case_max_gpu_length_mm is not None:
        if gpu.gpu_length_mm > case.case_max_gpu_length_mm:
            return -12.0
    # Cooler ↔ Case: висота кулера (для air)
    if cooler.cooler_type == "air":
        if cooler.cooler_height_mm is not None and
        case.case_max_cooler_height_mm is not None:
            if cooler.cooler_height_mm >
            case.case_max_cooler_height_mm:
                return -12.0
    # Cooler тип 'aio' ↔ підтримка водянки в корпусі
    if cooler.cooler_type == "aio":
        if case.supports_aio == 0:
            return -12.0
    # 3) Продуктивність (дуже грубо: CPU+GPU, з
    маленьким бонусом за кращий кулер)
    cpu_perf = cpu.perf_score or 0.0
    gpu_perf = gpu.perf_score or 0.0
    cooler_bonus = (cooler.perf_score or 0.0) / 100.0 * 0.05 #
    максимум +0.05
    perf_norm = (cpu_perf + gpu_perf) / 200.0 + cooler_bonus
    # ~0..1+

    # 4) Ціна щодо бюджету (для new_cost)
    if new_cost <= req.budget_uah:
        price_score = 1.0 - (new_cost / req.budget_uah) # чим
        дешевше – тим вище
    else:
        # трохи вище бюджету – штраф, але не тотальний
        price_score = - (new_cost - req.budget_uah) /
        req.budget_uah
    # 5) Ваги
    w_perf = 0.7
    w_price = 0.3
    if "gaming" in req.purpose:
        w_perf = 0.8
        w_price = 0.2

    if req.preferences and req.preferences.price_level ==
    "budget":
        w_price += 0.2
        w_perf -= 0.2
        if w_perf < 0.3:
            w_perf = 0.3
    fitness_val = w_perf * perf_norm + w_price * price_score
    return fitness_val

// Селекція, кросовер, мутація
def next_generation(
    scored: List[Tuple[Dict[str, int], float]],
    components_by_type: Dict[str, List[Component]],
    req: OptimizeRequest,
    locked_types: Set[str],
    locked_components: Dict[str, Component],
) -> List[Dict[str, int]]:
    # Еліта
    scored_sorted = sorted(scored, key=lambda x: x[1],
    reverse=True)
```

```

elites = [ind for ind, score in
scored_sorted[:ELITE_COUNT]]

new_pop: List[Dict[str, int]] = [dict(e) for e in elites] # копії еліти

# Решту створюємо за допомогою селекції + кросовер + мутація
while len(new_pop) < POP_SIZE:
    parent1 = tournament_select(scored)
    parent2 = tournament_select(scored)
    child1, child2 = crossover(parent1, parent2)

    child1 = mutate(child1, components_by_type,
locked_types, mutation_rate=0.2)
    child2 = mutate(child2, components_by_type,
locked_types, mutation_rate=0.2)

    new_pop.append(child1)
    if len(new_pop) < POP_SIZE:
        new_pop.append(child2)

return new_pop

def tournament_select(scored: List[Tuple[Dict[str, int], float]],
k: int = 3) -> Dict[str, int]:
    k = min(k, len(scored))
    contenders = random.sample(scored, k)
    best_ind, best_fit = max(contenders, key=lambda x: x[1])
    return dict(best_ind) # копія

def crossover(p1: Dict[str, int], p2: Dict[str, int]) ->
Tuple[Dict[str, int], Dict[str, int]]:
    types = list(p1.keys())
    if len(types) < 2:
        return dict(p1), dict(p2)

    point = random.randint(1, len(types) - 1)
    child1: Dict[str, int] = {}
    child2: Dict[str, int] = {}
    for i, t in enumerate(types):
        if i < point:
            child1[t] = p1[t]
            child2[t] = p2[t]
        else:
            child1[t] = p2[t]
            child2[t] = p1[t]
    return child1, child2

def mutate(
    ind: Dict[str, int],
    components_by_type: Dict[str, List[Component]],
    locked_types: Set[str],
    mutation_rate: float = 0.1,
) -> Dict[str, int]:
    new_ind = dict(ind)
    for t in REQUIRED_TYPES:
        if t not in new_ind:
            continue
        if t in locked_types:
            continue # не змінюємо зафіксовані компоненти
        if random.random() < mutation_rate:
            options = components_by_type.get(t)
            if options:
                new_ind[t] = random.choice(options).id
    return new_ind

```

```

// пояснення для Gemini
def parse_intent_with_gemini(req: ParseIntentRequest) ->
ParseIntentResponse:
    """
    Розбирає текстовий запит користувача і намагається
    витягнути:
    - build_type: "new"/"upgrade"
    - budget_uah
    - purpose (gaming, work, streaming, ...)
    - price_level ("budget"/"balanced"/"premium")
    - focus_part ("gpu"/"cpu"/...)
    """

```

prompt = f"""
Ти — парсер запитів для конфігуратора ПК.

Отримуєш один рядок українською, російською або англійською, наприклад:
- "Привіт. Хочу зібрати новий ПК для ігор до 50000 грн."
- "Потрібен апгрейд поточного комп'ютера для роботи в 3D, бюджет 60к."
- "I want to upgrade my PC for streaming and gaming, around 1500\$."

Твоє завдання — спокійно та без вигадок витягнути з цього запиту такі поля:

1) build_type:
- "new", якщо користувач явно говорить про збірку з нуля:
("з нуля", "зібрати ПК", "нова збірка", "new pc", "build a pc").
- "upgrade", якщо явно йдеться про оновлення існуючого ПК:
("оновити ПК", "апгрейд", "upgrade", "замінити відеокарту" тощо).
- null, якщо однозначно зрозуміти не можна.

2) budget_uah:
- витягни числове значення бюджету в гривнях.
- приклади виразів: "до 50000 грн", "бюджет 30-35к", "на суму 64 200", "до 40 тисяч".
- Якщо явно вказані гривні — просто візьми число.
- Якщо сума в доларах/євро — МИ Ї НЕ конвертуємо, просто залишай budget_uah = null.
- Якщо бюджету взагалі немає в тексті — залиш budget_uah = null.

3) purpose:
- Список коротких тегів-призначень:
"gaming", "work", "streaming", "study", "editing", "3d", "office" і т.п.
- Якщо користувач пише "для ігор та роботи" — це ["gaming", "work"].
- Якщо чітко не сказано — можна залишити пустий список.

4) price_level:
- "budget", якщо користувач підкреслює дешевизну:
"бюджетний", "дешевше", "мінімум витрат", "якнайдешевше".
- "premium", якщо просить "максимум продуктивності", "без обмежень по бюджету".
- "balanced" — якщо явно не написано, але запит звучить як "нормальний ПК за свій бюджет".
- Якщо не впевнений — залиш null.

5) focus_part:
- Якщо із тексту видно, що користувачу головне "відеокарта" — "gpu".

- Якщо акцент на "процесор" — "cpu".
- Якщо на "багато пам'яті"/"оперативки" — "ram".
- Якщо на тихій роботі — "silent".
- Якщо на маленькому корпусі — "compact".
- Інакше залиш null.

Будь дуже обережним:

- НЕ вигадуй бюджет, якщо його немає в тексті.
- Якщо запит виглядає неоднозначним, краще залиш поле null.
- Відповідь **ПОВИННА** бути у форматі JSON, який відповідає схемі ParseIntentResponse.

Оригінальний запит користувача:

```
{req.user_message!r}
"""
```

```
response = client.models.generate_content(
    model=MODEL_NAME,
    contents=prompt,
    config={
        "response_mime_type": "application/json",
        "response_schema": ParseIntentResponse,
    },
)
```

```
return response.parsed # type: ignore
```

```
def explain_builds_with_gemini(req: ExplainRequest) ->
```

```
ExplainResponse:
```

```
"""
    Викликає Gemini, щоб:
    - вибрати рекомендовану збірку,
    - зробити короткі резюме для кожної,
    - згенерувати фінальне пояснення українською
    (Markdown).
    """
```

```
    builds_text = _format_builds_for_prompt(req)
```

```
    prompt = f"""
```

```
    Ти — асистент, який допомагає користувачу обрати
    конфігурацію ПК.
```

```
    Тобі передано:
```

```
    - оригінальний запит користувача (user_message):
    {req.user_message!r}
    - режим збірки (build_type): "{req.build_type}"
    - бюджет (budget_uah): {req.budget_uah} грн
    - цілі використання (purpose): {"", ".join(req.purpose) if
    req.purpose else "не вказано"}
    - список можливих збірок (builds) від генетичного
    алгоритму з їхньою ціною, upgrade_cost та компонентами.
    - Для деяких компонентів тобі також передано url
    (посилання на магазин).
    Вони виглядають у вхідних даних як: "[url: https://...]"
    поруч із компонентом.
```

```
    Як поводитись із url:
```

```
    - НЕ вигадуй жодних посилань самостійно.
    - Використовуй ТІЛЬКИ ті url, які явно вказані у вхідних
    даних (поруч з компонентами).
    - Якщо url є, ти можеш вставити його у
    final_message_markdown у форматі Markdown-посилання:
    наприклад: `[NVIDIA GeForce RTX 3060
    12GB](https://...)`.
    - Не обов'язково вставляти посилання для КОЖНОГО
    компонента, достатньо для 1–3 ключових (CPU, GPU,
    можливо RAM/SSD),
```

щоб не перевантажувати текст.

Ось усі збірки у зручному вигляді:

```
{builds_text}
```

Твоє завдання:

- 1) Обрати одну рекомендовану збірку (id з поля "id" — наприклад, "build_1").
- 2) Для **КОЖНОЇ** збірки сформувати:
 - коротку назву (title),
 - короткий опис (short_summary),
 - список 3–6 плюсів (pros),
 - список 2–5 мінусів (cons),
 - позначити, яка збірка є рекомендованою (is_recommended = true/false).
- 3) Сформувати фінальне повідомлення українською мовою у форматі Markdown, яке пояснює користувачу різницю між варіантами та причину вибору рекомендованої збірки.

Важливі умови:

- НЕ вигадуй власні ціни або компоненти — використовуй **ТІЛЬКИ** ті, що присутні в списку builds.
- Звертай особливу увагу на:
 - upgrade_cost: реальна сума, яку потрібно витратити на нові