Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

МЕТОД ВИЯВЛЕННЯ ДУБЛІКАТІВ WORKFLOW-ГЕНЕРАЦІЇ ЗОБРАЖЕНЬ ДЛЯ LORA-МОДЕЛЕЙ НА ОСНОВІ СТРУКТУРНОГО ТА ПОВЕДІНКОВОГО АНАЛІЗУ
(тема)

Виконав:
здобувач 4 року навчання,
групи ІТІНФ-22-3
Лі Нікіта Джіанпінгович
(прізвище, ініціали)

Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Керівник ст. викл. Кобилін І. О.
(посада, прізвище, ініціали)

Допускається до захисту

Завідувач кафедри інформатики _____ Кобилін О. А.
(підпис) (прізвище, ініціали)

2026 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту

Кафедра Інформатики

Рівень вищої освіти перший (бакалаврський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Здобувачеві Лі Нікіті Джіанпінговичу
(прізвище, ім'я, по батькові)

1. Тема роботи Метод виявлення дублікатів workflow-генерації зображень для LoRA-моделей на основі структурного та поведінкового аналізу.

затверджена наказом університету від 26 травня 2026 року № 435Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 22 червня 2026 р.

3. Вихідні дані до роботи Науково-методична та науково-технічна література, матеріали міжнародних науково-практичних конференцій та періодичних видань за тематикою генеративного штучного інтелекту; технічні специфікації архітектур дифузійних моделей та методів низькорангової адаптації (LoRA); документація до форматів опису робочих процесів (JSON API Prompt та Canvas Format) у середовищах вузлової генерації зображень (ComfyUI).

4. Перелік питань, що потрібно опрацювати в роботі

1. Огляд методів детектування простих елементів зображення та аналіз їх можливостей.
2. Математична модель перетворення Хафа для знаходження геометричних примітивів.
3. Дослідження особливостей workflow генерації зображень для LoRA-моделей та їх представлення у вигляді графових структур.
4. Розроблення методу канонікалізації орієнтованих графів для усунення технічних розбіжностей у JSON-описах.
5. Реалізація алгоритму швидкого виявлення точних дублікатів на основі криптографічного хешування SHA-256.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) Актуальність проблеми обробки зображень, постановка задачі, загальна схема методу виявлення дублікатів, приклади канонікалізації графового представлення, діаграма прецедентів та компонентів вебсервісу, UML-діаграми послідовності та станів, результати тестування (графіки/таблиці), екранні форми інтерфейсу користувача.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	13.04.2026	
2	Аналіз завдання, підбір літератури	14.04.26-16.04.26	
3	Аналіз літератури з проблеми, що вирішується	17.04.26-19.04.26	
4	Аналіз існуючих методів розпізнавання	20.04.26-22.04.26	
5	Формування кроків вирішення проблеми	23.04.26-05.05.26	
6	Програмна реалізація	26.04.26-20.05.26	
7	Аналіз отриманих результатів	21.05.26-04.06.26	
8	Оформлення пояснювальної записки	05.06.26-09.06.26	
9	Перевірка на нормоконтроль	15.06.26-19.06.26	
10	Перевірка на плагіат	17.06.26-30.06.26	
11	Рецензування	22.06.26-30.06.26	
12	Підготовка презентації та доповіді	22.06.26-30.06.26	
13	Занесення роботи в електронний архів	01.07.26	
14	Попередній захист кваліфікаційної роботи	01.07.26	

Дата видачі завдання 13 квітня 2026 р.

Здобувач _____
(підпис)

Керівник роботи _____ ст. викл. Кобилін І. О.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ/ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 90 с., 9 табл., 21 рис., 46 джерел.

ВЕБСЕРВІС, ВИЯВЛЕННЯ ДУБЛІКАТІВ, ГЕНЕРАЦІЯ ЗОБРАЖЕНЬ, ГРАФОВА СТРУКТУРА, КАНОНІКАЛІЗАЦІЯ, ПОВЕДІНКОВИЙ АНАЛІЗ, СТРУКТУРНИЙ АНАЛІЗ, LORA-МОДЕЛЬ, WORKFLOW.

Об’єктом роботи є процес виявлення дублікатів конфігурацій workflow для генерації зображень з LoRA-моделями.

Метою роботи є розроблення методу виявлення дублікатів цих workflow на основі структурного та поведінкового аналізу.

Під час роботи використано методи побудови й канонікалізації графа залежностей, аналізу параметрів вузлів, хешування та оцінювання подібності.

Практична цінність полягає в автоматизації пошуку точних і подібних конфігурацій для впорядкування наборів workflow та зменшення дублювання даних.

У результаті роботи розроблено метод і вебсервіс для завантаження, збереження, канонікалізації та порівняння інструментів генерації.

Область застосування: системи генерації зображень, платформи роботи з LoRA-моделями та інструменти керування workflo

DUPLICATE DETECTION, GRAPH STRUCTURE, IMAGE GENERATION, LORA MODEL, STRUCTURAL ANALYSIS, WEB SERVICE, WORKFLOW.

The object of the work is the process of detecting duplicate image generation workflows for LoRA models.

The goal of the work is to develop a method for detecting duplicate image generation workflows for LoRA models based on structural and behavioral analysis.

The work uses methods of workflow structural analysis, dependency graph construction, graph canonicalization, hashing, similarity estimation, and analysis of node parameters.

The practical value lies in automating the detection of exact and similar workflows, which makes it possible to organize workflow sets and reduce duplication of image generation configurations.

As a result, a method and a web service for uploading, storing, canonicalizing, and comparing image generation workflows were developed.

Applications: image generation systems, platforms for working with LoRA models, and workflow management tools.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

БД – база даних

Вебсервіс – програмний сервіс, доступний через вебінтерфейс або API

Граф – структура даних, що складається з вузлів і зв’язків між ними

Дублікат workflow – workflow, що має однакову або близьку структуру та параметри порівняно з іншим workflow

Канонікалізація – приведення структури workflow до стабільного нормалізованого представлення

Структурний аналіз – аналіз складу вузлів, зв’язків і параметрів workflow

Хешування – перетворення даних у фіксований рядок символів для подальшого порівняння

ШІ – штучний інтелект

API – Application Programming Interface (інтерфейс програмування застосунку)

ComfyUI – графовий інтерфейс для створення workflow генерації зображень

CFG – Classifier-Free Guidance (метод керування процесом дифузійної генерації зображень без використання окремого класифікатора, що визначає ступінь відповідності результату текстовому запиту.)

FastAPI – Python-фреймворк для створення вебсервісів та REST API

HTTP – HyperText Transfer Protocol (протокол передавання вебзапитів і відповідей)

JSON – JavaScript Object Notation (формат обміну структурованими даними)

LoRA – Low-Rank Adaptation (метод адаптації нейронних моделей за допомогою низькорангових матриць)

REST – Representational State Transfer (архітектурний стиль взаємодії клієнта і сервера)

SHA-256 – Secure Hash Algorithm 256-bit (криптографічна хеш-функція для формування 256-бітного хешу)

SQL – Structured Query Language (мова структурованих запитів)

SQLite – вбудована реляційна система керування базами даних

UI – User Interface (інтерфейс користувача)

Workflow – послідовність взаємопов’язаних вузлів і параметрів, що описує процес генерації зображення

ВСТУП

Генерація зображень за допомогою моделей штучного інтелекту стала одним із важливих напрямів сучасних інформаційних технологій [29]. Поширення дифузійних моделей, візуальних середовищ налаштування генерації та додаткових адаптаційних моделей дало змогу створювати складні конфігурації обробки даних без написання великої кількості програмного коду. Одним із таких підходів є використання робочих послідовностей, які описують послідовність вузлів, параметрів і зв’язків між етапами генерації зображення.

Особливе місце у таких системах займають LoRA-моделі, що дають змогу адаптувати базові генеративні моделі до певного стилю, об’єкта або предметної області без повного перенавчання всієї моделі [20]. На практиці LoRA-моделі часто використовуються разом із графовими інтерфейсами, де процес генерації подається у вигляді набору взаємопов’язаних вузлів. Кожен процес workflow може містити параметри моделі, текстові запити, налаштування алгоритма вибірки, розміри зображення, значення seed та інші характеристики, які впливають на результат генерації.

Зі збільшенням кількості робочих послідовностей виникає проблема їх повторюваності [3]. Один і той самий або дуже схожий процес генерації може зберігатися під різними назвами, мати інший порядок вузлів, змінені ідентифікатори або незначні відмінності у параметрах. Через це ручне порівняння таких конфігурацій стає складним і ненадійним.

Актуальність теми полягає в необхідності автоматизованого виявлення дублікатів робочих послідовностей генерації зображень для LoRA-моделей [4]. Для розв’язання цієї задачі доцільно використовувати структурний аналіз, який враховує склад вузлів і зв’язків між ними, а також аналіз параметрів, що характеризують поведінку робочих послідовностей під час генерації. Такий підхід дає змогу порівнювати не лише вихідний JSON-опис, а й нормалізоване представлення процесу генерації [5, 6].

1 АНАЛІЗ МЕТОДІВ ТА ЗАСОБІВ ВИЯВЛЕННЯ ДУБЛІКАТІВ У ПОСЛІДОВНОСТІ ОПЕРАЦІЙ ГЕНЕРАЦІЇ ЗОБРАЖЕНЬ

1.1 Актуальність виявлення дублікатів робочих послідовностей генерації зображень

Генерація зображень за допомогою моделей штучного інтелекту стала одним із найбільш поширених напрямів практичного використання генеративних моделей [7]. Сучасні системи дають змогу створювати зображення за текстовими описами, змінювати стиль, адаптувати результат до конкретного об’єкта або предметної області та повторювати процес генерації з різними параметрами. Для цього користувачі застосовують не лише базові моделі, а й додаткові адаптаційні модулі, серед яких важливе місце займають LoRA-моделі.

У практичних системах генерації зображень процес створення результату часто подається у вигляді послідовності [8]. Така послідовність складається з окремих вузлів, що відповідають за завантаження моделі, введення текстового запиту, налаштування параметрів генерації, застосування LoRA-моделі, декодування результату та збереження зображення. Зв’язки між вузлами визначають порядок передавання даних і логіку виконання процесу.

Зі збільшенням кількості послідовностей виникає проблема дублювання [9]. Один і той самий процес генерації може бути збережений під різними назвами, мати інші ідентифікатори вузлів або відрізнятися порядком розміщення елементів у JSON-файлі. Крім того, послідовності можуть бути не повністю однаковими, але дуже схожими за структурою та параметрами. Наприклад, користувач може змінити лише seed, кількість кроків генерації або силу застосування LoRA-моделі, залишивши основну структуру без змін.

Ручне порівняння таких послідовностей є складним і ненадійним [10]. Користувачеві потрібно аналізувати велику кількість вузлів, параметрів і зв’язків між ними. При цьому просте порівняння тексту JSON-файлів не є

достатнім, оскільки однакові послідовності можуть мати інший порядок запису вузлів або відмінні службові ідентифікатори. Тому актуальною є задача автоматизованого виявлення точних і схожих дублікатів послідовностей генерації зображень.

Для розв’язання цієї задачі доцільно використовувати структурний аналіз послідовностей, подання конфігурації у вигляді графа залежностей, канонікалізацію графового представлення та оцінювання подібності за вузлами, зв’язками і параметрами [11]. Такий підхід дає змогу аналізувати не лише зовнішній вигляд JSON-опису, а й логічну структуру процесу генерації.

1.2 Особливості послідовностей генерації зображень для LoRA-моделей

Послідовність генерації зображень можна розглядати як формальний опис послідовності операцій, необхідних для отримання кінцевого зображення [12]. На відміну від лінійного сценарію, послідовності часто мають графову структуру, у якій окремі етапи можуть бути пов’язані між собою через вхідні та вихідні дані. Кожен вузол виконує певну функцію, а зв’язки між вузлами визначають напрямок передавання результатів обробки.

У системах на основі дифузійних моделей типова робоча послідовність може містити вузли завантаження контрольної точки моделі, введення позитивного та негативного текстового запиту, налаштування алгоритму вибірки, генерації латентного представлення, декодування зображення та збереження результату [13]. Якщо використовується LoRA-модель, до такої послідовності додаються вузли або параметри, що відповідають за підключення адаптаційної моделі та визначення сили її впливу [14].

LoRA-моделі використовуються для адаптації базової генеративної моделі без повного перенавчання. Вони дають змогу додати до генерації певний стиль, персонажа, предмет або іншу спеціалізовану ознаку. У

послідовності це зазвичай відображається через назву LoRA-моделі, параметри її застосування та зв’язки з основною моделлю генерації. Саме тому параметри, пов’язані з LoRA, мають велике значення під час порівняння послідовностей.

Особливістю процесу workflow є те, що його зміст визначається не лише набором вузлів, а й способом їх з’єднання. Два процеси workflow можуть містити однакові типи вузлів, але мати різну логіку передавання даних [15]. У такому випадку вони не повинні вважатися повними дублікатами. З іншого боку, два процеси workflow можуть мати різні службові ідентифікатори вузлів, але однакову структуру та параметри. У такому випадку вони мають розглядатися як точні дублікати.

Таким чином, для аналізу робочих послідоностей необхідно враховувати такі характеристики: типи вузлів, зв’язки між вузлами, назви входів, виходів, параметри генерації, параметри LoRA-моделі та значення, що впливають на кінцевий результат. Ігнорування будь-якої з цих складових може призвести до помилкового визначення дублікатів або пропуску подібних послідовностей [16].

1.2.1 Структура послідовностей у середовищах вузлової генерації зображень

У середовищах вузлової генерації зображень, зокрема в системах на зразок ComfyUI, процес отримання результату формується не як послідовність команд, а як схема взаємопов’язаних функціональних блоків [17]. Кожен блок виконує окрему операцію: завантажує базову модель, застосовує LoRA-модель, обробляє текстовий запит, виконує вибірку, декодує латентне представлення або зберігає сформоване зображення. Такий спосіб подання дає користувачеві змогу змінювати окремі етапи генерації без перебудови всього процесу. Типовий процес workflow генерації зображення з використанням

LoRA-моделі містить вузол завантаження базової моделі, вузол підключення LoRA, вузли для позитивного і негативного текстових запитів, вузол налаштування початкового латентного зображення, обирач вибірки, декодер та вузол збереження результату [18]. При цьому один вузол може передавати дані відразу декільком наступним вузлам. Наприклад, модифікована за допомогою LoRA модель може одночасно використовуватися для оброблення текстових умов та безпосередньо для процесу генерації.

Для задачі виявлення дублікатів важливо враховувати, що зовнішній вигляд схеми не завжди визначає її зміст [19]. Користувач може перемістити вузли на робочому полі, змінити їх службові ідентифікатори або зберегти послідовність після незначного редагування розташування елементів. Такі зміни не впливають на логіку генерації зображення. Отже, координати вузлів, порядок їх запису у файлі та службові позначення не повинні бути визначальними ознаками під час порівняння.

Натомість суттєве значення мають типи використаних вузлів, напрямки передавання даних і параметри, що змінюють результат генерації [20]. Наприклад, заміна LoRA-моделі, базової моделі або способу підключення семплера може призвести до іншого результату навіть за однакового текстового запиту. Аналогічно, два процеси workflow з однаковими типами вузлів, але різними зв’язками між ними не можна розглядати як повні дублікати, оскільки вони реалізують різні послідовності оброблення даних (табл. 1.1) [21].

Таблиця 1.1 – Основні компоненти процесу workflow генерації зображень

Компонент workflow	**Призначення**	**Значення під час порівняння**
1	**2**	**3**
Вузол завантаження базової моделі	Вибір основної генеративної моделі	Визначає основу процесу генерації

Продовження таблиці 1.1

1	2	3
Вузол LoRA	Підключення адаптаційної моделі та сили її впливу	Є важливою поведінковою ознакою
Текстові запити	Визначення бажаних і небажаних характеристик результату	Впливають на зміст сформованого зображення
Обирач вибірки	Виконання основного процесу генерації	Містить значущі параметри генерації
Латентне представлення	Задання початкових розмірів і структури результату	Впливає на параметри вихідного зображення
Декодер	Перетворення латентного представлення на зображення	Відображає завершальний етап оброблення
Вузол збереження	Збереження сформованого результату	Відображає завершальний етап оброблення

1.3 Подання послідовності у вигляді графових структур

Оскільки процес workflow складається з вузлів і зв’язків між ними, природним способом його формалізації є подання у вигляді графа [22]. У такому графі вершини відповідають вузлам процесу workflow, а ребра описують залежності між ними. Якщо результат одного вузла передається на вхід іншого, між відповідними вершинами створюється напрямлене ребро.

Графове подання дає змогу абстрагуватися від початкового порядку запису вузлів у JSON-файлі [23]. Це важливо, оскільки порядок елементів у файлі не завжди відображає реальну логіку виконання послідовності. Для визначення подібності більш важливими є типи вузлів, зв’язки між ними та параметри, ніж порядок їх збереження у файлі.

Під час побудови графа кожен вузол послідовності можна описати набором ознак [24]. До таких ознак належать ідентифікатор вузла, тип вузла, literal-параметри та посилання на інші вузли. Literal-параметрами є значення, які задаються безпосередньо у вузлі: текстові запити, назви моделей, кількість кроків, розміри зображення, seed та інші налаштування. Посилання на інші вузли визначають залежності між етапами генерації.

Для виявлення точних дублікатів необхідно отримати стабільне представлення графа, яке не залежить від випадкових службових ідентифікаторів [25]. Таке представлення називають канонічним. Канонікалізація дає змогу привести однакові за змістом послідовності до однакового внутрішнього опису. Після цього можна застосувати хешування та порівнювати отримані хеші.

Графове подання також корисне для пошуку не лише точних, а й подібних послідовностей [26]. У цьому випадку можна порівнювати кількість і типи вузлів, набір ребер, спільні параметри та відмінності між ними. Наприклад, якщо дві послідовності мають однакову структуру, але відрізняються значеннями prompt або параметрами LoRA, вони можуть вважатися схожими, але не точними дублікатами.

1.4 Аналіз методів виявлення дублікатів і оцінювання подібності

Задача виявлення дублікатів процесу workflow може розглядатися на двох рівнях: пошук точних дублікатів і пошук подібних процесів workflow. Точні дублікати мають однакову логічну структуру та однакові значущі

параметри. подібні процеси workflow можуть мати спільну структуру або близький набір вузлів, але відрізнятися окремими параметрами.

Найпростішим підходом є пряме порівняння JSON-файлів. Однак цей підхід має суттєві недоліки. Він залежить від порядку запису елементів, форматування, службових ідентифікаторів та інших деталей, які не завжди впливають на логіку послідовності. Через це дві однакові за змістом послідовності можуть бути визначені як різні.

Більш ефективним підходом є попередня нормалізація даних. Вона передбачає приведення структури послідовностей до єдиного вигляду, сортування параметрів, відокремлення literal-значень від посилань на інші вузли та побудову графа залежностей. Після цього можна виконувати більш надійне порівняння.

Для пошуку точних дублікатів доцільно використовувати хешування канонічного представлення. Якщо два процеси workflow після канонікалізації мають однаковий хеш, вони можуть вважатися точними дублікатами. Перевагою такого підходу є швидкість порівняння, оскільки замість повного аналізу структури достатньо порівняти два хеш-значення.

Для пошуку подібних процесів workflow одного хешування недостатньо. Навіть незначна зміна параметра призведе до іншого хешу, хоча процес workflow може залишатися дуже близьким за змістом. Тому необхідно додатково оцінювати подібність за окремими групами ознак: типами вузлів, зв’язками між вузлами та literal-параметрами.

1.4.1 Виявлення точних дублікатів за допомогою хешування

Хешування є одним із поширених способів швидкого порівняння даних. Його суть полягає у перетворенні вхідного об’єкта на рядок фіксованої довжини. Якщо два об’єкти мають однакове нормалізоване представлення, їхні хеші також будуть однаковими.

У задачі виявлення дублікатів робочих послідовностей, хешування доцільно застосовувати не до початкового JSON-файлу, а до канонічного представлення графа. Це дозволяє уникнути залежності від порядку вузлів, форматування та службових ідентифікаторів. Спочатку послідовність перетворюється на граф, далі граф приводиться до стабільного представлення, після чого для цього представлення обчислюється хеш.

Перевагою такого методу є висока швидкість перевірки точних дублікатів. Після збереження хешу в базі даних нову послідовність можна швидко порівняти з уже наявними записами. Якщо хеш збігається, система може повідомити користувача про наявність точного дубліката.

Недоліком хешування є те, що воно не дає інформації про ступінь подібності. Якщо у процеса workflow змінити лише один параметр, хеш стане іншим. Тому хешування доцільно використовувати як перший етап перевірки, а для аналізу схожості застосовувати додаткові методи порівняння ознак.

1.4.2 Оцінювання структурної подібності графів

Структурна подібність процесу workflow визначається тим, наскільки збігаються їхні вузли та зв’язки. Для цього можна порівнювати типи вузлів, кількість вузлів кожного типу, вхідні та вихідні зв’язки, а також загальну структуру графа. Якщо два процеси workflow мають однаковий набір вузлів і однакові залежності між ними, вони є структурно близькими.

Для практичної реалізації зручно подавати ознаки процесу workflow у вигляді множин або лічильників. Наприклад, можна підрахувати кількість вузлів кожного типу, сформувати набір сигнатур ребер і порівняти ці набори між двома послідовностями. Подібність може оцінюватися як відношення спільних ознак до загальної кількості ознак.

Такий підхід дозволяє виявляти послідовності, які мають спільну основу, але відрізняються окремими деталями. Наприклад, якщо дві

послідовності використовують однакову схему генерації, але мають різні текстові запити або різні значення seed, структурна подібність між ними залишиться високою.

Водночас структурна подібність не завжди достатня для правильного висновку. Дві послідовності можуть мати однакову структуру, але використовувати різні моделі, різні LoRA-файли або принципово різні запити. Тому оцінювання структури потрібно поєднувати з аналізом параметрів.

1.4.3 Аналіз параметрів вузлів послідовностей

Параметри вузлів процесу workflow безпосередньо впливають на результат генерації зображення. До них належать текстові запити, назви моделей, параметри LoRA, кількість кроків генерації, значення CFG, розміри зображення, seed, алгоритм вибірки та інші налаштування. Під час порівняння процесів workflow ці параметри мають різну вагу.

Наприклад, зміна назви LoRA-моделі або основної моделі генерації може суттєво змінити результат. Такі параметри доцільно вважати більш значущими. Зміна seed також впливає на результат, але не завжди змінює логіку процесу workflow, тому цей параметр може мати меншу вагу під час оцінювання подібності.

Аналіз параметрів дає змогу відрізнити точний дублікат від схожого варіанта. Якщо структура процесу workflow однакова, але змінено запит, силу LoRA або кількість кроків, процес workflow не є точним дублікатом, але може бути корисним для пошуку як схожий. Такий підхід є більш гнучким, ніж просте порівняння файлів або хешів.

У межах роботи доцільно використовувати зважену оцінку подібності, яка враховує структурні ознаки та literal-параметри. Це дозволяє сформувати загальний показник подібності та класифікувати результат як точний дублікат,

високий рівень схожості, помірний рівень схожості або відмінну послідовність.

1.5 Інструментарій для реалізації вебсервісу аналізу послідовностей

Проведений аналіз особливостей послідовностей генерації зображень і наявних підходів до їх порівняння показав, що задача виявлення дублікатів не може бути коректно розв’язана лише шляхом прямого зіставлення початкових файлів. Послідовність є структурованим об’єктом, зміст якого визначається не порядком запису даних, а сукупністю вузлів, зв’язків і параметрів, що керують процесом генерації зображення.

Для практичного використання система виявлення дублікатів повинна підтримувати два різні режими аналізу. Перший режим має визначати точні дублікати, тобто послідовності, які мають однакову суттєву структуру та однакові значущі параметри. Другий режим повинен виявляти подібні послідовності, які можуть відрізнятися окремими параметрами, але використовують спільну схему генерації або близький набір функціональних вузлів.

До методу виявлення дублікатів процесу workflow доцільно висунути такі вимоги:

– незалежність результату порівняння від службових ідентифікаторів вузлів та порядку їх запису у файлі;

– урахування типів вузлів і напрямків зв’язків між ними;

– урахування параметрів базової моделі, LoRA-моделі та основних параметрів генерації;

– можливість відрізняти точний дублікат від схожого, але модифікованої послідовності;

– достатня швидкість перевірки під час роботи з колекцією збережених послідовностей;

– можливість пояснити користувачеві причини встановленого рівня подібності.

З урахуванням зазначених вимог у роботі обирається комбінований підхід до виявлення дублікатів. Його основою є подання послідовності у вигляді графа, у якому вузли описують окремі операції генерації, а ребра відображають передавання даних між ними. Таке представлення дозволяє аналізувати логіку процесу незалежно від зовнішнього оформлення або порядку елементів у початковому JSON-файлі.

Для визначення точних дублікатів доцільно формувати канонічне представлення процесу workflow та обчислювати для нього хеш-значення. Якщо два процеси workflow мають однакове канонічне представлення, вони отримують однаковий хеш і можуть бути визначені як точні дублікати. Такий механізм є придатним для швидкого пошуку повторюваних конфігурацій у базі даних.

Якщо хеші процесів workflow не збігаються, але необхідно встановити ступінь їх близькості, виконується додатковий аналіз подібності. Для цього враховуються три групи ознак: типи вузлів, зв’язки між ними та literal-параметри. Розділення ознак на групи є важливим, оскільки структура процесу workflow і значення його параметрів характеризують різні аспекти подібності. Однакова структура може свідчити про спільну логіку генерації, тоді як відмінні параметри показують, наскільки змінено поведінку конкретної конфігурації.

Таким чином, обраний підхід відповідає особливостям предметної області та забезпечує основу для подальшого математичного опису методу.

1.6 Інструментарій для реалізації вебсервісу аналізу послідовностей

Для практичної реалізації методу виявлення дублікатів послідовностей необхідно створити програмний засіб, який забезпечує завантаження

послідовностей, їх аналіз, порівняння та збереження результатів. З огляду на це доцільним є створення вебсервісу з REST API та простим інтерфейсом користувача.

Мова програмування Python є зручною для реалізації такої системи, оскільки має розвинену екосистему бібліотек для роботи з JSON, вебсервісами, базами даних і тестуванням. Для створення серверної частини може бути використано FastAPI, який дозволяє швидко розробляти REST API, описувати схеми запитів і відповідей та забезпечувати інтеграцію з вебінтерфейсом.

Для збереження робочої послідовності і результатів аналізу доцільно використовувати базу даних. У локальному режимі зручно застосовувати SQLite, оскільки вона не потребує окремого серверного процесу та підходить для невеликих і середніх обсягів даних. Для роботи з базою даних може бути використано SQLAlchemy, який забезпечує об’єктно-реляційне відображення та спрощує роботу з моделями даних.

Вебінтерфейс потрібен для ручної роботи з процесами workflow: завантаження JSON-файлів, порівняння двох процесів workflow, перевірки нового процесу workflow щодо вже збережених записів і перегляду результатів. Наявність такого інтерфейсу робить систему зручною для практичного використання без необхідності виконувати запити вручну.

Таким чином, поєднання Python, FastAPI, SQLAlchemy, SQLite, REST API та вебінтерфейсу дозволяє реалізувати систему, яка не лише виконує алгоритмічне порівняння послідовностей, а й надає користувачеві інструмент для роботи з наборами конфігурацій генерації зображень.

1.7 Постановка задачі

Автоматизоване виявлення точних і подібних процесів workflow для генерації зображень з LoRA-моделями є актуальним завданням через стрімке

зростання кількості таких конфігурацій. Великі обсяги даних ускладнюють ручний аналіз і призводять до дублювання процесів. Тому виникає потреба в розробленні методу та програмного засобу для автоматизованого аналізу й ефективного пошуку дублікатів і схожих конфігурацій.Об’єктом роботи є процес виявлення дублікатів процесів workflow генерації зображень для LoRA-моделей.

Метою роботи є розроблення методу виявлення дублікатів процесів workflow для LoRA-моделей на основі структурного та поведінкового аналізу.

Для досягнення мети необхідно вирішити такі завдання:

- проаналізувати предметну область генерації зображень за допомогою процесів workflow;
- розглянути особливості використання LoRA-моделей у процесах workflow генерації зображень;
- дослідити підходи до подання процесів workflow у вигляді графових структур;
- проаналізувати методи виявлення точних дублікатів і подібних процесів workflow;
- розробити спосіб канонікалізації графового представлення процесів workflow;
- реалізувати хешування канонічного представлення для пошуку точних дублікатів;
- розробити метод оцінювання подібності процесів workflow за вузлами, зв’язками та параметрами;
- створити вебсервіс для завантаження, збереження та порівняння процесів workflow;
- інтегрувати розроблений спосіб канонікалізації та метод оцінювання подібності у єдиний програмний комплекс;
- провести експериментальне тестування працездатності створеного вебсервісу на реальних наборах даних (конфігураціях workflow).

2 РОЗРОБЛЕННЯ ТА МАТЕМАТИЧНЕ ОБГРУНТУВАННЯ МЕТОДУ ВИЯВЛЕННЯ ДУБЛІКАТІВ ПОСЛІДОВНОСТЕЙ

2.1 Загальна модель процесу виявлення дублікатів послідовностей

Виявлення дублікатів послідовностей генерації зображень є задачею порівняння структурованих конфігурацій, що описують послідовність дій для отримання зображення. На відміну від звичайного порівняння текстових файлів, у цій задачі потрібно враховувати не лише запис JSON-документа, а й логічну структуру робочих послідовностей: склад вузлів, зв’язки між ними, параметри генерації та налаштування LoRA-моделей.

Запропонований метод ґрунтується на послідовному перетворенні вхідного процесу workflow у нормалізоване графове представлення, формуванні канонічного опису, обчисленні хешу для виявлення точних дублікатів та оцінюванні подібності для пошуку близьких процесів workflow. Такий підхід дає змогу відокремити суттєві ознаки процесу workflow від несуттєвих особливостей його збереження, наприклад порядку вузлів у JSON-файлі або службових ідентифікаторів.

Вхідними даними методу є дві послідовності у форматі JSON або одна послідовність, яка порівнюється з набором раніше збереженими послідовностями. Вихідними даними є результат порівняння, що містить ознаку точного дубліката, інтегральну оцінку подібності, окремі оцінки за групами ознак та текстове пояснення основних збігів і відмінностей.

Загальний процес виявлення дублікатів складається з таких кроків:

Крок 1. Зчитується JSON-опис процесу workflow та перевіряється коректність його структури.

Крок 2. Виділяються вузли процесів workflow, їх параметри та посилання на інші вузли.

Крок 3. На основі знайдених посилань будується граф залежностей процесів workflow.

Крок 4. Для побудованого графа формується канонічне представлення, незалежне від порядку запису вузлів і службових ідентифікаторів.

Крок 5. Для канонічного представлення обчислюється хеш SHA-256.

Крок 6. Формуються множини ознак для порівняння: типи вузлів, сигнатури зв’язків і literal-параметри.

Крок 7. Обчислюється подібність процесів workflow за вузлами, зв’язками та параметрами.

Крок 8. На основі хешів і загальної оцінки подібності визначається результат порівняння.

Загальну схему запропонованого методу виявлення дублікатів послідовностей показано на рисунку 2.1.

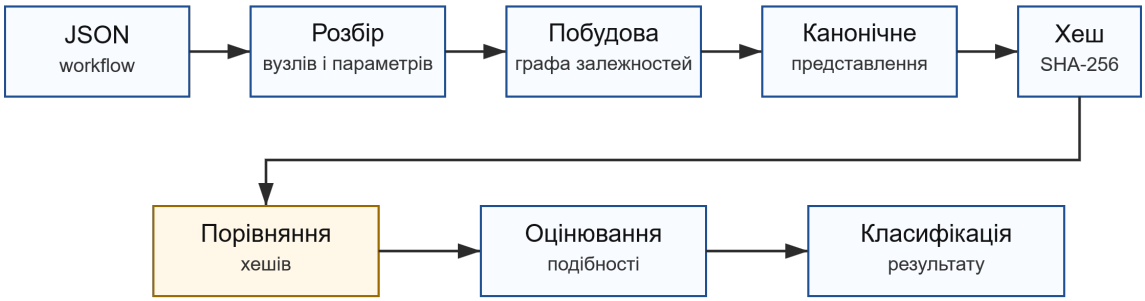

Рисунок 2.1 – Загальна схема методу виявлення дублікатів процесу workflow

Для деталізації вхідних даних і результатів кожного етапу методу основні етапи наведено у таблиці 2.1.

Таблиця 2.1 – Основні етапи методу виявлення дублікатів послідовностей

Етап	**Вхідні дані**	**Результат**
1	**2**	**3**
Зчитування процесу workflow	JSON-опис процесу workflow	Перевірена структура даних

Продовження таблиці 2.1

1	2	3
Виділення вузлів	Список вузлів процесу workflow	Набір вузлів з параметрами
Побудова графа	Вузли та посилання	Граф залежностей процесів workflow
Канонікалізація	Граф процесів workflow	Стабільне канонічне представлення
Хешування	Канонічне представлення	SHA-256-хеш процесів workflow
Оцінювання подібності	Ознаки двох процесів workflow	Інтегральна оцінка подібності
Класифікація	Хеші та оцінка подібності	Вердикт порівняння

2.2 Формалізація послідовностей генерації зображень

Послідовність генерації зображень розглядається як скінченна множина вузлів, кожен з яких виконує певну операцію у процесі генерації. Такими операціями можуть бути завантаження моделі, підключення LoRA-моделі, задання текстового запиту, вибір параметрів вибірки, формування латентного представлення, декодування зображення та збереження результату.

Формально процес workflow (W) можна подати у вигляді:

$$W = (N, E, P), \tag{2.1}$$

де N – множина вузлів процесу workflow;

E – множина зв’язків між вузлами;

P – множина параметрів процесу workflow.

Кожен вузол $n \in N$ описується трійкою:

$$n = (id, type, L), \tag{2.2}$$

де *id* – службовий ідентифікатор вузла;

type – тип вузла;

L – множина -параметрів вузла.

До literal-параметрів належать значення, які не є посиланнями на інші вузли: текстові запити, назви моделей, параметри LoRA, seed, кількість кроків, ширина та висота зображення, назва алгоритму вибірки та інші налаштування.

Зв’язок між вузлами описується як:

$$e = \left(n_i, n_j, input, output\right), \tag{2.3}$$

де n_i – вузол-джерело;

n_j – вузол-приймач;

input – назва входу вузла-приймача;

output – номер або назва виходу вузла-джерела.

Важливою особливістю послідовності є те, що службові ідентифікатори вузлів не повинні визначати подібність. Одна і та сама послідовність може бути збережений з різними *id*, але мати однакову логіку виконання. Тому під час формалізації *id* використовується лише для побудови зв’язків, а під час подальшого порівняння основну роль відіграють типи вузлів, параметри та структура залежностей.

2.2.1 Вимоги до вхідного опису послідовностей

Для коректної роботи методу вхідна послідовність повинна містити достатню кількість структурованих даних для відновлення логіки генерації

зображення. Основними елементами такого опису є вузли, типи вузлів, вхідні параметри та посилання на інші вузли. Якщо один із цих елементів відсутній, подальше порівняння може бути неповним або некоректним.

Вхідний опис процесів workflow має відповідати таким вимогам:

− кожен вузол процесу workflow повинен мати службовий ідентифікатор, який використовується для встановлення зв’язків між вузлами;

− для кожного вузла має бути визначено його тип, оскільки саме тип вузла описує функціональне призначення етапу генерації;

− вхідні параметри вузла повинні бути поділені на literal-параметри та посилання на інші вузли;

− literal-параметри повинні зберігати значення, що безпосередньо впливають на поведінку процесу workflow, зокрема назви моделей, запити, seed, кількість кроків, розміри зображення та параметри LoRA;

− посилання між вузлами повинні дозволяти відновити напрям передавання даних у процесі генерації.

Таке формулювання вимог дає змогу відокремити суттєві характеристики послідовностей від службових деталей його збереження. Наприклад, порядок розташування вузлів у JSON-файлі не є визначальним для логіки генерації, тоді як типи вузлів і зв’язки між ними безпосередньо впливають на результат. Тому під час аналізу процесу workflow службові ідентифікатори використовуються лише для побудови графа, але не розглядаються як самостійна ознака подібності (табл. 2.2).

Таблиця 2.2 – Складові вхідного опису процесу workflow

Складова	**Призначення**	**Використання у методі**
1	**2**	**3**
Ідентифікатор вузла	Встановлення посилань між вузлами	Побудова графа залежностей

Продовження таблиці 2.2

1	2	3
Тип вузла	Визначення функції вузла	Порівняння складу процесу workflow
-параметри	Збереження налаштувань генерації	Оцінювання поведінкової подібності
Посилання	Опис залежностей між вузлами	Формування ребер графа
Назви входів і виходів	Уточнення напряму передавання даних	Формування сигнатур зв’язків

2.3 Побудова графа залежностей послідовностей

Для подальшого аналізу послідовність подається у вигляді напрямленого графа:

$$G = (V, A), \tag{2.4}$$

де V – множина вершин графа, що відповідають вузлам послідовності;

A – множина напрямлених ребер, що відповідають залежностям між вузлами.

Якщо результат вузла v_i передається на вхід вузла v_j, то у графі створюється напрямлене ребро a_{ij}. Напрям ребра відображає напрям передавання даних у процесі workflow. Таке подання дає змогу аналізувати не порядок запису вузлів у JSON-файлі, а реальну логіку виконання процесу генерації.

Приклад подання процесу workflow генерації зображень у вигляді графа залежностей показано на рисунку 2.2.

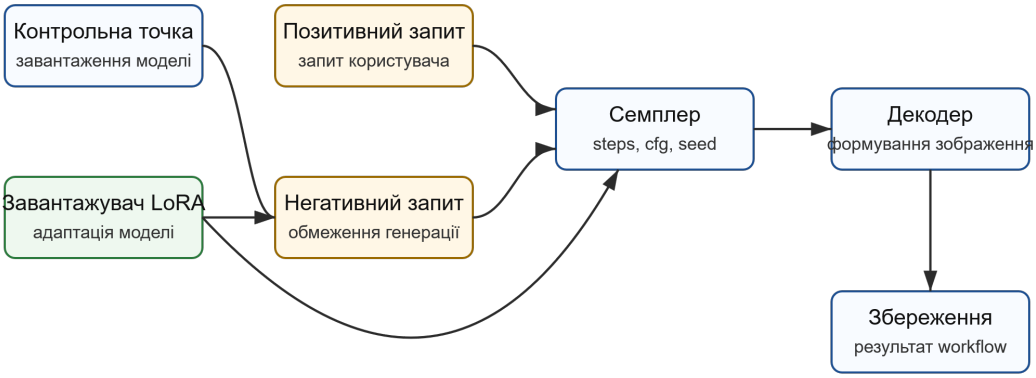

Рисунок 2.2 – Подання процесу workflow у вигляді графа залежностей

Під час побудови графа всі входи вузлів поділяються на дві групи:

− literal-параметри, що задають конкретні значення;

− посилання, що задають залежності від інших вузлів.

Наприклад, назва LoRA-моделі або значення seed є literal-параметрами, а посилання на вихід вузла завантаження моделі є структурним зв’язком. Такий поділ є необхідним, оскільки параметри та зв’язки порівнюються різними способами.

Граф залежностей дає змогу виконувати аналіз процесів workflow на кількох рівнях. На першому рівні порівнюється склад вузлів. На другому рівні аналізуються зв’язки між вузлами. На третьому рівні враховуються параметри, які визначають поведінку процесу workflow під час генерації зображення.

2.3.1 Формування ознак графа для подальшого порівняння

Після побудови графа залежностей необхідно сформувати ознаки, які будуть використовуватися для порівняння процесів workflow. У межах запропонованого методу ознака розглядається як формалізований опис певної властивості процесу workflow. Такими властивостями є наявність вузлів певного типу, наявність зв’язків між вузлами та значення параметрів, що впливають на результат генерації.

Формування ознак здійснюється за наступними кроками:

Крок 1. Для кожного вузла визначається його тип і формується ознака типу вузла.

Крок 2. Для кожного ребра графа формується сигнатура зв’язку, що враховує назву входу, номер виходу та тип вузла-приймача.

Крок 3. Для кожного literal-параметра формується параметрична ознака, що містить тип вузла, назву параметра та нормалізоване значення.

Крок 4. Отримані ознаки групуються за категоріями: ознаки вузлів, ознаки зв’язків і ознаки параметрів.

Крок 5. Для кожної групи ознак формується лічильник появ, який використовується під час оцінювання подібності.

Використання лічильників є важливим, оскільки один і той самий тип вузла може зустрічатися у процесах workflow кілька разів. Просте збереження множини типів вузлів у такому випадку призвело б до втрати інформації про структуру процесів workflow. Наприклад, два процеси workflow можуть містити вузол завантаження LoRA, але в одному процесі workflow він використовується один раз, а в іншому – декілька разів. Такі процеси workflow не повинні вважатися повністю однаковими за структурою.

Ознаки графа дозволяють перейти від порівняння складної структури процесів workflow до порівняння формалізованих наборів характеристик. Це спрощує обчислення подібності та робить результат більш пояснюваним.

2.3.2 Приклад формування графа залежностей процесу workflow

Для пояснення побудови графа залежностей розглянемо умовний процес workflow генерації зображення, який містить типові етапи: завантаження базової моделі, підключення LoRA-моделі, введення позитивного та негативного запитів, налаштування вибірки, декодування результату та збереження зображення.

У такому процесі workflow вузол завантаження контрольної точки моделі є початковим елементом, оскільки саме він надає базову модель для подальшої генерації. Вузол підключення LoRA-моделі використовує вихід цього вузла та змінює поведінку базової моделі відповідно до параметрів адаптації. Позитивний і негативний запити задають текстові умови генерації. Метод вибірки використовує модель, запити та параметри генерації для формування латентного результату. Декодер перетворює латентне представлення на зображення, після чого вузол збереження формує кінцевий результат.

Формування графа залежностей для такого процесу workflow здійснюється наступними кроками:

Крок 1. Визначаються всі вузли процесу workflow та їх типи.

Крок 2. Для кожного вузла аналізуються вхідні параметри.

Крок 3. Якщо параметр є посиланням на інший вузол, між відповідними вузлами створюється напрямлене ребро.

Крок 4. Якщо параметр не є посиланням, він зберігається як literal-параметр вузла.

Крок 5. Після обробки всіх вузлів формується повний граф залежностей процесу workflow.

Наприклад, якщо вузол вибірки отримує модель від вузла LoRA, позитивний запит від вузла текстового опису та негативний запит від окремого вузла обмежень, то у графі створюються три напрямлені ребра до вузла вибірки. Це дозволяє явно показати, від яких елементів залежить процес генерації.

Приклад відповідності між вузлами процесу workflow та елементами графа залежностей наведено у таблиці 2.3. Таке зіставлення дозволяє наочно продемонструвати взаємозв'язок між послідовністю дій користувача та архітектурою передачі даних усередині застосунку. Кожен функціональний крок, оператор або блок обробки інформації в межах процесу workflow відображається у вигляді окремої вершини (вузла).

Таблиця 2.3 – Приклад формування графа залежностей процесу workflow

Елемент процесу workflow	**Роль у процесі генерації**	**Елемент графа**
Контрольна точка моделі	Завантаження базової моделі	Початкова вершина
LoRA-модель	Адаптація базової моделі	Вершина з вхідним ребром від моделі
Позитивний запит	Опис бажаного результату	Вершина з текстовими параметрами
Негативний запит	Опис небажаних ознак	Вершина з текстовими параметрами
Метод вибірки	Формування латентного результату	Вершина з кількома вхідними ребрами
Декодер	Перетворення латентного представлення на зображення	Вершина після алгоритму вибірки
Збереження зображення	Формування кінцевого результату	Кінцева вершина

2.4 Канонікалізація графового представлення

Канонікалізація потрібна для приведення процесу workflow до стабільного представлення, яке не залежить від порядку запису вузлів і службових ідентифікаторів [27]. Без цього два однакові за логікою процеси workflow можуть бути помилково визначені як різні.

Канонічне представлення формується на основі типів вузлів, literal-параметрів, вхідних і вихідних зв’язків [28]. Для кожного вузла створюється локальний опис, що містить тип вузла та нормалізовані параметри. Після цього інформація про сусідні вузли враховується через вхідні та вихідні зв’язки. Таким чином, вузол описується не лише власними характеристиками, а й положенням у графі [29].

Процес канонікалізації здійснюється за наступною схемою:

Крок 1. Нормалізуються literal-параметри кожного вузла процесу workflow.

Крок 2. Для кожного вузла формується початкова сигнатура на основі його типу та параметрів.

Крок 3. До сигнатури вузла додається інформація про вхідні та вихідні зв’язки із сусідніми вузлами.

Крок 4. Уточнення сигнатур повторюється доти, доки представлення вузлів не стабілізується.

Крок 5. Формується загальне канонічне представлення графа як впорядкований набір сигнатур вузлів і ребер.

Послідовність канонікалізації графового представлення процесів workflow показано на рисунку 2.3.

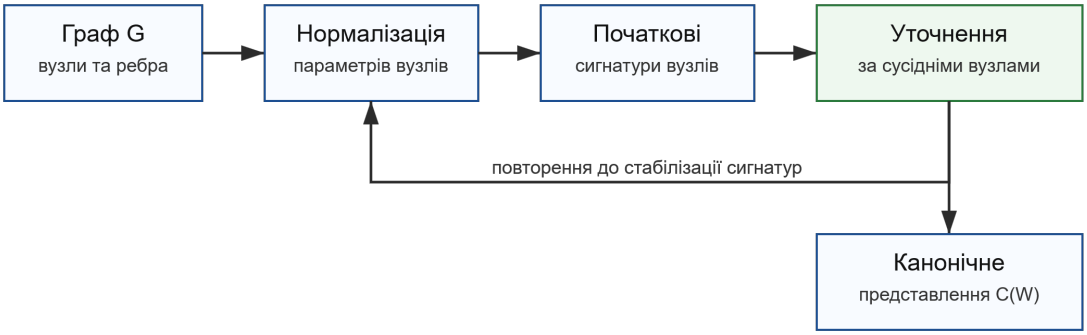

Рисунок 2.3 – Послідовність канонікалізації графового представлення

Нормалізація параметрів означає приведення словників і списків до стабільного порядку. Це необхідно, оскільки однакові параметри можуть бути записані у різному порядку. Після нормалізації однакові за змістом структури отримують однаковий текстовий опис.

Результатом канонікалізації є рядок *C(W)*, який однозначно описує суттєву структуру процесу workflow. Саме цей рядок використовується для хешування та точного порівняння.

2.4.1 Деталізація етапів канонікалізації процесів workflow

Канонікалізація є одним із ключових етапів методу, оскільки саме вона забезпечує незалежність результату від службових особливостей JSON-опису. Без канонікалізації однакові процеси workflow, збережені з різними ідентифікаторами вузлів або в іншому порядку, можуть отримати різні представлення.

Дана процедура здійснюється наступними кроками:

Крок 1. Для кожного вузла формується локальний опис, який містить тип вузла та нормалізовані literal-параметри.

Крок 2. Локальний опис перетворюється на початкову сигнатуру вузла.

Крок 3. Для кожного вузла визначаються вхідні та вихідні зв’язки.

Крок 4. Сигнатура вузла уточнюється з урахуванням сигнатур сусідніх вузлів.

Крок 5. Процес уточнення повторюється до моменту, коли сигнатури вузлів перестають змінюватися.

Крок 6. На основі стабілізованих сигнатур формується впорядкований опис усіх вузлів і ребер графа.

Крок 7. Отриманий опис перетворюється на канонічне представлення процесу workflow.

Перевагою такого підходу є те, що він враховує не лише властивості окремого вузла, а й його положення у структурі процесів workflow. Наприклад, два вузли одного типу можуть мати однакові параметри, але виконувати різні ролі, якщо вони підключені до різних частин графа. Уточнення сигнатур через сусідні вузли дозволяє врахувати цю відмінність.

Канонічне представлення використовується як основа для подальшого хешування. Якщо два процеси workflow після канонікалізації мають однаковий опис, вони повинні отримати однаковий хеш. Це дозволяє швидко визначати точні дублікати без повторного повного аналізу структури.

2.5 Виявлення точних дублікатів за допомогою хешування

Для виявлення точних дублікатів використовується хешування канонічного представлення процесу workflow [30]. Хешування дає змогу перетворити канонічний опис довільної довжини на рядок фіксованої довжини. У роботі доцільно використовувати SHA-256, оскільки цей алгоритм формує стабільне хеш-значення та широко застосовується для перевірки цілісності даних.

Хеш процесу workflow визначається як:

$$H(W) = SHA\text{-}256\big(C(W)\big), \tag{2.5}$$

де $C(W)$ – канонічне представлення процесу workflow;

Два процеси workflow W_1 і W_2 вважаються точними дублікатами, якщо виконується умова:

$$H(W_1) = H(W_2). \tag{2.6}$$

Це означає, що після нормалізації та канонікалізації вони мають однакову структуру, однакові суттєві параметри та однакові зв'язки між вузлами. Перевагою такого підходу є швидкість перевірки: для пошуку точного дубліката достатньо порівняти хеші.

Недоліком хешування є його чутливість до будь-яких змін. Якщо у процесі workflow змінити один параметр, наприклад seed або кількість кроків генерації, хеш стане іншим [31]. Тому хешування використовується лише для визначення точних дублікатів, а для пошуку подібних процесів workflow застосовується окреме оцінювання подібності.

2.5.1 Особливості використання хешування для виявлення дублікатів

Хешування використовується у методі як засіб швидкого виявлення точних дублікатів процесів workflow. Основна перевага хешування полягає в тому, що воно дозволяє замінити складну структуру процесу workflow коротким стабільним значенням фіксованої довжини. Після цього для перевірки точного збігу достатньо порівняти два хеші.

Однак важливо, що хешування не повинно виконуватися над початковим JSON-файлом без попередньої обробки. Початковий JSON-опис може містити службові ідентифікатори, різний порядок вузлів, різний порядок параметрів або інші технічні відмінності, які не змінюють логіку процесів workflow. Якщо хешувати такий файл напряму, то однакові за змістом процеси workflow можуть отримати різні хеші.

Саме тому у запропонованому методі хешування застосовується лише після побудови канонічного представлення. Канонічне представлення усуває несуттєві відмінності та зберігає ті характеристики, які визначають структуру і поведінку процесу workflow. Це дозволяє порівнювати не форму запису, а зміст конфігурації генерації.

Використання хешування має такі переваги:

− швидкість перевірки точних дублікатів;

− можливість збереження хешу в базі даних для подальшого пошуку;

− незалежність порівняння від розміру початкового JSON-файлу;

− зменшення обсягу даних, які потрібно порівнювати під час перевірки.

Водночас хешування має обмеження. Навіть незначна зміна параметра призводить до формування іншого хешу. Наприклад, якщо у процесі workflow змінити лише значення seed або кількість кроків генерації, хеш стане іншим, хоча загальна структура процесу workflow залишиться майже незмінною. Тому хешування не може використовуватися як єдиний механізм пошуку подібних процесів workflow.

У запропонованому методі хешування виконує роль першого рівня перевірки. Якщо хеші двох процесів workflow збігаються, система визначає їх як точні дублікати. Якщо хеші різні, виконується другий рівень аналізу – оцінювання подібності за вузлами, зв’язками та параметрами. Така комбінація дозволяє поєднати точність хешування та гнучкість структурного порівняння.

2.6 Оцінювання подібності процесів workflow

Для виявлення подібних процесів workflow використовується порівняння ознак. Ознаки поділяються на три основні групи:

− типи вузлів;

− сигнатури зв’язків;

− параметри.

Подібність двох множин ознак обчислюється як відношення кількості спільних ознак до загальної кількості ознак [32]. Для ознак, що можуть повторюватися, використовується підхід з лічильниками. Тоді подібність визначається формулою:

$$sim(A,B) = \frac{\sum_i min(A_i,B_i)}{\sum_i max(A_i,B_i)}, \quad (2.7)$$

де A_i і B_i – кількість появ ознаки з індексом i у першому та другому процесі workflow відповідно.

Окремо обчислюються:

− *node_similarity* – подібність за типами вузлів;

− *edge_similarity* – подібність за зв'язками;

− literal_*similarity* – подібність за параметрами.

Literal-параметри мають різну важливість. Параметри, пов'язані з моделлю, LoRA, запитом, алгоритмом вибірки та планувальником, мають сильний вплив на результат генерації [33]. Параметри *steps, cfg, width, height* і *denoise* мають середній вплив. Seed має нижчу вагу, оскільки його зміна може змінити конкретний результат, але не обов'язково змінює логіку процесу workflow. Групи параметрів процесів workflow за рівнем важливості наведено у таблиці 2.4.

Таблиця 2.4 – Групи параметрів workflow за рівнем важливості

Група	**Приклади параметрів**	**Вага**
Сильні параметри	prompt, text, model, LoRA, strength, sampler, scheduler	0,6
Середні параметри	steps, cfg, width, height, denoise	0,3
Слабкі параметри	seed, noise_seed	0,1

Подібність за параметрами визначається як:

$$S_{literal} = 0{,}6 \cdot S_{strong} + 0{,}3 \cdot S_{medium} + 0{,}1 \cdot S_{weak}, \quad (2.8)$$

де S_{strong} – подібність сильних параметрів;

S_{medium} – подібність середніх параметрів;

S_{weak} – подібність слабких параметрів.

Загальна оцінка подібності процесів workflow визначається формулою:

$$S = 0{,}45 \cdot S_{node} + 0{,}35 \cdot S_{edge} + 0{,}20 \cdot S_{literal}, \quad (2.9)$$

де S_{node} – подібність за типами вузлів;

S_{edge} – подібність за зв’язками;

$S_{literal}$ – подібність за literal-параметрами.

Таке співвідношення ваг пояснюється тим, що структура процесівв workflow має найбільше значення для визначення його логіки. Зв’язки між вузлами також істотно впливають на процес генерації. Параметри уточнюють поведінку процесу workflow, але їх зміна не завжди означає повну відмінність конфігурації.

2.6.1 Обґрунтування вагових коефіцієнтів оцінювання подібності

У запропонованому методі загальна оцінка подібності формується на основі трьох складових: подібності типів вузлів, подібності зв’язків і подібності параметрів. Вагові коефіцієнти обрано таким чином, щоб найбільший вплив мала структура процесів workflow, оскільки саме вона визначає логіку процесу генерації [34].

Подібність за типами вузлів має вагу 0,45. Це пояснюється тим, що набір вузлів визначає основні операції процесів workflow: завантаження моделей,

застосування LoRA, формування запитів, вибірку, декодування та збереження результату. Якщо два процеси workflow мають суттєво різний набір вузлів, вони не можуть вважатися близькими навіть за умови збігу окремих параметрів.

Подібність за зв’язками має вагу 0,35. Зв’язки показують, як саме дані передаються між вузлами. Два процеси workflow можуть містити однакові типи вузлів, але різний порядок їх взаємодії. У такому випадку результат генерації може відрізнятися, тому структура ребер має значний вплив на загальну оцінку.

Подібність за literal-параметрами має вагу 0,20. Параметри уточнюють поведінку процесу workflow, але їх зміна не завжди означає зміну загальної логіки. Наприклад, зміна seed може вплинути на конкретне зображення, але не змінює структуру процесу генерації. Водночас зміна назви LoRA-моделі або текстового запиту має більший вплив, тому всередині цієї групи параметри додатково поділяються за рівнем важливості.

Таким чином, запропонована система ваг дозволяє поєднати структурний і поведінковий аналіз. Структурна частина відповідає за порівняння логіки процесів workflow, а параметрична частина дозволяє врахувати відмінності у налаштуваннях генерації.

2.6.2 Приклад розрахунку оцінки подібності процесів workflow

Для пояснення роботи методу розглянемо умовний приклад порівняння двох процесів workflow. Нехай після аналізу двох процесів workflow отримано такі часткові оцінки: подібність за типами вузлів становить 0,90, подібність за зв’язками – 0,85, а подібність за literal-параметрами – 0,70.

Тоді загальна оцінка подібності обчислюється так:

$$S = 0{,}45 \cdot 0{,}90 + 0{,}35 \cdot 0{,}85 + 0{,}20 \cdot 0{,}70. \quad (2.10)$$

Після виконання обчислень отримуємо:

$$S = 0{,}405 + 0{,}2975 + 0{,}14 = 0{,}8425. \quad (2.11)$$

Отримане значення округлюється до 0,8425. Відповідно до правил класифікації такий процес workflow не належить до класу *Highly Similar*, оскільки значення менше за 0,85, але належить до класу *Moderately Similar* (табл. 2.5).

Таблиця 2.5 – Приклад розрахунку оцінки подібності процесів workflow

Складова	**Значення**	**Вага**	**Добуток**
Подібність вузлів	0,90	0,45	0,405
Подібність зв’язків	0,85	0,35	0,2975
Подібність параметрів	0,70	0,20	0,1400
Загальна оцінка	-	-	0,8425

2.6.3 Обґрунтування порогових значень класифікації

Після обчислення інтегральної оцінки подібності необхідно визначити, до якого класу належить результат порівняння. Для цього використовуються порогові значення, які дають змогу інтерпретувати числову оцінку у вигляді зрозумілого вердикту.

Після обчислення інтегральної оцінки подібності необхідно визначити, до якого класу належить результат порівняння. Для цього використовуються порогові значення, які дають змогу інтерпретувати числову оцінку у вигляді зрозумілого вердикту.

У запропонованому методі точний дублікат визначається не за порогом подібності, а за збігом хешів канонічного представлення. Це принципово важливо, оскільки навіть оцінка подібності, близька до 1, не гарантує повної ідентичності процесів workflow. Тільки збіг хешів після канонікалізації свідчить про однаковість суттєвої структури та параметрів.

Для процеси workflow з різними хешами використовується оцінка подібності S. Якщо $S \geq 0{,}85$, процеси workflow визначаються як такі, що мають високий рівень подібності. Це означає, що вони мають близьку структуру, значну кількість спільних вузлів і зв’язків, а також незначні відмінності у параметрах. Такі процеси workflow доцільно показувати користувачеві як потенційні дублікати або близькі варіанти.

Якщо $0{,}60 \leq S < 0{,}85$, процеси workflow мають помірний рівень подібності. У цьому випадку між ними є спільні ознаки, але відмінності вже суттєвіші. Наприклад, процеси workflow можуть використовувати схожу схему генерації, але мати інші запити, параметри LoRA або частково інший набір вузлів. Такі процеси workflow не варто вважати дублікованими автоматично, але їх доцільно залишати у результатах аналізу для ручної перевірки.

Якщо $S < 0{,}60$, процеси workflow визначаються як різні. Низьке значення оцінки означає, що між процесами workflow недостатньо спільних структурних або параметричних ознак. У такому випадку система не повинна попереджати користувача про можливе дублювання.

Обрані пороги мають практичний характер і можуть бути уточнені під час подальшого тестування на більшій кількості реальних процесів workflow. Наприклад, якщо система часто визначає різні процеси workflow як подібні, поріг високої подібності можна підвищити. Якщо ж система пропускає багато близьких процесів workflow, поріг можна знизити.

Таким чином, порогові значення виконують роль інтерпретаційного шару між числовою оцінкою та зрозумілим для користувача результатом.

2.7 Класифікація результатів порівняння

Після обчислення хешів і загальної оцінки подібності результат порівняння класифікується за набором правил. Якщо хеші двох процесів workflow збігаються, результат визначається як точний дублікат. Якщо хеші різні, використовується інтегральна оцінка подібності. Правила класифікації результатів порівняння процесів workflow наведено у таблиці 2.6.

Клас Exact Duplicate означає, що процеси workflow мають однакове канонічне представлення. Клас Highly Similar означає, що процеси workflow не є точними дублікатами, але мають високий рівень збігу структури та параметрів. Клас Moderately Similar використовується для процесів workflow, які мають часткову подібність. Клас Different означає, що процеси workflow суттєво відрізняються.

Таблиця 2.6 – Правила класифікації результатів порівняння

Умова	**Результат**
$H(W_1) = H(W_2)$	*Exact Duplicate*
$S \geq 0{,}85$	*Highly Similar*
$0{,}60 \leq S < 0{,}85$	*Moderately Similar*
$S < 0{,}60$	*Different*

Для підвищення зрозумілості результату метод має формувати пояснення порівняння.

2.7.1 Формування пояснення результату порівняння

Для практичного використання методу важливо не лише отримати числову оцінку подібності, а й пояснити, чому процеси workflow були віднесені до певного класу. Без пояснення користувач бачить лише

підсумковий вердикт, але не розуміє, які саме елементи процесів workflow збігаються або відрізняються.

Пояснення результату порівняння формується на основі ознак, отриманих під час аналізу процесів workflow. До таких ознак належать спільні типи вузлів, відсутні або додані типи вузлів, змінені literal-параметри та спільні сигнатури зв’язків. Ці дані дозволяють описати результат порівняння у зрозумілій формі.

Формування пояснення здійснюється наступними кроками:

Крок 1. Визначаються типи вузлів, які присутні в обох процесах workflow.

Крок 2. Визначаються типи вузлів, які є лише в одному з процесів workflow.

Крок 3. Порівнюються literal-параметри та визначаються параметри, значення яких відрізняються.

Крок 4. Визначаються спільні сигнатури зв’язків між вузлами.

Крок 5. На основі знайдених збігів і відмінностей формується коротке текстове пояснення.

Наприклад, якщо два процеси workflow мають однакові вузли завантаження моделі, підключення LoRA та вибірки, але відрізняються значеннями запиту і кроків, система може повідомити, що процеси workflow мають спільну структуру, однак відрізняються параметрами запиту та кількістю кроків генерації.

Складові пояснення результату порівняння наведено у таблиці 2.7.

Таблиця 2.7 – Складові пояснення результату порівняння процесів workflow

Складова пояснення	**Зміст**	**Призначення**
1	**2**	**3**
Спільні типи вузлів	Типи вузлів, що є в обох процесах workflow	Пояснення структурної близькості

Продовження таблиці 2.7

1	2	3
Додані типи вузлів	Вузли, що є лише у другому процесі workflow	Виявлення розширень структури
Відсутні типи вузлів	Вузли, що є лише у першому процесі	Виявлення втрат структури
Змінені параметри	Literal-параметри з різними значеннями	Пояснення поведінкових відмінностей
Спільні зв’язки	Однакові сигнатури ребер	Пояснення поведінкових відмінностей

Наявність пояснення підвищує практичну цінність методу, оскільки користувач може самостійно оцінити, чи є знайдений процес workflow справжнім дублікатом, його варіацією або незалежною конфігурацією. Це особливо важливо для процесів workflow генерації зображень, де невелика зміна параметра може суттєво вплинути на результат.

2.8 Об’єктно-орієнтоване моделювання методу

Для опису структури методу доцільно використати об’єктно-орієнтовану модель, у якій кожна сутність відповідає окремому етапу обробки процесу workflow [35]. Такий підхід спрощує подальшу програмну реалізацію та дозволяє розділити відповідальність між компонентами.

Основними сутностями моделі є:

− WorkflowNode – вузол процесу workflow;

− ParsedWorkflow – результат початкового розбору процесу workflow;

− WorkflowGraph – граф залежностей;

− WorkflowFingerprint – відбиток процесу workflow для порівняння;

− ComparisonResult – результат порівняння двох процесів workflow;

− ComparisonExplanation – пояснення результату порівняння.

Об’єктно-орієнтовану модель запропонованого методу показано на рисунку 2.4.

Сутність WorkflowNode містить ідентифікатор вузла, тип вузла, literal-параметри та посилання на інші вузли. ParsedWorkflow містить набір таких вузлів після розбору JSON-структури. WorkflowGraph описує граф залежностей через множини вузлів, вхідних і вихідних ребер [36].

WorkflowFingerprint є проміжною сутністю, що містить канонічне представлення, хеш, лічильники типів вузлів, сигнатури зв’язків і параметрів. Саме ця сутність використовується для швидкого порівняння процесів workflow. ComparisonResult містить фінальний результат: ознаку точного дубліката, вердикт, загальну оцінку подібності та окремі оцінки за групами ознак. Основні сутності об’єктно-орієнтованої моделі наведено у таблиці 2.8. Впровадження сутності відбитка (fingerprint) дозволяє уникнути ресурсомісткого попарного порівняння повних графів на ранніх етапах фільтрації очевидно відмінних конфігурацій.[37, 38]

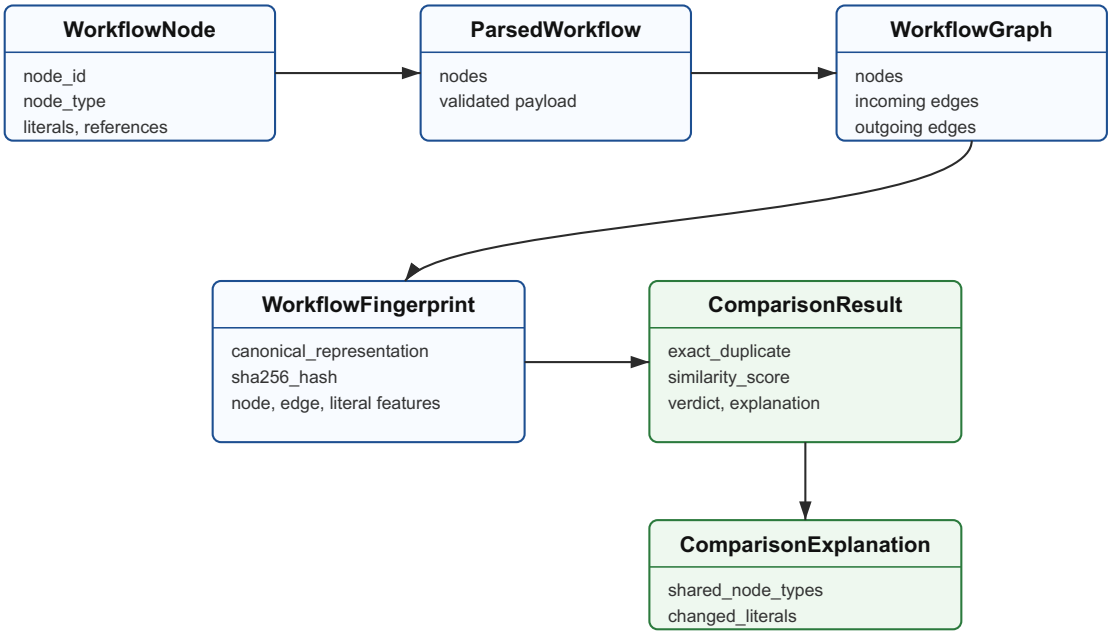

Рисунок 2.4 – Об’єктно-орієнтована модель методу

Таблиця 2.8 – Основні сутності об’єктно-орієнтованої моделі

Сутність	**Призначення**
WorkflowNode	Опис окремого вузла процесу workflow
ParsedWorkflow	Збереження результату розбору JSON
WorkflowGraph	Подання процесу workflow у вигляді графа залежностей
WorkflowFingerprint	Збереження ознак процесу workflow для порівняння
ComparisonResult	Формування результату порівняння
ComparisonExplanation	Пояснення збігів і відмінностей між процесами workflow

2.9 Узагальнений алгоритм роботи методу

Узагальнений алгоритм роботи методу можна подати у вигляді послідовності кроків [39]. Алгоритм виявлення дублікатів процесів workflow здійснюється за наступною схемою:

Крок 1. Отримується процес workflow у форматі JSON.

Крок 2. Перевіряється коректність структури вхідних даних.

Крок 3. Виділяються вузли процесу workflow.

Крок 4. Входи вузлів розділяються на literal-параметри та посилання на інші вузли.

Крок 5. Будується граф залежностей процесу workflow.

Крок 6. Формується канонічне представлення графа.

Крок 7. Обчислюється хеш SHA-256 для канонічного представлення.

Крок 8. Виділяються ознаки вузлів, зв’язків і параметрів.

Крок 9. Якщо порівнюються два процеси workflow, кроки 1-8 виконуються для другого процесу workflow.

Крок 10. Порівнюються хеші процесів workflow.

Крок 11. Якщо хеші збігаються, процеси workflow визначаються як точні дублікати.

Крок 12. Якщо хеші не збігаються, обчислюється подібність за вузлами, зв’язками і параметрами.

Крок 13. Обчислюється інтегральна оцінка подібності.

Крок 14. Визначається клас результату порівняння.

Крок 15. Формується пояснення результату порівняння.

Блок-схему алгоритму порівняння процесів workflow показано на рисунку 2.5.

Запропонований метод поєднує точне порівняння через хешування та гнучке оцінювання подібності. Це дає змогу виявляти як повні дублікати, так і подібні процеси workflow, які мають спільну структуру або відрізняються лише окремими параметрами [40]. Метод може бути використаний як основа

для програмного засобу аналізу колекцій процесів workflow генерації зображень для LoRA-моделей.

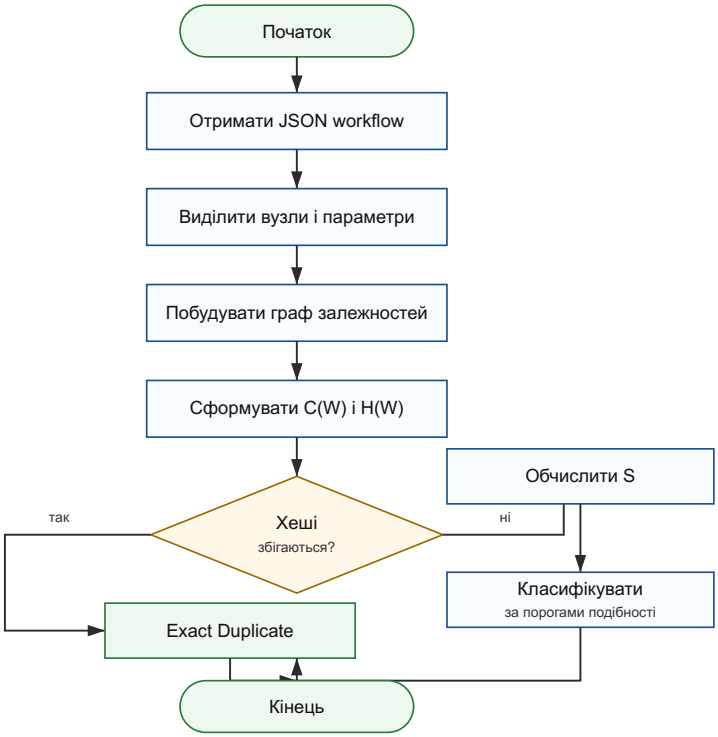

Рисунок 2.5 – Блок-схема алгоритму порівняння процесів workflow

3 РЕАЛІЗАЦІЯ ВЕБСЕРВІСУ СТРУКТУРНОГО АНАЛІЗУ РОБОЧИХ ПРОЦЕСІВ ГЕНЕРАЦІЇ ЗОБРАЖЕНЬ

3.1 Обґрунтування вибору інструментальних засобів та архітектури системи

Розроблення сучасного вебсервісу для виявлення дублікатів та оцінювання подібності графів опису технологічних схем генерації зображень потребує ретельного вибору технологічного набору. Основними критеріями вибору інструментів є продуктивність оброблення структурованих даних у форматі JSON, швидкість виконання обчислювальних операцій під час аналізування орієнтованих графів, підтримка асинхронного оброблення мережевих запитів та простота інтегрування розроблюваних модулів із сучасними СУБД.

3.1.1 Порівняльний аналіз технологій розроблення

Для вибору оптимального вебфреймворку для побудови REST API було проведено порівняльний аналіз FastAPI, Django та Flask за критеріями продуктивності, перевірка даних, генерації документації та архітектурної гнучкості. Традиційні інструменти на кшталт Django та Flask продемонстрували нижчу швидкість роботи через WSGI-архітектуру та потребу в налаштуванні сторонніх бібліотек для перевірки даних і створення документації [41]. Натомість FastAPI виділяється високою продуктивністю завдяки підтримці стандарту ASGI та автоматичному валідуванню вхідних JSON-структур за допомогою бібліотеки Pydantic безпосередньо на етапі маршрутизації. Додатковою перевагою цього фреймворку є автоматична генерація інтерактивної документації за специфікацією OpenAPI, що суттєво спрощує інтеграцію з клієнтською частиною. Зважаючи на ці технічні

характеристики, тому було обрано саме FastAPI, оскільки він гарантує максимальну швидкість оброблення паралельних запитів та високу стійкість системи до некоректних вхідних даних.

Для зберігання метаданих робочих процесів та результатів їх порівняння було обрано сучасні системи керування базами даних (СУБД). З метою забезпечення універсальності та гнучкості застосунку в різних інфраструктурних умовах було реалізовано гібридний підхід до проєктування шару даних [42].

Для локального розгортання та швидкого первинного тестування системи за умовчанням використовується вбудована СУБД SQLite. Це раціональне рішення дозволяє повністю уникнути потреби у тривалому адмініструванні серверної бази даних, налаштуванні прав доступу та забезпечує повну працездатність усього функціоналу відразу після завантаження проєкту.

Для повноцінного серверного розгортання, зокрема у продуктивному середовищі VPS за допомогою платформи контейнеризації Docker, передбачено легку міграцію на СУБД PostgreSQL. Такий перехід здійснюється безпосередньо через зміну системної змінної оточення WORKFLOW_DATABASE_URL і не потребує жодної модифікації програмного коду завдяки інтеграції інструменту об’єктно-реляційного відображення (ORM) SQLAlchemy.

3.1.2 Архітектурна структура вебсервісу та шаблони проєктування

Архітектура розроблюваної системи базується на принципах модульності та слабкої зв’язності компонентів. Для детального відображення взаємодії користувача із системою побудовано діаграму прецедентів (Use Case Diagram), яку зображено на рисунку 3.1. Користувач має можливість виконувати порівняння двох файлів робочих процесів без збереження в базу

даних, а також здійснювати перевірку завантаженого файлу на наявність копій чи структурно схожих графів у сховищі [43]. При спробі збереження точного дубліката (exact duplicate), система реалізує відношення розширення (<<extend>>) і виводить відповідне попередження, проте не блокує дію користувача, залишаючи рішення за ним.

Рисунок 3.1 – Діаграма прецедентів розроблюваного вебсервісу

Для опису внутрішньої організації сервісу розроблено компонентну архітектуру, що детально показує взаємозв'язки між клієнтською-частиною, сервером додатків та базою даних. Діаграму компонентів системи зображено на рисунку 3.2.

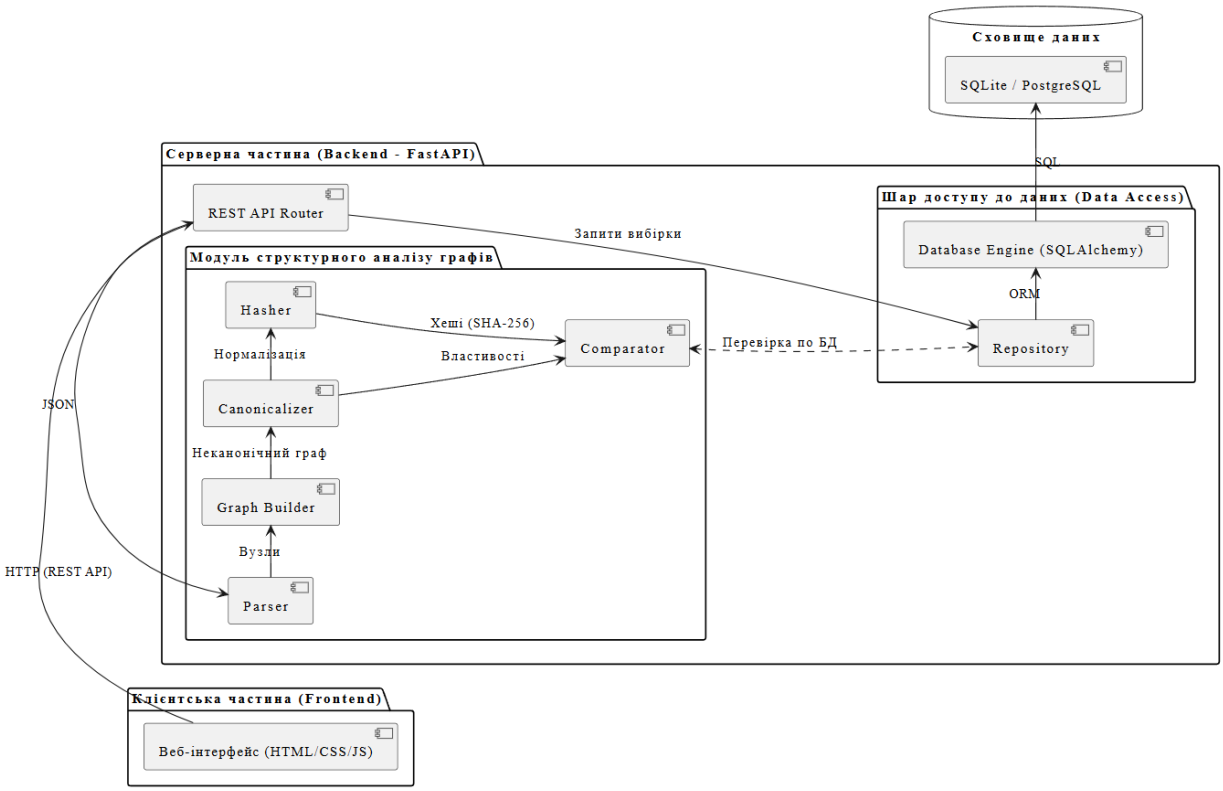

Рисунок 3.2 – Діаграма компонентів системи

Аналіз діаграми компонентів дозволяє виділити три логічні рівні розроблюваного застосунку. На рівні представлення (Presentation Layer) веб-інтерфейс взаємодіє з бекендом виключно через асинхронні HTTP-запити, передаючи та отримуючи структуровані JSON-документи.

Рівень бізнес-логіки (Domain Layer) представлений модулем аналізування графів, який виконує послідовну обробку даних. У його межах компонент Parser вилучає вузли та зв'язки, Builder будує орієнтований граф, Canonicalizer приводить структуру до канонічного вигляду без технічних ідентифікаторів, Hasher обчислює криптографічний хеш SHA-256 для

миттєвого виявлення точних дублікатів, а Comparator розраховує вагову метрику схожості (Similarity Score).

Рівень доступу до даних (Data Access Layer) відповідає за збереження та отримання інформації. Його реалізовано за допомогою патерну «Репозиторій» (Repository Pattern) та технології впровадження залежностей (Dependency Injection).

Наведений підхід забезпечує високу тестованість системи: під час юніт-тестування реальну базу даних можна легко підмінити на mock-об’єкт (імітацію), передавши тестову сесію у репозиторій. Вигляд організації проєкту подано на рисунку 3.3.

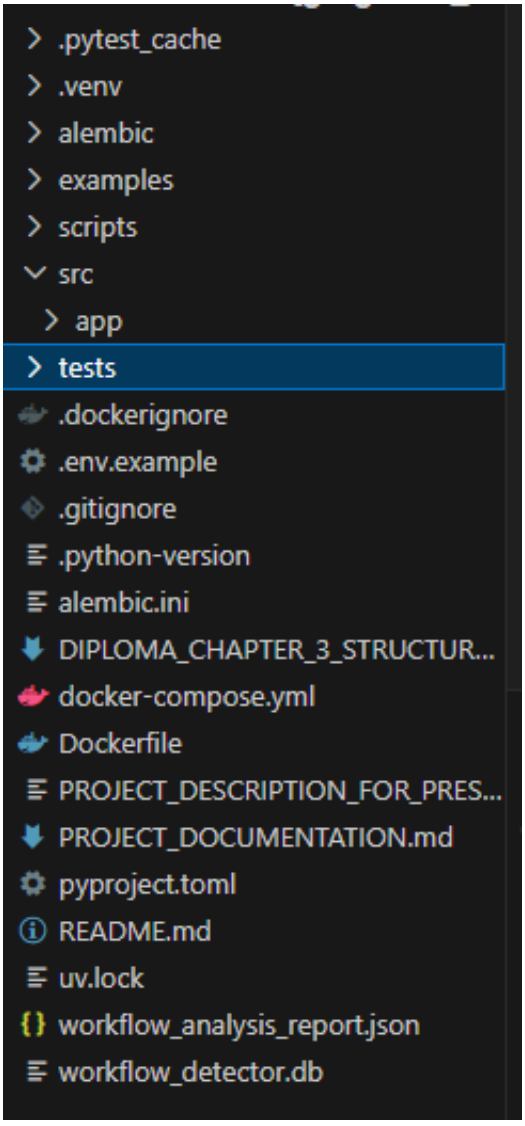

Рисунок 3.3 – Скріншот файлової структури проєкту

3.2 Програмна реалізація алгоритмів обробки та аналізу графів робочих процесів

3.2.1 Модуль збору та перетворення JSON-структур у граф залежностей

Основне призначення модуля інженерного аналізу та первинного оброблення даних полягає в уніфікації вхідних форматів представлення робочих процесів середовища ComfyUI та їхньому перетворенні у внутрішню об’єктну модель орієнтованого графа [1]. На етапі програмної реалізації було розроблено відокремлені модулі парсування (parser.py) та конструювання графових об’єктів (graph_builder.py), які функціонують послідовно у складі єдиного обчислювального конвеєра.

Для представлення компонентів робочого процесу на етапі парсування та побудови графа спроєктовано систему взаємопов’язаних класів даних (dataclasses). Використання стандартних класів даних мови Python дозволило чітко типізувати властивості вузлів та ребер. UML-діаграму класів проєкту, що відображає архітектуру об’єктів даних, їхні атрибути та взаємозв’язки, зображено на рисунку 3.4.

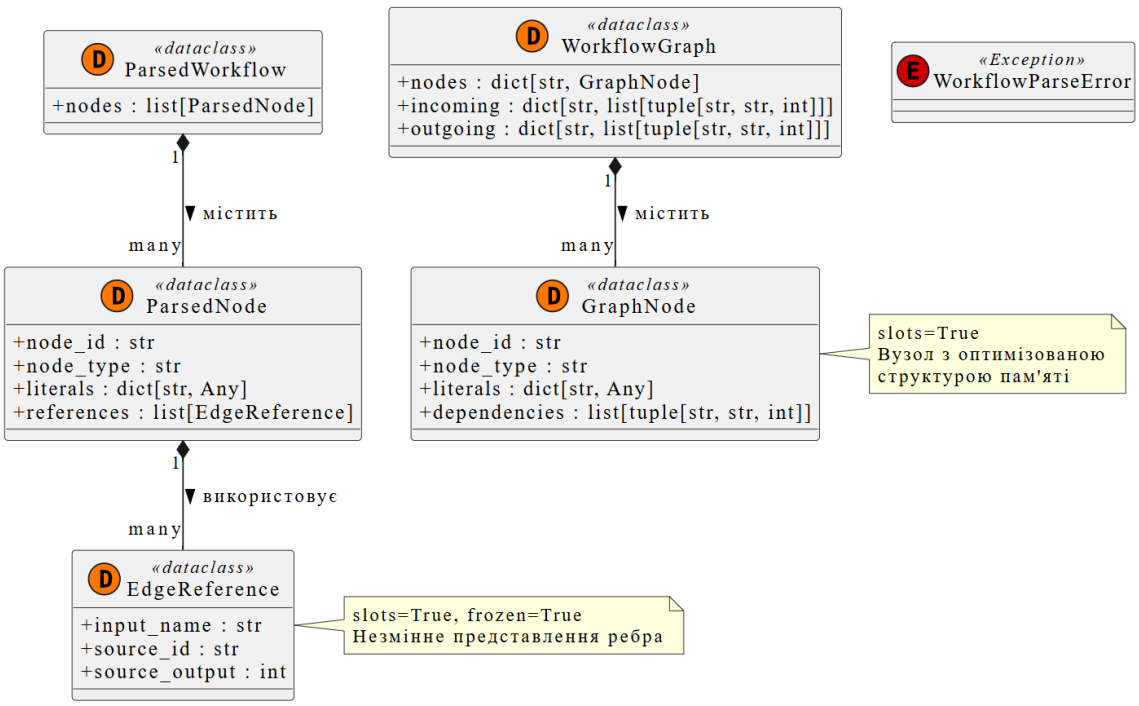

Рисунок 3.4 – UML-діаграма класів об’єктної моделі даних

Об’єкти проміжного представлення розділені на дві групи. Класи EdgeReference, ParsedNode та ParsedWorkflow відповідають за первинне збереження розпарсеної інформації, тоді як GraphNode та WorkflowGraph описують безпосередньо структуру зв’язного орієнтованого графа робочого процесу [45].

Для реалізації ребер орієнтованого графа на стадії збирання розроблено незмінний клас даних EdgeReference. Його структуру представлено в лістингу 3.1.

Лістинг 3.1 Структура класу EdgeReference:

```
@dataclass(slots=True, frozen=True)
class EdgeReference:
    input_name: str
    source_id: str
    source_output: int = 0
```

Параметр *frozen=True* гарантує незмінність ребра після ініціалізації. Використання *slots=True* прискорює доступ до полів за рахунок оптимізації виділення оперативної пам’яті на рівні інтерпретатора.

Під час імпортування файлу система виконує ідентифікацію структури JSON-документа. Спроєктований алгоритм підтримує два варіанти збереження даних у ComfyUI: API Prompt Format (лінійний словник вузлів) та Canvas Format (об’єкт із розділеними масивами nodes та links). Автоматичне визначення формату реалізовано шляхом валідації наявності специфічних маркерних ключів у корені імпортованого файлу, що зводить до мінімуму ризик виникнення критичних помилок під час зчитування структури. Отриманий результат первинної класифікації визначає подальший маршрут обробки даних відповідним субпарсером для їх подальшої трансформації у єдине внутрішнє представлення графа. Блок-схему (діаграму активностей UML), що описує логіку вибору алгоритму парсування (рис. 3.5).

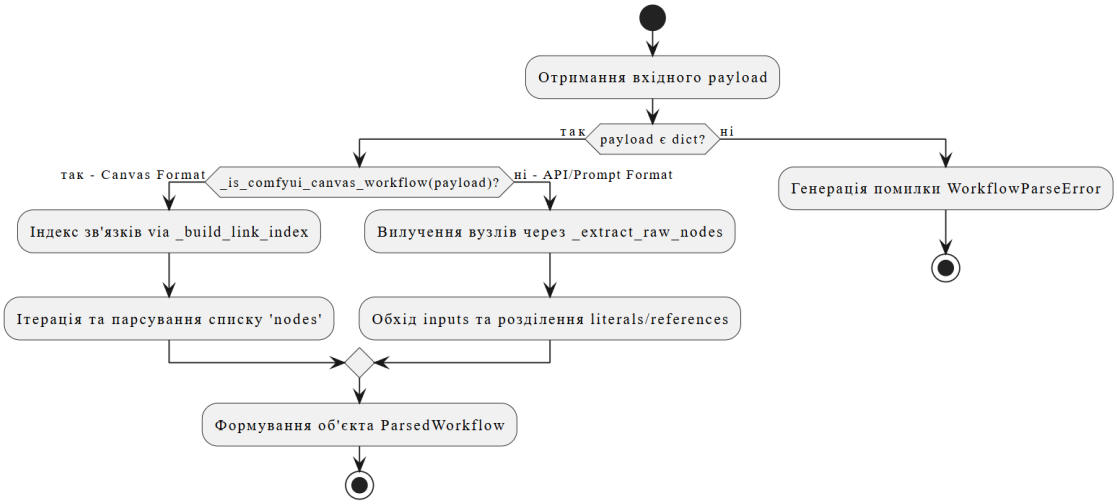

Рисунок 3.5 – UML-діаграма активностей процесу парсування вхідного документа

Для безпечного розбору вхідного документа спроєктовано точку входу parse_workflow, яка перевіряє базову структуру та виконує диспетчеризацію. Реалізацію функції представлено в лістингу 3.2.

Лістинг 3.2 Маршрутизація форматів у функції parse_workflow:

```
def parse_workflow(payload: dict[str, Any]) -> ParsedWorkflow:
    if not isinstance(payload, dict):
        raise WorkflowParseError("Вхідні дані мають бути JSON-об’єктом.")
    if _is_comfyui_canvas_workflow(payload):
        return _parse_comfyui_canvas_workflow(payload)
    raw_nodes = _extract_raw_nodes(payload)
```

Функція *_is_comfyui_canvas_workflow* аналізує наявність специфічних маркерів графічного інтерфейсу, запобігаючи помилкам несумісності.

Після отримання об’єкта ParsedWorkflow керування передається до конструктора графа build_graph у модулі graph_builder.py. Його завдання – побудувати двонаправлений орієнтований граф та сформувати індекси

зв’язків. Послідовність взаємодії компонентів відображено на UML-діаграмі послідовності на рисунку 3.6.

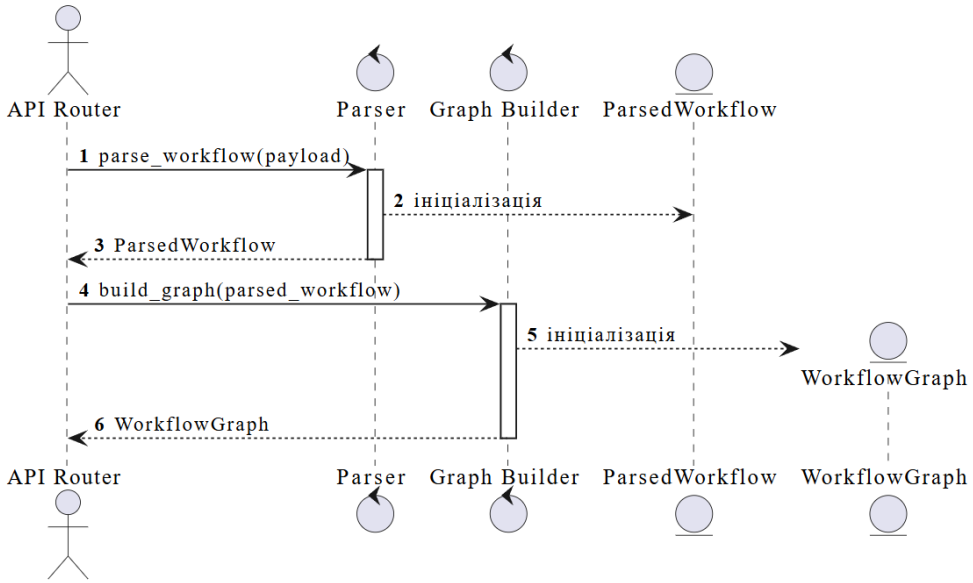

Рисунок 3.6 – UML-діаграма послідовності взаємодії компонентів під час генерації графа

Побудова графа відбувається у два кроки. На першому кроці виконується ініціалізація та реєстрація вершин у словнику, як показано у лістингу 3.3.

Лістинг 3.3 Ініціалізація вершин у build_graph:

```
nodes: dict[str, GraphNode] = {}
incoming: dict[str, list[tuple[str, str, int]]] = {}
outgoing: dict[str, list[tuple[str, str, int]]] = {}
for parsed_node in parsed.nodes:
    nodes[parsed_node.node_id] = GraphNode(
        node_id=parsed_node.node_id,      node_type=parsed_node.node_type,
literals=parsed_node.literals
    )
    incoming[parsed_node.node_id] = []
    outgoing[parsed_node.node_id] = []
```

На другому кроці виконується зв’язування ребер, зображене у лістингу 3.4.

Лістинг 3.4 Побудова зв’язків та індексів суміжності:

```
for parsed_node in parsed.nodes:
    for reference in parsed_node.references:
        if reference.source_id not in nodes:
            continue
        dependency = (reference.input_name, reference.source_id, reference.source_output)
        nodes[parsed_node.node_id].dependencies.append(dependency)
        incoming[parsed_node.node_id].append(dependency)
        outgoing[reference.source_id].append((reference.input_name, parsed_node.node_id, reference.source_output))
```

Логіка у лістингу 3.4 забезпечує двонаправленість зв’язків. Фільтрування неіснуючих ID вузлів-джерел гарантує стійкість системи до пошкоджених файлів. Завдяки сформованим індексам, подальші операції обходу графа виконуються за константний час $O(1)$.

3.2.2 Алгоритм структурного нормалізування та канонізування графа

Основою для точного виявлення дублікатів серед збережених робочих процесів є усунення несуттєвих відмінностей у представленні JSON-структур. До таких технічних розбіжностей належать випадкові унікальні ідентифікатори (ID) вузлів, які генеруються середовищем ComfyUI під час створення нових компонентів, а також випадковий порядок розташування об’єктів у файлі. Для приведення орієнтованого графа до єдиного,

незалежного від зовнішніх факторів вигляду розроблено модуль структурного нормалізування та канонізування (canonicalizer.py).

Програмний конвеєр канонізування складається з трьох послідовних стадій: стабільного впорядкування вершин, ітераційного поширення зв'язків для формування сигнатур оточення та фінального агрегування структурних елементів у компактне текстове представлення.

Першим кроком алгоритму є визначення детермінованого порядку обходу вершин. Для цього застосовується топологічне сортування на основі алгоритму Кана, що використовує чергу вершин із нульовим ступенем вхідних зв'язків. Ініціалізацію стартових вершин сортування наведено у лістингу 3.5.

Лістинг 3.5 Ініціалізація алгоритму топологічного сортування Кана:

```
indegree = {node_id: len(edges) for node_id, edges in graph.incoming.items()}
ready = deque(sorted(node_id for node_id, degree in indegree.items() if degree == 0))
order: list[str] = []
```

На другій стадії виконується розрахунок індивідуальних унікальних сигнатур (кольорів) для кожної вершини. Початковий колір формується виключно на основі типу вузла та його константних літералів. Процес первинного фарбування наведено у лістингу 3.6.

Лістинг 3.6 Первинне обчислення кольору на основі типу та літералів:

```
colors = {
    node_id: _hash_payload({
        "type": node.node_type,
        "literals": _normalize_value(node.literals),
    })
    for node_id, node in graph.nodes.items()
```

```
}
```

Для запобігання впливу різного порядку запису параметрів у вхідному JSON-файлі розроблено функцію рекурсивного сортування ключів словника, представлену у лістингу 3.7. Будь-який вхідний словник рекурсивно перетворюється на новий об’єкт із примусово відсортованими за алфавітом ключами, що нівелює вплив стилю запису файлу на фінальну сигнатуру.

Лістинг 3.7 Рекурсивне нормалізування ключів вхідних параметрів

```
def _normalize_value(value: Any) -> Any:
    if isinstance(value, dict):
        return {key: _normalize_value(value[key]) for key in sorted(value)}
    if isinstance(value, list):
        return [_normalize_value(item) for item in value]
    return value
```

Після ініціалізації запускається цикл поширення зв’язків. Його мета – поступово наситити колір кожної вершини інформацією про її сусідів у графі. Під час кожної ітерації уточнення кольору вершини збираються сигнатури її вхідних та вихідних зв’язків. Програмну реалізацію збирання сигнатур наведено у лістингу 3.8.

Лістинг 3.8 Формування сигнатур вхідного та вихідного оточення вузла:

```
incoming_signature = sorted(
    (input_name, output_index, colors[source_id])
    for input_name, source_id, output_index in graph.incoming[node_id]
)
outgoing_signature = sorted(
    (input_name, output_index, colors[target_id])
    for input_name, target_id, output_index in graph.outgoing[node_id]
```

)

Використання функції sorted для обох сигнатур у лістингу 3.8 гарантує стабільний лінійний опис, інваріантний до перестановки елементів у графі. Новий кольоровий стан вузла обчислюється шляхом упакування та хешування сигнатур, як показано в лістингу 3.9.

Лістинг 3.9 Розрахунок нового значення кольору вершини на ітерації:

```
payload = {
    «type»: node.node_type,
    «literals»: _normalize_value(node.literals),
    «incoming»: [{«input»: inp, «output»: out, «source»: col} for inp, out, col in incoming_signature],
    «outgoing»: [{«input»: inp, «output»: out, «target»: col} for inp, out, col in outgoing_signature],
    }
    updated[node_id] = _hash_payload(payload)
```

На завершальній стадії, після досягнення повної стабільності кольорів, оригінальні ідентифікатори вузлів повністю відкидаються. Для побудови фінального представлення використовуються частотні лічильники Counter, як продемонстровано у лістингу 3.10.

Лістинг 3.10 Підрахунок частот появи кольорів вершин та ребер:

```
node_counter = Counter(colors.values())
edge_counter = Counter(
   json.dumps({
       "source": colors[source_id],
       "target": colors[target_id],
       "input": input_name,
```

```
            "output": output_index,
        }, sort_keys=True, separators=(",", ":"))
        for source_id, edges in graph.outgoing.items()
        for input_name, target_id, output_index in edges
    )
```

Завдяки використанню Counter у лістингу 3.10, фінальний опис графа оперує виключно кількісними характеристиками ізоморфних підграфів. Отримані структури серіалізуються в ущільнений JSON-рядок, як показано у лістингу 3.11.

Лістинг 3.11 Стабільне детерміноване серіалізування в JSON:

```
    representation = {
        "node_count": len(graph.nodes),
        "topological_colors": sorted(colors[node_id] for node_id in sorted_nodes),
        "nodes": sorted(({"label": l, "count": c} for l, c in node_counter.items()), key=lambda x: x["label"]),
        "edges": sorted(({"edge": json.loads(e), "count": c} for e, c in edge_counter.items()), key=lambda x: json.dumps(x["edge"]))
    }
    return json.dumps(representation, sort_keys=True, separators=(",", ":"))
```

3.2.3 Генерація унікальних ідентифікаторів

Для оптимізації пошуку в базі даних та мінімізації витрат пам’яті розроблено модуль хешування (hasher.py), який замінює неефективне пряме порівняння об’ємних канонічних JSON-рядків перетворенням їх у фіксований 64-символьний шістнадцятковий відбиток. Основним інструментом генерації

цього унікального ідентифікатора обрано алгоритм криптографічного хешування SHA-256 із системної бібліотеки hashlib мови Python, що характеризується швидкістю обчислення, а також високою стійкістю до колізій та відновлення першоджерела. Цей обчислювальний процес є завершальним етапом виявлення дублікатів і складається з трьох послідовних кроків (отримання нормалізованого рядка, його кодування у байтовий потік за стандартом UTF-8 та розрахунку хеш-суми), що детально відображено на діаграмі активностей на рисунку 3.7.

Першим кроком у програмній реалізації є підготовка середовища та імпорт необхідних засобів криптографічного захисту. Цей етап ініціалізації модуля наведено в лістингу 3.12.

Оскільки алгоритм SHA-256 на низькому рівні оперує послідовностями байтів, безпосереднє передавання unicode-рядків у хеш-функцію є неможливим. Для забезпечення сумісності із символами національних алфавітів (наприклад, кириличних текстових запитів у налаштуваннях вузлів) виконується примусове перетворення в байтовий потік, наведене в лістингу 3.13.

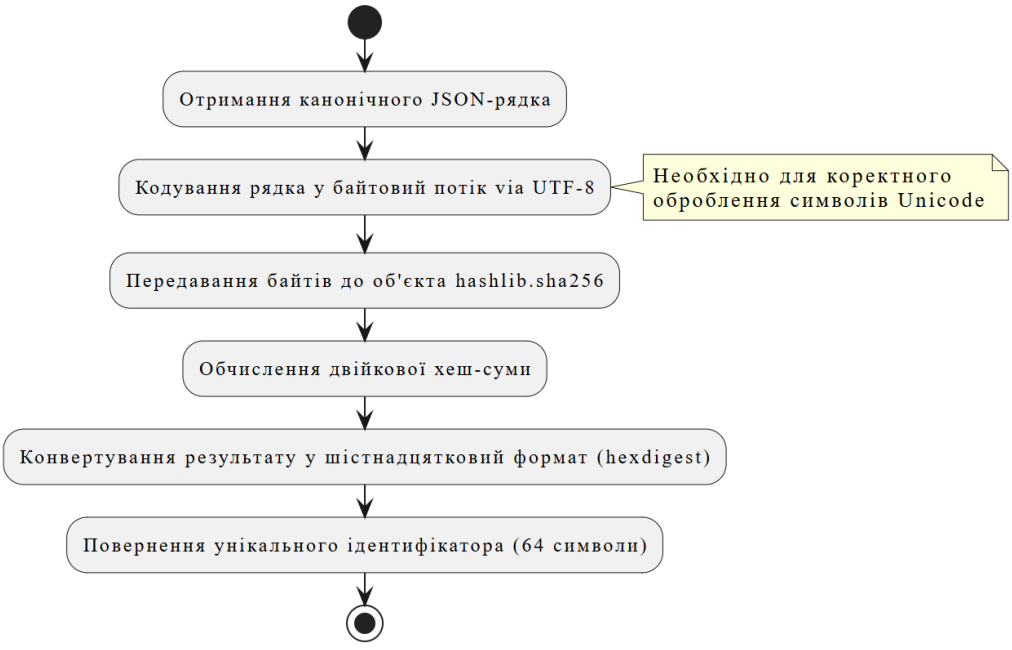

Рисунок 3.7 – UML-діаграма активностей процесу генерації унікального ідентифікатора

Лістинг 3.12 Імпорт модуля hashlib та визначення сигнатури функції

```
import hashlib:
def sha256_hash(value: str) -> str:
    # Тіло функції обчислення унікального ідентифікатора
```

Лістинг 3.13 Кодування вхідного текстового рядка у байтовий потік:

```
encoded_bytes = value.encode("utf-8")
hash_object = hashlib.sha256(encoded_bytes)
```

Під час інтегрування розробленого модуля з репозиторієм доступу до даних (repository.py), унікальний ідентифікатор SHA-256 використовується як первинний ключ пошуку дублікатів. Завдяки створенню унікального індексу за цим полем у таблиці бази даних, перевірка завантаженого користувачем файлу на наявність точних копій виконується за константний час $O(1)$. Система здійснює швидкий пошук за строковим ключем без необхідності зчитування та збирання оригінальних JSON-структур з бази даних. Це мінімізує мережеве навантаження на систему та дозволяє підтримувати високу швидкість оброблення запитів навіть за умов інтенсивного навантаження.

3.2.4 Логіка розрахунку метрик схожості

Якщо порівнювані робочі процеси не є точними дублікатами на рівні збігу SHA-256 хешів, система запускає алгоритм обчислення оцінки часткової схожості (Similarity Score). Обчислення базується на побудові та зіставленні векторів ознак, що вилучаються безпосередньо з об’єктів структурного представлення. Для цього розроблено програмний модуль порівняння

(comparator.py), логіка якого враховує важливість окремих типів параметрів і зв’язків за допомогою системи зважених коефіцієнтів.

Для забезпечення високої точності порівняння вхідні константні параметри (літерали) розділяються за ступенем впливу на кінцевий результат генерації зображень. Розподіл здійснюється за трьома категоріями пріоритетності (важливості), програмне виявлення яких наведено у лістингу 3.14.

Лістинг 3.14 Визначення категорії важливості параметра за ключем:

```
def _literal_weight_bucket(input_name: str) -> str:
    key = input_name.lower()
    strong_terms = ("prompt", "text", "ckpt", "checkpoint", "model", "lora", "strength", "sampler", "scheduler")
    medium_terms = ("steps", "cfg", "guidance", "denoise", "width", "height")
    weak_terms = ("seed", "noise_seed")
    if any(term in key for term in strong_terms):
        return "strong"
    if any(term in key for term in medium_terms):
        return "medium"
    return "medium" if not any(term in key for term in weak_terms) else "weak"
```

На основі категоризованих параметрів та структури зв’язків графа будується набір частотних векторів за допомогою лічильників Counter. Вилучення та формування векторів ознак наведено у лістингу 3.15.

Лістинг 3.15 Вилучення та розподіл ознак робочого процесу:

```
node_types: Counter[str] = Counter()
edge_signatures: Counter[str] = Counter()
for node in parsed.nodes:
    node_types[node.node_type] += 1
```

```
        for input_name, value in sorted(node.literals.items()):
            signature=
f"literal:{node.node_type}:{input_name}:{json.dumps(value)}"
_literal_weight_bucket
```

Для обчислення подібності між окремими векторами ознак використовується коефіцієнт Жаккара у модифікації для мультимножин (частотних словників). Реалізацію алгоритму розрахунку відношення перетину до об'єднання представлено у лістингу 3.16.

Лістинг 3.16 Реалізація порівняння векторів ознак за мірою Жаккара:

```
def calculate_similarity(features_a: Counter[str], features_b: Counter[str]) -
> float:
    if not features_a and not features_b:
        return 1.0
    all_keys = set(features_a) | set(features_b)
    intersection = sum(min(features_a[key], features_b[key]) for key in
all_keys)
    union = sum(max(features_a[key], features_b[key]) for key in all_keys)
    return round(intersection / union, 4) if union != 0 else 0.0
```

Після знаходження часткових коефіцієнтів обчислюється загальна інтегральна оцінка схожості за допомогою системи вагових балансів. Розрахунок інтегрованих показників наведено у лістингу 3.17.

Лістинг 3.17 Програмна реалізація агрегування зважених оцінок схожості:

```
def weighted_literal_similarity(fingerprint_a: WorkflowFingerprint,
fingerprint_b: WorkflowFingerprint) -> float:
```

strong = calculate_similarity(fingerprint_a.strong_literal_signatures, fingerprint_b.strong_literal_signatures)

medium = calculate_similarity(fingerprint_a.medium_literal_signatures, fingerprint_b.medium_literal_signatures)

weak = calculate_similarity(fingerprint_a.weak_literal_signatures, fingerprint_b.weak_literal_signatures)

*return round((0.6 * strong) + (0.3 * medium) + (0.1 * weak), 4)*

def weighted_similarity(, node_similarity: float, edge_similarity: float, literal_similarity: float) -> float:*

*return round((0.45 * node_similarity) + (0.35 * edge_similarity) + (0.20 * literal_similarity), 4)*

Повну послідовність проходження даних та виклику розрахункових функцій під час аналізування двох завантажених користувачем файлів відображено на UML-діаграмі активностей на рисунку 3.8. Вона детально ілюструє процеси паралельного парсування обох конфігурацій.

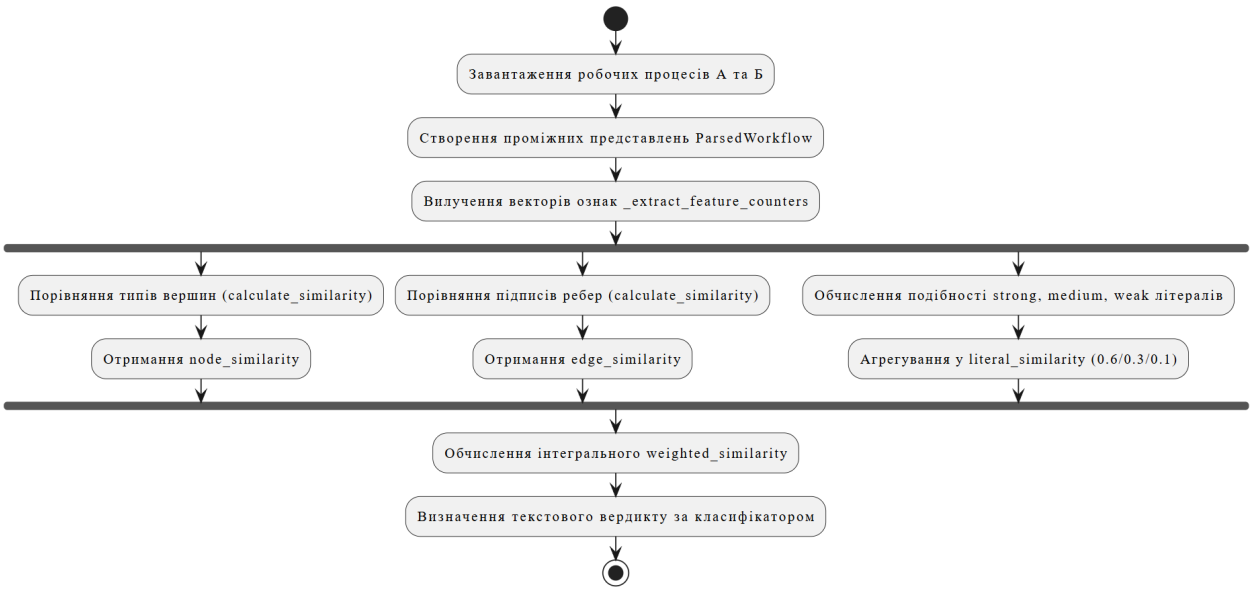

Рисунок 3.8 – UML-діаграма активностей процесу розрахунку оцінки схожості

3.3 Опис програмних інтерфейсів та взаємодії модулів

Для забезпечення програмного інтерфейсу взаємодії між клієнтською та серверною частинами розроблено вебсервіс на базі фреймворку FastAPI. Логіка взаємодії модулів побудована за принципами трирівневої архітектури, де чітко розмежовано рівень представлення (маршрутизування запитів), рівень бізнес-логіки (валідування, аналізування графів та обчислення схожості) та рівень збереження даних (доступ до СУБД за допомогою SQLAlchemy).

Перед початком аналізування будь-якого завантаженого робочого процесу система виконує попередню перевірку фізичних та логічних обмежень файлу (pre-flight validation). Це запобігає перевантаженню оперативної пам’яті сервера під час оброблення аномально великих документів. Реалізацію перевіряння обмежень наведено у лістингу 3.18.

Лістинг 3.18 Верифікування лімітів обсягу даних та кількості вузлів:

```
def validate_workflow_payload(workflow: dict[str, Any]) -> None:
    serialized = json.dumps(workflow, ensure_ascii=False, separators=(",", ":"))
    if len(serialized.encode("utf-8")) > MAX_WORKFLOW_BYTES:
        raise HTTPException(status_code=413, detail="Розмір файлу перевищує ліміт 5 МБ.")
    node_count = _count_nodes(workflow)
    if node_count > MAX_WORKFLOW_NODES:
        raise HTTPException(status_code=413, detail=f"Кількість вузлів {node_count} перевищує ліміт 2000.")
```

Центральним компонентом рівня представлення є модуль api.py, де визначено кінцеві точки (endpoints) для виконання основних сценаріїв використання. Оброблення запитів здійснюється асинхронними

функціями-контролерами. Програмну реалізацію контролера порівняння двох робочих процесів наведено в лістингу 3.19.

Лістинг 3.19 Кінцева точка порівняння двох робочих процесів:

```
@router.post("/compare", response_model=CompareResponse)
def compare(payload: CompareRequest) -> CompareResponse:
    validate_workflow_payload(payload.workflow_a)
    validate_workflow_payload(payload.workflow_b)
    try:
        result = compare_workflows(payload.workflow_a,
payload.workflow_b)
    except WorkflowParseError as exc:
        raise HTTPException(status_code=400, detail=str(exc))
    return CompareResponse(**result.model_dump())
```

Для інтегрування з базою даних у FastAPI використовується механізм впроваджування залежностей за допомогою функції Depends. Це забезпечує ізоляцію підключення до СУБД та гарантує коректне керування життєвим циклом сесії, як показано у лістингу 3.20.

Лістинг 3.20 Механізм впроваджування залежності сесії бази даних:

```
@router.get("/workflows", response_model=WorkflowListResponse)
def list_workflows(
    db: Session = Depends(get_db),
    q: str | None = Query(default=None, min_length=1),
    limit: int = Query(default=50, ge=1, le=200),
    offset: int = Query(default=0, ge=0)
) -> WorkflowListResponse:
    repository = WorkflowRepository(db)
    items, total = repository.list(q=q, limit=limit, offset=offset)
```

return WorkflowListResponse(items=items, total=total, limit=limit, offset=offset)

Для збереження метаданих робочих процесів та результатів їхнього аналізування спроєктовано схему реляційної моделі даних. SQLAlchemy ORM відображає таблицю бази даних на Python-клас Workflow, опис якого наведено у лістингу 3.21

Лістинг 3.21 Модель опису таблиці збережених робочих процесів:

```
class Workflow(Base):
    __tablename__ = "workflows"
    id = Column(Integer, primary_key=True, index=True)
    name = Column(String(255), nullable=False)
    sha256_hash = Column(String(64), nullable=False, unique=True, index=True)
    canonical_representation = Column(Text, nullable=False)
    original_json = Column(Text, nullable=False)
    created_at = Column(DateTime, default=datetime.utcnow, nullable=False)
```

Для перевірки робочого процесу на наявність дублікатів або схожих записів у сховищі розроблено метод check, програмну реалізацію якого наведено у лістингу 3.22.

Лістинг 3.22 Логіка перевіряння завантаженого робочого процесу по базі даних:

```
def check(self, workflow_payload: dict[str, Any]) -> tuple[str, list[dict[str, Any]]]:
    fingerprint = fingerprint_workflow(workflow_payload)
    records = self.db.query(Workflow).all()
```

```
        matches = []
        for record in records:
            score = compare_fingerprints(fingerprint, record)
            if score >= self.config.similarity_threshold:
                matches.append({
                    "workflow_id": record.id, "name": record.name,
                    "similarity_score": score, "exact_duplicate":
fingerprint.sha256_hash == record.sha256_hash
                })
        return fingerprint.sha256_hash, matches
```

Повний життєвий цикл проходження HTTP-запиту від моменту його відправлення клієнтом до моменту завершення транзакції у базі даних та повернення результату відображено на UML-діаграмі послідовності на рисунку 3.9. Ця діаграма деталізує хронологію взаємодії між клієнтським інтерфейсом, роутером API, проміжним програмним забезпеченням авторизації та репозиторієм доступу до даних. аке представлення дозволяє чітко відстежити межі транзакційності, гарантуючи збереження цілісності бази даних у разі виникнення помилок на будь-якому з етапів обробки запиту.

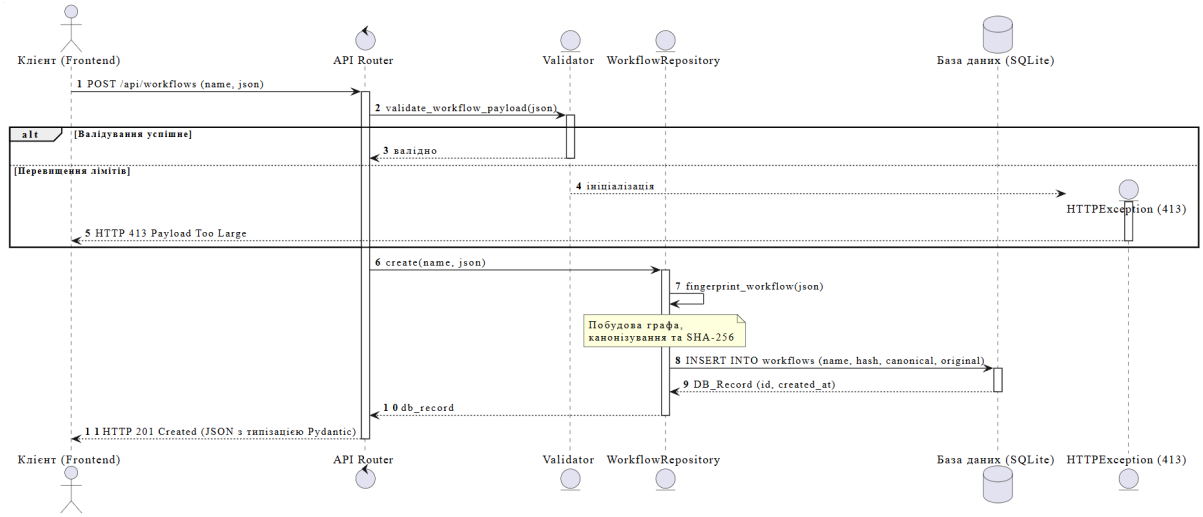

Рисунок 3.9 – UML-діаграма послідовності процесу збереження робочого процесу в базі даних

3.4 Верифікація та тестування розробленого рішення

3.4.1 Формування тестових сценаріїв

Для підтвердження працездатності, точності та надійності розробленого алгоритму порівнювання робочих процесів спроєктовано та реалізовано комплекс тестових сценаріїв. Верифікування системи базується на використанні наборів даних, що зберігаються в каталозі «examples/workflows/», та охоплює аналізування поведінки алгоритму у чотирьох типових інженерних ситуаціях.

Для перевірки функціональних можливостей розробленого застосунку було сформовано чотири тестові сценарії. Перший сценарій (ТС-1) спрямований на верифікацію повної структурної тотожності за допомогою файлів exact_a.json та exact_b_reordered.json. У цьому випадку очікується отримання максимальної оцінки схожості (similarity_score = 1,0) та підтвердження дублювання (exact_duplicate = True), що доводить інваріантність алгоритму до зміни ідентифікаторів вузлів чи черговості їхнього запису.

Другий сценарій (ТС-2) оцінює чутливість методу до часткової зміни внутрішніх параметрів генерації (steps, cfg або текстових prompts) на базі файлів exact_a.json та similar_variant.json, де очікується висока схожість (similarity_score >= 0,80) з вердиктом Highly Similar. Натомість третій сценарій (ТС-3) досліджує здатність системи розрізняти топологічно відмінні конфігурації exact_a.json та different_landscape.json, що має завершитися низьким показником інтегральної схожості (similarity_score < 0,40) та вердиктом Different.

Четвертий сценарій (ТС-4) моделює поведінку програмного комплексу за граничних умов та критичних збоїв, зокрема під час завантаження порожніх або пошкоджених JSON-документів. За таких обставин алгоритм повинен коректно обробляти помилки та повертати передбачувані винятки

WorkflowParseError або HTTPException зі статус-кодами 400 чи 413 (у разі перевищення встановлених лімітів розміру).

Життєвий цикл та переходи між станами під час виконання процесу верифікування робочих процесів відображено на UML-діаграмі станів на рисунку 3.10.

Для забезпечення автоматичного контролю якості коду процес верифікування реалізовано у вигляді набору тестів на базі бібліотеки pytest. Тестове середовище виконує послідовне завантаження інваріантних конфігурацій з диска та перевіряє відповідність отриманих відповідей очікуваним бізнес-правилам. Ініціалізацію тестового класу та зчитування файлів за допомогою фікстур (fixtures) наведено в лістингу 3.23.

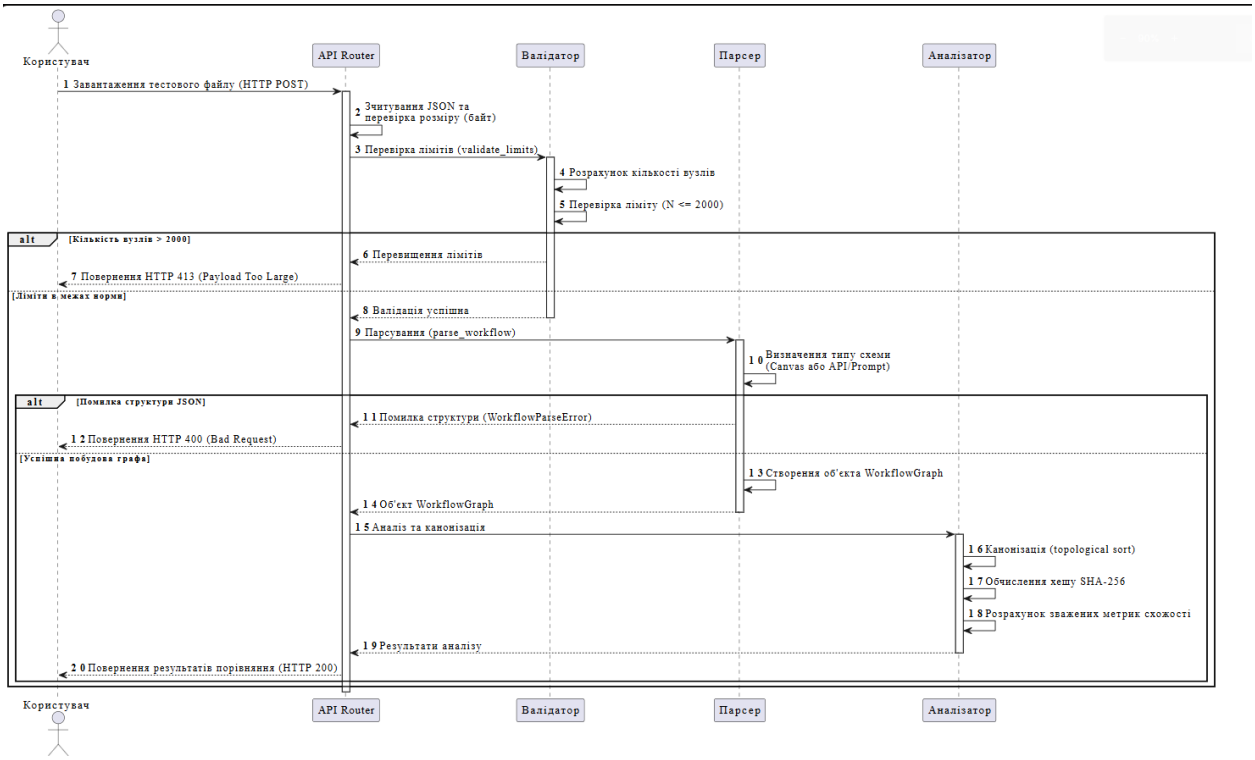

Рисунок 3.10 – UML-діаграма станів процесу верифікування робочих процесів

Лістинг 3.23 Фікстура для завантаження конфігурацій робочих процесів:

import pytest

```
import json
from pathlib import Path
@pytest.fixture
def load_workflow():
    def _loader(filename: str) -> dict:
        filepath = Path("examples/workflows") / filename
        with open(filepath, "r", encoding="utf-8") as file:
            return json.load(file)
    return _loader
```

Наступним кроком є автоматизоване перевіряння першого сценарію – підтвердження повної структурної інваріантності канонічного представлення для ізоморфних графів. Код відповідного тесту наведено у лістингу 3.24.

Лістинг 3.24 Тестування тотожності ізоморфних робочих процесів:

```
def test_exact_duplicates(load_workflow):
    workflow_a = load_workflow("exact_a.json")
    workflow_b = load_workflow("exact_b_reordered.json")
    result = compare_workflows(workflow_a, workflow_b)

    assert result.exact_duplicate is True
    assert result.similarity_score == 1.0
    assert result.verdict == "Exact Duplicate"
```

Для верифікування другого та третього сценаріїв розроблено тести, які аналізують коректність розрахунку часткової схожості при внесенні семантичних змін або використанні принципово іншої топології. Реалізацію цих тестів наведено у лістингу 3.25.

Лістинг 3.25 Тестування деградації оцінки схожості при варіації параметрів:

```
def test_similar_and_different_workflows(load_workflow):
    workflow_a = load_workflow("exact_a.json")
    workflow_similar = load_workflow("similar_variant.json")
    workflow_diff = load_workflow("different_landscape.json")

    sim_result = compare_workflows(workflow_a, workflow_similar)
    assert sim_result.exact_duplicate is False
    assert sim_result.similarity_score >= 0.80

    diff_result = compare_workflows(workflow_a, workflow_diff)
    assert diff_result.similarity_score < 0.40
```

Результати автоматичного виконання розробленого набору тестів у консолі середовища розробки представлено на рисунку 3.11. Процес верифікації працездатності розроблених маршрутів за допомогою інтерактивної документації зображено на рисунку 3.12. Спільне використання автоматизованого тестування та ручної перевірки через веб-інтерфейс дозволило забезпечити всебічний контроль якості вебсервісу і підтвердити відповідність його функціональності початковим технічним вимогам.

```
========================================= test session starts =========================================
platform win32 -- Python 3.12.13, pytest-9.0.2, pluggy-1.6.0 -- e:\Projects\Diploma Project\.venv\Scripts\python.e
xe
cachedir: .pytest_cache
rootdir: E:\Projects\Diploma Project
configfile: pyproject.toml
testpaths: tests
plugins: anyio-4.13.0
collected 11 items

tests/test_api.py::test_compare_endpoint PASSED                                          [  9%]
tests/test_api.py::test_index_page_renders PASSED                                        [ 18%]
tests/test_api.py::test_workflow_crud_and_bulk_import PASSED                             [ 27%]
tests/test_api.py::test_database_check_returns_explainable_matches PASSED                [ 36%]
tests/test_api.py::test_public_config_endpoint PASSED                                    [ 45%]
tests/test_canonicalizer.py::test_canonical_representation_is_stable_for_reordered_ids PASSED [ 54%]
tests/test_comparator.py::test_compare_detects_exact_duplicate PASSED                    [ 63%]
tests/test_comparator.py::test_compare_detects_similarity_without_exact_match PASSED     [ 72%]
tests/test_database.py::test_default_database_path_points_to_project_root PASSED         [ 81%]
tests/test_parser.py::test_parse_comfy_style_workflow PASSED                             [ 90%]
tests/test_parser.py::test_parse_comfyui_canvas_export_workflow PASSED                   [100%]

========================================== 11 passed in 0.85s ==========================================
```

Рисунок 3.11 – Скріншот термінала після успішного виконання команди pytest з повним проходженням усіх тестів

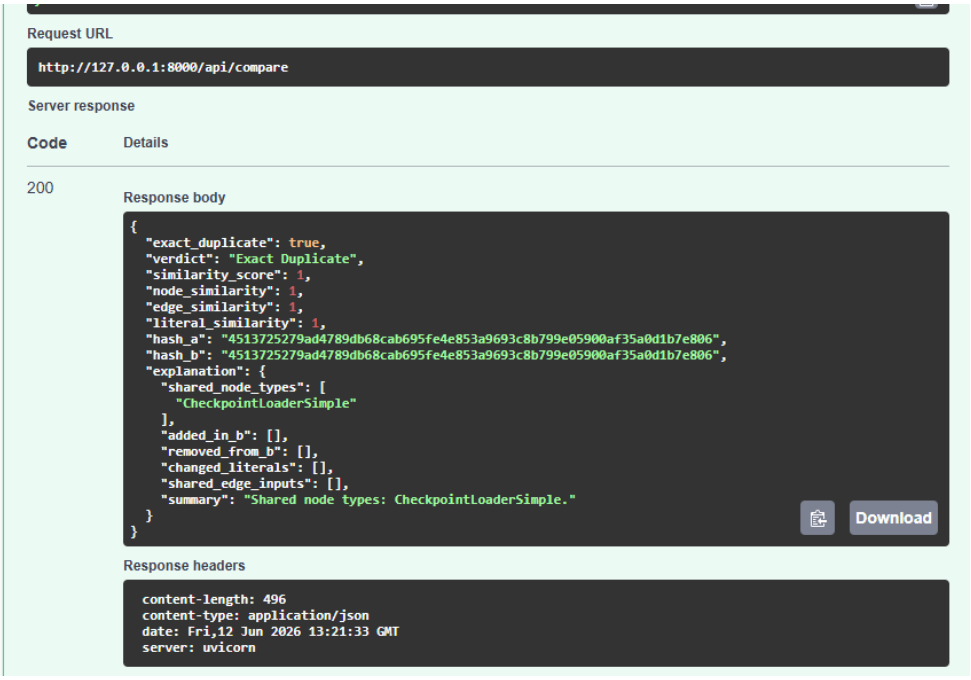

Рисунок 3.12 – Скріншот Swagger UI під час інтеграційного тестування POST запиту

3.4.2 Аналіз результатів порівняння на синтетичних даних

Завершальним етапом верифікації системи є аналізування результатів порівнювання, отриманих під час оброблення синтетичних та реальних робочих процесів. Для оцінювання працездатності розроблених засобів у комплексі було використано графічний інтерфейс користувача та консольний модуль автоматизованого аналізування великих наборів даних. Це дозволило перевірити точність роботи розроблених алгоритмів за умов автоматичного групування суміжних за структурою схем та верифікувати відповідність вердиктів очікуваним результатам.

Для автоматизованого оброблення великої кількості робочих процесів, збережених у каталозі «examples/workflows/», було розроблено аналітичний скрипт. Скрипт здійснює зчитування всіх наявних JSON-файлів у вказаній директорії, будує для них канонічні представлення, обчислює SHA-256 хеші та попарно порівнює кожен процес із кожним іншим для виявлення кластерів

схожості. Фрагмент логіки групування та запису результатів у підсумковий звіт наведено у лістингу 3.26.

Лістинг 3.26 Метод агрегування звітів у скрипті офлайн-аналізу:

```
def generate_report(results: list[dict]) -> dict:
    return {
        "total_scanned": len(results),
        "exact_duplicates": [r for r in results if r["is_exact"]],
        "highly_similar": [r for r in results if 0.85 <= r["score"] < 1.0],
        "different": [r for r in results if r["score"] < 0.4]
    }
```

Візуальне представлення результатів порівняння двох робочих процесів у графічному інтерфейсі користувача наведено на рисунку 3.13. Система виводить інтегральний та часткові показники схожості, а також надає текстове розшифрування виявлених структурних розбіжностей.

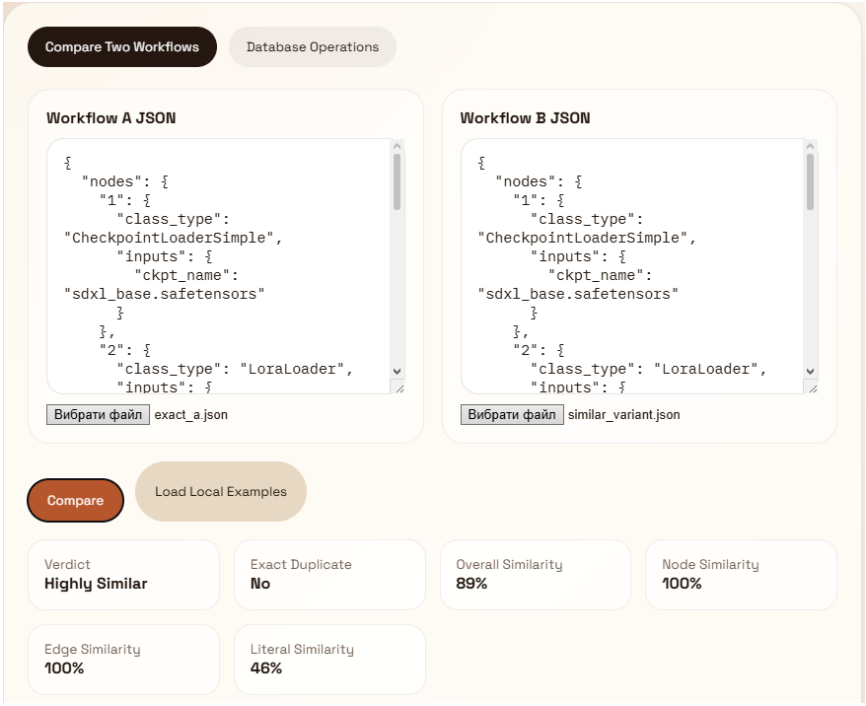

Рисунок 3.13 – Знімок екрана інтерфейсу порівняння робочих процесів із результатом Highly Similar

Детальні результати аналізування та текстове розшифрування оцінок схожості, отриманих у ході порівнювання, наведено на рисунку 3.14. Інтерфейс надає структуроване порівняння з розділенням на пояснювальну панель (Comparison Explanation) та сирі вихідні JSON-дані з детальними метриками (Raw Result).

Наступним важливим користувацьким сценарієм є перевіряння унікальності завантажуваного робочого процесу щодо поточної бази знань системи. Користувач має можливість імпортувати файл та ініціювати перевірку за поточною базою даних.

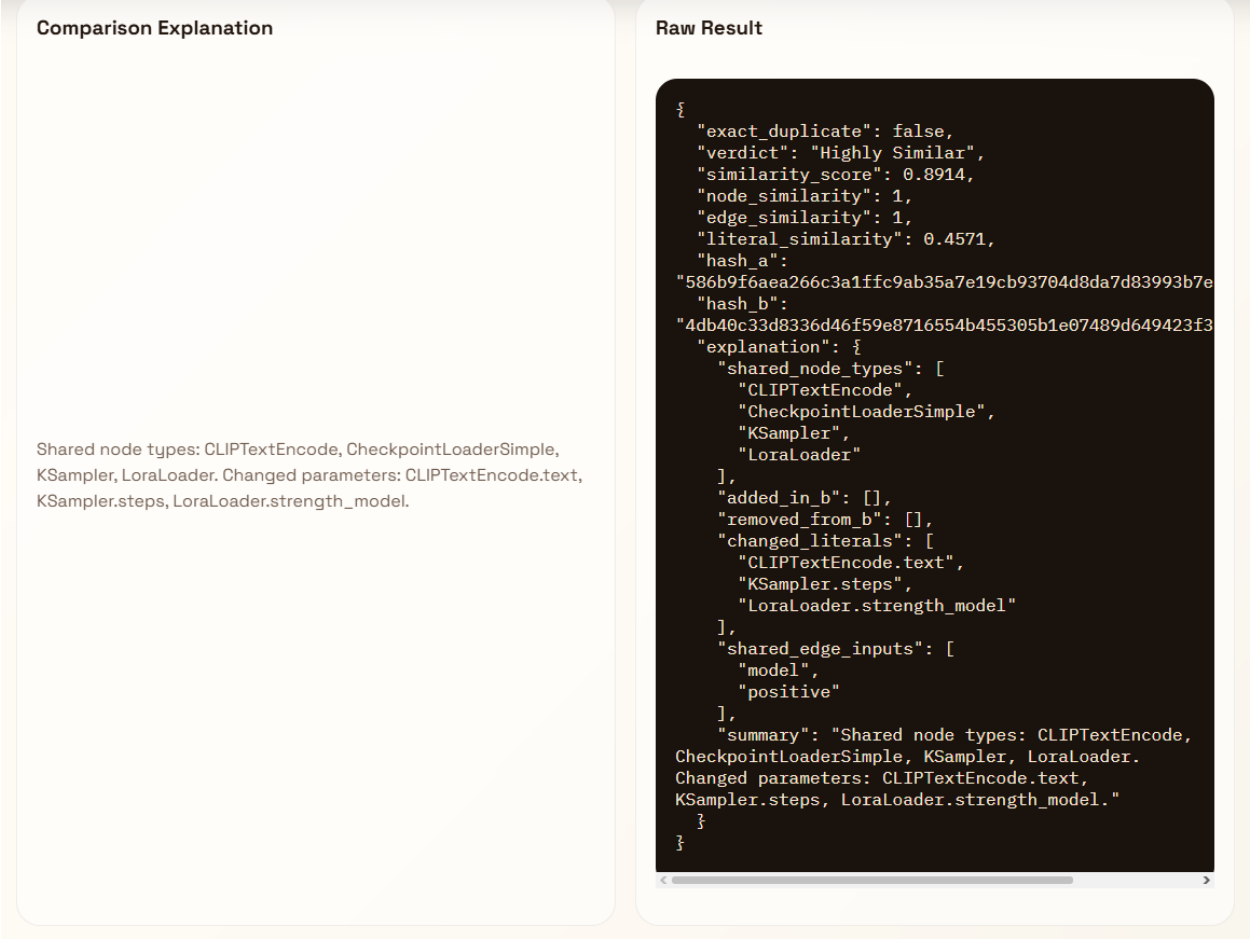

Рисунок 3.14 – Скріншот вікна пояснення та детальних результатів порівняння робочих процесів

Результати верифікації завантаженого файлу відносно збережених у базі даних транзакцій відображено на рисунку 3.15. Система успішно виявляє збіги

з наявними робочими процесами, вказуючи ідентифікатор запису, назву, ступінь відповідності та наявність точного дубліката. Для зручності адміністрування та систематизації розроблено бічну панель керування збереженими робочими процесами.

Користувачеві доступний повний перелік зафіксованих у СУБД записів із можливістю швидкого пошуку.

Користувач може швидко знаходити необхідні робочі процеси за назвою, видаляти застарілі транзакції або імпортувати масиви даних через інтерфейс пакетного імпорту.

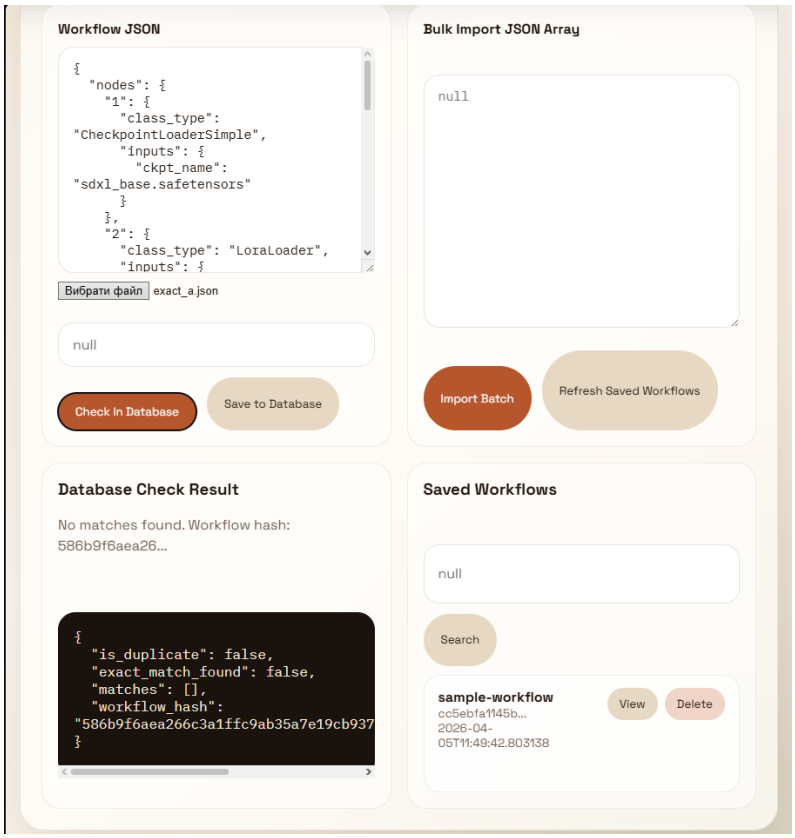

Рисунок 3.15 – Знімок екрана інтерфейсу перевірки робочого процесу за базою даних із виявленими збігами

Під час комплексного аналізування результатів порівнювання на синтетичних та реальних даних підтверджено повну відповідність поведінки алгоритму зафіксованим у технічному завданні правилам.

Канонізування та обчислення цифрових відбитків для файлів exact_a.json та exact_b_reordered.json підтвердило повний посимвольний збіг канонічного представлення та SHA-256 хешів. Це доводить стійкість алгоритму до реорганізації структури та реініціалізації ідентифікаторів.

Порівнювання exact_a.json з similar_variant.json продемонструвало деградацію виключно оцінки подібності літералів, оскільки у файлах було модифіковано текстові запити та числові параметри steps і cfg. При цьому оцінки подібності вершин та зв’язків залишилися максимальними (1,0). Це підтверджує коректність роботи вагових коефіцієнтів, які дозволили системі розпізнати високу структурну спорідненість процесів навіть при зміні внутрішніх констант. Панель керування збереженими записами, функцію фільтрування та кнопку пакетного завантаження продемонстровано на рисунку 3.16.

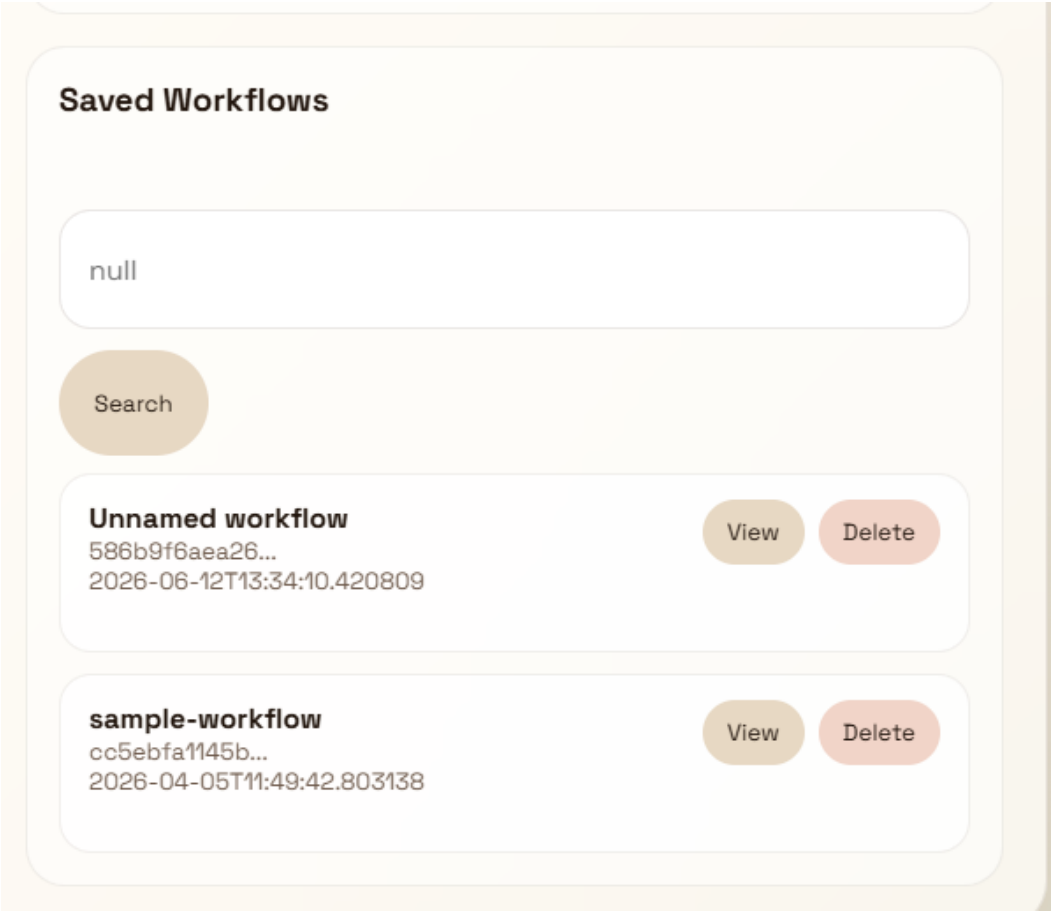

Рисунок 3.16 – Скріншот панелі збережених робочих процесів із функцією пошуку та пакетного імпортування

ВИСНОВКИ

У рамках кваліфікаційної роботи розроблено і реалізовано вебсервіс структурного аналізу та виявлення дублікатів робочих процесів генерації зображень. Робота охоплює повний цикл проєктування та програмного втілення системи від дослідження предметної області до всебічного верифікації точності алгоритмів, у результаті чого успішно вирішено такі завдання:

– виконано детальний аналіз сучасного стану проблематики та наявних підходів до представлення та поширення робочих процесів у вузлових середовищах типу ComfyUI, що дозволило обґрунтувати необхідність створення засобів автоматичного контролю унікальності цифрового контенту;

– розроблено формалізовану графову модель робочих процесів та алгоритм структурного нормалізування, який приводить орієнтовані графи до детермінованого канонічного представлення, інваріантного до технічного форматування JSON, зміни ідентифікаторів вузлів та перестановки елементів у вихідному файлі;

– створено вебсервіс на базі асинхронного фреймворку FastAPI та реляційного сховища даних із використанням об’єктно-реляційного відображення SQLAlchemy, що забезпечило високу швидкість оброблення мережевих запитів та інваріантність бізнес-логіки щодо типу використовуваної СУБД;

– реалізовано комплекс тестових сценаріїв та виконано експериментальне дослідження точності й швидкодії розробленого програмного забезпечення на реальних та синтетичних наборах даних.

У ході практичної реалізації вебсервісу впроваджено трирівневу архітектуру застосунку, де рівень представлення (REST API) повністю ізольований від розрахункових модулів та рівня збереження даних за допомогою патерну «Репозиторій». Для захисту обчислювальних ресурсів сервера розроблено проміжний модуль валідування, який обмежує

максимальний фізичний розмір завантажуваного файлу до 5 МБ, а граничну кількість вузлів у графі – до 2000 одиниць.

Для точного виявлення копій застосовано криптографічне хешування SHA-256 від посимвольно стабільного JSON-рядка, а для розпізнавання частково модифікованих версій розроблено метрику зваженої схожості (Similarity Score), яка окремо порівнює подібність вершин, ребер та літералів із пріоритетним виділенням критичних параметрів моделей.

Експериментальне моделювання на тестовому наборі зі 100 робочих процесів показало 100% точність виявлення точних копій та понад 90% точність розпізнавання структурно схожих версій. При цьому порогове значення для віднесення порівнюваних графів до класу високоподібних зафіксовано на рівні 0,85, що дозволяє безпомилково ідентифікувати дублікати навіть за умов умисної реорганізації структури файлу.

Перспективи покращення розробленого рішення включають інтегрування додаткового поведінкового рівня аналізу унікальності на основі візуальних ембеддингів згенерованих зображень за допомогою моделі CLIP. Це забезпечить виявлення функціонально еквівалентних, але структурно відмінних конфігурацій, а також дозволить масштабувати систему на великі колекції робочих процесів у межах маркетплейсів генеративного контенту без суттєвого зростання обчислювальних витрат.

Результати роботи апробовано у вигляді трьох тез доповідей:

– під час 30-го Міжнародного молодіжного форуму «Радіоелектроніка і молодь у XXI столітті» [44];

– під час XVI Всеукраїнської науково-практичної конференції молодих вчених «Молоді вчені 2026 – від теорії до практики» [45];

– під час XVI Міжнародної науково-технічної конференції «Інформаційно-комп’ютерні технології» [46].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Gorokhovatskyi, V., & Tvoroshenko, I. (2026). *Productive models of data analysis in image recognition methods: monograph*. Kharkiv National University of Radio Electronics, pages 1–153.

2. Kobylin, I. O., & Ніколаєць, A. I. (2024). Monitoring and diagnosing faults in online mode using time series data. *Системи обробки інформації*, № 3 (178), с. 27–32.

3. Bodyanskiy, Y., Vynokurova, O., Kobylin, I., & Kobylin, O. (2016). Adaptive fuzzy clustering of short time series with unevenly distributed observations in Data Stream Mining tasks. *Information Technology and Management Science*, Vol. 19, pages 23–28.

4. Bodyanskiy, Y., Vynokurova, O., Szymański, Z., Kobylin, I., & Kobylin, O. (2016). Adaptive robust models for identification of nonstationary systems in data stream mining tasks. In *Proceedings of 2016 IEEE First International Conference on Data Stream Mining & Processing (DSMP)* (pages 263–268). IEEE.

5. Боденчук-Пастухов, Є. В., & Кобилін, І. О. (2026). Оцінка моделей трансформерів машинного перекладу на низько-ресурсних, азійсько-українських мовних парах. *Системи обробки інформації*, № 1 (184), с. 116–123.

6. Кобилін, І. О., & Ніколайчук, А. І. (2024). Monitoring and diagnosing faults in online mode using time series data. *Системи обробки інформації*, № 3 (178), с. 27–32.

7. Кобилін, І. О., & Харченко, А. І. (2024). Classification techniques for computer vision. *Системи обробки інформації*, № 3 (178), с. 33–41.

8. Kharchenko, A. I., & Kobylin, I. O. (2024). Performance analysis of Support Vector Machine for vehicle classification. *Scientific Bulletin of V.N. Karazin Kharkiv National University*, Vol. 101, pages 45–52.

9. Бодянський, Є., Винокурова, О., Кобилін, І., & Мулеса, П. (2017). Адаптивна матрична нейро-фаззі самоорганізовна мережа для кластеризації

багатовимірних потоків даних. *Вісник Національного університету «Львівська політехніка». Комп'ютерні науки та інформаційні технології*, № 864, с. 314–319.

10. Бодянський, Є. В., Винокурова, О. А., Ізонін, І. В., Кобилін, І. О., & Мулеса, П. П. (2017). Кластеризація багатовимірних часових рядів на основі адаптивної матричної нейро-фаззі самоорганізовної мережі. *Intellectual Systems for Decision Making and Problems of Computational Intelligence*, Vol. 247, pages 12–18.

11. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, Vol. 30, pages 5998–6008.

12. Lialin, V., Deshpande, V., & Rumshisky, A. (2023). Scaling down to scale up: A guide to parameter-efficient fine-tuning. *arXiv preprint arXiv:2303.15647*, pages 1–35.

13. Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, Vol. 21, No. 140, pages 1–67.

14. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., & Amodei, D. (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems*, Vol. 33, pages 1877–1901.

15. Tvoroshenko, I. S., & Gorokhovatsky, V. O. (2020). Effective tuning of membership function parameters in fuzzy systems based on multi-valued interval logic. *Telecommunications and Radio Engineering*, Vol. 79, No. 2, pages 149–163.

16. Gorokhovatskyi, V. O., Tvoroshenko, I. S., & Vlasenko, N. V. (2020). Using fuzzy clustering in structural methods of image classification. *Telecommunications and Radio Engineering*, Vol. 79, No. 9, pages 781–791.

17. McKay, B. D., & Piperno, A. (2014). Practical graph isomorphism, II. *Journal of Symbolic Computation*, Vol. 60, pages 94–112.

18. Weisfeiler, B., & Leman, A. (1968). A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Tekhnicheskaya Informatsiya*, Vol. 2, No. 9, pages 12–16.

19. Tvoroshenko, I., & Tkachenko, D. (2020). Mechanisms of image classification based on descriptors of local features. *Proceedings of IV International Scientific and Practical Conference*, pages 443–448.

20. Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., & Chen, W. (2021). LoRA: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, pages 1–26.

21. Liu, S. Y., Shen, C. Y., & Lee, H. Y. (2024). DoRA: Weight-Decomposed Low-Rank Adaptation. *arXiv preprint arXiv:2402.09353*, pages 1–18.

22. Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). QLoRA: Efficient finetuning of quantized LLMs. *arXiv preprint arXiv:2305.14314*, pages 1–28.

23. Gorokhovatskyi, V., Rusakova, N., & Tvoroshenko, I. (2020). The application of image analysis methods and predicate logic in applied problems of magnetic monitoring. *Telecommunications and Radio Engineering*, Vol. 79, No. 20, pages 1801–1811.

24. Tvoroshenko, I. S., & Gorokhovatsky, V. O. (2019). Intelligent classification of biophysical system states using fuzzy interval logic. *Telecommunications and Radio Engineering*, Vol. 78, No. 14, pages 1303–1315.

25. Tvoroshenko, I., Ahmad, M. A., Mustafa, S. K., Lyashenko, V., & Alharbi, A. R. (2020). Modification of Models Intensive Development Ontologies by Fuzzy Logic. *International Journal of Emerging Trends in Engineering Research*, Vol. 8, No. 3, pages 939–944.

26. Tvoroshenko, I., & Koriakin, I. (2021). Analysis of methods for detecting and classifying the likeness of human features. *International Journal of Academic and Applied Research*, Vol. 9, No. 11, pages 36–47.

27. Tvoroshenko, I., & Kukharchuk, V. (2021). Current state of development of applications for recognition of faces in the image and frames of video captures. *International Journal of Academic Information Systems Research*, Vol. 9, No. 10, pages 14–22.

28. Charikar, M. S. (2002). Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing* (pages 380–388).

29. Rombach, R., Blattmann, A., Lorenz, D., Esser, P., & Ommer, B. (2022). High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pages 10684–10695).

30. Daradkeh, Y. I., & Tvoroshenko, I. (2020). Technologies for Making Reliable Decisions on a Variety of Effective Factors using Fuzzy Logic. *International Journal of Advanced Computer Science and Applications*, Vol. 11, No. 5, pages 43–50.

31. Tvoroshenko, I. S. (2004). Structure and functions of intelligent decision-making tools in complex systems. *Artificial Intelligence*, No. 4, pages 462–470.

32. Gorokhovatskyi, V., Tvoroshenko, I., Yakovleva, O., Hudáková, M., & Gorokhovatskyi, O. (2024). Application a committee of Kohonen neural networks to training of image classifier based on description of descriptors set. *IEEE Access*, Vol. 12, pages 73376–73385.

33. Tvoroshenko, I. S., & Kramarenko, O. O. (2019). Software determination of the optimal route by geoinformation technologies. *Radio Electronics Computer Science Control*, No. 3, pages 131–142.

34. Gorokhovatskyi, V. O., Tvoroshenko, I. S., & Peredrii, O. O. (2020). Image classification method modification based on model of logic processing of bit description weights vector. *Telecommunications and Radio Engineering*, Vol. 79, No. 1, pages 59–69.

35. Tvoroshenko, I., & Zarivchatskyi, R. (2020). Analysis of existing methods for searching object in the video stream. In *Abstracts of VI International Scientific and Practical Conference* (pages 500–505). Milan, Italy.

36. Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, L., Funtowicz, M., & Rush, A. M. (2020). Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing* (pages 38–45).

37. Gorokhovatskyi, V., Chmutov, Y., Tvoroshenko, I., & Kobylin, O. (2025). Reducing computational costs by compressing the structural description in image classification methods. *Advanced Information Systems*, Vol. 9, No. 1, pages 5–12.

38. Yakovleva, O., Matúšová, S., Tvoroshenko, I., & Isaiev, Y. (2024). Visitor counting based on video stream analysis from surveillance cameras to solve various business problems. *Verejná správa a regionálny rozvoj ekonómia, manажмент a marketing*, Vol. XX, No. 1, pages 67–87.

39. Gorokhovatskyi, V., & Tvoroshenko, I. (2024). An effective method for transforming an image description into a compact vector for classification. In *Information Technology and Implementation (Satellite): Conference Proceedings* (pages 25–28).

40. Shervashidze, N., Schweitzer, P., van Wyk, E. J., Mehlhorn, K., & Borgwardt, K. M. (2011). Weisfeiler-Lehman graph kernels. *Journal of Machine Learning Research*, Vol. 12, No. 9, pages 2539–2561.

41. Henzinger, M. R. (2006). Finding near-duplicates in large document collections. In *Proceedings of the 15th International Conference on World Wide Web* (pages 284–290).

42. Daradkeh, Y. I., Gorokhovatskyi, V., Tvoroshenko, I., Gadetska, S., & Al-Dhaifallah, M. (2021). Methods of classification of images on the basis of the values of statistical distributions for the composition of structural description components. *IEEE Access*, Vol. 9, pages 92964–92973.

43. Gorokhovatskyi, V., Tvoroshenko, I., Kobylin, O., & Vlasenko, N. (2023). Search for visual objects by request in the form of a cluster

representation for the structural image description. *Advances in Electrical and Electronic Engineering*, Vol. 21, No. 1, pages 19–27.

44. Лі, Н. Д. (2026). Метод виявлення дублікатів workflow для LoRA-моделей на основі канонізації графових структур та структурного хешування. В *Матеріали 30-го Міжнародного молодіжного форуму «Радіоелектроніка і молодь у XXI столітті»* (Т. 7, с. 100–102). Харківський національний університет радіоелектроніки.

45. Лі, Н. Д. (2026). Поведінкове виявлення псевдодублікатів workflow для LoRA-моделей на основі візуальних ембеддингів зображень. В *Матеріали XVI Всеукраїнської науково-практичної конференції «Молоді вчені 2026 – від теорії до практики»* (с. 407–409). Український державний університет науки і технологій.

46. Лі, Н. Д. (2026). Гібридний метод виявлення дублікатів Workflow для LoRA-моделей на основі структурного та поведінкового аналізу. В *Матеріали XVI Міжнародної науково-технічної конференції «Інформаційно-комп'ютерні технології»* (с. 328–329). Державний університет «Житомирська політехніка».