Хмельницький національний університет
Факультет інформаційних технологій
Кафедра комп’ютерної інженерії та інформаційних систем

КВАЛІФІКАЦІЙНА РОБОТА

Розподілена система аналізу зображень для формування навчальних датасетів
Назва теми

Рівень вищої освіти *перший (бакалаврський)*

Галузь знань *12 «Інформаційні технології»*
Шифр, назва

Спеціальність *123 «Комп’ютерна інженерія»*
Шифр, назва

Освітня програма *«Комп’ютерна інженерія та програмування»*
Назва

Шифр КвРКІ 022083.22.03.70 ПЗ

Виконав здобувач IV курсу, група КІ2-22-3 ______ Володимир СИВОКОНЬ
Підпис Ініціали, прізвище

Керівник доктор філософії ______ Павло РЕГІДА
Науковий ступінь, учене звання Підпис Ініціали, прізвище

Нормоконтролер канд.фіз.-мат.наук, доц. ______ Тетяна КИСІЛЬ
Науковий ступінь, учене звання Підпис Ініціали, прізвище

До захисту допускаю:
завідувач кафедри КІІС ______ Ольга ПАВЛОВА
Підпис Ініціали, прізвище
«01» червня 2026 р.
дата

Хмельницький 2026

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій

Кафедра Комп'ютерної інженерії та інформаційних систем

Рівень вищої освіти перший (бакалаврський)

Галузь знань 12 Інформаційні технології

Спеціальність 123 Комп'ютерна інженерія

Освітня програма «Комп'ютерна інженерія та програмування»

ЗАТВЕРДЖУЮ
Завідувачка кафедри КІІС
Ольга ПАВЛОВА

" 10 " 01 2026 р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Сивоконю Володимиру Олександровичу

Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи) Розподілена система аналізу зображень для формування навчальних датасетів

Керівник проекту (роботи) Регіда Павло Геннадійович, доцент.

Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 20.01.2026 р. № 7

2. Термін подання здобувачем роботи на кафедру 01.06.2026 р.

3. Вихідні дані до роботи Завдання на кваліфікаційну роботу

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

Розподілена система аналізу зображень для формування навчальних датасетів

Проєктування та програмна реалізація розподіленої системи аналізу зображень

Експериментальне дослідження та оцінка ефективності розробленої системи

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

Загальна архітектура та життєвий цикл

Алгоритм роботи воркера та діаграма станів задачі

Структурна схема компонентів

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання «_10_»________01__________2026 р.

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів (розділів) дипломного проекту (роботи)	Термін виконання етапів проекту (роботи)	Примітка
1	Вибір напряму дослідження та узгодження тематики кваліфікаційної роботи з керівником	10.01.2026	виконано
2	Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження	01.02.2026	виконано
3	Робота над розділом 1 – аналіз розподілених систем та обґрунтування архітектури майстер-воркер	01.03.2026	виконано
4	Робота над розділом 2 – проєктування та реалізація розподіленої системи аналізу зображень	01.04.2026	виконано
5	Робота над розділом 3 – тестування системи та аналіз результатів розподіленої обробки	29.04.2026	виконано
6	Оформлення пояснювальної записки згідно вимог	24.05.2026	виконано
7	Попередній захист ВКР	25.05.2026	виконано
8	Захист ВКР на засіданні ЕК	Червень 2026 року	

Здобувач________________ ____Володимир СИВОКОНЬ___
Підпис Імя, ПРІЗВИЩЕ

Керівник кваліфікаційної роботи______________ ____Павло РЕГІДА____
Підпис Імя, ПРІЗВИЩЕ

№ рядка	формат	Позначення	Найменування	Кіл. листів	№ екз	Примітка
			Текстові документи			
1		КвРКІ 022083.22.03.70 ПЗ	Пояснювальна записка	61		
			Графічні матеріали			
2		КвРКІ 022083.22.03.70 Е8	Блок-схема алгоритму	1		
			робочого вузла			
3		КвРКІ 022083.22.03.70 Е8	Структурна схема	1		
			компонентів системи			
4		КвРКІ 022083.22.03.70 Е8	Загальна архітектура	1		
			та життєвий цикл задачі			
5		КвРКІ 022083.22.03.70 Е8	Лістинг коду модуля диспетчеризації	1		

Зм	Арк	№ докум	Підпис	Дата
Розробив		Сивоконь		01.06
Перевір.		Регіда		01.06
Н. контр.		Кисіль		01.06
Затв.		Павлова		01.06

КвРКІ 022083.22.03.70 ВП

Відомість проєкту

Літера	Аркуш	Аркушів
У	1	1

ХНУ, КІ2-22-3

АНОТАЦІЯ

Тема кваліфікаційної роботи: «Розподілена система аналізу зображень для формування навчальних датасетів».

Автор роботи: Володимир СИВОКОНЬ.

Керівник роботи: Павло РЕГІДА.

Пояснювальна записка: 61 с., 12 рис., 6 табл., дод., 50 джерел.

Графічна частина: 3 креслення.

РОЗПОДІЛЕНА СИСТЕМА, МАЙСТЕР-ВОРКЕР, НЕЙРОННА МЕРЕЖА, ІНФЕРЕНС, ДАТАСЕТ.

Кваліфікаційна робота присвячена проєктуванню і реалізації розподіленої системи автоматичного анотування зображень на основі архітектури майстер-воркер (master-worker). Ручне формування датасетів для задач комп’ютерного зору є трудомістким процесом, який погано масштабується – саме це і стало основою для постановки задачі.

Метою роботи є побудова системи, що дозволяє залучити кілька звичайних робочих ПК до паралельної обробки зображень без спеціалізованої серверної інфраструктури. Центральний вузол керує чергою задач, робочі вузли виконують інференс моделі Moondream2 і повертають текстові описи. Взаємодія між компонентами побудована на HTTP/REST з pull-механізмом отримання задач, що дозволяє розгортати воркери у довільній мережевій конфігурації.

У роботі проведено аналіз типів розподілених систем і платформ обробки задач, обґрунтовано вибір стеку FastAPI, SQLite і PyTorch, вирішено проблему атомарного призначення задач без явних блокувань. Реалізовано графічний інтерфейс для керування системою і два формати експорту результатів: HTML-звіт для перегляду і CSV для технічної обробки.

Підпис здобувача

30.05.2026

Дата

ЗМІСТ

Зм.	Арк.	№докум.	Підпис	Дата
Виконав		Волод. Сивоконь		01.06
Перевір.		Павло РЕГІДА		01.06
Н.контр.		Тетяна КИСІЛЬ		01.06
Затвер.		Ольга ПАВЛОВА		01.06

КвРКІ. 022083.22.03.70 ПЗ

Розподілена система аналізу зображень для формування навчальних датасетів.
Пояснювальна записка

Літера	Аркуш	Аркушів
у	2	61

ХНУ КІ2-22-3

ВСТУП

Формування навчальних датасетів для задач комп’ютерного зору залишається трудомістким процесом, що погано масштабується в умовах виробничого середовища. Більшість підходів до навчання моделей детектування об’єктів, класифікації дефектів і семантичної сегментації потребують великих обсягів анотованих зображень[6]. Зображення накопичуються у різних галузях щодня, проте без структурованих описів залишаються практично непридатними для задач автоматизованого аналізу[36, 37].

Автоматизувати анотування дозволяють зорово-мовні моделі, здатні генерувати текстові описи зображень без участі оператора[9, 23, 26]. Проблема полягає в тому, що запуск таких моделей на одному вузлі обмежує пропускну здатність системи. Це зумовлює необхідність розподіленої архітектури, де обчислювальне навантаження рознесено між кількома робочими вузлами під керуванням центрального сервера.

Реалізувати таку архітектуру можна різними способами, від важких промислових рішень на кшталт Celery або Ray до простих HTTP-сервісів з мінімумом залежностей[11,45]. Перші дають більше можливостей, але потребують складного налаштування і мають обмеження на платформі Windows, де запускається більшість офісних робочих місць. Другий підхід простіший у розгортанні і достатній для задачі паралельної обробки зображень у локальній мережі. Саме він і ліг в основу цієї роботи.

Технічну основу системи складають кілька принципових рішень. Атомарне призначення задач через конструкцію UPDATE...RETURNING у SQLite усуває проблему одночасного захоплення однієї задачі кількома воркерами без потреби у зовнішніх механізмах синхронізації[19, 20]. Pull-модель взаємодії дозволяє воркерам працювати за NAT і у мережах з обмеженим доступом, оскільки всі з’єднання ініціює сам воркер. Графічний інтерфейс об’єднує керування майстром і воркером в одному застосунку і дозволяє спостерігати за розподілом задач між вузлами в реальному часі.

Метою роботи є проєктування і реалізація розподіленої системи аналізу зображень для автоматизованого формування навчальних датасетів на основі архітектури майстер-воркер (master-worker) з використанням vision-language моделі Moondream2.

Для досягнення поставленої мети вирішувались такі завдання. Проведено аналіз типів розподілених систем та обґрунтовано вибір архітектури майстер-воркер для задачі паралельної обробки зображень. Здійснено порівняння відомих платформ розподіленої обробки задач з урахуванням обмежень середовища Windows. Проаналізовано бібліотеки і моделі для аналізу зображень та формування текстових описів, обґрунтовано вибір Moondream2 як основної моделі інференсу. Спроєктовано та реалізовано прототип системи з використанням стеку FastAPI, SQLite, PyTorch і Transformers. Проведено тестування системи в умовах локальної мережі з двома робочими вузлами та підтверджено коректність розподіленої обробки задач.

Об’єктом дослідження є процес автоматизованого формування навчальних датасетів на основі розподіленої обробки зображень у локальній мережі.

Предметом дослідження є архітектурні рішення, протоколи взаємодії та інструментальні засоби побудови системи типу майстер-воркер для задач комп’ютерного зору.

Практична цінність роботи полягає в тому, що реалізований прототип може бути розгорнутий на будь-яких робочих ПК під управлінням Windows без спеціалізованої серверної інфраструктури. Система зберігає метадані кожного результату, що забезпечує відтворюваність датасету при зміні моделі або параметрів інференсу. Розподілене виконання задач без явного планувальника дозволяє масштабувати систему без додаткових налаштувань через просте підключення нових вузлів.

1 РОЗПОДІЛЕНА СИСТЕМА АНАЛІЗУ ЗОБРАЖЕНЬ ДЛЯ ФОРМУВАННЯ НАВЧАЛЬНИХ ДАТАСЕТІВ

1.1 Аналіз типів розподілених систем

Розподіленою називають систему, в якій обчислення та дані розміщені на кількох вузлах, що обмінюються інформацією через мережу, тоді як користувач взаємодіє з нею як з єдиним сервісом. Стосовно задачі аналізу зображень це означає, що обробку виконує не один комп’ютер, а кілька одночасно. Центральний сервер розподіляє завдання між робочими вузлами, контролює їх виконання та збирає результати у єдиний набір даних.

Розподілений підхід для задач комп’ютерного зору дає дві ключові переваги: продуктивність і стійкість. Оскільки інференс VLM на одному зображенні займає десятки секунд на CPU, послідовна обробка великих наборів стає вузьким місцем – паралельна робота кількох вузлів усуває це обмеження. Падіння одного воркера при цьому не зупиняє весь процес: центральний сервер просто перерозподіляє незавершені задачі між тими, що залишились.

В основу реалізованої системи покладено підхід майстер-воркер. Сервер роздає задачі на обробку зображень, робочі вузли виконують інференс моделі і повертають текстовий результат. Схема добре масштабується: кожне зображення обробляється незалежно, а продуктивність зростає пропорційно до кількості підключених воркерів. Для забезпечення стабільної роботи сервер веде стан кожної задачі: pending (нова), in_progress (у роботі), done (завершена), error (помилка). Разом із результатом зберігаються метадані: ідентифікатор воркера, назва і ревізія моделі, час обробки, параметри генерації тексту. Це дозволяє повторити обробку при зміні моделі або параметрів інференсу і відстежити, якою саме конфігурацією отримано конкретний опис.

Принцип розподілу обов’язків у майстер-воркер системі простий: майстер не рахує, воркер не запам’ятовує. Майстер лише веде чергу задач, роздає їх на запит і зберігає результати. Воркер після кожного зображення одразу запитує наступне, не зберігаючи жодного стану між задачами. Відсутність збереженого стану між задачами спрощує перезапуск воркерів і додавання нових вузлів без зміни конфігурації майстра.

Для коректного обґрунтування вибору архітектури розглянуто основні класи розподілених систем: кластерний, сітковий, кордонний та туманний типи, а також хмарно-орієнтований підхід як методологію організації сервісів.

Кластерні системи. Кластер об’єднує кілька однорідних вузлів, з’єднаних швидкою локальною мережею, що працюють як єдиний обчислювальний ресурс[44]. Для задач комп’ютерного зору кластерний підхід доцільний при пакетній обробці великих обсягів даних або стабільному потоковому інференсі. Керування ресурсами реалізується через контейнеризацію (Docker) та оркестрацію (Kubernetes). Docker пакує програму разом із залежностями для однакового запуску на різних вузлах. Kubernetes керує контейнерами: створює потрібну кількість воркерів, перезапускає їх після збоїв, розподіляє навантаження і дозволяє динамічно додавати вузли при зростанні обсягу задач.

Сильна сторона кластера полягає у передбачуваності та контролі. Можна задати ліміти ресурсів на кожен контейнер, налаштувати автоматичні перезапуски при перевищенні порогів CPU чи пам’яті, забезпечити балансування навантаження між вузлами. Для системи аналізу зображень це означає стабільний час обробки і можливість прогнозувати пропускну здатність системи. Водночас кластер як інфраструктура потребує окремих серверних вузлів, налаштованої мережі, спільного сховища для файлів і результатів, а також спеціалізованих знань для розгортання та підтримки. Для навчального прототипу, де воркерами виступають звичайні робочі ПК під управлінням Windows, повноцінний Kubernetes створює надлишкову інфраструктурну складність. Кластерний підхід розглядається як варіант масштабування системи у майбутньому.

Мережеві сіткові системи. Сіткові обчислення об’єднують ресурси різних організацій або географічно розподілених вузлів. Учасники надають частину власних ресурсів для виконання спільних задач, що вимагає механізмів автентифікації, журналювання та контролю середовища виконання. Комп’ютери в гріді часто різнорідні: одні потужні, інші слабші, у частини є GPU, у частини немає, можуть відрізнятись операційні системи і версії бібліотек. Система має вміти підлаштовуватись під ці умови і розподіляти простіші задачі слабшим вузлам, тоді як ресурсоємні задачі направляти на продуктивніші вузли.

Для аналізу зображень грід демонструє цікаві властивості, зокрема механізми обліку використання ресурсів і контракти виконання задач. У задачі формування датасету всі вузли працюють у довіреному локальному середовищі, тому складна інфраструктура автентифікації і контролю доступу є надлишковою. Грід корисний як порівняльний клас що демонструє важливість правил взаємодії між вузлами.

Кордонні обчислення (Edge computing). Кордонний підхід переносить частину обчислень ближче до джерела даних[16]. Переваги полягають у меншій затримці, зменшенні навантаження на центральний сервер і можливості працювати при нестабільному зв’язку. У контексті реалізованої системи воркери фактично є edge-вузлами: вони отримують зображення, виконують інференс локально і повертають лише текстовий результат на сервер[17]. Центральний сервер не виконує обчислень. Його роль зводиться до керування чергою задач, контролю тайм-аутів і забезпечення цілісності даних.

Edge-логіка на стороні воркера охоплює кілька етапів передобробки. Спершу виконується перевірка цілісності файлу: чи не пошкоджене зображення, чи відповідає воно очікуваному формату. Далі застосовується конвертація колірного простору в RGB, оскільки модель Moondream2 очікує саме такий формат входу. Великі зображення масштабуються до максимальної сторони 1536 пікселі, що зменшує час інференсу без помітної втрати якості опису. Лише після цих етапів зображення передається на вхід нейромережі. Такий поділ роботи між сервером і воркером відповідає класичній edge-моделі, де "розумна" частина обробки винесена на периферію.

Туманні обчислення (Fog computing). Проміжний рівень між кордонними і хмарними обчисленнями[18]. У типовому сценарії fog-вузлом виступає локальний сервер у мережі підприємства, що надає спільні обчислювальні сервіси для групи робочих місць. На відміну від хмари, fog-вузол фізично знаходиться у локальній мережі, що забезпечує низьку затримку і незалежність від інтернет-з’єднання. На відміну від кордонного, туманний вузол має більше ресурсів і може централізовано керувати кількома кордонними пристроями.

Реалізований master-сервер відповідає саме цій логіці. Керування відбувається локально в мережі підприємства або лабораторії, виконання інференсу покладено на робочі ПК-воркери, спільне сховище зображень і база даних розміщені на майстрі. За наявності виділеного GPU fog-вузол міг би брати на себе також ресурсоємні етапи обробки, наприклад усунення дублікатів зображень на основі векторних представлень або повторну обробку накопичених задач після оновлення моделі. У поточній реалізації ці функції залишені на майбутнє, проте архітектура системи дозволяє їх додати без переробки протоколу обміну.

Cloud-native. Cloud-native є не окремим типом мережі, а методологією побудови системи з акцентом на декомпозицію сервісів, контейнеризацію та спостережуваність[3, 50]. У такому підході функції системи розділяються на компоненти: сервіс видачі задач, сервіс зберігання результатів, сервіс керування моделями, сервіс моніторингу. Кожен компонент оновлюється і масштабується незалежно. Контрольованість стану системи реалізується через структуроване журналювання, метрики виконання і трасування запитів між сервісами[10].

У межах цієї роботи хмарно-орієнтований підхід застосовано як принцип структуризації коду, а не як вимогу до інфраструктури. Сервер, воркер і сховище реалізовані як незалежні модулі з чітко визначеними API та форматами повідомлень, де на стороні сервера використовуються кінцеві точки REST, на стороні воркера застосовується HTTP-клієнт з механізмом повторів, а налаштування винесені в окремий модуль конфігурації через .env-файл. Навіть без Kubernetes така декомпозиція спрощує тестування, налагодження і потенційне перенесення компонентів у контейнери. Відмовостійкість забезпечена через повтор задач, контроль тайм-аутів і журналювання ідентифікаторів воркерів.

Таким чином, найбільш придатним для умов реалізації є підхід майстер-воркер у локальній мережі з опорою на ідеї кордонних та туманних обчислень. Кластерні і сіткові системи розглядаються як порівняльні класи та варіанти розширення системи. Cloud-native підхід застосовується як спосіб правильно структурувати компоненти, навіть якщо реалізація залишається легкою. Порівняльний аналіз розглянутих типів розподілених систем у контексті поставленої задачі наведено у таблиці 1.1.

Таблиця 1.1 – Порівняння типів розподілених систем у контексті завдання

Тип	Ознаки	Переваги для задачі	Недоліки та ризики
Кластерні системи	Однорідні вузли, керування ресурсами,контейнеризація/оркестрація	Стабільний інференс, масштабування, моніторинг	Потребує інфраструктури та адміністрування
Мережеві сіткові системи	Гетерогенні й розподілені ресурси, складні правила доступу	Показує механізми довіри та обліку	Надмірна складність для локальної мережі

Кінець таблиці 1.1

Кордонні обчислення	Обробка ближче до джерела даних, локальні вузли	Паралельна обробка на робочих ПК, менша затримка	Залежить від ресурсів воркерів, потрібен контроль версій
Туманні обчислення	Проміжний рівень між кордонним і хмарним, локальний сервер	Зручний майстер вузол у локальній мережі, спільні сервіси	Потрібно забезпечити стабільність і сховище
Хмарно-орієнтований підхід	Мікросервіси, автоматизація, спостережуваність	Добра структуризація компонентів і API	Може бути надмірним без контейнерів
Майстер-воркер	Диспетчер і незалежні воркери, задачі дрібні й паралельні	Найпростіша відповідність постановці задачі	Потрібні правила тайм-аутів і повтору

1.2 Аналіз платформ і протоколів для реалізації системи

Розподіл задач між вузлами можна організувати двома способами. У push-моделі сервер або брокер надсилає задачі воркерам через чергу. Воркер отримує задачу автоматично, як тільки вона з’являється у системі. У pull-моделі воркер сам звертається до сервера і запитує наступну задачу, коли звільняється після попередньої. Обидва варіанти підходять для паралельної обробки зображень, але відрізняються обсягом необхідної інфраструктури та складністю реалізації.

Push-черга зручна в кластері з брокером повідомлень типу RabbitMQ або Redis[11]. Брокер виступає посередником між сервером і воркерами, гарантуючи доставку повідомлень навіть при тимчасових збоях мережі. Така схема спрощує підтвердження виконання та повтор задач при збоях, оскільки логіка обробки помилок зосереджена в самому брокері. Натомість для розгортання push-черги потрібен окремий сервіс брокера, його налаштування і моніторинг. У промислових системах це виправдано, проте для локального розгортання на робочих ПК під Windows створює надлишкову залежність від інфраструктури.

Pull-диспетчеризація простіша для локальної мережі. Достатньо однієї кінцевої точки HTTP, яка видає наступну задачу при запиті, і другого – для прийому результату. Повтор реалізується на сервері, де задача повертається у статус pending після спливання тайм-ауту. Воркер не потребує постійного з’єднання з брокером і може приєднатись або від’єднатись від системи у будь-який момент. Це особливо цінно для умов, де воркерами виступають робочі ПК, які періодично вимикаються або використовуються для інших задач. За цих умов pull-модель обрано як базову.

Celery є популярним інструментом для фонових задач у Python. Він реалізує типову схему "задача, воркер, результат" через брокер повідомлень і підтримує автоматичний повтор при збоях, періодичні задачі за розкладом і пріоритизацію черг. На практиці Celery часто поєднують з FastAPI: API приймає запит від користувача, важку обробку передає у чергу, а воркери забирають задачі по мірі готовності[11]. Система сама вирівнює навантаження. Якщо задач багато, вони накопичуються в черзі, а воркери забирають їх з потрібною швидкістю. Це зручно для обробки зображень, оскільки дозволяє контролювати паралельність і уникати перевантаження ресурсів.

Водночас у документації Celery зазначено, що Windows офіційно не підтримується як платформа для запуску воркерів. Розробники Celery пояснюють це обмеженнями, пов’язаними з механізмами форкання процесів, які працюють інакше у Windows порівняно з Linux. Хоча існують неофіційні способи запуску Celery на Windows через зміну executor-а на solo або gevent, для дипломного проєкту це створює ризик: при питаннях нормоконтролю або захисту складно обґрунтувати використання офіційно непідтримуваної конфігурації. Тому Celery розглядається як джерело термінології (broker, worker, task, ack) і порівняльний приклад, а не як основа реалізації. Головним критерієм вибору залишається мінімальна залежність від зовнішньої інфраструктури.

Ray позиціонується як платформа для розподілених застосунків на Python з підтримкою задач, акторів і кластерного середовища[45, 46]. Платформа надає готові механізми, корисні для систем типу майстер-воркер. Можна задавати чергу задач і обмежувати паралельність, щоб не перевантажувати GPU або CPU на конкретних вузлах. Ray надає зручні інструменти моніторингу: видно які задачі запущені, скільки часу вони виконуються, на якому вузлі і з яким результатом завершились. У більш розгорнутих сценаріях Ray дозволяє поєднати обробку зі збереженням проміжних результатів і повтором задач, що важливо для відтворюваності датасетів.

Проте у документації Ray зазначено, що багатовузлові кластери на Windows позначені як експериментальні і не рекомендовані для промислового використання. Single-node режим працює стабільно, але для даної задачі не дає переваги: на одному вузлі можна обійтись звичайним Python без додаткових бібліотек. Цей факт підтверджує доцільність простішого протоколу на базі HTTP. Ray залишається корисним як порівняльний приклад – деякі його ідеї перенесено у реалізацію простими засобами: контроль тайм-аутів і журналювання ідентифікаторів воркерів.

Apache Airflow і Prefect належать до іншого класу інструментів – workflow-платформ. Вони дозволяють описати процес обробки даних як граф задач (DAG, directed acyclic graph) із залежностями, розкладом і повторами при помилках. Workflow-платформи зазвичай добре підходять, коли потрібно керувати станом конвеєра в часі. Можна запускати обробку щодня, повторювати кроки при помилці, зберігати історію виконань і бачити, на якому етапі виник збій.

Для формування датасетів workflow-підхід доцільний у двох сценаріях. Перший – регулярне поповнення датасету новими даними, де кожен день обробляється свіжа партія зображень. Другий – повторна генерація результатів після оновлення моделі або зміни правил фільтрації. Workflow-інструменти спрощують контроль залежностей між етапами: збереження результатів виконується тільки після успішного інференсу та валідації, а при збої система автоматично ставить задачу на повтор або на ручну перевірку.

Натомість у реалізованій системі задача інша. Воркер бере одне зображення, обробляє і одразу повертає результат майстру. Тут важливіший простий і надійний обмін повідомленнями, а не складна оркестрація багатоетапних конвеєрів. Workflow-платформи розглядаються як альтернативний клас і опція для майбутнього розширення. Якщо знадобиться пакетна перегенерація описів після зміни моделі, тоді workflow-шар стане доречним. У поточній роботі ці сценарії враховані на рівні схеми даних і журналювання: метадані кожного результату дозволяють у майбутньому повторно обробити будь-яку підмножину датасету.

NVIDIA Triton Inference Server і TorchServe реалізують ще один підхід – розгортання моделей як окремого мережевого сервісу. У такому випадку воркер стає тонким клієнтом, який отримує задачу, передає зображення на сервер обробки і повертає текст на майстер. Це зручно при наявності одного або кількох потужних серверів з графічними прискорювачами і слабких клієнтських машин. Сервер Triton орієнтований на високу продуктивність і підтримує як роботу з популярними фреймворками машинного навчання (PyTorch, TensorFlow), так і відкритий стандарт представлення моделей ONNX, тоді як TorchServe призначений виключно для моделей PyTorch. У середовищі Kubernetes для керування такими сервісами часто додають інструмент KServe.

При розгортанні та наданні доступу до моделей важливо визначити формат обміну даними. Найчастіше воркер не пересилає саме зображення на сервер обробки при кожному запиті, а передає посилання або ідентифікатор у спільному сховищі, щоб зменшити мережевий трафік. Разом із результатом інференсу повертаються версія моделі та параметри генерації, що дозволяє пізніше повторити обробку і порівняти якість при різних конфігураціях. Такі рішення підходять для промислових систем, проте потребують значно більше налаштувань і інфраструктури.

У межах цієї роботи розгортання моделей як окремого сервісу розглядається як варіант розвитку системи. Для базового прототипу обрано інший сценарій: інференс запускається локально на воркері, а через мережу передається тільки задача і результат. Це спрощує систему, зменшує залежність від окремого GPU-сервера і дозволяє розгорнути воркер на будь-якому ПК з достатньою кількістю оперативної пам’яті.

Основою взаємодії в розробленій системі є протокол зв’язку між майстром і воркером. На практиці потрібно мінімум дві операції: видати задачу і прийняти результат. Для цього вистачає JSON і стандартних HTTP-методів, тому HTTP/REST є логічним вибором для першої версії системи[21]. REST має додаткові переваги: легке налагодження через звичайний браузер або curl, широка підтримка у всіх мовах програмування, можливість використання інтерактивної документації Swagger UI прямо в браузері.

FastAPI обрано для реалізації серверного API. Бібліотека дозволяє швидко описати запити і відповіді, виконати валідацію вхідних даних через Pydantic-схеми і автоматично згенерувати OpenAPI-специфікацію[1]. У документації FastAPI окремо зазначено, що важкі обчислення краще виносити в окремі воркери, а API залишати легким[14]. Це повністю узгоджується з архітектурою майстер-воркер, де майстер виконує лише функції диспетчера, не виконуючи самого інференсу. На стороні воркера для HTTP-запитів застосовано бібліотеку httpx, що підтримує синхронні та асинхронні запити з вбудованими таймаутами і повторами.

gRPC є альтернативою HTTP/REST, коли потрібні чітко типізовані повідомлення, потокова передача і ефективніший транспорт поверх HTTP/2. gRPC використовує бінарний формат Protobuf замість JSON, що зменшує обсяг переданих даних і прискорює серіалізацію. Для навчального прототипу gRPC є зайвим: переваги виявляються при високих навантаженнях і строгих вимогах до затримки, тоді як ускладнення розробки та налагодження не компенсуються. gRPC варто розглядати як можливе покращення при зростанні навантаження або появі потреби в стабільних типізованих контрактах між компонентами, написаними різними мовами програмування.

Окремий практичний момент стосується надійності обміну повідомленнями. Навіть при простому HTTP/REST потрібні базові правила: унікальний ідентифікатор задачі, тайм-аут на обробку, повторна видача задачі при відсутності відповіді та підтвердження прийому результату. Без цих правил воркер, який завершився з помилкою або втратив зв’язок із сервером, залишає задачу у "висячому" стані: вона помічена як in_progress, але фактично ніхто над нею не працює. У реалізованій системі цю проблему вирішено через механізм reclaim: якщо задача знаходиться у статусі in_progress довше ніж заданий тайм-аут, сервер автоматично повертає її у pending для повторної видачі. Кожна задача має лічильник retry_count, і після перевищення максимальної кількості повторів вона переводиться у статус error для ручного розгляду.

Окремою вимогою до реалізації є захист від одночасної видачі однієї задачі двом воркерам. Класичні рішення на базі SELECT FOR UPDATE непридатні через архітектурні особливості SQLite[19, 20]. Детальний опис обраного рішення через конструкцію UPDATE...RETURNING наведено у підрозділі 2.2.

Після порівняння підходів ключовим критерієм став відповідність умовам розгортання: воркерами є робочі ПК під Windows, мережа локальна, потрібен простий і відтворюваний прототип. За цих умов найбільш керованим рішенням є власний сервіс-диспетчер на FastAPI та воркери на Python, що працюють циклом "отримати задачу – завантажити зображення – запустити модель – відправити результат". Зберігання даних розділено на дві частини: зображення зберігаються у файловій системі майстра, метадані задач і результати – у таблицях SQLite. Так простіше працювати з великими бінарними файлами і окремо з структурованими даними, а також виконувати експорт датасету у CSV-форматі без додаткової обробки.

Для стабільності закладено базові правила: унікальні UUID-ідентифікатори задач, контроль тайм-аутів обробки, автоматичний повтор при збоях воркера, ідентифікація воркера через hostname і UUID. Цього достатньо для демонстрації роботи системи в навчальному проєкті без розгортання складної інфраструктури. Дослідивши відомі подібні системи (Celery, Ray, workflow-платформи, інференс-сервери), обрано компактну майстер-воркер систему на HTTP/REST, яку можна розширити в майбутньому або перенести на більш індустріальні рішення без зміни архітектурних принципів. Порівняння платформ і підходів для реалізації системи, а також мережевих протоколів передачі даних наведено у таблицях 1.2 і 1.3.

Таблиця 1.2 – Порівняння платформ і підходів для реалізації системи

Підхід/система	Модель взаємодії	Переваги	Обмеження
FastAPI + власний диспетчер	HTTP/REST (pull), JSON	Простота, контроль протоколу, мінімум залежностей	Потрібно самостійно реалізувати повтор/тайм-аути
Celery + брокер (RabbitMQ/Redis)	Push-черга задач	Готові ack/повтори, зрозуміла модель задач	Windows не підтримується офіційно для воркерів
Ray (tasks/actors)	Кластерний рантайм	Розподілення задач, моніторинг, масштабування	Multi-node на Windows експериментальний

Кінець таблиці 1.2

Workflow (Airflow/Prefect)	DAG/оркестрація	Зручно для пакетної обробки і перегенерацій	Зайва складність для онлайн-диспетчера
Model serving (Triton/TorchServe)	RPC/HTTP API до інференс-сервера	GPU-централізація, стабільність і масштабування	Потрібен окремий сервер, налаштування сервінгу

Таблиця 1.3 – Порівняння мережевих протоколів та способів передачі даних

Варіант	Переваги	Недоліки	Висновок
HTTP/REST + JSON	Простий стек, легко налагоджувати, підтримка всюди	Немає типізації контрактів з коробки, зайві накладні витрати на JSON	Базовий вибір для прототипу
gRPC + Protobuf	Типізовані контракти, висока ефективність, HTTP/2	Складніше налагодження, потребує генерації клієнтів	Опція для масштабування
Файловий обмін + метадані через API	Мінімальний трафік у запитах, зручно в локальній мережі	Потрібно налаштувати доступ до сховища	Практичний варіант для великих файлів

1.3 Аналіз інструментальних засобів та моделей для формування текстових описів зображень

На стороні воркера задіяно два рівні інструментів[38, 39]. Перший рівень охоплює базові бібліотеки для підготовки даних: читання файлів, перевірка якості, приведення формату і нормалізація розмірів. Другий рівень – бібліотеки для запуску моделей, що виконують інференс і генерують текстовий опис. Такий поділ дозволяє чітко розмежувати етапи обробки і забезпечити відтворюваність результатів через фіксацію версій бібліотек, моделі та параметрів генерації.

Підготовка зображень перед інференсом є важливою частиною конвеєра. Більшість зорово-мовних моделей очікує входу у конкретному форматі: трьохканальне RGB-зображення, певний діапазон значень пікселів, обмежена роздільна здатність[23, 24]. Якщо подати на вхід зображення з невідповідним форматом або пошкодженим файлом, модель або поверне помилку, або згенерує опис низької якості. Через це передобробка має виконуватись надійно і прогнозовано на всіх воркерах, незалежно від конфігурації окремого ПК.

До першого рівня належать OpenCV, Pillow та scikit-image. OpenCV орієнтований на швидку роботу з матрицями і містить широкий набір алгоритмів класичного комп’ютерного зору: фільтрація, виявлення країв, морфологічні операції, обчислення гістограм. Бібліотека написана на C++ з Python-обгорткою, що забезпечує високу продуктивність на великих зображеннях. Pillow (відомий також як PIL – Python Imaging Library) зручний для читання і збереження файлів та простих перетворень: зміна розміру, конвертація колірних просторів, обертання, обрізка. На відміну від OpenCV, Pillow має чистий Python-інтерфейс і простіший у використанні для базових операцій. scikit-image має науково орієнтований інтерфейс і фокусується на алгоритмах обробки і фільтрації зображень, інтегрується з NumPy і часто застосовується в дослідницьких задачах.

У реалізованій системі ці бібліотеки утворюють інструментальний шар, що не задає архітектуру, але безпосередньо впливає на якість вхідних даних для моделі. На практиці застосовано поєднання Pillow для стабільного читання файлів і OpenCV для швидких перетворень. Pillow обрано основним інструментом для I/O через простий API і добру підтримку різних форматів (JPEG, PNG, BMP, WEBP, TIFF). При читанні зображення виконується автоматична конвертація в RGB-режим, оскільки PNG-файли можуть мати альфа-канал (RGBA), а деякі TIFF-файли – режими L (градації сірого) або P (індексовані кольори), що несумісно з вимогами Moondream2. Якщо максимальна сторона зображення перевищує 1536 пікселі, виконується пропорційне масштабування з використанням алгоритму Lanczos, що зберігає деталі при зменшенні. Це забезпечує переносимість воркера на різні робочі ПК і зменшує залежність від специфічних функцій окремих бібліотек.

Стандартом для запуску сучасних моделей комп’ютерного зору в Python є PyTorch[2]. Він надає тензорний інтерфейс, схожий на NumPy, але з підтримкою автоматичного диференціювання і виконання обчислень на GPU через CUDA. PyTorch розроблений Meta AI і має широку підтримку у спільноті дослідників машинного навчання, що проявляється у великій кількості готових реалізацій моделей і регулярних оновленнях бібліотеки[41]. Альтернативою є TensorFlow від Google, проте PyTorch обрано через простіший Python-інтерфейс і кращу інтеграцію з Hugging Face Transformers, що є основним інструментом завантаження моделей у даному проєкті.

У реалізованій системі PyTorch потрібен не лише для запуску моделі, а й для забезпечення відтворюваності результатів. Фіксується версія PyTorch, версія CUDA за наявності GPU, назва моделі, ревізія ваг на Hugging Face Hub та параметри генерації тексту (max_new_tokens, num_beams). Це зменшує розкид результатів між запусками і робить датасет керованим артефактом. При зміні будь-якого параметра можна зрозуміти, чому саме змінились описи. Torchvision доповнює PyTorch готовими перетвореннями даних і типовими операціями підготовки зображень: зміна розміру, нормалізація під конкретні ваги моделі, перетворення з PIL Image у тензор. У даному проєкті більшість таких операцій виконує сама модель Moondream2 через свій метод encode_image, тому torchvision використовується мінімально.

Для завантаження і запуску зорово-мовних моделей застосовано бібліотеку Hugging Face Transformers[4, 42]. Вона надає уніфікований інтерфейс для завантаження моделей через AutoModelForCausalLM і AutoTokenizer, токенізаторів для обробки тексту і процесорів зображень для перетворення вхідних даних у формат, очікуваний моделлю. Бібліотека також підтримує генерацію тексту з різними стратегіями декодування: жадібний пошук, пучковий пошук та випадкова вибірка. Hugging Face надає не тільки бібліотеку, а й Hub – централізований репозиторій моделей, де кожна модель має унікальний ідентифікатор виду organization/model-name і може мати кілька ревізій (тегів) для різних версій ваг.

Загальний принцип інтеграції моделі зафіксовано на рівні воркера: отримати задачу, завантажити зображення, передати його у метод encode_image моделі, сформувати підказку (prompt), викликати метод answer_question, отримати текстовий результат і повернути його з метаданими. При зміні моделі змінюється лише спосіб виклику, а не загальна логіка системи. Контроль версій бібліотеки і кешу моделі є обов’язковим: різні версії Transformers можуть мати несумісні API для одних і тих самих моделей, що було підтверджено практично під час розробки[5]. Зокрема, Moondream2 ревізії 2024-08-26 несумісна з Transformers версій 4.45 і вище через зміни в обробці attention_mask, тому у проєкті зафіксовано версію transformers==4.42.4.

Як основну модель інференсу обрано Moondream2 (vikhyatk/moondream2, ревізія 2024-08-26)[22, 25]. Це компактна vision-language модель з відкритим кодом, що поєднує SigLIP-енкодер зображень із мовною моделлю на базі Phi-1.5[29, 30, 31]. Розмір моделі становить близько 1,86 мільярда параметрів, що значно менше за більшість конкурентів у класі (BLIP-2 має 2,7-12 мільярдів параметрів, LLaVA – від 7 мільярдів)[7]. Завдяки компактності Moondream2 запускається на CPU з прийнятною швидкістю: одне зображення обробляється за 20-30 секунд на сучасному процесорі без GPU. Це робить модель придатною для розгортання на звичайних робочих ПК без виділеного GPU-сервера, що відповідає умовам поставленої задачі.

Принцип роботи моделі побудований на двоетапній обробці. Спочатку SigLIP-енкодер перетворює зображення на послідовність векторних представлень (embeddings), що описують візуальний вміст у формі, придатній для мовної моделі[35]. Далі ці представлення передаються в Phi-1.5 як контекст разом із текстовою підказкою користувача. Мовна модель генерує відповідь токен за токеном, базуючись на візуальному контексті і заданому промпті. Для задачі формування датасету застосовано підказку "Describe this image in detail.", що забезпечує розгорнутий опис вмісту зображення з деталізацією об’єктів, їх взаємного розташування і характеристик.

Вага моделі становить приблизно 3,7 ГБ і завантажується один раз при старті воркера з Hugging Face Hub, після чого зберігається в локальному кеші (~/.cache/huggingface/hub/). Це виключає повторне завантаження при наступних запусках і забезпечує стабільний час ініціалізації, що становить близько 4 секунд на сучасному CPU після першого скачування. Фіксація конкретної ревізії 2024-08-26 у коді гарантує, що всі воркери використовуватимуть однакову версію ваг, навіть якщо модель оновлюється на Hub. Це принципово важливо для відтворюваності датасету: при тих самих вхідних зображеннях і параметрах генерації різні воркери повинні видавати однакові описи[39].

На момент аналізу розглядались також альтернативні моделі. Florence-2 від Microsoft є уніфікованою моделлю, що підтримує широкий спектр задач комп’ютерного зору через текстову підказку: captioning, детектування об’єктів, OCR, сегментація[8, 15]. Модель доступна у двох розмірах: Florence-2-base (230 мільйонів параметрів) і Florence-2-large (770 мільйонів). Перевага полягає в універсальності: змінюючи лише підказку, можна отримати різні типи відповідей без зміни архітектури системи. Проте під час практичної перевірки виявилось, що нові ревізії Florence-2 потребують бібліотеки pyvips, яка погано встановлюється на Windows і потребує додаткових системних залежностей. Це зробило розгортання Florence-2 на робочих ПК під Windows проблематичним.

BLIP-2 від Salesforce поєднує візуальний енкодер CLIP з мовною моделлю OPT або Flan-T5 через проміжний модуль Q-Former. Модель спеціалізована саме для задачі генерації описів зображень і має багато готових реалізацій[7, 13]. Якість описів BLIP-2 високо оцінюється у дослідженнях, проте розмір моделі (від 2,7 до 12 мільярдів параметрів залежно від конфігурації) робить її повільною на CPU: обробка одного зображення може займати кілька хвилин без GPU. Для умов поставленої задачі це неприйнятно[12].

Інші розглянуті моделі (LLaVA, MiniGPT-4, InstructBLIP) мають аналогічні обмеження: вони потребують GPU для прийнятної швидкості і мають значно більший розмір ваг. Moondream2 обрано як оптимальний компроміс між якістю описів, швидкістю інференсу на CPU і простотою розгортання на робочих ПК.

Ще одним суттєвим аспектом розробки є підходи до оцінювання якості згенерованих результатів. Для генерації описів зображень існують автоматичні метрики, що порівнюють згенерований опис із еталонним[27, 28, 32, 33, 34]. BLEU обчислює перетин n-грам між генерованим і еталонним текстом, METEOR враховує синонімію і морфологію, ROUGE-L базується на найдовшій спільній підпослідовності. CIDEr і SPICE розроблені спеціально для captioning і враховують семантичну схожість[47, 48]. CLIPScore використовує семантичні ембединги від моделі CLIP для порівняння зображення і опису без потреби в еталонному тексті[49].

Жодна метрика окремо не дає повної картини, тому на практиці застосовують кілька показників одночасно і виконують ручну перевірку частини результатів. Складність оцінювання captioning полягає в тому, що для одного зображення може існувати кілька правильних описів різного рівня деталізації, і автоматичні метрики можуть штрафувати валідні варіанти, які відрізняються від еталонних формулювань.

У межах поточної роботи реалізовано базовий контроль якості на рівні системи. Перевіряється наявність порожніх відповідей, обмежується максимальна довжина виведення (256 токенів), відстежуються випадки повторення однакових конструкцій, що іноді трапляються при некоректних параметрах генерації[40]. Для відтворюваності датасету разом із кожним описом зберігаються назва та ревізія моделі, ідентифікатор воркера, час обробки та параметри генерації тексту в JSON-форматі. Ці метадані дозволяють пояснювати різницю в результатах при перегенерації датасету з оновленою моделлю і коректно порівнювати ефективність різних конфігурацій. Повноцінне оцінювання якості з застосуванням автоматичних метрик і ручної перевірки виходить за межі поточного прототипу і розглядається як напрям подальшого розвитку системи.

Порівняльний аналіз бібліотек і моделей для задачі аналізу зображень та формування текстових описів наведено у таблиці 1.4.

Таблиця 1.4 – Порівняння моделей і бібліотек

Компонент	Призначення	Вимоги	Переваги	Недоліки
Moondream2	Генерація описів зображень через текстову підказку	PyTorch, CPU достатньо	Компактна, працює без GPU, фіксована ревізія	Повільніша за великі моделі на CPU

Кінець таблиці 1.4

Florence-2	Універсальні vision-таски через текстовий вихід	PyTorch, бажано GPU	Один підхід до різних задач	Потрібно підбирати промпти і параметри
BLIP-2	Генерація описів зображень	PyTorch, ресурси залежать від конфігурації	Якісні описи, багато готових реалізацій	Може бути важкою на CPU
OpenCV	Препроцесинг, перевірки якості	CPU, нативні залежності	Швидко і практично	Більший поріг входу для окремих задач
Pillow	I/O і базові перетворення	CPU	Легко інтегрувати, прості операції	Не замінює повний CV-стек
Transformers	Завантаження VLM-моделей, генерація тексту	Python-екосистема	Уніфікований API, багато моделей	Потрібен контроль версій і кешу

1.4 Висновки до першого розділу

За результатами аналізу типів розподілених систем обґрунтовано вибір архітектури майстер-воркер з опорою на ідеї edge та fog. Підхід забезпечує паралельну обробку зображень на кількох робочих ПК, тоді як сервер зберігає контроль задач, статусів і підсумкових результатів. Кластерні і грід-системи розглянуто як порівняльні класи і варіанти розширення системи у майбутньому. Cloud-native підхід застосовано як методологію структуризації коду через декомпозицію на незалежні модулі з чіткими API.

Порівняння відомих платформ показало, що Celery і Ray є логічними кандидатами для задач подібного масштабу. Проте обидва інструменти мають обмеження підтримки Windows, тому обрано компактну майстер-воркер схему без окремого брокера на базі FastAPI з pull-механізмом видачі задач. Для зберігання задач і результатів застосовано SQLite з WAL-режимом, а атомарне призначення задач реалізовано через UPDATE ... RETURNING, що усуває race condition при одночасному запиті задач кількома воркерами.

Аналіз бібліотек і моделей підтвердив доцільність стеку PyTorch і Transformers для інференсу на стороні воркера. Як основну модель обрано Moondream2 (ревізія 2024-08-26): компактна vision-language модель з 1,86 мільярда параметрів, що запускається на CPU без виділеного GPU-сервера. Florence-2 і BLIP-2 розглянуто як альтернативи і відхилено через обмеження сумісності з Windows та вищі вимоги до ресурсів. Для підготовки зображень застосовано Pillow з автоматичною конвертацією у RGB і масштабуванням до 1536 пікселів по довшій стороні.

Обрана архітектура майстер-воркер з pull-механізмом взаємодії є не просто теоретичним вибором, а практично перевіреним рішенням для умов локальної мережі з робочими ПК під Windows. Кожен з розглянутих класів систем підтвердив свою роль: кластерний і сітковий підходи як орієнтири для майбутнього масштабування, кордонний і туманний як концептуальна основа поточної архітектури.

Результати аналізу узагальнено у таблицях 1.1-1.4. Сформовані висновки є основою для проєктування і реалізації системи у наступному розділі.

2 ПРОЄКТУВАННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ РОЗПОДІЛЕНОЇ СИСТЕМИ АНАЛІЗУ ЗОБРАЖЕНЬ

2.1 Вибір мови програмування та бібліотек

Для реалізації розподіленої системи аналізу зображень обрано мову Python версії 3.10 і вище. Вибір зумовлений трьома причинами. Python є фактичним стандартом для задач машинного навчання і комп'ютерного зору, тому всі сучасні vision-language моделі мають готові реалізації саме для цієї екосистеми. Python однаково добре підтримує мережеві сервіси і нейромережі, що дозволяє реалізувати майстер і воркер в одній кодовій базі. Кросплатформеність забезпечує запуск однакового коду на Windows, Linux і macOS без модифікацій.

Серверну частину системи реалізовано на FastAPI – асинхронному веб-фреймворку, що використовує стандарт ASGI[1, 14]. FastAPI автоматично генерує OpenAPI-специфікацію і надає інтерактивну документацію Swagger UI прямо у браузері за адресою `/docs`. Це спрощує налагодження, кожну кінцеву точку можна перевірити без написання окремого клієнта. Бібліотека також виконує автоматичну валідацію вхідних даних через Pydantic-схеми, де некоректний запит від воркера повертає зрозумілу помилку з вказівкою на проблемне поле, без зависань і непрозорих 500-х відповідей.

Для виконання FastAPI-додатку застосовано Uvicorn – високопродуктивний ASGI-сервер на базі бібліотек `uvloop` і `httptools`. Uvicorn запускається у фоновому потоці програми майстра, що дозволяє графічному інтерфейсу і HTTP-серверу працювати одночасно у межах одного процесу.

Pydantic версії 2.x описує структури даних, що передаються між майстром і воркером. Усі схеми (`TaskAssigned, ResultPayload, ErrorPayload, StatsOut`) задано як Python-класи з типізацією полів. Pydantic перевіряє типи на льоту і конвертує дані у потрібні формати: рядки у дати, словники у JSON і навпаки. Це усуває типовий клас помилок, коли воркер надсилає неправильний формат і сервер падає при спробі обробити запит.

Робота з базою даних побудована на SQLAlchemy 2.x[43]. Бібліотека дозволяє описати таблицю як клас Python (`Task`) і працювати з рядками як з об'єктами, без написання сирого SQL для більшості операцій. У поточній системі сирий SQL використано лише в одному місці, а саме для атомарного призначення задачі через `UPDATE ... RETURNING`, де ORM-абстракція призвела б до race condition.

Як рушій бази даних обрано SQLite[19, 20]. На відміну від PostgreSQL чи MySQL, він не потребує окремого серверного процесу і зберігає всю базу в одному файлі (`dataset.db`). Це ідеально підходить для прототипу і малих та середніх обсягів даних. SQLite запускається у режимі `WAL` (Write-Ahead Logging), що дозволяє кільком воркерам одночасно читати базу під час того як майстер пише результати. Починаючи з версії 3.35, SQLite підтримує конструкцію `UPDATE ... RETURNING`, яка є ключовою для атомарного призначення задач.

На стороні воркера для HTTP-запитів використано httpx замість більш відомої requests. httpx підтримує синхронний і асинхронний інтерфейси з однаковим API, має вбудовану підтримку HTTP/2 і строге керування таймаутами. У реалізованій системі застосовано синхронний клієнт з механізмом повторів: при тимчасовій недоступності майстра запит повторюється до трьох разів із зростаючою затримкою.

Для запуску vision-language моделі обрано бібліотеку Hugging Face Transformers фіксованої версії 4.42.4[4, 42]. Версія зафіксована свідомо: новіші релізи містять зміни в обробці attention-механізмів, які несумісні з обраною ревізією моделі Moondream2 від 2024-08-26. Така фіксація відповідає принципу відтворюваності, тобто будь-який воркер незалежно від часу встановлення отримає ідентичне середовище.

PyTorch виступає основою для моделі. Він надає тензорний інтерфейс, підтримку CUDA для GPU-прискорення (якщо доступне) і можливість виконання у режимі `eval()` без обчислення градієнтів. Для CPU-режиму застосовується тип `torch.float32`, для GPU – `torch.float16`, що зменшує споживання пам’яті удвічі без помітної втрати якості описів.

Pillow виконує роль основної бібліотеки роботи із зображеннями. Через неї воркер читає файли, конвертує колірні простори і масштабує великі зображення до 1536 пікселів по довшій стороні. Pillow обрано замість OpenCV: чистий Python-інтерфейс без потреби у нативних залежностях, простіша обробка PNG з прозорістю, добра підтримка форматів `WEBP` і `TIFF`.

Графічний інтерфейс побудовано на Tkinter – стандартній бібліотеці GUI, що входить у поставку Python. Вибір Tkinter замість більш сучасних альтернатив (PyQt, PySide, Kivy) зумовлений двома міркуваннями. Tkinter не потребує окремого встановлення і працює одразу після інсталяції Python, що спрощує розгортання на нових ПК. Для функціоналу системи (одне головне вікно, кілька кнопок, прогрес-бар, таблиця воркерів) можливостей Tkinter достатньо, а додавання важкої GUI-бібліотеки збільшило б розмір дистрибутиву на сотні мегабайтів без реальної вигоди.

Для конфігурації застосовано pydantic-settings – розширення Pydantic для читання змінних середовища з файлу `.env`. Усі параметри системи (адреса майстра, шлях до бази, тайм-аути, параметри генерації моделі) винесено у конфігураційний файл, що дозволяє змінювати поведінку без модифікації коду.

Управління версіями залежностей реалізовано через файл `requirements.txt`. Більшість пакетів задано через мінімальну версію (`>=`), що дозволяє pip встановити найновіший сумісний реліз. Винятком є дві позиції: `transformers==4.42.4` і `numpy==1.26.4`, для яких версія зафіксована точно. Точна фіксація версій обрана після практичної перевірки сумісності: новіші релізи transformers порушують сумісність з обраною ревізією Moondream2, а numpy 2.x несумісний з transformers 4.42.x через зміну внутрішніх API. Такий підхід відомий як "pinning" і широко застосовується у промислових Python-проєктах[5].

Обраний стек є цілісним і збалансованим: Python як основа, FastAPI з Pydantic для серверної частини, httpx для мережевого клієнта, PyTorch з Transformers для інференсу, SQLAlchemy з SQLite для зберігання, Tkinter для інтерфейсу. Усі компоненти мають активну підтримку спільноти і не дублюють функціонал один одного.

2.2 Реалізація серверної частини

Серверна частина системи, що отримала умовну назву "майстер", відповідає за зберігання задач, координацію роботи воркерів і збір результатів обробки. Майстер не виконує жодних обчислювальних операцій з зображеннями. Його функції зводяться до управління чергою задач, видачі задач воркерам на запит та збереження результатів у базу даних. Така мінімалістична роль відповідає принципу єдиної відповідальності: кожен компонент виконує одну задачу.

Реалізація майстра побудована з кількох логічних шарів, кожен з яких винесено в окремий модуль[3]. Моделі даних і підключення до бази (файли app/server/models.py і app/server/db.py) відокремлені від решти логіки. FastAPI у файлі app/server/api.py лише приймає запити і валідує їх через Pydantic-схеми з app/server/schemas.py, після чого передає роботу модулю бізнес-логіки app/server/task_manager.py, де зосереджені всі операції з задачами: додавання, призначення, збереження результату і обробка помилок.

Структура таблиці tasks охоплює всі потреби системи. Поле `task_id` зберігає унікальний UUID-ідентифікатор задачі. Поля `image_name` і `image_path` зберігають оригінальне ім’я файлу і повний шлях на диску майстра. Поле `image_hash` містить MD5-хеш вмісту файлу для дедуплікації: те саме зображення під різними іменами не створює дублікату. Поле `status` приймає чотири значення: `pending`, `in_progress`, `done`, `error`. Окремі поля зберігають метадані результату: `caption` з текстовим описом, `model_name` і `model_version` з ідентифікацією моделі, `processing_time` з часом обробки, `inference_params` з JSON-серіалізованими параметрами генерації. Часові мітки `created_at`, `assigned_at` і `finished_at` фіксують ключові моменти життєвого циклу задачі.

Найскладніша частина реалізації майстра пов’язана з атомарним призначенням задач воркерам. Без спеціальних заходів два воркери, що звертаються за задачею майже одночасно, можуть отримати один `task_id` і виконати дублюючу роботу. Класичним рішенням у реляційних базах є `SELECT ... FOR UPDATE SKIP LOCKED`. Проте SQLite не підтримує row-level locking, він блокує всю базу при запису. Це робить класичне рішення непридатним.

Замість блокування застосовано конструкцію UPDATE ... RETURNING, що з’явилась у SQLite починаючи з версії 3.35[19, 20]. Запит виконує пошук, оновлення статусу і повернення task_id як одну атомарну операцію на рівні рушія БД:

```
UPDATE tasks
SET status = ‘in_progress’, assigned_at = :now
WHERE task_id = (
    SELECT task_id FROM tasks
    WHERE status = ‘pending’
    ORDER BY created_at LIMIT 1
)
RETURNING task_id
```

При одночасних запитах SQLite виконає UPDATE серіально завдяки WAL-блокуванню на запис. Перший воркер отримає task_id першої вільної задачі, другий – наступної. Якщо вільних задач немає, RETURNING поверне порожній результат, і майстер відповість HTTP-кодом `204 (No Content)`. Цей механізм гарантує відсутність race condition без використання явних блокувань.

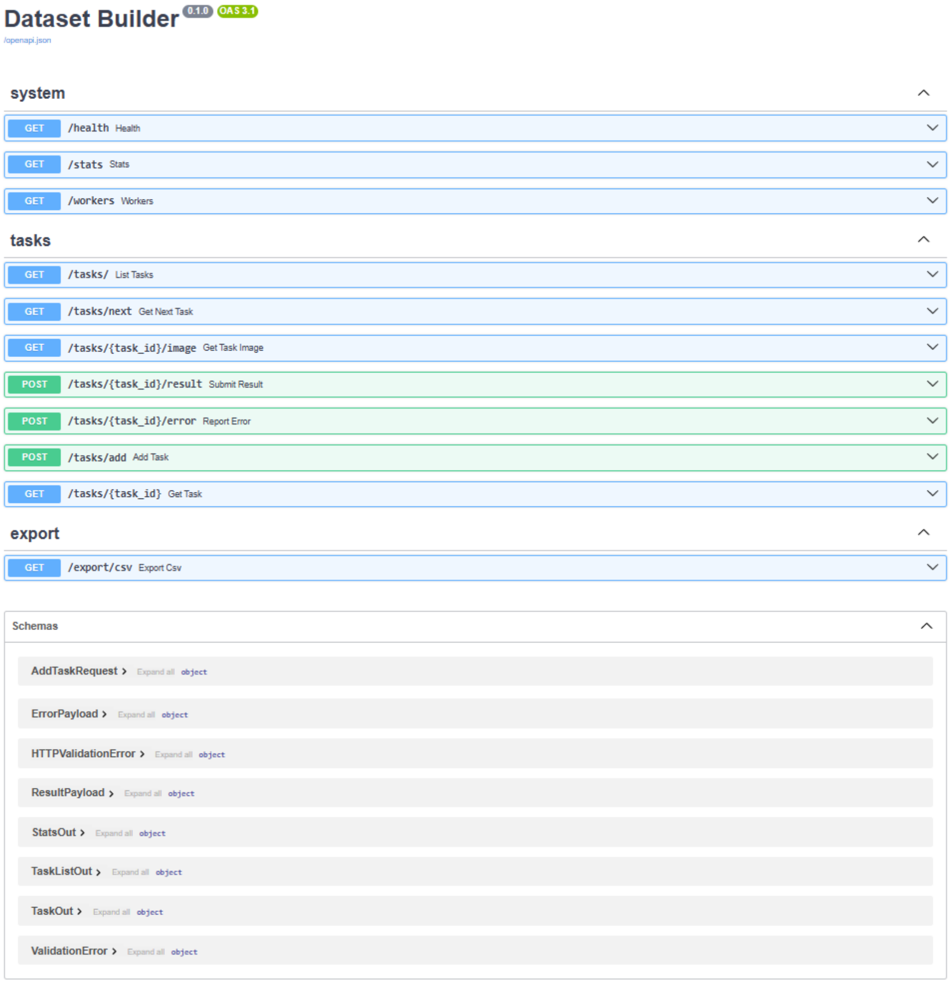
Обробку зависаючих задач реалізовано через механізм reclaim. Якщо воркер взяв задачу і впав посеред обробки, вона залишається у статусі `in_progress` назавжди. При кожному запиті `/tasks/next` сервер у фоновій задачі перевіряє всі задачі у статусі `in_progress` і повертає у `pending` ті, що знаходяться в обробці довше тайм-ауту (за замовчуванням 300 секунд). Кожна задача має лічильник `retry_count`, і після перевищення максимальної кількості повторів задача переводиться у статус `error` для ручного розгляду. Цей підхід дозволяє системі самовідновлюватися без втручання адміністратора.

API майстра складається з невеликого набору кінцевих точок. `GET /health` повертає статус сервера і використовується воркером перед початком роботи для перевірки доступності. `GET /stats` повертає агреговану статистику задач за статусами. `GET /workers` віддає таблицю активних воркерів з кількістю оброблених задач і середнім часом обробки. Ці дані використовуються графічним інтерфейсом для візуалізації навантаження. `GET /tasks/next` є найважливішим – через нього воркери отримують роботу. `GET /tasks/{id}/image` передає бінарний вміст зображення воркеру у вигляді потоку. `POST /tasks/{id}/result` приймає готовий опис із метаданими. `POST /tasks/{id}/error` обробляє повідомлення про збій. `GET /export/csv` віддає весь датасет у CSV-форматі для скачування.

Окремо реалізовано експорт у HTML-форматі. На відміну від CSV, що є технічним форматом для подальшої обробки, HTML призначений для людського перегляду. Звіт містить таблицю з мініатюрами всіх оброблених зображень, що вбудовані у файл через base64-кодування. Це робить HTML-звіт повністю автономним: один файл містить і дані, і зображення, і стилі оформлення. Звіт відкривається у будь-якому браузері без встановлення додаткових програм. Такий формат значно зручніший для презентації результатів замовнику або членам комісії, оскільки одразу видно, які описи були згенеровані для яких зображень.

Усі кінцеві точки описані як Python-функції з типізованими параметрами і Pydantic-схемами для тіл запитів. FastAPI автоматично виконує валідацію вхідних даних, генерує OpenAPI-специфікацію і створює інтерактивну документацію Swagger UI за адресою `/docs`. Це спрощує налагодження, дозволяючи перевірити будь-яку кінцеву точку прямо у браузері без написання окремого клієнта.

Усі кінцеві точки автоматично документуються FastAPI через стандарт OpenAPI 3.1. Інтерактивна документація відображає три логічні групи маршрутів: системні запити (health, stats, workers), операції з задачами (отримання, завантаження зображення, відправка результату) і експорт датасету. Окремо задокументовано Pydantic-схеми всіх запитів і відповідей. Приклад документації наведено на рисунку 2.1



Dataset Builder 0.1.0 OAS 3.1
/openapi.json

system
GET /health Health
GET /stats Stats
GET /workers Workers

tasks
GET /tasks/ List Tasks
GET /tasks/next Get Next Task
GET /tasks/{task_id}/image Get Task Image
POST /tasks/{task_id}/result Submit Result
POST /tasks/{task_id}/error Report Error
POST /tasks/add Add Task
GET /tasks/{task_id} Get Task

export
GET /export/csv Export Csv

Schemas
AddTaskRequest › Expand all object
ErrorPayload › Expand all object
HTTPValidationError › Expand all object
ResultPayload › Expand all object
StatsOut › Expand all object
TaskListOut › Expand all object
TaskOut › Expand all object
ValidationError › Expand all object

Рисунок 2.1 – Інтерактивна документація API майстра

Збереження зображень розділене між двома сховищами. Самі бінарні файли лежать у файловій системі майстра в папці `images/`, що задається через конфігурацію. Метадані задач і результати зберігаються у SQLite. Такий поділ обраний свідомо: реляційні бази погано працюють з великими бінарними даними, тоді як файлова система оптимізована саме для цього.

Режим WAL, у якому працює база майстра, заслуговує окремого пояснення, оскільки саме він робить можливою одночасну роботу кількох воркерів з одним файлом бази. У стандартному режимі rollback journal SQLite блокує всю базу на час будь-якого запису, через що читання і запис не можуть відбуватися паралельно. У режимі WAL нові зміни дописуються в окремий файл випереджального журналу (dataset.db-wal), а основний файл бази залишається доступним для читання. Завдяки цьому воркери можуть безперешкодно читати дані, наприклад отримувати статистику або перевіряти статус задач, у той самий момент, коли майстер записує черговий результат. Періодично відбувається контрольна точка (checkpoint), під час якої накопичені у журналі зміни переносяться в основний файл бази, а допоміжний файл спільної пам’яті (dataset.db-shm) координує доступ між процесами. Серіалізуються лише самі операції запису, що цілком прийнятно для прототипу з помірною кількістю воркерів.

Механізм дедуплікації за хешем безпосередньо впливає на якість підсумкового датасету. Для кожного доданого файлу обчислюється MD5-хеш його вмісту, який зберігається у полі image_hash. Перед створенням нової задачі майстер перевіряє, чи немає у базі задачі з таким самим хешем, і за наявності збігу пропускає файл. Такий підхід виявляє повторювані зображення навіть тоді, коли вони мають різні імена файлів, що типово для реальних наборів даних, зібраних з кількох джерел. Дедуплікація на рівні вмісту запобігає появі однакових пар «зображення – опис» у навчальному датасеті, які могли б спотворити подальше навчання моделей через надлишкову вагу повторюваних прикладів.

Конфігурація майстра повністю винесена у файл `.env`. Усі параметри сервера такі як: адреса прослуховування, порт, шлях до бази даних, тайм-аут обробки задач, максимальна кількість повторів читаються бібліотекою `pydantic-settings` автоматично при старті програми. При зміні параметрів достатньо відредагувати .env без перекомпіляції коду.

Ефективність такої реалізації перевірена на практиці. SQLite з WAL-режимом без проблем витримує одночасні запити від кількох воркерів. FastAPI з Uvicorn обробляє сотні запитів на секунду на звичайному ПК. Майстер як процес споживає менше 100 МБ оперативної пам’яті у спокої і не потребує GPU, що дозволяє розгорнути його на будь-якому офісному ПК.

2.3 Реалізація клієнтської частини

Клієнтська частина системи, що отримала умовну назву "воркер", виконує саму обчислювальну роботу, тобто обробку зображень через нейромережу і формування текстових описів. На відміну від майстра, воркер не приймає з’єднань ззовні і не має HTTP-сервера. Він є чистим клієнтом, що ініціює всі взаємодії з майстром у режимі pull: сам запитує задачі, сам надсилає результати, сам повідомляє про помилки. Така роль дозволяє воркеру працювати у будь-якій мережевій конфігурації, включно з мережами за NAT або firewall.

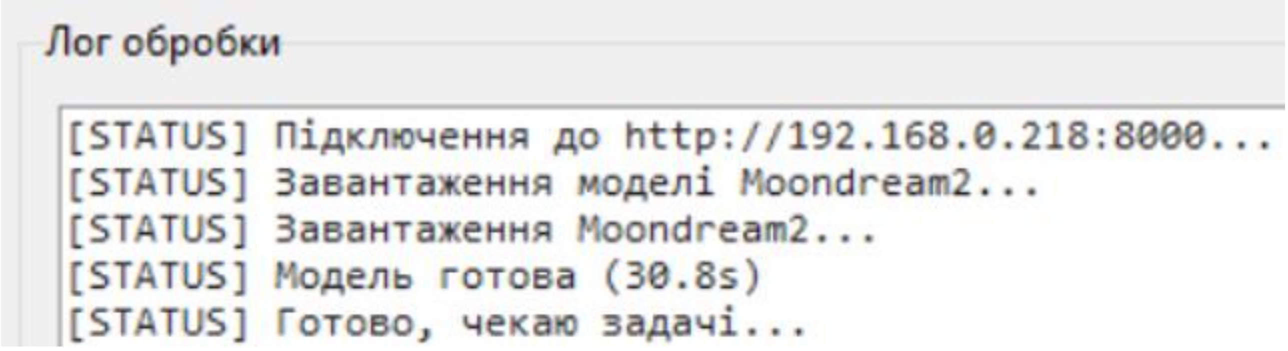
Реалізація воркера побудована з трьох основних компонентів. Мережевий клієнт `MasterClient` (файл `app/shared/http_client.py`) інкапсулює всю взаємодію з HTTP-API майстра. Сервіс інференсу `MoondreamService` (файл `app/worker/moondream_service.py`) відповідає за завантаження моделі і виконання обчислень. Ядро воркера `WorkerCore` (файл `app/worker/worker.py`) поєднує перших два у безперервний цикл обробки задач і надає callback-інтерфейс для графічного інтерфейсу. Такий поділ дозволяє тестувати кожен компонент окремо і замінювати їх без впливу на інші.

Життєвий цикл воркера починається з ініціалізації. При старті завантажується конфігурація з `.env` файлу, генерується унікальний ідентифікатор у форматі `{hostname}-{uuid}` (наприклад, `DESKTOP-CNAN4TJ-`

`95652ccd)`, визначається пристрій для обчислень. Далі виконується перевірка доступності майстра через `GET  /health`. Якщо майстер недоступний, воркер повідомляє про це і завершує роботу без спроб обробки. Цей крок захищає від ситуацій, коли воркер витрачає час на завантаження моделі, а потім виявляє, що працювати немає з ким.

Після успішного контакту з майстром починається завантаження моделі Moondream2. При першому запуску модель скачується з Hugging Face Hub і кешується у локальній папці користувача (`~/.cache/huggingface/`). Розмір моделі приблизно 3,7 ГБ. Усі наступні запуски використовують кешовану версію і завантажуються за 4-5 секунд. Завантаження моделі виконано як одноразова операція в конструкторі класу `MoondreamService`, завантажувати її кожного разу при обробці задачі було б неефективно.

Процес ініціалізації воркера відображається у лозі у реальному часі: від підключення до майстра через перевірку доступності до повідомлення про готовність моделі з часом завантаження. Приклад логу ініціалізації наведено на рисунку 2.2.



```
[STATUS] Підключення до http://192.168.0.218:8000...
[STATUS] Завантаження моделі Moondream2...
[STATUS] Завантаження Moondream2...
[STATUS] Модель готова (30.8s)
[STATUS] Готово, чекаю задачі...
```

Рисунок 2.2 – Лог ініціалізації воркера

Питання сумісності зі старою ревізією моделі вирішено через патчинг конфігурації. Moondream2 ревізії 2024-08-26 очікує наявності атрибута `pad_token_id`, який нові версії transformers не встановлюють автоматично. У коді реалізовано патч: перед створенням моделі завантажується об’єкт конфігурації, перевіряється наявність потрібних атрибутів і додаються відсутні. Аналогічний патч застосовано до вкладеного об’єкта `text_config`. Без цих патчів модель не запускається на сучасних версіях transformers.

Основний робочий цикл воркера побудований як нескінченний поллінг майстра. На кожній ітерації виконується запит `GET /tasks/next`. Якщо майстер повертає задачу, воркер обробляє її. Якщо повертається код `204`, то воркер засинає на заданий інтервал (3 секунди) і повторює запит. Інтервал обрано як компроміс: занадто короткий створив би зайве навантаження на майстра, занадто довгий уповільнив би обробку.

Після отримання `task_id` воркер завантажує бінарний вміст зображення через окрему кінцеву точку `GET /tasks/{id}/image`, що економить близько 33% трафіку порівняно з base64 у JSON. Файл декодується у `PIL Image`, конвертується у RGB і масштабується до 1536 пікселів по довшій стороні. Підготовлене зображення передається у метод `encode_image`, потім викликається answer_question із підказкою "Describe this image in detail.", і отриманий опис разом із метаданими відправляється на майстер через `POST /tasks/{id}/result`.

Обробка помилок реалізована на двох рівнях. На рівні мережевого клієнта застосовано механізм автоматичних повторів: при тимчасових збоях (`NetworkError`, `TimeoutException`) запит повторюється до трьох разів із зростаючою затримкою (2, 4, 6 секунд). Це робить воркер стійким до короткочасних обривів зв’язку у бездротових мережах. На рівні робочого циклу будь-яке необроблене виключення під час обробки задачі перехоплюється і надсилається на майстер через `POST /tasks/{id}/error`. Майстер повертає задачу у статус `pending` для повторного видавання. Цикл воркера після помилки не зупиняється, він просто переходить до наступної задачі.

Інтеграція з графічним інтерфейсом виконана через систему callback-функцій. Клас `WorkerCore` приймає у конструкторі п’ять callback-параметрів: `on_status`, `on_task_start`, `on_task_done`, `on_task_error`, `on_idle`.

Графічний інтерфейс передає у конструктор свої функції оновлення вікна, і воркер викликає їх у потрібні моменти. Ядро воркера не знає нічого про Tkinter, а інтерфейс не знає нічого про HTTP і нейромережі. Така архітектура спрощує тестування і дозволяє додати інші типи інтерфейсів у майбутньому.

Запуск моделі винесено в окремий потік через `threading.Thread`, щоб вікно інтерфейсу не зависало під час інференсу. Tkinter є однопоточною бібліотекою, тому будь-яка тривала операція в основному потоці заморожує вікно. Рішенням було виконувати модель у фоновому потоці, а оновлення інтерфейсу планувати через метод `root.after(0, callback)`, що безпечно переносить виклик у головний потік Tkinter.

Ефективність реалізованого воркера перевірена на практиці. Один воркер на сучасному CPU обробляє одне зображення за 20-30 секунд, що дає продуктивність приблизно 120-180 зображень на годину. При підключенні двох воркерів продуктивність зростає вдвічі без змін у коді. Споживання пам’яті після завантаження моделі стабілізується на рівні 4-5 ГБ. Жоден з компонентів воркера не прив’язаний до конкретної моделі, тому заміна Moondream2 на іншу VLM потребуватиме змін лише у класі `MoondreamService`.

2.4 Мережева взаємодія клієнта та сервера

Мережева взаємодія є визначальним компонентом розподіленої системи, оскільки через неї передається уся інформація між компонентами. Помилка у мережевому шарі призводить до втрачених задач, дублювання роботи або повного зависання системи. Через це вибір протоколу і його реалізація потребують окремого розгляду.

Як основний транспортний протокол обрано HTTP з REST-підходом. Цей вибір зумовлений трьома причинами. HTTP є стандартом де-факто для веб-сервісів і має широку підтримку у будь-якій мові, що залишає можливість замінити воркер на реалізацію іншою мовою без зміни майстра. HTTP працює поверх TCP, що забезпечує гарантовану доставку повідомлень. HTTP легко налагоджувати, оскільки будь-який запит можна перевірити через браузер, curl або Swagger UI.

Альтернативою розглядався протокол gRPC. Він використовує бінарний формат Protobuf замість текстового JSON, працює поверх HTTP/2 з мультиплексуванням потоків. У системах з інтенсивним навантаженням gRPC дає помітний приріст продуктивності. Проте для прототипу його переваги не виправдовують ускладнення розробки. JSON через HTTP залишається оптимальним вибором для систем з помірним навантаженням і пріоритетом простоти. gRPC залишається опцією для майбутнього розвитку.

Як модель взаємодії обрано pull-схему замість push. У push-моделі майстер сам надсилав би задачі воркерам через відкриті з’єднання. Це вимагало б, щоб кожен воркер мав публічну адресу, доступну майстру, що неможливо за NAT. У pull-моделі воркер сам ініціює всі з’єднання, а саме: запитує задачі, надсилає результати, повідомляє про помилки. Майстру не потрібно знати адреси воркерів, достатньо приймати вхідні запити.

Формат повідомлень побудований на JSON з UTF-8 кодуванням, що забезпечує підтримку текстів будь-якою мовою. Винятком є передача зображень оскільки вони передаються потоком як бінарні дані з MIME-типом `application/octet-stream`, без проміжного кодування у `base64`. Це економить близько 33% мережевого трафіку. Структури JSON-повідомлень формалізовано через Pydantic-схеми, що автоматично виконують валідацію на стороні сервера: некоректний запит повертає `HTTP 422` з вказівкою на проблемне поле, замість падіння обробника.

Послідовність обміну для обробки однієї задачі складається з чотирьох ключових кроків. Спершу воркер надсилає `GET /tasks/next`. Майстер атомарно призначає вільну задачу і повертає JSON з `task_id` і `image_name`, або код `204` якщо задач немає. Далі воркер надсилає `GET /tasks/{task_id}/image`. Майстер стримить файл у відповіді. Після завершення інференсу воркер надсилає `POST /tasks/{task_id}/result` з

JSON-тілом, що містить опис і метадані. У разі помилки замість результату воркер надсилає `POST /tasks/{task_id}/error`.

Надійність обміну забезпечена через механізм автоматичних повторів. У бездротових мережах короткочасні обриви з’єднання трапляються регулярно. Кожен запит виконується у циклі до трьох спроб. Між спробами вставляється затримка, що зростає лінійно (2, 4, 6 секунд). Повторюються тільки запити, що завершились мережевою помилкою або серверною помилкою `5xx`. Клієнтські помилки `4xx` не повторюються, оскільки повтор не змінить результату.

На стороні майстра впроваджено симетричний механізм відновлення, а саме reclaim зависаючих задач. Якщо воркер впав посеред обробки і не повернувся, його задача залишається у `in_progress`. При кожному запиті `/tasks/next` майстер у фоновій задачі перевіряє такі задачі і повертає у `pending` ті, що не оновлювались довше тайм-ауту. Поєднання клієнтських повторів і серверного reclaim гарантує, що жодна задача не буде втрачена через тимчасові проблеми.

Кожен HTTP-запит від воркера містить заголовок `X-Worker-ID` з унікальним ідентифікатором у форматі `{hostname}-{uuid_short}`. Завдяки цьому майстер відстежує активність кожного вузла без підтримки постійного з’єднання і без окремого механізму реєстрації. Воркер просто з’являється у системі з першим запитом і зникає коли перестає надсилати запити, тобто жодного handshake або сесійного управління не потрібно. Це також означає що воркер можна перезапустити у будь-який момент і він просто продовжить роботу з наступною вільною задачею

HTTP-статусні коди виконують роль частини протоколу обміну. Відповідь `200` підтверджує успішне виконання, `204` сигналізує про відсутність вільних задач і не є помилкою, `404` повідомляє про відсутність задачі з вказаним ідентифікатором, `422` вказує на некоректний формат даних. Така семантика дозволяє воркеру коректно реагувати на кожну ситуацію без додаткових полів у тілі відповіді.

Важливим аспектом мережевої взаємодії є керування таймаутами, оскільки саме невизначені зависання запитів є найпоширенішою причиною непомітних збоїв у мережевих системах. У реалізованому клієнті httpx таймаути задано роздільно для кожної фази з'єднання: таймаут установлення TCP-з'єднання (connect) обрано коротким (5 секунд), оскільки в межах локальної мережі з'єднання встановлюється майже миттєво, а його відсутність протягом цього часу однозначно свідчить про недоступність майстра. Таймаут читання відповіді (read) задано більшим, оскільки він має покривати передачу найбільшого з можливих тіл відповіді. Принциповим є те, що жоден окремий HTTP-запит не охоплює час інференсу: воркер спершу отримує задачу і зображення, виконує обчислення локально, і лише потім окремим запитом надсилає результат. Через це таймаути розраховуються виключно під мережеві операції, а не під тривалу роботу нейромережі. Найбільшим за обсягом запитом є завантаження зображення через GET /tasks/{id}/image, тому саме для нього передбачено найбільший запас часу, тоді як решта запитів оперує невеликими JSON-повідомленнями і завершується за частки секунди.

Передача зображень окремим бінарним потоком замість вбудовування у JSON має кількісно вимірюване обґрунтування. Кодування base64 представляє кожні три байти двійкових даних чотирма символами ASCII, що збільшує обсяг приблизно на третину. Для зображення розміром 2 МБ це означає близько 0,67 МБ зайвого трафіку, а на повному датасеті з сотень зображень накопичується істотний обсяг. Крім економії трафіку, потокова передача дає виграш у споживанні пам'яті: майстер віддає файл блоками через StreamingResponse, не завантажуючи його повністю в оперативну пам'ять і не утримуючи розширене base64-представлення. Воркер так само читає відповідь потоком і одразу декодує її у PIL Image. Таким чином споживання пам'яті при передачі залишається сталим незалежно від розміру зображення, що важливо для одночасного обслуговування кількох воркерів.

Поєднання клієнтських повторів і серверного reclaim створює ситуації, у яких один результат теоретично може надійти двічі. Якщо воркер обробляв задачу довше за тайм-аут reclaim, майстер повертає її у pending і видає іншому воркеру; коли ж перший воркер усе-таки завершує роботу із запізненням, він надсилає свій результат на ту саму задачу. Щоб такі випадки не порушували цілісність даних, операції майстра спроєктовано ідемпотентними щодо task_id: повторне надсилання результату для вже завершеної задачі не створює дубльованого рядка, а лише перезаписує метадані за принципом «останній запис перемагає». Аналогічно повторний запит GET /tasks/next ніколи не видає вже призначену задачу завдяки атомарному UPDATE...RETURNING. Тому єдиним наслідком рідкісного повторного опрацювання є марно витрачений процесорний час одного воркера, тоді як датасет залишається консистентним[44].

Реалізована мережева взаємодія повністю покриває потреби системи без надмірної складності. HTTP/REST з JSON забезпечує універсальність і легкість налагодження, pull-модель спрощує мережеву конфігурацію, повтори і reclaim гарантують надійність, ідентифікація воркерів дає видимість стану системи. Усі ці рішення приймались з розрахунком на просте розширення, де додати новий тип кінцевої точки або замінити протокол можна без зміни загальної логіки обміну.

2.5 Загальна архітектура та узагальнення системи

Розглянуті у попередніх підрозділах компоненти серверної та клієнтської частини та мережева взаємодія у сукупності утворюють єдину систему з чітко визначеною архітектурою. Загальна структура відповідає класичній моделі майстер-воркер з опорою на ідеї edge-обчислень. Майстер виступає координатором, що тримає в собі стан усіх задач. Воркери є незалежними обчислювальними вузлами, що виконують реальну обробку зображень. Зв’язок здійснюється виключно через HTTP-протокол, що робить систему слабко зв’язаною: будь-який компонент можна замінити, оновити або перезапустити без впливу на інші.

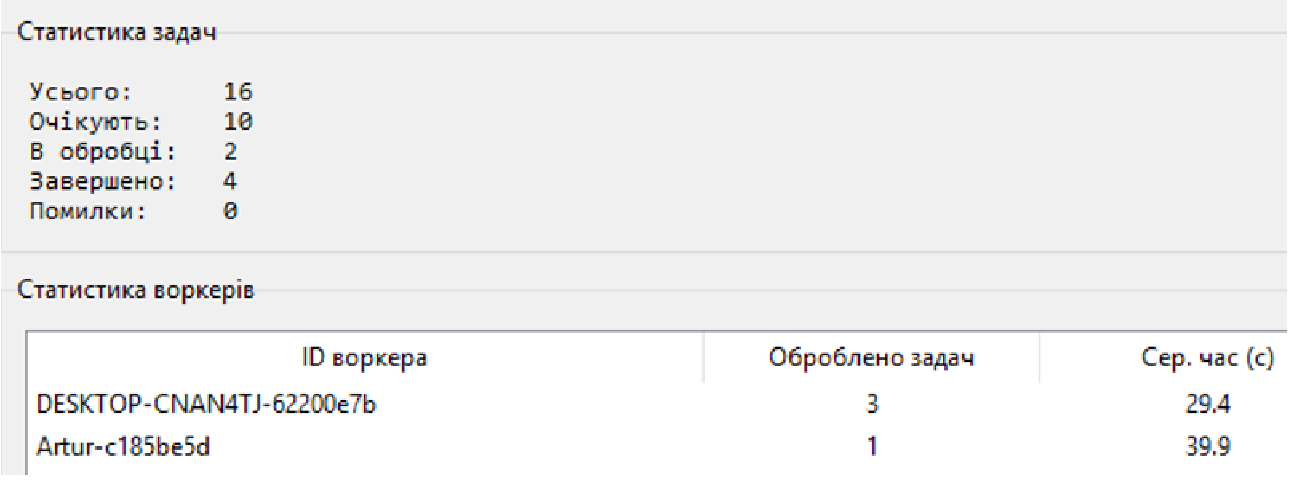
Фізично система розгортається на двох або більше комп’ютерах у локальній мережі. На одному з них запускається майстер, це може бути звичайний робочий ПК без особливих вимог до продуктивності. На решті комп’ютерів запускаються воркери, для них найважливішим є об’єм оперативної пам’яті (мінімум 6 ГБ) і продуктивність CPU. Мережа має бути локальною з пропускною здатністю від 100 Мбіт/с. Кількість воркерів не обмежена архітектурно: можна підключити два, п’ять або десять, лише пропорційно зростатиме сумарна продуктивність.

Запуск системи у штатному режимі починається на вузлі майстра. Користувач запускає графічну оболонку через файл `start.bat`, яка відкриває стартове вікно з вибором режиму. Після натискання кнопки "Майстер" відкривається головне вікно, де відображається IP-адреса для воркерів і кнопка запуску сервера. При запуску сервера автоматично виконується ініціалізація: створюється папка для зображень, відкривається SQLite-база, активується режим `WAL`. Uvicorn-сервер стартує у фоновому потоці на заданому порту, після чого вікно майстра показує статус "Працює" і починає періодично оновлювати статистику задач.

На вузлах воркерів процедура запуску дещо інша. Користувач так само запускає `start.bat` і обирає режим "Воркер" у стартовому вікні. Відкривається вікно воркера з полем введення IP-адреси майстра. Після введення коректної адреси і натискання кнопки "Запустити воркер" відбувається багатоетапна ініціалізація: перевірка доступності майстра, завантаження моделі Moondream2 (при першому запуску 5-15 хвилин, надалі 4-5 секунд), перехід у робочий цикл.

Потік даних у системі під час нормальної роботи виглядає наступним чином. На стороні майстра користувач додає зображення через інтерфейс. Файли копіюються у папку `images/` на диску майстра, для кожного обчислюється `MD5`-хеш для дедуплікації, у базу записуються нові рядки зі статусом `pending`. Вільні воркери у режимі поллінгу при наступному запиті отримують задачі через атомарну операцію `UPDATE ... RETURNING`. Кожен воркер завантажує своє зображення, виконує інференс і повертає текстовий опис. Майстер зберігає результат і змінює статус на `done`. Коли всі задачі завершені, користувач отримує датасет через експорт у форматі CSV (для технічної обробки) або HTML-звіт з мініатюрами (для перегляду людиною).

Стан системи під час активної обробки відображається у вікні майстра в реальному часі. На рисунку 2.3 показано момент коли два воркери одночасно обробляють зображення: перший вузол з ідентифікатором DESKTOP-CNAN4TJ вже завершив 3 задачі із середнім часом 29,4 секунди, другий вузол artur обробив 1 задачу за 39,9 секунди. Статистика задач у верхній частині вікна показує що з 16 загальних задач 4 вже завершено, 2 знаходяться в обробці і 10 очікують у черзі. Помилок немає. Така інформаційна панель дозволяє оператору бачити реальний стан системи без звернення до бази даних або логів.



```
Усього:      16
Очікують:    10
В обробці:   2
Завершено:   4
Помилки:     0
```

ID воркера	Оброблено задач	Сер. час (с)
DESKTOP-CNAN4TJ-62200e7b	3	29.4
Artur-c185be5d	1	39.9

Рисунок 2.3 – Вікно майстра під час розподіленої обробки зображень

Поведінка системи при підключенні кількох воркерів одночасно демонструє ключову перевагу архітектури. Завдяки атомарному призначенню задач на рівні бази даних, два воркери ніколи не отримують одну й ту саму задачу. Якщо один воркер швидший за інший, він просто візьме більше задач за одиницю часу, що автоматично балансує навантаження. Не потрібен жоден додатковий механізм розподілу: SQLite з UPDATE ... RETURNING гарантує коректність на рівні рушія БД. Якщо у системі є 100 зображень і два воркери з продуктивністю 25 і 30 секунд на зображення, то приблизно через 22 хвилини обидва закінчать роботу, обробивши 55 і 45 задач відповідно. Розподіл виходить природно, без явного планувальника.

Стійкість до збоїв забезпечена на кількох рівнях. Якщо воркер втрачає зв’язок з майстром на короткий час, то мережевий клієнт автоматично повторює запит. Якщо воркер падає або зависає посеред обробки тоді механізм reclaim на майстрі через 5 хвилин повертає задачу у pending. Якщо воркер зіткнувся з винятковою ситуацією при обробці, він надсилає `POST /tasks/{id}/error`, і задача також повертається у `pending`. Кожна задача має лічильник повторів, і після перевищення ліміту вона переводиться у статус `error` для ручного аналізу.

Реалізована система демонструє основні якості розподіленого програмного забезпечення. Масштабованість досягнута горизонтально: кожен новий воркер пропорційно збільшує продуктивність без змін у коді. Відмовостійкість забезпечена комбінацією клієнтських повторів і серверного reclaim. Незалежність компонентів і прозорість роботи через графічний інтерфейс завершують картину системи придатної до практичного використання.

З точки зору супроводження системи кожен компонент можна оновлювати незалежно. Заміна версії моделі Moondream2 потребує змін лише у файлі `moondream_service.py` на стороні воркера без жодних змін на майстрі. Додавання нової кінцевої точки на майстрі не вимагає оновлення воркерів, якщо вони не використовують цю кінцеву точку.

Розгортання системи на новому вузлі зведено до мінімуму кроків: встановлення Python, копіювання папки проєкту, встановлення залежностей через requirements.txt і введення IP-адреси майстра у файлі .env. Жодних додаткових налаштувань мережі, реєстрації вузла або синхронізації конфігурацій між комп’ютерами не потрібно. Для виробничого підприємства це означає можливість одночасно задіяти 5-10 машин для автоматичної анотації зображень за лічені години замість тижнів ручної розмітки. Архітектура залишає простір для подальшого розвитку: підключення GPU-сервера, контейнеризація через Docker, перехід на PostgreSQL при масштабуванні.

Додавання нового воркера під час активної обробки не потребує зупинки системи і є стандартним сценарієм використання. Воркер при запуску перевіряє доступність майстра, завантажує модель і без будь-яких додаткових узгоджень починає запитувати вільні задачі з тієї точки, де зупинилась черга. З погляду майстра новий вузол нічим не відрізняється від вже активних і просто з'являється у таблиці статистики з першим надісланим результатом. Аналогічно відключення одного з воркерів під час обробки не призводить до втрати задач, оскільки механізм відновлення повертає незавершені задачі у чергу очікування автоматично.

Повнота збережених метаданих забезпечує відтворюваність датасету при майбутніх змінах. Кожен запис у базі містить назву та ревізію моделі, ідентифікатор вузла, час обробки і параметри генерації. Якщо модель оновлюється, адміністратор може скинути потрібні задачі у стан очікування і отримати нові описи без повторного завантаження зображень.

Модульний устрій проєкту спрощує внесення змін без ризику зачепити непов'язані частини. Код розбито на три директорії: серверна частина, клієнтська частина і спільні компоненти. Заміна поточної зорово-мовної моделі на іншу потребує змін лише в одному файлі клієнтської частини, тоді як серверна частина, мережевий клієнт і графічний інтерфейс залишаються недоторканими.

Три напрями розвитку, згадані вище, зводяться до точкових змін. Підтримка відеокарти вимагає змінити лише логіку визначення пристрою, після чого модель автоматично завантажиться у режимі зниженої точності. Перехід на PostgreSQL зводиться до зміни рядка підключення у файлі налаштувань, оскільки об'єктно-реляційний відображувач SQLAlchemy однаково сумісний з обома рушіями.

3 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ РОЗРОБЛЕНОЇ СИСТЕМИ

3.1 Розподіл задач та механізми балансування навантаження

Розподіл задач у розподілених системах традиційно реалізується через два підходи: push-модель і pull-модель. У push-моделі центральний координатор сам надсилає задачі воркерам через відкриті з'єднання або черги повідомлень. Такий підхід вимагає щоб кожен воркер мав адресу, доступну координатору, і підтримував постійне з'єднання. У pull-моделі ініціативу проявляє воркер: він сам звертається до сервера, запитує наступну задачу, обробляє її і повертає результат. Майстру не потрібно знати адреси воркерів і підтримувати з'єднання з ними. Достатньо приймати вхідні запити і відповідати на них[44].

У реалізованій системі обрано pull-модель з кількох причин. Перша причина полягає у простоті мережевої конфігурації. Воркер може запускатись у будь-якій мережевій інфраструктурі, навіть за NAT або корпоративним firewall, оскільки сам ініціює всі з'єднання. Друга причина пов'язана зі стійкістю до збоїв воркерів. Якщо воркер падає, він просто перестає робити запити, без жодного впливу на майстер. У push-моделі довелось би реалізовувати окремий механізм виявлення мертвих з'єднань. Третя причина – простота балансування навантаження. Швидші воркери природно отримують більше задач, оскільки частіше їх запитують. Жоден явний планувальник не потрібен.

Найскладнішою технічною проблемою у pull-моделі є атомарність призначення задач. Розглянемо ситуацію коли два воркери одночасно надсилають запит `GET /tasks/next`. Без спеціальних заходів обидва воркери можуть прочитати з бази однакову вільну задачу і почати її обробляти паралельно. Це призводить до `race condition`: один і той самий `task_id` обробляється двічі, а отримані результати конфліктують. Виявлення такого збою на практиці утруднене, оскільки система продовжує працювати, але дані стають неконсистентними.

Класичним рішенням race condition у реляційних базах є конструкція `SELECT ... FOR UPDATE SKIP LOCKED`, що блокує рядок на час транзакції і пропускає вже заблоковані рядки. Цей підхід працює у PostgreSQL, MySQL InnoDB та інших базах з підтримкою `row-level locking`. Проте SQLite, обраний як рушій бази у поточній системі, не підтримує row-level locking за архітектурними особливостями. Замість блокування окремих рядків SQLite блокує всю базу при операції запису через механізм WAL. Це робить класичний підхід SELECT FOR UPDATE непридатним.

Рішення цієї проблеми детально описано у підрозділі 2.2. Конструкція UPDATE...RETURNING виконує пошук, оновлення статусу і повернення `task_id` як одну атомарну операцію, що гарантує відсутність race condition. При одночасних запитах від двох воркерів механізм WAL у SQLite серіалізує операції запису: перший воркер отримує першу вільну задачу, другий – наступну. Якщо вільних задач немає, майстер відповідає HTTP-кодом `204 No Content` і воркер повторює запит через 3 секунди.

Балансування навантаження у розробленій системі реалізовано без явного планувальника. Замість централізованого механізму, що приймає рішення про розподіл задач, використовується природна динаміка pull-моделі. Швидший воркер обробляє задачу за менший час і раніше повертається за наступною. Повільніший воркер ще працює над попередньою задачею і не запитує нову. Внаслідок цього швидший вузол природно отримує більше задач за одиницю часу, пропорційно до своєї продуктивності.

Цей підхід має кілька переваг перед класичним явним балансуванням. По-перше, не потрібна попередня інформація про продуктивність кожного воркера: система сама виявляє швидкість через реальну поведінку. По-друге, при зміні умов (наприклад, на повільнішому ноутбуці запустили іншу важку програму) розподіл автоматично коригується: воркер просто починає рідше запитувати задачі. По-третє, додавання нових воркерів не потребує жодної зміни у конфігурації майстра. Новий воркер просто запускається, починає запитувати задачі і автоматично інтегрується у систему[45].

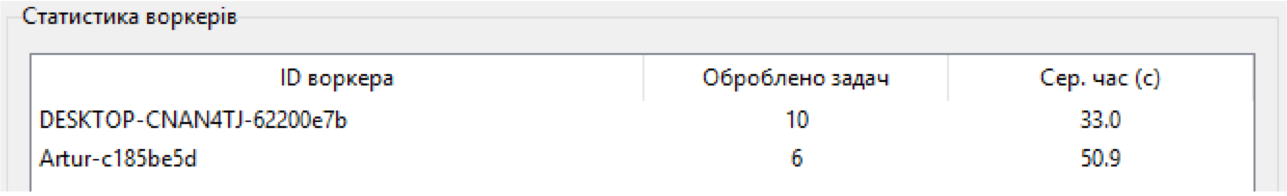
Механізм відновлення після збоїв `reclaim` забезпечує стійкість системи при падінні воркерів. Без додаткових заходів задача, взята воркером на обробку, може залишитись у статусі `in_progress` назавжди, якщо воркер впав посеред роботи. Така задача блокує відповідне зображення від повторної обробки і поступово зменшує ефективну продуктивність системи. Для розв’язання цієї проблеми майстер при кожному запиті `GET /tasks/next` запускає фонову перевірку всіх задач у статусі in_progress. Якщо задача знаходиться в обробці довше тайм-ауту (за замовчуванням 300 секунд), її статус повертається у pending і вона стає доступною іншому воркеру.

Кожна задача має лічильник повторів `retry_count`. При кожному поверненні задачі у `pending` через `reclaim` або через помилку обробки лічильник збільшується на одиницю. Якщо `retry_count` перевищує максимально допустиму кількість (за замовчуванням 3), задача переводиться у термінальний статус `error` для ручного аналізу. Цей механізм захищає систему від нескінченних повторів обробки явно некоректних файлів, наприклад пошкоджених зображень або файлів у непідтримуваних форматах.

Ідентифікація воркерів реалізована через HTTP-заголовок `X-Worker-ID`, що передається у кожному запиті. Ідентифікатор має формат `{hostname}-{uuid_short}`, наприклад `DESKTOP-CNAN4TJ-edfe1a22`. Перша частина дає людиночитану інформацію про комп’ютер, друга гарантує унікальність навіть при кількох воркерах на одному хості. Ідентифікатор зберігається у базі разом з кожним результатом обробки. Завдяки цьому майстер веде статистику по кожному воркеру: кількість оброблених задач, середній час обробки одного зображення, час останньої активності. Ці дані відображаються у таблиці статистики у вікні майстра і дозволяють спостерігати за роботою кожного вузла в реальному часі.

Практичне тестування механізму розподілу виконано з використанням двох ноутбуків різної конфігурації, підключених через локальну мережу. Перший воркер запущено на Lenovo Legion 5 з процесором AMD Ryzen 7 5800H (8 ядер, 16 потоків). Другий воркер працював на Acer Nitro 5 з процесором AMD Ryzen 5 5600H (6 ядер, 12 потоків). Обидва воркери одночасно підключались до одного майстра, що працював на Legion 5 паралельно з першим воркером.

Результат тестування підтвердив коректну роботу механізму розподілу. Усі 16 задач були оброблені без помилок і без дублювання. Перший воркер на Legion 5 обробив 10 задач із середнім часом 33 секунди на одну задачу. Другий воркер на Nitro 5 обробив 6 задач із середнім часом 51 секунда. Скріншот статистики воркерів у вікні майстра наведений на рисунку 3.1.



Статистика воркерів

ID воркера	Оброблено задач	Сер. час (с)
DESKTOP-CNAN4TJ-62200e7b	10	33.0
Artur-c185be5d	6	50.9

Рисунок 3.1 – Таблиця статистики воркерів у вікні майстра

Таблиця статистики у вікні майстра є наочним підтвердженням коректної роботи розподіленого механізму. Два окремі рядки з різними ідентифікаторами і різною кількістю оброблених задач свідчать про те що кожен воркер отримував власні унікальні задачі без жодного перетину. Детальний опис інтерфейсу майстра і таблиці статистики наведено у підрозділі 3.2.

Розподіл 10/6 є прямим наслідком різниці у продуктивності двох вузлів. Legion 5 з 8-ядерним процесором виконував інференс приблизно у 1,5 раза швидше за Nitro 5 з 6-ядерним процесором. Відношення задач 10:6 (1,67) близьке до відношення продуктивностей (51 ÷ 33 ≈ 1,55). Незначне відхилення пояснюється тим, що Legion 5 одночасно обслуговував і майстер-сервер, що створювало невелике додаткове навантаження на його CPU. У сценарії з виділеним майстром на окремій машині відношення задач було б ще точніше пропорційне продуктивностям.

Цей результат підтверджує ключову властивість pull-архітектури: розподіл навантаження виникає природно, без явного планувальника або налаштування. Якби до системи додали третій воркер, наприклад ще один Legion 5, він би автоматично почав отримувати приблизно стільки ж задач скільки перший. Якби один з воркерів став повільнішим через інше навантаження на CPU, його частка задач автоматично зменшилась би. Жодної ручної конфігурації не потрібно.

Поточна реалізація має кілька обмежень, які варто враховувати при подальшому розвитку системи. Перше обмеження пов’язане з масштабуванням SQLite. При кількості одночасних воркерів понад 20-30 серіалізація операцій запису через WAL може стати вузьким місцем. Для більших обсягів доцільно перейти на PostgreSQL, що підтримує справжнє row-level locking. Друге обмеження стосується `SPOF` (single point of failure): майстер є єдиним координатором, і його падіння зупиняє всю систему. Розв’язання цієї проблеми потребує реплікації бази даних і механізму вибору лідера, що значно ускладнить архітектуру. Третє обмеження пов’язане з відсутністю пріоритизації задач: усі задачі обробляються у порядку додавання (`FIFO`). При потребі обробляти термінові задачі швидше доцільно додати поле `priority` до таблиці і модифікувати `SELECT` у запиті призначення.

Реалізований механізм розподілу задач забезпечує коректну роботу системи з кількома воркерами, природне балансування навантаження пропорційно до продуктивності кожного вузла, стійкість до збоїв через reclaim і простоту масштабування через додавання нових воркерів. Поєднання атомарного `UPDATE...RETURNING` з pull-моделлю дає компактне і ефективне рішення, що задовольняє поставлені вимоги без використання важкої серверної інфраструктури.

3.2 Принцип роботи системи

Демонстрація принципу роботи системи дозволяє показати її практичне застосування у вигляді послідовності дій, які виконує користувач для формування навчального датасету. Розділ побудований навколо реальних знімків екрана, що ілюструють кожен етап від запуску програми до отримання готового результату. Послідовність викладу відповідає типовому сценарію використання системи у локальній мережі з двома комп'ютерами: один з них виконує роль майстра, інший – роль воркера.

Запуск програми відбувається через файл start.bat подвійним кліком, після чого з'являється стартове вікно з вибором режиму роботи. Це вікно є єдиною точкою входу до системи незалежно від того, який саме режим буде обрано на конкретному комп'ютері. Інтерфейс стартового меню показаний на рисунку 3.2.

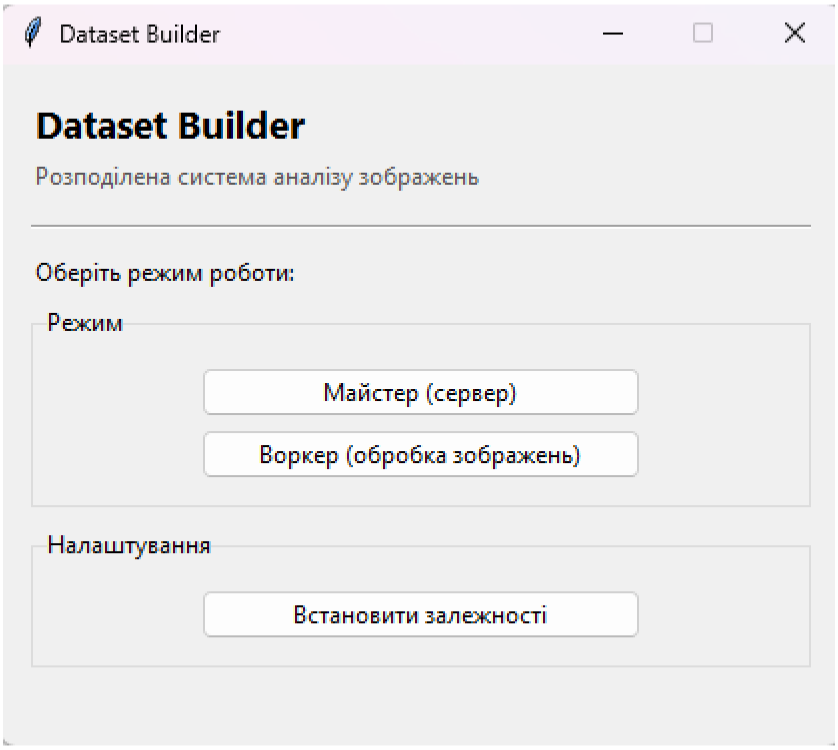
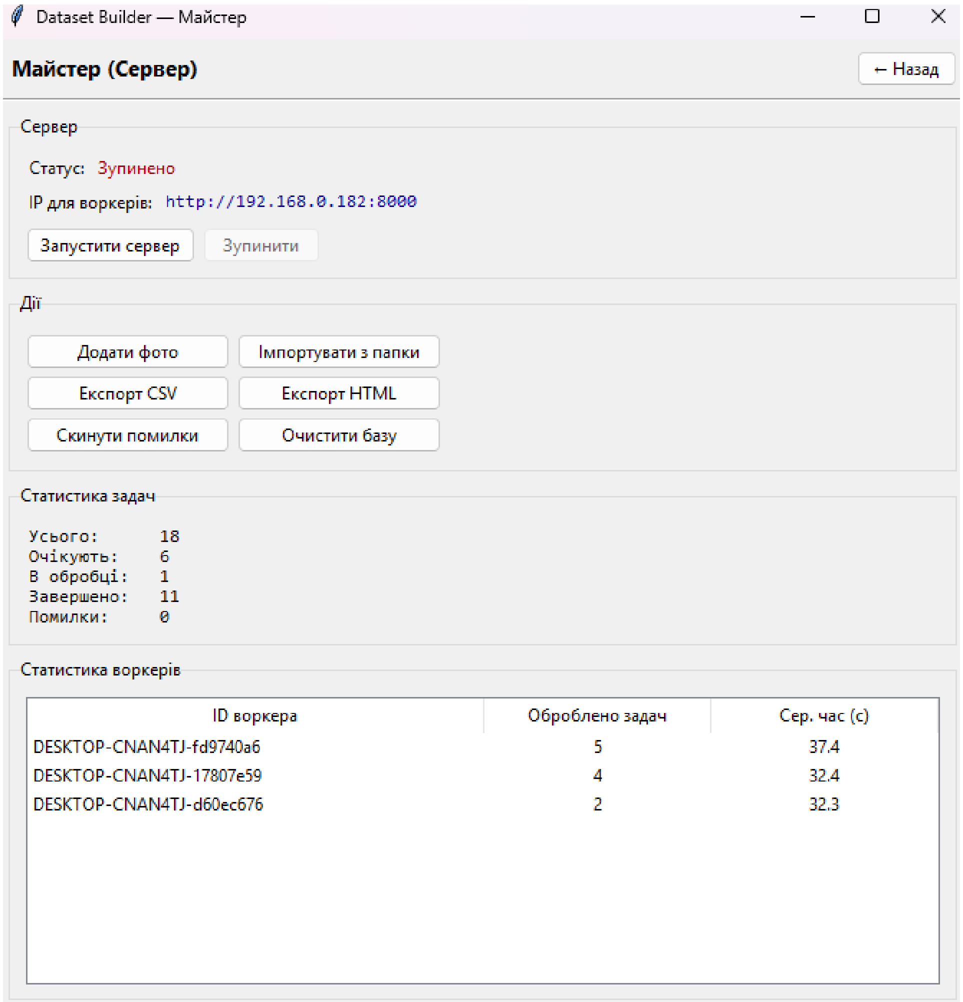


Рисунок 3.2 – Стартове меню програми

Стартове меню містить три основні елементи. Кнопка "Майстер (сервер)" запускає вікно центрального вузла системи, відповідального за зберігання задач і координацію роботи. Кнопка "Воркер (обробка зображень)" відкриває вікно клієнтської частини, що виконує інференс моделі. Кнопка "Встановити залежності" викликає допоміжне вікно для автоматичного встановлення Python-бібліотек з файлу requirements.txt. Останній варіант використовується лише при першому запуску системи на новому комп'ютері.

Після натискання кнопки "Майстер" стартове вікно закривається і відкривається головне вікно майстра. Воно містить декілька функціональних блоків: статус сервера з IP-адресою, кнопки управління, групу дій з даними, статистику задач і таблицю активних воркерів. Загальний вигляд вікна майстра у початковому стані показано на рисунку 3.3.



Статус: Зупинено

IP для воркерів: http://192.168.0.182:8000

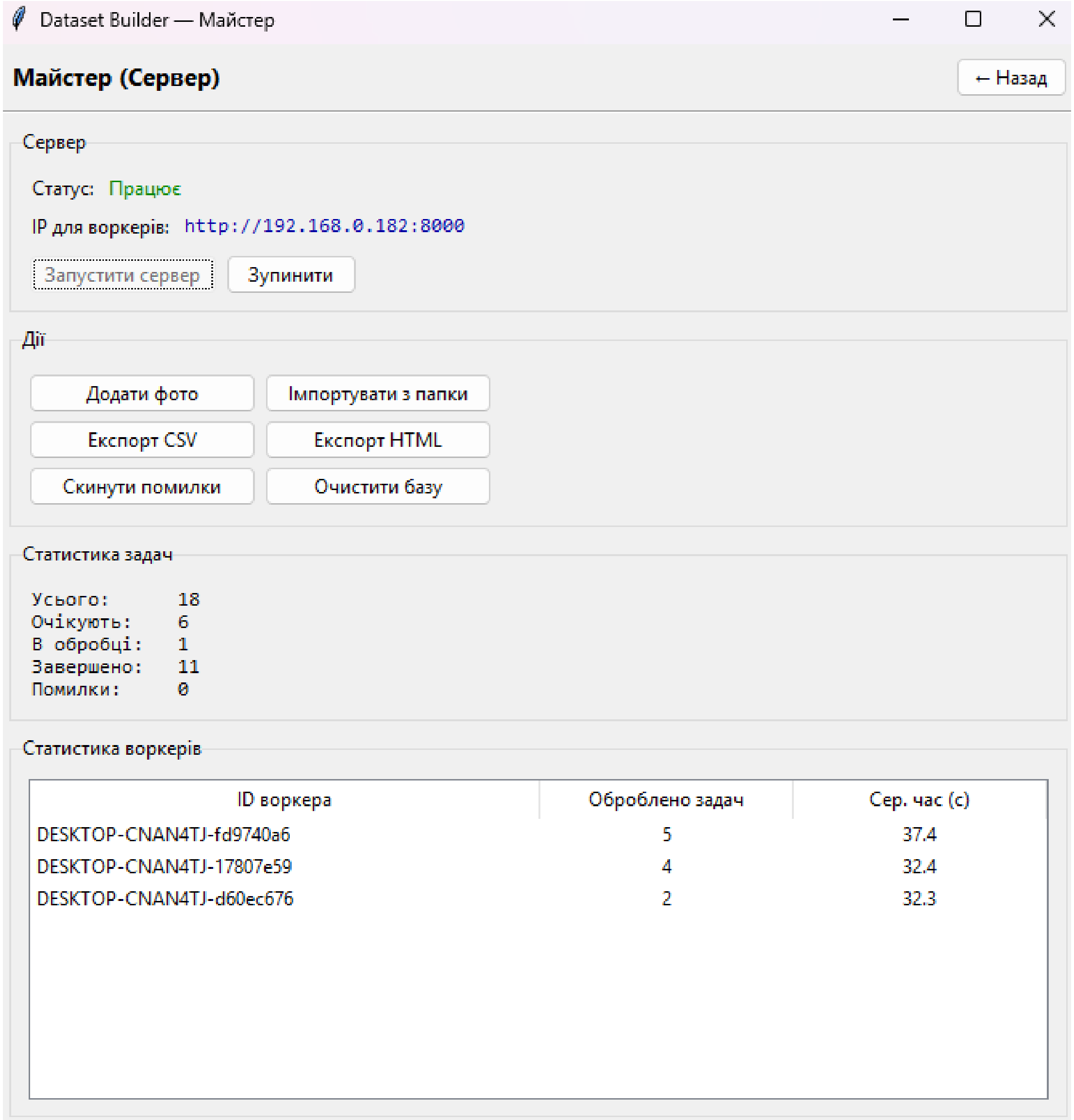
```
Усього:      18
Очікують:    6
В обробці:   1
Завершено:   11
Помилки:     0
```

ID воркера	Оброблено задач	Сер. час (с)
DESKTOP-CNAN4TJ-fd9740a6	5	37.4
DESKTOP-CNAN4TJ-17807e59	4	32.4
DESKTOP-CNAN4TJ-d60ec676	2	32.3

Рисунок 3.3 – Головне вікно майстра у початковому стані

Перед початком роботи майстер знаходиться у стані "Зупинено", а IP-адреса для воркерів автоматично визначається при відкритті вікна. У наведеному прикладі IP-адреса має формат http://192.168.x.x:8000, де перша частина – це адреса комп’ютера у локальній мережі, а 8000 – номер порту за замовчуванням. Саме цю адресу необхідно ввести на воркері для встановлення з’єднання.

Запуск сервера виконується натисканням кнопки "Запустити сервер". Після натискання статус змінюється на "Працює", а кнопка "Зупинити" стає активною. З цього моменту майстер починає приймати запити від воркерів. Стан вікна після успішного запуску сервера показано на рисунку 3.4.



Статус: Працює

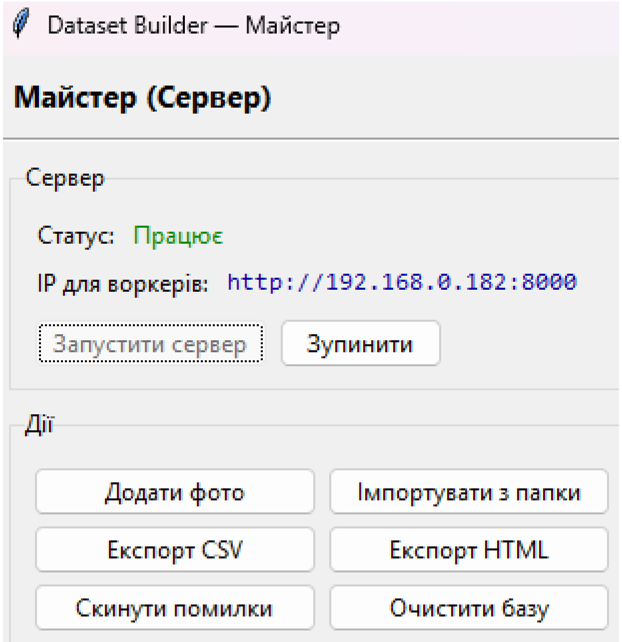
IP для воркерів: `http://192.168.0.182:8000`

```
Усього:      18
Очікують:    6
В обробці:   1
Завершено:   11
Помилки:     0
```

ID воркера	Оброблено задач	Сер. час (с)
DESKTOP-CNAN4TJ-fd9740a6	5	37.4
DESKTOP-CNAN4TJ-17807e59	4	32.4
DESKTOP-CNAN4TJ-d60ec676	2	32.3

Рисунок 3.4 – Вікно майстра з запущеним сервером

Група функцій майстра охоплює всі основні дії з даними. Кнопка "Додати фото" відкриває стандартний діалог Windows для вибору файлів, після чого вибрані зображення копіюються у папку images і додаються до бази задач. Кнопка "Імпортувати з папки" сканує існуючий вміст папки images і додає всі знайдені зображення як нові задачі без копіювання файлів. Дві наступні кнопки відповідають за експорт результатів у форматах CSV і HTML. Кнопки "Скинути помилки" і "Очистити базу" виконують службові операції з даними. Інтерфейс групи функцій майстра представлено на рисунку 3.5.

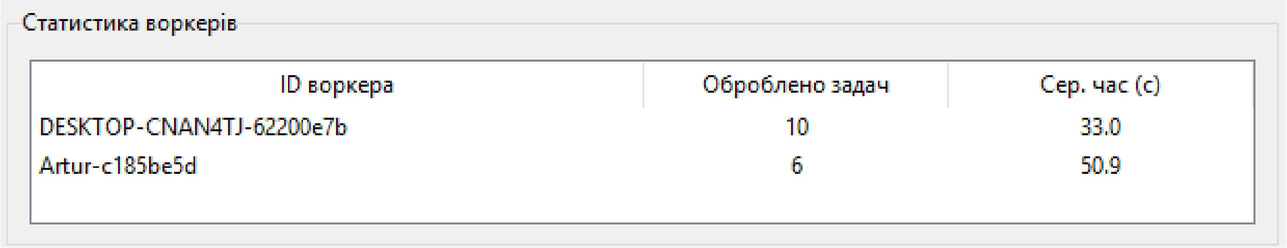


Статус: Працює

IP для воркерів: http://192.168.0.182:8000

Рисунок 3.5 – Група функцій майстра

Після додавання зображень до системи у нижній частині вікна майстра з’являється статистика задач: кількість задач у кожному статусі і таблиця активних воркерів. Статистика оновлюється автоматично кожні 2 секунди, що дозволяє спостерігати за роботою системи в реальному часі. У таблиці воркерів відображається ідентифікатор кожного підключеного воркера, кількість оброблених задач і середній час обробки одного зображення. Приклад вікна майстра з активними задачами і підключеним воркером показано на рисунку 3.6.

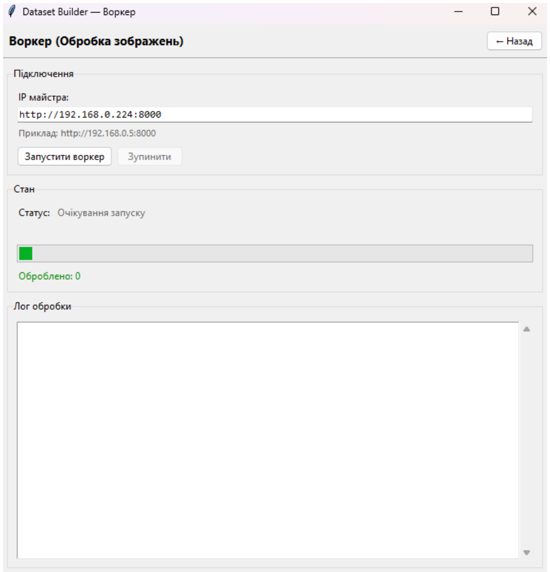

Статистика воркерів

ID воркера	Оброблено задач	Сер. час (с)
DESKTOP-CNAN4TJ-62200e7b	10	33.0
Artur-c185be5d	6	50.9

Рисунок 3.6 – Статистика і моніторинг активних воркерів

Особливість таблиці воркерів полягає у тому, що вона дає наочне підтвердження коректної роботи розподіленого механізму. Якщо до системи підключено два воркери, таблиця показує два окремі рядки з різними ідентифікаторами і різною кількістю оброблених задач. Сума оброблених задач завжди дорівнює загальній кількості задач у статусі done, що підтверджує відсутність дублювання роботи.

Запуск воркера здійснюється на другому комп’ютері за тією самою процедурою: подвійний клік на start.bat, у стартовому вікні обирається режим "Воркер". Відкривається вікно воркера, що містить групу підключення з полем для введення IP-адреси майстра, групу стану з прогрес-баром та лічильником, а також велике текстове поле для виведення логу обробки. Початковий вигляд вікна воркера наведено на рисунку 3.7.


Dataset Builder — Воркер

Воркер (Обробка зображень)

IP майстра: http://192.168.0.224:8000

Приклад: http://192.168.0.5:8000

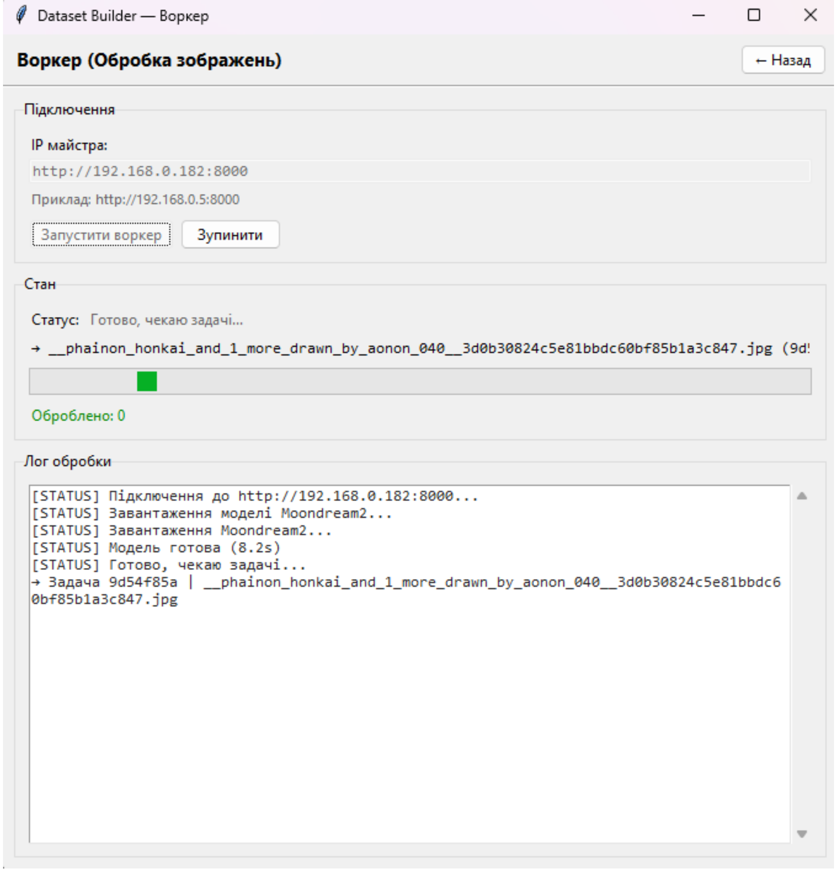
Статус: Очікування запуску

Оброблено: 0

Рисунок 3.7 – Головне вікно воркера у початковому стані

У поле "IP майстра" вводиться адреса, отримана з вікна майстра на першому комп’ютері. Після натискання кнопки "Запустити воркер" система виконує послідовність дій: перевірку доступності майстра, завантаження моделі Moondream2 у оперативну пам’ять, перехід у режим запиту задач. Перший запуск передбачає завантаження ваг моделі з Hugging Face Hub (близько 3,7 ГБ), що займає 5-15 хвилин залежно від швидкості інтернету. Усі наступні запуски використовують локальний кеш і завершуються за 4-5 секунд.

Після завершення ініціалізації воркер автоматично починає запитувати задачі у майстра і виконувати обробку зображень. Процес обробки візуалізується у вікні воркера через прогрес-бар, що рухається під час інференсу, та через лог, де відображається кожна оброблена задача з її ідентифікатором, часом обробки і коротким фрагментом згенерованого опису. Активний процес обробки задач показано на рисунку 3.8.



IP майстра: http://192.168.0.182:8000

Приклад: http://192.168.0.5:8000

Статус: Готово, чекаю задачі...

→ __phainon_honkai_and_1_more_drawn_by_aonon_040__3d0b30824c5e81bbdc60bf85b1a3c847.jpg (9d

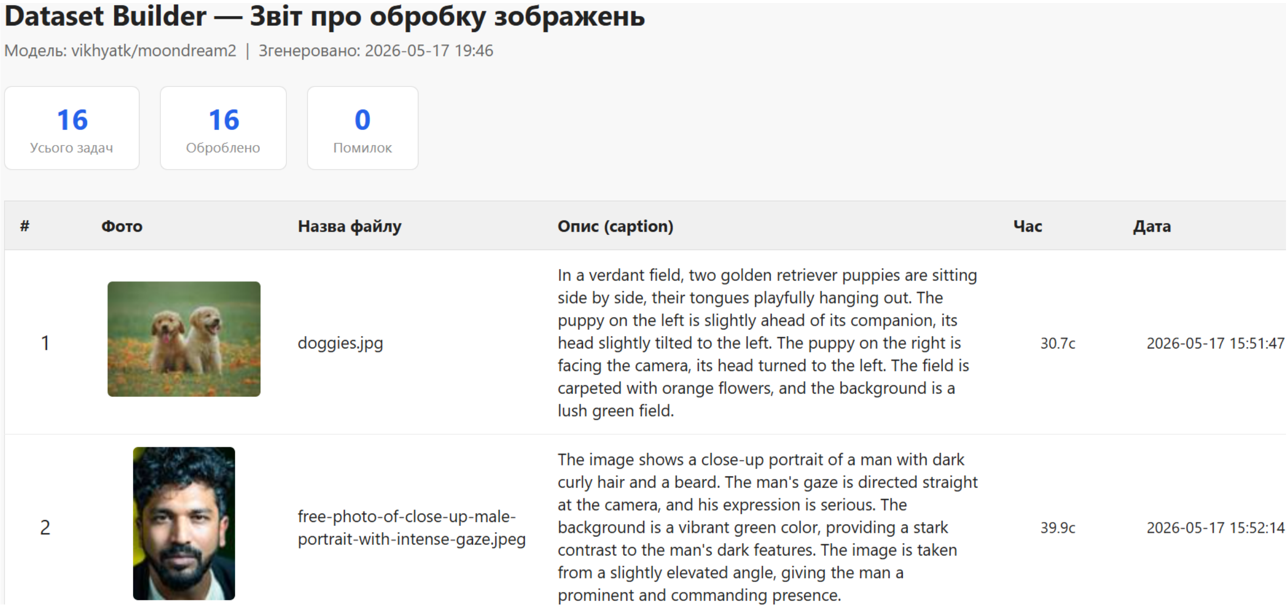
Оброблено: 0

```
[STATUS] Підключення до http://192.168.0.182:8000...
[STATUS] Завантаження моделі Moondream2...
[STATUS] Завантаження Moondream2...
[STATUS] Модель готова (8.2s)
[STATUS] Готово, чекаю задачі...
→ Задача 9d54f85a | __phainon_honkai_and_1_more_drawn_by_aonon_040__3d0b30824c5e81bbdc6
0bf85b1a3c847.jpg
```

Рисунок 3.8 – Воркер у процесі обробки задач

Після завершення обробки всіх задач воркер переходить у режим очікування і чекає на нові задачі. Це дозволяє залишити систему працювати у фоновому режимі і додавати нові пакети зображень за потребою без перезапуску воркера.

Отримання кінцевого результату виконується через інтерфейс майстра шляхом експорту датасету. Натискання кнопки "Експорт HTML" відкриває стандартний діалог збереження файлу, після чого формується HTML-звіт з мініатюрами всіх оброблених зображень, їх описами та метаданими. Згенерований звіт автоматично відкривається у браузері. Приклад HTML-звіту наведено на рисунку 3.9.



Dataset Builder — Звіт про обробку зображень

Модель: vikhyatk/moondream2 | Згенеровано: 2026-05-17 19:46

#	Фото	Назва файлу	Опис (caption)	Час	Дата
1		doggies.jpg	In a verdant field, two golden retriever puppies are sitting side by side, their tongues playfully hanging out. The puppy on the left is slightly ahead of its companion, its head slightly tilted to the left. The puppy on the right is facing the camera, its head turned to the left. The field is carpeted with orange flowers, and the background is a lush green field.	30.7с	2026-05-17 15:51:47
2		free-photo-of-close-up-male-portrait-with-intense-gaze.jpeg	The image shows a close-up portrait of a man with dark curly hair and a beard. The man's gaze is directed straight at the camera, and his expression is serious. The background is a vibrant green color, providing a stark contrast to the man's dark features. The image is taken from a slightly elevated angle, giving the man a prominent and commanding presence.	39.9с	2026-05-17 15:52:14

Рисунок 3.9 – HTML-звіт з результатами обробки

HTML-формат обрано як основний для презентації результатів через його універсальність і зручність перегляду. Файл відкривається у будь-якому браузері без встановлення додаткових програм, містить вбудовані мініатюри зображень у форматі base64, що робить його повністю автономним і придатним для передачі замовнику одним файлом. Для технічних потреб (інтеграція з іншими системами, статистична обробка) передбачено також експорт у форматі CSV з повним набором метаданих.

3.3 Результати виконання та подання

Підрозділ присвячений аналізу результатів роботи реалізованої системи на основі тестування з реальними зображеннями[48]. Аналіз охоплює якість згенерованих текстових описів, продуктивність системи у режимі з двома воркерами і формати подання сформованого датасету.

Для тестування підготовано набір з 16 зображень різного типу: фотографії тварин, пейзажів, портретів, побутових інтер’єрів, транспортних засобів і архітектурних об’єктів. Такий вибір дозволяє оцінити роботу моделі Moondream2 на різноманітному матеріалі. Обробку виконано з використанням двох ноутбуків у локальній мережі: Lenovo Legion 5 з процесором Ryzen 7 5800H і Acer Nitro 5 з процесором Ryzen 5 5600H. Усі 16 зображень оброблено успішно, кількість помилок склала 0.

Аналіз якості описів демонструє здатність моделі точно ідентифікувати об’єкти, їх характеристики і контекст сцени. Для фотографії спортивного автомобіля модель сформувала опис: "The image shows a vibrant blue sports car with a black hood parked on a grassy field. The car’s hood is open, revealing the engine and exhaust pipes. The car is positioned in the foreground, with a line of trees and a building visible in the background". Модель точно визначила колір і тип автомобіля, зафіксувала відкритий капот з видимим двигуном і вихлопними трубами, а також коректно описала фон сцени. Час обробки склав 40,5 секунди.

Зображення чорного ворона отримало опис: "A black crow is captured in a moment of quiet activity, standing on a patch of vibrant green grass. The crow’s feathers are a deep, dark blue, contrasting with the lighter green of the grass". Показовим є те, що модель розрізнила темно-синій відтінок у чорному пір’ї, хоча людським оком ця деталь не завжди помітна одразу.

Для пейзажної фотографії з озером згенеровано: "The image depicts a serene lake surrounded by a forested mountain. A wooden pier with a railing extends into the calm waters, where two individuals are standing, one wearing an orange jacket and the other wearing a black jacket". Модель розпізнала не лише загальну сцену, а й кольори одягу конкретних людей на задньому плані. Фотографія кота отримала опис з точним визначенням типу забарвлення (tabby), кольору очей, пози і кольору меблів. Жодних вигаданих деталей у відповідях протягом усього тестування не виявлено. Детальні результати обробки кожного зображення з розподілом між воркерами наведено у таблиці 3.1.

Таблиця 3.1 – Час інференсу зображень у двовузловому тесті

№	Опис Сцени	Lenovo Legion 5	Acer Nitro 5
1	Два цуценята в полі	30.7	–
2	Портрет чоловіка	–	39,9
3	Велосипед біля цегляної стіни	29,0	–
4	Собака	28,6	–
5	Спортивний автомобіль	40,5	–
6	Кіт на жовтому дивані	35,0	–
7	Параглайдинг	–	55,9
8	Троє людей у парку	36,2	–
9	Озеро з дерев’яним пірсом	34,6	–
10	Ворон на траві	–	50,6
11	Підвісний міст	38,2	–
12	Група людей під деревом	–	58,8
13	Затишна вітальня з каміном	37,9	–
14	Кам’яна церква у лісі	–	32,4
15	Собака у полі з квітами	–	59,6
16	Човен в океані	27,8	–

Кінець таблиці 3.1

Задач оброблено		10	6
Загальний час (с)		338,5	297,2
Середній час (с)		33,9	49,5

Аналіз продуктивності системи показав середній час обробки одного зображення 39,7 секунди при роботі двох воркерів одночасно. Найшвидше оброблено зображення човна (27,8 секунди), найповільніше – фотографія собаки у полі (59,6 секунди). Різниця пояснюється складністю сцен: зображення з великою кількістю об'єктів і персонажів потребують більшого часу для генерації розгорнутого опису.

Перший воркер на Legion 5 обробляв зображення зі середнім часом 33,9 секунди і взяв 10 із 16 задач. Другий воркер на Nitro 5 показав середній час 49,5 секунди і обробив 6 задач. Нерівномірний розподіл є природним наслідком pull-моделі: Legion 5 швидше завершував обробку і одразу запитував наступну задачу, тоді як Nitro 5 ще працював над попередньою. Жоден явний планувальник не використовувався. Детальніший аналіз механізму розподілу задач наведено у підрозділі 3.1.

Сумарний процесорний час обробки склав 635,7 секунди, розподілений між двома вузлами. Порівняння сценаріїв показує практичну цінність розподіленої архітектури. При послідовній обробці лише на Legion 5 знадобилось би 542,4 секунди, тобто близько 9 хвилин. На Nitro 5 в одиночному режимі – 792 секунди, понад 13 хвилин. Натомість реальний час очікування користувача при паралельній роботі двох воркерів визначається часом завершення найповільнішої гілки[44]: max(338,5; 297,2) = 338,5 секунди, тобто 5 хвилин і 38 секунд. Двовузлова система прискорила обробку у 1,6 раза порівняно з Legion 5 окремо і у 2,3 раза порівняно з Nitro 5 окремо. При масштабуванні до десятків воркерів приріст продуктивності зростатиме майже лінійно.

Для кількісної оцінки якості розпаралелювання застосовано показник ефективності, що враховує неоднорідність вузлів. У системах з однаковими вузлами ефективність визначають як відношення прискорення до кількості вузлів, проте у поточному тесті вузли мають різну продуктивність, тому коректнішим є порівняння з теоретично можливим часом при ідеальному балансуванні. Сумарна пропускна здатність двох вузлів дорівнює сумі їх індивідуальних швидкостей: $1/33{,}9 + 1/49{,}5 \approx 0{,}0497$ задачі за секунду. За такої пропускної здатності 16 задач у разі ідеально рівномірного завантаження були б оброблені за $16 \,/\, 0{,}0497 \approx 322$ секунди. Фактичний час склав 338,5 секунди, що дає ефективність розпаралелювання $322 \,/\, 338{,}5 \approx 0{,}95$, тобто 95%. Втрата близько 5% пояснюється двома причинами: дискретністю призначення задач, оскільки задачу не можна розділити між вузлами і останню задачу повільнішого воркера неможливо передати швидшому, та додатковим навантаженням на Legion 5, що одночасно виконував роль майстра. Такий показник є високим для системи без явного планувальника і підтверджує, що природне балансування pull-моделі наближається до теоретичного оптимуму[50]. Зведені результати аналізу прискорення наведено у таблиці 3.2

Таблиця 3.2 – Аналіз прискорення та ефективності розпаралелювання

Показник	Значення
Послідовна обробка на Legion 5	542,4 с
Послідовна обробка на Nitro 5	792,0 с
Паралельна обробка (фактична)	338,5 с
Теоретичний мінімум при ідеальному балансуванні	322,0 с
Прискорення відносно Legion 5	1,60×
Прискорення відносно Nitro 5	2,34×
Ефективність розпаралелювання	95%
Пропускна здатність системи	≈170 зображень/год

У перерахунку на пропускну здатність двовузлова система обробляє приблизно 170 зображень за годину, тоді як окремо Legion 5 дає близько 106, а Nitro 5 – близько 73 зображень за годину. Сума індивідуальних пропускних здатностей (179 зображень/год) майже збігається з фактичною, що ще раз підтверджує мінімальні втрати на координацію. Оскільки кожне призначення задачі є однією короткою операцією UPDATE...RETURNING тривалістю кілька мілісекунд, додавання нових воркерів збільшує сумарну пропускну здатність майже лінійно: десять вузлів класу Legion 5 теоретично оброблятимуть понад тисячу зображень за годину. Практичною межею такого зростання, як зазначено у підрозділі 3.1, є серіалізація операцій запису в SQLite, що стає відчутною при 20–30 одночасних воркерах; до досягнення цієї межі масштабування залишається близьким до лінійного і не потребує змін у коді.

З практичного погляду досягнута продуктивність повністю відповідає умовам поставленої задачі. Ручне анотування кількох тисяч зображень кваліфікованим фахівцем зайняло б тижні роботи, тоді як п’ять-десять звичайних робочих ПК, об’єднаних у розглянуту систему, формують датасет порівнянного обсягу за лічені години. При цьому жоден з вузлів не потребує виділеного GPU-сервера, а розгортання зводиться до встановлення Python і введення IP-адреси майстра. Саме поєднання прийнятної швидкості, нульових вимог до спеціалізованого обладнання і простоти масштабування робить розподілений підхід практично виправданим.

Формати подання результатів реалізовано у двох варіантах. Основним є HTML-звіт, призначений для перегляду людиною. Він містить зведену статистику у верхній частині і таблицю з мініатюрами всіх зображень, їх описами, часом обробки і датою. Зображення вбудовано у файл через base64-кодування, що робить HTML-звіт повністю автономним. Приклад сформованого HTML-звіту наведено на рисунку 3.9.

Перевагами HTML-формату є наочність, автономність і кросплатформеність. Файл відкривається у будь-якому браузері на будь-якому пристрої без встановлення додаткових програм, що робить його зручним для передачі замовнику або демонстрації на захисті.

Другим форматом є CSV-файл для технічної обробки. На відміну від HTML, він зберігає повний набір метаданих кожної задачі: унікальний ідентифікатор task_id, ім’я файлу, текст опису, назву і ревізію моделі, ідентифікатор воркера, час обробки в секундах, параметри генерації у JSON-форматі, дату створення і дату завершення задачі. Застосовано кодування UTF-8 з BOM-маркером для коректного відображення кириличних символів в Excel.

Формат CSV виконує роль не лише таблиці результатів, а й журналу обробки, придатного для аудиту. Кожен рядок містить ревізію моделі, ідентифікатор вузла і параметри генерації, що дозволяє пояснити розбіжності між описами при зміні конфігурації. Якщо якість частини записів виявиться недостатньою, адміністратор може відібрати конкретні рядки за ревізією або вузлом і перегенерувати лише їх без повторного опрацювання всього набору зображень[48].

Окрім технічних показників, тестування підтвердило зручність системи з точки зору кінцевого користувача. Весь процес від завантаження зображень до отримання готового HTML-звіту не потребує жодних технічних знань і зводиться до кількох кліків у графічному інтерфейсі.

Поєднання двох форматів забезпечує універсальність: HTML для людського перегляду, CSV для технічної обробки і подальшого використання у пайплайнах машинного навчання. Тестування підтвердило що система стабільно обробляє зображення у розподіленому режимі без помилок і втрат даних, а якість згенерованих описів відповідає рівню сучасних vision-language моделей.

ВИСНОВКИ

У роботі за результатами виконаних теоретичних та практичних досліджень спроєктовано і реалізовано розподілену систему аналізу зображень для автоматизованого формування навчальних датасетів на основі архітектури майстер-воркер з використанням vision-language моделі Moondream2.

У першому розділі проведено аналіз типів розподілених систем та платформ обробки задач. Розглянуто підходи cluster, grid, edge, fog і cloud-native, обґрунтовано вибір архітектури майстер-воркер як найбільш відповідної умовам розгортання на робочих ПК під Windows. Проаналізовано бібліотеки комп’ютерного зору і vision-language моделі, обґрунтовано вибір Moondream2 як компактної моделі з 1,86 мільярда параметрів, здатної працювати на CPU без виділеного GPU-сервера.

У другому розділі описано проєктування і реалізацію системи. Серверну частину побудовано на FastAPI з SQLite у WAL-режимі, проблему race condition вирішено через атомарну конструкцію UPDATE...RETURNING. Клієнтська частина реалізує pull-цикл обробки з механізмом автоматичних повторів, а розгортання системи зведено до мінімуму кроків без потреби у налаштуванні мережевої інфраструктури.

У третьому розділі продемонстровано практичну роботу системи і проаналізовано результати тестування. Обробку 16 зображень виконано без помилок двома воркерами, розподіл задач у співвідношенні 10 до 6 підтвердив природне балансування навантаження без явного планувальника. Результати збережено у форматах HTML і CSV.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Voron F. Building Data Science Applications with FastAPI. 2nd ed. Birmingham : Packt Publishing, 2023. 422 p.
2. Ayyadevara V. K., Reddy Y. Modern Computer Vision with PyTorch : Explore deep learning concepts and implement over 50 real-world image applications. Birmingham : Packt Publishing, 2020. 824 p.
3. Percival H., Gregory B. Architecture Patterns with Python : Enabling Test-Driven Development, Domain-Driven Design, and Event-Driven Microservices. Sebastopol : O’Reilly Media, 2020. 304 p.
4. Tunstall L., von Werra L., Wolf T. Natural Language Processing with Transformers : Building Language Applications with Hugging Face. Sebastopol : O’Reilly Media, 2022. 406 p.
5. McMahon A. P. Machine Learning Engineering with Python. Birmingham : Packt Publishing, 2021. 276 p.
6. Schuhmann C. et al. LAION-5B: An open large-scale dataset for training next generation image-text models. *Advances in Neural Information Processing Systems*. 2022. Vol. 35. P. 25278–25294.
7. Li J., Li D., Savarese S., Hoi S. BLIP-2: Bootstrapping Language-Image Pre-training with Frozen Image Encoders and Large Language Models. *Proceedings of the 40th International Conference on Machine Learning (ICML)*. 2023. P. 19730–19742.
8. Xiao B. et al. Florence-2: Advancing a Unified Representation for a Variety of Vision Tasks. *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2024. P. 4818–4829.
9. Stefanini M. et al. From Show to Tell: A Survey on Deep Learning-Based Image Captioning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 2023. Vol. 45, No. 1. P. 539–559.

10. Kreuzberger D., Kühl N., Hirschl S. Machine Learning Operations (MLOps): Overview, Definition, and Architecture. *IEEE Access*. 2023. Vol. 11. P. 31866–31879.

11. Herman M. Asynchronous Tasks with FastAPI and Celery. URL: https://testdriven.io/blog/fastapi-and-celery/ (дата звернення: 16.02.2026).

12. Khalusova M. Zero-shot image-to-text generation with BLIP-2. URL: https://huggingface.co/blog/blip-2 (дата звернення: 18.02.2026).

13. Salesforce. LAVIS: A Library for Language-Vision Intelligence. URL: https://github.com/salesforce/LAVIS (дата звернення: 20.02.2026).

14. Ramírez S. FastAPI Framework Official Documentation. URL: https://fastapi.tiangolo.com/ (дата звернення: 22.02.2026).

15. Microsoft. Florence-2-large : Model Repository. URL: https://huggingface.co/microsoft/Florence-2-large (дата звернення: 25.02.2026)

16. Cao K., Liu Y., Meng G., Sun Q. An overview on edge computing research. *IEEE Access*. 2020. Vol. 8. P. 85714–85728.

17. Li H. Research on fault detection algorithm of pantograph based on edge computing image processing. *IEEE Access*. 2020. Vol. 8. P. 84652–84659.

18. López M. R., Spillner J. Towards quantifiable boundaries for elastic horizontal scaling of microservices. *Proceedings of the 10th IEEE/ACM International Conference on Utility and Cloud Computing Companion*. 2017. P. 35–40.

19. Pawlaszczyk D. SQLite. Mobile Forensics – The File Format Handbook: Common File Formats and File Systems Used in Mobile Devices. Cham : Springer International Publishing, 2022. P. 129–155.

20. Johnson R. SQLite Essentials: Definitive Reference for Developers and Engineers. HiTeX Press, 2025.

21. Caires V., Vasconcelos J., Pinto D., Freitas V., Aveiro D. Rapid REST API Management. *Advances in Enterprise Engineering XVII*. 2024. Vol. 510. P. 73.

22. Reid E. Moondream Segmentation: From Words to Masks. *arXiv preprint arXiv:2604.02593*. 2026. URL: https://arxiv.org/abs/2604.02593 (дата звернення: 10.05.2026).

23. Pallonetto L., D'Arco L., Rossi S. Contextual Reasoning in Healthcare Social Robotics: A Preliminary Study Using Multimodal Language Models. *arXiv preprint*. 2025. URL: https://arxiv.org/abs/2503.07236 (дата звернення: 10.05.2026).

24. Niemir M., Grajewska D., Niton B. Vision-Language Models for E-commerce: Detecting Non-Compliant Product Images in Online Catalogs. *ICAART*. 2025. P. 1116–1123.

25. Cieplicka P., Klos J., Morawski M. VisionQAries at MEDIQA-MAGIC 2024: Small Vision Language Models for Dermatological Diagnosis. *CLEF (Working Notes)*. 2024. P. 1545–1551.

26. Cañas P. N., Hernández A., Nieto M., Rodriguez I. Exploring visual language models for driver gaze estimation: A task-based approach to debugging AI. *Computer Vision and Image Understanding*. 2025. P. 104593.

27. Thobhani A., Zou B., Kui X., Abdussalam A., Asim M., Shah S., Elaffendi M. A Survey on Enhancing Image Captioning with Advanced Strategies and Techniques. *Computer Modeling in Engineering & Sciences*. 2025. Vol. 142, No. 3.

28. Ghandi T., Pourreza H., Mahyar H. Deep learning approaches on image captioning: A review. *ACM Computing Surveys*. 2023. Vol. 56, No. 3. P. 1–39.

29. Tschannen M. et al. Siglip 2: Multilingual vision-language encoders with improved semantic understanding, localization, and dense features. *arXiv preprint arXiv:2502.14786*. 2025. URL: https://arxiv.org/abs/2502.14786 (дата звернення: 10.05.2026).

30. Abdin M. et al. Phi-4 technical report. arXiv preprint *arXiv:2412.08905*. 2024. URL: https://arxiv.org/abs/2412.08905 (дата звернення: 10.05.2026).

31. Abouelenin A. et al. Phi-4-mini technical report: Compact yet powerful multimodal language models via mixture-of-loras. arXiv preprint *arXiv:2503.01743*. 2025. URL: https://arxiv.org/abs/2503.01743 (дата звернення: 10.05.2026).

32. Hossain M. Z., Sohel F., Shiratuddin M. F., Laga H. A comprehensive survey of deep learning for image captioning. *ACM Computing Surveys*. 2019. Vol. 51, No. 6. P. 1–36.

33. Chohan M., Khan A., Mahar M. S., Hassan S., Ghafoor A., Khan M. Image captioning using deep learning: A systematic review. *International Journal of Advanced Computer Science and Applications*. 2020. Vol. 11, No. 5. P. 62.

34. Castro R., Pineda I., Lim W., Morocho-Cayamcela M. E. Deep learning approaches based on transformer architectures for image captioning tasks. *IEEE Access*. 2022. Vol. 10. P. 33679–33694.

35. Song Y., Soleymani M. Polysemous visual-semantic embedding for cross-modal retrieval. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019. P. 1979–1988.

36. Murthy V. N., Maji S., Manmatha R. Automatic image annotation using deep learning representations. *Proceedings of the 5th ACM International Conference on Multimedia Retrieval*. 2015. P. 603–606.

37. Adnan M. M., Rahim M. S. M., Rehman A., Mehmood Z., Saba T., Naqvi R. A. Automatic image annotation based on deep learning models: a systematic review and future challenges. *IEEE Access*. 2021. Vol. 9. P. 50253–50264.

38. Khan A. A., Laghari A. A., Awan S. A. Machine learning in computer vision: A review. *EAI Endorsed Transactions on Scalable Information Systems*. 2021. Vol. 8, No. 32.

39. Navarro P. J., Pérez F., Weiss J., Egea-Cortines M. Machine learning and computer vision system for phenotype data acquisition and analysis in plants. *Sensors*. 2016. Vol. 16, No. 5. P. 641.

40. Gupta N. et al. Data quality toolkit: Automatic assessment of data quality and remediation for machine learning datasets. arXiv preprint *arXiv:2108.05935*. 2021. URL: https://arxiv.org/abs/2108.05935 (дата звернення: 10.05.2026).

41. Li S. et al. PyTorch distributed: Experiences on accelerating data parallel training. arXiv preprint *arXiv:2006.15704*. 2020. URL: https://arxiv.org/abs/2006.15704 (дата звернення: 10.05.2026).

42. Wolf T., Debut L., Sanh V., Chaumond J., Delangue C., Moi A., Rush A. M. Transformers: State-of-the-art natural language processing. *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. 2020. P. 38–45.

43. Myers J., Copeland R., Copeland R. D. Essential SQLAlchemy. Sebastopol : O’Reilly Media, 2015.

44. Pierfederici F. Distributed Computing with Python. Birmingham : Packt Publishing, 2016.

45. Moritz P. et al. Ray: A distributed framework for emerging AI applications. *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018. P. 561–577.

46. Karau H., Lublinsky B. Scaling Python with Ray. Sebastopol : O’Reilly Media, 2022

47. Cui Y., Yang G., Veit A., Huang X., Belongie S. Learning to evaluate image captioning. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018. P. 5804–5812.

48. Luo G., Cheng L., Jing C., Zhao C., Song G. A thorough review of models, evaluation metrics, and datasets on image captioning. *IET Image Processing*. 2022. Vol. 16, No. 2. P. 311–332.

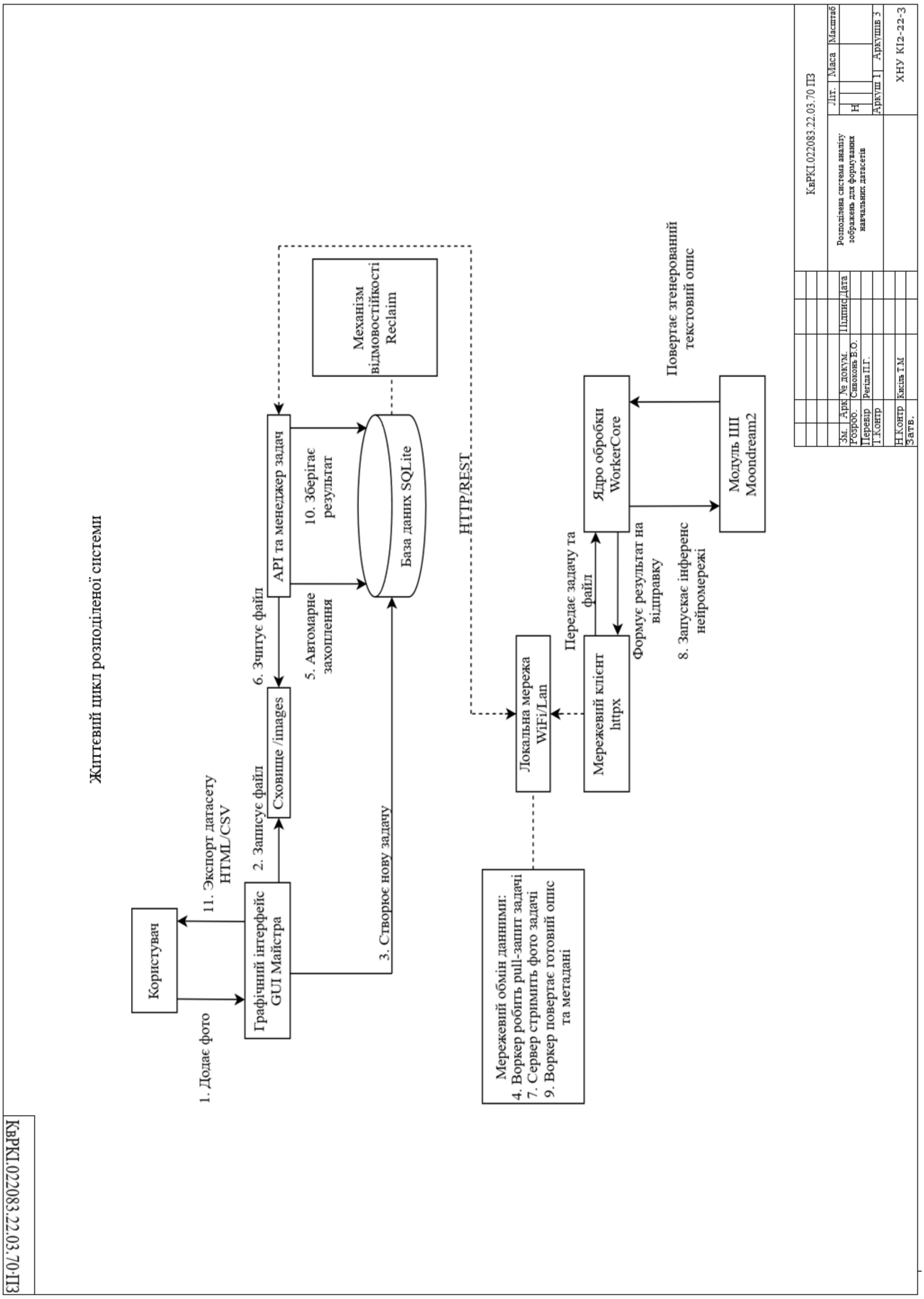
49. Hessel J., Holtzman A., Forbes M., Le Bras R., Choi Y. CLIPScore: A reference-free evaluation metric for image captioning. *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 2021. P. 7514–7528.

50. Киселевич В. В., Усата О. Ю., Сікора Я. Б., Вербівський Д. С., Іванов Д. Є. Мікросервісна архітектура: переваги та недоліки її практичного застосування. *Information Technology: Computer Science, Software Engineering and Cyber Security*. 2024. № 2. С. 50–59.

ДОДАТОК А

(обов’язковий)

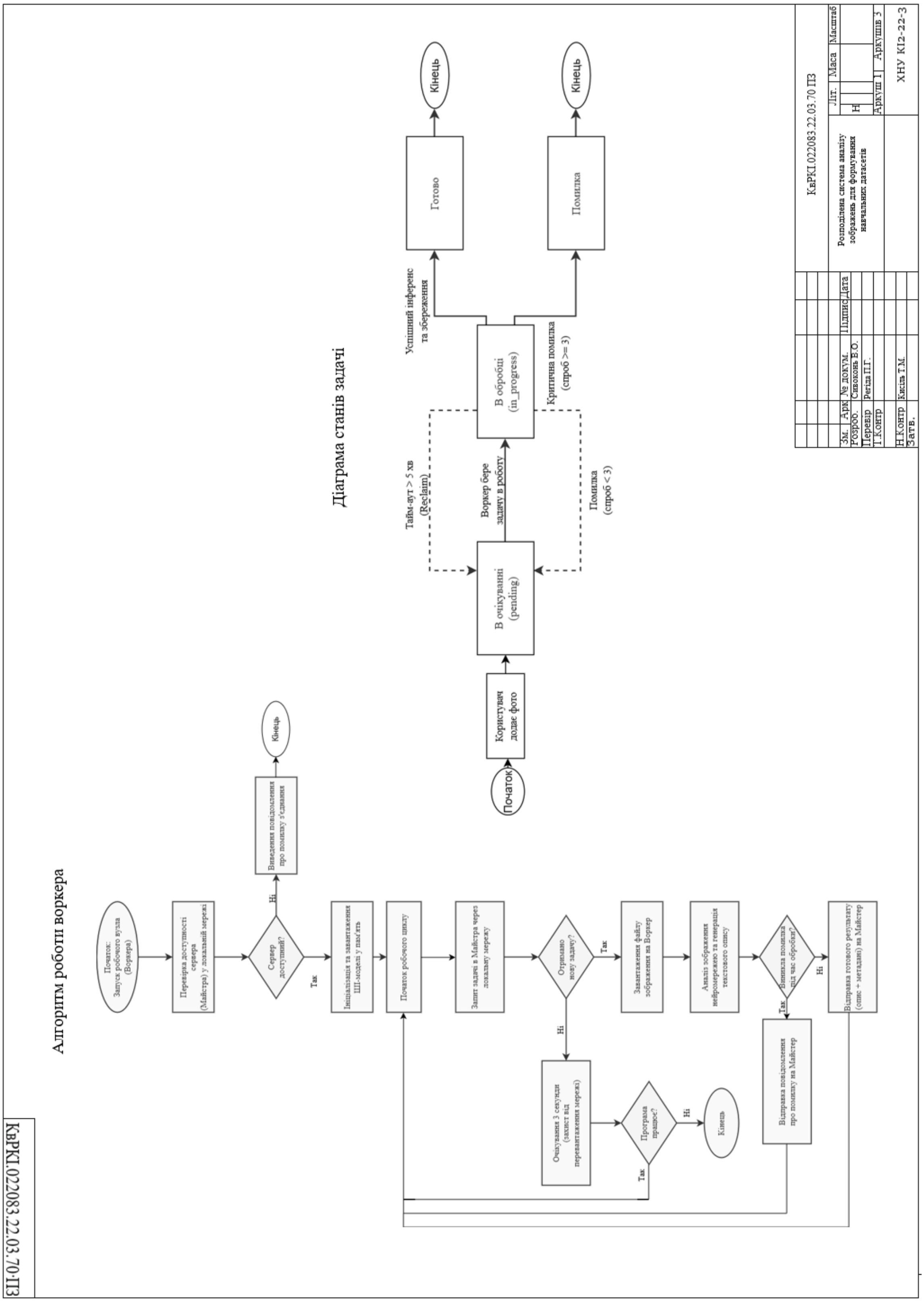
Копія креслення «Загальна архітектура та життєвий цикл»



ДОДАТОК Б

(обов’язковий)

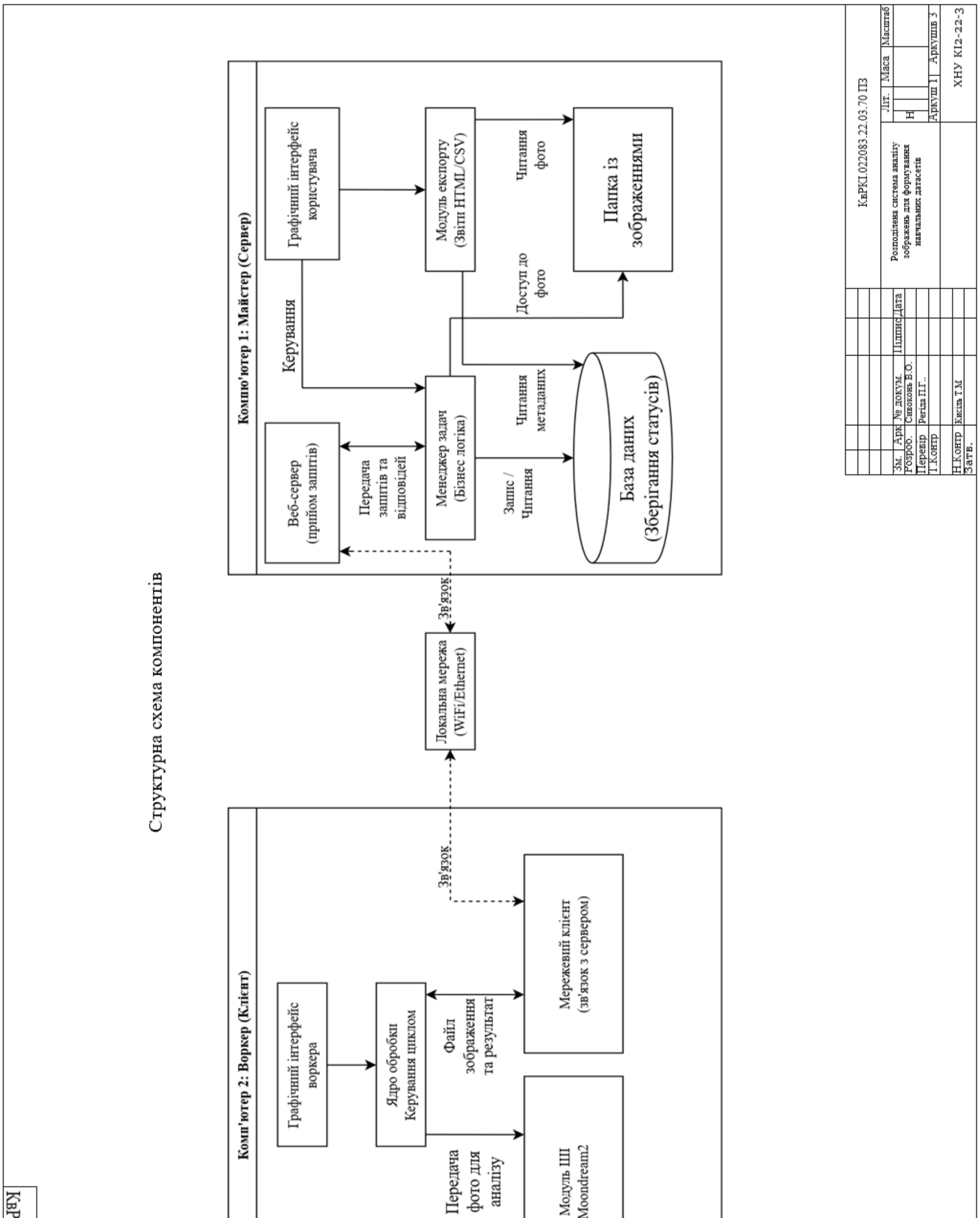
Копія креслення «Алгоритм роботи воркера та діаграма станів задачі»



ДОДАТОК В

(обов’язковий)

Копія креслення «Структурна схема компонентів»



ДОДАТОК Г

(обов’язковий)

Копія креслення «Лістинг коду модуля диспетчеризації»

```
     """app/server/task_manager.py"""
     from __future__ import annotations

     import json
     import logging
     from pathlib import Path
     from typing import Optional

     from sqlalchemy import select, func, text
     from sqlalchemy.orm import Session

     from app.server.models import Task
     from app.server.schemas import ResultPayload, StatsOut
     from app.shared.utils import elapsed_seconds, file_md5,
is_image_file, new_task_id, now_iso

     logger = logging.getLogger(__name__)


     def add_task_from_path(db: Session, image_path: Path) ->
Optional[Task]:
         if not image_path.exists() or not is_image_file(image_path):
             return None

         img_hash = file_md5(image_path)
         existing = db.execute(select(Task).where(Task.image_hash ==
img_hash)).scalar_one_or_none()
         if existing:
             return None
```

```
        task = Task(
            task_id=new_task_id(),
            image_name=image_path.name,
            image_path=str(image_path.resolve()),
            image_hash=img_hash,
            status="pending",
            retry_count=0,
            created_at=now_iso(),
        )
        db.add(task)
        db.commit()
        db.refresh(task)
        return task


    def import_images_from_folder(db: Session, folder: Path) -> int:
        added = 0
        for path in sorted(folder.iterdir()):
            if is_image_file(path):
                if add_task_from_path(db, path):
                    added += 1
        return added



    def claim_next_task(db: Session) -> Optional[Task]:
        """Атомарне призначення задачі через UPDATE...RETURNING (SQLite
3.35+)."""
        assigned_at = now_iso()
        result = db.execute(
            text("""
                UPDATE tasks
                SET    status = 'in_progress', assigned_at = :assigned_at
                WHERE  task_id = (
```

```
                SELECT task_id FROM tasks
                WHERE  status = 'pending'
                ORDER  BY created_at LIMIT 1
            )
            RETURNING task_id
        """),
        {"assigned_at": assigned_at},
    )
    row = result.fetchone()
    db.commit()
    if row is None:
        return None
    return db.get(Task, row[0])


def reclaim_timed_out_tasks(db: Session, timeout_seconds: int, max_retries: int) -> int:
    in_progress = db.execute(select(Task).where(Task.status == "in_progress")).scalars().all()
    reclaimed = 0
    for task in in_progress:
        if task.assigned_at and elapsed_seconds(task.assigned_at) > timeout_seconds:
            if task.retry_count >= max_retries:
                task.status = "error"
                task.error_message = f"Перевищено ліміт повторів ({max_retries})"
            else:
                task.status = "pending"
                task.retry_count += 1
                task.worker_id = None
                task.assigned_at = None
            reclaimed += 1
    if reclaimed:
```

```
             db.commit()
         return reclaimed


     def save_result(db: Session, task_id: str, payload: ResultPayload) ->
Optional[Task]:
         task = db.get(Task, task_id)
         if task is None:
             return None
         task.status = "done"
         task.worker_id = payload.worker_id
         task.caption = payload.caption
         task.model_name = payload.model_name
         task.model_version = payload.model_version
         task.processing_time = payload.processing_time
         task.inference_params = (
             json.dumps(payload.inference_params, ensure_ascii=False)
             if payload.inference_params else None
         )
         task.finished_at = now_iso()
         db.commit()
         db.refresh(task)
         return task


     def save_error(db: Session, task_id: str, worker_id: str,
error_message: str) -> None:
         task = db.get(Task, task_id)
         if task is None:
             return
         task.status = "pending"
         task.retry_count += 1
         task.worker_id = None
         task.assigned_at = None
```

```
    task.error_message = error_message[:1000]
    db.commit()


def get_stats(db: Session) -> StatsOut:
    rows = db.execute(select(Task.status, func.count().label("cnt")).group_by(Task.status)).all()
    counts = {row.status: row.cnt for row in rows}
    return StatsOut(
        total=sum(counts.values()),
        pending=counts.get("pending", 0),
        in_progress=counts.get("in_progress", 0),
        done=counts.get("done", 0),
        error=counts.get("error", 0),
    )


def get_worker_stats(db: Session) -> list:
    """Повертає статистику по кожному воркеру: кількість задач, середній час."""
    rows = db.execute(
        text("""
            SELECT worker_id,
                   COUNT(*) as tasks_done,
                   AVG(processing_time) as avg_time,
                   MAX(finished_at) as last_seen
            FROM tasks
            WHERE status = 'done' AND worker_id IS NOT NULL
            GROUP BY worker_id
            ORDER BY tasks_done DESC
        """)
    ).all()
    return [
        {
```

```
                "worker_id": row.worker_id,
                "tasks_done": row.tasks_done,
                "avg_time": round(row.avg_time or 0, 1),
                "last_seen": row.last_seen,
            }
            for row in rows
        ]


  def reset_failed_tasks(db: Session) -> int:
        """Скидає task в статусах error і in_progress назад у pending."""
        tasks = db.execute(
            select(Task).where(Task.status.in_(["error", "in_progress"]))
        ).scalars().all()
        for t in tasks:
            t.status = "pending"
            t.retry_count = 0
            t.worker_id = None
            t.assigned_at = None
            t.error_message = None
        db.commit()
        return len(tasks)


  def clear_all_tasks(db: Session) -> int:
        """Видаляє всі задачі з бази."""
        count = db.query(Task).count()
        db.query(Task).delete()
        db.commit()
return count
```

Протокол аналізу звіту подібності експертом

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Володимир СИВОКОНЬ

Співавтор:

Назва: Розподілена система аналізу зображень для формування навчальних датасетів

Експерт: Павло РЕГІДА

Підрозділ: Кафедра комп'ютерної інженерії та інформаційних систем

Коефіцієнт подібності 1:4.84%

Коефіцієнт подібності 2:1.79%

Мікропробіли: 3

Заміна букв: 0

Інтервали: 0

Білі знаки: 0

Дата створення звіту: 2026-05-26 14:40:01.0

Після аналізу Звіту подібності констатую наступне:

☒ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

2026-05-26 Доцент Андрій Нічепорук

Дата експерт

Anti-Plagiarism (http://ap.km.ua) v-15.701

Максимальне співпадіння з одним документом 2.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 14%**

	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
ID: 272313 Назва: БКР Розподілена система аналізу зображень для формування навчальних датасетів Додано в БД: 2026-05-26 Автора: Володимир СИВОКОНЬ Керівники: Павло РЕГІДА Консультанти: Опоненти:	107258	938	3980 (4%)	53 (6%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Дипломник: Сивоконь Володимир Олександрович

Тема: Розподілена система аналізу зображень для формування навчальних датасетів

Спеціальність: 123 «Комп'ютерна інженерія»

Обсяг кваліфікаційної роботи:

Кількість листів креслень ___3____ Кількість сторінок записки ___61___

1. Короткий зміст роботи та прийнятих рішень: Метою кваліфікаційної роботи була розробка розподіленої системи аналізу зображень для формування навчальних датасетів.

2. Висновок про відповідність роботи дипломному завданню: Робота повністю відповідає поставленому завданню.

3. Характеристика виконання кожного розділу, ступінь використання останніх досягнень науки і техніки і передових методів роботи: У першому розділі проведено теоретичне дослідження методів і засобів побудови розподілених систем обробки зображень: проаналізовано основні архітектурні підходи (кластерний, сітковий, кордонний, туманний та хмарно-орієнтований), виконано порівняльний огляд платформ розподіленої обробки (Celery, Ray, Apache Airflow/Prefect, NVIDIA Triton), досліджено протоколи мережевої взаємодії та сучасні зорово-мовні моделі. Обґрунтовано вибір архітектури головний і робочий вузол з pull-механізмом та моделі Moondream2 для інференсу. У другому розділі виконано проєктування та програмну реалізацію системи: розроблено серверну частину на FastAPI з SQLite у WAL-режимі, реалізовано атомарне призначення задач через UPDATE...RETURNING для усунення стану гонитви, спроєктовано клієнтську частину з циклом обробки задач, механізмом повторів і функціями оновлення інтерфейсу, реалізовано мережеву взаємодію на базі HTTP/REST з ідентифікацією робочих вузлів за допомогою унікальних ідентифікаторів, а також графічний інтерфейс на Tkinter та експорт датасету у форматах HTML і CSV. У третьому розділі проведено експериментальну оцінку

системи у локальній мережі з двома обчислювальними вузлами різної продуктивності. Оброблено 16 зображень без помилок. Підтверджено, що попередньо натренована модель Moondream2 успішно розпізнала об'єкти на фото та згенерувала точні текстові описи. Також доведено коректність атомарного призначення задач і динамічне балансування навантаження у співвідношенні 10:6. Встановлено, що система з двома робочими вузлами забезпечує прискорення обробки у 1,6 раза у порівнянні із менш потужним вузлом.

4. Позитивні сторони роботи: висока практична цінність роботи.

5. Негативні сторони роботи: -

6. Оцінка графічного оформлення та пояснювальної записки роботи: Пояснювальна записка оформлена коректно, згідно діючих стандартів оформлення документації.

7. Відгук про роботу в цілому: робота виконана на високому технічному рівні.

8. Інші зауваження: ____

9. Оцінка дипломної роботи: відмінно (A / 96)

Рецензент (прізвище, ім’я, по батькові, посада, місце роботи) ________________

Седюк Микола Васильович, PhD, ст. викладач, к. Кібербезпеки

____

“28” 05 ______________2026 р. __________________(підпис)

Зав. кафедри КІІС
д-р. філософії Ользі ПАВЛОВІЙ

Сивоконь ВОЛОДИМИР
ПІБ здобувача вищої освіти

ФІТ, 4 курсу, групи КІ2-22-3

ЗАЯВА

З правилами чинного Положення про систему забезпечення академічної доброчесності у Хмельницькому національному університеті, згідно з яким виявлення академічного плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів академічної відповідальності, ознайомлений (а). Про використання спеціалізованих програмних засобів (СПЗ) StrikePlagiarism та Anti-Plagiarism для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність академічного плагіату оповіщений (а). Надаю університету право на передачу моєї роботи для обробки та збереження в базах даних СПЗ і використання роботи для виявлення академічного плагіату в інших роботах, які перевіряються СПЗ.

Також надаю свою згоду на обробку й збереження університетом моєї роботи в Інституційному репозитарії Хмельницького національного університету.

Робота надається для перевірки в електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

1 травня 2026 року

РІШЕННЯ ЕКСПЕРТНОЇ КОМІСІЇ

КАФЕДРИ КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Назва кваліфікаційної роботи Програмно-технічний засіб реалізації Onvif сервера на базі Raspberry PI
Автор Володимир СИВОКОНЬ
Освітня програма Комп'ютерна інженерія та програмування
Рівень вищої освіти перший (бакалаврський)
Спеціальність 123 Комп'ютерна інженерія
Науковий керівник: д-р філософії, Павло РЕГІДА

На основі аналізу кваліфікаційної роботи на дотримання вимог академічної доброчесності (у т.ч. відсутності ознак академічного плагіату) з урахуванням результатів перевірки роботи спеціалізованим програмним засобом(ами) комісія зробила такий висновок:

№	Висновок	Позначка про відповідність
1	Ознаки академічного плагіату	
1.1	Запозичення, виявлені в роботі, є законними і не є академічним плагіатом (далі – зазначаються підстави віднесення запозичень до правомірних, якщо потрібно). Робота приймається до захисту.	відповідає
1.2	Виявлені запозичення не є академічним плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи (далі – зазначаються детальні та аргументовані підстави віднесення запозичень до правомірних). Робота приймається до захисту, але має бути відкоригована.	
1.3	Виявлені запозичення не є академічним плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота може бути допущена до захисту після того як буде відкоригована та доопрацьована і успішно пройде повторну перевірку на академічний плагіат.	
1.4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття текстових запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
2	Інші види порушень академічної доброчесності	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) усі запозичення фрагментарні, або мають належним чином оформленні посилання;
2) окремі виявлені збіги є загальновживаними фразами або виразами, про що свідчить посилання системи на збіг з джерелами на один фрагмент речення;
3) всі зафіксовані системою ознаки модифікації тексту відносяться до комбінування латинських символів зі україномовними скороченнями індексів в формулах, що не є модифікацією тексту.
4) значна частина знайденого плагіату відноситься до списку використаних джерел

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів/ ідентичності/схожості StrikePlagiarism, складає 4.84%; та системою Anti-Plagiarism складає 4.0%, що, з урахуванням наведених обґрунтувань, відповідає характеру наукового дослідження і свідчить на користь кваліфікаційної роботи.

01.06.2026

Завідувач кафедри ________ Ольга ПАВЛОВА
Підпис Ім'я, ПРІЗВИЩЕ

Гарант освітньої програми ________ Андрій НІЧЕПОРУК
Підпис Ім'я, ПРІЗВИЩЕ

Керівник кваліфікаційної роботи ________ Павло РЕГІДА
Підпис Ім'я, ПРІЗВИЩЕ