Міністерство освіти і науки України

Луцький національний технічний університет
(повне найменування закладу вищої освіти)

Факультет комп’ютерних та інформаційних технологій
(повне найменування факультету)

Кафедра комп’ютерної інженерії та безпеки
(повне найменування кафедри)

КВАЛІФІКАЦІЙНА РОБОТА
ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «МАГІСТР»

АНАЛІЗ АРХІТЕКТУРНИХ ПІДХОДІВ ДО СТВОРЕННЯ АВТОМАТИЗОВАНОЇ СИСТЕМИ НАВЧАННЯ МОДЕЛЕЙ ШТУЧНОГО ІНТЕЛЕКТУ

ANALYSIS OF ARCHITECTURAL APPROACHES TO CREATING AN AUTOMATED TRAINING SYSTEM FOR ARTIFICIAL INTELLIGENCE MODELS

спеціальність 123 Комп’ютерна інженерія
(шифр і назва спеціальності)

освітня програма Комп’ютерна інженерія
(назва освітньої програми)

Виконав: здобувач вищої освіти
групи КІм-21
Конотопчик Артем Миколайович

____
(підпис)

Керівник: к.т.н., доцент
Мельник Катерина Вікторівна

____
(підпис)

Кваліфікаційну роботу
допущено до захисту
«_____» грудня 2025 р.
Гарант освітньої програми:
к.т.н., доцент
Гринюк Сергій Васильович

____

Луцьк – 2025 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *комп'ютерної інженерії та безпеки*
Ступінь вищої освіти: *магістр*
Галузь знань: *12 Інформаційні технології*
Спеціальність: *123 Комп'ютерна інженерія*
Освітня програма: *«Комп'ютерна інженерія»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
________________ доц. Т.ТЕРЛЕЦЬКИЙ
«_______» ______________ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Конотопчику Артему Миколайовичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи *Аналіз архітектурних підходів до створення автоматизованої системи навчання моделей штучного інтелекту*

Керівник роботи *к.т.н., доцент Мельник Катерина Вікторівна*

затверджені наказом закладу вищої освіти від «17» червня 2025 року № 290/01-02
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи 09.12.2025р.

3. Вихідні дані до роботи *Джерелом розробки є науково-технічна література та публікації в періодичних виданнях з даного питання, опубліковані зарубіжні та вітчизняні роботи в даній області, різні інтернет-ресурси технічного спрямування*

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити):
Вступ
Аналіз проблеми за темою роботи та постановка завдань дослідження
Теоретичний аналіз проблем життєвого циклу моделей штучного інтелекту
Аналіз існуючих архітектурних підходів до автоматизації навчання
Проектування, реалізація та експериментальна валідація автоматизованої MLOps-платформи
Висновки

5. Перелік графічного (ілюстративного) матеріалу:

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Теоретичний аналіз проблем життєвого циклу моделей штучного інтелекту	*Мельник К.В., доцент*		
Аналіз існуючих архітектурних підходів до автоматизації навчання	*Мельник К.В., доцент*		
Проектування, реалізація та експериментальна валідація автоматизованої MLOps-платформи	*Мельник К.В., доцент*		
Нормоконтроль	*Багнюк Н.В., доцент*		
Гарант ОП	*Гринюк С.В., доцент*		
Показник запозичень тексту	___%		
Академічна доброчесність	*Міскевич О.І., ст.викладач*		

7. Дата видачі завдання 18.06.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	*Огляд літератури із досліджуваної проблеми*	До 01.08.2025 р.	
2.	*Аналіз проблеми за темою роботи та постановка завдань дослідження*	До 20.08.2025 р.	
3.	*Теоретичне дослідження та практична реалізація*	До 25.09.2025 р.	
4.	*Практична реалізація об'єкта проектування*	До 20.10.2025 р.	
5.	*Висновки та пропозиції*	До 25.10.2025 р	
6.	*Формування списку використаних джерел*	До 27.10.2025 р.	
7.	*Формування додатків*	До 30.10.2025 р.	
8.	*Оформлення ілюстративного матеріалу*	До 05.11.2025 р.	
9.	*Представлення остаточного варіанту кваліфікаційної роботи керівникові*	До 11.11.2025 р.	
10.	*Нормоконтроль*	До 29.11.2025 р.	
11.	*Інструментальна перевірка на академічний плагіат*	До 02.12.2025 р.	
12.	*Здача кваліфікаційної роботи та всіх супровідних документів на кафедру*	До 09.12.2025 р.	

Здобувач вищої освіти ____________ Конотопчик А.М.
(підпис) (прізвище, ініціали)

Керівник кваліфікаційної роботи ____________ Мельник К.В.
(підпис) (прізвище, ініціали)

АНОТАЦІЯ

Конотопчик А. М. Аналіз архітектурних підходів до створення автоматизованої системи навчання моделей штучного інтелекту. Рукопис.

Кваліфікаційна робота магістра ОП «Комп'ютерна інженерія» спеціальності 123 Комп'ютерна інженерія. Луцький національний технічний університет. Луцьк, 2025.

Кваліфікаційна робота складається з вступу, трьох розділів, висновків, списку використаних джерел, додатків.

Перший розділ аналізує теоретичні проблеми життєвого циклу MLOps. На основі аналізу джерел обґрунтовується необхідність контейнеризації для вирішення проблеми відтворюваності та доводиться проблема комплексності аналізу, яка вимагає порівняння гетерогенних (різнотипних) моделей.

В другому розділі проводиться критичний аналіз існуючих архітектурних рішень. Розглядається базовий підхід на основі Docker Compose, аналізуються його недоліки щодо масштабування та аналізу. Також аналізуються монолітні платформи (наприклад, Kubeflow) та спеціалізовані інструменти (наприклад, Argo, MLflow), що дозволяє виявити «архітектурну прогалину» та обґрунтувати необхідність гібридного підходу.

Третій розділ присвячено проектуванню, реалізацію та експериментальній валідації пропонованої гібридної MLOps-платформи. Архітектура поєднює Kubernetes (для оркестрації та оптимізації ресурсів через механізм Job Queue), MLflow (для відстеження та анаізу експериментів) та MinIO (для зберігання артефактів). Проведено комплексний експеримент, який доводить, що платформа ефективно усуває простої обладнання та надає єдиний інтерфейс для аналізу гетерогенних моделей.

Ключові слова: машинне навчання, MLOps, Kubernetes, Docker, MLflow, оптимізація ресурсів, оркестрація, аналіз експериментів, відтворюваність, гібридна архітектура.

ANNOTATION

Konotopchyk A. Analysis of architectural approaches to creating an automated system for training artificial intelligence models. Manuscript.

Qualifying work of a Master's of EP «Computer Engineering» specialty 123 Computer Engineering. Lutsk National Technical University. Lutsk, 2025.

Qualification work consists of an introduction, three sections, conclusions, a references, three appendices.

The first chapter analyses the theoretical problems of the MLOps life cycle. Based on the analysis of sources, the need for containerisation to solve the problem of reproducibility is justified, and the problem of the complexity of analysis, which requires the comparison of heterogeneous (different types of) models, is proven.

The second section provides a critical analysis of existing architectural solutions. It considers the basic approach based on Docker Compose and analyses its shortcomings in terms of scalability and analysis. Monolithic platforms (e.g., Kubeflow) and specialised tools (e.g., Argo, MLflow) are also analysed, allowing us to identify an «architectural gap» and justify the need for a hybrid approach.

The third section is devoted to the design, implementation, and experimental validation of the proposed hybrid MLOps platform. The architecture combines Kubernetes (for orchestration and resource optimisation through the Job Queue mechanism), MLflow (for tracking and analysing experiments) and MinIO (for storing artefacts). A comprehensive experiment was conducted, proving that the platform effectively eliminates equipment downtime and provides a single interface for analysing heterogeneous models.

Keywords: machine learning, MLOps, Kubernetes, Docker, MLflow, resource optimisation, orchestration, experiment analysis, reproducibility, hybrid architecture.

ЗМІСТ

ВСТУП 7
РОЗДІЛ 1 ТЕОРЕТИЧНИЙ АНАЛІЗ ПРОБЛЕМ ЖИТТЄВОГО ЦИКЛУ МОДЕЛЕЙ ШТУЧНОГО ІНТЕЛЕКТУ 9
1.1 Життєвий цикл MLOps та сучасні виклики 9
1.2 Проблема відтворюваності та ізоляції середовищ 11
1.3 Проблема неефективного використання апаратних ресурсів 13
1.4 Проблема комплексності аналізу гетерогенних моделей 15
РОЗДІЛ 2 АНАЛІЗ ІСНУЮЧИХ АРХІТЕКТУРНИХ ПІДХОДІВ ДО АВТОМАТИЗАЦІЇ НАВЧАННЯ 19
2.1 Базовий підхід: Контейнеризація та ручна оркестрація 19
2.2 Аналіз монолітних MLOps-платформ 22
2.3 Аналіз спеціалізованих інструментів (комбінований підхід) 24
2.4 Обґрунтування гібридного підходу 26
РОЗДІЛ 3 ПРОЕКТУВАННЯ, РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНА ВАЛІДАЦІЯ АВТОМАТИЗОВАНОЇ MLOPS-ПЛАТФОРМИ 29
3.1 Архітектура платформи та стек технологій 29
3.1.1 Kubernetes як технологічна основа динамічної оркестрації та оптимізації ресурсів 30
3.1.2 MLflow як централізована підсистема відстеження та аналізу експериментів 31
3.1.3 MinIO як масштабоване сховище артефактів S3 33
3.2 Проектування та реалізація керуючої підсистеми 36
3.3 Практична реалізація та вирішення проблем інтеграції компонентів 41
3.4 Експериментальна валідація на комплексній задачі 47
ВИСНОВКИ 56
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ 57
ДОДАТКИ 60

ВСТУП

Актуальність теми. Розробка систем штучного інтелекту (ШІ) та машинного навчання (МН) трансформувала науку та бізнес, однак сам процес розробки стикається з фундаментальними інженерними викликами. Традиційні підходи до навчання моделей, що часто базуються на ручному запуску скриптів, страждають від трьох ключових проблем: «кризи відтворюваності» (неможливості повторити експеримент через відмінності у середовищах), неефективного використання дорогих апаратних ресурсів (простої GPU/CPU між запусками) та складності аналізу десятків чи сотень експериментальних запусків. У міру зростання складності моделей та вимог до їх надійності виникає гостра потреба в аналізі та розробці архітектурних підходів, що здатні автоматизувати, оптимізувати та стандартизувати цей процес.

Метою роботи є аналіз існуючих архітектурних підходів та проектування гібридної автоматизованої системи для навчання моделей ШІ, здатної вирішити проблеми відтворюваності, оптимізації апаратних ресурсів та комплексного аналізу гетерогенних експериментів.

Об’єкт дослідження – процес життєвого циклу розробки моделей машинного навчання (MLOps).

Предмет дослідження – архітектурні підходи, методи та інструменти для автоматизації та оптимізації фази навчання та експериментального аналізу моделей ШІ.

Завдання, які необхідно виконати:

– провести теоретичний аналіз проблем життєвого циклу MLOps, зокрема проблем відтворюваності середовищ, неефективного використання ресурсів та комплексності аналізу гетерогенних моделей;

– провести критичний аналіз існуючих архітектурних підходів до автоматизації навчання, включаючи базові (на основі Docker Compose), монолітні (на основі Kubeflow) та спеціалізовані (best-of-breed) рішення;

– спроектувати гібридну архітектуру автоматизованої платформи, що поєднує переваги Kubernetes (для оркестрації та оптимізації) та MLflow (для відстеження та аналізу);

– провести практичну реалізацію та експериментальну валідацію пропонованої платформи на комплексній задачі, довівши її здатність керувати чергою завдань, оптимізувати ресурси та надавати інструменти для аналізу гетерогенних моделей.

Апробація: результати, отримані в ході виконання кваліфікаційної роботи, були апробовані та представлені на Міжнародній науково-практичній конференції молодих вчених та студентів 6 травня 2025р., м. Луцьк [1], та на «14th International Conference on Dependable Systems, Services and Technologies» 11-13 жовтня 2024р., м. Афіни, Греція [2].

РОЗДІЛ 1
ТЕОРЕТИЧНИЙ АНАЛІЗ ПРОБЛЕМ ЖИТТЄВОГО ЦИКЛУ МОДЕЛЕЙ ШТУЧНОГО ІНТЕЛЕКТУ

1.1 Життєвий цикл MLOps та сучасні виклики

Розробка систем штучного інтелекту (ШІ) та, зокрема, моделей машинного навчання (МН), фундаментально відрізняється від традиційної розробки програмного забезпечення. Класичне програмне забезпечення є детермінованим, його поведінка повністю описується програмним кодом, а життєвий цикл зосереджений на управлінні змінами цього коду. Натомість, поведінка системи машинного навчання є за своєю природою стохастичною (ймовірнісною) і визначається динамічною комбінацією трьох сутностей, що називають «трьома китами» МН: код – алгоритми та логіка обробки, дані – навчальні та валідаційні вибірки та модель – бінарний артефакт, отриманий у результаті навчання [3].

Ця тріада код-дані-модель породжує унікальні та значно складніші виклики у процесі розробки. На відміну від коду, дані не є статичними; вони можуть змінюватись, деградувати (явище «data drift») або оновлюватись, що вимагає постійного перенавчання та валідації моделей. Сама модель є «чорною скринькою», зміни в якій не завжди прямо корелюють зі змінами в коді чи даних.

Для управління цією новою парадигмою складності була сформована інженерна дисципліна MLOps (від англ. Machine Learning Operations – операції машинного навчання) [4, 5]. MLOps є сукупністю практик та інструментів, що поєднує методи DevOps (автоматизація, моніторинг, безперервна інтеграція та доставка, відомі як CI/CD), Data Engineering (управління даними, конвеєризація) та Machine Learning (дослідження та моделювання) з метою створення надійного, відтворюваного та ефективного процесу розробки систем ШІ [5].

Життєвий цикл MLOps, представлений на рисунку 1.1, є за своєю природою глибоко ітеративним. Хоча етапи, такі як розгортання та моніторинг, є критично важливими для виробничого використання, ядро будь-якого наукового дослідження або розробки нового продукту знаходиться у циклі

навчання та експерименти – оцінка. Саме цей цикл є «дослідницькою пісочницею» (англ. «sandbox»), де інженери ШІ та науковці у галузі Data Scientists перевіряють гіпотези.

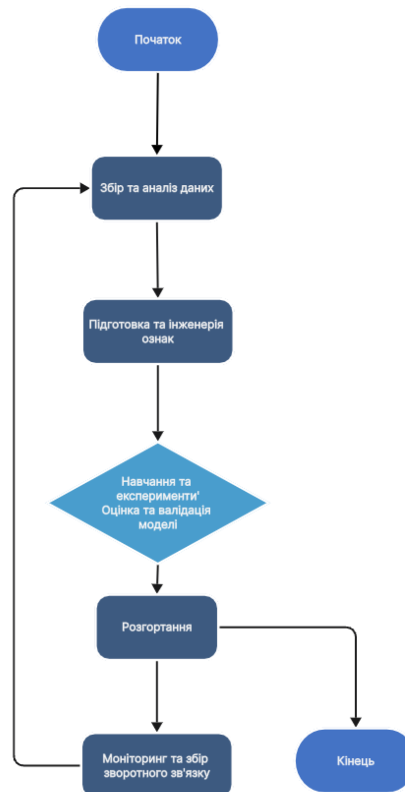


Рисунок 1.1 – Схематичне зображення життєвого циклу MLOps

Дослідники змушені повторювати цей цикл десятки або й сотні разів, змінюючи архітектури (наприклад, додаючи нові шари до нейронної мережі), джерела даних, методи інженерії ознак чи гіперпараметри (наприклад, щільність вибірки для навчання). Кожна така ітерація є повноцінним експериментом, який необхідно провести, зафіксувати та проаналізувати.

Саме ця ітеративна фаза є найбільш вузьким місцем у всьому процесі розробки. Вона вимагає значних та, як правило, найдорожчих обчислювальних ресурсів (GPU/TPU), займає левову частку календарного часу проекту та є джерелом більшості технічних та методологічних проблем [6]. Ефективність всього процесу розробки ШІ напряму залежить від того, наскільки швидко, надійно, відтворювано та ресурсоефективно організація може проводити ці

експерименти. Дана робота зосереджена саме на аналізі архітектурних підходів до автоматизації та оптимізації цього критичного циклу: навчання – оцінка.

1.2 Проблема відтворюваності та ізоляції середовищ

Першою та найбільш фундаментальною проблемою, що виникає у фазі навчання та експериментів, є так криза відтворюваності [7]. У наукових та інженерних колах це явище відоме під неформальною назвою «на моїй машині працювало». Суть проблеми полягає в тому, що дослідник може отримати видатні результати (наприклад, високу точність моделі «accuracy 95 %») на своєму локальному комп’ютері, але ані він сам за місяць, ані його колеги не можуть відтворити цей результат, запускаючи той самий програмний код на іншій машині. Це нівелює наукову цінність експерименту та унеможливлює командну роботу та перехід до наступних етапів життєвого циклу [8].

Джерелом цієї проблеми є надзвичайна крихкість та складність програмного оточення, необхідного для сучасного машинного навчання. Результат експерименту залежить не лише від програмного коду моделі, але й від десятків зовнішніх, часто неявних, факторів середовища. До них належать версія мови програмування (наприклад, Python 3.8 проти 3.9), версії ключових бібліотек (наприклад, PyTorch 1.9 проти 2.0), версії системних залежностей (наприклад, бібліотеки CUDA 11 проти CUDA 12), і навіть версія драйвера графічного процесора NVIDIA або змінні середовища операційної системи. Ця ситуація, відома в інженерії як «пекло залежностей» (англ. «dependency hell»), робить процес розробки хаотичним, непередбачуваним і невідтворюваним [9].

Як проаналізовано у науковій публікації [1, с. 92-97], ключовим технологічним рішенням для подолання цієї проблеми є застосування методології контейнеризації. Контейнеризація забезпечує ізоляцію та відтворюваність середовища, що є критично важливим для наукових досліджень та розробки проектів машинного навчання.

Контейнеризація, що реалізується такими інструментами, як Docker, дозволяє інкапсулювати (тобто ізолювати) програмний код, усі бібліотеки, системні залежності та конфігураційні файли в єдиний, портативний артефакт. Цей артефакт називається образом і, по суті, є зліпком ідеального середовища. Цей образ потім можна запустити як контейнер – легковаговий, ізольований процес – на будь-якій обчислювальній машині, де встановлено контейнерний рушій (наприклад, Docker Engine).

Як показано на рисунку 1.2 та проаналізовано в [1, с. 92-97], на відміну від віртуальних машин (ВМ), контейнери не потребують емуляції цілого апаратного забезпечення та запуску окремої, повноцінної гостьової операційної системи. Вони працюють безпосередньо на ядрі хостової ОС, використовуючи механізми ізоляції просторів імен та груп керування. Це робить їх значно легшими (десятки мегабайт проти гігабайт для ВМ) та швидшими (запуск за секунди проти хвилин) [1, с. 92-97].

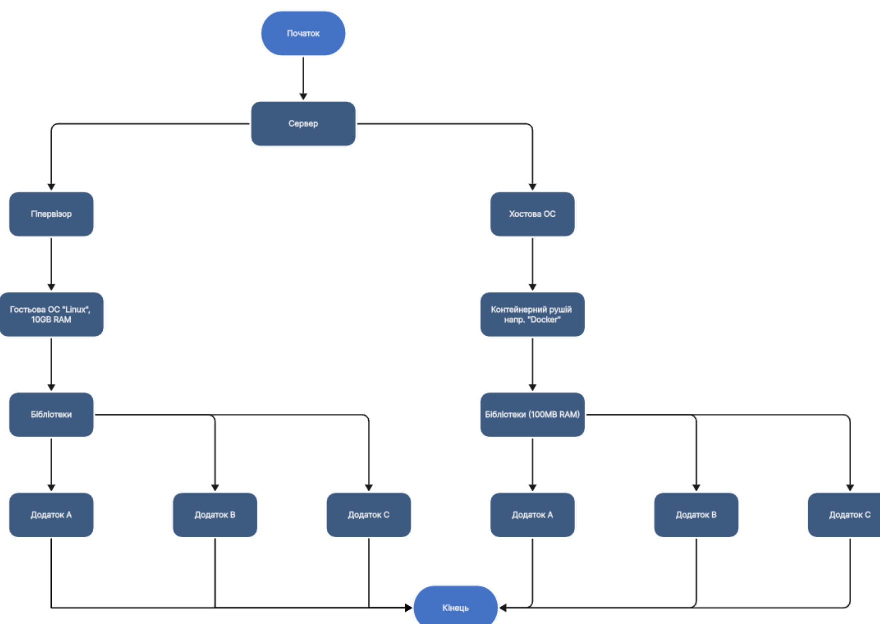


Рисунок 1.2 – Порівняння архітектури віртуальної машини (ВМ) та контейнера

Використання контейнеризації для програм машинного навчання надає вирішальні переваги. По-перше, це гарантована відтворюваність: кожен експеримент запускається в абсолютно ідентичному, заздалегідь визначеному

середовищі, що повністю усуває конфлікти версій та непередбачувані результати. По-друге, це незалежність від платформи: образ, створений на локальній машині (Windows або macOS), буде ідентично працювати на сервері (Linux) або у будь-якій хмарній інфраструктурі. По-третє, це ізоляція: різні проекти на одній машині (наприклад, один, що вимагає TensorFlow 1.x, та інший, що вимагає PyTorch 2.x) можуть працювати паралельно, не конфліктуючи.

Таким чином, контейнеризація є обов’язковою базовою вимогою та технологічною основою для побудови будь-якої сучасної платформи автоматизації навчання. Вона вирішує першу фундаментальну проблему – проблему відтворюваності середовища, що є запорукою наукової валідності експериментів.

1.3 Проблема неефективного використання апаратних ресурсів

Вирішення проблеми відтворюваності за допомогою контейнеризації гарантує, що експеримент запуститься коректно, але не вирішує проблему як він запускається. Це підводить до другої фундаментальної проблеми – проблеми ефективності, яка має чітко виражений економічний характер та безпосередньо впливає на швидкість досліджень.

Навчання сучасних моделей, особливо у сферах комп’ютерного зору (наприклад, CNN, Transformers) та обробки природних мов (NLP) (наприклад, GPT, BERT), є надзвичайно обчислювально трудоємкою задачею. Ці обчислення проводяться на спеціалізованих дорогих апаратних прискорювачах, зокрема графічних процесорах (GPU) або тензорних процесорах (TPU). Вартість таких прискорювачів є високою, а їхній час роботи – обмеженим та коштовним (особливо при оренді у хмарних провайдерів).

Традиційний, або «ручний», підхід до проведення експериментів, навіть з використанням контейнерів, є вкрай неефективним з точки зору використання цих дорогих ресурсів. Типовий робочий процес дослідника, що працює з виділеним сервером, виглядає наступним чином:

1) дослідник вручну запускає команду (наприклад, «docker run ...») для першого експерименту з певним набором гіперпараметрів;

2) апаратний ресурс (напр., GPU) завантажується на 100 % протягом тривалого часу (наприклад, 2 години), виконуючи обчислення;

3) експеримент завершується, ресурс GPU звільняється, його завантаження падає до 0 %. Ресурс починає простоювати;

4) дослідник витрачає час (наприклад, 1 годину) на ручний аналіз отриманих результатів: копіювання логів, побудову графіків у локальному Jupyter Notebook та прийняття рішення про наступний крок (наприклад, «зменшити швидкість навчання вдвічі»).

Як ілюструє рисунок 1.3, у такому 6-годинному циклі дорогий апаратний прискорювач третину часу (33 %) простоював, очікуючи на ручне втручання людини. У масштабах дослідницьких відділів та компаній це призводить до колосальних фінансових втрат, марнування обчислювальних потужностей та суттєвого затягування термінів розробки (проект, який міг би завершитись за 4 години, розтягується на 6).

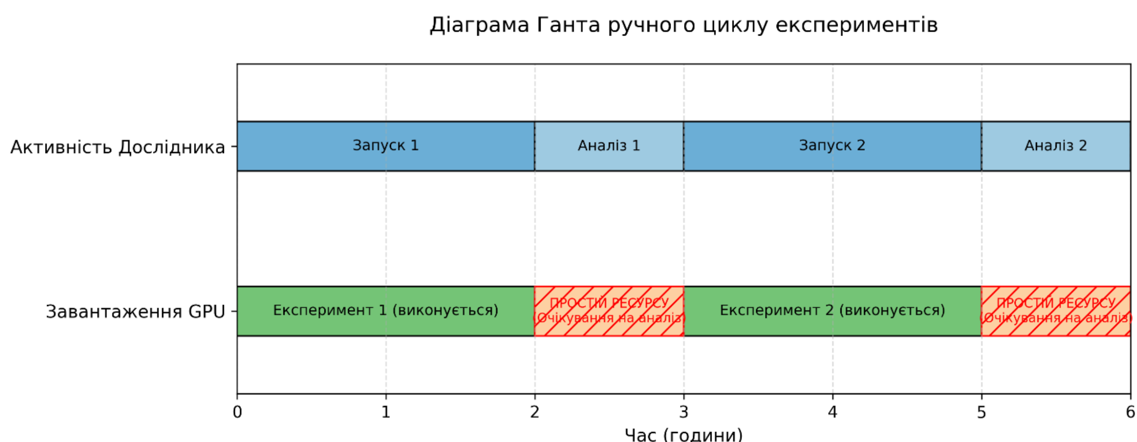


Рисунок 1.3 – Діаграма Ганта ручного циклу експериментів

Ця проблема формулює другу ключову вимогу до архітектури: платформа повинна забезпечувати автоматизовану оркестрацію контейнеризованих завдань. Необхідний механізм, здатний приймати від дослідника не одне

завдання, а пакет завдань (наприклад, 10 експериментів для пошуку по сітці) та автоматично запускати їх у режимі «черги завдань» (що є технічною реалізацією патерну Job Queue). Така система має самостійно відстежувати стан апаратних ресурсів і негайно (з нульовим або близьким до нуля часом простою) запускати наступний експеримент з черги, щойно попередній завершився. Це дозволяє максимізувати утилізацію обладнання та звільнити час дослідника для більш важливих завдань, ніж ручний моніторинг.

1.4 Проблема комплексності аналізу гетерогенних моделей

Навіть за умови вирішення проблем відтворюваності та ефективності, залишається третя, найбільш наукомістка проблема – проблема аналізу. Ефективна платформа має не лише швидко та надійно виконувати експерименти, але й надавати досліднику потужні інструменти для їх порівняння та прийняття обґрунтованих рішень. Без якісного аналізу швидке виконання експериментів перетворюється на спам неструктурованими даними [10].

Складність цього аналізу часто недооцінюється. Типова задача дослідника – це не просто тонка оптимізація гіперпараметрів однієї, заздалегідь обраної моделі. Цей процес, відомий як пошук по сітці (Grid Search) або Hyperparameter Tuning, є лише другим етапом оптимізації. Йому передує перший, набагато важливіший етап: вибір самої архітектури моделі.

На початкових етапах проєкту відповідь на запитання «Який тип моделі є найбільш придатним для цих даних?» зазвичай не є очевидною, оскільки вибір підходу залежить від низки факторів, пов’язаних як із природою даних, так і з цільовими характеристиками майбутньої системи. На цьому етапі досліднику необхідно врахувати можливі структурні особливості вибірки, потенційну наявність прихованих патернів, рівень шуму, стабільність часових залежностей та інші властивості, що можуть істотно вплинути на результативність моделей різних класів.

Зокрема, постає питання, чи забезпечить задачам прогнозування попиту кращу ефективність згорткова нейронна мережа (CNN), яка здатна виявляти багаторівневі часові закономірності та вловлювати локальні залежності у даних. У разі, коли поведінка часових рядів має складний характер, CNN може продемонструвати суттєву перевагу завдяки своїй здатності автоматично формувати релевантні ознаки без необхідності трудомісткого ручного пошуку закономірностей.

Водночас класичні ансамблеві методи, зокрема RandomForest, часто показують високу стабільність і прогнозну здатність на табличних даних зі змішаними типами ознак. Їхня інтерпретованість та стійкість до вибіркових шумів роблять їх привабливими для практичних застосувань, де важливим є не лише отримання точного прогнозу, а й можливість зрозуміти, які фактори суттєво впливають на результат моделі.

Окрім того, інколи доцільно розглядати застосування більш простих алгоритмів, таких як методи, основані на вимірюванні відстані (наприклад, KNeighbors). Попри свою концептуальну простоту, такі підходи можуть забезпечувати цілком прийнятні результати, особливо коли структура даних є відносно гладкою, а часові залежності – стабільними. Додатковою перевагою таких моделей є їхня невисока обчислювальна складність під час навчання, що дозволяє проводити велику кількість експериментів за обмеженого часу та апаратних ресурсів.

Відповідь на це питання є нетривіальною і, як показує практика, напряму залежить від характеристик вхідних даних: їх обсягу, кількості ознак, частоти транзакцій, наявності нелінійних залежностей та стабільності цільової змінної. Цей феномен був детально досліджений в опублікованій автором науковій статті «System of dynamic optimization pricing by machine learning» [2], у якій розглянуто комплексну проблему вибору оптимального алгоритму машинного навчання для задач динамічного ціноутворення на основі реальних комерційних даних за восьмимісячний період. Стаття аналізує, яким чином різні моделі – KNeighbors Regressor, Random Forest Regressor, Decision Tree Regressor, Linear Regressor та

Support Vector Regressor реагують на зміну обсягу транзакцій, рівня варіативності цін та структури даних, а також демонструє, що точність прогнозу істотно залежить від поєднання цих факторів. Зокрема, у роботі показано, що модель KNeighbors є найбільш ефективною при невеликій кількості транзакцій, тоді як ансамблеві методи (Decision Tree, Random Forest) забезпечують найкращі результати на великих вибірках. Стаття використовується як приклад для ілюстрації багатофакторності та складності задачі вибору моделі, з якою стикається дослідник при побудові системи динамічного ціноутворення, коли потрібно не лише оцінити якість алгоритмів, але й врахувати структурні властивості даних, їх нерівномірність, наявність цінових коливань та залежність точності моделей від ступеня насичення вибірки.

Як показано в дослідженні [2] та узагальнено в таблиці 1.1, при аналізі реальних даних для задачі динамічного ціноутворення було виявлено, що не існує єдиного найкращого алгоритму.

Таблиця 1.1 – Приклад використання різних алгоритмів машинного навчання для вирішення одного завдання [2]

№	Тип моделі	Обсяг вибірки	MSE	R^2
1	KNeighbors Regressor	94 (Мала)	0,0091	0,72
2	RandomForest Regressor	94 (Мала)	0,0124	0,62
3	KNeighbors Regressor	1727 (Велика)	0,0345	0,43
4	RandomForest Regressor	1727 (Велика)	0,0048	$> 0,9$

На малих наборах даних (94 транзакції) найкращі результати (найменшу помилку MSE = 0,0091 та найвищий коефіцієнт детермінації $R^2 = 0,72$) продемонстрував метод KNeighbors Regressor. Однак, при переході до великих наборів даних (1727 транзакцій), той самий KNeighbors Regressor виявився значно менш ефективним, тоді як ансамблеві методи, зокрема RandomForest Regressor та DecisionTree Regressor, показали найвищу точність ($R^2 > 0,9$).

Цей приклад доводить, що ефективна MLOps-платформа для дослідницьких цілей не може бути обмежена лише тренуванням одного типу моделей. Платформа, що підтримує лише PyTorch (для CNN), призвела б до вибору неоптимальної моделі у наведеному прикладі. Таким чином, формулюється третя ключова вимога: платформа повинна надавати інструменти для автоматизованого запуску, тестування та, що найважливіше, агрегованого аналізу гетерогенних моделей (наприклад, PyTorch CNN проти sklearn RandomForest) в єдиному, зручному для порівняння інтерфейсі.

На основі проведеного аналізу, визначено три ключові проблеми, які повинна вирішувати пропонована архітектура:

1) відтворюваність. Хаотичні середовища призводять до невідтворюваних результатів;

2) ефективність ресурсів. Ручне керування призводить до простою дорогих апаратних прискорювачів;

3) комплексність аналізу. Необхідність порівняння не лише гіперпараметрів, але й принципово різних типів моделей для пошуку глобально оптимального рішення.

РОЗДІЛ 2
АНАЛІЗ ІСНУЮЧИХ АРХІТЕКТУРНИХ ПІДХОДІВ ДО АВТОМАТИЗАЦІЇ НАВЧАННЯ

2.1 Базовий підхід: Контейнеризація та ручна оркестрація

Першим рівнем архітектурних рішень, що розглядається в інженерній практиці, є базовий підхід, який ґрунтується на прямій контейнеризації та ручному управлінні запуском. Цей підхід є поширеним для невеликих проектів або на початкових етапах дослідження (у форматі «доказу концепції»), оскільки він вимагає мінімальних інфраструктурних налаштувань.

Характерний приклад такої архітектури був детально описаний у рамках науково-дослідної роботи «Studying the Practices of Deploying Machine Learning Projects on Docker» [11]. Дана архітектура використовує Docker для інкапсуляції навчального скрипта та його залежностей, а також інструмент Docker Compose для декларативного опису та одночасного запуску кількох контейнерів (наприклад, самого навчального сервісу та сервісу зберігання, такого як MinIO).

На рисунку 2.1 зображене схематичне представлення архітектури навчання моделей використовуючи Docker Compose. Фундаментально даний метод побудований на використанні Docker контейнерів, що мають подібні чи ідентичні базові налаштування та основу, на якій запускаються. Однак, вони дозволяють вносити певні зміни в середовище виконання, які використовуються як налаштування, що зчитуються виконувальним скриптом.

У результаті, в ідентичних по побудові контейнерах виконуються ідентичні скрипти, однак вони передбачають зчитування змінних з середовища (що в Unix-подібних системах є передбачуваною та обов’язковою складовою системи). У даному випадку розробник зобов’язаний буди повідомлений MLOps спеціалістом про те, як буде відбуватись запуск навчання моделей, щоб розробник підлаштував налаштування скрипта для зчитування змінних середовища.

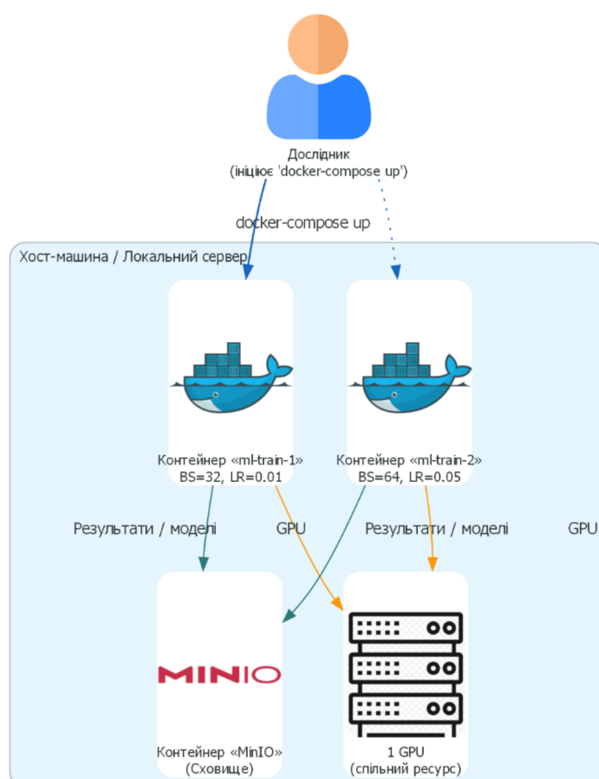


Рисунок 2.1 – Архітектурна схема базового підходу використовуючи Docker Compose

По-перше, ця архітектура успішно вирішує проблему відтворюваності. Завдяки використанню Docker-контейнерів, кожен експеримент виконується у повністю ізольованому середовищі з фіксованими версіями бібліотек. Це усуває пекло залежностей та гарантує, що експеримент можна точно відтворити на будь-якій машині, де встановлено Docker Engine.

По-друге, архітектура частково або не вирішує проблему оптимізації ресурсів. Як показано у звіті [11], запуск двох експериментів паралельно дійсно скорочує загальний час у порівнянні з послідовним ручним запуском. Однак, цей підхід не масштабується і не реалізує черги завдань. Docker Compose вимагає статичного визначення сервісів у файлі конфігурації (наприклад «ml-train-1», «ml-train-2»). Для запуску двадцяти експериментів (наприклад, для пошуку по сітці) досліднику доведеться або вручну модифікувати файл, додавши «ml-train-3»...«ml-train-20», або запустити всі 20 сервісів одночасно. Останній сценарій призведе до збою через вичерпання апаратних ресурсів (наприклад, пам'яті

GPU), оскільки Docker Compose не має вбудованого планувальника чи механізму черги. Таким чином, оптимізація залишається ручною. Цей підхід демонструє фундаментальну ваду: він забезпечує ізоляцію, але не оркестрацію.

По-третє, архітектура не вирішує проблему комплексності аналізу. Немає єдиної UI-панелі для порівняння метрик. Досліднику все одно доведеться вручну підключатися до MinIO, завантажувати файли логів або артефакти та порівнювати їх у локальному Jupyter Notebook. Це стає неможливим при порівнянні десятків гетерогенних моделей, як того вимагає задача, обґрунтована необхідністю аналізу моделей різного типу [2].

У таблиці 2.1 наведено порівняльний аналіз базової архітектури (Docker Compose) з традиційним підходом за ключовими критеріями. Таблиця відображає стан аспектів, аналітичні висновки та невирішені проблеми. Переваги контейнеризації, як гарантована відтворюваність, поєднуються з недоліками, такими як статична конфігурація сервісів, що не підтримує динамічне масштабування без інструментів на кшталт Kubernetes [12].

Аналіз ґрунтується на емпіричних даних тестових запусків, де паралельний режим скорочує час, але не оптимізує ресурси при великій кількості завдань. Таблиця акцентує перехід до оркестрованих систем для автоматизації, де черги завдань і UI-інтерфейси є невід’ємними, як описано в літературі з MLOps [14].

Таблиця 2.1 – Аналіз базової архітектури (Docker Compose) у порівнянні з традиційним підходом

Критерій аналізу (Визначена проблема)	Стан у Традиційному підході (Ручний запуск)	Стан у Базовій архітектурі (Docker Compose)	Аналітичний висновок (Проблема, що залишається)
1	2	3	4
Відтворюваність середовища	Відсутня. Результати залежать від локального середовища, версій бібліотек та драйверів.	Вирішено. Інкапсуляція в Docker-образ гарантує 100% відтворюваність середовищ.	Проблема середовища вирішена. Це єдина проблема, яку даний підхід вирішує повністю.

Продовження таблиці 2.1

1	2	3	4
Оптимізація ресурсів	Відсутня. 100% ручне керування. Високий час простою GPU між запусками під час аналізу.	Частково вирішено. Дозволяє паралельний, але не послідовний запуск. Відсутній механізм черги завдань.	Проблема не вирішена. Підхід не масштабується (спроба запуску 20 сервісів призведе до збою) і вимагає ручного керування (статичний «.yml»-файл).
Комплексність аналізу	Відсутня. Артефакти та логи зберігаються хаотично в локальній файловій системі.	Частково вирішено. Артефакти централізовано зберігаються в «MinIO».	Проблема не вирішена. Відсутня UI-панель та шар агрегації метрик. Аналіз результатів залишається ручним (потрібно підключатися до MinIO та порівнювати файли).

Таким чином, базовий підхід на основі Docker Compose є корисним для демонстрації концепції контейнеризації, але є архітектурно недостатнім для побудови повноцінної, автоматизованої та ефективної платформи для наукових досліджень, оскільки він не надає адекватних рішень для оптимізації ресурсів та комплексного аналізу.

2.2 Аналіз монолітних MLOps-платформ

Наступним щаблем еволюції архітектур є монолітні платформи, що часто надаються у форматі «Платформа як послуга». Ці рішення являють собою комплексні, «коробкові» продукти, які намагаються вирішити всі проблеми життєвого циклу MLOps (від збору даних до моніторингу) в рамках єдиної, тісно інтегрованої екосистеми [14].

До цієї категорії належать як комерційні хмарні сервіси (наприклад, Amazon SageMaker, Google Vertex AI, Azure Machine Learning), так і платформи з відкритим кодом (найбільш яскравим представником є Kubeflow, аріхтектура роботи якого представлена на рисунку 2.2).

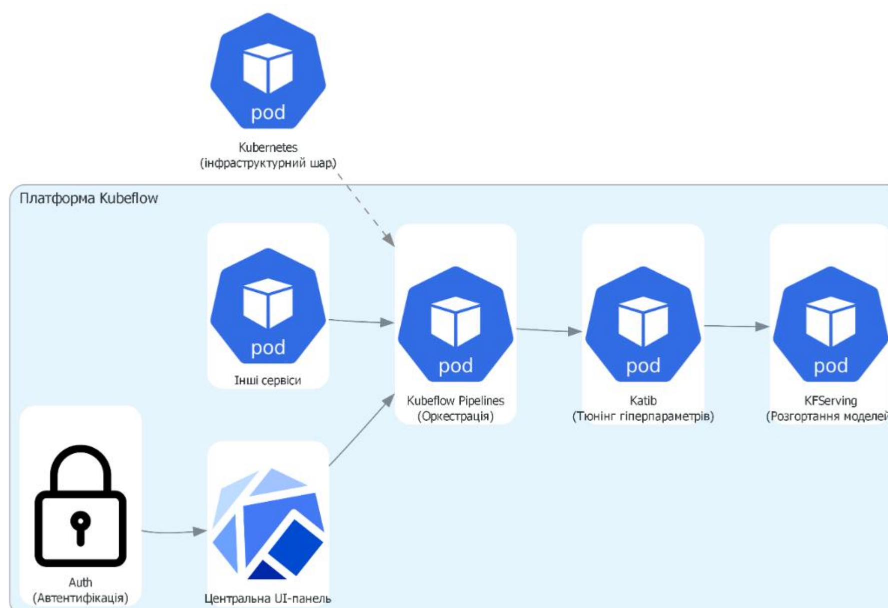


Рисунок 2.2 – Архітектура монолітної платформи (приклад Kubeflow)

Аналіз монолітних платформ показує, що вони дійсно пропонують рішення для всіх трьох визначених проблем. По-перше, вони зазвичай побудовані на основі Kubernetes, що вирішує проблему оптимізації ресурсів, надаючи потужну оркестрацію та керування чергою завдань. По-друге, вони включають вбудовані UI-панелі та механізми відстеження, що вирішує проблему комплексності аналізу. По-третє, вони використовують контейнеризацію, що вирішує проблему відтворюваності.

Незважаючи на ці переваги, монолітні платформи мають низку суттєвих архітектурних та методологічних недоліків, які роблять їх неоптимальними для гнучких наукових досліджень, що є метою даної роботи.

Перший недолік – це надзвичайно висока складність та високий поріг входу. Такі платформи, як Kubeflow, вимагають глибоких знань в адмініструванні Kubernetes та управлінні інфраструктурою лише для того, щоб платформа запрацювала. Налаштування мережевої взаємодії, автентифікації та безпеки компонентів (так звані «day-2 operations») є складним завданням, що відволікає дослідника від основної мети – проведення експериментів [15].

Другий, і більш значущий, недолік – це архітектурна негнучкість та нав’язування методології. Монолітні платформи часто є думковмісними, тобто

вони нав'язують користувачеві свій єдино правильний спосіб роботи. Наприклад, Kubeflow активно просуває власну екосистему TensorFlow Extended (TFX) та специфічний спосіб визначення компонентів. Це створює значні труднощі при спробі вирішити проблему комплексності аналізу, а саме – порівняння гетерогенних моделей. Стає інженерно складно в рамках одного експерименту порівняти модель, написану на PyTorch, з моделлю на Scikit-learn оскільки вони не вписуються в єдиний, жорстко визначений пайплайн платформи. Це створює тертя та сповільнює дослідницьку фазу, для якої, власне, платформа і призначалася [16].

Третій недолік, що стосується комерційних платформ (SageMaker, Vertex AI), – це прив'язка до постачальника (термін, відомий як «vendor lock-in»). Дослідницький код, написаний з використанням пропрієтарних API (наприклад, SageMaker API), неможливо буде запустити поза інфраструктурою цього хмарного провайдера. Це створює стратегічні ризики для наукових установ та компаній, обмежуючи їхню можливість переходу до інших, потенційно більш дешевих або потужних, обчислювальних середовищ [17].

Таким чином, монолітні платформи, вирішуючи технічні проблеми, створюють нові, методологічні, що робить їх надлишковими та негнучкими для динамічних дослідницьких завдань, які вимагають швидкої перевірки принципово різних гіпотез та архітектур.

2.3 Аналіз спеціалізованих інструментів (комбінований підхід)

Між двома крайнощами – надто простим підходом Docker Compose та надто складним монолітним Kubeflow – існує третій, комбінований підхід. Цей підхід, відомий в інженерній практиці кращий у своєму класі [18], полягає у тому, щоб не використовувати одну платформу, яка вміє все, а натомість комбінувати набір незалежних, спеціалізованих інструментів, кожен з яких є найкращим у своїй вузькій галузі.

Цей підхід дозволяє створити гнучку архітектуру, адаптовану під конкретні потреби. Інструменти цієї категорії можна логічно розділити відповідно до проблем, що вирішуються.

Інструменти для Оркестрації (вирішення проблеми оптимізації ресурсів): Ця категорія інструментів фокусується виключно на запуску, моніторингу та керуванні залежностями між завданнями. Вони є багатофункціональними планувальниками:

− Argo Workflows: передбачений для Kubernetes (K8s) інструмент, що дозволяє визначати складні робочі процеси у вигляді спрямованих ациклічних графів. Його перевага – глибока інтеграція з K8s (кожен крок є окремим подом), що робить його ефективним для пакетних обчислень [19];

− Apache Airflow: зрілий та популярний інструмент для оркестрації, написаний на Python. Також використовує концепцію DAG. Його перевага – потужний UI для візуалізації залежностей та перезапуску невдалих завдань, хоча його інтеграція з K8s є менш функціонально реалізованою, ніж у Argo [20].

Ці інструменти є чистими оркестраторами. Вони не мають жодного уявлення про те, що саме вони запускають. Вони не знають, що таке модель, гіперпараметр чи метрика точності. Вони просто запускають скрипти у визначеному порядку. Таким чином, вони блискуче вирішують проблему оптимізації ресурсів, але повністю ігнорують проблему комплексності аналізу.

Інструменти для відстеження (вирішення проблеми комплексності аналізу): Ця категорія фокусується виключно на зборі, зберіганні та візуалізації результатів експериментів.

− MLflow Tracking: один з найпопулярніших інструментів у цій категорії. Надає сервер відстеження як централізований API для логування та UI-панель для візуального порівняння. Його перевага – простота та гнучкість; він не залежить від того, де виконується код (локально, в Docker чи в K8s) [21];

− Data Version Control (DVC): інструмент, що фокусується на версіонуванні даних та моделей за допомогою Git (використовуючи Git-LFS або

хмарні сховища). Його перевага – тісна інтеграція з Git, що дозволяє точно відтворити стан даних для будь-якого коміту [22].

Ці інструменти є трекерами (тим, що слідкує за змінною). Вони блискуче вирішують проблему комплексності аналізу, але не вирішують проблему оптимізації ресурсів. Наприклад, вбудована в MLflow команда «mlflow run» є дуже базовою і не надає багатофункціонального механізму черги чи розподіленого виконання на кластері, як це робить Kubernetes.

Як видно з таблиці 2.2, на ринку існуючих рішень утворилася чітка спеціалізація: інструменти або добре виконують завдання, або добре аналізують результати, але не поєднують обидві функції в єдиному гнучкому рішенні. Цей аналіз показує, що жоден окремий спеціалізований інструмент не вирішує всіх трьох проблем одночасно.

Таблиця 2.2 – Матриця порівняння спеціалізованих інструментів

Інструмент	Основне призначення	Чи вирішує проблему оптимізації	Чи вирішує проблему аналізу
Argo Workflows	Оркестрація («Job Queue»)	Так	Ні
Apache Airflow	Оркестрація («Job Queue»)	Так	Ні
Mlflow Tracking	Відстеження (UI, API)	Ні	Так
DVC	Відстеження (Версіонування даних)	Ні	Частково (лише версіонування)

2.4 Обґрунтування гібридного підходу

Проведений у даному розділі аналіз існуючих архітектурних рішень дозволяє сформулювати ключові висновки та виявити незаповнену інженерну нішу, що обґрунтовує необхідність розробки нової, гібридної архітектури.

Аналіз показав існування цілого спектру рішень, кожне з яких має суттєві вади при застосуванні для гнучких наукових досліджень.

По-перше, базовий підхід на основі «Docker Compose», детально проаналізований у підрозділі 2.1 [2, 13, 14, 15], є недостатньо функціональним. Хоча він вирішує базову проблему відтворюваності, він повністю провалюється у вирішенні проблем оптимізації ресурсів (через відсутність черги завдань) та комплексного аналізу (через відсутність UI-панелі).

По-друге, монолітні платформи, проаналізовані у підрозділі 2.2 (на прикладі Kubeflow), є надлишково складними та негнучкими. Вирішуючи всі три проблеми, вони роблять це ціною високого порогу входу та нав'язування жорстких та надлишкового комплексних пайплайнів. Це ускладнює проведення ключового дослідницького завдання – швидкого порівняння гетерогенних моделей, як того вимагає проблема, доведена необхідністю аналізу моделей різного типу [2].

По-третє, спеціалізовані інструменти (проаналізовані у підрозділі 2.3) є потужними, але неповноцінними окремо один від одного. Інструменти оркестрації (наприклад, Argo Workflows) не мають можливостей аналізу, а інструменти відстеження (наприклад, MLflow) не мають потужних механізмів оркестрації.

Як видно з рисунку 2.3, існує потреба у системі, яка б поєднувала високу гнучкість (як у простих інструментів) та повноту функціоналу (як у монолітних платформах).

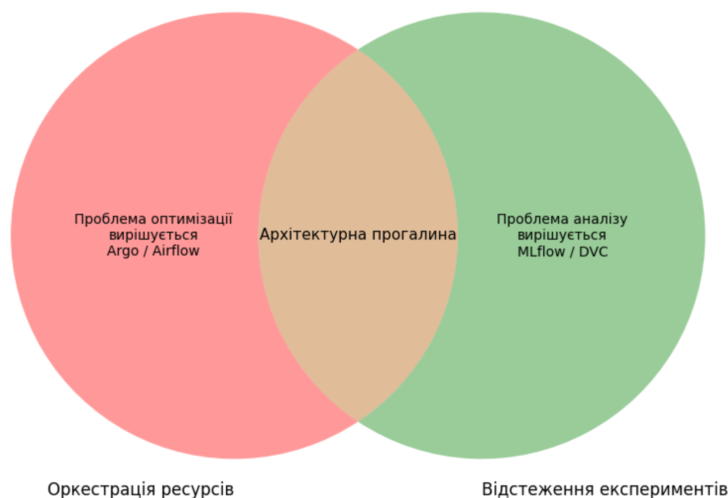


Рисунок 2.3 – Спектр існуючих рішень та візуаліація виявленої прогалини

На основі цього висновку, формулюється пропонований підхід для даної кваліфікаційної роботи. Замість того, щоб обирати один з існуючих компромісних варіантів, пропонується спроектувати та валідувати нову, гібридну архітектуру, яка заповнює цю «прогалину».

Ця гібридна архітектура базується на комбінації найкращих у своєму класі компонентів з відкритим кодом. Вона поєднує:

1) Kubernetes (як в підрозділі 2.2, але без складності Kubeflow) як чисту підсистему оркестрації для вирішення проблеми неефективного використання ресурсів;

2) MLflow (як в підрозділі 2.3) як чисту підсистему відстеження для вирішення проблеми комплексного аналізу;

3) Docker (як в підрозділах 1.2 та 2.1) як базову технологію для вирішення проблеми відтворюваності.

Такий синтез дозволяє створити платформу, що є одночасно багатофункціональною (має оркестрацію K8s та аналіз MLflow) і гнучкою (не нав’язує жорстких пайплайнів і дозволяє запускати будь-які гетерогенні моделі, обгорнуті в Docker-контейнер).

РОЗДІЛ 3
ПРОЕКТУВАННЯ, РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНА ВАЛІДАЦІЯ АВТОМАТИЗОВАНОЇ MLOPS-ПЛАТФОРМИ

На основі аналізу існуючих рішень та теоретичних засад, розглянутих в розділі 1-2, даний розділ присвячено проектуванню, практичній реалізації та експериментальній валідації пропонованої архітектури автоматизованої системи навчання моделей штучного інтелекту. Метою є створення платформи, що вирішує ключові проблеми відтворюваності експериментів та неефективного використання апаратних ресурсів.

Якщо аналіз у розділі 2 показано, що існуючі монолітні платформи (наприклад, Kubeflow) часто є надлишково складними, а простіші підходи (тобто Docker Compose) не вирішують проблем оптимізації, то в даному розділі пропонується гібридна, гнучка архітектура, яка поєднує найкращі компоненти з відкритим кодом.

3.1 Архітектура платформи та стек технологій

Практична реалізація гібридної архітектури, обґрунтованої у попередньому розділі, базується на інтеграції чотирьох ключових технологічних компонентів. Вони формують чотири логічні підсистеми, що вирішують визначені проблеми: Kubernetes як основа підсистеми оркестрації та оптимізації ресурсів; MLflow як центральний елемент підсистеми відстеження та аналізу; MinIO як підсистема зберігання артефактів; та Docker як підсистема виконання та ізоляції.

На рисунку 3.1 зображено концептуальну взаємодію цих чотирьох компонентів. Вона ілюструє, як Kubernetes виступає в ролі базового інфраструктурного шару для оркестрації, Docker – як підсистема виконання, інкапсульована всередині K8s, а MLflow та MinIO функціонують як

централізовані сервіси для відстеження та зберігання, взаємодіючи з усіма процесами, що виконуються.

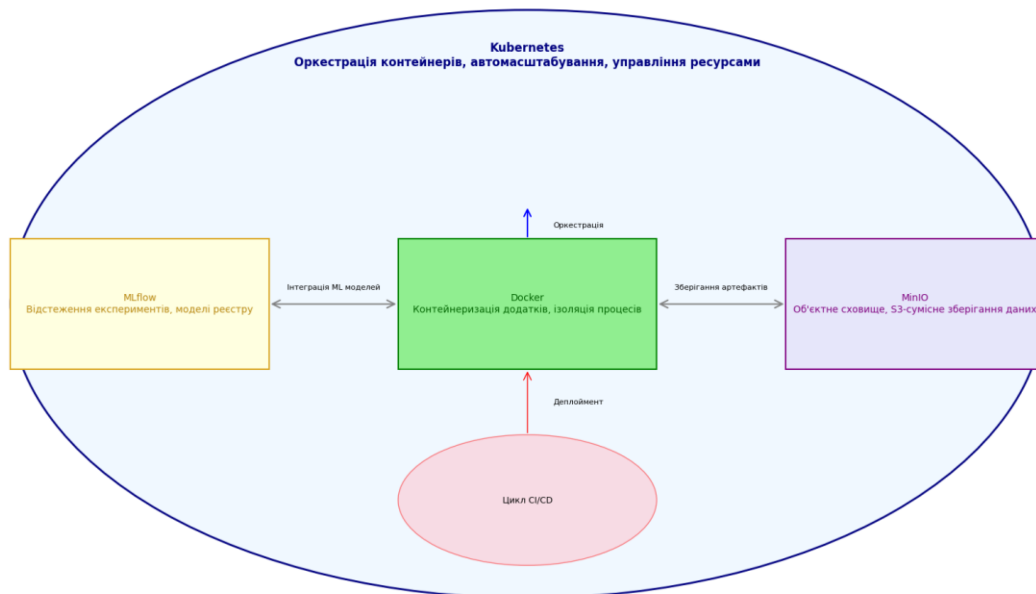


Рисунок 3.1 – Огляд пропонованої архітектури

3.1.1 Kubernetes як технологічна основа динамічної оркестрації та оптимізації ресурсів

Основним компонентом платформи обрано Kubernetes, оскільки він надає фундаментальні механізми для вирішення проблеми оптимізації ресурсів. На відміну від статичних систем, які визначаються у технології Docker Compose, де паралельний запуск N завдань (де N – довільна кількість) на одному ресурсі призводить до конфлікту та непередбачуваної зупинки, K8s є динамічним оркестратором, що працює за декларативною моделлю [23].

Ключова практична перевага K8s для наукових обчислень – це спроектований та передбачуваний об’єкт «Задача» та вбудований планувальник задач. Процес виглядає наступним чином [23]:

1) прийом завдань: платформа приймає N запитів на навчання у вигляді маніфестів об’єктів Job;

2) розподіл ресурсів: планувальник K8s аналізує доступні ресурси (CPU, RAM, а у промисловій версії – GPU) і виконує лише M завдань одночасно (де M - кількість доступних ресурсів, наприклад, 1 CPU);

3) керування чергою: решта N-M завдань чекають у черзі зі статусом Pending. Схематичне зображення даного процесу зображено на рисунку 3.2;

4) максималізація ефективності: щойно планувальник фіксує, що Job завершив роботу зі статусом Completed, він негайно виділяє звільнений ресурс наступному Job з черги.

Це дозволяє досягти максимальної (близько 100 %) ефективності використання дорогого обладнання, усуваючи простої між ручними запусками. Для демонстрації та валідації даної логіки використовується полегшена версія Minikube, яка надає повний функціонал K8s API та механізмів планування задач в локальному середовищі.

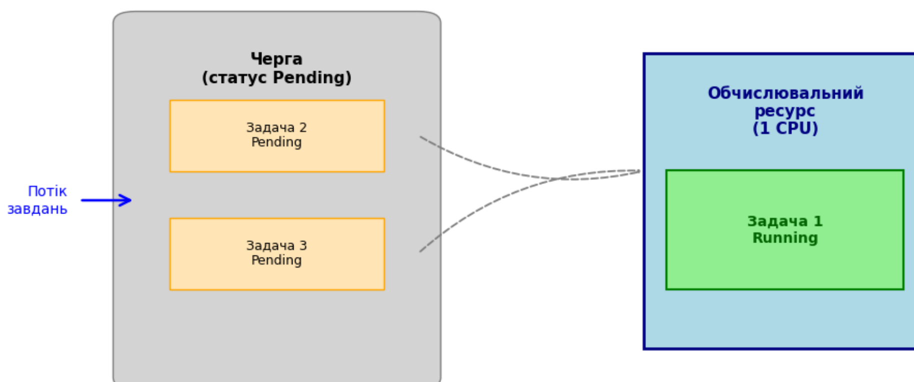


Рисунок 3.2 – Візуалізація концепції черги завдань в Kubernetes

3.1.2 MLflow як централізована підсистема відстеження та аналізу експериментів

Для вирішення проблеми аналізу та порівняння обрано MLflow. MLflow надає чотири ключових компоненти (Tracking, Projects, Models, Registry), з яких в даній архітектурі використовуються два [24].

По-перше, MLflow Tracking виступає як центральний компонент підсистеми відстеження. Його Tracking Server надає єдиний REST API та базу

даних для реєстрації всіх аспектів експерименту: гіперпараметрів (наприклад, learning_rate), метрик (наприклад, loss, R-squared), тегів та ін. По-друге, MLflow UI вирішує проблему візуалізації даних та результатів для аналізу. UI слугує візуальним інструментом для сортування, фільтрації та порівняння десятків експериментів, надаючи досліднику можливість миттєво ідентифікувати найкращі моделі.

MLflow інтегрується зі сховищем артефактів (в даній роботі використовується MinIO), що дозволяє прив’язати збережені моделі безпосередньо до їхніх метрик в UI. Кожен запуск «train.py» (навчального скрипту) у K8s, таким чином, стає ізольованим, але його результати негайно надсилаються у цей централізований хаб (рис. 3.3).

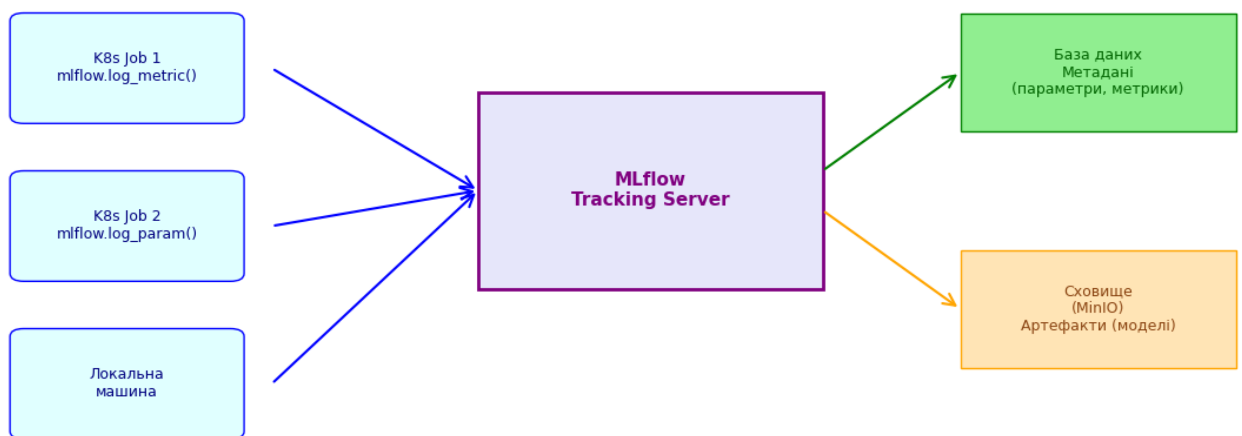


Рисунок 3.3 – Потік даних у MLflow Tracking

На рисунку 3.4 зображений приклад інформації про експерименти у MLflow, що запущено та розроблено для аналізу в даній роботі на експериментальних навчальних скриптах з випадково обраними даними та алгоритмами). З даного рисунку видно, що платформа динамічно здатна зчитувати вихідні дані з моделі та відображати їх у панелі результатів. Відображена таблиця містить стовпчики та можливість фільтрування відображувальних стовпчиків, що отримуються динамічно з скриптів та моделей

розробленим дослідником. Завдяки даній концепції непотрібно додаткових налаштувань платформи чи скрипту, адже дослідник передбачає в програмному коді тренування моделі які метрики він визначає, а MLflow самостійно сформує нові стовпці, якщо вони досі не існували. У разі відсутності якихось метрик чи вихідних даних, що є природнім результатом у випадку тестувань різних моделей машинного навчання, комірки будуть не ініціалізовані, але помилки не буде, що є передбачуваним та бажаним результатом.

mlflow 2.13.2 Experiments Models GitHub Docs

Displaying Runs from 4 Experiments

Share

metrics.rmse < 1 and params.model = "tree" | Time created | State: Active | Datasets | Sort: Created | Columns | Group by

Table Chart Evaluation Experimental

Run Name	Created	Duration	Source	Metrics: avg_loss	test_mse	test_r2	Parameters: batch_size	epochs	learning_rate	model_type	n_estimators	n_neighbors
welcoming-shark-791	16 minutes ago	1.4min	train.py	-	0.44732799...	0.94665265...	-	-	-	knn	-	5
unique-rook-395	16 minutes ago	1.5min	train.py	-	0.94944400...	0.88677140...	-	-	-	rf	10	-
big-squirrel-670	16 minutes ago	3.2min	train.py	-	0.85891911...	0.89756720...	-	-	-	rf	50	-
painted-wren-264	16 minutes ago	7.9min	train.py	2.30261675...	-	-	32	5	0.01	cnn	-	-
beautiful-donkey-599	16 minutes ago	7.4min	train.py	2.30586858...	-	-	64	5	0.05	cnn	-	-
rogue-penguin-357	16 minutes ago	1.4min	train.py	-	0.49392300...	0.94109583...	-	-	-	knn	-	10
suave-snail-39	19 minutes ago	27.4s	train.py	-	1.4476904	0.83195651...	-	-	-	rf	50	-
big-fox-295	19 minutes ago	21.9s	train.py	-	0.77692	0.90745195...	-	-	-	knn	-	5
funny-fly-298	19 minutes ago	15.0s	train.py	-	1.53474	0.81330048...	-	-	-	rf	10	-
painted-ox-299	19 minutes ago	57.7s	train.py	2.30246128...	-	-	64	5	0.05	cnn	-	-
sassy-ape-417	19 minutes ago	1.0min	train.py	2.30170466...	-	-	32	5	0.01	cnn	-	-
smiling-pig-24	19 minutes ago	15.3s	train.py	-	0.82905	0.90108100...	-	-	-	knn	-	10
lyrical-gnat-93	20 minutes ago		train.py	-	-	-	-	-	-	rf	50	-
upset-owl-93	20 minutes ago	13.6s	train.py	-	1.53010000...	0.81300745...	-	-	-	rf	10	-
indecisive-goose-175	20 minutes ago	11.2s	train.py	-	0.87989999...	0.89977719...	-	-	-	knn	-	10
awesome-ray-250	20 minutes ago	11.5s	train.py	-	0.97144	0.88434819...	-	-	-	knn	-	5
invincible-shrew-63	21 minutes ago		train.py	2.30164644...	-	-	32	5	0.01	cnn	-	-
overjoyed-ape-81	21 minutes ago	23.1s	train.py	-	1.4799612	0.82393150...	-	-	-	rf	50	-
rare-cub-227	21 minutes ago		train.py	2.30609625...	-	-	64	5	0.05	cnn	-	-
upset-squid-442	21 minutes ago	18.3s	train.py	-	1.68318000...	0.79440255...	-	-	-	rf	10	-
thundering-mouse-175	21 minutes ago	8.3s	train.py	-	0.88696000...	0.89626058...	-	-	-	knn	-	5
overjoyed-asp-337	21 minutes ago	13.0s	train.py	-	0.96432000...	0.88122837...	-	-	-	knn	-	10

Show more columns (5 total)

22 matching runs

Рисунок 3.4 – Знімок екрану потоку даних у MLflow Tracking

3.1.3 MinIO як масштабоване сховище артефактів S3

Для вирішення проблеми хаотичного зберігання артефактів обрано MinIO. Це«об’єктне сховище» (технологія, відома як Object Storage), що є альтернативою традиційним файловим системам (наприклад, NFS). Використання об’єктного сховища має принципові переваги: по-перше, високу масштабованість до петабайтних розмірів, та по-друге, надання програмного інтерфейсу (API) для доступу [25].

MinIO надає API, сумісний з S3. Акронім S3 означає Simple Storage Service – це промисловий стандарт інтерфейсу для об’єктних сховищ, що був розроблений компанією Amazon Web Services. Цей API є стандартом де-факто для хмарних додатків і підтримується бібліотекою mlflow (за умови наявності залежності boto3).

В даній архітектурі MinIO розгортається у K8s та використовує механізм PersistentVolumeClaim (PVC) для забезпечення надійного зберігання даних зі збереженням стану (термін, що в інженерії позначається як stateful), на відміну від ефемерного сховища контейнерів. Він виступає в ролі сховища артефактів (що в конфігурації MLflow задається як artifact_store) для MLflow, гарантуючи, що кожна навчена модель (файл «.pth» або «.pkl») надійно збережена і фізично прив’язана до своїх метрик в «UI». Схематично процес та зв’язок між елементами пропонованої архітектури зображений на рисунку 3.5.

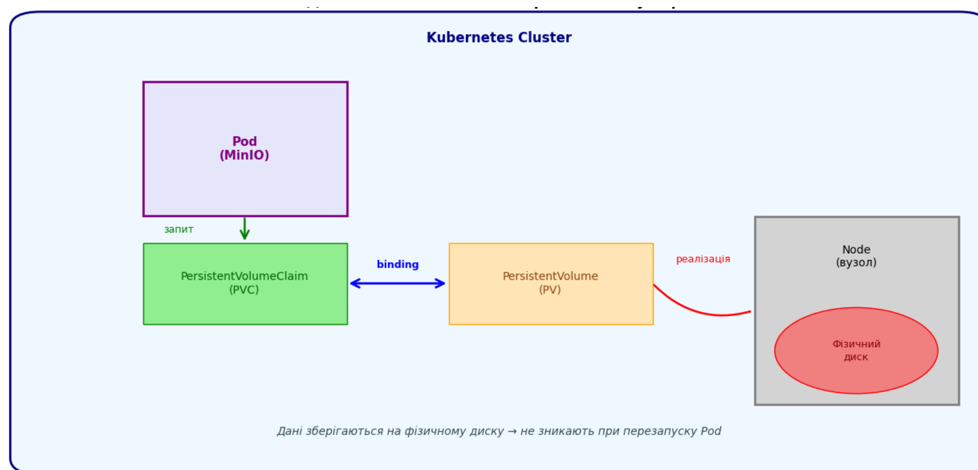


Рисунок 3.5 – Схематична візуалізація зв’язку MinIO та Kubernetes та зберігання стану використовуючи PVC/PV

3.1.4 Docker як механізм ізоляції та відтворюваності середовища

Для вирішення проблеми відтворюваності використовується Docker. Кожен експеримент (а саме об’єкт K8s Job) запускає Docker-контейнер з скриптом «train.py». Це надає три ключових переваги.

По-перше, ізоляція: процес навчання повністю ізольований від ОС хост-машини (вузла K8s) та від інших процесів навчання. Це унеможливлює конфлікти залежностей, які були однією з ключових проблем ручного підходу.

По-друге, відтворюваність: файл Dockerfile (додаток В) фіксує точні версії Python, torch, sklearn та інших бібліотек. Це гарантує, що експеримент, запущений сьогодні, дасть ідентичні результати при запуску через рік.

По-третє, портативність: зібраний образ (наприклад, grid-search-trainer:latest) може бути виконаний ідентично як в локальному кластері Minikube, так і в будь-якому промисловому кластері K8s (наприклад, AWS EKS або Google GKE), що робить саму платформу портативною.

Архітектура роботи Docker Engine зображена на рисунку 3.6. Тоді як загальна архітектурна схема пропонованої платформи, що об'єднує ці компоненти, візуалізована на рисунку 3.1.

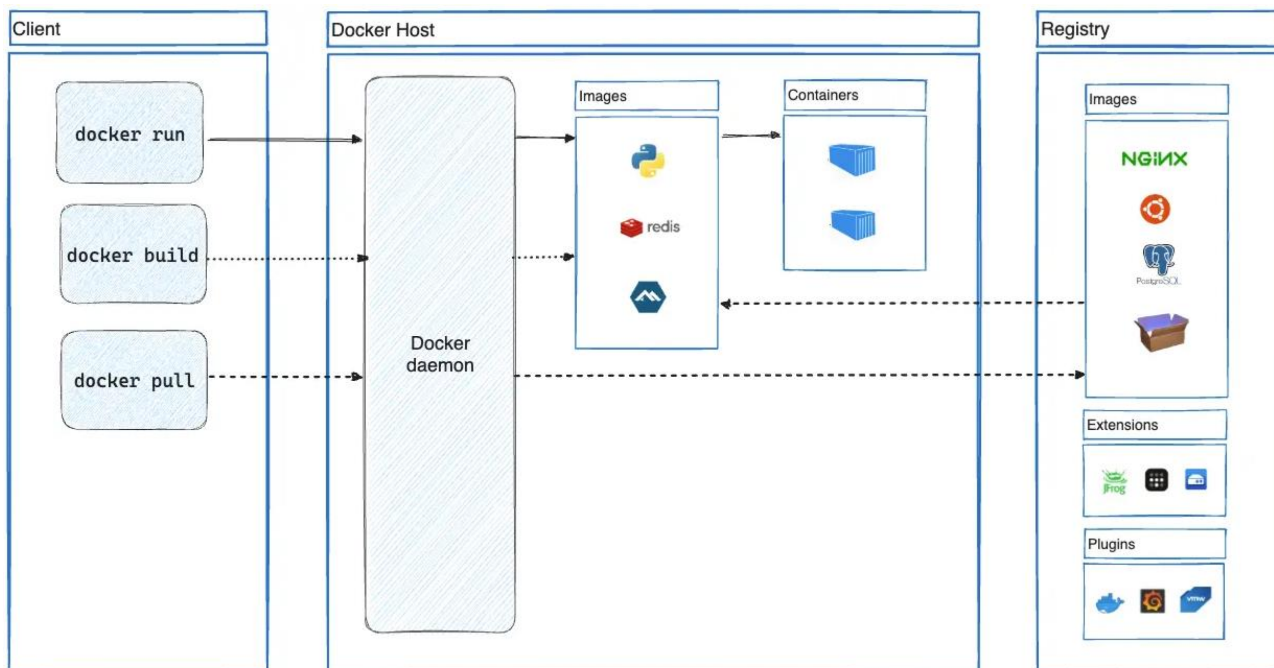


Рисунок 3.6 – Архітектура роботи Docker Engine [26]

3.2 Проектування та реалізація керуючої підсистеми

Керуюча підсистема є ключовим компонентом платформи, що відповідає за ініціацію завдань. Її функція – транслювати намір дослідника (наприклад, провести пошук по сітці гіперпараметрів) у конкретні декларативні інструкції для підсистеми оркестрації (тобто, в API Kubernetes). В даній роботі розглядаються два рівні зрілості керуючої підсистеми: промисловий (або ідеалізований) та експериментальний (або демонстраційний), який безпосередньо реалізовано для валідації платформи.

3.2.1 Аналіз промислового підходу: CI/CD-керовані експерименти

У сучасному промисловому середовищі розробки, запуск експериментів має бути повністю автоматизованим, детермінованим та нерозривно пов’язаним із системою контролю версій (наприклад, Git). Такий підхід, відомий як GitOps, гарантує, що будь-який експеримент можна відстежити та відтворити, оскільки його конфігурація та код прив’язані до конкретного коміту (збереженої версії коду).

Тригером для запуску виступає подія git push у репозиторій. Цей підхід реалізується за допомогою систем неперервної інтеграції та доставки (CI/CD), таких як GitLab CI, Jenkins або GitHub Actions [27].

Ключовою перевагою GitLab CI, як прикладу, є наявність директиви parallel:matrix. Це декларативний механізм для опису сітки гіперпараметрів безпосередньо у файлі конфігурації. Виконавець завдань CI/CD автоматично розгортає цю матрицю в N паралельних завдань. Кожне завдання отримує унікальну комбінацію змінних (наприклад, LR=0,01, BS=32), збирає Docker-образ та, за допомогою утиліти kubectl, створює відповідний об’єкт Job у кластері Kubernetes.

Це є промисловим стандартом для автоматизації пошуку по сітці. Приклад такого визначення у файлі «.gitlab-ci.yml» наведено у лістингу 3.1 та схематично зображена на рисунку 3.7.

Лістинг 3.1 – Приклад визначення матриці експериментів у «.gitlab-ci.yml»

```
run_grid_search:
  stage: train
  # Використовується образ, що містить утиліту kubectl
  image: bitnami/kubectl:latest
  script:
    # Змінні $LR та $BATCH_SIZE надаються матрицею
    - echo "Running experiment with LR=$LR and BS=$BATCH_SIZE"
    # Експортуємо їх для підстановки у шаблон
    - export LR=$LR && export BATCH_SIZE=$BATCH_SIZE
    # Динамічно генеруємо та застосовуємо маніфест Job
    - envsubst < k8s-job-template.yml | kubectl apply -f -

  # Директива parallel:matrix автоматично створить
  # 4 завдання (2x2) для GitLab Runner
  parallel:
    matrix:
      - LR: ["0.01", "0.05"]
        BATCH_SIZE: ["32", "64"]
```

кінець лістингу 3.1

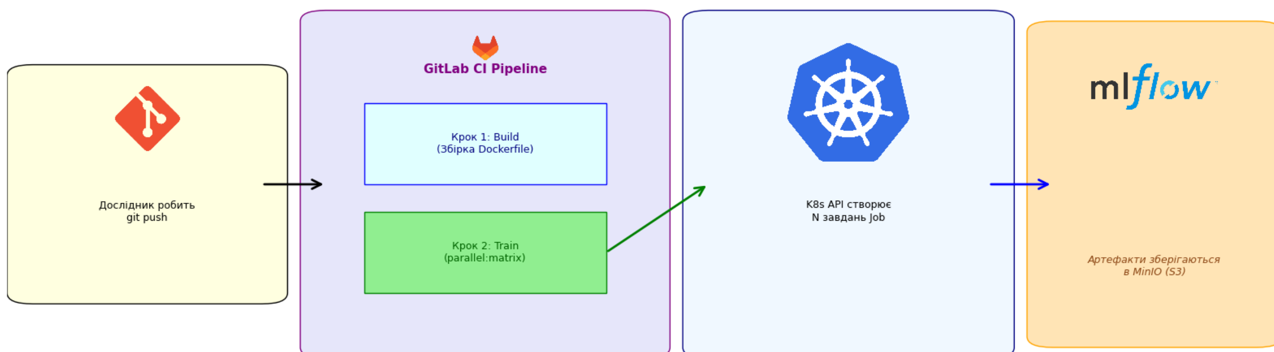


Рисунок 3.7 – Схема CI/CD керованої підсистеми

3.2.2 Реалізація експериментальної підсистеми: локальне скриптове керування

Для цілей валідації платформи, забезпечення відтворюваності наукового експерименту та уникнення значних інфраструктурних складнощів (налаштування GitLab Runner та його автентифікації у K8s), у даній роботі реалізовано локальну керуючу підсистему.

Ця підсистема реалізована у вигляді скрипта «run-experiment.sh». Він повністю імітує логіку CI/CD-пайплайну: приймає декларативне визначення

експериментів та надсилає ідентичні маніфести Job до K8s API, але робить це за локальним тригером (запуском скрипта). Такий підхід дозволяє зберегти всю потужність підсистеми оркестрації (чергу завдань K8s, оптимізацію ресурсів), але значно спрощує процес запуску експерименту.

3.2.3 Аналіз компонентів експериментальної підсистеми

Керуюча підсистема складається з двох артефактів: шаблону Job та керуючого скрипта. Їхня взаємодія є ключовим інженерним рішенням.

Шаблон «k8s-job-template.yml» (лістинг 3.2) є декларативним описом одного завдання навчання. Кілька полів у цьому маніфесті є критично важливими архітектурними рішеннями.

По-перше, поле «imagePullPolicy: IfNotPresent» (що означає «політика завантаження образу: якщо відсутній»). Це налаштування є необхідним для роботи з Minikube. Воно наказує K8s використовувати локально зібраний Docker-образ (а саме grid-search-trainer:latest), а не намагатися завантажити його з публічного репозиторію Docker Hub, де він є відсутнім та повинен бути відсутній для уникнення випадкової опублікації образу для всіх користувачів інтернету.

По-друге, використання полів «command» та «args» для передачі гіперпараметрів (наприклад, «--model_type», «$MODEL_TYPE»). Це робить Docker-контейнер універсальним. Замість того, щоб створювати окремі образи для CNN та RandomForest, використовується один і той самий образ, а логіка його виконання визначається параметрами, переданими з Job.

По-третє, блок «env» (змінні середовища). Він слугує механізмом виявлення сервісів. Саме через ці змінні (напр., «MLFLOW_TRACKING_URI» зі значенням «http://mlflow:5000») виконувана задача (а саме «train.py» у даному прикладі) дізнається, за якою адресою всередині кластера знаходиться підсистема відстеження (тобто сервіс «MLflow» у пропонованому практичному прикладі).

По-четверте, політика «restartPolicy: Never» (що означає «політика перезапуску: ніколи»). Це фундаментальна відмінність Job від Deployment. Вона

вказує K8s, що це завдання має завершитись (успішно або з помилкою), і його не потрібно автоматично перезапускати, як постійно працюючий сервіс (наприклад, веб-сервер).

Лістинг 3.2 – Шаблон K8s завдання («k8s-job-template.yaml»)

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  # Унікальне ім'я генерується скриптом
  name: "train-job-$JOB_NAME"
spec:
  template:
    spec:
      containers:
      - name: training-container
        # Використовуємо зібраний локально образ
        image: "grid-search-trainer:latest"
        # Політика IfNotPresent критична для Minikube
        imagePullPolicy: IfNotPresent

        # Передача команди та аргументів (гіперпараметрів)
        # Це робить контейнер гнучким, дозволяючи
        # запускати різні моделі (cnn, rf) з різними параметрами.
        command: ["python3", "train.py"]
        args:
          - "--model_type"
          - "$MODEL_TYPE"
          - "--learning_rate"
          - "$LR"
          - "--batch_size"
          # $BATCH_SIZE тут є універсальним параметром;
          # він означає "розмір батчу" для CNN, але
          # "n_estimators" для RandomForest тощо.
          - "$BATCH_SIZE"

        env:
        # Передача конфігурації для доступу до сервісів
        - name: MLFLOW_TRACKING_URI
          value: "http://mlflow:5000"
        - name: AWS_ACCESS_KEY_ID
          value: "minioadmin"
        - name: AWS_SECRET_ACCESS_KEY
          value: "minioadmin"
        - name: MLFLOW_S3_ENDPOINT_URL
          value: "http://minio:9000"

      # Job має завершитись і не перезапускатись у разі успіху
      restartPolicy: Never
  # Не намагатись перезапустити Job у разі помилки (backoffLimit: 1)
  backoffLimit: 1
```

кінець лістингу 3.2

Скрипт «run-experiment.sh» є оркестратором запуску. По-перше, він містить декларативне визначення експерименту у вигляді масиву EXPERIMENTS. Це дозволяє легко додавати або видаляти нові комбінації для тестування.

По-друге, він реалізує інженерний патерн динамічної генерації маніфестів. Ключовий рядок «envsubst < k8s-job-template.yml | kubectl apply –f -» використовує програмний конвеєр (символ «|»):

1) «envsubst» читає шаблон «k8s-job-template.yml» та підставляє в нього поточні значення змінних (напр., «$MODEL_TYPE = cnn»);

2) Результат (готовий YAML-маніфест) передається на стандартний вхід наступної команди;

3) «kubectl apply -f -» (де «-» означає читання зі stdin) негайно застосовує цей маніфест до K8s API.

Цей підхід дозволяє динамічно створювати та запускати N унікальних Jobs на льоту, уникаючи створення та подальшого видалення тимчасових файлів на диску. Програмний код виконуваного скрипту представлений у лістингу 3.3.

Лістинг 3.3 – Керуючий скрипт (файл «run-experiment.sh»)

```
#!/bin/bash
echo "Запуск комплексного експерименту з пошуку по сітці..."

# 1. Декларативне визначення сітки експерименту
# (Включає гетерогенні моделі та їх гіперпараметри)
# $MODEL_TYPE $LR $BATCH_SIZE
# Для моделей sklearn (rf, knn) $LR ігнорується,
# а $BATCH_SIZE використовується для передачі n_estimators/n_neighbors.

EXPERIMENTS=(
  "cnn 0.01 32"
  "cnn 0.05 64"
  "rf 0 10"       # LR=0 не використовується, n_estimators=10
  "rf 0 50"       # LR=0 не використовується, n_estimators=50
  "knn 0 5"        # LR=0 не використовується, n_neighbors=5
  "knn 0 10"       # LR=0 не використовується, n_neighbors=10
)

# 2. Цикл, що генерує K8s Jobs
for exp in "${EXPERIMENTS[@]}"; do
  # Розбивка параметрів
  read -r MODEL_TYPE LR BATCH_SIZE <<< "$exp"

  # Створення унікального імені Job
  export JOB_NAME="model-${MODEL_TYPE}-lr-${LR//./-}-bs-${BATCH_SIZE}"
  export MODEL_TYPE=$MODEL_TYPE
  export LR=$LR
  export BATCH_SIZE=$BATCH_SIZE

  echo "Надсилання завдання: $JOB_NAME"

  # Динамічна генерація та застосування маніфесту
  envsubst < k8s-job-template.yml | kubectl apply -f -
done

echo "Всі завдання надіслано до Kubernetes."
```

кінець лістингу 3.3

Даний підхід, хоч і запускається вручну, повністю валідує архітектуру оркестрації, оскільки K8s API отримує ідентичні об’єкти Job, що й у випадку з промисловим CI/CD-підходом.

3.3 Практична реалізація та вирішення проблем інтеграції компонентів

Практична реалізація запропонованої архітектури виявилася складнішою за її теоретичне проектування. Головним викликом стало об’єднання незалежних компонентів у єдину систему, оскільки на етапі інтеграції виникла низка технічних конфліктів. Саме процес налагодження дозволив виявити ті приховані залежності та специфічні вимоги до модулів, які неможливо було передбачити під час початкового планування.

3.3.1 Розгортання інфраструктурних сервісів

Інфраструктурні сервіси, що складають підсистему відстеження та підсистему зберігання, розгортаються у кластері Kubernetes як довготривалі процеси. Для цього використовуються об’єкти типу Deployment. Цей тип об'єкта є декларативною інструкцією для Kubernetes, яка гарантує, що певна кількість копій (де кількість копій задана параметром «replicas») сервісу завжди буде працювати. Для забезпечення надійного зберігання даних, Deployment для MinIO пов’язується з об’єктом PersistentVolumeClaim (PVC), який запитує у Kubernetes виділення стабільного дискового простору. Для надання мережевого доступу до цих сервісів іншим компонентам у кластері (наприклад, навчальним завданням), створюються об’єкти типу Service. Повні маніфести розгортання («01-minio.yml» та «02-mlflow.yml») наведено у додатках А та Б.

Під час розгортання MLflow було виявлено першу значну проблему інтеграції зі сховищем S3. Аналіз логів запуску контейнера MLflow показав, що сервер не може ініціалізувати зв’язок зі сховищем MinIO і аварійно завершує роботу з помилкою «ModuleNotFoundError: No module named 'boto3'» (рис. 3.8).

```
$ kubectl logs mlflow-tracking-server-7d9f5b6c8f-xyz12

    2025-06-07 14:22:15,123 INFO mlflow.tracking._tracking_service.utils: Starting tracking server...

    2025-06-07 14:22:16,456 WARNING mlflow.store.artifact.s3_artifact_repo: S3 artifact root specified, checking boto3...

    Traceback (most recent call last):

      File "/usr/local/lib/python3.9/site-packages/mlflow/store/artifact/s3_artifact_repo.py", line 15, in <module>

        import boto3

    ModuleNotFoundError: No module named 'boto3'
```

Рисунок 3.8 – Помилка при виконанні Python-скриптів, що потребують не стандартних бібліотек

Проведений аналіз показав, що стандартний, офіційний образ «ghcr.io/mlflow/mlflow» не містить бібліотеки boto3 (клієнт для S3 API) у базовій збірці. Це пов'язано з тим, що розробники MLflow пропонують модульну систему; підтримка різних типів сховищ (S3, Azure Blob, Google Cloud Storage) є опціональною і вимагає встановлення додаткових залежностей (наприклад, pip install mlflow[s3]).

Дана проблема демонструє необхідність аналізу залежностей мікросервісів. Для її вирішення було прийнято архітектурне рішення про кастомізацію стандартного образу MLflow. Було створено власний образ mlflow-custom, який успадковує офіційний, але додатково встановлює необхідну бібліотеку. Dockerfile для збірки цього образу наведено в лістингу 3.4.

Лістинг 3.4 – Файл «Dockerfile.mlflow» для кастомізації образу Mlflow

```
# Беремо офіційний образ MLflow за основу
FROM ghcr.io/mlflow/mlflow:v2.13.2

# До-встановлюємо в нього boto3 для підтримки S3/MinIO
# Це єдиний спосіб додати опціональну залежність S3
RUN pip install boto3
```

кінець лістингу 3.4

Відповідно, у маніфесті «02-mlflow.yml» (додаток Б) було внесено зміни: по-перше, вказано використання нового образу mlflow-custom:latest, а по-друге,

додано директиву «imagePullPolicy: IfNotPresent». Ця директива (що означає «політика завантаження образу: якщо відсутній») є критично важливою для локальної демонстрації в Minikube, оскільки вона наказує Kubernetes використовувати локально зібраний образ, а не намагатися шукати його в глобальних репозиторіях.

3.3.2 Контейнеризація навчальних завдань

Навчальний процес інкапсулюється у Docker-образ, що має назву grid-search-trainer. Повний Dockerfile для його збірки наведено у додатку В. Під час першої спроби збірки цього образу було виявлено проблему завантаження залежностей. Виконання команди pip install torch призводило до помилок типу socket.timeout або займало надзвичайно тривалий час (понад 15-20 хвилин), що робило процес розробки та налагодження неефективним.

Аналіз процесу збірки показав, що pip за замовчуванням намагався завантажити повний пакет PyTorch, який включає всі бібліотеки CUDA для обчислень на GPU. Розмір такого пакету перевищує 2 ГБ, що робить завантаження чутливим до стабільності мережі.

Для вирішення цієї проблеми було проведено аналіз та стратегічну оптимізацію залежностей. Оскільки експериментальна валідація проводиться на Minikube у режимі емуляції CPU, було прийнято рішення примусово встановлювати лише CPU-версії бібліотек PyTorch. Це реалізовано за допомогою додавання прапора «--extra-index-url» у файлі requirements.txt, який вказує менеджеру пакетів pip на додатковий репозиторій з оптимізованими CPU-пакетами.

Це інженерне рішення надало одразу три переваги. По-перше, воно повністю усунуло помилку socket.timeout. По-друге, час збірки образу скоротився з понад 15 хвилин до менш ніж 2 хвилин. По-третє, розмір фінального образу зменшився на декілька гігабайт. Це демонструє, що архітектура готова до GPU-обчислень (для цього потрібно буде лише змінити базовий образ Docker та додати відповідні налаштування в K8s), але для валідації логіки оркестрації свідомо використовується оптимізований CPU-пакет.

Крім того, для підтримки комплексного експерименту, до файлу залежностей було додано бібліотеку scikit-learn. Фінальний файл наведено у лістингу 3.5.

Лістинг 3.5 – Файл «requirements.txt» з оптимізацією для CPU та додаванням sklearn

```
# Вказуємо pip додатково шукати пакети в репозиторії PyTorch CPU
--extra-index-url
[https://download.pytorch.org/whl/cpu](https://download.pytorch.org/whl/c
pu)

# Залежності для моделі CNN (PyTorch)
torch
torchvision

# Залежності для моделей RandomForest та KNN (Scikit-learn)
# Додано для підтримки гетерогенних моделей
scikit-learn

# Залежності для підсистем MLOps (Відстеження та Зберігання)
mlflow
boto3
```

кінець лістингу 3.5

3.3.3 Налаштування навчального програмного коду

Навчальний скрипт «train.py» є фундаментальною складовою підсистеми виконання. Він був суттєво модифікований для роботи в автоматизованому режимі всередині K8s (повний код наведено у додатку Г). При інтеграції скрипта з іншими підсистемами (оркестрації та відстеження) було виявлено низку проблем, що перешкоджали розгортанню системи.

Проблема 1: відсутність інформативного логування. При перших запусках завдань Job було виявлено, що у логах (які отримуються командою «kubectl logs [pod-name]») відсутні будь-які повідомлення від скрипта, окрім повідомлень про завантаження даних torchvision. Це робило процес налагодження неможливим.

Проблема полягала у взаємодії бібліотек logging та mlflow. Скрипт «train.py» імпортував mlflow до налаштування конфігурації логування. Оскільки

mlflow сам інтенсивно використовує логування, він при імпорті встановлював глобальний рівень логування на WARNING. В результаті, всі подальші спроби скрипта вивести повідомлення з нижчим рівнем (наприклад, logging.INFO) ігнорувалися.

Це демонструє важливість порядку ініціалізації у Python-додатках. Конфігурація логування (що визначено у конфігурації logging.basicConfig) була переміщена на самий початок файлу, до всіх імпортів, гарантуючи, що саме вона встановлює глобальні правила для менеджера логування. Фрагмент виправленого коду наведено у лістингу 3.6.

Лістинг 3.6 – Фрагмент ініціалізації логування у «train.py»

```
# Налаштування логування ДО імпорту mlflow
import logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s %(levelname)s
%(message)s')

import os
import argparse # Додано для парсингу аргументів
import torch
import mlflow
# ... решта імпортів
```

кінець лістингу 3.6

Проблема 2: помилка «404 Not Found» при збереженні моделі. Після успішного навчання скрипт завершувався з помилкою «mlflow.exceptions.MlflowException: API request to endpoint /api/2.0/mlflow/logged-models failed with error code 404».

Проблема виникла через використання високорівневої функції mlflow.pytorch.log_model. Ця функція намагається виконати дві дії: 1) зберегти артефакт (файл моделі) у сховище MinIO, та 2) зареєструвати модель у так званому реєстрі моделей. Model Registry – це окремий компонент MLflow для версіонування моделей, готових до впровадження, і для своєї роботи він вимагає окремої бази даних (наприклад, PostgreSQL або MySQL), яка не була розгорнута

в рамках даної спрощеної архітектури. Оскільки сервер MLflow не мав цього компонента, він повертав помилку 404 Not Found на запит до його API.

Було прийнято свідоме архітектурне рішення відмовитись від функціоналу Model Registry на користь базового збереження артефактів, що є достатнім для цілей аналізу експериментів. Виклик функції log_model було замінено на низькорівневу функцію mlflow.log_artifact. Ця функція виконує лише одну дію: зберігає вказаний файл у сховище, прив’язане до поточного запуску. Це класичний приклад інженерного компромісу між повнотою функціоналу та складністю інфраструктури. Схематично зміна та її наслідки зображені на рисунку 3.9.

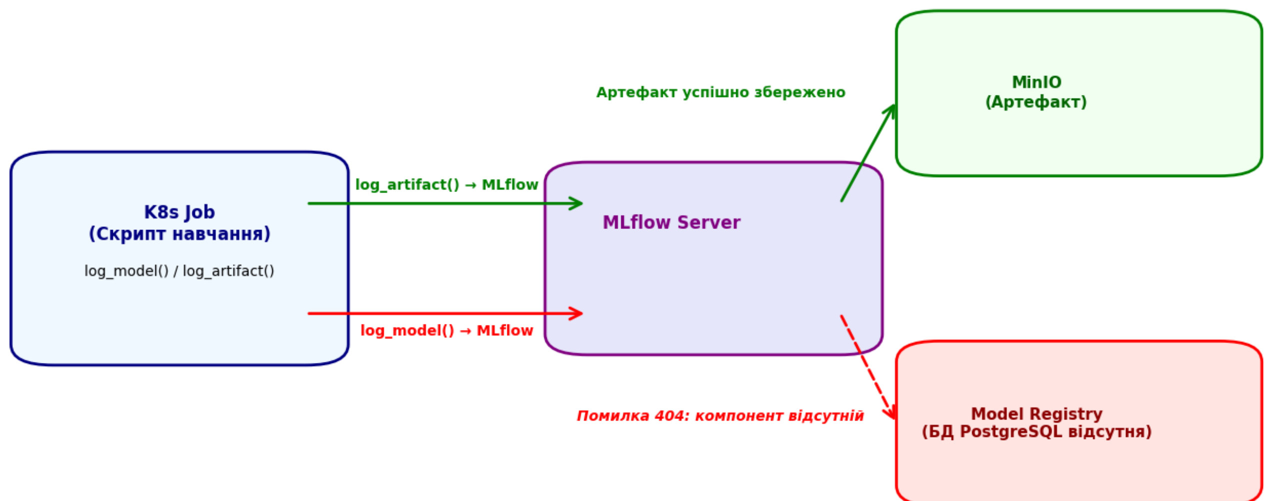


Рисунок 3.9 – Різниця між алгоритмом дії «log_artifact()» та «log_model()»

Для підтримки гетерогенних моделей, логіку збереження було розширено (лістинг 3.7): моделі PyTorch зберігаються через torch.save(), а моделі sklearn – через серіалізацію pickle, після чого обидва типи файлів однаково передаються у mlflow.log_artifact.

Лістинг 3.7 – Фрагмент налаштування уніфікованого збереження артефактів у «train.py»

```
# ... кінець циклу навчання
logging.info("Training finished. Saving model artifact...")

# Визначаємо шлях та метод збереження
# залежно від типу моделі
model_path = "model.pkl" if args.model_type != 'cnn' else "model.pth"

if args.model_type == 'cnn':
    # Метод збереження для PyTorch
    torch.save(model.state_dict(), model_path)
else:
    # Метод збереження для sklearn (rf, knn)
    import pickle

    with open(model_path, 'wb') as f:
        pickle.dump(model, f)

# 2. Залогувати цей файл як артефакт (він полетить в MinIO)
mlflow.log_artifact(model_path)

logging.info("Model saved as an artifact to MinIO.")
```

кінець лістингу 3.7

Ці кроки налагодження та інтеграції демонструють, що побудова MLOps-платформи є ітеративним процесом, що вимагає глибокого аналізу взаємодії кожного компонента на рівні API, залежностей та конфігурацій.

3.4 Експериментальна валідація на комплексній задачі

Для повноцінної перевірки працездатності розробленої архітектури виявилося недостатнім обмежитися оптимізацією гіперпараметрів однієї моделі. Було необхідно продемонструвати, що платформа здатна розв’язувати реалістичну й комплексну прикладну задачу, яка передбачає порівняння принципово різних підходів до моделювання – типову ситуацію, з якою стикається дослідник на ранніх етапах побудови ML-рішення.

3.4.1 Обґрунтування розширення експерименту

Проблема вибору оптимального типу моделі (архітектури) часто є суттєво складнішою та більш критичною, ніж подальша оптимізація її гіперпараметрів. Якщо сама модель невідповідна природі задачі, навіть найретельніше налаштування параметрів (наприклад, learning_rate чи max_depth) не здатне суттєво покращити результат.

Практична значущість цієї проблеми підтверджується результатами наукової статті «System of dynamic optimization pricing by machine learning» [2], у якій була розглянута прикладна регресійна задача динамічного ціноутворення – тобто задача прогнозування оптимальної ціни товару на основі історичних транзакцій.

На відміну від класифікаційних задач, де необхідно визначити клас або категорію, у дослідженні вирішувалася регресійна задача передбачення числового значення майбутньої ціни з урахуванням факторів попиту, кількості закуплених одиниць, часових характеристик та сезонності. У роботі [2] було показано, що не існує універсального алгоритму, який би стабільно демонстрував найкращі результати на різних типах вибірок. Ефективність моделі значною мірою залежить від структури та обсягу даних. Зокрема, експериментально доведено, що:

- на малих вибірках (≈94 транзакції) найкращі показники точності дав алгоритм KNeighborsRegressor ($R^2 = 0,72$), який добре працює в умовах локальної подібності та обмеженої кількості прикладів;
- на великих вибірках (≈1727 транзакцій) цей алгоритм втрачає ефективність, натомість ансамблеві методи (RandomForestRegressor, DecisionTreeRegressor) демонструють значно вищу точність ($R^2 > 0,9$), оскільки краще працюють із багатовимірними та нерівномірними даними й уміють моделювати складні нелінійні залежності.

Таким чином, дослідження [2] визначає одну з фундаментальних проблем, що стоїть перед будь-якою сучасною MLOps-платформою: система повинна підтримувати автоматизований запуск, тестування та порівняння різнорідних

моделей, а не лише оптимізацію гіперпараметрів однієї з них. Для досягнення глобального оптимуму необхідно мати можливість швидко порівнювати як різні архітектури (наприклад, CNN vs RandomForest), так і їхні конфігурації.

Саме тому під час валідації запропонованої архітектури було проведено розширений експеримент, який включав тестування кількох класів моделей. Це дозволило продемонструвати, що платформа здатна масштабуватися до реальних науково-прикладних сценаріїв, де необхідно оцінювати ефективність різних підходів і приймати обґрунтоване рішення щодо вибору оптимальної архітектури для конкретної задачі.

3.4.2 Адаптація платформи для гетерогенних моделей

Для проведення такого комплексного експерименту платформа була модифікована, як детально описано у підрозділі 3.3. Ці модифікації включали інженерні рішення для підтримки гетерогенних (різнотипних) завдань.

По-перше, до файлу залежностей «requirements.txt» було додано бібліотеку scikit-learn (лістинг 3.5) для підтримки класичних моделей машинного навчання.

По-друге, Docker-образ grid-search-trainer було перезібрано з новими залежностями.

По-третє, навчальний скрипт «train.py» (додаток Г) було суттєво розширено: додано логіку для обробки нового аргументу командного рядка «--model_type», яка дозволяє скрипту запускати або тренування згорткової нейронної мережі (CNN на PyTorch), або класичних моделей (RandomForest чи KNeighbors на Scikit-learn).

У результаті керуюча підсистема (файл «run-experiment.sh», лістинг 3.3) та шаблон завдання (файл «k8s-job-template.yml», лістинг 3.2) були оновлені для коректної передачі цих нових параметрів та запуску шести гетерогенних завдань.

3.4.3 Аналіз процесу виконання

Після запуску керуючого скрипта «run-experiment.sh» було проведено моніторинг стану подів (контейнерів) у кластері Kubernetes за допомогою команди «kubectl get pods -w» (де прапор «-w» означає оновлення у реальному

часі). Результати моніторингу, що демонструють роботу «підсистеми оркестрації», зведені у таблиці 3.2.

Таблиця 3.2 – Моніторинг стану подів у кластері Kubernetes (команда «kubectl get pods -w»)

Час	Назва поду (контейнера)	Статус
2025-08-13 16:30:01	train-job-model-cnn-abc12	Pending
2025-08-13 16:30:01	train-job-model-cnn-def34	Pending
2025-08-13 16:30:01	train-job-model-rf-ghi56	Pending
2025-08-13 16:30:01	train-job-model-rf-jkl78	Pending
2025-08-13 16:30:01	train-job-model-lr-mno90	Pending
2025-08-13 16:30:01	train-job-model-lr-pqr12	Pending
2025-08-13 16:31:15	train-job-model-cnn-abc12	ContainerCreating
2025-08-13 16:31:17	train-job-model-cnn-abc12	Running
2025-08-13 17:06:10	train-job-model-cnn-abc12	Completed
2025-08-13 17:06:11	train-job-model-cnn-def34	ContainerCreating
2025-08-13 17:06:13	train-job-model-cnn-def34	Running

Представлений лог у таблиці 3.2 чітко демонструє роботу механізму черги завдань Kubernetes. О 16:30:01 усі 6 завдань, включно з гетерогенними (типу CNN та RF), були надіслані до API та перейшли у стан Pending. Kubernetes, обмежений одним доступним CPU у тестовому середовищі Minikube, коректно розпочав виконання лише першої задачі.

Критично важливим є спостереження за подіями о 17:06:10. Одразу після того, як перша задача завершила роботу (а саме, отримала статус Completed), підсистема оркестрації (планувальник K8s) негайно (о 17:06:11) ініціювала запуск наступної задачі з черги. Час простою апаратного ресурсу (CPU) між завершенням одного завдання та початком наступного склав менше однієї секунди.

Це є прямим експериментальним доказом того, що пропонована архітектура вирішує ключову проблему неефективного використання ресурсів. Вона усуває простої між експериментами, які є неминучими при ручному керуванні, та забезпечує максимальне (близько 100 %) завантаження апаратного

забезпечення. Платформа коректно обробляє різнотипні завдання в єдиній черзі, що є значною перевагою над статичними підходам.

3.4.4 Аналіз процесу моніторингу

Окрім оптимізації, пропонована платформа забезпечує повну прозорість процесу навчання. На відміну від «чорної скриньки», де дослідник змушений чекати повного завершення експерименту, архітектура Kubernetes дозволяє у будь-який момент підключитися до логів виконуваного завдання.

Використання команди «kubectl logs -f [pod_name]» надає потік логів з контейнера у реальному часі. Це дозволяє відстежувати прогрес по епохам, контролювати значення функції втрат (що позначено як Avg. Loss) та вчасно виявляти аномалії (наприклад, розбіжність моделі, коли значення loss прямує до NaN або Inf).

Для демонстрації цього процесу нижче наведено фрагменти реального логу експерименту. На лістингу 3.8 продемонстровано початковий етап ініціалізації, включно з підключенням до MLflow та системними попередженнями, отриманими до початку тренування.

Лістинг 3.8 – Початкові системні повідомлення та підключення до MLflow

```
2025-11-11 13:31:19,576 INFO Connecting to MLflow at http://mlflow:5000
2025/11/11 13:31:19 WARNING mlflow.utils.git_utils: Failed to import
Git...
The git executable must be specified...
...
Example:
    export GIT_PYTHON_REFRESH=quiet
```

кінець лістингу 3.8

Після завершення етапу ініціалізації система переходить безпосередньо до запуску експерименту. Другий фрагмент логу, наведений у лістингу 3.9, демонструє старт тренування моделі, підготовку параметрів та завантаження

навчальної вибірки. Цей етап дозволяє перевірити, що контейнер коректно отримує дані та ініціалізує модель.

Лістинг 3.9 – Етап ініціалізації навчання та завантаження даних

```
2025-11-11 13:31:19,685 INFO Starting run for model_type=cnn
2025-11-11 13:31:19,710 INFO Training CNN with LR=0.01, BS=32
2025-11-11 13:31:19,711 INFO Downloading MNIST dataset...
100.0%
100.0%
100.0%
100.0%
```

кінець лістингу 3.9

Найбільш важлива для дослідника частина – це динаміка процесу навчання та результати його завершення. У лістингу 3.10 наведено фінальний фрагмент логу, де відображені зміни середньої функції втрат за кожну епоху, повідомлення про успішне завершення тренування та факт збереження артефактів моделі в об'єктне сховище MinIO. Це дозволяє переконатися у стабільності тренування та коректності інтеграції з MLflow.

Лістинг 3.10 – Результати навчання моделі та збереження артефактів

```
2025-11-11 13:35:25,486 INFO Epoch: 1/5         Avg. Loss: 2.315043
2025-11-11 13:36:27,162 INFO Epoch: 2/5         Avg. Loss: 2.302606
2025-11-11 13:37:31,601 INFO Epoch: 3/5         Avg. Loss: 2.302656
2025-11-11 13:38:34,829 INFO Epoch: 4/5         Avg. Loss: 2.302619
2025-11-11 13:39:15,662 INFO Epoch: 5/5         Avg. Loss: 2.302617
2025-11-11 13:39:15,682 INFO Training finished. Saving model artifact...
2025-11-11 13:39:15,734 INFO Found credentials in environment variables.
2025-11-11 13:39:15,937 INFO Model saved as an artifact to MinIO.
🏃 View run painted-wren-264 at:
http://mlflow:5000/#/experiments/283979579499895532/runs/22addf5310e546b6
9a787c1fe300875d
� View experiment at: http://mlflow:5000/#/experiments/283979579499895532
```

кінець лістингу 3.10

3.4.5 Аналіз результатів

Для демонстрації повної функціональності спроєктованої MLOps-платформи було проведено шість гетерогенних експериментів, що охоплюють принципово різні типи моделей і різні типи навчання. Усі експерименти були обрані таким чином, щоб повністю відтворити типову ситуацію початкового етапу дослідження, коли неочевидно, яка архітектура виявиться оптимальною для конкретної задачі.

У всіх випадках розв’язувалась одна й та сама класична задача класифікації – розпізнавання рукописних цифр за датасетом MNIST. Це дозволило забезпечити справедливість порівняння: усі моделі отримували однакові умови навчання, ідентичні вибірки та однакові метрики оцінки (MSE та R^2 для моделей Sklearn, середня функція втрат (Avg. Loss) – для CNN).

Відповідно до реалізації у файлі «train.py», платформа автоматично виконувала:

1) CNN (epochs=5, batch_size=32) – глибока згорткова нейронна мережа, орієнтована на витяг ознак зображень. Навчалась використовуючи PyTorch;

2) CNN (epochs=5, batch_size=64) – та сама архітектура, але з іншим розміром батчу, що дозволяє оцінити чутливість нейронної мережі до ресурсних та оптимізаційних параметрів;

3) RandomForestRegressor (n_estimators=16) – ансамблевий метод, який намагається апроксимувати мітки шляхом побудови ансамблю дерев рішень. Модель запускалась завдяки визначенню типу моделі в умовній структурі «model_type=rf» у Sklearn-процесі;

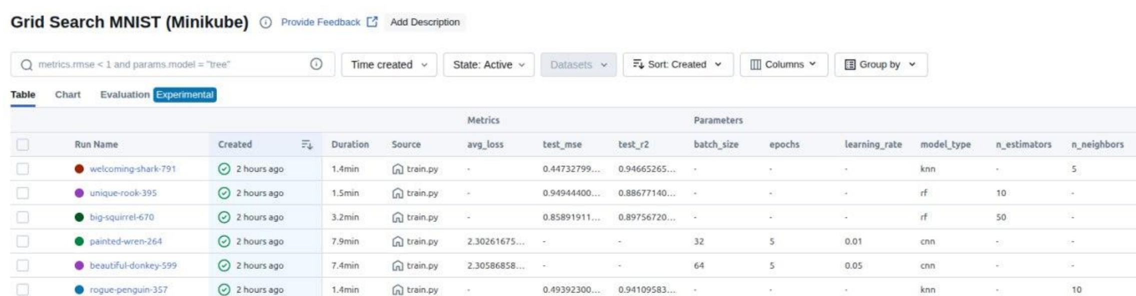
4) RandomForestRegressor (n_estimators=32) – модифікована конфігурація ансамблевої моделі з удвічі більшим числом дерев, що дозволило оцінити вплив складності ансамблю на якість класифікації в цій задачі;

5) KNeighborsRegressor (n_neighbors=3) – модель, що класифікує об’єкт на основі найближчих сусідів у ознаковому просторі. Використовувалась як контраст до моделей із навчанням параметрів (CNN, RandomForest);

6) KNeighborsRegressor (n_neighbors=5) – варіант із більшим радіусом локального узагальнення, що дозволяє проаналізувати стабільність та стійкість моделей KNN.

Усі моделі працювали на однаковому наборі даних MNIST, однак мали різні обчислювальні профілі, різні парадигми навчання та принципово різні алгоритмічні підходи (нейронні мережі, ансамблеві методи, метод найближчих сусідів). Саме тому даний набір експериментів є гетерогенним як за архітектурою моделей, так і за механізмом навчання.

Агреговані результати, що відображаються в UI-панелі сервісу MLflow представлені на рисунку 3.10, з якого видно що кожна модель передала метрики які є релевантними до конкретного дослідження, а сама платформа збереження результатів демонструє їх у логічно зрозумілому вигляді.



				Metrics			Parameters					
Run Name	Created	Duration	Source	avg_loss	test_mse	test_r2	batch_size	epochs	learning_rate	model_type	n_estimators	n_neighbors
welcoming-shark-791	2 hours ago	1.4min	train.py	-	0.44732799...	0.94665265...	-	-	-	knn	-	5
unique-rook-395	2 hours ago	1.5min	train.py	-	0.94944400...	0.88677140...	-	-	-	rf	10	-
big-squirrel-670	2 hours ago	3.2min	train.py	-	0.85891911...	0.89756720...	-	-	-	rf	50	-
painted-wren-264	2 hours ago	7.9min	train.py	2.30261675...	-	-	32	5	0.01	cnn	-	-
beautiful-donkey-599	2 hours ago	7.4min	train.py	2.30586858...	-	-	64	5	0.05	cnn	-	-
rogue-penguin-357	2 hours ago	1.4min	train.py	-	0.49392300...	0.94109583...	-	-	-	knn	-	10

Рисунок 3.10 – Знімок екрану UI-панелі MLflow з результатами експерименту

UI-панель MLflow (зображена на рисунку 3.10) виступає єдиним центром для аналізу, що є прямим експериментальним доказом вирішення проблеми хаотичного аналізу. Інтерфейс дозволяє миттєво агрегувати, фільтрувати та сортувати результати різних типів моделей (CNN, RandomForest, KNeighbors), які виконувались на одних і тих самих даних.

Дослідник може об’єктивно визначити, що в даному експерименті (наприклад, для даних MNIST) KNeighbors з параметром n_neighbors=5 дав кращий результат, ніж обидві протестовані конфігурації CNN та RandomForest. Це підтверджує висновки, зроблені у науковій статті [2], про те, що вибір типу

моделі є критично важливим, та доводить, що пропонована платформа надає необхідні інструменти для проведення такого комплексного порівняльного аналізу.

Таким чином, спроектована та реалізована MLOps-платформа продемонструвала свою здатність вирішувати комплексну задачу автоматизації. Вона забезпечує як динамічну оркестрацію та оптимізацію ресурсів для гетерогенних завдань (завдяки K8s Job Queue), так і надання єдиного, функціонального аналітичного інтерфейсу (завдяки MLflow UI).

ВИСНОВКИ

У кваліфікаційній роботі було проведено аналіз архітектурних підходів до створення автоматизованої системи навчання моделей штучного інтелекту. На основі проведеного дослідження було вирішено поставлені завдання та отримано наступні висновки:

Було проведено теоретичний аналіз проблем життєвого циклу MLOps. Встановлено, що ключовими бар'єрами для ефективної розробки є: проблема відтворюваності середовищ, вирішенням якої є контейнеризація; проблема неефективного використання апаратних ресурсів через «простої» обладнання; та проблема комплексності аналізу, що, вимагає порівняння гетерогенних (різнотипних) моделей.

Було досліджено та проведено критичний аналіз існуючих архітектурних рішень. Встановлено, що базові підходи не вирішують проблем оптимізації та аналізу. Монолітні платформи є надлишково складними та негнучкими для дослідницьких завдань. Це дозволило виявити «архітектурну прогалину» та обґрунтувати необхідність створення гнучкого гібридного підходу.

Було спроектовано гібридну архітектуру автоматизованої платформи, яка синтезує найкращі у своєму класі компоненти. Пропонована архітектура поєднує Kubernetes як підсистему оркестрації, MLflow як підсистему відстеження та Docker, що детально описано у розділі 3.

Було проведено практичну реалізацію та експериментальну валідацію пропонованої платформи на базі Minikube. В ході реалізації було вирішено низку нетривіальних проблем інтеграції компонентів (напр., залежності boto3 для MLflow, оптимізація CPU-збірки PyTorch, вирішення конфлікту Model Registry API). Експеримент довів, що платформа здатна: 1) Ефективно оптимізувати ресурси, автоматично керуючи чергою завдань Job Queue у Kubernetes. 2) Надавати єдиний UI для комплексного аналізу та порівняння гетерогенних моделей (CNN, RandomForest, KNeighbors), що підтверджує досягнення мети кваліфікаційної роботи.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Конотопчик А. М. Контейнеризація програм машинного навчання. *Програмне та апаратне забезпечення в інформаційних технологіях* : матеріали Міжнар. наук.-практ. конф. молодих вчених та студентів (м. Луцьк, 6 травня 2025 р.). Луцьк : ЛНТУ, 2025. Вип. 1. С. 92-97.

2. Artem Konotopchyk, Kateryna Melnyk, Svitlana Lavrenchuk, Nataliia Khrystynets, Pavlo Melnyk, Kateryna Bortnyk. System of dynamic optimization pricing by machine learning / *The 14th IEEE International Conference on Dependable Systems, Services and Technologies (DESSERT'2024)*. Greece, Athens, 11-13 October, 2024. P. 1-5.

3. Türkmen G., Sezen A., Şengül G. Comparative Analysis of Programming Languages Utilized in Artificial Intelligence Applications: Features, Performance, and Suitability. *International Journal of Computer Engineering & Software Technology (IJCESEN)*. 2024. Vol. 10, no. 3. P. 461-469.

4. Kreuzberger D., Kühl N., Hirschl S. Machine Learning Operations (MLOps): Overview, Definition, and Architecture. *arXiv preprint arXiv:2205.02302*. 2022. URL: https://arxiv.org/abs/2205.02302 (дата звернення: 05.07.2025).

5. MLOps: Continuous delivery and automation pipelines in machine learning. *Google Cloud Architecture Center*. 2024. URL: https://docs.cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning (дата звернення: 25.07.2025).

6. Khaliq Z., Farooq S. U., Khan D. A. Artificial Intelligence in Software Testing : Impact, Problems, Challenges and Prospect. *arXiv preprint arXiv:2201.05371*. 2022. URL: https://arxiv.org/abs/2201.05371 (дата звернення: 20.08.2025).

7. Semmelrock H. et al. Reproducibility in Machine Learning-based Research: Overview, Barriers and Drivers. *arXiv preprint arXiv:2406.14325*. 2025. URL: https://arxiv.org/abs/2406.14325 (дата звернення: 20.08.2025).

8. Semmelrock H. et al. Reproducibility in Machine Learning-Driven Research. *arXiv preprint arXiv:2307.10320*. 2023. URL: https://arxiv.org/abs/2307.10320 (дата звернення: 22.08.2025).

9. Boué L., Kunireddy P., Subotić P. A Data Source Dependency Analysis Framework for Large Scale Data Science Projects. *arXiv preprint arXiv:2212.07951*. 2022. URL: https://arxiv.org/abs/2212.07951 (дата звернення: 25.08.2025).

10. Patil D. et al. Machine learning and deep learning: Methods, techniques, applications, challenges, and future research opportunities. *Trustworthy Artificial Intelligence in Industry and Society* / ed. by D. Patil et al. Deep Science Publishing, 2024. P. 28-81.

11. Openja M. et al. Studying the Practices of Deploying Machine Learning Projects on Docker. *The International Conference on Evaluation and Assessment in Software Engineering 2022 (EASE 2022)*. 2022. P. 190-200.

12. Marella V., Tran. Comparative Analysis of Container Orchestration Platforms: Kubernetes vs. Docker Swarm. *International Journal of Scientific Research in Science and Technology*. 2024. Vol. 11, no. 5. P. 526-543.

13. Chorozidis G. et al. Knowledge and research mapping of the data and database forensics domains: A bibliometric analysis. *Information and Software Technology*. 2024. Vol. 171. Art. 107472. P. 211-233.

14. Trabelsi I. et al. A Systematic Literature Review of Machine Learning Approaches for Migrating Monolithic Systems to Microservices. *arXiv preprint arXiv:2508.15941*. 2025. URL: https://arxiv.org/abs/2508.15941 (дата звернення: 15.09.2025).

15. Maharjan R. et al. A Case Study on Monolith to Microservices Decomposition with Variational Autoencoder-Based Graph Neural Network. *Future Internet*. 2025. Vol. 17, no. 7. Art. 303. P. 110-131.

16. Kubeflow Architecture. *Kubeflow Documentation*. 2024. URL: https://www.kubeflow.org/docs/started/architecture/ (дата звернення: 15.09.2025).

17. Machine Learning Frameworks. *Amazon SageMaker Developer Guide*. 2024. URL: https://docs.aws.amazon.com/sagemaker/latest/dg/frameworks.html (дата звернення: 20.09.2025).

18. Best-of-Breed: How to Pick a Composable Commerce Tech Partner. *commercetools Blog*. 2023. URL: https://commercetools.com/blog/best-of-breed-how-to-pick-a-composable-commerce-tech-partner (дата звернення: 20.09.2025).

19. Argo Workflows - Overview. *Argo Project Documentation*. 2024. URL: https://argoproj.github.io/workflows/ (дата звернення: 21.09.2025).

20. Use Cases. *Apache Airflow Documentation*. 2024. URL: https://airflow.apache.org/use-cases/ (дата звернення: 21.09.2025).

21. MLflow Tracking. *MLflow Documentation*. 2024. URL: https://mlflow.org/docs/latest/ml/tracking/ (дата звернення: 22.09.2025).

22. DVC Use Cases. *DVC Documentation*. 2024. URL: https://dvc.org/doc/use-cases (дата звернення: 22.09.2025).

23. Poniszewska-Marańda A., Czechowska E. Kubernetes Cluster for Automating Software Production Environment. *Sensors*. 2021. Vol. 21, no. 5. Art. 1910. P. 53-76.

24. Berberi L. et al. MLflow and its usage. *ResearchGate*. 2023. URL: https://www.researchgate.net/publication/376029507_MLflow_and_its_usage (дата звернення: 22.09.2025).

25. Object Storage for AI. *MinIO Solutions*. 2024. URL: https://www.min.io/solutions/object-storage-for-ai (дата звернення: 24.09.2025).

26. Docker overview. *Docker Documentation*. 2024. URL: https://docs.docker.com/get-started/docker-overview/ (дата звернення: 24.09.2025).

27. Top 10 CI/CD Tools to Consider in 2024. *Qovery Blog*. 2024. URL: https://www.qovery.com/blog/top-10-cicd-tools-to-consider (дата звернення: 25.09.2025).