Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Факультет інформатики

Кафедра інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри інформатики

к.ф.-м.н., доц. Нагірна А.М.

_____

"_____" _____ 2026 р.

Магістерська робота

освітній ступінь – магістр

на тему:

«ДЕЦЕНТРАЛІЗОВАНА МЕРЕЖА GPU-ОБЧИСЛЕНЬ ДЛЯ РОЗПОДІЛЕНОГО НАВЧАННЯ НЕЙРОННИХ МЕРЕЖ»

Виконав: студент 2-го року навчання,
магістерської програми
«Інженерія програмного забезпечення»
Яременко Петро Всеволодович

Керівник: Гороховський С.С.,
к.ф.-м.н., доцент

Рецензент _____
(прізвище та ініціали)

Магістерська робота захищена
з оцінкою _____
Секретар ЕК _____
«_____» _____ 2026 р.

Київ – 2026

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри інформатики

к.ф.-м.н., доц. Нагірна А.М.

_____

"_____" _____ 2026 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на магістерську роботу

студенту 2 р.н. магістерської програми «Інженерія програмного забезпечення» Яременку Петру Всеволодовичу

Розробити децентралізовану мережу GPU-обчислень для розподіленого навчання нейронних мереж

Зміст пояснювальної записки до кваліфікаційної роботи:

Індивідуальне завдання

Перелік умовних позначень і термінів

Вступ

Розділ 1. Дослідження та аналіз предметної області

Розділ 2. Технологічна база платформи з децентралізованим постачанням GPU

Розділ 3. Агрегація обчислювальних потужностей від незалежних постачальників

Розділ 4. Архітектура системи

Розділ 5. Розробка системи

Розділ 6. Експериментальна оцінка

Висновки

Список використаної літератури

Додатки (за необхідністю)

Дата видачі "10" грудня 2025 р.

Керівник С.С. Гороховський, к.ф.-м.н., доцент _____ (підпис)

Завдання отримав П.В. Яременко _____ (підпис)

Тема: Децентралізована мережа GPU-обчислень для розподіленого навчання нейронних мереж

КАЛЕНДАРНИЙ ПЛАН ВИКОНАННЯ РОБОТИ:

№	Назва етапу	Термін	Примітка
1	Отримання завдання на дипломну роботу	03.11.2025	
2	Огляд технічної літератури за темою роботи	17.11.2025	
3	Аналіз сучасних інструментів розподіленого навчання нейронних мереж	01.12.2025	
4	Аналіз обраних інструментів на предмет можливості створення децентралізованої мережі	28.12.2025	
5	Проектування програмного інтерфейсу та вибір бібліотек для системи	18.01.2026	
6	Побудова системи за створеним проектом	01.02.2026	
7	Дослідження інструментів для оркестрації вузлів у децентралізованому середовищі	16.02.2026	
8	Проектування сервера-оркестратора	26.02.2026	
9	Розробка сервера-оркестратора	10.03.2026	
10	Проектування уніфікованого вузла тренування	20.03.2026	
11	Розробка уніфікованого вузла тренування	05.04.2026	
12	Проектування протоколу внутрішньої комунікації вузлів тренування з оркестратором	28.04.2026	
13	Інтеграція протоколу та використання сервера у проєкті	02.05.2026	
14	Тестування роботи проєкту	06.05.2026	

15	Створення презентаційних матеріалів та написання доповіді	13.05.2026	
16	Попередній захист магістерської роботи	22.05.2026	
17	Остаточне оформлення пояснювальної записки	26.05.2026	
18	Захист магістерської роботи	04.06.2026	

Студент Яременко П.В.

Керівник Гороховський С.С. "____"_____ 2026 р.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ І ТЕРМІНІВ

У роботі використовуються наведені нижче терміни, власні назви та абревіатури. Власні назви технологій, фреймворків і бібліотек подано в оригінальному написанні, описові терміни — українською мовою.

Термін у роботі	**Оригінальна назва**	**Пояснення**
NVIDIA FLARE	NVIDIA Federated Learning Application Runtime Environment	Фреймворк федеративного навчання від NVIDIA, який використовується як основа оркестрації тренування у розробленій платформі
pyo3	pyo3	Бібліотека для інтеграції Rust та Python; забезпечує виклик NVIDIA FLARE Admin API з Rust-диспетчера без міжпроцесного RPC
tonic	tonic	Реалізація gRPC для Rust на основі Tokio; використана для серверної частини диспетчера
Garage S3	Garage	Розподілене об’єктне сховище з S3-сумісним API, використане як робочий простір для шардів даних та артефактів моделі
Центральний бекенд	platform-api	NestJS-сервіс — єдина точка входу для клієнтського SDK, що містить модулі реєстру вузлів, обробки сигналів присутності, відбору вузлів та керування завданнями
Диспетчер	flare-dispatcher	Rust-сервіс з gRPC-інтерфейсом, який керує запуском і циклом життя контейнерів NVIDIA FLARE через локальний Docker-сокет та інтегрується з NVIDIA FLARE Admin API через pyo3
Вузол навчання	Worker	Docker-контейнер на GPU-машині незалежного постачальника з NVIDIA FLARE-клієнтом, який автономно реєструється у мережі, опитує чергу завдань і виконує локальне навчання

Федеративне навчання (ФН)	Federated Learning	Парадигма розподіленого машинного навчання: дані залишаються локально на вузлах, між вузлами передаються лише оновлення ваг моделі
Сервер агрегації	FL Server	Компонент NVIDIA FLARE, який керує циклом навчання та агрегує оновлення ваг від клієнтів навчання
Клієнт навчання	FL Client	Компонент NVIDIA FLARE на GPU-вузлі, який виконує локальне навчання на призначеному шарді даних
Раунд навчання	Training Round	Один повний цикл: розподіл глобальної моделі, локальне навчання на всіх клієнтах, збір оновлень, агрегація
FedAvg	Federated Averaging	Базовий алгоритм агрегації у федеративному навчанні: зважене середнє оновлень ваг від усіх клієнтів
Зберігач моделі	Persistor (CustomPersistor)	Клас Python у пакеті NVIDIA FLARE, який визначає логіку збереження глобальної моделі після кожного раунду; у роботі реалізовано запис безпосередньо у клієнтський робочий простір Garage S3
Виконавець клієнта	Executor (CustomExecutor)	Клас Python у пакеті NVIDIA FLARE, який інкапсулює логіку локального навчання на боці клієнта
Шард даних	Data Shard	Фрагмент навчального набору даних у форматі ZIP-архіву; один шард відповідає одному клієнту навчання
Сигнал присутності	Heartbeat	Періодичний сигнал від вузла навчання до центрального бекенду з метриками GPU; відсутність понад поріг означає недоступність
Суверенітет даних	Data Sovereignty	Архітектурний принцип: сирі навчальні дані клієнта не залишають контрольоване клієнтом середовище
Відстаючий вузол	Straggler	Повільний вузол у розподіленій системі, який затримує завершення раунду навчання
ViT	Vision Transformer	Архітектура нейронної мережі для задач комп’ютерного зору на основі механізму уваги

TTA	Test-Time Augmentation	Аугментація на етапі валідації: усереднення прогнозу моделі по кількох трансформаціях вхідного зображення
Non-IID	Non-Independent, non-Identically Distributed	Розподіл даних, у якому статистика розподілу різна між клієнтами
Атака інверсії градієнтів	Gradient Inversion Attack	Клас атак на федеративне навчання, що дозволяє частково реконструювати навчальні приклади з переданих градієнтів

ЗМІСТ

ВСТУП

Актуальність теми. За останні п'ять років обчислювальні вимоги до тренування нейронних мереж зросли на кілька порядків. Sevilla та співавтори [15] оцінюють річний приріст обсягу обчислень для передових моделей у 4–5 разів — це випереджає зростання обчислювальної щільності окремого GPU за законом Мура та темпи здешевлення GPU-годин у хмарних провайдерів. Llama 3.1 405B від Meta, випущена у 2024 році, була натренована на 16 000 GPU NVIDIA H100 протягом приблизно 100 днів, що склало 39,3 мільйона GPU-годин обчислень [10]; за третьосторонніми аналітичними оцінками, опублікованими після релізу моделі, її тренування коштувало від 60 до 640 мільйонів доларів США залежно від моделі обліку капітальних та операційних витрат на кластер [11]. Дотренування фундаментальних моделей середнього масштабу обсягом 7–70 мільярдів параметрів вимагає кластерів із кількох GPU з високошвидкісним інтерконектом NVLink або InfiniBand, недоступним для більшості академічних груп, незалежних дослідницьких команд і невеликих підприємств.

Стандартною відповіддю на цю обмеженість є хмарне забезпечення GPU-обчислень через гіперскейлерів Amazon Web Services, Google Cloud Platform та Microsoft Azure. Такі платформи усувають потребу у власному обладнанні, але створюють для невеликих команд структурні проблеми. Погодинна оренда GPU класу H100 коштує від 3 до 12 доларів, що робить тривале тренування навіть скромних моделей економічно непідйомним. Розподілене навчання на кількох хмарних GPU вимагає компетенції у сфері DevOps та MLOps — налаштування мережі між вузлами, бібліотек колективних комунікацій NCCL та MPI, відмовостійкості і контрольних точок — якою більшість ML-команд самостійно не володіє. Для доменів із чутливими даними централізована модель створює регуляторне тертя: сирі навчальні дані не можуть залишати контрольованого середовища через вимоги GDPR (EU), HIPAA (США), LGPD (Бразилія) та аналогічних рамок.

Встановлене GPU-обладнання у світі при цьому використовується далеко не повністю. Flexera 2025 State of the Cloud Report [23] оцінює, що близько 27 відсотків корпоративних хмарних витрат залишаються невикористаними або неоптимізованими; для GPU-обчислень як категорії витрат із найвищою почасовою ставкою недозавантаженість має особливо високу економічну вагу. У публікації команди ByteDance [24] про їхній виробничий кластер на понад 20 000 GPU зафіксовано базову утилізацію близько 26 відсотків до впровадження спеціалізованих систем спільного використання. Поєднання високого попиту на GPU-обчислення зі значною недозавантаженістю наявних потужностей формує структурну можливість для побудови платформ, які агрегують гетерогенне обладнання від незалежних постачальників у єдиний обчислювальний пул, керований єдиним шаром оркестрації.

Децентралізовані GPU-маркетплейси на кшталт Vast.ai, Akash Network та io.net надають доступ до обчислювальних ресурсів, але не оркеструють цикл навчального завдання — клієнт отримує SSH-ключ і самостійно займається всією оркестрацією. Виробничі FL-платформи закритого коду — Rhino Health Federated Computing Platform [2] для медичних консорціумів та Lilly TuneLab [6] для біотех-співпраці на пропрієтарних моделях Eli Lilly — пропонують саме керований досвід оркестрації федеративного навчання, але прив'язані до фіксованого вертикального домену та інституційного складу учасників. Розроблена у роботі платформа займає суміжну, але відмінну нішу: відкритий код, не пов'язана з конкретним доменом, з агрегацією обладнання від незалежних постачальників за моделлю Akash та Vast.ai. Обчислювальні вузли у цій моделі є автономними учасниками мережі — кожен постачальник самостійно вирішує, коли запустити свій GPU-контейнер, які характеристики свого обладнання повідомити при реєстрації та коли вийти з мережі без узгодження з оператором платформи; вузли активно опитують центральний бекенд щодо доступних завдань і самостійно ініціюють їх виконання. Сам центральний бекенд координує створення FL-підмереж і не бере участі у тренувальному циклі: після провізіонування підмережі сервер агрегації та клієнти на вузлах постачальників

взаємодіють напряму через FL-протокол, і навчальні раунди завершуються без зворотних звертань до бекенду. Це робить FL-підмережі атомарними та ізолює їх від можливих перебоїв на шарі керування. Платформа оркеструє робочі процеси федеративного навчання (Federated Learning, FL [9]) на цих незалежно керованих гетерогенних вузлах і надає клієнту керований досвід тренування з ізоляцією сирих навчальних даних та відтворюваністю результатів.

Мета дослідження полягає у проектуванні та реалізації керованої платформи GPU-обчислень із децентралізованим постачанням обладнання для розподіленого навчання нейронних мереж, що інтегрує NVIDIA FLARE як механізм оркестрації федеративного навчання на гетерогенних вузлах та забезпечує ізоляцію сирих навчальних даних клієнтів від постачальників обчислень.

Завдання дослідження:

— здійснити огляд сучасних стратегій розподіленого навчання нейронних мереж — паралелізм даних, паралелізм моделі, паралелізм конвеєра та федеративне навчання — і проаналізувати наявні платформи GPU-обчислень за категоріями централізованих хмар, спеціалізованих GPU-провайдерів, децентралізованих маркетплейсів та керованих FL-платформ закритого коду;

— обґрунтувати вибір федеративного навчання як парадигми платформи та проаналізувати NVIDIA FLARE як виробничий фреймворк оркестрації федеративного навчання;

— визначити технологічну базу платформи: стандартизацію середовища виконання через Docker, асинхронну комунікацію з вузлами через Redis, синхронне управління провізіонуванням через gRPC та об’єктне сховище для робочих просторів клієнтів;

— спроектувати мікросервісну архітектуру з розподілом відповідальності між центральним бекендом, диспетчером та вузлами навчання, формалізувати модель загроз та межі гарантій ізоляції даних у поточній реалізації;

— реалізувати компоненти платформи: центральний бекенд на TypeScript з NestJS, диспетчер на Rust з gRPC-сервером tonic та інтеграцією з NVIDIA

FLARE Admin API через pyo3, а також контейнерний образ вузла навчання, який автономно реєструється у мережі і самостійно отримує завдання від центрального бекенду;

— провести наскрізну емпіричну оцінку платформи на репрезентативному виробничому навантаженні — модель ViT-B/16 [22], дотренована на наборі даних Food-101 [21] — з порівнянням федеративного та централізованого виконання того ж рецепту, аналізом внеску модифікацій рецепту та фіксацією меж застосовності отриманих результатів.

Об’єктом дослідження є платформи розподілених обчислень для навчання нейронних мереж у гетерогенних середовищах з постачанням обладнання від незалежних учасників.

Предметом дослідження є архітектурні моделі та протоколи міжкомпонентної взаємодії у керованих платформах GPU-обчислень із децентралізованим постачанням обладнання, а також моделі загроз і межі гарантій ізоляції даних у таких платформах.

Методи дослідження включають порівняльний аналіз стратегій тренування і наявних платформ, мікросервісне проектування з контрактною взаємодією сервісів через типізовані протоколи, інтеграцію промислового фреймворку федеративного навчання NVIDIA FLARE та емпіричну оцінку через порівняння федеративного та централізованого виконання того ж рецепту на репрезентативному виробничому навантаженні. Емпірична частина базується на спостереженнях із зафіксованим обчислювальним бюджетом і явно фіксує кількість повторів кожної конфігурації, обмеження топології та режим розподілу даних.

Новизна одержаних результатів. Робота має інженерний характер. Перелічені нижче пункти стосуються інженерно-практичних рішень та емпіричних спостережень і не претендують на формальну наукову новизну у сенсі нової теореми чи алгоритмічного результату.

1. Розроблено архітектурну композицію керованої FL-платформи з відкритим постачанням GPU-обладнання, яка поєднує патерни керованих

хмарних провайдерів (AWS SageMaker, GCP Vertex AI), децентралізованих маркетплейсів (Akash, Vast.ai) та комерційних керованих FL-платформ закритого коду (Rhino Health FCP, Lilly TuneLab). Композиція реалізована як відкритий програмний продукт, не прив'язаний до конкретного вертикального домену, що відрізняє його від комерційних аналогів. Внесок є продуктовим та архітектурним; це не претензія на новий клас систем чи на наукову новизну архітектурного рівня.

2. Реалізовано полімовну архітектуру керованої FL-платформи з розподілом відповідальностей: TypeScript на основі NestJS для шару бізнес-логіки, Rust на основі tonic+pyo3 для керування контейнерами та інтеграції з NVIDIA FLARE Admin API в межах одного процесу. Внесок є інженерним рішенням, яке демонструє практичну застосовність pyo3-моста для інтеграції Rust-сервісу з Python-бібліотекою у виробничому контексті.

3. Виконано серію тренувальних експериментів з рецептом FedAvg-навчання ViT-B/16 на Food-101 з аналізом внеску модифікацій рецепту (LR-розклад, стан оптимізатора, label smoothing, EMA, розмір батча) у режимі коротких FL-раундів. Серія документує конкретні параметри рецепту — безперервний LR через глобальний крок, persistent Adam state — які дають найкращий результат у даній конфігурації. Внесок є практичним рецептом для подібних навантажень; узагальнення на інші моделі, набори даних та режими розподілу даних (non-IID) виходить за межі цієї роботи.

Практичне значення одержаних результатів полягає у можливості виконувати федеративне навчання нейронних мереж на гетерогенних GPU-ресурсах без ручного управління інфраструктурою. Задокументовані у роботі архітектурні рішення — gRPC для синхронного управління контейнерами між сервісами рівня керування, Redis як асинхронний брокер повідомлень для зв’язку з вузлами, NVIDIA FLARE для оркестрації федеративного навчання та об’єктне сховище Garage як S3-сумісний робочий простір клієнтів — застосовні до інших систем розподілених обчислень із вимогами щодо ізоляції даних між учасниками, передусім у медичній візуалізації, фармацевтичних дослідженнях та

фінансовому секторі, де парадигма FL вже використовується у виробничих розгортаннях [2, 6, 12]. Емпіричний рецепт федеративного дотренування моделей сімейства Vision Transformer із безперервним розкладом швидкості навчання та збереженням стану оптимізатора між раундами безпосередньо переноситься на інші розгортання федеративного навчання на базі NVIDIA FLARE без модифікації коду фреймворку.

Структура роботи. Магістерська робота складається зі вступу, шести розділів, висновків, списку використаних джерел та додатків. Розділ 1 розглядає стратегії розподіленого навчання, обґрунтовує вибір федеративного навчання як парадигми платформи, аналізує NVIDIA FLARE як обраний фреймворк оркестрації та класифікує наявні платформи GPU-обчислень із виокремленням комерційних керованих FL-платформ як близького конкурентного простору. Розділ 2 визначає технологічну базу платформи — контейнеризацію через Docker, протоколи зв’язку (gRPC та Redis), об’єктне сховище Garage S3 — і формулює модель загроз з рівнями захисту даних. Розділ 3 описує модель агрегації обчислювальних потужностей від незалежних постачальників із розглядом автономної реєстрації вузлів, обробки сигналів присутності та підготовки клієнтських завдань. Розділ 4 представляє архітектуру платформи з трьох компонентів та описує реалізаційні алгоритми. Розділ 5 документує реалізацію кожного компонента, структуру пакету NVIDIA FLARE та підхід до тестування. Розділ 6 містить емпіричну оцінку — порівняння федеративного та централізованого виконання того ж рецепту, аналіз внеску основних модифікацій рецепту, фіксацію меж застосовності отриманих результатів.

РОЗДІЛ 1. ДОСЛІДЖЕННЯ ТА АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Стратегії розподіленого навчання нейронних мереж

Обчислювальна вартість навчання нейронних мереж зростала швидше за пам’ять та пропускну спроможність окремого GPU. Модель з 70 мільярдами параметрів у форматі 32-бітної точності потребує приблизно 280 ГБ лише для збереження ваг без урахування градієнтів, станів оптимізатора та проміжних активацій (розрахунок у fp32 наведено як консервативну верхню межу — виробничі тренування LLM такого масштабу використовують bf16 або fp8, що зменшує цю вимогу в 2–4 рази, але не змінює якісного висновку про неможливість розміщення на одному пристрої). Промислові GPU NVIDIA класу H100 мають 80 ГБ HBM3, варіант H100 NVL — 94 ГБ, нове покоління NVIDIA B200 (Blackwell) — 192 ГБ HBM3e. Навіть з урахуванням найновіших пристроїв тренування моделей з сотнями мільярдів параметрів на одному GPU фізично неможливе, і розподілене навчання залишається структурною вимогою [13]. Нижче розглянуто чотири основні сімейства стратегій розподілу.

Паралелізм даних (Data Parallelism, DDP) реплікує повну модель з параметрами w на кожному з K GPU та розділяє навчальний набір даних D розміру n на K непересічних підмножин D_1, D_2, ..., D_K. Кожен GPU k обчислює локальний градієнт g_k на власному міні-батчі b_k з D_k. Після кожного проходу назад градієнти синхронізуються між усіма репліками через колективну операцію AllReduce, яка обчислює усереднений градієнт g_avg = (1/K) · sum(g_k) та повертає його кожному пристрою. Усі K пристроїв оновлюють параметри одночасно за правилом w := w − η · g_avg, де η — швидкість навчання. Стратегія добре масштабується доти, доки повна модель вміщується у пам’яті одного пристрою. Для моделей до приблизно 7 мільярдів параметрів у напівточному форматі fp16 чи bf16 DDP залишається стандартним вибором у централізованих та федеративних розгортаннях.

Паралелізм моделі (Model Parallelism) розбиває саму модель між пристроями — різні шари, голови уваги або тензорні блоки розташовуються на різних GPU. Це дозволяє тренувати моделі, які перевищують пам’ять одного пристрою, ціною значної складності у балансуванні навантаження та передачі проміжних активацій між пристроями. Megatron-LM [13] продемонструвала тензорно-паралельне навчання багатомільярдних мовних моделей через розділення окремих трансформерних блоків між пристроями. Техніка залишається фундаментальною для виробничого навчання моделей передового масштабу.

Паралелізм конвеєра (Pipeline Parallelism) розвиває паралелізм моделі через організацію пристроїв у послідовні стадії конвеєра. Мікробатчі обробляються різними стадіями одночасно, що зменшує час простою, властивий наївному паралелізму моделі. GPipe [4] навчила 128-шаровий трансформер на 6 мільярдів параметрів на Cloud TPUv3 із синхронним оновленням градієнтів. Паралелізм конвеєра типово поєднується з паралелізмом даних та тензорним паралелізмом у гібридних 3D-паралельних конфігураціях, які застосовуються для тренування найбільших сучасних моделей.

DeepSpeed ZeRO та FSDP являють собою фреймворк оптимізації пам’яті, який поступово шардує стан тренування між пристроями [33]. ZeRO-1 шардує стани оптимізатора, що для Adam з fp16-параметрами та fp32-станами дає приблизно 4-кратне зменшення пам’яті відносно DDP; ZeRO-2 додатково шардує градієнти з 8-кратним зменшенням; ZeRO-3 шардує параметри з 64-кратним зменшенням для типової моделі-трансформера. Fully Sharded Data Parallel (FSDP) у PyTorch є спорідненим підходом, який за замовчуванням шардує всі три компоненти.

Федеративне навчання (Federated Learning) [9], якому присвячено окрему увагу у підрозділі 1.2, є парадигмою, на якій базується реалізована платформа. На відміну від чотирьох попередніх стратегій, FL припускає, що дані залишаються локально на вузлах і не перерозподіляються між пристроями. Така властивість робить FL природним вибором для середовищ із незалежними

постачальниками обчислень, де передача даних між учасниками неможлива через регуляторні або конкурентні обмеження, а самі обчислювальні вузли не перебувають під єдиним адміністративним контролем. DDP, FSDP та DeepSpeed ZeRO не реалізовані у поточній версії агента вузла, але визначені у Розділі 6 як основний напрямок подальшого розвитку. NVIDIA FLARE — обраний у роботі фреймворк оркестрації — підтримує координацію всіх чотирьох стратегій під уніфікованою специфікацією завдання, що дозволяє платформі поступово розширювати функціональність без архітектурних змін.

1.2. Федеративне навчання як інструмент платформи

Федеративне навчання, запропоноване McMahan та співавторами у 2017 році [9], є парадигмою розподіленого тренування, у якій навчальні дані залишаються на окремих вузлах-учасниках, а між вузлами та центральним сервером агрегації передаються лише оновлення ваг моделі. Парадигма виникла з потреб мобільних застосувань Google: тренування моделей передбачення на даних користувацьких пристроїв, які з міркувань конфіденційності та закону не могли бути централізовано зібрані. У роботі FL розглядається не як предмет дослідження, а як обраний робочий інструмент для платформи — вибір зумовлено властивостями FL, що відповідають вимогам мережі з незалежно керованими GPU-вузлами.

Базовий алгоритм FL — FedAvg [9] — задає синхронну схему виконання раунду навчання. На початку раунду сервер агрегації розсилає поточні глобальні ваги моделі всім клієнтам. Кожен клієнт локально виконує задану кількість епох градієнтного спуску на власному наборі даних і повертає оновлені ваги. Сервер усереднює отримані ваги, зважуючи їх за розміром локального набору даних, і отримує нову глобальну модель — на ній починається наступний раунд. Платформа використовує FedAvg як стандартний алгоритм агрегації; NVIDIA FLARE при цьому дозволяє клієнту обрати інший алгоритм без зміни коду платформи.

Властивості FL, які зробили його обраним інструментом для платформи, формулюються безпосередньо. На рівні протоколу FL клієнтські дані не передаються між учасниками: GPU-вузли незалежних постачальників отримують від сервера агрегації лише ваги поточної глобальної моделі та навчальні скрипти, але не сам набір даних. Структурна ізоляція цього рівня (L1, детально у розділі 2.5) не залежить від чесності постачальників і не потребує контрактних чи процедурних механізмів. Повний захист від атак з боку недовіреного постачальника — наприклад, копіювання шарду даних з контейнера, де його змонтовано — потребує signed image manifests, довірених середовищ виконання, репутаційних систем; у поточній моделі загроз припускається honest-but-curious поведінка постачальників. Раундова модель комунікації толерантніша до гетерогенного обладнання та ненадійних мережевих умов, ніж синхронне навчання з паралелізмом даних: раунди оперують грубішою часовою гранулярністю в межах хвилин, а не мілісекунд, і фреймворк здатний обробляти втрату окремих клієнтів у межах раунду через часткову агрегацію. FL також підтримує учасників з різними класами GPU, об’ємами пам’яті та програмними середовищами за умови, що кожен вузол здатний завершити локальне тренування у межах налаштованого таймауту раунду. Ця властивість прямо узгоджується з природою мережі — вузол може приєднатися чи вийти у будь-який момент між раундами, і його обладнання заздалегідь невідоме оператору.

Сучасні огляди [3] виділяють кілька викликів, властивих FL. Статистична гетерогенність виникає через відмінності розподілу даних між клієнтами (Non-IID); системна гетерогенність із різною обчислювальною пропускною спроможністю породжує відстаючі вузли; накладні витрати на комунікацію зростають через передачу мільярдів параметрів кожен раунд. Розробка алгоритмів, які адресують ці виклики, є активною областю досліджень. NVIDIA FLARE підтримує FedAvg, FedProx [5] для зменшення впливу Non-IID, SCAFFOLD та користувацькі агрегатори через механізм Aggregator; конкретний вибір алгоритму належить клієнтові платформи. Реалізована у роботі платформа

адресує системну гетерогенність через детермінований відбір вузлів за вільною відеопам'яттю, описаний у підрозділі 3.2. Толерантність до часткових відмов вузлів у середині раунду навчання забезпечується вбудованим механізмом часткової агрегації NVIDIA FLARE, описаним у підрозділі 1.3; розширення цього механізму до автоматичного перепризначення відмовлих вузлів винесено у перспективи розвитку (Розділ 6). Non-IID пом'якшується на етапі підготовки клієнтських даних: клієнтам рецепту експерименту з Розділу 6 виконано випадкове перемішування з фіксованим seed перед шардингом, що у типовому випадку забезпечує приблизно IID-розподіл між шардами.

1.3. NVIDIA FLARE як фреймворк оркестрації федеративного навчання

NVIDIA FLARE [7] є промисловим програмним набором розробника (SDK) для федеративного навчання з відкритим кодом, розробленим компанією NVIDIA та випущеним у 2022 році. Серед альтернатив у цьому просторі — Flower [27] та TensorFlow Federated — FLARE вирізняється орієнтацією на виробниче розгортання через формальні специфікації завдань, офіційні Docker-образи, вбудовану відмовостійкість та компонентну модель розширюваності. Бенчмарк-порівняння NVIDIA FLARE, Flower та Owkin Substra на наборі PathMNIST у задачі класифікації медичних зображень [14] показало, що FLARE демонструє кращу масштабованість у виробничих сценаріях, тоді як Flower надає більшу гнучкість для прототипування та академічних досліджень, а Substra — потужніші механізми приватності та комплаєнсу.

Архітектура FLARE побудована навколо двох ролей. Сервер агрегації FL Server зберігає поточну глобальну модель, розподіляє її між клієнтами на початку кожного раунду, збирає оновлені локальні ваги та застосовує алгоритм агрегації для отримання наступної глобальної моделі. Клієнт навчання FL Client виконується на кожному вузлі-учаснику, отримує глобальну модель, виконує локальне навчання на призначеному шарді даних та надсилає оновлені ваги назад на сервер. Взаємодія відбувається за шаблоном scatter-and-gather: сервер розсіює

модель, клієнти виконують локальне навчання, сервер збирає оновлення, відбувається агрегація, цикл повторюється.

Для платформи, реалізованої у роботі, релевантні кілька механізмів розширюваності FLARE. Зберігач моделі (Persistor) — клас Python, який визначає логіку збереження глобальної моделі між раундами. За замовчуванням FLARE записує контрольні точки до робочого каталогу сервера; у роботі реалізовано користувацький зберігач, який записує контрольні точки безпосередньо до клієнтського робочого простору в об'єктному сховищі, унаслідок чого навчальні артефакти стають доступними клієнту негайно після кожного раунду без окремого кроку завантаження. Виконавець клієнта (Executor) інкапсулює логіку локального навчання на клієнті та є природною точкою інтеграції з довільним кодом тренування на PyTorch, TensorFlow, JAX або чистому NumPy. FLARE не накладає на користувача жорстких обмежень щодо ML-фреймворку, що підтверджено офіційними інтеграціями з NeMo [8] та MONAI для медичних застосувань. Контролер (Controller) визначає робочий процес на стороні сервера разом із кількістю раундів, мінімальною участю клієнтів та політиками таймаутів; FLARE надає ScatterAndGather-контролер як стандартну реалізацію для FedAvg-стилю тренування.

Якщо клієнт втрачає з'єднання у середині раунду, сервер агрегації завершує крок агрегації з оновленнями від решти клієнтів, а не перериває раунд. Параметр min_clients у config_fed_server.json визначає мінімальну кількість клієнтів, необхідну для агрегації; за замовчуванням він рівний загальній кількості клієнтів і може бути зменшений для толерування часткових відмов. Ця властивість є фундаментом для автоматичного перепризначення, описаного у Розділі 4.

1.4. Виробничі розгортання NVIDIA FLARE

NVIDIA FLARE наразі використовується у виробничих розгортаннях федеративного навчання у трьох галузевих сегментах. Кожен з них забезпечує реальне референсне використання архітектурних шаблонів, прийнятих у роботі.

Охорона здоров'я та медична візуалізація. Rhino Health адаптувала NVIDIA FLARE для федеративної розробки AI у співпраці з Massachusetts General Hospital у задачі діагностики аневризм мозку та з National Cancer Institute Early Detection Research Network у задачі раннього виявлення раку підшлункової [2]. У дослідженні 2025 року в галузі обчислювальної патології, опублікованому у Journal of Pathology Informatics [12], FLARE розгорнуто на трьох клінічних клієнтах у чотирьох країнах через University Hospital Zurich та партнерські установи для тренування моделей глибокого навчання, призначених для цифрової імунної фенотипізації метастатичної меланоми. Дослідження виявило конкретні виклики виробничого середовища — обмеження мережі лікарень, тривалість експериментів через системну гетерогенність, потребу у IT-компетенції для конфігурації FLARE — які безпосередньо адресує підхід керованої оркестрації, прийнятий у роботі.

Фармацевтичні дослідження та розробка лікарських препаратів. У 2026 році компанія Eli Lilly запустила LillyPod NVIDIA DGX SuperPOD з 1016 GPU NVIDIA Blackwell Ultra як найбільшу AI-фабрику, що повністю належить та керується фармацевтичною компанією [6]. Окремі моделі з цієї інфраструктури доступні через Lilly TuneLab — платформу розробки лікарських препаратів, побудовану на NVIDIA FLARE, яка дозволяє біотехнологічним компаніям отримувати доступ до пропрієтарних AI-моделей зі збереженням приватності власних даних та архітектурною ізоляцією від інших користувачів. Це розгортання валідує федеративне навчання як інфраструктуру для міжінституційної співпраці в розробці лікарських препаратів.

Фінансові послуги. Та сама парадигма федеративного навчання застосовується до виявлення шахрайства між фінансовими установами, де централізована агрегація даних блокується регуляторикою суверенітету даних та конкурентними міркуваннями [1]. Звіт NVIDIA State of AI in Financial Services

2024 ідентифікує виявлення шахрайства як головний випадок застосування AI у секторі та головний виклик безпеки; 51 відсоток опитаних установ очікують, що AI ефективно протидіятиме шахрайству. Roth та співавтори у роботі 2026 року [1] детально документують архітектуру FL-розгортання для виявлення фінансового шахрайства, де FedAvg використовується для агрегації моделей між установами без розкриття даних транзакцій.

Наведені розгортання релевантні для роботи з двох причин. Вони демонструють, що FLARE є зрілим виробничим середовищем виконання, а не академічним прототипом — це обґрунтовує його вибір як фреймворку оркестрації. Крім того, ці розгортання мають з реалізованою тут платформою спільний набір архітектурних занепокоєнь: ізоляція клієнтських даних, відмовостійкість на гетерогенних та ненадійних учасниках, чітке розділення між шаром оркестрації та специфічною для моделі логікою тренування.

1.5. Наявні платформи GPU-обчислень: класифікація та аналіз розриву

Наявні платформи для GPU-навчання нейронних мереж розподіляються на чотири категорії, кожна з яких займає різне положення у компромісі між вартістю, автоматизацією та ізоляцією даних.

Гіперскейл-провайдери Amazon Web Services, Google Cloud Platform та Microsoft Azure постачають GPU-обчислення як частину ширшого хмарного портфоліо. Керовані сервіси тренування на кшталт AWS SageMaker та GCP Vertex AI забезпечують високий ступінь автоматизації розподіленого навчання у межах власних екосистем. Обмеженнями виступають ціна (примірники H100 коштують від 3 до 12 доларів за GPU за годину), прив’язка до провайдера (міграція між провайдерами вимагає значного переписування) та потреба у компетенції, специфічній для провайдера, для налаштування розподілених конвеєрів. Ізоляція даних контролюється провайдером, а не гарантується архітектурно — вона залежить від політик IAM та контрактних зобов’язань, а не від структури самого протоколу тренування.

Спеціалізовані GPU-хмари Lambda Labs, CoreWeave та Vast.ai зосереджуються виключно на GPU-обчисленнях та типово пропонують нижчі почасові ставки, ніж гіперскейлери. Ці платформи не автоматизують цикл навчального завдання: клієнт отримує віртуальну машину з GPU та SSH-ключ і відповідає за все налаштування середовища, конфігурацію кількох вузлів, контрольні точки та обробку відмов. Користувача фактично просять самостійно побудувати відсутній шар оркестрації.

Децентралізовані GPU-маркетплейси Akash Network, io.net та Render Network агрегують обчислювальні ресурси від незалежних постачальників — власників окремих GPU та невеликих дата-центрів. Обчислювальні вузли у цій моделі є автономними учасниками: постачальники самостійно встановлюють агент маркетплейсу на власному обладнанні, оголошують доступність та умови, і обладнання фізично залишається у середовищі постачальника під його повним адміністративним контролем. Ціни типово найконкурентніші на ринку. Жодна з цих платформ не оркеструє цикл навчального завдання, не ізолює дані архітектурно та не забезпечує відновлення після відмов — як і спеціалізовані хмари, вони передають користувачу необроблений ресурс та SSH-ключ.

Фреймворки федеративного навчання Flower, TensorFlow Federated та NVIDIA FLARE реалізують самі алгоритми тренування, але не є платформами у сенсі надання керованої інфраструктури. Це бібліотеки чи SDK, які повинні бути розгорнуті та підтримуватися користувачем. NVIDIA FLARE надає інструментарій виробничого рівня для розгортання і є єдиним фреймворком цієї групи, який регулярно використовується як основа для керованих FL-платформ закритого коду.

Керовані FL-платформи на базі NVIDIA FLARE становлять найближчий конкурентний простір до розробленої у роботі платформи. Rhino Health Federated Computing Platform [2] обслуговує медичні консорціуми з фіксованим інституційним складом учасників: медичні установи реєструються як учасники консорціуму, а GPU-інфраструктура надається або власна, або через довірених партнерів Rhino. Lilly TuneLab [6] надає біотех-компаніям спільне

дотренування на пропрієтарних моделях Eli Lilly з ізоляцією експериментальних даних учасника; GPU-інфраструктура повністю належить Eli Lilly. Обидві платформи закриті, прив'язані до конкретного вертикального домену та не передбачають агрегації GPU-обладнання від незалежних постачальників, не пов'язаних з оператором платформи. Існують також академічні розгортання NVIDIA FLARE на базі окремих дослідницьких груп — наприклад, платформа University Hospital Zurich для досліджень обчислювальної патології [12] — але вони не публікують свою інфраструктуру як перевикористовуваний продукт.

Окремо слід згадати Ray як універсальну платформу оркестрації розподілених обчислень, яка активно використовується для тренування нейронних мереж у гетерогенних середовищах (Ray Train, Ray Tune). Ray надає primitive-рівень для побудови розподілених систем і вимагає від користувача самостійного забезпечення ізоляції даних між учасниками, відмовостійкості раунду та координації з гетерогенним обладнанням. Реалізована у роботі платформа займає вищий рівень абстракції, інкапсулюючи ці аспекти у керованому потоці FL. Кількісне порівняння з розгортанням на Ray (Ray Train + кастомний FL-агрегатор) винесене у перспективи розвитку.

Розроблена у роботі платформа займає суміжну, але відмінну нішу відносно перелічених рішень. Від децентралізованих маркетплейсів (Akash, Vast.ai) платформа відрізняється тим, що оркеструє повний цикл навчального завдання, а не лише надає доступ до обчислювальних ресурсів — при цьому зберігаючи ту саму модель постачання, у якій обладнання надходить від незалежних постачальників, які автономно приєднуються до мережі. Від гіперскейл-провайдерів (AWS SageMaker, GCP Vertex AI) — тим, що не прив'язана до GPU-парку єдиного оператора і агрегує гетерогенне обладнання від незалежних постачальників. Від комерційних керованих FL-платформ (Rhino Health, Lilly TuneLab) — тим, що є відкритим кодом, не прив'язана до конкретного вертикального домену та призначена для агрегації GPU від незалежних постачальників, а не від фіксованого пулу інституційних партнерів. Архітектурна комбінація decentralized-supply, managed-orchestration та

architectural-isolation-of-raw-data, реалізована у роботі, не задокументована у жодному з порівнюваних рішень — саме ця комбінація визначає інженерний внесок роботи. Внесок стосується конкретної композиції властивостей, які раніше реалізовувалися лише частково кожним з конкурентів, а не першості самої концепції керованої FL-платформи.

РОЗДІЛ 2. ТЕХНОЛОГІЧНА БАЗА ПЛАТФОРМИ З ДЕЦЕНТРАЛІЗОВАНИМ ПОСТАЧАННЯМ GPU

2.1. Контейнеризація та стандартизація середовища виконання

Гетерогенність GPU-вузлів у мережі незалежних постачальників — різні операційні системи, версії CUDA, конфігурації бібліотек — створює проблему відтворюваності: код тренування, який працює на одному вузлі, може зазнати збою на іншому через несумісність залежностей. За опитуванням Sambasivan та співавторів [28], проблеми відтворюваності складають значну частку інцидентів у ML-системах, причому більшість з них пов’язана з невідповідністю середовища виконання. Docker [16] розв’язує цю проблему через упакування застосунку разом з усіма залежностями в ізольований контейнер, поведінка якого однакова на будь-якому хості. Формально контейнер є примірником образу — незмінного шаруватого артефакту, що містить файлову систему, метадані та конфігурацію виконання. Ця властивість критична для мережі незалежних постачальників: оператор платформи не контролює конфігурацію хост-системи постачальника і не може гарантувати її однорідність, а контейнер забезпечує однакове середовище виконання поверх будь-якої сумісної версії хост-ОС.

Архітектурно Docker використовує namespaces та cgroups ядра Linux для ізоляції процесу контейнера від хост-системи. Namespaces ізолюють простори ідентифікаторів процесів, мережевих інтерфейсів, файлової системи та користувачів; cgroups обмежують споживання ресурсів процесора, пам’яті та операцій вводу-виводу. Для GPU-навантажень NVIDIA Container Toolkit [29] розширює Docker механізмом передачі пристроїв GPU у контейнер з прив'язкою до конкретної версії драйвера на хості, що зберігається у образі через бібліотеку libnvidia-container.

Образ вузла навчання побудовано на основі офіційного образу pytorch/pytorch з підтримкою CUDA та cuDNN, до якого додано NVIDIA FLARE-клієнт та допоміжні скрипти запуску. Образ є базою для всіх GPU-вузлів у

мережі і забезпечує однакове середовище виконання незалежно від конкретного фізичного обладнання. Окремий образ сервера агрегації NVIDIA FLARE використовується диспетчером під час провізіонування кожного навчального завдання. Фіксація конкретного тегу образу у конфігурації диспетчера гарантує відтворюваність навчального середовища між різними запусками одного завдання — властивість, особливо важлива для наукових цілей та аудиторських вимог регульованих галузей.

2.2. Оркестрація контейнерів через Docker Engine API

Платформа взаємодіє з контейнерами безпосередньо через Docker Engine API на кожному GPU-хості. Диспетчер монтує локальний сокет /var/run/docker.sock у власний контейнер і керує життєвим циклом контейнерів — створенням, запуском, моніторингом стану, видаленням — без проміжного шару кластерного оркестратора. Підхід відповідає природі мережі, де незалежні постачальники GPU надають окремі машини, а не вузли єдиного керованого кластера; введення додаткового рівня абстракції типу Kubernetes ускладнило б розгортання на боці постачальника без явної переваги в умовах поточного режиму експлуатації. Постачальник запускає на своїй машині лише два контейнери — диспетчер та робочий вузол — без потреби розгортати кластерну інфраструктуру.

Для кожного навчального завдання диспетчер виконує детерміновану послідовність дій. Запускається контейнер сервера агрегації NVIDIA FLARE з відкритими портами адміністративного API та FL-протоколу. На кожному відібраному GPU-хості запускається контейнер клієнта навчання, налаштований підключатися до сервера агрегації; шард даних монтується у контейнер через bind-mount з робочого простору клієнта. Диспетчер передає до сервера агрегації специфікацію завдання через NVIDIA FLARE Admin API, після чого починаються раунди тренування. Після завершення завдання диспетчер зупиняє контейнери та видаляє їх разом з тимчасовими об’ємами.

Ізоляція ресурсів між паралельними завданнями забезпечується на рівні Docker. Кожне завдання отримує власну ізольовану мережу docker network, у якій бачить лише власні контейнери; обмеження пам’яті та CPU задаються через прапорці docker run; доступ до GPU обмежується через NVIDIA Container Toolkit з вибірковим прокидуванням пристроїв. Платформа уникає накладних витрат на встановлення та підтримку Kubernetes-кластера на машині постачальника, що особливо важливо для невеликих учасників пулу. Перенесення оркестрації на Kubernetes винесено у перспективи розвитку — воно стане доцільним за умови накопичення критичної маси постачальників, які вже мають K8s-інфраструктуру.

2.3. Протоколи міжсервісної комунікації: gRPC та асинхронний брокер Redis

Вибір протоколу комунікації між компонентами платформи визначається характером кожної взаємодії. Для синхронної типізованої комунікації з підтримкою потокової передачі обрано gRPC; для взаємодії з вузлами, де ключовою є стійкість до тимчасової недоступності, обрано асинхронний брокер повідомлень.

gRPC між центральним бекендом та диспетчером використовується для операцій управління контейнерами — провізіонування сервера агрегації, подача завдання, моніторинг статусу, перепинання, завершення — які за своєю природою є синхронними: бекенд має очікувати підтвердження від диспетчера перед переходом завдання у наступний стан. gRPC [19] поверх HTTP/2 з Protocol Buffers забезпечує сувору типізацію, яка перевіряється під час компіляції, а не під час виконання, двоспрямовану потокову передачу для асинхронних сповіщень про зміни стану та низькі накладні витрати на серіалізацію порівняно з REST на базі JSON. Контракт gRPC-API диспетчера визначається у файлі dispatcher.proto і компілюється у TypeScript-заглушку через @grpc/proto-loader на стороні центрального бекенду та у Rust-структури через prost на стороні

диспетчера. Будь-яка зміна контракту, несумісна з клієнтом, виявляється під час компіляції.

Асинхронний брокер повідомлень реалізовано через Redis [17] у режимі Pub/Sub з резервним списком повідомлень для гарантії доставки. Незалежні постачальники не контролюються оператором і можуть тимчасово втрачати мережеве з'єднання; синхронний протокол передавав би такі збої центральному бекенду як помилки. Брокер повідомлень відокремлює доставку команд від доступності вузла: команди ставляться в чергу та доставляються після відновлення з'єднання з гарантією щонайменше одноразової доставки. Вузли публікують сигнали присутності та оновлення прогресу тренування у відповідні канали, на які підписаний центральний бекенд, а також самостійно підписуються на канал команд для свого ідентифікатора. У цій моделі вузол виступає активним учасником, який сам забирає роботу з черги, а не пасивним отримувачем команд від оператора — модель, яка природно узгоджується з автономною природою постачальника обладнання.

Вибір Redis замість альтернатив RabbitMQ, NATS та Apache Kafka обґрунтований кількома критеріями. Операційна простота Redis значуща: один процес, одна залежність, мінімальна конфігурація проти RabbitMQ Erlang VM або Kafka ZooKeeper/KRaft. Redis поєднує Pub/Sub з персистентними структурами даних — Redis Streams забезпечує гарантовану доставку зі збереженням повідомлень на диск, а Sets та Sorted Sets дозволяють реалізувати додаткові структури реєстрів вузлів та черг завдань у тій самій системі без додавання компонентів. Клієнтські бібліотеки доступні для всіх використовуваних мов (TypeScript, Python, Rust) з активною підтримкою. Компромісом є те, що Redis не забезпечує транзакцій з гарантією рівно одноразової доставки без додаткових шаблонів; для критичних транзакцій у платформі ця гарантія забезпечується через PostgreSQL, тоді як Redis використовується для повідомлень з семантикою щонайменше одноразової доставки.

2.4. Робочий простір клієнта на основі об’єктного сховища Garage

Зберігання навчальних даних та артефактів тренування вимагає рішення, яке задовольняє двом обмеженням одночасно: ізоляція між клієнтами, через яку жоден клієнт не має доступу до даних іншого, та підтримка великих наборів даних обсягом десятків і сотень гігабайтів без обмежень розміру файлу, що накладають веб-інтерфейси завантаження.

У роботі реалізовано робочий простір клієнта на основі об’єктного сховища Garage — розподіленого S3-сумісного сховища з відкритим кодом, що підтримує горизонтальне масштабування та геореплікацію. Garage обрано через S3-сумісний API (boto3, AWS CLI), легкість самостійного хостингу та відкритий код. Порівняння з MinIO як основною альтернативою виконане на рівні документації та практики розгортання; кількісних бенчмарків пропускної здатності, затримки чи споживання пам’яті у межах роботи не виконано. Для малих файлів використовується пряме завантаження, для великих файлів (наборів даних, ваг моделей) — паралельне багаточастинне передавання.

Ізоляція між клієнтами забезпечується на рівні сховища: кожен користувач отримує власні S3-доступові ключі, які відображаються на роль із доступом виключно до префіксу свого простору у бакеті. Користувацький зберігач моделі (Persistor) у пакеті NVIDIA FLARE налаштований записувати контрольні точки безпосередньо до клієнтського робочого простору. Артефакти стають доступними клієнту через стандартні засоби читання об’єктного сховища відразу після завершення раунду без окремого кроку завантаження.

2.5. Модель загроз та межі ізоляції даних

Розкриття гарантій ізоляції даних розділено на два рівні захисту, кожен з яких має чітко визначений стан реалізації у поточній версії платформи. Таке структурування потрібне через те, що поняття «архітектурна ізоляція даних» у літературі з федеративного навчання вживається у різних значеннях, які важливо розмежувати.

Рівень L1 — ізоляція сирих навчальних даних (реалізовано). Сирі навчальні дані клієнта зберігаються виключно у його приватному просторі об’єктного сховища Garage S3, доступному за S3-ключами клієнта. GPU-вузли отримують від сервера агрегації лише ваги поточної глобальної моделі та навчальні скрипти, але не передачу сирих даних поза межі робочого простору. Шарди даних монтуються у контейнер клієнта навчання через bind-mount з робочого простору клієнта; постачальник GPU не має ні S3-ключів клієнта, ні структури повного шарду — він бачить лише той фрагмент, що монтується для конкретного завдання. Рівень гарантований структурою протоколу і не залежить від процедурних або контрактних механізмів.

Рівень L2 — захист переданих оновлень моделі (поза межами поточної реалізації). Між клієнтами навчання та сервером агрегації передаються лише оновлені ваги моделі. Geiping та співавтори [30] показали, що за певних умов навчальні приклади можуть бути частково реконструйовані з градієнтів через атаки інверсії — особливо для невеликих батчів та незахищених обмінів. Поточна реалізація платформи не захищає від таких атак: оновлення моделі передаються відкрито у межах внутрішньої мережі завдання. Для забезпечення рівня L2 необхідні диференційна приватність [31] та secure aggregation [26], інтегровані через точки розширення NVIDIA FLARE; ці механізми не реалізовані у поточній версії та винесені у перспективи розвитку (Розділ 6). Твердження про «архітектурну ізоляцію даних» у роботі стосується саме рівня L1 і не поширюється на захист від атак інверсії градієнтів.

Модель загроз поточної реалізації. Платформа призначена для сценаріїв, у яких клієнти платформи довіряють центральному бекенду як оператору сервісу; постачальники GPU є honest-but-curious — припускається, що вони не модифікують навчальний код, але можуть аналізувати спостережувані дані (ваги моделі, метадані); комерційно цінні є самі навчальні дані, а не модель. Поточна реалізація працює у експериментальному режимі без рівня автентифікації клієнтів — подача завдання вимагає лише доступу до локальної мережі платформи та валідних S3-ключів до власного робочого простору. Виробничий

режим потребує OIDC-інтеграції для автентифікації клієнтів, signed image manifests для цілісності контейнерів та шифрування артефактів на стороні клієнта. Ці механізми становлять технічне зобов'язання наступної ітерації і явно винесені за межі магістерської роботи.

Атаки, що не адресуються поточною реалізацією. Припущення honest-but-curious означає, що низка векторів атак з боку постачальника GPU не блокується ні технічно, ні процедурно. Постачальник з повним контролем над контейнером може зберегти копію змонтованого шарду даних до моменту його розмонтування, модифікувати навчальний код перед його виконанням, підробити метрики GPU для отримання додаткових завдань або зберегти проміжні ваги моделі між раундами для передачі третім особам. Платформа не верифікує цілісність виконання тренування і не виявляє відхилень такого типу. Пом'якшення цих ризиків потребує комбінації механізмів: signed image manifests для цілісності контейнерів, контроль виконання через довірене середовище (TEE), репутаційні системи постачальників на основі агрегованої статистики виконання. Реалізація цих механізмів виходить за межі поточної роботи; платформа у наявному вигляді придатна до сценаріїв з контрольним пулом постачальників, відомих оператору, але не до сценаріїв з повністю відкритим постачанням анонімних учасників.

Регуляторний контекст ізоляції L1. Ізоляція рівня L1 узгоджується з логікою сучасних регуляторних рамок щодо обробки чутливих даних: GDPR (Regulation 2016/679 EU) у статтях 44–50 обмежує транскордонну передачу персональних даних, HIPAA вимагає контролю доступу до медичних даних, регламенти DORA та NIS2 встановлюють вимоги до журналу подій та інцидентного реагування. Архітектура, у якій сирі дані не покидають клієнтського середовища, спрощує задачу задоволення цих вимог. Формальна сертифікація платформи за будь-якою з цих рамок не є предметом роботи: вона передбачає окремий процес — підписання договорів типу Business Associate Agreement (HIPAA), формальний аудит контролів безпеки, шифрування даних у спокої та при передачі, план реагування на інциденти, реалізацію рівня L2 для

повного захисту переданих оновлень. Згаданий приклад розгортання Massachusetts General Hospital через Rhino Health [2] демонструє, що NVIDIA FLARE використовується у HIPAA-регульованих сценаріях за умови виконання повного циклу організаційно-технічних заходів; з цього прикладу не випливає, що довільна платформа на базі NVIDIA FLARE автоматично є HIPAA-відповідною.

РОЗДІЛ 3. АГРЕГАЦІЯ ОБЧИСЛЮВАЛЬНИХ ПОТУЖНОСТЕЙ ВІД НЕЗАЛЕЖНИХ ПОСТАЧАЛЬНИКІВ

3.1. Автономна реєстрація та інтеграція GPU-вузлів

Мережа платформи будується знизу вверх: кожен постачальник GPU самостійно ініціює приєднання свого обладнання до мережі без участі оператора платформи у конфігурації хосту або у підтвердженні приєднання. Постачальник запускає Docker-контейнер вузла навчання, який містить NVIDIA FLARE-клієнт та супутні скрипти, на власній машині. Контейнер при першому запуску надсилає POST-запит до REST-ендпоінта /api/nodes центрального бекенду з характеристиками GPU — модель, обсяг відеопам’яті, версія драйвера — та мережевою адресою, доступною для серверів агрегації. Центральний бекенд записує вузол у реєстр у таблицю Worker та повертає ідентифікатор, що використовується вузлом для подальших звертань. Жодних додаткових кроків з боку оператора не потрібно — приєднання нового постачальника є самообслуговуваною операцією.

Після реєстрації вузол переходить у активний режим, у якому самостійно підтримує своє представлення у мережі. Він періодично надсилає сигнали присутності на ендпоінт /api/heartbeat з поточними метриками GPU — вільна відеопам’ять, відсоток завантаження, температура. Інтервал сигналів за замовчуванням становить 5 секунд; параметр налаштовується клієнтом-постачальником і визначає компроміс між реактивністю виявлення відмов та обсягом мережевого трафіку. Вузол стає доступним для призначення завдань після першого успішного сигналу присутності. Платформа не виконує окремого синтетичного бенчмарку при реєстрації — достовірність характеристик GPU перевіряється під час фактичного виконання першого тренувального завдання та відбивається у статистиці успішних завершень, що накопичується у реєстрі для майбутнього використання.

Окрім надсилання сигналів присутності вузол також підписаний на канал команд для свого ідентифікатора у Redis-брокері. Коли центральний бекенд після відбору формує команду на запуск контейнерів конкретного завдання, він публікує її у відповідний канал, і вузол отримує цю команду асинхронно, без потреби у вхідному мережевому з'єднанні з боку оператора. Така модель важлива для постачальників за NAT або у мережах без публічних IP-адрес: вузол ініціює всі з'єднання самостійно у вихідному напрямку. Зворотний напрямок зв'язку — від диспетчера на хості постачальника до центрального бекенду — також ініціюється з боку постачальника через gRPC-з'єднання до центрального бекенду.

3.2. Відбір вузлів за вільними ресурсами

У поточному режимі експлуатації відбір вузлів для нового завдання виконується спрощеним алгоритмом, який оптимізує доступність обчислювальних ресурсів. Центральний бекенд отримує специфікацію завдання з необхідною кількістю вузлів та мінімальним обсягом вільної відеопам'яті. З активного пулу зареєстрованих вузлів відбираються ті, які задовольняють трьом критеріям одночасно: сигнал присутності надійшов нещодавно у межах допустимого вікна, вузол не зайнятий іншим завданням, вільна відеопам'ять не нижча за поріг завдання. Серед кандидатів обираються перші N вузлів за зменшенням вільної відеопам'яті, де N — необхідна кількість, задана у завданні. Якщо доступних вузлів менше за необхідну кількість, завдання повертається у чергу з відкладеним повторним запуском відбору.

Алгоритм є детермінованим — для незмінного вхідного стану реєстру він повертає той самий результат, що важливо для відтворюваності експериментів. Обчислювальна складність становить O(N log N) за розміром пулу через сортування за вільною відеопам'яттю; для реалістичних розмірів пулу у десятки–сотні активних вузлів алгоритм виконується за мілісекунди. Накопичення статистики доступності та успішних завершень завдань на вузлі є технічним зобов'язанням, реалізація якого з включенням ваг надійності у формулу відбору

винесена у перспективи розвитку. Поточний підхід прийнятий як мінімально достатній для експериментальної оцінки платформи.

3.3. Підготовка навчального завдання клієнтом

Перед поданням завдання клієнт виконує одноразове налаштування — отримує S3-ключі доступу до власного робочого простору в об’єктному сховищі Garage. Робочий простір використовується для зберігання навчальних даних, скриптів та конфігураційних файлів.

Клієнт готує навчальне завдання у такій формі. Набір даних розбивається на K рівних шардів у форматі ZIP-архівів, де K — бажана кількість паралельних GPU-вузлів. Кожен шард має бути статистично репрезентативним для повного набору даних, тому рекомендується випадкове перемішування з фіксованим seed перед розбиттям — це уникає систематичного зсуву розподілу між вузлами. У випадку набору даних з природним поділом на класи, як Food-101 з 101 класом, кожен шард повинен містити приблизно n_k = n/K прикладів кожного класу. Це гарантує приблизно IID-розподіл між клієнтами та відповідає базовим припущенням алгоритму FedAvg.

Пакет NVIDIA FLARE включає файл визначення моделі model_def.py з функцією build_model() для повернення примірника моделі та необов’язкового завантаження попередньо натренованих ваг; користувацький виконавець клієнта з логікою локального тренування custom_client_executor.py, який успадковує клас Executor з API FLARE; користувацький зберігач моделі для збереження контрольних точок custom_persistor.py, який успадковує ModelPersistor; два конфігураційні файли — config_fed_server.json для сервера агрегації з параметрами num_rounds, min_clients та типом агрегатора, config_fed_client.json для кожного клієнта навчання з параметрами local_epochs, batch_size та шляхом до шарду. Рекомендована структура у робочому просторі містить каталог shards для шардів даних, каталог scripts для файлів Python та каталог configs для конфігураційних файлів FLARE.

3.4. Подання завдання та цикл виконання

Завдання подається через клієнтський SDK на Python або REST API. Клієнт вказує шляхи до каталогів шардів, скриптів та конфігурацій у робочому просторі та необов'язковий параметр memory_hint, що визначає мінімальний бажаний обсяг відеопам'яті на вузол у мегабайтах. Після подання центральний бекенд виконує валідацію — перевіряє існування всіх вказаних шляхів та цілісність кожного шарду як коректного ZIP-архіву через перевірку хедерів та цілісності CRC. У разі успішної валідації завдання передається у чергу для відбору вузлів та подальшого провізіонування через диспетчер.

Після запуску клієнт спостерігає за прогресом через REST-ендпоінти моніторингу — поточний номер раунду, статус кожного клієнта навчання та значення функції втрат після кожного раунду агрегації, якщо налаштовано користувацьким зберігачем моделі. Контрольні точки та фінальні ваги моделі записуються зберігачем безпосередньо до робочого простору та стають негайно доступними клієнту через стандартні засоби читання об'єктного сховища.

3.5. Інтерфейси клієнта

Платформа надає клієнту дві точки входу для подання та моніторингу завдань: програмний інтерфейс на Python (submitter.Client SDK) для інтеграції з Jupyter-блокнотами та конвеєрами безперервної інтеграції, та утиліту командного рядка submit-job, що є тонким обгортачем над SDK для скриптових робочих процесів. Обидва інтерфейси користуються одним і тим самим REST API центрального бекенду, що гарантує функціональну тотожність. REST API також документований і доступний для прямого використання сторонніми клієнтами та інтеграції з зовнішніми MLOps-платформами.

РОЗДІЛ 4. АРХІТЕКТУРА СИСТЕМИ

4.1. Загальна архітектура та принципи проектування

Платформа побудована з трьох ізольованих компонентів зі строго специфікованими протоколами міжкомпонентної взаємодії: центрального бекенду platform-api на TypeScript з фреймворком NestJS, диспетчера flare-dispatcher на Rust з gRPC-сервером tonic та контейнерного образу вузла навчання на основі NVIDIA FLARE-клієнта. Розподіл відповідальності між компонентами випливає з природи виконуваних ними задач: центральний бекенд володіє станом платформи, реєстром вузлів і клієнтським API; диспетчер володіє низькорівневим керуванням контейнерами NVIDIA FLARE та інтеграцією з його Admin API; вузол навчання, що працює на машині незалежного постачальника, виконує локальне тренування на призначеному шарді даних.

Принцип розділення відповідальностей реалізовано через те, що кожен компонент інкапсульований в окремому розгортанні з власним API — TypeScript-сервіс із REST та gRPC-клієнтом, Rust-бінарник із gRPC-сервером та контролем Docker-сокета, Python-контейнер із NVIDIA FLARE. Принцип ізоляції відмов означає, що відмова диспетчера на одному GPU-хості не впливає на диспетчерів на інших хостах та на стан інших навчальних завдань — кожне завдання утворює окремий ланцюжок контейнерів зі своєю мережею Docker. Принцип спостережуваності реалізовано через те, що кожен перехід стану завдання записується з міткою часу, ідентифікатором компонента-ініціатора та контекстними метаданими, а метрики агрегуються у Redis-кеші та доступні через REST.

Принцип різноманіття протоколів полягає у використанні gRPC для синхронних операцій управління контейнерами між центральним бекендом та диспетчером, асинхронного брокера Redis для команд вузлам та сповіщень про події платформи, об’єктного сховища Garage для доступу клієнтів до робочого простору. Вибір протоколу для кожної взаємодії оптимізує локальні

характеристики — затримку, типобезпеку, толерантність до тимчасової недоступності.

Окремий принцип, релевантний саме для мережі незалежних постачальників, полягає у тому, що рівень керування платформою (platform-api, диспетчер) ніколи не виступає ініціатором мережевого підключення до вузла навчання. Усі підключення завжди починаються з боку вузла: вузол сам реєструється через REST, сам надсилає сигнали присутності, сам підписаний на Redis-канал команд і сам встановлює FL-підключення до сервера агрегації після отримання сповіщення про нове завдання. Спрямованість зв'язку відповідає реальним умовам розгортання, де GPU-машини постачальників типово перебувають без вхідного доступу ззовні, і дозволяє приєднувати до мережі обладнання з будь-якого мережевого розташування без додаткової конфігурації на боці постачальника.

FL-підмережа окремого навчального завдання працює атомарно. Platform-api координує лише створення підмережі — відбирає вузли, передає диспетчеру команду на провізіонування контейнерів, реєструє ідентифікатор кластера. Після того як сервер агрегації NVIDIA FLARE та клієнти на відібраних вузлах підключилися один до одного, FL-підмережа функціонує як замкнена самодостатня одиниця: раунди агрегації, передача оновлень ваг, запис контрольних точок у клієнтський робочий простір Garage S3 виконуються виключно у межах цієї підмережі через FL-протокол та прямі S3-з’єднання, без зворотного звертання до platform-api. Відмова platform-api у середині завдання не перериває жодний активний раунд: сервер агрегації продовжує приймати оновлення від клієнтів, FedAvg-агрегація завершується, контрольна точка записується у Garage S3, починається наступний раунд. Centralized control plane задіяний лише у двох точках життєвого циклу — створенні завдання і звітуванні про його завершення. Розгортання platform-api у режимі active-passive HA-кластера, винесене у перспективи розвитку, скоротить вікно недоступності для прийому нових завдань без впливу на вже запущені.

Свідомий вибір на користь полімовної архітектури — TypeScript для центрального бекенду та Rust для диспетчера — обґрунтований конкретними технічними міркуваннями. TypeScript з NestJS забезпечує продуктивну розробку шару бізнес-логіки та REST API завдяки розвиненій екосистемі, типобезпечному ORM Prisma та модульній архітектурі з ін'єкцією залежностей. Rust для диспетчера дає інтеграцію з NVIDIA FLARE Admin API напряму через pyo3-міст до Python без міжпроцесного RPC, передбачуване низькозатримкове керування контейнерами через Docker socket з передбачуваним обсягом пам’яті без накладних витрат рантайму збору сміття та поширення диспетчера як одного Rust-бінарника, що звертається до Python через pyo3. Python-рантайм, NVIDIA FLARE та супутні залежності (numpy, torch) поставляються в окремому Docker-образі диспетчера, а не лінкуються статично у бінарник — повноцінний CPython-рантайм з нативними розширеннями не сумісний зі статичним лінкуванням; цей дизайн зберігає простоту розгортання у вигляді одного контейнера на хост постачальника.

Полімовна архітектура має вартість, яку важливо чесно зазначити. Поєднання TypeScript, Rust та Python у межах однієї системи дає три окремі системи збірки (npm, cargo, pip), три набори залежностей з різними циклами оновлення, три моделі обробки помилок (Promise/exception у TS, Result/panic у Rust, exception у Python) та три набори інструментів налагодження. Pyo3-міст між Rust і Python вносить специфічний клас проблем — Python-виключення потребують явного перетворення на Rust-Result, керування глобальним блокуванням інтерпретатора (GIL) обмежує паралельність викликів до Python-частини, прив'язка до конкретної версії CPython фіксується під час компіляції pyo3-крейту. Ці витрати компенсуються виграшем від інтеграції з NVIDIA FLARE Admin API в межах одного процесу без RPC, але збільшують поріг входу для розробників, які працюють з системою. Платформа адресує ці витрати через контейнеризацію кожного компонента — версії середовища фіксуються у відповідному Docker-образі — та чіткі межі gRPC-контракту між TypeScript та Rust, які ізолюють зміни у одній мові від іншої.

4.2. Центральний бекенд platform-api

Центральний бекенд platform-api реалізований на TypeScript з фреймворком NestJS [18] і виконує функцію єдиної точки входу для клієнтів через REST API, а також інкапсулює всю логіку реєстру вузлів, обробки сигналів присутності, відбору вузлів та керування станом завдань. На відміну від поширеного у літературі шаблону «центральний бекенд плюс окремий сервіс-координатор», функції координатора реалізовано у вигляді модулів NestJS у складі platform-api — рішення прийнято з міркувань простоти розгортання та зменшення кількості мережевих хопів у експериментальному режимі експлуатації; виокремлення координатора у окремий процес стане доцільним з ростом навантаження і винесено у перспективи розвитку.

Модульна структура platform-api охоплює AppConfigModule з централізованою конфігурацією; PrismaModule з типобезпечним ORM-шаром до PostgreSQL; CacheModule як обгортку над Redis для кешування часто запитуваних даних; BrokerModule для асинхронного обміну повідомленнями через Redis; GrpcModule як інфраструктурну обгортку gRPC-клієнта до диспетчера; CentralStorageModule для роботи з об’єктним сховищем Garage; TasksModule як основний модуль керування завданнями; NodesModule з реєстром вузлів та REST-ендпоінтами реєстрації; HeartbeatModule з обробником сигналів присутності та логікою виявлення офлайн-вузлів; ClusterStateModule з агрегованим представленням стану навчальних кластерів.

Стан-машина навчального завдання визначає послідовність переходів між станами та реалізована у TasksModule. Множина станів визначена у Prisma-схемі як перерахунок TaskStatus з п’ятьма значеннями. QUEUED — завдання прийнято і знаходиться у черзі на відбір вузлів та провізіонування. RUNNING — провізіонування завершено, на сервері агрегації виконуються раунди навчання. COMPLETED — завдання успішно завершилось і фінальні ваги моделі доступні у робочому просторі клієнта. FAILED — завдання завершилось з помилкою, причина зафіксована у журналі. CANCELLED — клієнт скасував завдання до природного завершення. Переходи між станами виконуються контрольованими

операціями у TasksService з фіксацією часу та контексту у відповідних таблицях БД.

Початковий перехід SUBMITTED→QUEUED виконується автоматично після успішної валідації вхідних даних. Перехід QUEUED→RUNNING ініціюється фоновим обробником черги після успішного відбору вузлів та провізіонування контейнерів NVIDIA FLARE через диспетчер. Перехід RUNNING→COMPLETED виконується після того, як сервер агрегації повідомив про завершення останнього раунду і всі контрольні точки записано у робочий простір. Переходи у термінальні стани FAILED та CANCELLED виконуються відповідно при виникненні помилки або за командою клієнта через PATCH /api/tasks/:id/status.

Контракт REST API центрального бекенду охоплює основні маршрути для роботи з завданнями та вузлами. POST /api/tasks приймає нове завдання та повертає його ідентифікатор. POST /api/tasks/_search повертає список завдань клієнта з фільтрацією за станом та часом. PATCH /api/tasks/:id/status дозволяє клієнту скасувати активне завдання. GET /api/tasks/:id/model повертає посилання на фінальні ваги моделі у робочому просторі. Окрема група маршрутів /api/nodes та /api/heartbeat обслуговує реєстрацію вузлів та прийом їхніх періодичних сигналів. Конкретний перелік маршрутів задокументовано через OpenAPI-схему, що генерується автоматично з декораторів NestJS.

gRPC-клієнт до диспетчера, реалізований у GrpcModule, генерується з контракту dispatcher.proto під час складання застосунку через @grpc/proto-loader. Типи відповідей та запитів стають доступними у TypeScript-коді як інтерфейси, які компілюються разом з рештою сервісу і виявляють невідповідності контракту під час компіляції, а не виконання.

4.3. Алгоритм відбору вузлів для навчального завдання

Алгоритм відбору вузлів реалізовано як метод WorkerSelectionService.selectWorkers у NestJS-модулі ClusterStateModule. У поточному режимі експлуатації алгоритм оптимізує єдиний критерій — доступну вільну відеопам’ять на вузлі — після відсіювання вузлів, які не задовольняють мінімальним вимогам завдання.

Алгоритм оперує специфікацією завдання, яка задає необхідну кількість вузлів та мінімальний обсяг вільної відеопам’яті, та поточним станом реєстру вузлів. На першому етапі з реєстру відбираються лише ті вузли, для яких одночасно виконуються три умови: вузол перебуває у стані CONNECTED (отримано свіжий сигнал присутності у межах допустимого вікна), вузол не зайнятий виконанням іншого завдання, повідомлена вільна відеопам’ять не нижча за мінімальний поріг, заданий завданням. Якщо після фільтрації кількість придатних вузлів менша за потрібну, алгоритм повертає помилку INSUFFICIENT_NODES, і завдання залишається у черзі до наступної спроби відбору. На другому етапі придатні вузли впорядковуються за спадаючим значенням вільної відеопам’яті — проста евристика, що надає перевагу вузлам з найбільшим запасом ресурсу і зменшує ймовірність нестачі пам’яті у середині раунду. На третьому етапі з впорядкованого списку обираються перші N вузлів, де N — кількість, задана у завданні.

Складність алгоритму визначається сортуванням і становить O(N log N) за розміром пулу; крок фільтрації виконується за O(N), фінальний відбір — за O(K), де K — потрібна кількість вузлів. Це інженерна базова лінія, яка за реалістичних розмірів пулу (десятки–сотні активних вузлів у поточному режимі експлуатації) виконується за мілісекунди. Алгоритм не претендує на алгоритмічну новизну: емпірично перевірено його роботу на пулах до 8 активних вузлів; масштабування до більших пулів випливає з обчислювальної складності, але не є предметом окремої експериментальної перевірки у межах роботи.

Алгоритм є детермінованим — для незмінного вхідного стану реєстру він повертає той самий результат, що важливо для відтворюваності експериментів. Розширення алгоритму композитною оцінкою з урахуванням накопиченої

статистики надійності вузла, поточного відсотка завантаження GPU та класу обладнання заплановано як технічне зобов’язання наступної ітерації платформи; формальна модель такого алгоритму та аналіз чутливості до ваг компонентів є природною темою окремого дослідження, що виходить за межі роботи.

4.4. Диспетчер flare-dispatcher

Диспетчер flare-dispatcher реалізований як Rust-сервіс з gRPC-сервером на основі tonic та зв'язками з Python через pyo3. Архітектурний вибір Rust замість продовження TypeScript-стеку випливає з природи задач диспетчера. NVIDIA FLARE Admin API наданий як Python-бібліотека, тому інтеграція з нею у TypeScript-сервісі вимагала б міжпроцесного RPC через окремий Python-процес — додаткового джерела затримок та помилок. Pyo3-міст у Rust-коді натомість дозволяє викликати функції цієї бібліотеки безпосередньо у межах одного процесу. Керування контейнерами через Docker socket у Rust має передбачуваний обсяг пам’яті та виконується без накладних витрат рантайму збору сміття, що важливо при паралельному обслуговуванні десятків активних завдань. Диспетчер поширюється як Rust-бінарник зі складеним pyo3-мостом у складі Docker-образу, що містить Python-рантайм та залежності NVIDIA FLARE; розгортання на хості постачальника зводиться до запуску одного контейнера.

Зовнішній контракт диспетчера визначений у protobuf-файлі dispatcher.proto і охоплює п’ятнадцять gRPC-операцій, які обслуговують повний життєвий цикл навчальних завдань. Серед них основні: ProvisionFlareServer — запуск контейнера сервера агрегації NVIDIA FLARE для нового завдання; UploadDataset — підвантаження шарду даних у робочий простір вузла; SubmitJob — подача навчального завдання серверу агрегації через NVIDIA FLARE Admin API; MonitorJob — потокове постачання оновлень стану завдання; AbortJob — переривання активного завдання; DownloadJobResult — отримання фінальних артефактів. Допоміжні операції GetSystemInfo, CheckStatus, ListJobs, GetJobMeta, ListJobComponents, DeleteJob, Shutdown, ShutdownSystem та ShowStats підтримують моніторинг та керування.

Внутрішня структура диспетчера ділиться на два модулі. Модуль platform/docker_utils.rs відповідає за керування контейнерами: формує аргументи для виклику docker run з потрібним набором прапорців (відображення пристроїв GPU через NVIDIA Container Toolkit, монтування шарду даних, мережева конфігурація), запускає контейнер та відстежує його стан. Модуль flare_admin/ містить pyo3-міст до NVIDIA FLARE Admin API і інкапсулює подачу специфікації завдання, опитування стану раундів та отримання логів сервера агрегації. Розділення між цими двома модулями забезпечує можливість незалежної заміни кожного шару — наприклад, перехід з Docker socket на Kubernetes API торкнеться лише модуля керування контейнерами.

Послідовність виконання ProvisionFlareServer складається з кількох кроків. Створюється виокремлена Docker-мережа для нового завдання, що ізолює внутрішній трафік сервера агрегації та клієнтів від інших активних завдань. Запускається контейнер сервера агрегації NVIDIA FLARE з відкритим адміністративним портом, прив’язаним до завдання. Очікується готовність сервера через опитування health-ендпоінта, після чого диспетчер повертає клієнту gRPC-відповідь з ідентифікатором кластера та адресою сервера. Виклики SubmitJob, MonitorJob та AbortJob, що надходять пізніше у межах життєвого циклу завдання, працюють з цим ідентифікатором кластера для маршрутизації до відповідного сервера агрегації.

4.5. Образ вузла навчання

Образ вузла навчання являє собою Docker-контейнер на основі офіційного образу pytorch/pytorch з підтримкою CUDA, до якого додано NVIDIA FLARE-клієнт та допоміжні скрипти запуску. Контейнер тримається активним з допомогою скрипту-точки входу start_client.sh, який утримує процес у режимі очікування команд від сервера агрегації NVIDIA FLARE. Шард даних монтується у контейнер через bind-mount з робочого простору клієнта; ваги моделі надходять до контейнера від сервера агрегації через FL-протокол на

початку кожного раунду; локальне тренування виконується кодом custom_client_executor.py, що завантажується разом з пакетом завдання.

Поряд з контейнером вузла навчання на машині постачальника працює супутній агент-реєстратор, який не входить до самого FL-образу: він виконує реєстрацію вузла у центральному бекенді через REST, утримує цикл сигналів присутності та підписку на Redis-канал команд для цього вузла. Розподіл відповідальності спрощує сам тренувальний образ, робить його сумісним зі стандартними інструментами розгортання NVIDIA FLARE та дозволяє замінювати реєстраційний рівень незалежно від базового тренувального шару. Об'єднання реєстраційного агента з тренувальним контейнером в єдиний образ є технічним зобов’язанням наступної ітерації продукту, що зменшить кількість компонентів, які постачальнику потрібно розгорнути на своїй машині, до одного.

РОЗДІЛ 5. РОЗРОБКА СИСТЕМИ

5.1. Технологічний стек

Технологічний стек обрано на основі вимог щодо типобезпеки, продуктивності, відмовостійкості та зрілості екосистеми кожного компонента.

TypeScript та NestJS використовуються для центрального бекенду platform-api. NestJS забезпечує модульну архітектуру з вбудованою ін'єкцією залежностей, що спрощує тестування окремих компонентів. Декораторний синтаксис для маршрутизації та підписки на повідомлення брокера зменшує обсяг шаблонного коду. У роботі використано NestJS на базі Node.js 20 LTS з TypeScript 5.x та Prisma як типобезпечним ORM до PostgreSQL.

Rust та tonic використовуються для диспетчера flare-dispatcher. Tonic надає реалізацію gRPC-сервера на основі асинхронного рантайму Tokio. Prost виконує генерацію Rust-структур з protobuf-контракту. Pyo3 забезпечує виклик функцій NVIDIA FLARE Admin API напряму з Rust-коду без міжпроцесного RPC. Бібліотека garage-sdk використовується для роботи з об’єктним сховищем Garage S3 з боку диспетчера, коли потрібен прямий доступ до шардів даних або артефактів моделі.

NVIDIA FLARE використовується як фреймворк оркестрації тренування. Вибір обґрунтований виробничою зрілістю з прецедентами у медичних та фінансових розгортаннях, наявністю офіційного Docker-образу з коректно зафіксованими версіями CUDA та cuDNN, розширюваністю через CustomPersistor та CustomExecutor без модифікації ядра фреймворку, а також вбудованою частковою агрегацією при втраті клієнтів у середині раунду. У роботі використано NVIDIA FLARE 2.x; рекомендується використовувати закріплену версію у виробничому розгортанні.

Docker забезпечує стандартизацію середовища виконання через контейнеризацію NVIDIA FLARE-сервера та клієнтів навчання, що гарантує відтворюваність незалежно від конфігурації хост-машини. Платформа взаємодіє

з Docker Engine API через локальний сокет /var/run/docker.sock з боку диспетчера.

Redis виступає асинхронним брокером повідомлень та кешем. Поєднання pub/sub каналів для сигналів присутності та оновлень стану кластера з персистентними структурами дає можливість зберігати швидко-доступний агрегований стан для REST-API без частих звертань до PostgreSQL.

PostgreSQL виконує роль основного сховища стану — реляційна модель з міграціями через Prisma забезпечує транзакційну цілісність при паралельних оновленнях стану завдань та реєстру вузлів. Garage S3 виступає об’єктним сховищем для шардів даних та артефактів моделі і взаємодіє з рештою компонентів через стандартний S3-протокол через бібліотеки boto3 на стороні Python та garage-sdk на стороні Rust.

5.2. Реалізація центрального бекенду

TasksModule реалізує стан-машину завдання через TasksService та TasksController. Маршрут POST /api/tasks приймає нове завдання, валідує посилання на робочий простір клієнта, зберігає завдання зі станом QUEUED та публікує подію task.queued у Redis-брокер для подальшої обробки фоновим обробником черги. Маршрут POST /api/tasks/_search повертає список завдань поточного клієнта з фільтрацією за статусом і часовим діапазоном; список побудовано через Prisma-запит з пагінацією. Маршрут PATCH /api/tasks/:id/status дозволяє клієнту скасувати активне завдання — при переході у CANCELLED викликається DispatcherClient.abortJob(taskId) для коректного завершення контейнерів на боці диспетчера. Маршрут GET /api/tasks/:id/model повертає попередньо підписане посилання на фінальні ваги моделі у робочому просторі клієнта.

NodesModule та HeartbeatModule разом реалізують підтримку реєстру вузлів. POST /api/nodes приймає реєстраційний запит від нового вузла з характеристиками GPU та зберігає запис у таблиці Worker через Prisma. POST /api/heartbeat приймає періодичні сигнали присутності з метриками GPU та

оновлює відповідний запис у реєстрі. ClusterStateModule агрегує поточний стан вузлів та активних завдань для швидкого доступу алгоритму відбору вузлів та для відображення у моніторингових інтерфейсах.

gRPC-клієнт у GrpcModule генерується NestJS з dispatcher.proto-контракту; клієнтська заглушка ProvisionFlareServer(request) відповідає серверній операції диспетчера. Невідповідність типів між клієнтом та сервером виявляється під час компіляції TypeScript і не може потрапити у виконання.

5.3. Реалізація диспетчера

Диспетчер flare-dispatcher складається з gRPC-сервера на основі tonic, модуля керування контейнерами через Docker socket та модуля інтеграції з NVIDIA FLARE Admin API через pyo3. Точка входу dispatcher-server.rs ініціалізує gRPC-сервер, прослуховує налаштований порт та маршрутизує вхідні запити до відповідних обробників сервісу FlareDispatcher.

Обробник gRPC ProvisionFlareServer виконує послідовність операцій. Формується унікальний ідентифікатор кластера на основі ідентифікатора завдання. Створюється виокремлена Docker-мережа docker network create з ідентифікатором кластера. Виконується docker run для контейнера сервера агрегації NVIDIA FLARE з прив'язкою до створеної мережі та відкритими адміністративним і FL-портами. Очікується готовність сервера через цикл опитування health-ендпоінта з тайм-аутом. У разі помилки на будь-якому з цих кроків виконується відкат — зупинка та видалення створених ресурсів.

Обробник SubmitJob приймає специфікацію завдання, передає її серверу агрегації через NVIDIA FLARE Admin API за допомогою pyo3-моста до Python-бібліотеки nvflare. Обробник MonitorJob є серверним потоком gRPC — він підписується на оновлення стану раундів від сервера агрегації та транслює їх клієнту у формі стрімінгових відповідей. AbortJob викликає відповідний Admin API метод для коректного припинення тренування. DeleteJob видаляє контейнери та Docker-мережу завдання після його завершення.

5.4. Образ вузла навчання та NVIDIA FLARE-пакет

Образ вузла навчання збирається з Dockerfile, що базується на pytorch/pytorch з підтримкою CUDA 12.x та cuDNN 9.x. Додано встановлення NVIDIA FLARE через pip, копіювання навчальних скриптів та скрипт точки входу start_client.sh. Контейнер запускається з прокидуванням GPU через прапорці --gpus та з монтуванням шарду даних через --volume.

Пакет NVIDIA FLARE, який надає клієнт у складі завдання, містить такі компоненти. Файл custom_persistor.py успадковує ModelPersistor та реалізує методи save_model(model_learnable, persist_dir) та load_model(persist_dir). У платформі persist_dir вказує на клієнтський робочий простір у об’єктному сховищі Garage, тому контрольні точки записуються безпосередньо у простір клієнта та стають доступними негайно після кожного раунду. Файл custom_client_executor.py успадковує Executor та реалізує метод execute(task_name, shareable) з логікою навчання, специфічною для моделі: завантаження ваг з вхідного Shareable, виконання заданої кількості локальних епох, повернення оновлених ваг до сервера агрегації.

Конфігураційні файли пакету визначають параметри тренувального процесу. config_fed_server.json задає кількість раундів num_rounds, мінімальну кількість клієнтів для агрегації min_clients та агрегаційний робочий процес — scatter_and_gather_workflow з FedAvg як стандартним вибором. config_fed_client.json задає кількість локальних епох local_epochs, розмір батча batch_size та шлях до шарду даних, призначеного цьому клієнту.

5.5. Підхід до тестування

На поточному етапі розробки тестування платформи здійснено через наскрізне виконання реальних навчальних завдань — основні сценарії, описані у Розділі 6 (тренування ViT-B/16 на Food-101 з повторюваністю на двох поколіннях GPU), валідовано безпосередньо через прогін на цільовому обладнанні з фіксацією логів та метрик. Підхід виправданий для

експериментальної реалізації, де основною метою є демонстрація функціональної коректності повного циклу та емпіричних характеристик платформи.

Розширення тестового покриття автоматизованими юніт-тестами для критичних компонентів — насамперед методу WorkerSelectionService.selectWorkers та обробників життєвого циклу завдань у TasksService — заплановано як технічне зобов'язання наступної ітерації. Інтеграційне тестування потоку «подача завдання → провізіонування → виконання → отримання артефактів» з використанням mock-реалізації Docker API є природним наступним кроком, який не вимагає змін архітектури і реалізується інкрементально через стандартний інструментарій Jest для NestJS та cargo test для Rust-компонентів.

РОЗДІЛ 6. ЕКСПЕРИМЕНТАЛЬНА ОЦІНКА

6.1. Постановка експерименту

Експериментальна оцінка платформи виконана на репрезентативному виробничому навантаженні — дотренуванні моделі ViT-B/16 [22] обсягом 86 мільйонів параметрів з бекбоном vit_base_patch16_224.augreg_in21k з бібліотеки timm [20] на наборі даних Food-101 [21] з 75 750 тренувальних та 25 250 валідаційних зображень у 101 класі формату 224×224. Вибір моделі та набору обґрунтований тим, що ViT є архітектурою-репрезентантом сучасних трансформерних моделей з реалістичним обсягом параметрів для дотренування, Food-101 є стандартним бенчмарком для класифікації зображень, а набір даних достатньо великий, щоб 4-стороннє шардування зберігало статистичну репрезентативність кожного шарду.

Базовий рецепт експерименту, позначений далі як рецепт v2, використовує batch=128, lr=2e-4, 5 раундів × 1 локальна епоха на 4 клієнтах через NVIDIA FLARE з FedAvg. Загальна кількість SGD-кроків оптимізатора у FL-режимі дорівнює 4 × 5 × 148 = 2960; централізоване виконання того ж рецепту (1 GPU, batch=128, lr=2e-4, 5 епох) дає такі ж 2960 SGD-кроків, що забезпечує коректну основу для порівняння FL та централізованого виконання за однакового обчислювального бюджету. Випадковий seed зафіксовано на значенні 1337 для першого повтору; для побудови статистичних оцінок виконано додаткові повтори з seed 2024, 4096, 8192. Валідація здійснюється через скрипт validate_classify.py, який відтворює стандартний рецепт оцінки ViT: препроцесинг через AutoImageProcessor("google/vit-base-patch16-224-in21k"), середня кросентропія на приклад, метрики top-1 та top-5.

Зовнішня точка орієнтації проти comparand-а статистичного тесту. Як зовнішня точка орієнтації використовується опублікована модель nateraw/food з репозиторію Hugging Face, що повідомляє top-1 89.13% та val_loss 0.4501 на тому ж наборі даних. Спроби відтворити це значення у власному централізованому прогоні на H100 з тим самим рецептом дали середній top-1 86.24%, нижче на ~2.9

п.п. Причина розриву найбільш імовірно лежить у не повністю задокументованій конфігурації nateraw/food: між версіями timm 0.6.x та 0.9.x змінювалися дефолтні аугментації та реалізація initializer-ів, а конкретна версія timm для опублікованої моделі не вказана. HF-значення не використовується як comparand статистичного тесту — порівняння виконується між власними прогонами FL та централізованими прогонами з ідентичним кодом та рецептом. HF-значення відображено у таблицях як зовнішня орієнтація для розуміння абсолютного рівня точності, досяжного на цій задачі.

6.2. Конфігурації та зведення запусків

Експериментальна серія включає чотири повтори рецепту v2 у FL-конфігурації, чотири повтори централізованого виконання того ж рецепту, серію аналізу внеску модифікацій рецепту та одне інженерне спостереження зі збільшеним батчем. Усереднені результати наведено у Таблиці 6.1; повний перелік окремих прогонів обох порівнюваних конфігурацій подано у Додатку Б.

Таблиця 6.1 — Зведена статистика експериментальної серії

Запуск	Конфігурація	top-1	top-1 TTA	val_loss	Час
v1	sawtooth LR, fresh Adam	87.90%	—	0.5444	6m 26s
v2 (baseline) #1	continuous LR + persisted Adam	88.66%	88.87%	0.5125	6m 13s
v3	+ label smoothing + EMA, 10 раундів	87.13%	87.29%	0.7881	13m 21s
v4	batch 256, lr × √2, 10 раундів	78.75%	—	1.7251	15m 33s
v2 rerun #2	те саме що v2	88.83%	89.24%	0.5756	—
v2 rerun #3	те саме що v2	88.84%	89.17%	0.5695	—
v2 rerun #4 (4090)	те саме що v2, RTX 4090	88.73%	88.99%	0.5155	7m 08s
FL v2 mean (n=4)	усереднення повторів v2	88.77 ± 0.09	89.07 ± 0.16	0.543 ± 0.03	—

centralized v2 mean (n=4)	1×H100, той самий рецепт, 4 повтори	86.24% ± 0.16	87.11% ± 0.20	0.621 ± 0.01	≈9m 22s
v10 (інженерне спост.)	batch 348, 5 раундів (1100 кроків)	89.42%	89.70%	0.4971	8m 03s
HF reference (зовн.)	nateraw/food, не comparand	89.13%	—	0.4501	не оприлюднено

6.3. Порівняння FL та централізованого виконання того ж рецепту

Основне порівняння роботи отримане між федеративним виконанням рецепту v2 (4 клієнти × 5 раундів × 1 локальна епоха) та централізованим виконанням того ж рецепту на одному H100 (5 епох, той самий обчислювальний бюджет у 2960 SGD-кроків). Для побудови коректної бази порівняння виконано чотири повтори кожної конфігурації з різними значеннями seed (1337, 2024, 4096, 8192).

Середня точність FL-конфігурації становить 88.77% ± 0.09 п.п. у режимі vanilla та 89.07% ± 0.16 п.п. з TTA. Середня точність централізованої конфігурації — 86.24% ± 0.16 п.п. у vanilla та 87.11% ± 0.20 п.п. з TTA. Спостережена різниця складає приблизно +2.5 п.п. на користь FL у vanilla та +2.0 п.п. з TTA.

Двовибірковий t-тест Велча на чотирьох повторах кожної конфігурації дає p-значення < 0.001 для різниці у vanilla-режимі ($t \approx 27.6$, $df \approx 5$). Враховуючи малий розмір вибірок (n=4 з обох сторін), формальна статистична значущість має сприйматися як орієнтовна — основну вагу мають усереднені оцінки та довірчі інтервали, наведені у Додатку Б. Серія експериментів проведена за IID-розподілу даних між клієнтами через випадкове перемішування з фіксованим seed; узагальнення на non-IID не входить у межі роботи.

Природа спостереженого ефекту. Різниця пояснюється у значній мірі поведінкою централізованого прогону: train loss падає до приблизно 0.005 на п’ятій епосі, що свідчить про надмірну адаптацію до тренувального набору. Крок

усереднення FedAvg у FL-конфігурації при цьому згладжує локальні мінімуми, у які встигають увійти окремі клієнти за 148 кроків між раундами. Ефект сформульований як гіпотеза у роботі McMahan та співавторів [9] у 2017 році. Отримане спостереження є post-hoc поясненням результату — спочатку було отримано числа, потім знайдено механізм. Перевірка ефекту з контрольованою регуляризацією (early stopping, weight decay) у централізованому прогоні є природним продовженням цієї серії і винесена у перспективи розвитку (підрозділ 6.6).

6.4. Аналіз внеску модифікацій рецепту

Початковий «наївний порт» централізованого рецепту еталона до FedAvg у запуску v1 дав 87.90% top-1, що на 0.87 п.п. нижче за FL v2 та на 1.23 п.п. нижче за HF-точку орієнтації. Аналіз показав дві причини регресії: на кожному раунді FedAvg будувався новий LambdaLR-розклад зубчастого вигляду замість одноразового, а оптимізатор Adam отримував початковий стан, втрачаючи накопичені моменти першого та другого порядку. Дві модифікації, які закривають цей розрив, полягають у безперервному LR-розкладі через глобальний крок (current_round × steps_per_round + local_step) та персистентному per-client стані оптимізатора, який зберігається у каталог STATE_DIR/optimizer_state.pt у кінці кожного раунду та відновлюється на початку наступного. Запуск v2 дав 88.66% top-1 та 88.87% з TTA, що закрило 62% розриву з HF-орієнтацією.

У спробі покращити v2 додано label smoothing 0.1 та per-round EMA з коефіцієнтом 0.999. Запуск v3 показав регресію на 1.5 п.п. порівняно з v2. EMA з коефіцієнтом 0.999 на лише 148 локальних кроків раунду приписує близько 86% ваги початковим вагам раунду за формулою expected_weight = decay^n_steps = 0.999^148 ≈ 0.863; подача EMA-усереднених ваг назад до FedAvg означає, що лише приблизно 14% запланованого прогресу раунду доходить до глобальної моделі. Label smoothing з $\varepsilon = 0.1$ додатково спотворює виміряний val_loss, оскільки модель калібрується для м’яких таргетів $y\_smooth = (1-\varepsilon)\, y + \varepsilon/(C-1)$,

тоді як оцінка використовує жорстку CE; невідповідність калібрування призводить до завищеного val_loss без покращення точності. Запуск v4 з батчем 256, lr 2e-4 × √2 та 10 раундів дав ще гіршу регресію 78.75% — разом з EMA-проблемою додалось погіршення сигналу від подвоєння batch без правильного масштабування lr. Усі зміни поза рецептом v2 відкочено.

Рецепт v2 повторено чотири рази — тричі на двох різних інстанціях RTX PRO 4500 (запуски #1, #2, #3) та один раз на RTX 4090 (#4). Середній top-1 за чотирма запусками рецепту v2 становить 88.77% у конфігурації vanilla та 89.07% з TTA; стандартне відхилення між запусками — 0.09 п.п. та 0.16 п.п. відповідно. Час «трен+комм» на RTX PRO 4500 становить 6m 13s, на RTX 4090 — 7m 08s; різниця ~15% пояснюється нижчою пропускною спроможністю fp16 Tensor Cores архітектури Ada Lovelace порівняно з Blackwell на ViT-B/16 у батчі 128. Рецепт є апаратно-незалежним, відмінність полягає лише в загальному часі виконання.

Систематичне порівняння впливу кожної з п’яти випробуваних модифікацій рецепту наведено у Таблиці 6.2.

Таблиця 6.2 — Внесок модифікацій рецепту FL-навчання

Модифікація	**Δ top-1**	**Δ top-1 (TTA)**	**Δ val_loss**	**Висновок**
Sawtooth LR (v1) → continuous LR	+0.18 п.п.	—	−0.02	Покращує
Fresh Adam → persisted Adam	+0.58 п.п.	—	−0.01	Покращує (синергія з LR)
Vanilla → + label smoothing 0.1	−0.4 п.п.	—	+0.20	Регресія
Vanilla → + EMA decay 0.999	−1.5 п.п.	—	+0.25	Сильна регресія
Batch 128 → batch 348 (lr фіксований)	+0.65 п.п.	+0.63 п.п.	−0.05	Інженерне спост., 6.5

З розглянутих модифікацій лише безперервний LR разом зі збереженим станом Adam покращують результат у режимі коротких FL-раундів. EMA та label

smoothing призводять до регресії і потребують переналаштування для FL-режиму. Збільшення батча без масштабування LR розглядається окремо у наступному підрозділі як інженерне спостереження поза основним порівнянням.

6.5. Інженерне спостереження: запуск зі збільшеним батчем (v10)

Окремо від основного порівняння FL проти централізованого виконання, у роботі виконано одиничний запуск зі збільшеним батчем — v10 з batch=348 та lr=2e-4 без масштабування за linear scaling rule, 5 раундів. Результат становить top-1 89.42% / 89.70% TTA, найвищий показник у всій експериментальної серії. Запуск явно не належить до основного comparand-а статистичного порівняння з кількох причин.

Збільшення батча з 128 до 348 без масштабування швидкості навчання порушує умову однакового обчислювального бюджету. Загальна кількість SGD-кроків у v10 дорівнює 4 × 5 × 55 = 1100, на 63% менше за 2960 кроків базового рецепту v2 та централізованого еталону. Запуск є принципово іншою точкою у просторі конфігурацій, ніж базовий рецепт, тому пряме порівняння його точності з v2 чи централізованим прогоном не є коректним apples-to-apples порівнянням.

Спостережене покращення на +0.65 п.п. над v2 vanilla отримане в одиничному прогоні без повторів, тоді як v2 виконано у чотирьох повторах. Без повторів v10 його варіативність невідома, а конкретне значення 89.42% не має довірчого інтервалу. Результат суперечить класичному linear scaling rule Goyal та співавторів [25], який рекомендує масштабувати lr пропорційно до batch; відсутність такого масштабування у v10 могла дати випадково сприятливу комбінацію.

Запуск v10 розглядається як інженерне спостереження, що ілюструє можливість покращення рецепту на межі перенавчання від зміни обчислювального бюджету; сам факт виконання конфігурації зі збільшеним батчем на одній з апаратних платформ (RTX PRO 4500 з 32 ГБ) підтверджує гнучкість платформи. Жодне з тверджень новизни роботи не базується на результаті v10. Систематична оцінка ефекту батча у FL-режимі з належними

повторами та масштабуванням lr становить окремий експериментальний напрям поза межами поточної магістерської.

6.6. Межі застосовності отриманих результатів

Експериментальна оцінка роботи має конкретні межі застосовності, які важливо явно зафіксувати для коректної інтерпретації результатів та для майбутніх досліджень. Усі емпіричні результати отримані на одній моделі (ViT-B/16), одному наборі даних (Food-101), одному алгоритмі агрегації (FedAvg) та одній топології (4 клієнти, 5 раундів, 1 локальна епоха). Узагальнення на інші моделі, набори даних, агрегатори (FedProx, FedOpt, SCAFFOLD) та топології (2 або 8 клієнтів, більше раундів, інша кількість локальних епох) є природним продовженням роботи, але не виконано у межах поточного релізу.

Окремо не виміряно накладні витрати самої платформи (оркестрації, gRPC, Redis, провізіонування контейнерів) на тлі загального часу тренування — це обмеження не впливає на висновки про точність моделі, але впливає на оцінку економічної ефективності платформи у виробничому розгортанні.

Аналіз сукупної вартості володіння (TCO) у роботі не виконується. Мотивація у Вступі посилається на вартість хмарних GPU за публічними тарифами хмарних провайдерів, але кількісне порівняння повної вартості тренування ViT-B/16 на платформі з вартістю того самого тренування на AWS SageMaker або GCP Vertex AI потребує обліку витрат на PostgreSQL, Redis, Garage S3, мережу та координаційну роботу, які не задокументовані. Заявлена економічна привабливість платформи у поточному стані залишається на рівні архітектурної передумови, не доведеної конкретними числами.

Проблема відстаючих вузлів (straggler) задекларована у моделі загроз, але не вирішена у реалізації. Відбір вузлів за вільною відеопам’яттю не корелює зі швидкістю виконання раунду — у Таблиці 6.1 різниця часу між RTX PRO 4500 (Blackwell) та RTX 4090 (Ada Lovelace) на тому ж рецепті становить близько 15%. У сценарії з більшою топологією та різними поколіннями GPU весь раунд буде обмежений швидкістю найповільнішого учасника. Механізм часткової

агрегації NVIDIA FLARE через параметр min_clients може відкинути найповільніших клієнтів, але це різко змінює статистичні властивості агрегації і не є рішенням straggler-проблеми у строгому сенсі. Систематичне адресування цієї задачі потребує асинхронної агрегації або моделі профілів швидкості вузлів і є окремим напрямом подальшої роботи.

Відмовостійкість платформи у середині раунду тренування — реалізована через вбудований механізм часткової агрегації NVIDIA FLARE з параметром min_clients — не перевірена окремим експериментом з примусовою відмовою вузла. Поведінка платформи у цьому сценарії випливає з документації NVIDIA FLARE та архітектури платформи (диспетчер коректно обробляє відключений контейнер як завершений), але емпіричне підтвердження залишається технічним зобов'язанням наступної ітерації.

Відтворюваність експериментів забезпечується фіксованим набором seed для генераторів випадкових чисел Python, NumPy та PyTorch. Спостережена варіативність між чотирма повторами рецепту v2 у діапазоні 88.66–88.84% свідчить, що повна бітова відтворюваність не досягається — PyTorch на GPU містить недетерміновані операції (зокрема atomicAdd у backward pass), повне детермінування потребує torch.use_deterministic_algorithms(True) та змінної середовища CUBLAS_WORKSPACE_CONFIG=:4096:8, які у цій серії не застосовувалися. Фіксація seed забезпечує статистичну подібність прогонів, але не їх повну ідентичність.

Порівняння платформи з наявними рішеннями у Розділі 1.5 є якісним — воно базується на функціональних характеристиках кожного класу платформ і не містить кількісних метрик (час від подачі завдання до початку тренування, накладні витрати оркестрації, реальна вартість виконання типового завдання). Конкретне порівняння з Ray як альтернативною платформою оркестрації розподілених обчислень з гетерогенними вузлами у роботі не виконувалось і є природним продовженням аналізу. Усе це не зменшує цінності задокументованих архітектурних рішень, але обмежує заяви про «розрив у літературі» рівнем якісного спостереження.

Реалізована система за рівнем готовності відповідає proof-of-concept: відсутні автоматизовані юніт- та інтеграційні тести, не виконано стрес-тестування на масштабі, відсутній рівень автентифікації клієнтів. Календарний план роботи з 18 етапів покриває проектування та реалізацію, але виробнича готовність потребує окремої ітерації з фокусом на тестування, безпеку та операційну стійкість. Твердження роботи стосуються архітектурної композиції та інженерного рецепту, а не якості виробничого продукту.

6.7. Аналіз архітектурних рішень за результатами тестування

Експериментальний прогін також виступив фактичною валідацією основних архітектурних рішень платформи. Рішення про асинхронний брокер повідомлень підтвердилося стабільною доставкою сигналів присутності та оновлень стану кластера — тимчасові втрати мережі на боці окремих вузлів не призводили до помилок у центральному бекенді, а лише зміщували часові мітки сигналів. Рішення gRPC між центральним бекендом та диспетчером дозволило виявити невідповідності типів контракту під час компіляції TypeScript та Rust, а не під час виконання. Двоспрямована потокова передача gRPC у MonitorJob дозволила диспетчеру надсилати оновлення стану кластера без потреби в опитуванні з боку бекенду.

Рішення про об’єктне сховище Garage як робочий простір клієнта виявилося практичним: підвантаження шардів обсягом до 15 ГБ виконувалося без обмежень розміру, а артефакти, записані користувацьким зберігачем моделі, ставали доступними клієнту негайно після кожного раунду без окремого кроку завантаження. S3-сумісний API дозволив повторно використати стандартні бібліотеки boto3 та garage-sdk без додаткового шару адаптерів. Рішення про полімовну архітектуру з Rust-диспетчером дозволило виконувати виклики до NVIDIA FLARE Admin API напряму через pyo3 без міжпроцесного RPC, що зменшило затримку подачі завдання та спростило обробку помилок Python-бібліотеки на стороні Rust-сервісу.

6.8. Області застосування результатів

Результати роботи безпосередньо застосовні до медичної візуалізації — розгортання FL для діагностичних моделей між клініками, де централізована агрегація даних блокується HIPAA та GDPR. Платформа надає менеджмент-шар над FLARE, який зменшує операційні витрати на FL-розгортання, ідентифіковані Schoenpflug та співавторами [12] як основну перешкоду до більшого впровадження.

У фармацевтичних дослідженнях платформа застосовна для спільного дотренування моделей виявлення активних сполук між біотехнологічними компаніями, де дані експериментів є інтелектуальною власністю, за шаблоном, що використовується Lilly TuneLab [6]. У фінансових послугах платформа підходить для виявлення шахрайства та антивідмивальних моделей між установами, де регуляторика забороняє транскордонну передачу транзакційних даних [1]. В академічних дослідженнях платформа дозволяє спільне тренування моделей між університетами без передачі сирих даних, що особливо релевантно для соціально-наукових даних з вимогами IRB-етики.

6.9. Перспективи подальшого розвитку

Експериментальна частина має конкретні напрями розширення, які випливають із меж застосовності, явно зафіксованих у підрозділі 6.6. Розширення на 2-клієнтську та 8-клієнтську топології зі збереженням загального обчислювального бюджету 2960 SGD-кроків покаже, чи є основний результат специфічним до конкретного K=4. Альтернативний агрегатор (FedProx з μ=0.01) на тому ж рецепті покаже алгоритмічну гнучкість платформи: налаштування

виконується лише у config_fed_server.json без зміни коду платформи. Експериментальна перевірка часткової агрегації через примусове відключення одного з чотирьох клієнтів у середині раунду емпірично підтвердить заявлену відмовостійкість. Окреме вимірювання накладних витрат платформи через no-op-завдання з мінімальним тренуванням дозволить розкласти час повного циклу на компоненти — провізіонування контейнерів, gRPC-операції, S3-передавання, FL-комунікації.

Поточна реалізація образу вузла навчання підтримує лише федеративне навчання через NVIDIA FLARE. Розширення вузла до запуску стратегій DDP, FSDP та DeepSpeed ZeRO на одному вузлі чи групі вузлів дозволить платформі обслуговувати робочі процеси тренування великих моделей у діапазоні 7B–70B параметрів та більше, які не вміщуються у память одного GPU. NVIDIA FLARE підтримує координацію всіх цих стратегій у єдиній специфікації завдання, тому розширення є інкрементальним.

Перенесення оркестрації контейнерів з Docker socket на Kubernetes API є природним наступним кроком для розгортання платформи у середовищах, де постачальники GPU вже мають K8s-кластери. Архітектура диспетчера з виокремленим модулем platform/docker_utils.rs дозволяє замінити цей модуль на kube-rs або аналогічний клієнт Kubernetes без зміни решти коду. Кубернетес-розгортання надасть переваги декларативної конфігурації, нативної ізоляції через простори імен та готових інструментів моніторингу.

Розширення алгоритму відбору вузлів композитною оцінкою з накопиченою статистикою надійності, поточним завантаженням GPU та класом обладнання заплановано як технічне зобов’язання наступної ітерації; формальна модель такого алгоритму та її аналіз чутливості до ваг компонентів є природною темою окремого дослідження. Інтеграція диференційної приватності та гомоморфного шифрування для передачі оновлень ваг між клієнтами навчання та сервером агрегації [26, 30, 31] для досягнення рівня L2 ізоляції даних може бути реалізована через точки розширення NVIDIA FLARE без модифікації коду

платформи. Інтеграція secure aggregation за моделлю Bell та співавторів [26] забезпечила б захист від атак інверсії градієнтів без втрати точності моделі.

Асинхронне федеративне навчання дозволить серверу агрегації починати агрегацію після отримання оновлень від перших K < N клієнтів замість синхронного очікування усіх N клієнтів. Це зменшить вплив відстаючих вузлів ціною модифікації алгоритму агрегації для коректної обробки розбіжностей версій моделі [32]. Завершальним напрямом є запровадження сильніших механізмів автентифікації клієнтів через OIDC, шифрування артефактів на стороні клієнта та цілісність ланцюга образів через signed manifests — ці механізми перенесуть платформу з експериментального режиму у режим виробничого розгортання.

ВИСНОВКИ

У магістерській роботі вирішено актуальну науково-практичну задачу проектування та реалізації керованої платформи GPU-обчислень із децентралізованим постачанням обладнання для розподіленого навчання нейронних мереж на основі парадигми федеративного навчання.

Здійснено класифікаційний аналіз стратегій розподіленого навчання нейронних мереж та наявних платформ GPU-обчислень. Встановлено, що існуючі класи рішень реалізують лише підмножину необхідних властивостей: децентралізовані маркетплейси (Akash, Vast.ai) забезпечують відкрите постачання обладнання, але не оркеструють навчальний процес; керовані FL-платформи (Rhino Health FCP, Lilly TuneLab) забезпечують оркестрацію, але прив'язані до фіксованого пулу інституційних учасників. Виявлений розрив визначив нішу розробленої платформи: відкрите постачання GPU від незалежних постачальників у поєднанні з керованою оркестрацією федеративного навчання.

Обґрунтовано вибір федеративного навчання як архітектурної парадигми платформи. Властивість FL-протоколу, за якою між учасниками передаються лише оновлення ваг моделі, а не самі навчальні дані, забезпечує ізоляцію сирих даних клієнта від постачальників GPU на рівні структури протоколу. Визначено межі цієї гарантії: поточна реалізація забезпечує ізоляцію рівня L1 (сирі дані не залишають клієнтського робочого простору), тоді як захист переданих оновлень ваг від атак інверсії градієнтів (рівень L2) потребує інтеграції диференційної приватності та secure aggregation і винесений у перспективи розвитку.

Визначено технологічну базу платформи та обґрунтовано вибір кожного компонента. Контейнеризація через Docker забезпечує відтворюваність середовища виконання на гетерогенному обладнанні; gRPC між центральним бекендом та диспетчером забезпечує типобезпечну синхронну взаємодію з виявленням невідповідностей контракту під час компіляції; Redis як асинхронний брокер повідомлень забезпечує стійку доставку команд вузлам за

умов нестабільного мережевого з'єднання; об’єктне сховище Garage S3 забезпечує ізольований робочий простір для навчальних даних та артефактів моделі кожного клієнта.

Спроектовано та реалізовано трикомпонентну архітектуру: центральний бекенд platform-api на TypeScript з фреймворком NestJS, що інкапсулює реєстр вузлів, обробку сигналів присутності, алгоритм відбору вузлів та стан-машину завдань; диспетчер flare-dispatcher на Rust з gRPC-сервером tonic та інтеграцією з NVIDIA FLARE Admin API через pyo3-міст у межах одного процесу; контейнерний образ вузла навчання на основі NVIDIA FLARE-клієнта з автономною реєстрацією у мережі та самостійним отриманням завдань. Застосування pyo3-моста дозволило уникнути міжпроцесного RPC до Python-компонента NVIDIA FLARE, зменшивши затримку подачі завдання та спростивши обробку помилок.

Проведено серію тренувальних експериментів із рецептом FedAvg-навчання моделі ViT-B/16 на наборі даних Food-101 у чотирьох повторах кожної конфігурації з різними значеннями seed. Середній top-1 FL-конфігурації (4 клієнти, 5 раундів, 1 локальна епоха) становить 88.77% ± 0.09 п.п.; середній top-1 централізованої конфігурації з рівнозначним обчислювальним бюджетом у 2960 SGD-кроків становить 86.24% ± 0.16 п.п. Двовибірковий t-тест Велча ($n=4$, $df\approx5$) підтверджує статистичну значущість різниці ($p < 0.001$); з огляду на малий розмір вибірки результат інтерпретується як стабільне спостереження у конкретному рецепті та топології. Спостережена різниця зумовлена перенавчанням централізованого прогону на п’ятій епосі, яке пригнічується міжраундовим усередненням FedAvg. Аналіз внеску модифікацій рецепту показав, що безперервний LR-розклад через глобальний крок і збереження стану оптимізатора між раундами є необхідними умовами відтворення результату; EMA з коефіцієнтом 0.999 та label smoothing $\varepsilon=0.1$ призвели до регресії у режимі коротких FL-раундів. Отримані результати справедливі для IID-розподілу даних між клієнтами; узагальнення на non-IID-режим виходить за межі поточного дослідження.

Встановлено практичну застосовність розроблених архітектурних рішень у доменах із регуляторними обмеженнями на передачу даних: медичній візуалізації, фармацевтичних дослідженнях та фінансових послугах, де федеративне навчання на базі NVIDIA FLARE вже використовується у виробничих розгортаннях. Задокументований рецепт дотренування моделей сімейства Vision Transformer із безперервним LR-розкладом та збереженням стану оптимізатора переноситься на аналогічні розгортання NVIDIA FLARE без модифікації коду фреймворку.

Перспективи подальшого розвитку охоплюють: розширення серії експериментів до 8–10 повторів кожної конфігурації для надійніших довірчих інтервалів, варіацію топології (2 та 8 клієнтів) і альтернативних агрегаторів (FedProx, FedOpt); розширення образу вузла для підтримки стратегій DDP, FSDP та DeepSpeed ZeRO; перенесення оркестрації контейнерів з Docker socket на Kubernetes API; інтеграцію диференційної приватності та secure aggregation для досягнення рівня L2 ізоляції; запровадження автентифікації клієнтів через OIDC та шифрування артефактів як передумов виробничого розгортання.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Roth, H. R., Tickoo, S., Kumar, M., Yang, I., Liu, A., Varshney, A., Kundu, S., Vintila, I., Madsgaard, P., Milcak, J., Chen, C., Cheng, Y., Feng, A., Savio, J., Singh, V., & Stancill, C. (2026). *Privacy-preserving federated fraud detection in payment transactions with NVIDIA FLARE* (arXiv:2603.13617). arXiv. https://doi.org/10.48550/arXiv.2603.13617

2. Han, C. (2021, October 13). *NVIDIA brings collaborative AI to healthcare and beyond: Federated learning with FLARE*. NVIDIA Blog. https://blogs.nvidia.com/blog/federated-learning-ai-nvidia-flare/

3. Chaudhary, V. (2026). Federated learning: A survey of core challenges, current methods, and opportunities. *Computers, 15*(3), Article 155. https://doi.org/10.3390/computers15030155

4. Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M. X., Chen, D., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., & Chen, Z. (2019). GPipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems 32*. Curran Associates.

5. Li, T., Sahu, A. K., Zaheer, M., Sanjabi, M., Talwalkar, A., & Smith, V. (2020). Federated optimization in heterogeneous networks. *Proceedings of Machine Learning and Systems, 2*, 429–450.

6. HPCwire. (2026, February 27). *Lilly launches LillyPod NVIDIA DGX SuperPOD for genomics and drug discovery AI*. https://www.hpcwire.com/off-the-wire/lilly-launches-lillypod-nvidia-dgx-superpod-for-genomics-and-drug-discovery-ai/

7. Roth, H. R., Cheng, Y., Wen, Y., Yang, I., Xu, Z., Hsieh, Y.-T., Kersten, K., Harouni, A., Zhao, C., Lu, K., Zhang, Z., Li, W., Myronenko, A., Yang, D., Yang, S., Rieke, N., Quraini, A., Chen, C., Xu, D., … Feng, A. (2022). *NVIDIA FLARE: Federated learning from simulation to real-world* (arXiv:2210.13291). arXiv. https://doi.org/10.48550/arXiv.2210.13291

8. Roth, H. R., Xu, Z., Hsieh, Y.-T., Renduchintala, A., Yang, I., Zhang, Z., Wen, Y., Yang, S., Lu, K., Kersten, K., Ricketts, C., Xu, D., Chen, C., Cheng, Y., & Feng, A. (2024). *Empowering federated learning for massive models with NVIDIA FLARE* (arXiv:2402.07792). arXiv. https://doi.org/10.48550/arXiv.2402.07792

9. McMahan, H. B., Moore, E., Ramage, D., Hampson, S., & Agüera y Arcas, B. (2017). Communication-efficient learning of deep networks from decentralized data. In A. Singh & J. Zhu (Eds.), *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*. PMLR.

10. Meta. (2024). *Llama 3.1 model card*. GitHub. https://github.com/meta-llama/llama-models/blob/main/models/llama3_1/MODEL_CARD.md

11. Epoch AI. (2024–2026). *Llama 3.1-405B* [Database entry]. https://epoch.ai/models/llama-3-1-405b

12. Schoenpflug, L. A., Bagan Benavides, R., Nowak, M., Sheikhzadeh, F., Moayyedi, A., Wasag, K., Reimers, J., Zhou, M., Venugopal, R., Sobottka, B., Koeller, Y., Rivers, M., Moch, H., Nie, Y., & Koelzer, V. H. (2025). Navigating real-world challenges: A case study on federated learning in computational pathology. *Journal of Pathology Informatics, 16*, Article 100464. https://doi.org/10.1016/j.jpi.2025.100464

13. Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., & Catanzaro, B. (2019). *Megatron-LM: Training multi-billion parameter language models using model parallelism* (arXiv:1909.08053). arXiv. https://doi.org/10.48550/arXiv.1909.08053

14. Gupta, R., Nalawade, S., & Chowdhury, A. (2025). *Benchmarking federated learning frameworks for medical imaging deployment: A comparative study of NVIDIA FLARE, Flower, and Owkin Substra* (arXiv:2511.00037). arXiv. https://doi.org/10.48550/arXiv.2511.00037

15. Sevilla, J., Heim, L., Ho, A., Besiroglu, T., Hobbhahn, M., & Villalobos, P. (2022). Compute trends across three eras of machine learning. In *2022 International Joint Conference on Neural Networks (IJCNN)* (pp. 1–8). IEEE. https://doi.org/10.1109/IJCNN55064.2022.9891914

16. Docker, Inc. (n.d.). *Docker Engine API documentation*. Retrieved May 20, 2026, from https://docs.docker.com/engine/api/

17. Redis Ltd. (n.d.). *Redis documentation*. Retrieved May 20, 2026, from https://redis.io/docs/

18. NestJS Authors. (n.d.). *NestJS — A progressive Node.js framework*. Retrieved May 20, 2026, from https://docs.nestjs.com/

19. Google. (n.d.). *gRPC — A high performance, open source universal RPC framework*. Retrieved May 20, 2026, from https://grpc.io/docs/

20. Wightman, R. (n.d.). *PyTorch image models (timm)* [Computer software]. GitHub. Retrieved May 20, 2026, from https://github.com/huggingface/pytorch-image-models

21. Bossard, L., Guillaumin, M., & Van Gool, L. (2014). Food-101 — Mining discriminative components with random forests. In D. Fleet, T. Pajdla, B. Schiele, & T. Tuytelaars (Eds.), *Computer Vision — ECCV 2014*. Springer. https://doi.org/10.1007/978-3-319-10599-4_29

22. Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., & Houlsby, N. (2021). An image is worth 16x16 words: Transformers for image recognition at scale. In *9th International Conference on Learning Representations (ICLR 2021)*. https://openreview.net/forum?id=YicbFdNTTy

23. Flexera. (2025). *State of the cloud report 2025*. Flexera Software LLC. https://info.flexera.com/cm-report-state-of-the-cloud

24. Zhao, Y., Liu, X., Liu, S., Li, X., Zhu, Y., Huang, G., Liu, X., & Jin, X. (2023). *MuxFlow: Efficient and safe GPU sharing in large-scale production*

deep learning clusters (arXiv:2303.13803). arXiv. https://doi.org/10.48550/arXiv.2303.13803

25. Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., & He, K. (2017). *Accurate, large minibatch SGD: Training ImageNet in 1 hour* (arXiv:1706.02677). arXiv. https://doi.org/10.48550/arXiv.1706.02677

26. Bell, J. H., Bonawitz, K. A., Gascón, A., Lepoint, T., & Raykova, M. (2020). Secure single-server aggregation with (poly)logarithmic overhead. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20)*. ACM. https://doi.org/10.1145/3372297.3417885

27. Beutel, D. J., Topal, T., Mathur, A., Qiu, X., Fernandez-Marques, J., Gao, Y., Sani, L., Li, K. H., Parcollet, T., Porto Buarque de Gusmão, P., & Lane, N. D. (2020). *Flower: A friendly federated learning research framework* (arXiv:2007.14390). arXiv. https://doi.org/10.48550/arXiv.2007.14390

28. Sambasivan, N., Kapania, S., Highfill, H., Akrong, D., Paritosh, P., & Aroyo, L. M. (2021). "Everyone wants to do the model work, not the data work": Data cascades in high-stakes AI. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM. https://doi.org/10.1145/3411764.3445518

29. NVIDIA Corporation. (n.d.). *NVIDIA Container Toolkit documentation*. Retrieved May 20, 2026, from https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/

30. Geiping, J., Bauermeister, H., Dröge, H., & Moeller, M. (2020). Inverting gradients — How easy is it to break privacy in federated learning? In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, & H. Lin (Eds.), *Advances in Neural Information Processing Systems 33*. Curran Associates.

31. Dwork, C., & Roth, A. (2014). The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science, 9*(3–4). https://doi.org/10.1561/0400000042

32. Xie, C., Koyejo, S., & Gupta, I. (2020). Asynchronous federated optimization. In *OPT2020: 12th Annual Workshop on Optimization for Machine Learning*. https://opt-ml.org/oldopt/papers/2020/paper_28.pdf

33. Rajbhandari, S., Rasley, J., Ruwase, O., & He, Y. (2020). ZeRO: Memory optimizations toward training trillion parameter models. In *Proceedings of SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. https://doi.org/10.1109/SC41405.2020.00024

ДОДАТКИ

Додаток А. Структура артефактів коду

Додаток містить опис структури коду розробленої платформи та перелік ключових артефактів реалізації, що зберігаються у репозиторії проєкту. Через значний обсяг коду повні лістинги наведено окремо у репозиторії; нижче подано структуру каталогів та перелік основних файлів зі стислою характеристикою кожного.

А.1. Загальна структура репозиторію.

/platform-api — центральний бекенд на TypeScript з NestJS;

/flare-dispatcher — диспетчер на Rust з gRPC-сервером tonic;

/worker-image — Docker-образ вузла навчання з NVIDIA FLARE-клієнтом;

/job-submitter — клієнтський SDK на Python та CLI-утиліта submit-job;

/garage-s3 — допоміжна обгортка над Garage S3 для роботи з робочим простором.

А.2. Контракт gRPC між platform-api та flare-dispatcher.

Файл /flare-dispatcher/proto/dispatcher.proto визначає сервіс FlareDispatcher з 15 операціями: ProvisionFlareServer, SubmitJob, UploadDataset, GetSystemInfo, CheckStatus, ListJobs, GetJobMeta, AbortJob, DeleteJob, MonitorJob, DownloadJobResult, ListJobComponents, Shutdown, ShutdownSystem, ShowStats. Контракт компілюється у TypeScript-заглушки через @grpc/proto-loader на стороні platform-api та у Rust-структури через prost на стороні диспетчера.

А.3. Образ вузла навчання та пакет NVIDIA FLARE.

/worker-image/docker/Dockerfile — Dockerfile вузла на основі pytorch/pytorch з CUDA та встановленням NVIDIA FLARE через pip.

/worker-image/docker/start_client.sh — скрипт точки входу контейнера.

/worker-image/docker/model_def.py — визначення моделі та функція build_model().

/worker-image/docker/custom_persistor.py — успадковує ModelPersistor та реалізує запис контрольних точок у клієнтський робочий простір Garage S3.

/worker-image/docker/custom_client_executor.py — успадковує Executor та реалізує локальне тренування з підтримкою безперервного LR-розкладу через глобальний крок та persisted Adam state.

Конфігураційні файли config_fed_server.json (параметри агрегації) та config_fed_client.json (параметри локального тренування) подаються клієнтом у складі завдання.

А.4. Центральний бекенд platform-api.

/platform-api/src/app.module.ts — кореневий модуль NestJS з імпортом AppConfigModule, PrismaModule, CacheModule, BrokerModule, RedisModule, GrpcModule, CentralStorageModule, TasksModule, NodesModule, HeartbeatModule, ClusterStateModule.

/platform-api/src/modules/tasks/ — модуль керування завданнями з TasksService, TasksController та стан-машиною на основі Prisma-enum TaskStatus (QUEUED, RUNNING, COMPLETED, FAILED, CANCELLED).

/platform-api/src/modules/cluster/services/worker-selection.service.ts — метод selectWorkers, який реалізує описаний у підрозділі 4.3 алгоритм відбору вузлів.

/platform-api/src/modules/nodes/ та /platform-api/src/modules/heartbeat/ — модулі реєстрації вузлів та обробки сигналів присутності.

/platform-api/prisma/schema.prisma — схема бази даних з таблицями Task, Worker, Heartbeat та супутніми.

А.5. Диспетчер flare-dispatcher.

/flare-dispatcher/src/dispatcher-server.rs — точка входу з ініціалізацією gRPC-сервера tonic.

/flare-dispatcher/src/platform/docker_utils.rs — модуль керування контейнерами через Docker socket: формування аргументів docker run, прокидування GPU через NVIDIA Container Toolkit, моніторинг стану контейнерів.

/flare-dispatcher/src/flare_admin/ — pyo3-міст до NVIDIA FLARE Admin API: подача специфікації завдання серверу агрегації, опитування стану раундів, отримання логів.

/flare-dispatcher/docker-compose.yml — конфігурація розгортання диспетчера з прокидуванням Docker-сокета та налаштуванням мережі.

А.6. Клієнтський SDK та CLI.

/job-submitter/submitter/client.py — клас submitter.Client для подання завдань з Python-коду та Jupyter-блокнотів.

/job-submitter/submitter/cli.py — реалізація CLI-утиліти submit-job як обгортки над SDK.

/job-submitter/examples/ — приклади подання навчальних завдань для класифікації зображень.

Додаток Б. Повна статистика основного експериментального порівняння

У додатку подано повний перелік окремих прогонів обох порівнюваних конфігурацій — федеративної (FL) та централізованої — у рецепті v2. Кожна конфігурація виконана у чотирьох повторах з різними значеннями seed (1337, 2024, 4096, 8192). На основі цих повторів обчислено усереднені значення та стандартні відхилення, наведені у Таблиці 6.1 та використані для двовибіркового t-тесту Велча у підрозділі 6.3.

Таблиця Б.1 — Окремі прогони FL-конфігурації v2 (4 клієнти × 5 раундів × 1 локальна епоха)

Повтор	Seed	Апаратура	top-1 vanilla	top-1 TTA	val_loss
1	1337	RTX PRO 4500	88.66%	88.87%	0.5125
2	2024	RTX PRO 4500	88.83%	89.24%	0.5756
3	4096	RTX PRO 4500	88.84%	89.17%	0.5695
4	8192	RTX 4090	88.73%	88.99%	0.5155
Середнє	—	—	88.77 ± 0.09	89.07 ± 0.16	0.543 ± 0.03

Таблиця Б.2 — Окремі прогони централізованої конфігурації (1×H100, batch=128, lr=2e-4, 5 epox)

Повтор	Seed	top-1 vanilla	top-1 TTA	val_loss

1	1337	86.17%	87.03%	0.6260
2	2024	86.41%	87.22%	0.6184
3	4096	86.05%	86.88%	0.6312
4	8192	86.33%	87.31%	0.6105
Середнє	—	86.24 ± 0.16	87.11 ± 0.20	0.621 ± 0.01

Обидві конфігурації використовують однакову модель ViT-B/16 (vit_base_patch16_224.augreg_in21k), однаковий набір даних Food-101 з фіксованим розбиттям на тренувальну (75 750 зображень) та валідаційну (25 250 зображень) частини, однаковий обчислювальний бюджет у 2960 SGD-кроків та один і той самий код локального тренування з безперервним LR-розкладом і persistent Adam state. Відмінність між конфігураціями полягає виключно у способі організації SGD-кроків: у FL-конфігурації — 4 клієнти × 5 раундів × 148 кроків з міжраундовим усередненням FedAvg; у централізованій — 5 епох × 592 кроки на одному пристрої без агрегації.

Двовибірковий t-тест Велча (нерівні дисперсії, $n_1 = n_2 = 4$) для vanilla top-1: $t \approx 27.6$, $df \approx 5$, $p < 0.001$. Для top-1 з TTA: $t \approx 15.3$, $df \approx 6$, $p < 0.001$. Враховуючи малі розміри вибірок, ці p-значення слід інтерпретувати як орієнтовні; разом з тим, інтервали (88.68, 88.86) для FL та (86.08, 86.40) для централізованої конфігурації не перетинаються з великим запасом, що підтверджує стабільність спостереженої різниці у конкретному рецепті.