Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя
Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра комп'ютерних систем та мереж
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

Магістр
(назва освітнього ступеня)

на тему: Методи та засоби MLOps для створення інтегрованої системи автоматизації безперервної побудови, розгортання та моніторингу моделей машинного навчання

Виконав: студент(ка) 6 курсу, групи СІм-61
спеціальності 123 «Комп'ютерна інженерія»
(шифр і назва спеціальності)

	(підпис)	Андруньків С.Р. (прізвище та ініціали)
Керівник	(підпис)	Луцків А.М. (прізвище та ініціали)
Нормоконтроль	(підпис)	Луцик Н.С. (прізвище та ініціали)
Завідувач кафедри	(підпис)	Осухівська Г.М. (прізвище та ініціали)
Рецензент	(підпис)	Фриз М.Є. (прізвище та ініціали)

Тернопіль
2025

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра комп'ютерних систем та мереж
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ *Осухівська Г.М.*
(підпис) (прізвище та ініціали)
«17» листопада 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня *магістр*
(назва освітнього ступеня)

за спеціальністю *123 «Комп'ютерна інженерія»*
(шифр і назва спеціальності)

студенту *Андрунькові Сергію Романовичу*
(прізвище, ім'я, по батькові)

1. Тема роботи *Методи та засоби MLOps для створення інтегрованої системи автоматизації безперервної побудови, розгортання та моніторингу моделей машинного навчання*

Керівник роботи Луцків Андрій Мирославович, к.т.н., доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від « *14* » *листопада 2025* року № 4/7-987

2. Термін подання студентом завершеної роботи *16 грудня 2025 р.*

3. Вихідні дані до роботи *наукові літературні джерела*

4. Зміст роботи (перелік питань, які потрібно розробити)

Вступ
1 Аналітична частина
2 Теоретична частина
3 Практична частина
4 Охорона праці та безпека в надзвичайних ситуаціях
Висновки

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)

1. Актуальність і мета дослідження.
2. Завдання, об'єкт і предмет, наукова новизна і практична цінність дослідження.
3. Архітектура платформи MLOps
4. Деталізований алгоритм життєвого циклу системи MLOps
5. Архітектура Linkerd та мережевої взаємодії в Kubernetes
6. Діграма класів та блок-схеми алгоритмів train.py і app.py
7. Діаграма потоків даних ML-процесу
8. Висновки

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Охорона праці	*Осухівська Г.М., зав.каф. КС*	*04.12.2025*	
Безпека в надзвичайних ситуаціях	*Стручок В.С., ст.викл.каф. ОХ*	*04.12.2025*	

7. Дата видачі завдання *17.11.2025 р.*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	*Огляд літературних джерел. Постановка завдання на кваліфікаційну роботу*	*17.11.2025р. – 23.11.2025р.*	*Викон.*
2.	*Проєктування архітектури платформи MLOps та вибір інструментальних засобів*	*24.11.2025р. – 02.12.2025р.*	*Викон.*
3.	*Розробка, розгортання, тестування та моніторинг компонентів системи MLOps*	*03.12.2025р. – 10.12.2025р.*	*Викон.*
4.	*Охорона праці та безпека в надзвичайних ситуаціях*	*04.12.2025р. – 10.12.2025р.*	*Викон.*
5.	*Оформлення пояснювальної записки і графічного матеріалу*	*11.12.2025р. – 19.12.2025р.*	*Викон.*
6.	*Попередній захист кваліфікаційної роботи магістра*	*16.12.2025р.*	*Викон.*
7.	*Захист кваліфікаційної роботи магістра*	*22.12.2025*	*Викон.*

Студент _____ *Андруньків Сергій Романович*
(підпис) (прізвище та ініціали)

Керівник роботи _____ *Луцків Андрій Мирославович*
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Андруньків С. Р. Методи та засоби MLOps для створення інтегрованої системи автоматизації безперервної побудови, розгортання та моніторингу моделей машинного навчання: робота на здобуття кваліфікаційного ступеня магістра: спец. 123 — комп’ютерна інженерія / наук.кер. Луцків А. М. Тернопіль: Тернопільський національний технічний університет імені Івана Пулюя, 2025.

Ключові слова: MLOps, GitOps, автоматизація, безперервна інтеграція, інфраструктура як код, контейнеризація, оркестрація, Kubernetes, Docker, версіонування даних, DVC, відстеження експериментів, MLflow, аналіз коду, моніторинг, Linkerd, Argo CD, безпека.

Кваліфікаційна робота присвячена вирішенню сучасної проблеми операціоналізації машинного навчання. Вона демонструє проєктування та практичну реалізацію комплексної, автоматизованої платформи для повного життєвого циклу моделі аналізу тональності. Метою є створення надійного, відтворюваного та безпечного конвеєра, який дозволяє розробникам швидко доставляти оновлення моделі від коміту коду до робочого середовища, мінімізуючи ручне втручання та ризики, пов'язані з розгортанням.

Дана робота поєднує стек інструментів DevOps та MLOps. Система побудована навколо GitLab CI як ядра автоматизації, Kubernetes як платформи оркестрації та Argo CD для реалізації GitOps-підходу. Процес автоматизує версіонування даних, відстеження експериментів, контроль якості коду та контейнеризацію. Особлива увага приділена просунутим концепціям, таким як Service Mesh, що забезпечує автоматичне mTLS-шифрування та глибокий моніторинг "золотих сигналів" розгорнутого API-сервісу.

ANNOTATION

Andrunkiv S.R. Methods and tools of MLOps for creating an integrated automation system for continuous building, deployment, and monitoring of machine learning models. Master's Graduation Thesis: speciality 123 — Computer engineering / supervisor Lutskiv A.M. Ternopil: Ternopil Ivan Puluj National Technical University, 2025.

Keywords: MLOps, GitOps, automation, continuous integration, infrastructure as code, containerization, orchestration, Kubernetes, Docker, data versioning, DVC, experiment tracking, MLflow, code analysis, monitoring, Linkerd, Argo CD, security.

The qualification thesis is dedicated to solving the modern problem of operationalizing machine learning (MLOps). It demonstrates the design and practical implementation of a comprehensive, automated platform for the complete lifecycle of a sentiment analysis model. The objective is to create a reliable, reproducible, and secure pipeline that allows developers to quickly deliver model updates from code commit to the production environment, minimizing manual intervention and deployment-related risks.
This work combines a stack of DevOps and MLOps tools. The system is built around GitLab CI as the core of automation, Kubernetes as the orchestration platform, and Argo CD for implementing the GitOps approach. The process automates data versioning, experiment tracking, code quality control, and containerization. Special attention is given to advanced concepts, such as a Service Mesh, which provides automatic mTLS encryption and deep monitoring of the deployed API service's "golden signals".

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ML (англ. Machine Learning) методи побудови алгоритмів, що навчаються на даних

MLOps (англ. Machine Learning Operations) практики для автоматизації життєвого циклу машинного навчання

DevOps (англ. Development and Operations) набір практик для автоматизації та інтеграції процесів між командами розробки та експлуатації

GitOps (англ. Git Operations) управління інфраструктурою через Git-репозиторій

CI/CD (англ. Continuous Integration / Continuous Deployment) безперервна інтеграція та розгортання коду

K8S (англ. Kubernetes) платформа для управління контейнеризованими додатками

IaC (англ. Infrastructure as Code) управління інфраструктурою через код

YAML (англ. YAML Ain't Markup Language) читабельний формат даних для файлів конфігурації

HCL (англ. HashiCorp Configuration Language) мова конфігурації інфраструктури від HashiCorp

NLTK (англ. Natural Language Toolkit) бібліотека Python для обробки природної мови

DVC (англ. Data Version Control) система контролю версій для даних та ML-моделей

SSH (англ. Secure Shell) протокол для безпечного віддаленого керування сервером

URL (англ. Uniform Resource Locator) адреса ресурсу в інтернеті

CRD (англ. Custom Resource Definitions) розширення API Kubernetes власними типами ресурсів

JSON (англ. JavaScript Object Notation) текстовий формат обміну даними на основі JavaScript

“Розрив операціоналізації” проблема у сфері ML, коли модель, яка розроблена та навчена в експериментальному середовищі, не здатна стабільно працювати у виробничому середовищі.

“Золоті сигнали” набір метрик, що дозволяють миттєво та точно оцінити загальний стан сервісу.

“Дрейф конфігурації” відмінність фактичного стану інфраструктури від її бажаного, задекларованого стану.

“Єдине джерело правди” єдине місце, де зберігається бажаний стан всієї системи.

ВСТУП

Актуальність роботи. Успішне впровадження моделей ML у виробництво часто стикається з викликами, такими як висока складність CI/CD пайплайнів, необхідність в адаптації до змін у даних, а також інтеграція з існуючими системами. Ці проблеми не тільки сповільнюють процеси, але і можуть призвести до зниження продуктивності моделей.

Сфера MLOps достатньо стрімко розвивається завдяки інтеграції практик DevOps у процеси машинного навчання, що сприяє ефективнішій розробці, тестуванню та розгортанню моделей. Цими завданнями займалися такі дослідники як Ioannis Karamitsos, Georgios Symeonidis та інші. Їхні публікації показують різні аспекти, проте існує потреба у більш глибокому вивченні питань моніторингу і управління даними. Багато досліджень фокусуються на окремих етапах, проте комплексні рішення, що об’єднують всі етапи життєвого циклу, досліджуються недостатньо.

У дослідженнях даних авторів не приділено достатньо уваги таким аспектам, як моніторинг продуктивності моделей, автоматизація обробки даних і управління залежностями та комплексні рішення для всього життєвого циклу ML.

Дані дослідження зосереджуються лише на етапах CI/CD, моніторингу, збору даних. При цьому недостатньо представлено рішень, що об’єднують усі етапи життєвого циклу моделей. Справжня інтеграція повинна вимагати єдиного підходу до всіх аспектів – від обробки даних до моніторингу в реальному часі.

Саме тому розробка інтегрованої системи автоматизації є актуальною темою для дослідження, що може підвищити ефективність життєвого циклу ML-моделей. Оскільки, методології MLOps та LLMOps є відносно новими, й тому потребують покращення та систематизації.

Метою кваліфікаційної роботи є проєктування та практична реалізація надійного MLOps-конвеєра, який автоматизує повний життєвий цикл ML-моделі шляхом інтеграції CI/CD, GitOps-розгортання, версіонування даних та просунутого моніторингу в Kubernetes.

Завдання кваліфікаційної роботи:

– провести аналіз існуючих проблем та викликів у процесі операціоналізації моделей машинного навчання;

– дослідити сучасні методології автоматизації, такі як GitOps, та архітектурні підходи до побудови відтворюваних CI/CD конвеєрів для ML-систем;

– спроєктувати комплексну архітектуру автоматизованої платформи, що інтегрує етапи версіонування даних, відстеження експериментів, контролю якості коду та оркестрації контейнерів;

– розробити та програмно реалізувати наскрізний конвеєр, що автоматизує повний життєвий цикл ML-додатку: від тренування моделі до її пакування та безпечного розгортання;

– інтегрувати в систему інструменти безпеки та спостережності для моніторингу "золотих сигналів" розгорнутого сервісу;

– провести експериментальну верифікацію та функціональне тестування розробленої системи для підтвердження її працездатності та ефективності.

Об'єкт дослідження: процес автоматизації повного життєвого циклу моделей машинного навчання у сучасному виробничому середовищі.

Предмет дослідження: Методи, засоби, моделі та архітектурні підходи для побудови інтегрованих конвеєрів, що забезпечують версіонування даних, якість коду, безпечне розгортання та спостережуваність сервісів.

Методи дослідження. На початковому етапі було проведено аналіз та синтез існуючих рішень для автоматизації MLOps-циклу, що дозволило змоделювати оптимальну архітектуру інтегрованої системи. Для практичної верифікації цієї архітектури були використані емпіричні методи. На завершальному етапі, статистичний аналіз та імітаційне моделювання були застосовані для оцінки метрик продуктивності моделі, виявлення аномалій та підтвердження стійкості всієї системи під прогнозованим навантаженням.

Наукова новизна дослідження. Вдосконалено архітектурний підхід до побудови MLOps-платформ шляхом комплексної інтеграції CI/CD-конвеєра з декларативною моделлю GitOps та проактивною системою безпеки і

спостережності. На відміну від відомих рішень, які часто фокусуються окремо на тренуванні моделі або її розгортанні, запропонована в роботі архітектура вперше об'єднує в єдиний, повністю автоматизований цикл усі етапи життєвого циклу, починаючи від статичного аналізу коду та версіонування даних, закінчуючи безпечним поступовим оновленням в середовищі оркестрації з миттєвим моніторингом кількості запитів на секунду на рівні мережі.

Практичне значення результатів кваліфікаційної роботи. Розроблений наскрізний конвеєр може бути використаний, як готова, відтворювана архітектура та практична рекомендація для компаній, що дозволяє суттєво прискорити та убезпечити процес впровадження моделей машинного навчання шляхом комплексної автоматизації, версіонування та моніторингу їхнього повного життєвого циклу.

Публікації. Результати дослідження апробовано на XIV міжнародній науково-практичній конференції молодих учених та студентів «Актуальні задачі сучасних технологій», XIII науково-технічній конференції Тернопільського національного технічного університету імені Івана Пулюя «Інформаційні моделі, системи та технології», у вигляді тез конференцій.

Структура роботи. Робота складається з пояснювальної записки та графічної частини. Пояснювальна записка складається із вступу, 4 розділів, висновків, списку використаних джерел та додатків.

РОЗДІЛ 1
АНАЛІТИЧНА ЧАСТИНА

1.1. Проблематика життєвого циклу ML-систем

Протягом останніх років моделі машинного навчання перетворилися з академічних досліджень на компоненти сучасних програмних продуктів. Проте, попри великі успіхи в розробці складних алгоритмів, індустрія зіткнулася з проблемою, відомою як “розрив операціоналізації”. Ця проблема виникає через культурну, процесну та інструментальну прірву між спеціалістами в галузі даних та інженерами з експлуатації. Ці дві групи працюють у різних парадигмах, мають різні цілі та виробляють кардинально відмінні артефакти.

Команда спеціалістів в галузі даних, як правило, зосереджена на експериментах та максимізації точності моделі. Їхнє робоче середовище – це інтерактивні блокноти та локальні машини, а кінцевий продукт – це часто статичний файл, який працює переважно лише на їхній машині. З іншого боку, команда DevOps відповідає за надійність, масштабованість та безпеку системи у виробничому середовищі. Їхня парадигма – це автоматизація, контейнеризація та безперервна інтеграція, а їхні артефакти – це образи Docker та маніфести Kubernetes. Цей конфлікт породжує плутанину, коли дослідники надають готовий файл моделі, а інженери не знають, як його надійно запустити, масштабувати, версіонувати та моніторити.

Цей розрив призводить до наступних проблем, які сильно сповільнюють впровадження машинного навчання:

– проблема відтворюваності – модель, натренована шість місяців тому, може бути неможливо відтворити через зміни в коді, бібліотеках або самих даних, що робить її некерованою;

– проблема “останньої милі” – перетворення статичного файлу моделі на масштабований, захищений REST API-сервіс є нетривіальною інженерною задачею;

– традиційний DevOps-моніторинг є сліпим до проблем машинного навчання, тобто сервіс працює, але починає давати неточні прогнози через зміну вхідних даних у реальному світі.

В результаті цього, за деякими оцінками, понад вісімдесят відсотків ML-проєктів так і не досягають реального впровадження. Ті ж, що розгортаються, часто є крихкими, непідтримуваними та непрозорими, оновлення яких займає місяці. Саме ця проблематика – необхідність подолання розриву між експериментальною наукою та надійною інженерією – формує основний запит на створення нової, комплексної методології, відомої як MLOps.

1.2. MLOps як комплексна методологія автоматизації та управління системами машинного навчання

У відповідь на раніше розглянуті проблеми, індустрія сформувала нову культуру та набір практик, відомих як MLOps – Machine Learning Operations. Це не просто набір інструментів, а, перш за все, філософія та методологія, яка запозичує принципи DevOps, такі як безперервна інтеграція, безперервне розгортання та моніторинг, і адаптує їх до унікальних викликів, пов'язаних із життєвим циклом моделей машинного навчання. MLOps вимагає тісної співпраці між командами Data Science, інженерами DevOps та фахівцями з обробки даних, об’єднуючи їхні компетенції [11].

Якщо DevOps фокусується на автоматизації розгортання коду, MLOps розширює цю парадигму, включаючи в неї також дані та моделі як артефакти першого класу, якими необхідно керувати. Як показано на діаграмі Венна (див. рис. 1.1), MLOps знаходиться на перетині дисциплін, що відповідають за тренування, супровід та загальну інфраструктуру машинного навчання. Він бере від DevOps культуру автоматизації та надійності, від Data Engineering – надійні конвеєри даних, а від Machine Learning – експерименти та моделі.

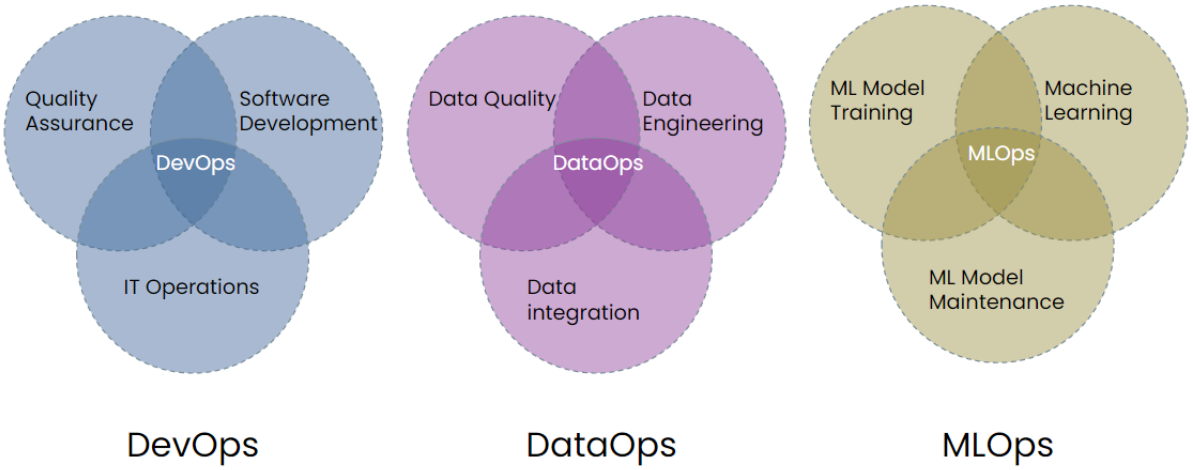

Рис. 1.1. MLOps як перетин інженерних дисциплін та машинного навчання

На відміну від традиційного життєвого циклу розробки програмного забезпечення, життєвий цикл MLOps є безперервним та ітеративним (див. рис. 1.2). Він не закінчується на етапі розгортання, а переходить у фазу постійного моніторингу. У цій фазі він відстежує не лише технічні показники, але і якість прогнозів моделі та характеристики вхідних даних, виявляючи деградацію моделі. Дані моніторингу, в свою чергу, стають тригером для нового циклу, починаючи з підготовки свіжих даних та перетренування моделі.

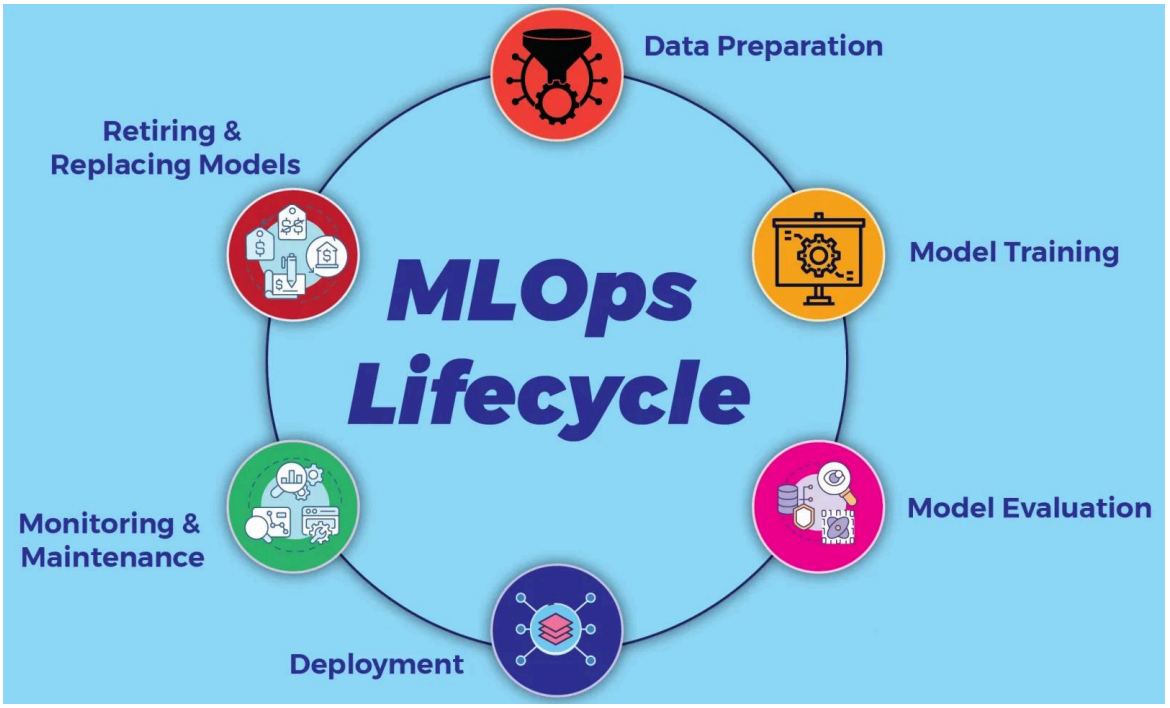

Рис. 1.2. Безперервний життєвий цикл MLOps

Описаний вище підхід базується на чотирьох фундаментальних принципах:

1) автоматизація – кожен етап, від тренування до розгортання, має бути автоматизованим конвеєром;

2) повне версіонування – версіонується не лише код через Git, але й дані через DVC та моделі через MLflow;

3) моніторинг продуктивності системи та якості прогнозів;

4) відтворюваність – можливість у будь-який момент часу точно відтворити будь-який експеримент або розгорнуту версію моделі, знаючи, який код, які дані та які параметри були використані для її створення.

1.3. Аналіз сучасних архітектурних підходів до розгортання сервісів ML

Розгортання моделі машинного навчання у виробничому середовищі вимагає вибору правильної архітектурної стратегії. Перші спроби інтеграції моделей машинного навчання відбувалися з використанням монолітного підходу. У цьому випадку натренована модель та код для її виконання просто додавалися як нова бібліотека або модуль у код основного додатку. Такий підхід досить швидко продемонстрував неефективність у зв’язку із конфліктами залежностей бібліотек машинного навчання. Життєві цикли були жорстко зв’язані, тобто оновлення моделі вимагало повної перебудови та перезапуску всього додатку. Крім того, неможливо було масштабувати ресурсоємну частину окремо від решти додатку.

Ці проблеми призвели до впровадження мікросервісного підходу, який став стандартом для сучасних систем машинного навчання. Суть підходу полягає у повному відокремленні моделі від основного додатку. Модель загортається у власний, легкий веб-сервер, який надає єдиний, чітко визначений API-інтерфейс. Такий підхід забезпечує технологічну ізоляцію, при якій сервіс ML може бути на одній мові, а основний додаток на іншій. Іншими перевагами є незалежне масштабування, що дозволяє запустити декілька копій сервісу ML і веб-сервісу, а також стійкість до відмов, коли збій у моделі не призведе до зупинки всього програмного комплексу.

Однак, мікросервісний підхід неможливий без контейнеризації та оркестрації. Контейнеризація, реалізована за допомогою Docker, допомагає вирішити проблему відтворюваності та залежностей. Docker упаковує модель, її API-сервер та різні залежності в єдиний портативний артефакт – контейнер. Цей контейнер гарантовано буде працювати однаково на машині розробника, на тестовому стенді та у виробничому середовищі.

За допомогою оркестрації контейнерів з використанням Kubernetes ми можемо керувати цими контейнерами у великих масштабах. Запуск одного контейнера є простим, але управління десятками його копій, забезпечення їх автоматичного перезапуску в разі збою, балансування навантаження між ними та виконання поступових оновлень без простою – це складне завдання, яке бере на себе Kubernetes [16]. Як показано на рисунку 1.3, Kubernetes виступає як операційною системою для контейнерів Docker, керуючи їх життєвим циклом на кластері серверів. Зв’язок між бажаним станом та реальною інфраструктурою відбувається через декларативні файли YAML. Така комбінація стала золотим стандартом для розгортання масштабованих систем ML.

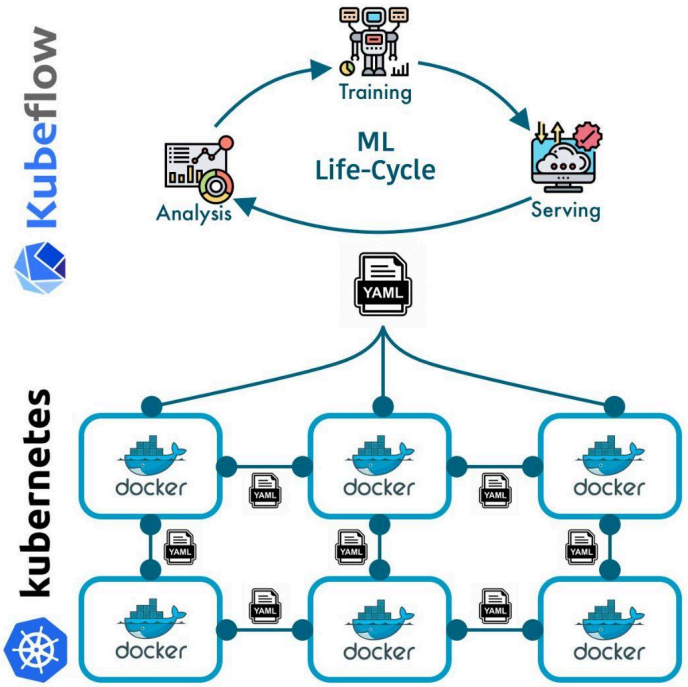

Рис. 1.3. Зв’язок між життєвим циклом ML, маніфестами YAML та середовищем оркестрації Kubernetes

1.4. Еволюція практик розгортання

Традиційні конвеєри CI/CD стали революцією в розробці програмного забезпечення, автоматизувавши процеси збірки, тестування та розгортання. Проте, у контексті Kubernetes-орієнтованих систем класичний підхід до розгортання на основі проштовхування, відомий як “push-based”, демонструє суттєві недоліки [22]. У цій моделі конвеєр CI отримує прямий адміністративний доступ до API Kubernetes і в кінці своєї роботи імперативно виконує команди, такі як “kubectl apply” або “helm upgrade”, щоб відправити зміни в кластер.

Таким способом створюється велика загроза в безпеці, при якій система повинна зберігати високопривілейовані облікові дані доступу до кластера, що робить її привабливою ціллю для атак. Інша проблема виникає тоді, коли інженер вручну вносить зміни у кластер і конвеєр не знає про цю розбіжність, тобто конфігурація в Git більше не відповідає реальному стану системи. Крім того, сам конвеєр ускладнюється, перетворюючись на набір імперативних скриптів, які важко підтримувати та масштабувати.

Для вирішення описаних вище проблем була розроблена еволюційна методологія GitOps. Її ідея полягає в тому, що репозиторій Git є “єдиним джерелом правди” не лише для коду, але й для всієї інфраструктури та конфігурацій додатків [27]. На рисунку 1.4 показано, як Git стає центральним, нескінченним циклом, який об’єднує процеси розробки та операцій.

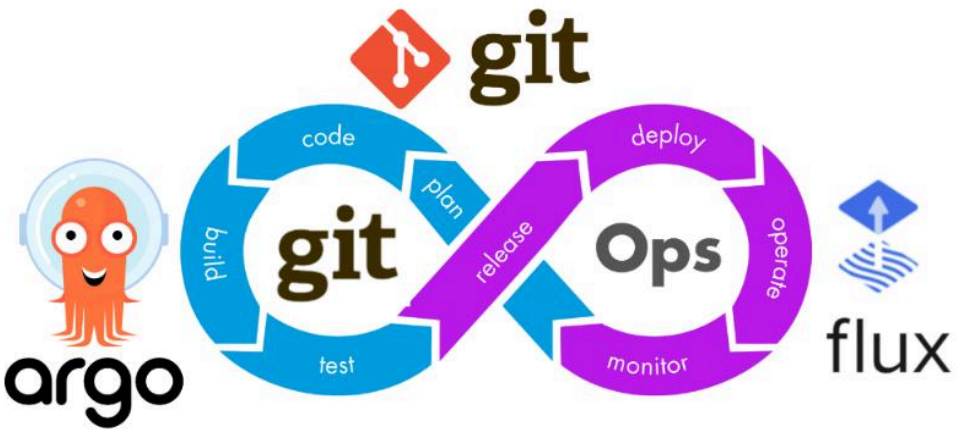

Рис. 1.4. Концептуальна схема GitOps

Вся інфраструктура описується декларативно у вигляді файлів YAML і зберігається у Git. Замість того, щоб конвеєр відправляв зміни в кластер, GitOps впроваджує модель “pull-based”, де спеціальний агент-оператор, наприклад Argo CD, що працює всередині Kubernetes, бере на себе роль синхронізатора.

Як показано на рисунку 1.5, GitOps чітко розділяє обов’язки. Верхній потік відповідає за код додатку, коли розробник робить push у репозиторій додатку, система CI збирає та тестує код, публікуючи артефакт у Container Registry. Нижній потік GitOps відповідає за стан інфраструктури, в якому конвеєр лише оновлює репозиторій стану, змінюючи, наприклад, тег образу в файлі YAML.

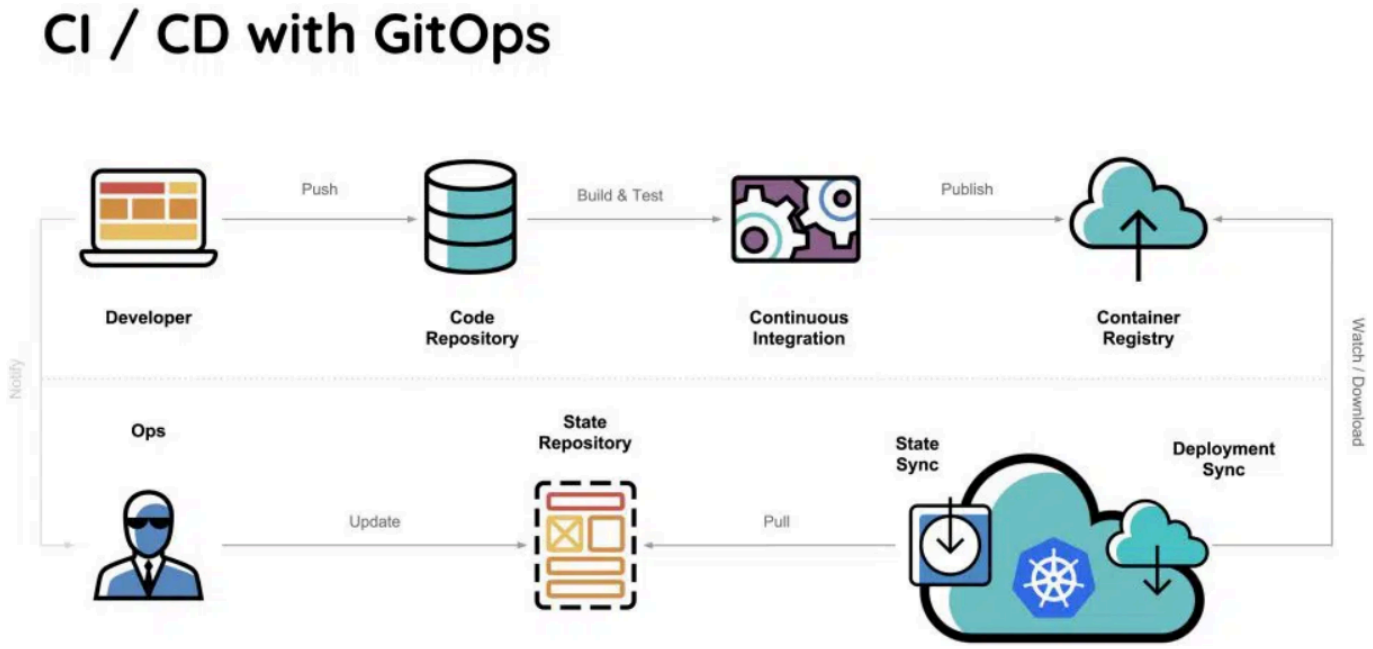

Рис. 1.5. Архітектура CI/CD з GitOps з розділенням потоків коду та конфігурації

Оператор Argo CD всередині Kubernetes постійно стежить за цим репозиторієм стану і витягує будь-які зміни, автоматично приводячи кластер у відповідність до декларативного стану в Git. В такому випадку конвеєр втрачає доступ до Kubernetes, підвищуючи безпеку, а будь-який “дрейф конфігурації” миттєво виявляється і виправляється Argo CD та забезпечує надійну систему.

1.5. Вимоги до безпеки та спостережності у мікросервісних архітектурах

Попри те, що архітектура мікросервісів вирішує проблеми масштабування та незалежного розгортання, вона водночас створює дві нові проблеми, які були відсутні у монолітних системах, це – мережева безпека та складність моніторингу. Коли додаток розбивається на десятки розподілених сервісів, те, що раніше було безпечним викликом функції всередині одного процесу, перетворюється на мережевий запит, який за замовчуванням є незахищеним.

У стандартній конфігурації Kubernetes трафік між сервісами часто є незашифрованим і неавтентифікованим. Будь-який сервіс, який отримав доступ до кластера, може безперешкодно прослуховувати трафік або навіть видавати себе за інший сервіс. Для вирішення цієї проблеми недостатньо звичайного TLS, як на веб-сайтах, де клієнт перевіряє сервер. Необхідна взаємна автентифікація mTLS, тобто підхід “нульової довіри”, при якому обидва сервіси повинні надати та верифікувати валідні сертифікати, перш ніж між ними буде встановлено захищений сеанс (див. рис. 1.6).

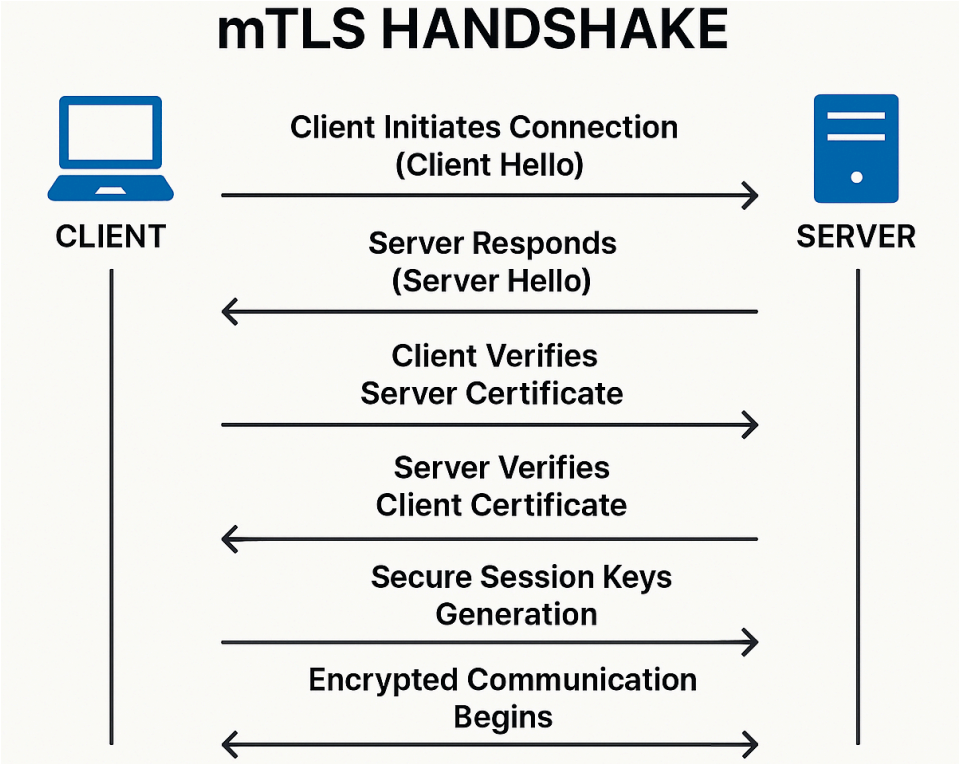

Рис. 1.6. Процес “рукостискання” mTLS

Друга важлива проблема – це спостережність. У мікросервісній архітектурі один запит користувача може викликати ланцюжок з декількох інших прихованих сервісів. Якщо запит стає повільним або повертає помилку, тоді знайти вузьке місце стає неможливо. Виникає потреба у вимірюванні для кожного сервісу в системі так званих “золотих сигналів”: рівня успішності, кількості запитів в секунду та часу затримки.

Реалізація mTLS та збору “золотих сигналів” вручну, шляхом додавання складного коду в кожен мікросервіс є складною, неефективною та схильною до помилок. Найкращим варіантом вирішення цих проблем є впровадження інфраструктурного шару під назвою Service Mesh. На рисунку 1.7 показано, як Service Mesh працює шляхом впровадження проксі-контейнера у кожен под додатку. Весь вхідний та вихідний мережевий трафік перехоплюється цим проксі. В такій реалізації, проксі-сервери автоматично встановлюють mTLS-з’єднання між собою та збирають детальну телеметрію з кожного запиту, надсилаючи її на Control Plane для подальшого аналізу.

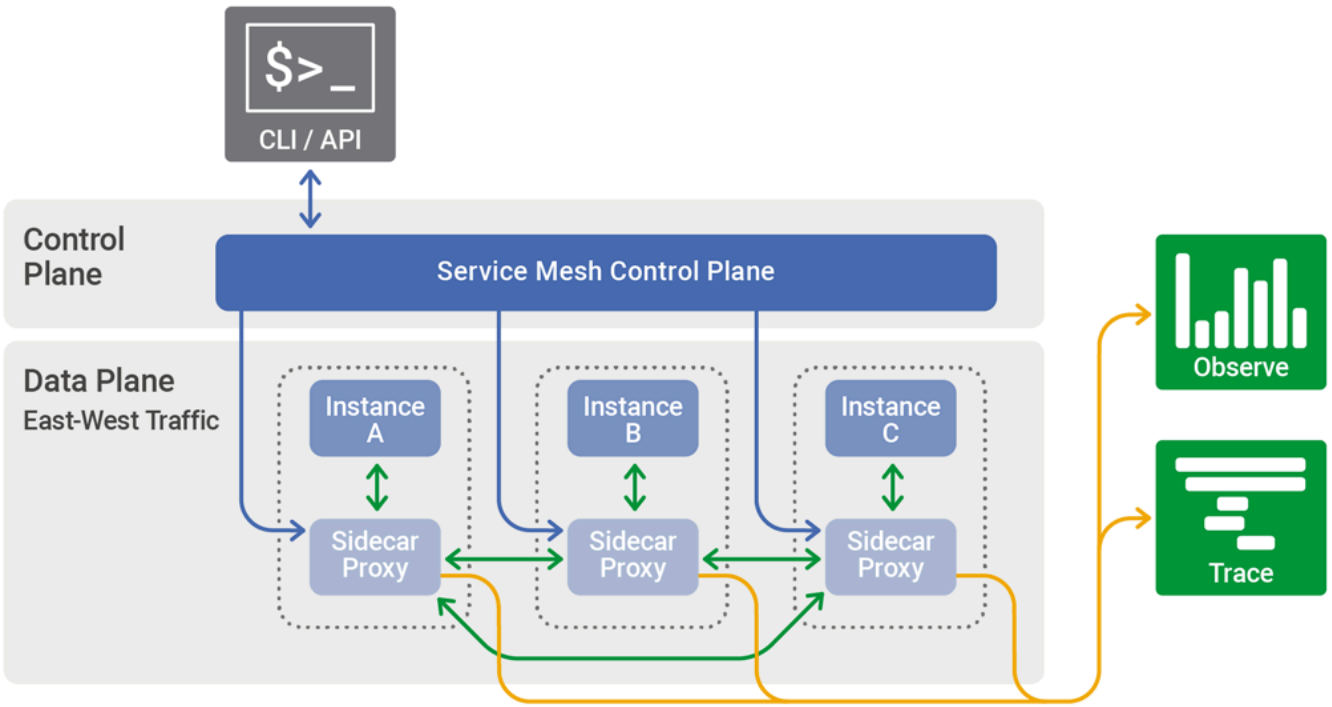

Рис. 1.7. Архітектура Service Mesh

1.6. Обґрунтування вибору інтегрованої платформи MLOps

Провівши у цьому розділі аналіз проблематики життєвого циклу систем машинного навчання, методології MLOps та сучасних архітектурних підходів, можна чітко сказати, що створення надійної, масштабованої та безпечної системи машинного навчання є комплексною задачею. Використання окремих, неінтегрованих інструментів може створити ризики безпеки та призвести до “дрейфу конфігурації”. Розгортання мікросервісів у Kubernetes без Service Mesh залишить їх вразливими, а фокус лише на інструментах ML без інженерних практик DevOps не вирішить проблем якості коду та відтворюваності оточення.

Тобто, для побудови ефективної системи, здатної вирішити всі вищезазначені проблеми, нам недостатньо застосувати один інструмент. Для реалізації комплексної, інтегрованої платформи MLOps, яка би забезпечила наскрізну автоматизацію життєвого циклу моделі, нам необхідно вирішити наступні завдання дослідження:

1) синтезувати архітектуру, що об’єднує парадигми безперервної інтеграції для збірки артефактів, GitOps для декларативного управління станом кластера та Service Mesh для забезпечення безпеки і спостережності;

2) розробити компонент системи, що включає скрипти для автоматизованого тренування моделі, версіонування вхідних даних та відстеження метрик експериментів;

3) реалізувати автоматизований конвеєр, який забезпечує статичний аналіз якості коду, збірку контейнерів та публікацію артефактів без ручного втручання;

4) впровадити підхід GitOps для розгортання, де єдиним джерелом правди про стан інфраструктури виступає репозиторій Git, а синхронізацію виконує автономний оператор всередині кластера;

5) забезпечити захист та моніторинг сервісу шляхом впровадження прозорого шифрування mTLS трафіку та збору на рівні інфраструктури “золотих сигналів” RPS, Latency, Success Rate.

Проєктована система повинна відповідати таким функціональним та нефункціональним вимогам:

– відтворюваність – можливість точно відтворити будь-яку версію моделі, маючи відповідний коміт коду та версію даних;

– атомарність розгортання – оновлення системи повинно відбуватися як єдина транзакція, коли всі компоненти оновлюються успішно, або система залишається у попередньому стабільному стані;

– безпека ланцюга постачання – відсутність довготривалих ключів доступу до кластера у системі CI;

– спостережуваність – наявність метрик продуктивності доступних у реальному часі без необхідності модифікації коду самого додатку ML.

Загальна схема запропонованого підходу зображена на рисунку 1.8. У цій архітектурі розробник взаємодіє лише з GitLab, який виступає єдиною точкою входу для коду, автоматизації та зберігання артефактів. Всередині Kubernetes працює оператор Argo CD, який безпосередньо слідкує за репозиторієм GitLab. Як тільки конвеєр CI/CD оновлює конфігурацію в Git, Argo CD виявляє цю зміну і автоматично оновлює стан кластера Kubernetes, реалізуючи таким чином повний, замкнений та безпечний цикл GitOps.

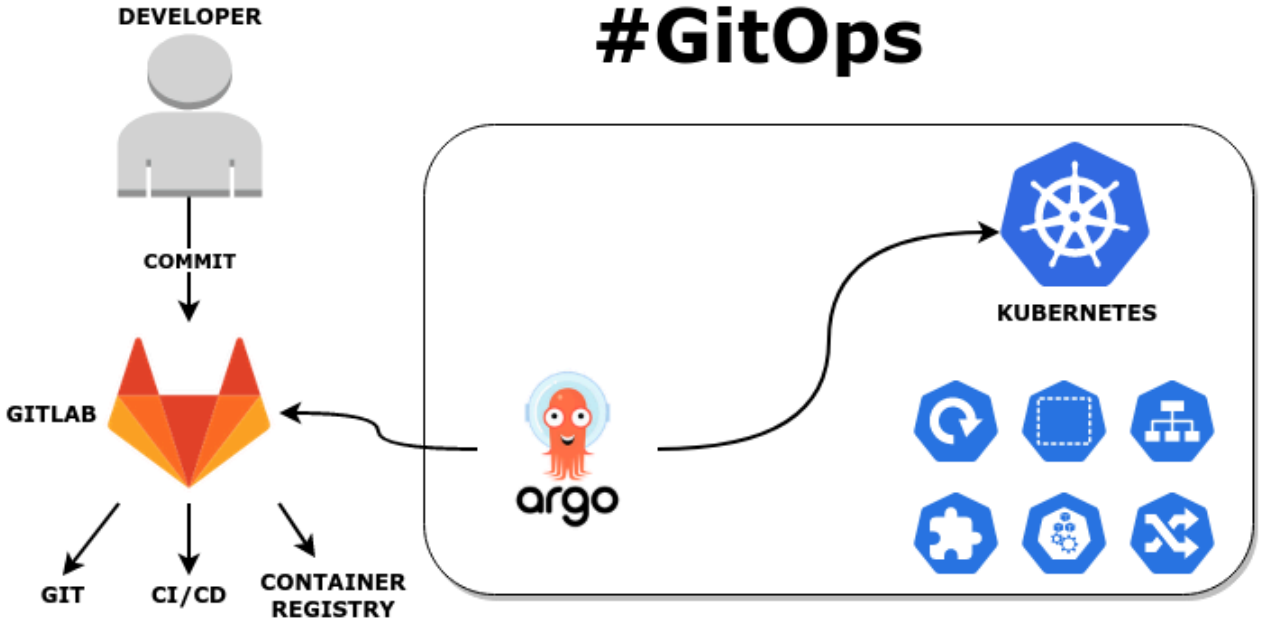

Рис. 1.8. Спрощена архітектурна схема планованої платформи MLOps

Отже, з аналітичної частини можемо зробити висновок, що розробка надійної системи MLOps вимагає виходу за межі традиційних практик DevOps та впровадження багатьох інших спеціалізованих методологій. Провівши аналіз, ми виявили потребу в інтегрованому рішенні, яке б усунуло розрив між експериментальною розробкою моделі та її стабільною експлуатацією. Обґрунтована в цьому розділі архітектура, що базується на поєднанні GitLab CI, Argo CD та Linkerd, дозволяє вирішити такі проблеми галузі як відсутність відтворюваності, вразливість безпеки та недостатню спостережуваність мікросервісів. Сформулювавши завдання та вимоги, ми чітко визначили подальші дослідження. У наступних розділах детально розглянемо теоретичні засади та механізми взаємодії обраних інструментів, також здійснимо їх практичну реалізацію та верифікацію в єдиному автоматизованому конвеєрі.

РОЗДІЛ 2
ТЕОРЕТИЧНА ЧАСТИНА

2.1. Загальна архітектура інтегрованої платформи MLOps

Архітектура розроблюваної системи базується на принципах слабкої зв’язаності компонентів та чіткого розділення відповідальності між середовищем розробки, інтеграції та експлуатації. В основу покладено мікросервісний підхід, де модель машинного навчання інкапсулюється в ізольований контейнер, та методологію GitOps, яка визначає репозиторій Git як “єдине джерело правди” про стан системи.

Її структурна організація базується на взаємодії чотирьох функціональних блоків: середовища розробки, конвеєра безперервної інтеграції, сховищ артефактів та середовища оркестрації.

Основою системи є платформа GitLab, яка є центральним вузлом керування кодом та процесами. Вона містить репозиторії App Repo для вихідного коду та CI/CD конфігурацій, а також Config Repo для декларативних маніфестів Kubernetes, що дозволяє реалізувати принцип розділення відповідальності. GitLab CI/CD забезпечує автоматизацію процесів тестування, тренування моделі та збірки контейнерів.

Згенеровані артефакти зберігаються у спеціально відведених для них сховищах. Образи Docker потрапляють у Container Registry, а версіоновані набори даних та моделі – у віддалене сховище DVC Google Drive.

Середовище оркестрації реалізовано на базі кластера Kubernetes. У ньому функціонують Argo CD Controller, який реалізує підхід GitOps шляхом синхронізації стану кластера з Config Repo, та Linkerd, який забезпечує використання mTLS та спостережуваність мережевого трафіку між мікросервісами. Така архітектура ізолює розробника від прямого втручання в середовище, зводячи його взаємодію до операцій з репозиторієм.

На рисунку 2.1 представлена загальна структура системи, у якій описується високорівнева взаємодія компонентів.

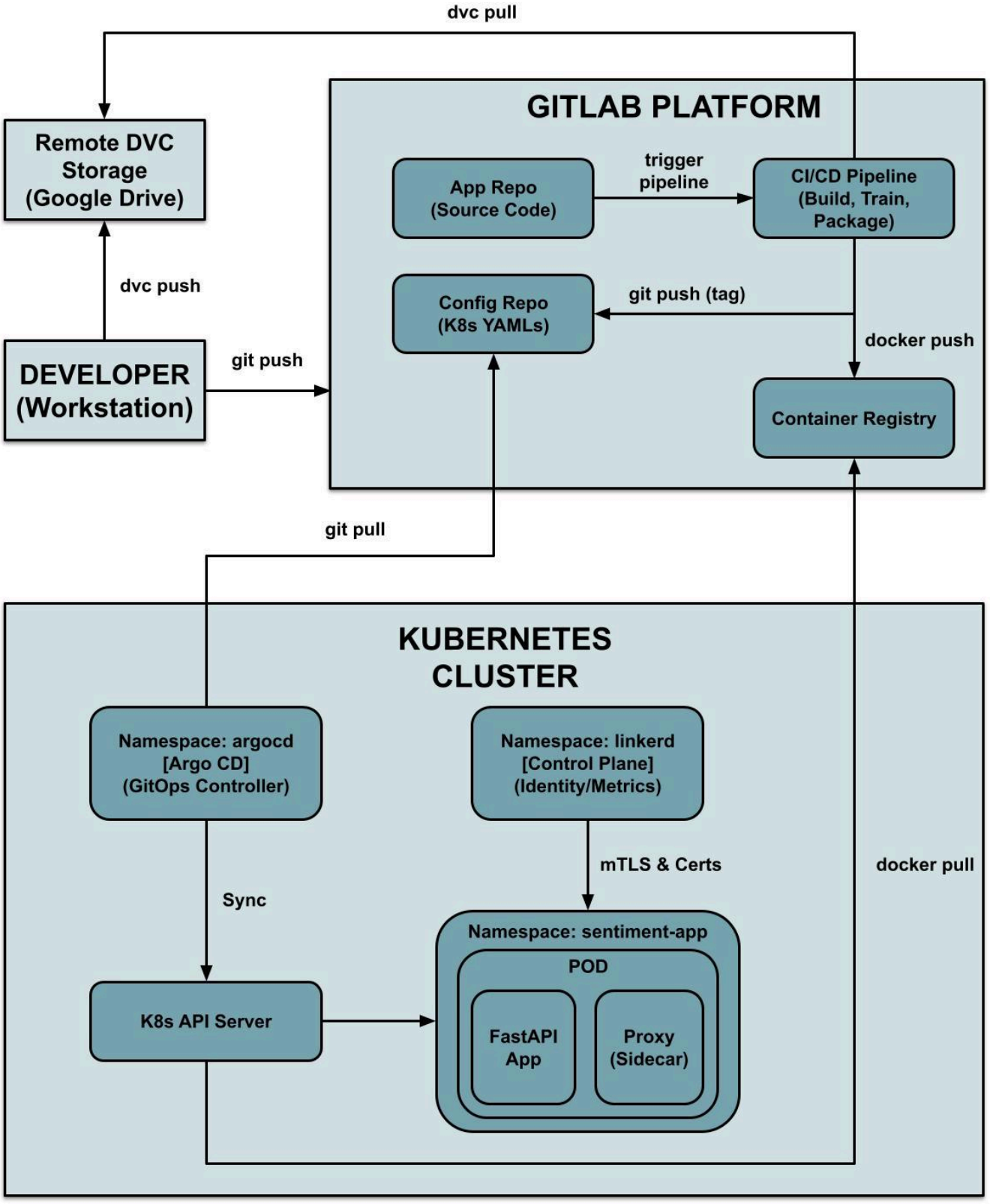

Рис. 2.1. Архітектура платформи MLOps

2.2. Обґрунтування вибору інструментів оркестрації та інфраструктури

В цьому підрозділі розглянемо детально вибір інфраструктурних компонентів, з яких формуватиметься основа платформи MLOps. Для забезпечення відтворюваності, масштабованості та надійності розгортання було обрано стек технологій, що складається з платформи контейнеризації Docker, оркестратора Kubernetes та інфраструктурного інструменту Terraform. Вони допоможуть вирішити проблему “дрейфу конфігурації” та ізолюють середовище виконання.

2.2.1. Платформа контейнеризації Docker та оркестратор Kubernetes

Для забезпечення відтворюваності середовища виконання у роботі обрано платформу контейнеризації Docker, яка упаковує сервіс FastAPI разом з моделлю та усіма залежностями у єдиний, ізольований контейнер. В такому випадку модель, яка натренована в середовищі CI, буде ідентичною тій, що запускається у виробничому середовищі, забезпечуючи повну відтворюваність процесу MLOps.

Для керування контейнерами у великих масштабах використаємо Kubernetes – платформу оркестрації, яка стане “операційною системою” для контейнерів Docker. На рисунку 2.2 показано, як Kubernetes автоматично розподіляє Pods між вузлами кластера, забезпечуючи самовідновлення, масштабування та поступове оновлення.

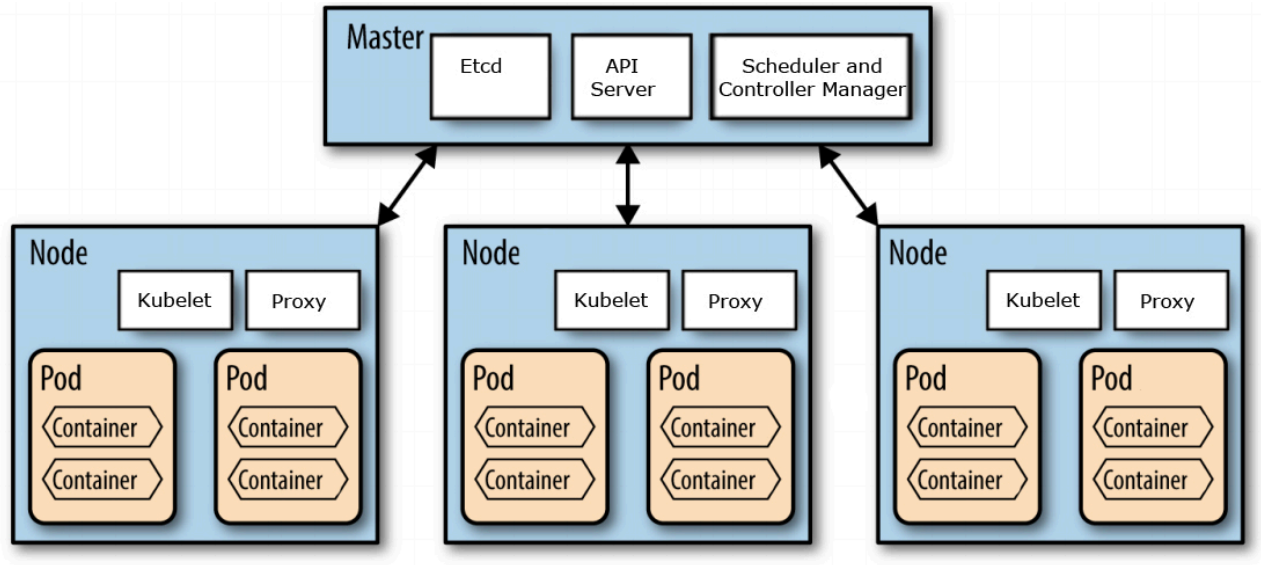

Рис. 2.2. Схема роботи Kubernetes із Docker

2.2.2. Інструмент автоматизації інфраструктури Terraform

Для забезпечення базових компонентів кластера будемо використовувати методологію IaC, реалізовану за допомогою інструменту Terraform. Terraform здатний керувати інфраструктурою декларативно, використовуючи уніфіковану мову конфігурації HCL, а також підтримує провайдерів для Kubernetes та Helm [19].

Завданням Terraform є підготування чистого кластера Kubernetes до роботи, встановлення в нього системних операторів, необхідних для функціонування платформи MLOps. На рисунку 2.3 представлено, як Terraform взаємодіє з API кластера для розгортання двох важливих підсистем:

1) Argo CD – GitOps-контролер, який надалі візьме на себе відповідальність за розгортання прикладного програмного забезпечення;

2) Linkerd – Control Plane сервісної сітки, що забезпечить безпеку та моніторинг.

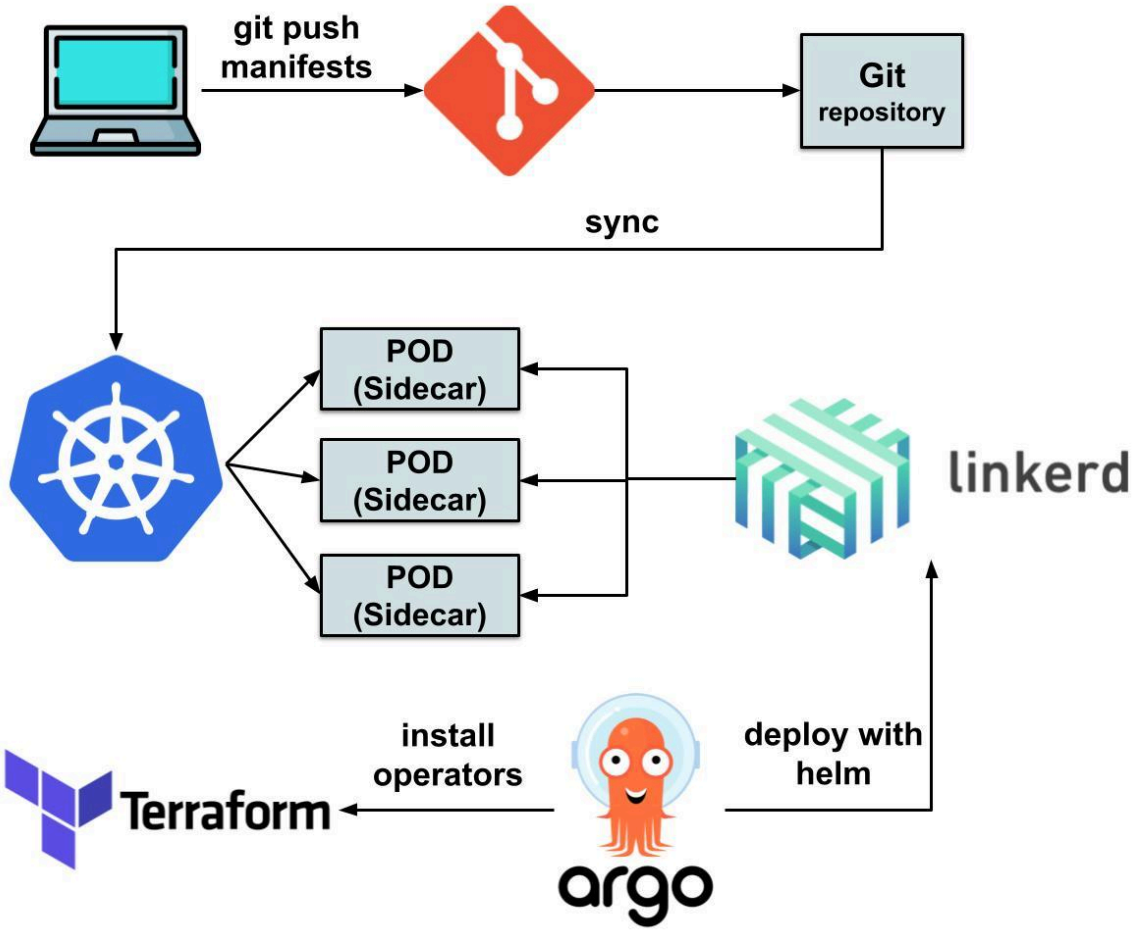

Рис. 2.3. Схема розгортання компонентів та операторів за допомогою Terraform

Головна перевага Terraform – декларативний підхід. Замість написання імперативних скриптів, у файлі main.tf описується бажаний кінцевий стан системи. Terraform автоматично визначає необхідні дії для досягнення цього стану, керує залежностями та зберігає стан інфраструктури у файлі terraform.tfstate. Так можна легко відтворювати середовище, вносити зміни та уникати конфігураційних помилок.

2.2.3. Мова розмітки YAML у конфігураційному управлінні

В основі управління конфігураціями в середовищі Kubernetes лежить YAML – мова серіалізації даних з декларативним підходом. На відміну від імперативних команд, маніфести YAML описують бажаний кінцевий стан системи, роблячи конфігурацію більш зрозумілою для людини, легкою для читання та ідеальною для версіонування у репозиторіях Git.

В даній роботі ми використаємо YAML для опису всіх об'єктів кластера. Першим маніфестом, який ми опишемо, є deployment.yaml, який визначає стратегію розгортання сервісу ML. У ньому декларуються наступні речі:

- специфікація подів – вказуємо образ контейнера Docker, необхідні порти та змінні середовища;
- масштабування – параметром replicas фіксуємо необхідну кількість копій сервісу для забезпечення відмовостійкості;
- інтеграція з Service Mesh – через механізм анотацій у файлі YAML надаємо інструкція контролеру Linkerd автоматично впровадити sidecar-проксі в кожен под, не змінюючи код самого додатку.

Іншим маніфестом є service.yaml, який реалізує необхідний рівень мережевої абстракції над динамічною інфраструктурою подів. Він визначає стабільну віртуальну IP-адресу та правила маршрутизації трафіку до динамічних подів, зіставляючи зовнішній порт з внутрішнім портом контейнера. Таким способом Argo CD автоматично відстежує розбіжності між кодом у репозиторії та станом у кластері, забезпечуючи синхронізацію інфраструктури.

2.3. Огляд засобів автоматизації CI/CD та реалізації GitOps

2.3.1. Платформа GitLab та архітектура CI-ранерів

В якості ядра автоматизації в даній роботі будемо використовувати платформу GitLab, яка буде нам потрібна не лише як система контролю версій, але і як повноцінний інструмент оркестрації процесів CI/CD. Архітектура GitLab CI базується на клієнт-серверній моделі, де процеси управління пайплайнами відділені від процесів їх безпосереднього виконання.

Центральний сервер GitLab Instance відповідає за зберігання коду, управління користувачами та планування завдань. Однак саме виконання коду, що складається з компіляції, тестування та збірки Docker-образів, делегується окремим агентам – GitLab Runners. Як показано на рисунку 2.4, сервер розподіляє завдання між доступними ранерами, які можуть бути розміщені на різних серверах або навіть у різних хмарних середовищах.

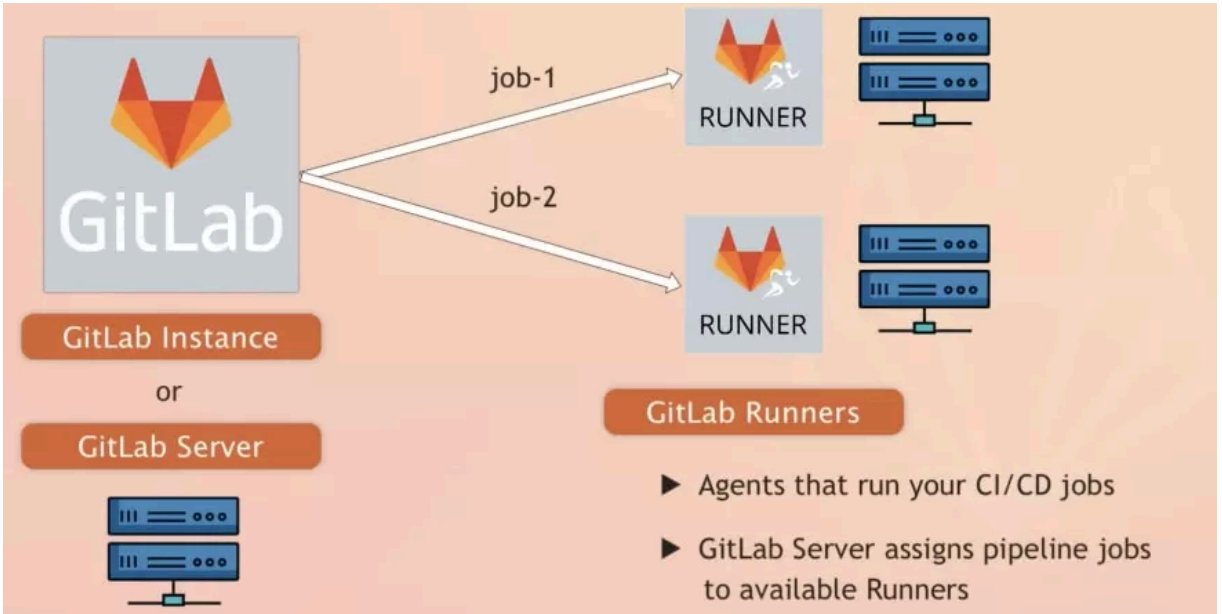

Рис. 2.4. Розподіл завдань між GitLab Server та GitLab Runners

У рамках даної роботи будемо створювати гібридну архітектуру з використанням локального виконавця, встановленого на робочій станції. Таке рішення зумовлене необхідністю мережевої взаємодії між CI-процесами та

інфраструктурними компонентами Minikube та SonarQube, які недоступні для публічних хмарних ранерів. Локальний ранер буде налаштований з використанням виконавця типу docker. Тобто кожне завдання конвеєра буде запускатися в чистому, ізольованому контейнері Docker, що виключить можливість виникнення конфліктів залежностей між різними етапами.

Для управління маршрутизацією завдань у системі використовуються теги та змінні середовища. Теги дозволяють створювати жорстку прив'язку завдань до конкретних ранерів. Під час реєстрації локального ранера йому присвоюється тег docker, і цей же тег вказується у файлі .gitlab-ci.yml для кожного завдання. Це наказує GitLab Server відправляти завдання на локальний ранер.

Змінні середовища використовуються для забезпечення безпеки. Чутливі дані, такі як токени доступу до SonarQube, паролі до Container Registry та приватні SSH-ключі для GitOps, не зберігаються у коді. Вони завантажуються у налаштування проєкту GitLab і передаються ранеру як змінні оточення лише під час виконання завдання. Це дозволяє дотримуватися принципів безпечної розробки та уникнути витоку секретів через репозиторій.

2.3.2. GitOps-оператор Argo CD

Для реалізації Continuous Delivery було обрано інструмент Argo CD – декларативний GitOps-контролер для Kubernetes. На відміну від традиційних систем розгортання, які ініціюють зміни ззовні, реалізуючи “push-модель”, Argo CD функціонує як агент всередині кластера, реалізуючи “pull-модель”. Його головне завдання – забезпечити відповідність фактичного стану кластера бажаному стану, який описується у Git-репозиторії конфігурацій.

На рисунку 2.5 опишемо загальну схему розгортання із використанням Argo CD. Розділимо процес на два логічні потоки. Верхній потік завершується тим, що пайплайн збирає образ Docker, відправляє його в реєстр та оновлює файл маніфесту в репозиторії конфігурацій. В цей момент вступає в дію Argo CD, безперервно слідкуючи за цим репозиторієм і, виявивши зміни, наприклад, новий тег образу, ініціює процес синхронізації маніфестів із цільовими середовищами.

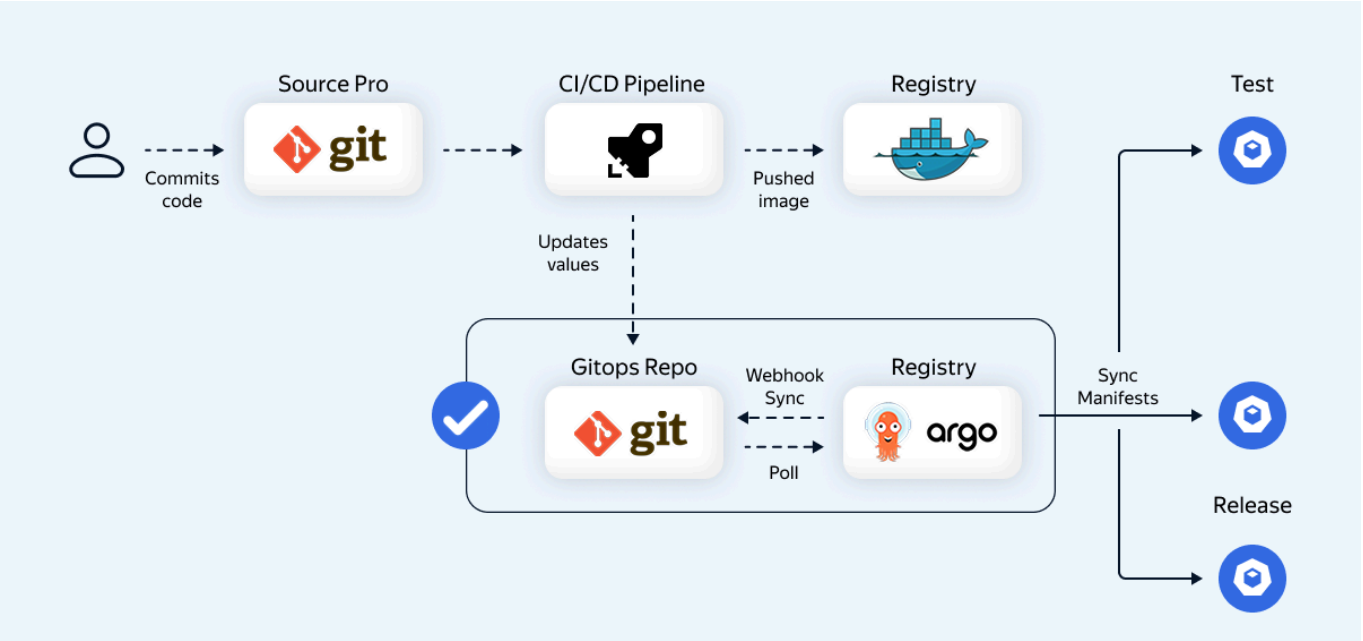

Рис. 2.5. Взаємодія CI-процесу та GitOps-синхронізації під управлінням Argo CD

Надійність Argo CD базується на циклі узгодження. Його суть наступна: контролер періодично порівнює поточну конфігурацію ресурсів у Kubernetes із версією, що зберігається в Git, і, у випадку виявлення розбіжності, змінює статус додатка на “OutOfSync”.

В даній роботі ми також налаштовуємо Argo CD з механізмом самовідновлення. Тепер система буде не просто сигналізувати про дрейф конфігурації, а й автоматично виправляти його. Якщо випадково видалиться сервіс або зміниться кількість реплік моделі через консоль kubectl, Argo CD миттєво виявить, що “живий” стан не відповідає Git-стану, і примусово поверне кластер до конфігурації, зафіксованої в репозиторії.

2.3.3. Система статичного аналізу коду SonarQube

У нашій системі ML буде інтегровано статичний аналізатор SonarQube. Він автоматично оцінює код за показниками надійності, безпеки, супроводжуваності, покриття тестами та дублювання, виявляє помилки і технічний борг до запуску тренування моделі. SonarQube буде розгорнуто як Docker-сервіс і буде використовуватися GitLab Runner на етапі validate, де sonar-scanner аналізуватиме код у директорії вихідного коду та надсилатиме результати на сервер.

Механізм Quality Gates забезпечуватиме контроль якості, задаючи порогові значення, такі як мінімальне покриття тестами або відсутність критичних вразливостей. Якщо код не проходитиме ці пороги, SonarQube поверне помилку й зупинить весь конвеєр CI/CD.

2.4. Програмний стек розробки моделей машинного навчання

Для реалізації самої моделі у даній роботі використано класичний стек технологій, що став стандартом в індустрії Data Science та ML. Основний акцент зроблено на використанні мови Python та її потужної екосистеми, яка забезпечує ефективну роботу з даними, швидке прототипування моделей та їх просту інтеграцію з інфраструктурними компонентами.

2.4.1. Мова програмування Python та бібліотеки аналізу даних

В якості основної мови програмування було обрано Python. Це інтерпретована мова високого рівня, яка завдяки своєму лаконічному синтаксису та динамічній типізації дозволяє швидко розробляти алгоритми машинного навчання. Основна перевага – наявність широкого спектру спеціалізованих бібліотек, які значно спрощують процес розробки моделей машинного навчання.

Для реалізації моделі будемо використовувати такі бібліотеки:

– pandas – надає для роботи високорівневі структури даних DataFrame для маніпуляції табличними даними, їх завантаження, очищення та фільтрації [24];

– scikit-learn – реалізує класичні алгоритми машинного навчання, такі як MultinomialNB, інструменти для векторизації тексту та метрики оцінки якості;

– NLTK – використовується для лінгвістичної обробки тексту, токенізації та видалення стоп-слів.

2.4.2. Фреймворк створення веб-сервісів FastAPI

Для перетворення статичного файлу моделі на повноцінний мікросервіс, який буде здатний обробляти HTTP-запити, будемо використовувати веб-фреймворк

FastAPI. На відміну від Flask або Django, FastAPI спроєктований для роботи з асинхронним кодом та базується на сучасних стандартах Python, що забезпечує виняткову продуктивність, порівняну з Node.js та Go [23].

Головні інструменти та можливості, що роблять FastAPI оптимальним вибором для ML-сервісів, представлено на рисунку 2.6.

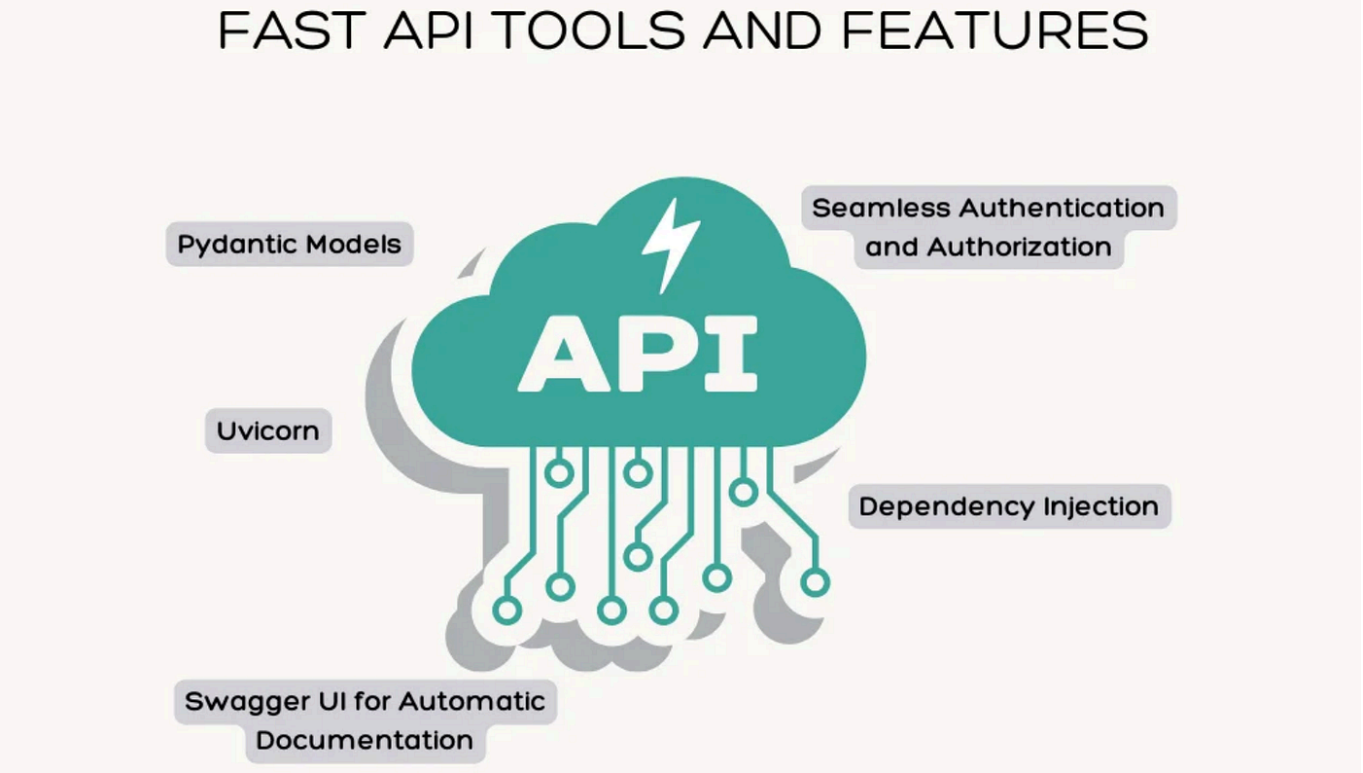

Рис. 2.6. Інструменти та функціональні можливості екосистеми FastAPI

Вибір цього інструменту обґрунтований трьома факторами. Перш за все, це висока продуктивність, що досягається завдяки використанню Uvicorn як ASGI-сервера. Uvicorn може ефективно обробляти тисячі запитів одночасно, а це важливо для швидкості відповіді моделі, яка безпосередньо впливає на користувацький досвід.

Другим фактором є автоматична документація. FastAPI має вбудовану підтримку стандарту OpenAPI, завдяки чому при кожному запуску додатку автоматично генерується інтерактивна документація API, що спрощує процес

розробки та тестування ендпоінтів, дозволяючи миттєво перевіряти функціонал сервісу без необхідності писати окрему документацію вручну.

Третім фактором для надійності ML-сервісу є валідація даних, реалізована через інтеграцію з Pydantic Models. Pydantic дозволяє декларативно описувати структуру вхідних даних за допомогою стандартних типів Python. У нашій роботі це буде реалізовано через клас TextRequest. Якщо клієнт надішле запит із некоректним форматом даних, Pydantic автоматично перехопить це і поверне чітку помилку валідації ще до того, як дані потраплять у модель.

2.4.3. Інструменти DVC та MLflow

Стандартні засоби контролю версій програмного коду виявляються неефективними при роботі з артефактами машинного навчання, зокрема з великими бінарними файлами даних та моделями. Для вирішення цієї проблеми буде використано DVC – інструмент, що розширює можливості Git для версіонування даних. DVC дозволяє зберігати великі датасети у зовнішньому хмарному сховищі, залишаючи у репозиторії Git лише легкі метафайли-вказівники [26]. Ці вказівники містять хеш-суми файлів, які пов'язують конкретну версію коду з конкретною версією даних.

Для управління життєвим циклом самих моделей використовується MLflow, а саме її компонент MLflow Tracking. Процес розробки ML-моделі є ітеративним і вимагає проведення безлічі експериментів з різними гіперпараметрами. MLflow виступає централізованим сервером, який протоколює кожен запуск скрипту тренування. Він зберігає параметри запуску, отримані метрики якості та самі згенеровані артефакти формату .pkl.

2.5. Проєктування системи безпеки та спостережності

Перехід до мікросервісної архітектури призводить до того, що мережа стає найвразливішим та найскладнішим компонентом системи. Забезпечення надійності, шифрування трафіку та діагностики помилок у розподіленому середовищі

вимагають впровадження спеціалізованого інфраструктурного шару. У цій роботі систему буде спроєктовано на базі Service Mesh, яка винесе логіку мережевої взаємодії, безпеки та моніторингу за межі коду додатку та реалізує її на рівні платформи.

2.5.1. Архітектура Linkerd Service Mesh

Linkerd є надлегкою реалізацією Service Mesh, яка забезпечує безпеку, надійність та спостережуваність мережевої взаємодії між мікросервісами. Архітектурно система Linkerd розділена на дві логічні складові: Data Plane та Control Plane [28]. Основою Data Plane є використання патерну Sidecar Proxy. На рисунку 2.7 показано, що Linkerd не замінює сервіси додатку, а впроваджує свій мікро-проксі в кожен под кластера. Цей проксі запускається в тому ж мережевому просторі імен, що й основний контейнер додатку.

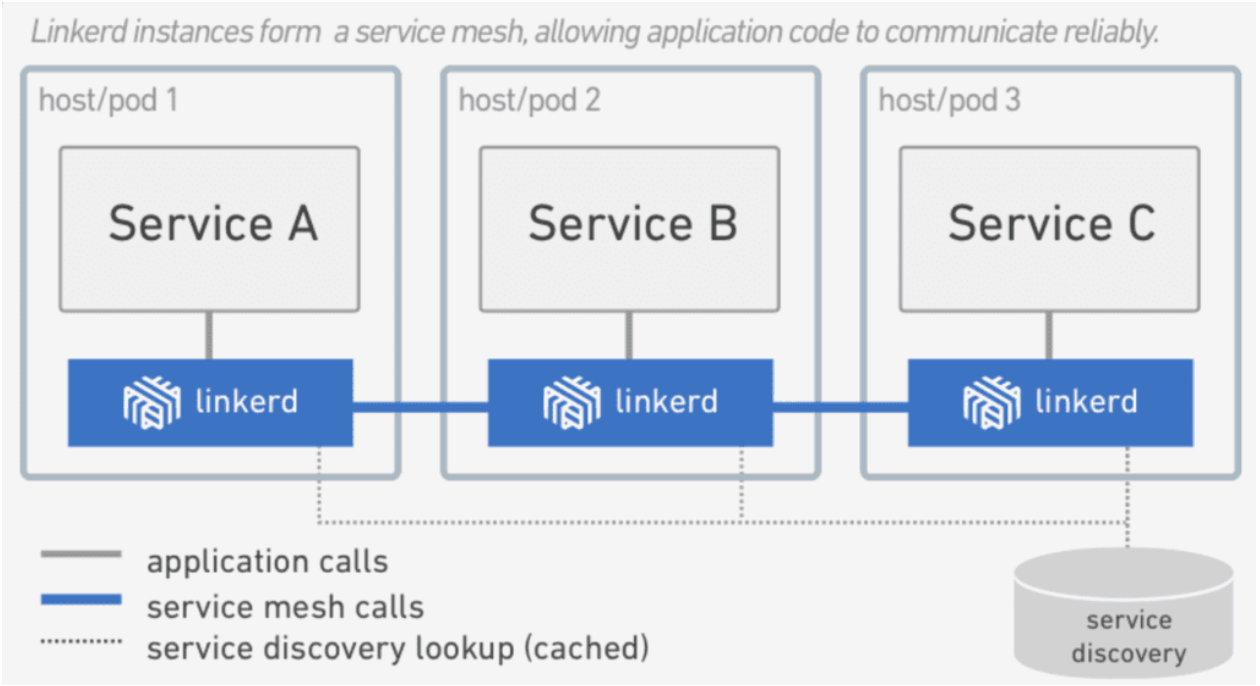

Рис. 2.7. Взаємодія сервісів через проксі Linkerd на шарі Data Plane

Вище відображено механізм прозорого перехоплення трафіку. Коли сервіс А намагається звернутися до сервісу Б, він робить це стандартним способом. Але

локальний проксі перехоплює цей запит і перетворює його на "service mesh call", виконуючи наступні дії:

– автоматично загортає трафік у шифрований тунель, використовуючи сертифікати для взаємної автентифікації;

– обирає найшвидший екземпляр цільового сервісу;

– вимірює час виконання запиту для моніторингу.

Друга складова Control Plane – це набір системних сервісів у просторі імен linkerd, яка керує всією цією мережею. До цієї складової входять:

– Identity Service – діє як внутрішній центр сертифікації, автоматично оновлюючи TLS-сертифікати проксі;

– Destination Service – виявляє сервіси та виконує маршрутизацію;

– Proxy Injector – автоматично додає контейнери проксі у нові поди при їх створенні.

2.5.2. Механізми збору метрик Prometheus та Grafana

У розроблюваній системі роль перетворення телеметричних даних на корисну інформацію будуть виконувати інструменти з відкритим кодом Prometheus та Grafana.

Prometheus виступає в ролі централізованого сховища метрик. Це база даних часових рядів, яка працює принципом періодичного опитування сервером визначених цілей та збирання з них поточних показників. Ми налаштуємо Prometheus на збір даних з інфраструктурного та прикладного рівнів. На інфраструктурному рівні будуть збиратися такі метрики як обсяг трафіку та статус з'єднань mTLS з кожного sidecar-проксі Linkerd. На прикладному рівні будуть збиратися метрики безпосередньо з ML-сервісу, що дозволяє аналізувати специфічні показники роботи Python-додатка.

Для оцінки стану системи використовується методологія “золотих сигналів”, розроблена Google SRE. Вона дозволяє діагностувати здоров'я мікросервісу, аналізуючи його поведінку як “чорної скриньки”, без необхідності знати деталі внутрішньої реалізації.

Візуалізація зібраних даних буде реалізована за допомогою Grafana. Дана платформа для аналітики підключається до Prometheus. У проєкті використаємо налаштовані дашборди Linkerd, які в реальному часі будуватимуть графіки "золотих сигналів" для кожного розгорнутого сервісу. Так ми зможемо миттєво виявляти деградацію продуктивності або доступності ML-моделі після розгортання.

2.6. Архітектура Linkerd та мережевої взаємодії в Kubernetes

В цьому розділі розглянемо схему структурної організації Service Mesh на базі Linkerd, яка чітко поділяється на Control Plane та Data Plane, а також потоки трафіку та телеметрії (див. рис. 2.8).

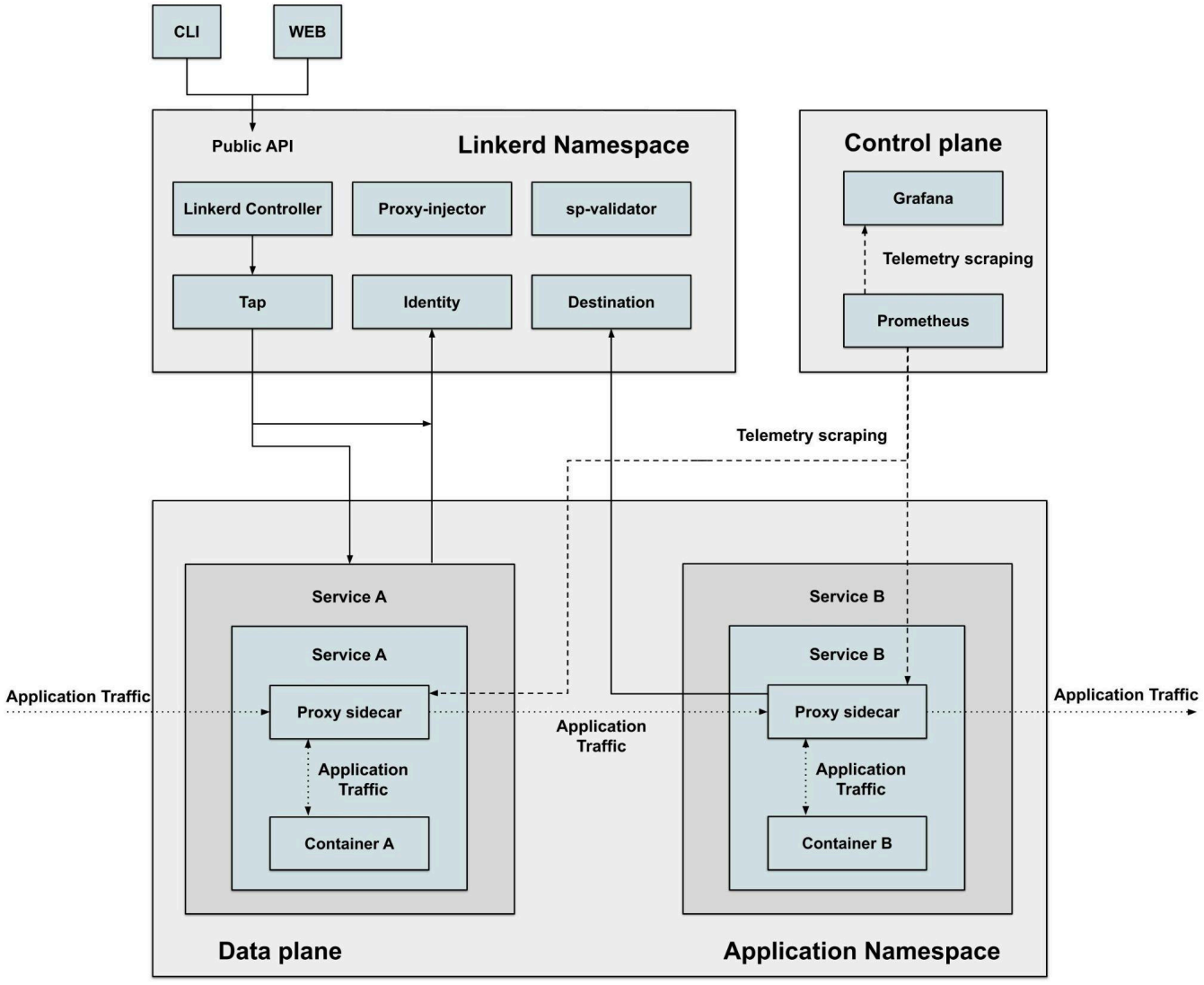

Рис. 2.8. Схема Linkerd та мережевої взаємодії в Kubernetes

У нижній частині схеми розміщена площина даних. У ній показано два сервіси, всередині кожного поду яких, поруч із основним контейнером додатку, працює легковаговий Proxy sidecar. Стрілками показано шлях запиту, по якому Container A хоче звернутися до Service B. Він відправляє звичайний запит, а локальний проксі перехоплює цей вихідний трафік. Проксі сервісу A встановлює захищене mTLS-з'єднання з проксі сервісу B, трафік між ними шифрується. Потім проксі сервіс B отримує запит, розшифровує його і передає локально своєму контейнеру B.

У верхній правій частині схеми знаходиться підсистема спостережуваності. Prometheus працює в режимі збирання, а пунктирні лінії показують, що він періодично опитує кожен проксі-сервер у Data Plane, збираючи метрики RPS, затримки та успішності. У свою чергу, Grafana візуалізує ці дані, надаючи дашборди.

На цій архітектурній схему добре видно, як Linkerd виносить складну логіку мережевої безпеки та моніторингу з коду додатку на інфраструктурний рівень і забезпечує прозорість та надійність комунікації в мікросервісному середовищі.

2.7. Огляд діаграми класів програмних модулів

На рисунку 2.9 наведено опис архітектурних компонентів системи аналізу тональності. Розглянемо функціональне призначення кожного модуля.

Модуль FastAPI_App є точкою входу в API, що обробляє HTTP-запити та повертає JSON-відповіді. Він відповідає за ініціалізацію моделі в пам’яті при старті, маршрутизацію даних для передбачення та збір технічних метрик, виступаючи сполучною ланкою між клієнтом і ядром ML.

TextRequest – Pydantic-модель, що діє як фільтр безпеки для вхідних даних. Вона гарантує, що система отримує лише коректний текст, автоматично відхиляючи пусті запити або неправильні типи даних ще до того, як вони потраплять до бізнес-логіки.

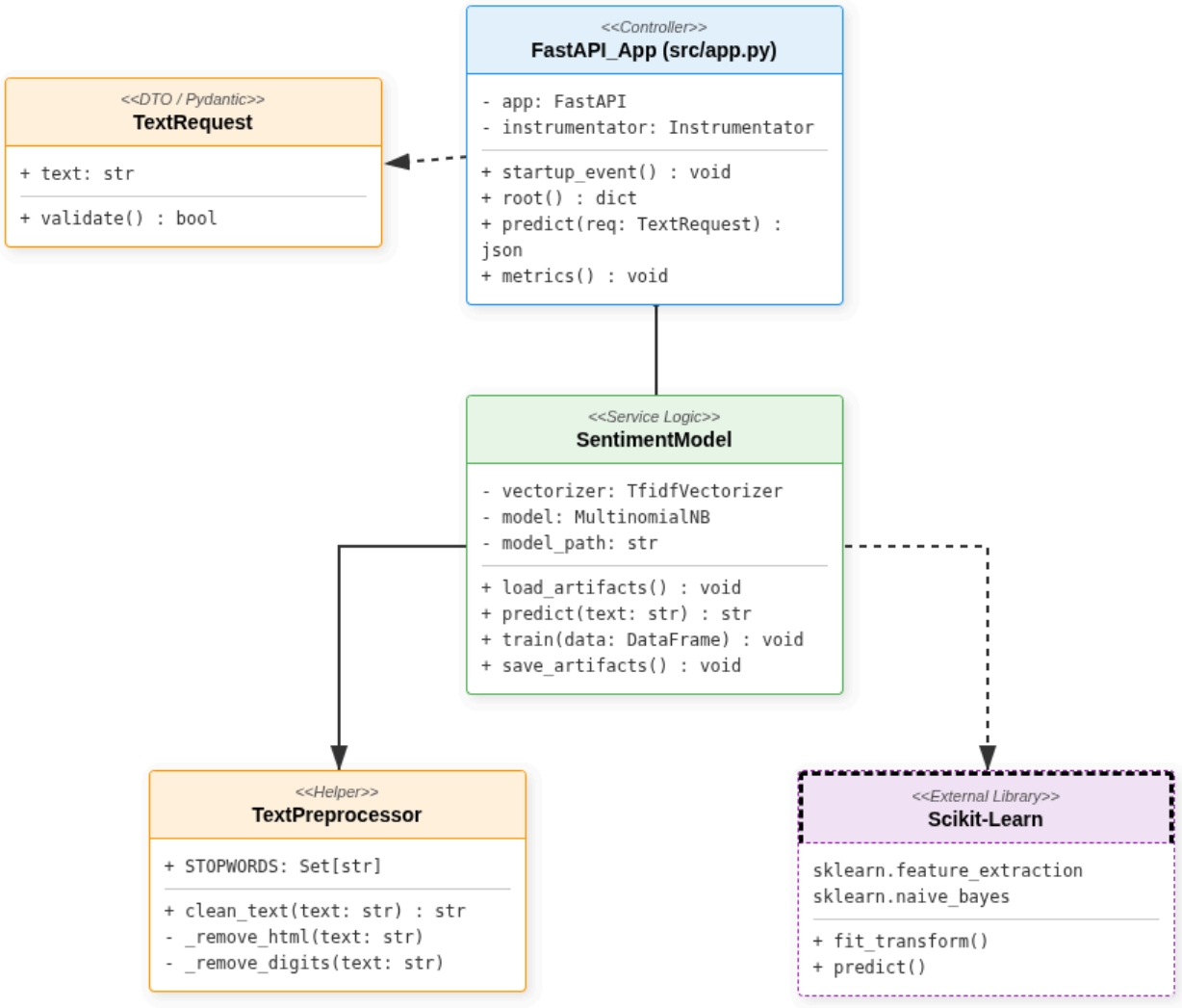

Рис. 2.9. Діаграма класів програмних модулів

SentimentModel – основний клас ядра ML, який керує повним життєвим циклом моделі. Він інкапсулює процеси тренування, збереження та завантаження артефактів і виконання безпосередніх прогнозів тональності за допомогою векторизатора та класифікатора.

TextPreprocessor слугує для нормалізації “сирих” даних, а саме видалення HTML-тегів, спецсимволів та приведення тексту до нижнього регістру. Він використовується під час навчання та роботи API і забезпечує єдиний стандарт обробки даних для стабільної точності прогнозів.

Scikit-Learn – зовнішня залежність, яка надає готові реалізації математичних алгоритмів TF-IDF та Naive Bayes. Це фундамент для обчислень, які виконує клас SentimentModel.

2.8. Блок-схеми алгоритмів роботи програмних модулів

Нарешті, розглянемо логіку роботи компонентів системи. У цьому підрозділі розкриємо послідовність дій під час навчання моделі та життєвого циклу обробки запитів API і візуалізуємо їх на відповідних блок-схемах, що представлені на рисунку 2.10.

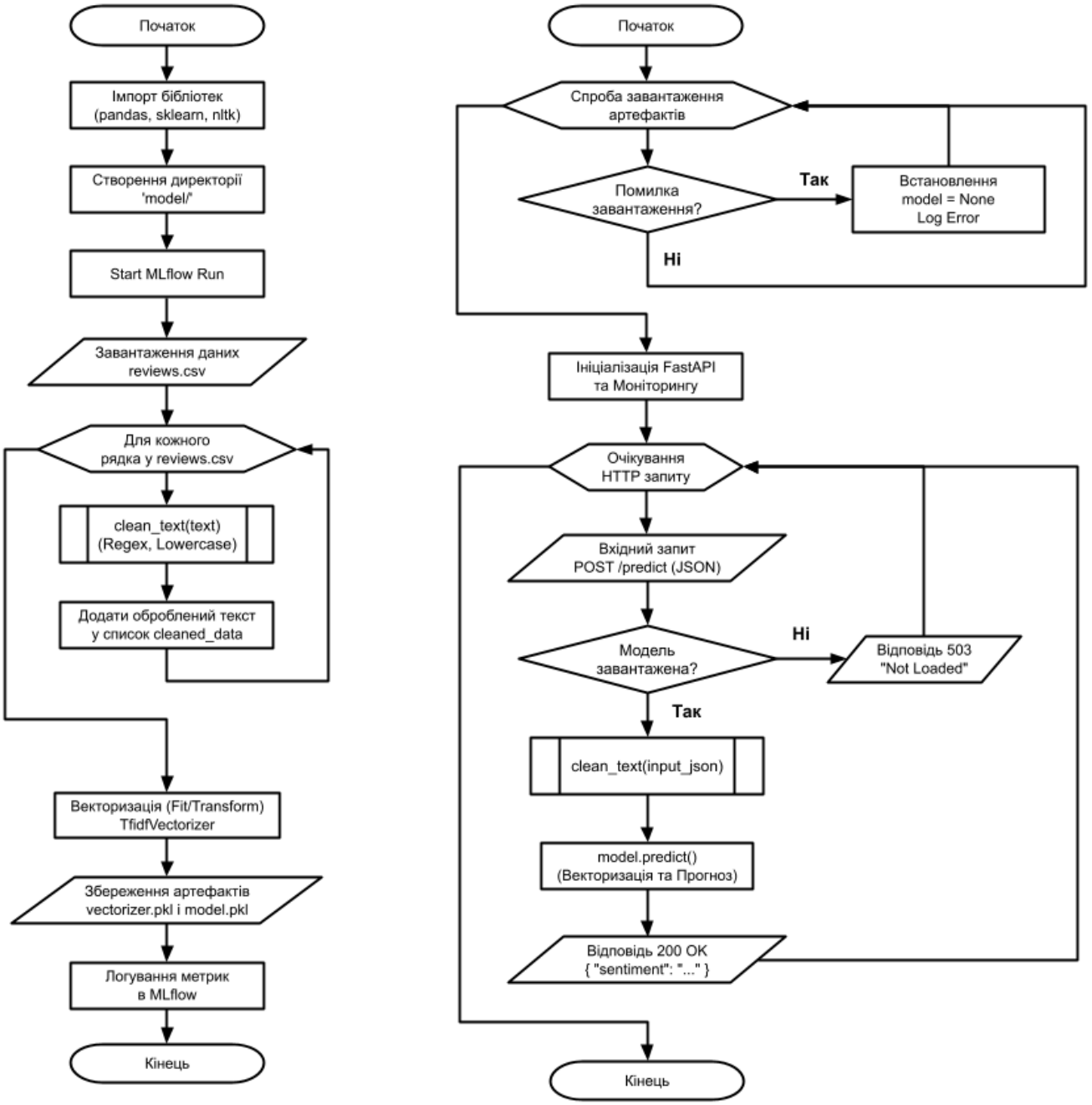

Рис. 2.10. Блок-схеми алгоритмів навчання моделі та роботи API-сервісу

Розглянемо спочатку алгоритм навчання моделі (див. рис. 2.10, зліва). Процес розпочинається із запуску скрипта, який підключає бібліотеки pandas, sklearn та nltk і створює директорію model/ для майбутніх файлів. Паралельно стартує сесія в системі MLflow для відстеження параметрів експерименту. Потім відбувається завантаження вихідних даних із файлу reviews.csv у структуру DataFrame.

Після цього скрипт послідовно перевіряє наявність необроблених рядків і для кожного запису викликає функцію clean_text. Вона виконує видаляє HTML-теги, спецсимволи та приводить текст до нижнього регістру. Очищені дані додаються до списку cleaned_data, і цей процес триває до повного опрацювання всіх рядків датасету.

На завершальній стадії виконується векторизація підготовленого тексту за допомогою TfidfVectorizer, що перетворює слова на числові масиви. Отримані артефакти зберігаються локально у файли .pkl. Вкінці система логує метрики точності та F1-score в MLflow.

На рисунку 2.10 (справа) показано алгоритм роботи API-сервісу. Життєвий цикл починається із запуску додатка FastAPI. Система одразу намагається завантажити файли моделі та векторизатора в оперативну пам’ять через механізм обробки винятків. У разі успіху компоненти стають активними, якщо ж файли пошкоджені або відсутні – змінній моделі присвоюється значення None, а помилка фіксується в логах (системних журналах). Після цього завершується налаштування додатка та підключаються метрики моніторингу Prometheus.

Сервер переходить у режим очікування на порту 8000. Коли надходить POST-запит на ендпоінт /predict із JSON-даними, система перевіряє, чи коректно завантажена модель. Якщо ініціалізація не відбулася, сервіс повертає клієнту помилку 503, захищаючи систему від збоїв.

У разі успішної перевірки вхідний текст проходить очищення функцією clean_text() і модель генерує прогноз. Результат формується у JSON-відповідь та відправляється клієнту. Після завершення операції сервер повертається до стану очікування нових запитів.

Спроєктована в даному розділі архітектура закладає фундамент для практичної частини роботи. У наступному розділі ми здійснимо реалізацію розробленого рішення: налаштуємо локальне середовище та репозиторії до запуску повного CI/CD-конвеєра, розгорнемо сервіс аналізу тональності та верифікуємо його роботу за допомогою інструментів моніторингу.

РОЗДІЛ 3
ПРАКТИЧНА ЧАСТИНА

3.1. Підготовка середовища та репозиторіїв для циклу MLOps

Перед початком створення системи MLOps налаштуємо середовище в операційній системі. Встановимо необхідні залежності та інструменти потрібним чином, такі як GitLab Runner для запуску конвеєра, ArgoCD для сучасного розгортання додатку і LinkerD для додаткового шару безпеки у системі та моніторингу мережі між сервісами Kubernetes (див. рис. 3.1).

```
[serhii@80LT ~]$ yay -S gitlab-runner
Sync Explicit (1): gitlab-runner-18.5.0-1
warning: gitlab-runner-18.5.0-1 is up to date -- reinstalling
resolving dependencies...
looking for conflicting packages...

Packages (1) gitlab-runner-18.5.0-1

Total Installed Size:  508.18 MiB
Net Upgrade Size:        0.00 MiB

:: Proceed with installation? [Y/n]
```

```
[serhii@80LT ~]$ sudo pacman -S argocd
resolving dependencies...
looking for conflicting packages...

Packages (1) argocd-3.1.9-1

Total Download Size:    40.59 MiB
Total Installed Size:  191.82 MiB

:: Proceed with installation? [Y/n] y
```

```
[serhii@80LT ~]$ curl -sL https://run.linkerd.io/install | sh
Downloading linkerd2-cli-edge-25.10.5-linux-amd64...
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100 79.0M  100 79.0M    0     0  1682k      0  0:00:48  0:00:48 --:--:-- 2543k
Download complete!

Linkerd edge-25.10.5 was successfully installed
```

Рис. 3.1. Встановлення стеку інструментів для реалізації системи MLOps

Далі виконаємо підготовку та налаштування середовища Docker (див. рис. 3.2). Командою “systemctl enable --now docker” запустимо фонову службу Docker та налаштуємо її на автоматичний старт при кожному завантаженні системи. Командою “usermod -aG docker $USER” надамо поточному користувачу необхідні привілеї, додаючи його до групи docker. Таким чином можна буде запускати команди docker без sudo, що важливо для роботи Minikube та GitLab Runner.

Підготовлений Docker використаємо для розгортання сервера аналізу коду SonarQube. Командою docker run завантажимо офіційний образ sonarqube:lts-community і запустимо його у фоновому контейнері з іменем sonarqube. Далі “прокинемо” веб-інтерфейс SonarQube зсередини контейнера на порт 9000 локальної машини. Аналізатор коду стане доступний за адресою http://localhost:9000.

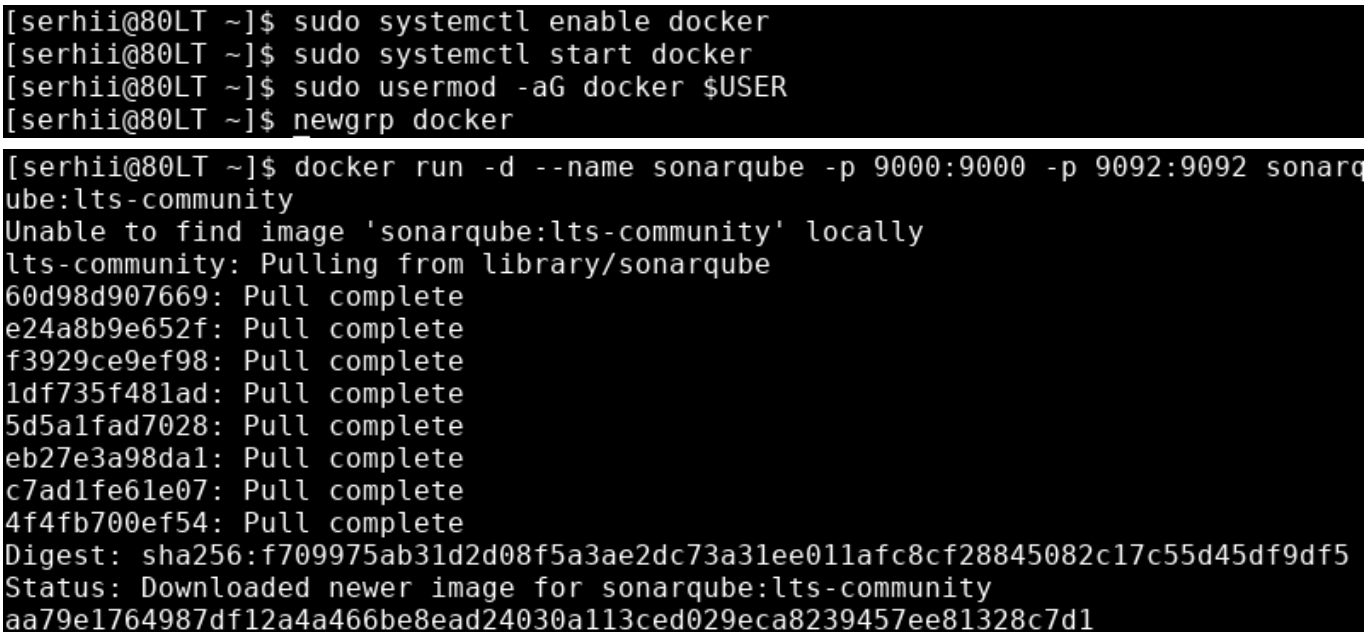

Рис. 3.2. Налаштування середовища Docker та SonarQube

Для реалізації раніше згаданого принципу GitOps створимо два окремі приватні репозиторії у системі GitLab. Перший, sentiment-app-repo (див. рис. 3.3), є репозиторієм для додатку, що призначений для зберігання Python-скриптів тренування, FastAPI-сервера, інструкцій для контейнеризації Dockerfile, файлів версіонування даних DVC, а також файлу .gitlab-ci.yml, який описує конвеєр CI/CD.

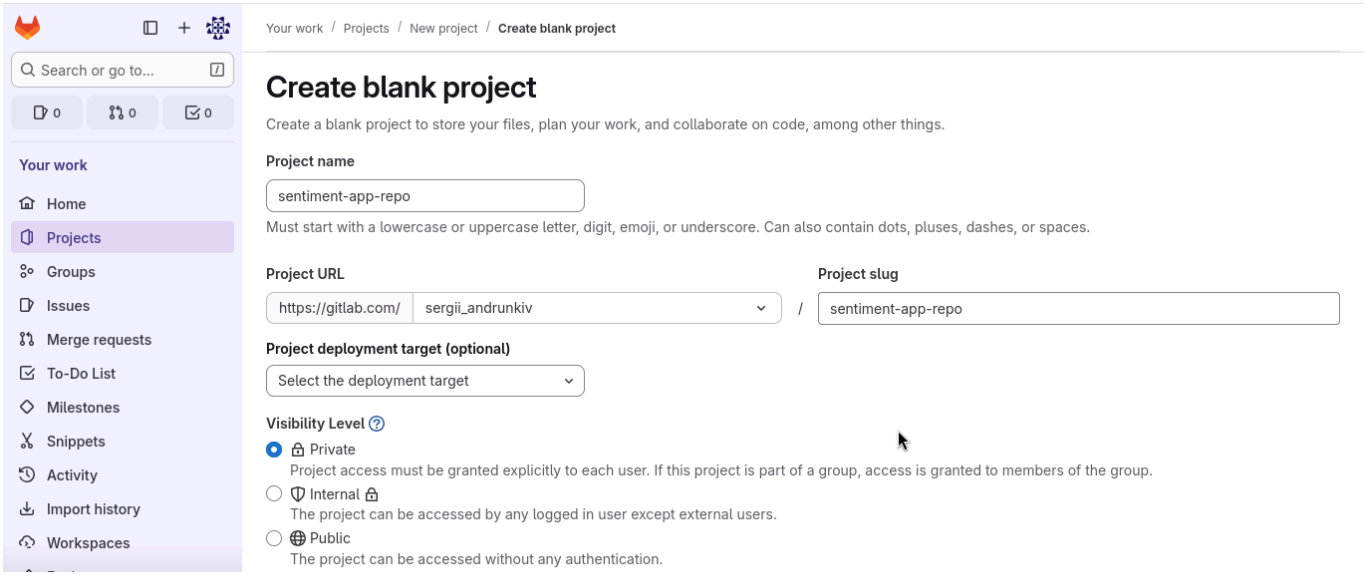

Рис. 3.3. Створення репозиторію для додатка

Другий репозиторій, sentiment-config-repo (див. рис. 3.4), буде містити конфігурації. Його єдина мета – зберігати декларативний опис бажаного стану системи у середовищі Kubernetes. Він містить лише маніфести Kubernetes – deployment.yaml, service.yaml та інші. За цим репозиторієм стежитиме оператор Argo CD та автоматично синхронізуватиме будь-які зміни з кластером, реалізуючи таким чином цикл GitOps. Приватність обох репозиторіїв забезпечить контроль доступу до коду та конфігурацій.

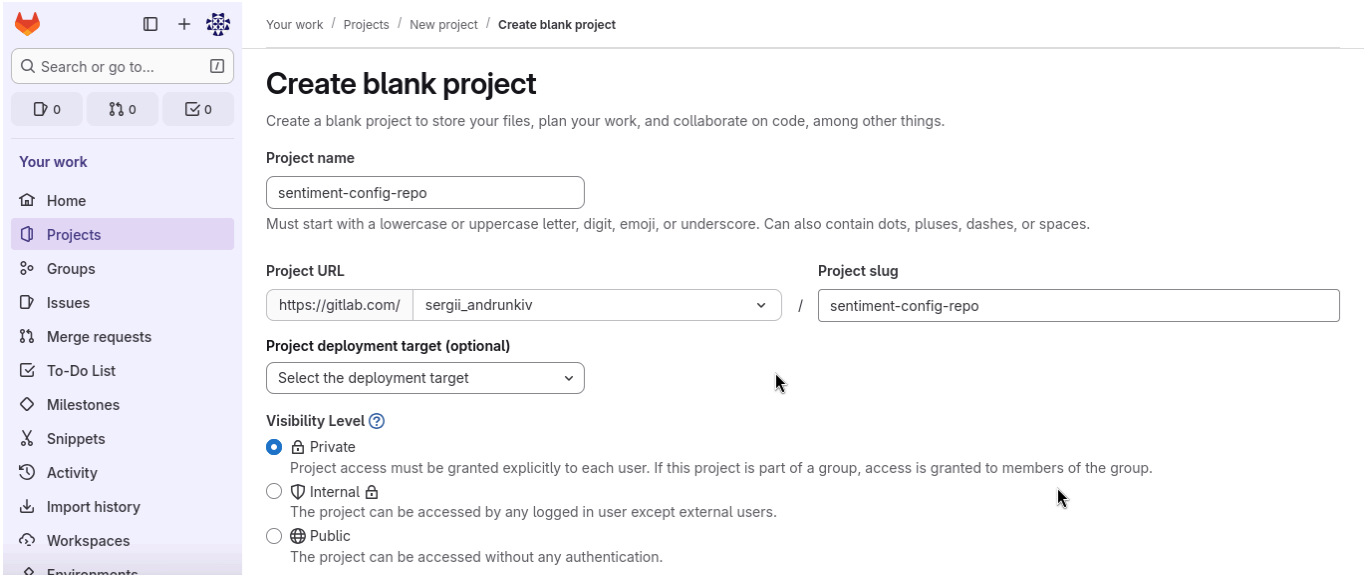

Рис. 3.4. Створення репозиторію для маніфестів Kubernetes

Для виконання майбутніх завдань CI/CD, що будуть визначені у конкретному маніфесті, потрібен виконавець GitLab Runner. Спочатку зареєструємо виконавця у GitLab в секції CI/CD на вкладці Runners, вказавши тег “docker”, для націлення завдань на локальний GitLab Runner (див. рис. 3.5).

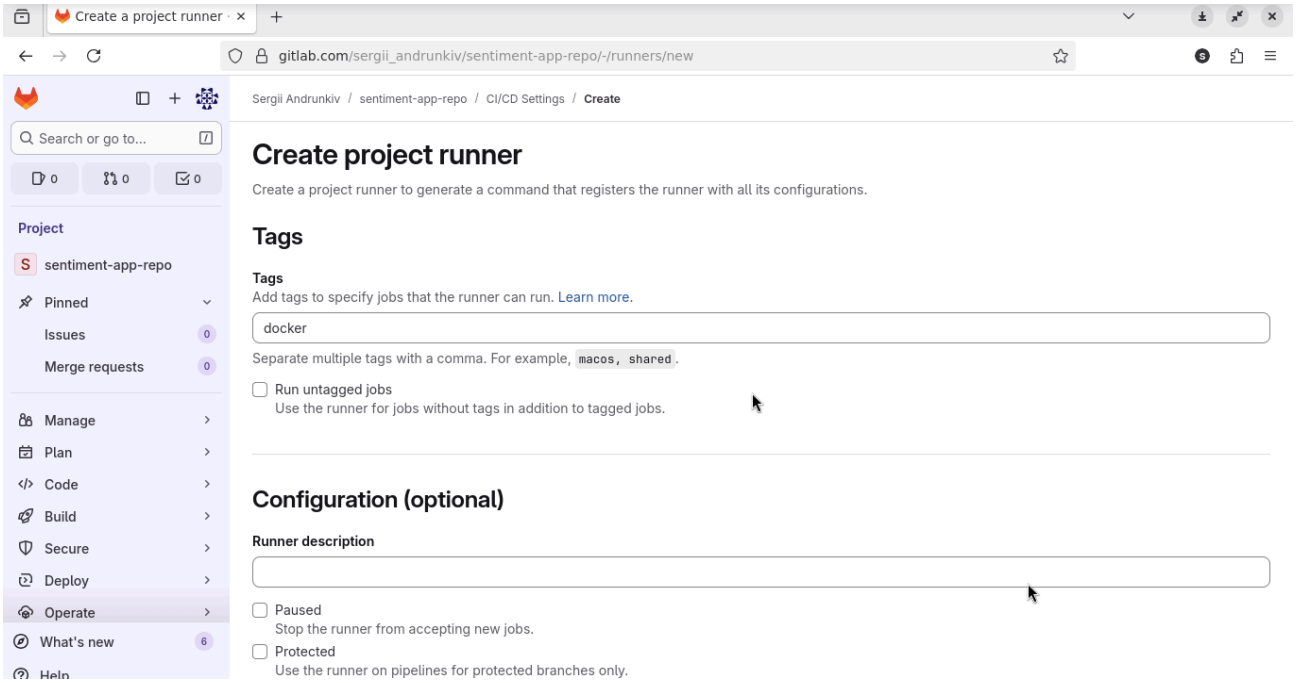

Рис. 3.5. Створення виконавця у GitLab

Оскільки конвеєр повинен мати мережевий доступ до локально розгорнутого сервісу SonarQube, проведемо реєстрацію локального ранера на машині. Виконанням команди sudo gitlab-runner register ми інтерактивно пов’язуємо виконавця із проєктом sentiment-app-repo, використовуючи унікальний URL та токен автентифікації з налаштувань CI/CD проєкту (див. рис. 3.6).

Після цієї вказівки кожне завдання на конвеєрі виконуватиметься у чистому, ізольованому Docker-контейнері. Такий підхід забезпечить відтворюваність і усуне конфлікти між залежностями. Тег “docker” у файлі .gitlab-ci.yml дозволить чітко націлювати завдання саме на цього локального виконавця, гарантувавши контрольоване і передбачуване середовище виконання.

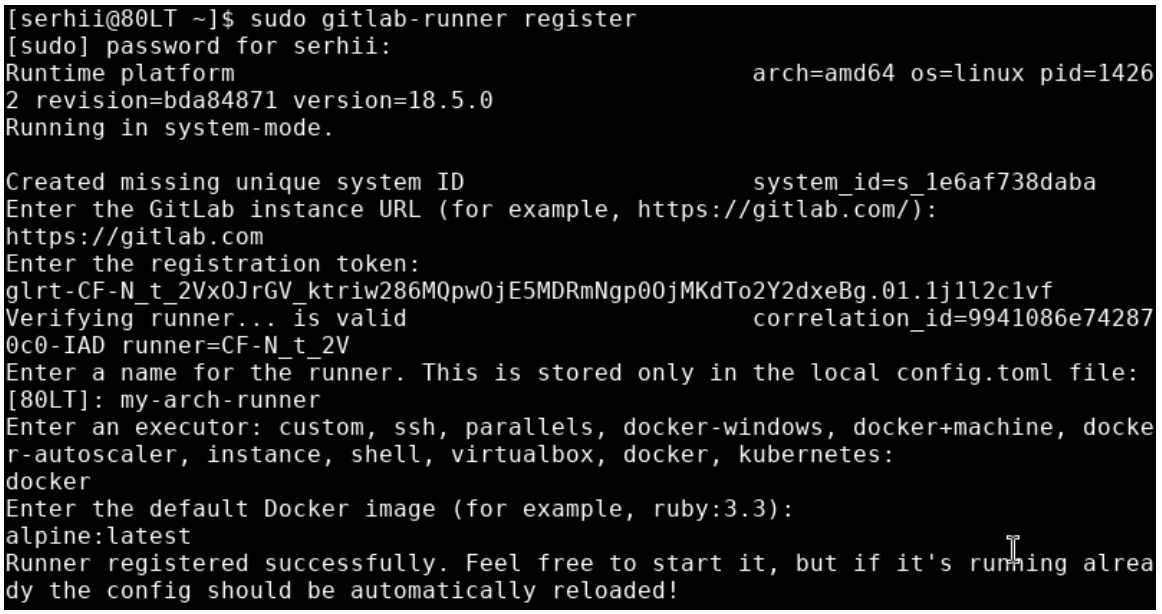

Рис. 3.6. Реєстрація GitLab Runner для запуску завдань конвеєра у контейнері Docker

Для коректної взаємодії виконавця з SonarQube, який також працює у контейнері Docker на тій самій хост-машині, налаштуємо мережевий доступ. Відкриємо конфігураційний файл /etc/gitlab-runner/config.toml і додамо у секцію [runners.docker] параметр extra_hosts = ["host.docker.internal:host-gateway"]. Тепер кожен контейнер CI/CD зможе додавати у свої /etc/hosts псевдонім host.docker.internal, який вказує на IP-адресу хост-машини (див. рис. 3.7).

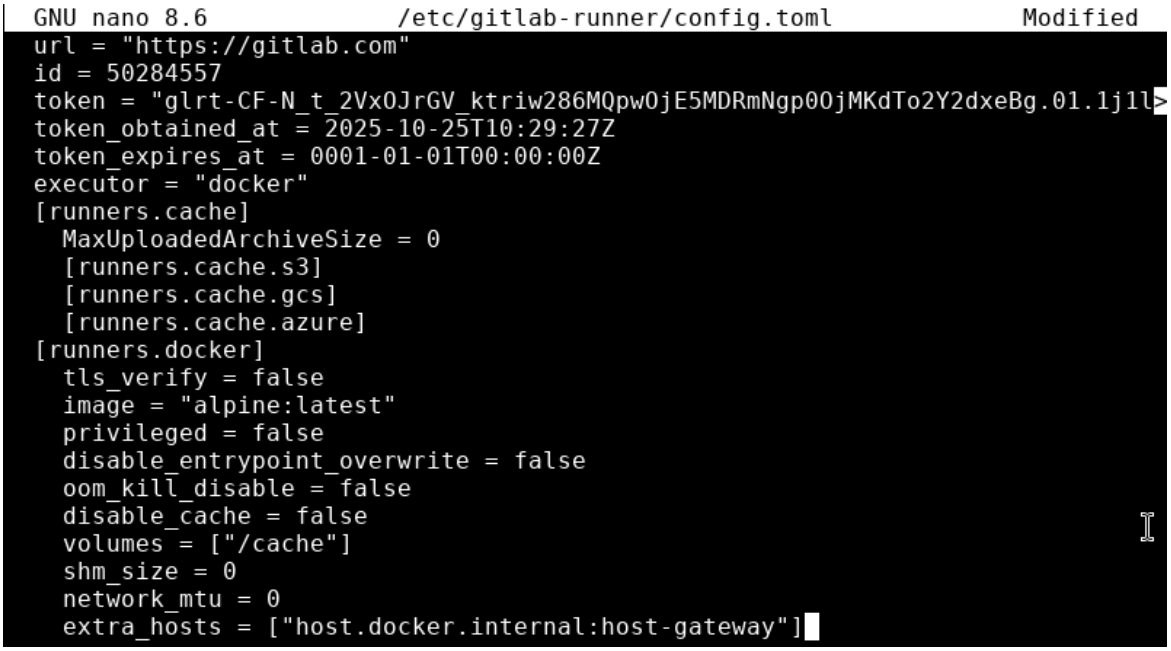

Рис. 3.7. Налаштування взаємодії GitLab Runner із SonarQube

Після реєстрації запускаємо GitLab Runner командою “sudo gitlab-runner start”, переходячи в режим очікування та опитування gitlab.com на наявність нових завдань (див. рис. 3.8).

```
[serhii@80LT ~]$ sudo gitlab-runner start
[sudo] password for serhii:
Runtime platform                                    arch=amd64 os=linux pid=1590
3 revision=bda84871 version=18.5.0
```

Рис. 3.8. Запуск GitLab Runner

3.2. Автоматизоване розгортання інфраструктури платформи

У цьому підрозділі ми створимо та розгорнемо основу для всієї платформи. Спершу запустимо локальний кластер Kubernetes, а потім, за допомогою Terraform, у ньому налаштуємо автоматичне розгортання та конфігурацію інструментів Service Mesh Linkerd та системи безперервного розгортання Argo CD.

Спочатку запускаємо локальний кластер Kubernetes за допомогою команди “minikube start --memory=6g --cpus=4” (див. рис. 3.9). Вказуємо достатню кількість обчислювальних ресурсів машини, оскільки і Argo CD, і Linkerd є ресурсомісткими системами. Виділення недостатнього обсягу пам'яті або CPU може призвести до помилок при запуску подів.

```
[serhii@80LT ~]$ minikube start --memory=8g --cpus=4
😄  minikube v1.37.0 on Arch
✨  Using the docker driver based on existing profile
❗  You cannot change the memory size for an existing minikube cluster. Please f
irst delete the cluster.
❗  You cannot change the CPUs for an existing minikube cluster. Please first de
lete the cluster.
🏃  Starting "minikube" primary control-plane node in "minikube" cluster
🚜  Pulling base image v0.0.48 ...
🔄  Restarting existing docker container for "minikube" ...
🐳  Preparing Kubernetes v1.34.0 on Docker 28.4.0 ...
🔎  Verifying Kubernetes components...
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟  Enabled addons: storage-provisioner, default-storageclass
🏄  Done! kubectl is now configured to use "minikube" cluster and "default" name
space by default
```

Рис. 3.9. Запуск кластера Kubernetes з виділеними ресурсами

Після ініціалізації кластера, клонуємо репозиторій sentiment-app-repo та створюємо в ньому директорію terraform/. Це дозволить зберігати код інфраструктури в тому ж репозиторії, де знаходиться код самого додатку. Створимо файл main.tf, який є декларативним описом бажаного стану всієї інфраструктури платформи.

Блок terraform визначає базові налаштування інфраструктури. Секція required_providers вказує Terraform, які плагіни потрібно завантажити для виконання цього коду. В даному випадку це hashicorp/kubernetes для взаємодії з K8s, hashicorp/helm для управління Helm-чартами та oboukili/argocd для специфічних ресурсів Argo CD (див. лістинг 3.1).

Лістинг 3.1 – Блок налаштування залежностей інфраструктури

```
# Define required Terraform providers
terraform {
  required_providers {
    kubernetes = {
      source  = "hashicorp/kubernetes"
      version = ">= 2.11.0"
    }
    helm = {
      source  = "hashicorp/helm"
      version = ">= 2.11.0"
    }
    argocd = {
      source  = "oboukili/argocd"
      version = ">= 5.0.0"
    }
  }
}
```

Наступні блоки конфігурують провайдерів kubernetes та helm. В обох випадках параметр config_path = "~/.kube/config" вказує їм використовувати стандартний файл конфігурації kubeconfig для автентифікації у кластері Kubernetes (див. лістинг 3.2).

Лістинг 3.2 – Блок конфігурації провайдерів

```
# Configure the Kubernetes provider to use the default kubeconfig
provider "kubernetes" {
```

Продовження лістингу 3.2

```
  config_path = "~/.kube/config"
}

# Configure the Helm provider to use the same kubeconfig
provider "helm" {
  kubernetes = {
    config_path = "~/.kube/config"
  }
}
```

Наступна частина коду Terraform відповідає за встановлення linkerd-crds за допомогою Helm. Це перша частина встановлення Linkerd, який створює у кластері визначення власних ресурсів. Параметр create_namespace = true гарантує, що namespace “linkerd” буде створений, якщо він ще не існує.

Друга частина встановлює основні компоненти Linkerd. Блок set_sensitive потрібний для безпеки. Він вставляє у Helm-чарт вміст файлів сертифікатів ca.crt, issuer.crt, та issuer.key, які пізніше ми згенеруємо через step-cli. Цей код вказує, що Linkerd буде використовувати власний центр сертифікації для mTLS. Останній блок depends_on гарантує, що цей ресурс буде створений лише після успішного створення linkerd_crds (див. лістинг 3.3).

Лістинг 3.3 – Блоки встановлення Linkerd

```
# Install Linkerd CRDs
resource "helm_release" "linkerd_crds" {
  name             = "linkerd-crds"
  namespace        = "linkerd"
  create_namespace = true
  chart            = "linkerd-crds"
  repository       = "https://helm.linkerd.io/stable"
}

# Install the Linkerd Control Plane with custom TLS certificates
resource "helm_release" "linkerd_control_plane" {
  name             = "linkerd-control-plane"
  repository       = "https://helm.linkerd.io/stable"
  chart            = "linkerd-control-plane"
  namespace        = "linkerd"
  create_namespace = true

  set_sensitive = [
    {
```

Продовження лістингу 3.3

```
      name  = "identityTrustAnchorsPEM"
      value = file("${path.module}/ca.crt")
    },
    {
      name  = "identity.issuer.tls.crtPEM"
      value = file("${path.module}/issuer.crt")
    },
    {
      name  = "identity.issuer.tls.keyPEM"
      value = file("${path.module}/issuer.key")
    }
  ]

  set = [
    {
      name  = "identity.issuer.crtExpiry"
      value = "8760h"
    }
  ]
  # This resource must wait for the CRDs to be created first
  depends_on = [helm_release.linkerd_crds]
}
```

Наступні три блоки argocd_auth_token, argocd_plain_text_password та ssh_private_key_content оголошують вхідні змінні. Вони дозволяють передавати в Terraform паролі та SSH-ключі під час виконання, а не жорстко кодувати їх у файлі. Атрибут sensitive = true запобігає відображенню цих значень у консолі або лог-файлах (див. лістинг 3.4).

Лістинг 3.4 – Блок з чутливими змінними

```
# Input variable for the Argo CD authentication token
variable "argocd_auth_token" {
  description = "Token for Argo CD authentication"
  type        = string
  sensitive   = true
}

# Input flag to tell the provider the token is a plain-text password
variable "argocd_plain_text_password" {
   description = "If true, the 'auth_token' is treated as a plain-text
password, not a JWT."
  type        = bool
  default     = false
}
```

Продовження лістингу 3.4

```
# Input variable for the GitOps SSH private key
variable "ssh_private_key_content" {
  type      = string
  sensitive = true
}
```

Далі створимо ресурс, який розгортає Argo CD. Він встановлює чарт argo-cd з офіційного репозиторію argoproj. В ньому також вказується конкретна версія, щоб забезпечити стабільність та відтворюваність розгортання. Найважливішим параметром є wait = true, який змушує Terraform призупинити виконання, доки всі поди Argo CD не перейдуть у стан “Ready”.

Одразу за ним знаходиться блок конфігурації провайдера argocd. Він вказує адресу сервера, очікуючи port-forward, передає токен авторизації зі змінної та використовує автентифікацію з паролем (див. лістинг 3.5).

Лістинг 3.5 – Блоки ресурсу та конфігурації ArgoCD

```
# Install Argo CD via Helm chart and wait for it to be ready
resource "helm_release" "argocd" {
  name             = "argocd"
  namespace        = "argocd"
  create_namespace = true
  chart            = "argo-cd"
  repository       = "https://argoproj.github.io/argo-helm"
  version          = "5.53.0"
  wait             = true
}

# Configure the Argo CD provider
provider "argocd" {
  server_addr = "localhost:8080"
  auth_token  = var.argocd_auth_token
  insecure    = true
  plain_text  = var.argocd_plain_text_password
}
```

Передостанній ресурс реєструє приватний репозиторій config-repo в Argo CD. Він вказує SSH URL та надає вміст ssh_private_key зі змінної. Так Argo CD зможе автентифікуватися в GitLab та отримати доступ до репозиторію конфігурацій.

Завершується файл опису інфраструктури найголовнішим ресурсом, що реалізує процес GitOps. Він створює додаток в Argo CD, який є зв'язком між Git та Kubernetes. Параметр source вказує, що конфігурацію треба взяти з папки k8s у sentiment-config-repo. Місце розгортання sentiment-app namespace у кластері вказується у параметрі destination.

Ще один важливий параметр sync_policy визначає як синхронізувати конфігурацію. Параметри automated, prune та self_heal створюють повністю автоматичний цикл, де будь-яка зміна в Git негайно застосовується до кластера, а будь-яка ручна зміна в кластері – скасовується (див. лістинг 3.6).

Лістинг 3.6 – Блоки реєстрації репозиторію та реалізації процесу GitOps

```
# Register the private GitOps repository in Argo CD
resource "argocd_repository" "sentiment_repo" {
  repo = "git@gitlab.com:sergii_andrunkiv/sentiment-config-repo.git"
  ssh_private_key = var.ssh_private_key_content
  depends_on = [helm_release.argocd]
}

# Create the main Argo CD GitOps application
resource "argocd_application" "sentiment_app" {
  metadata {
    name      = "sentiment-api-app"
    namespace = "argocd"
  }
  spec {
    project = "default"

    source {
                            repo_url                                        =
"git@gitlab.com:sergii_andrunkiv/sentiment-config-repo.git"
      target_revision = "HEAD"
      path            = "k8s" # It will monitor the k8s folder
    }
    destination {
      server    = "https://kubernetes.default.svc"
      namespace = "sentiment-app" # Where to deploy
    }
    sync_policy {
      automated {
        prune     = true
        self_heal = true
      }
      sync_options = ["CreateNamespace=true"] # Allow Argo to create a
namespace
    }
```

Продовження лістингу 3.6

```
  }
  # Ensure the repo is registered before creating the app
  depends_on = [argocd_repository.sentiment_repo]
}
```

Файл main.tf повністю готовий, а отже можна розпочати процес розгортання інфраструктури. Виконаємо команду terraform init -upgrade для завантаження провайдерів Kubernetes, Helm та Argo CD. Далі запустимо часткове застосування конфігурації. Параметр -target вказує Terraform застосувати лише ті ресурси, що відповідають за встановлення Helm-чартів Linkerd та Argo CD (див. рис. 3.10).

```
[serhii@80LT terraform]$ terraform init -upgrade
Initializing the backend...
Initializing provider plugins...
- Finding hashicorp/kubernetes versions matching ">= 2.11.0"...
- Finding hashicorp/helm versions matching ">= 2.11.0"...
- Finding oboukili/argocd versions matching ">= 5.0.0"...
- Using previously-installed hashicorp/kubernetes v2.38.0
- Using previously-installed hashicorp/helm v3.0.2
- Using previously-installed oboukili/argocd v6.2.0

  Warning: Additional provider information from registry

  The remote registry returned warnings for
  registry.terraform.io/oboukili/argocd:
  - This provider is deprecated, please change your configuration to use -
  https://registry.terraform.io/providers/argoproj-labs/argocd/latest

Terraform has been successfully initialized!
```

```
[serhii@80LT terraform]$ terraform apply -auto-approve -target=helm_release.link
erd_crds -target=helm_release.linkerd_control_plane -target=helm_release.argocd
helm_release.argocd: Refreshing state... [id=argocd]
helm_release.linkerd_crds: Refreshing state... [id=linkerd-crds]
helm_release.linkerd_control_plane: Refreshing state... [id=linkerd-control-plan
e]

Terraform used the selected providers to generate the following execution plan.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # helm_release.linkerd_control_plane will be created
  + resource "helm_release" "linkerd_control_plane" {
      + atomic                     = false
      + chart                      = "linkerd-control-plane"
      + cleanup_on_fail            = false
      + create_namespace           = true
      + dependency_update          = false
```

Рис. 3.10. Завантаження провайдерів та часткове застосування конфігурації

Цей двоетапний підхід є вимушеним, оскільки для подальшого налаштування Argo CD, Terraform-провайдеру argocd потрібен пароль доступу. Цей пароль створюється лише під час встановлення Helm-чарту argocd. Саме тому цей крок лише встановлює оператори, відкладаючи їх конфігурацію.

Перед остаточним розгортанням інфраструктури приготуємо власні сертифікати mTLS, які Terraform передасть у Linkerd. Згенеруємо та встановимо CRD для Linkerd, а також основні компоненти Linkerd (див. рис. 3.11).

```
[serhii@80LT terraform]$ linkerd install --crds | kubectl apply -f -
Rendering Linkerd CRDs...
Next, run `linkerd install | kubectl apply -f -` to install the control plane.

Warning: resource customresourcedefinitions/authorizationpolicies.policy.linkerd
.io is missing the kubectl.kubernetes.io/last-applied-configuration annotation w
hich is required by kubectl apply. kubectl apply should only be used on resource
s created declaratively by either kubectl create --save-config or kubectl apply.
 The missing annotation will be patched automatically.
customresourcedefinition.apiextensions.k8s.io/authorizationpolicies.policy.linke
rd.io configured
```

```
[serhii@80LT terraform]$ linkerd install --ignore-cluster | kubectl apply -f -
Warning: resource namespaces/linkerd is missing the kubectl.kubernetes.io/last-a
pplied-configuration annotation which is required by kubectl apply. kubectl appl
y should only be used on resources created declaratively by either kubectl creat
e --save-config or kubectl apply. The missing annotation will be patched automat
ically.
namespace/linkerd configured
clusterrole.rbac.authorization.k8s.io/linkerd-linkerd-identity created
clusterrolebinding.rbac.authorization.k8s.io/linkerd-linkerd-identity created
serviceaccount/linkerd-identity created
clusterrole.rbac.authorization.k8s.io/linkerd-linkerd-destination created
clusterrolebinding.rbac.authorization.k8s.io/linkerd-linkerd-destination created
serviceaccount/linkerd-destination created
secret/linkerd-sp-validator-k8s-tls created
validatingwebhookconfiguration.admissionregistration.k8s.io/linkerd-sp-validator
-webhook-config created
```

Рис. 3.11. Підготовка сертифікатів mTLS та встановлення CRD

За допомогою утиліти step-cli створимо самопідписаний кореневий сертифікат ca.crt та його ключ ca.key, який слугуватиме центром довіри для всього service mesh. Далі щойно створеним ca.key, згенеруємо проміжний сертифікат issuer.crt та issuer.key. Цей сертифікат буде використовуватися Linkerd для підпису всіх подальших сертифікатів mTLS для кожного поду в кластері (див. рис. 3.12).

```
[serhii@80LT terraform]$ step-cli certificate create root.linkerd.cluster.local
ca.crt ca.key --profile root-ca --no-password --insecure
✔ Would you like to overwrite ca.crt [y/n]: y
Your certificate has been saved in ca.crt.
Your private key has been saved in ca.key.
[serhii@80LT terraform]$ step certificate create identity.linkerd.cluster.local
issuer.crt issuer.key --profile intermediate-ca --ca ca.crt --ca-key ca.key --no
-password --insecure
bash: step: command not found
[serhii@80LT terraform]$ step-cli certificate create identity.linkerd.cluster.lo
cal issuer.crt issuer.key --profile intermediate-ca --ca ca.crt --ca-key ca.key
--no-password --insecure
Your certificate has been saved in issuer.crt.
Your private key has been saved in issuer.key.
[serhii@80LT terraform]$
```

Рис. 3.12. Створення кореневого та проміжного сертифікатів

Після того, як helm_release.argocd успішно завершив роботу при першому запуску Terraform, у кластері з'являється секрет з початковим паролем. Команда, що зображена на рисунку 3.13, витягує цей секрет, декодує пароль з Base64 і зберігає його у змінну середовища ARGOCD_PASSWORD.

```
[serhii@80LT terraform]$ export ARGOCD_PASSWORD=$(kubectl -n argocd get secret a
rgocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 -d)
```

Рис. 3.13. Збереження декодованого паролю у змінну середовища

Оскільки сервіс Argo CD доступний лише всередині кластера, командою “kubectl port-forward svc/argocd-server -n argocd 8080:443” створимо захищений тунель. Вона "прокидає" порт 443 сервісу argocd-server всередині кластера на порт 8080 локальної машини. Цей термінал залишаємо активним (див. рис. 3.14).

```
[serhii@80LT ~]$ kubectl port-forward svc/argocd-server -n argocd 8080:443
Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from [::1]:8080 -> 8080
```

Рис. 3.14. “Прокидання” порту 443 для Argo CD

Після цього веб-інтерфейс Argo CD стане доступний за посиланням http://localhost:8080 (див. рис. 3.15).

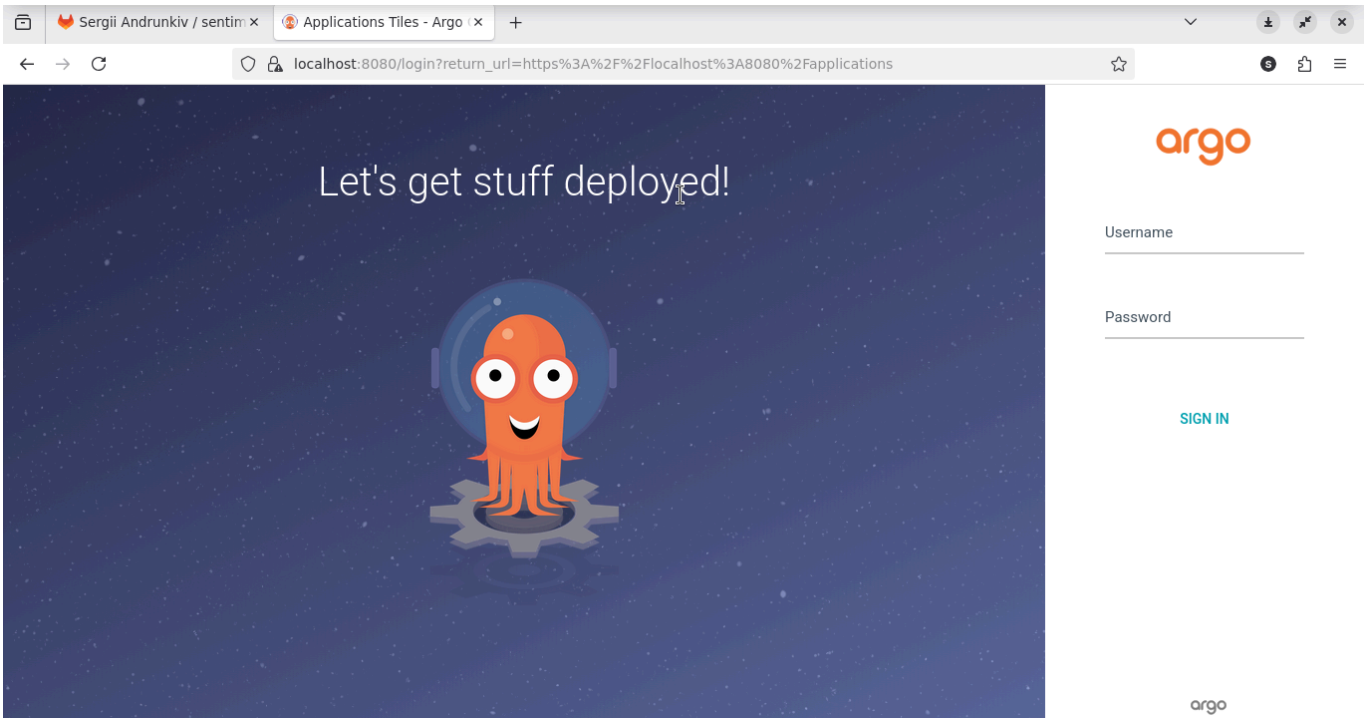

Рис. 3.15. Веб-інтерфейс Argo CD

Тепер пов’яжемо все разом. Командами на рисунку 3.16 створимо змінні середовища, які Terraform автоматично розпізнає як вхідні змінні. Передамо пароль TF_VAR_argocd_auth_token та параметр TF_VAR_argocd_plain_text_password=true для коректної роботи автентифікації паролем.

```
[serhii@80LT terraform]$ export TF_VAR_argocd_auth_token=$ARGOCD_PASSWORD
[serhii@80LT terraform]$ export TF_VAR_argocd_plain_text_password=true
```

Рис. 3.16. Передача значень у змінні середовища для Terraform

Тепер, коли argocd провайдер має доступ до API через port-forward та облікові дані, запустимо повторно terraform apply -auto-approve. Terraform побачить, що Helm-чарти вже встановлені, і застосує лише ресурси argocd_repository та argocd_application (див. рис. 3.17). Отже, налаштування GitOps завершено, і Argo CD починає моніторити config-repo.

```
[serhii@80LT terraform]$ terraform apply -auto-approve
helm_release.argocd: Refreshing state... [id=argocd]
helm_release.linkerd_crds: Refreshing state... [id=linkerd-crds]
helm_release.linkerd_control_plane: Refreshing state... [id=linkerd-control-plan
e]

Terraform used the selected providers to generate the following execution plan.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # argocd_application.sentiment_app will be created
  + resource "argocd_application" "sentiment_app" {
      + cascade  = true
      + id       = (known after apply)
      + status   = (known after apply)
      + validate = true
      + wait     = false

      + metadata {
          + generation       = (known after apply)
          + name             = "sentiment-api-app"
          + namespace        = "argocd"
          + resource_version = (known after apply)
          + uid              = (known after apply)
        }

      + spec {
          + project                = "default"
          + revision_history_limit = 10
```

Рис. 3.17. Повторний запуск розгортання інфраструктури

На рисунку 3.18 можна побачити стан Argo CD після встановлення, де видно, що система готова, але ще не використовується для розгортання конкретних додатків.

```
[serhii@80LT ~]$ kubectl get applications -n argocd
No resources found in argocd namespace.
[serhii@80LT ~]$ kubectl get secrets -n argocd
NAME                          TYPE                 DATA   AGE
argocd-initial-admin-secret   Opaque               1      80m
argocd-notifications-secret   Opaque               0      80m
argocd-secret                 Opaque               5      80m
sh.helm.release.v1.argocd.v1  helm.sh/release.v1   1      80m
```

Рис. 3.18. Поточний стан ArgoCD у кластері Kubernetes

3.3. Створення декларативних маніфестів додатка у GitOps-репозиторії

Після того, як інфраструктурні компоненти Argo CD та Linkerd розгорнуті в кластері Kubernetes, наступним кроком є налаштування “єдиного джерела правди” для нашого додатка. Цю роль виконує другий, окремий Git-репозиторій sentiment-config-repo. Argo CD вже налаштований постійно стежити за цим репозиторієм. Тепер потрібно наповнити репозиторій початковими маніфестами, які описують бажаний стан нашого ML-сервісу в кластері.

Спочатку склонуємо порожній репозиторій sentiment-config-repo на локальну машину. Всередині нього створимо директорію k8s/. Ця назва є обов’язковою, оскільки ми вказали цей шлях у ресурсі argocd_application під час налаштування Terraform. Argo CD буде сканувати лише цю директорію, ігноруючи будь-які інші файли в репозиторії (див. рис. 3.19).

```
[serhii@80LT ~]$ cd mlops/
[serhii@80LT mlops]$ ls
sentiment-app-repo
[serhii@80LT mlops]$ git clone https://gitlab.com/sergii_andrunkiv/sentiment-con
fig-repo.git
Cloning into 'sentiment-config-repo'...
Username for 'https://gitlab.com': sergii_andrunkiv
Password for 'https://sergii_andrunkiv@gitlab.com':
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (3/3), done.
[serhii@80LT mlops]$ cd sentiment-config-repo/
[serhii@80LT sentiment-config-repo]$ mkdir k8s
[serhii@80LT sentiment-config-repo]$ code k8s/deployment.yaml
[serhii@80LT sentiment-config-repo]$ code k8s/service.yaml
```

Рис. 3.19. Клонування репозиторію sentiment-config-repo

В каталозі k8s/ створимо ключовий маніфест deployment.yaml. У цьому файлі декларативно опишемо, як Kubernetes має запускати поди додатка. У metadata вкажемо ім'я sentiment-api та цільовий namespace sentiment-app, який Argo CD автоматично створить. Анотація linkerd.io/inject: enabled є прямою інструкцією, яка вказує контролеру Linkerd, щоб він при створенні будь-якого поду, що відповідає

цьому розгортанню, повинен автоматично впровадити свій проксі-контейнер як “sidecar”. Так ми організуємо прозоре mTLS-шифрування та збір метрик без будь-яких змін у коді самого Python-додатку. У spec вкажемо дві репліки для забезпечення базової відмовостійкості, а в описі контейнера використаємо заповнювач placeholder, щоб тег образу автоматично знаходився і замінювався конвеєром CI/CD під час кожного успішного збирання нового образу (див. лістинг 3.7).

Лістинг 3.7 – Вміст маніфесту deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sentiment-api
  namespace: sentiment-app
  annotations:
    linkerd.io/inject: enabled
spec:
  replicas: 2
  selector:
    matchLabels:
      app: sentiment-api
  template:
    metadata:
      labels:
        app: sentiment-api
    spec:
      imagePullSecrets:
      - name: gitlab-registry-creds
      containers:
      - name: api
                                                                    image:
registry.gitlab.com/sergii_andrunkiv/sentiment-app-repo:placeholder
        ports:
        - containerPort: 8000
```

Створимо файл k8s/service.yaml. Цей маніфест визначає, як інші сервіси в кластері будуть знаходити поди sentiment-api та спілкуватися з ними. Ресурс Service створить стабільну мережеву точку входу для групи подів (див. лістинг 3.8). Анотація selector: app: sentiment-api знайде усі поди, що мають таку мітку, яка була визначена у deployment.yaml. Сервіс прийме трафік на стандартний порт 80 і

перенаправить його на порт 8000, на якому додаток FastAPI запуститься всередині контейнера, згідно з Dockerfile, який створимо пізніше.

Лістинг 3.8 – Вміст маніфесту service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: sentiment-api-service
  namespace: sentiment-app
spec:
  selector:
    app: sentiment-api
  ports:
  - name: http
    port: 80
    targetPort: 8000
```

Згенеруємо SSH-ключ, щоб надати системі Argo CD безпечний доступ до приватного Git-репозиторію без використання логіна та пароля. Командою ssh-keygen створимо приватний ключ id_rsa та публічний ключ id_rsa.pub (див. рис. 3.20).

```
[serhii@80LT ~]$ ssh-keygen -t rsa -b 4096
Generating public/private rsa key pair.
Enter file in which to save the key (/home/serhii/.ssh/id_rsa):
Enter passphrase for "/home/serhii/.ssh/id_rsa" (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/serhii/.ssh/id_rsa
Your public key has been saved in /home/serhii/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:ngjfP9Aogu1Jik4X9C7nBTRsNX/VodaIU/3Xv10R6PE serhii@80LT
The key's randomart image is:
+---[RSA 4096]----+
|       o     o+..|
|    . . o   +o++ |
|   . =   . +.+o.+|
|  . + .   . o. E+|
|   +.o  So      +|
|  . *oo+o..     o|
| o * =oo+.      +|
|o o * .  ..    ..|
|..   .    ..     |
+----[SHA256]-----+
```

Рис. 3.20. Створення пари SSH-ключа

Скопіюємо вміст публічного ключа id_rsa.pub і вставимо його в налаштування sentiment-config-repo в GitLab. Так GitLab буде знати, що будь-хто, хто має відповідний приватний ключ, може отримати доступ (див. рис. 3.21).

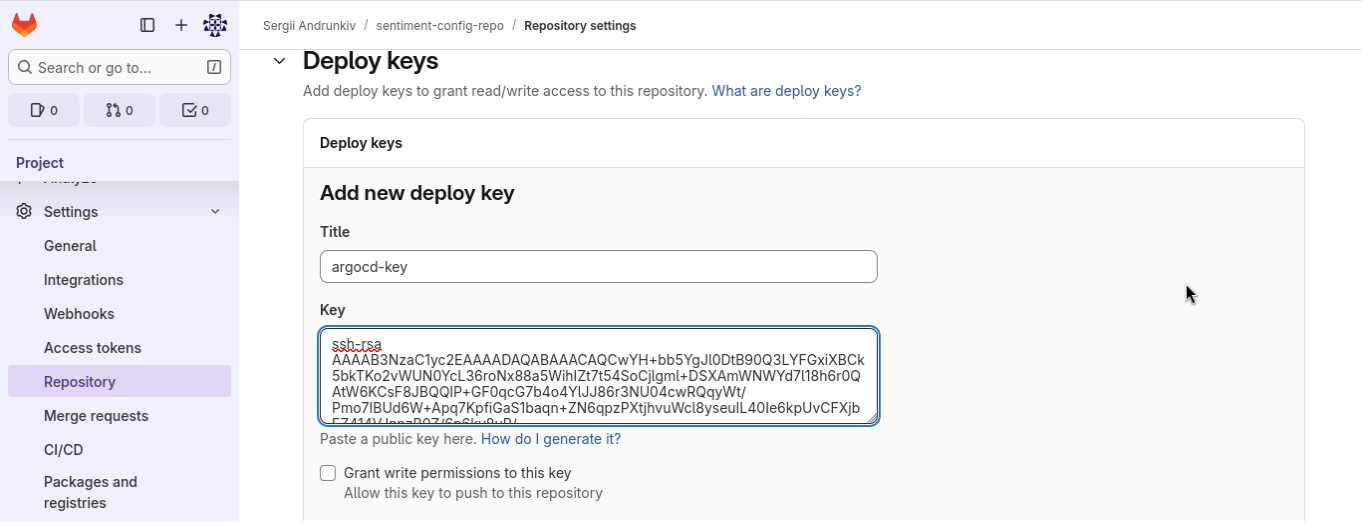

Рис. 3.21. Вставка ключа в репозиторій GitLab

Тепер візьмемо приватний ключ id_rsa, і безпечно збережемо його всередині Kubernetes за допомогою kubectl create secret. Створимо так званий “сейф” під назвою sentiment-repo-ssh-key у просторі імен argocd, щоб надати Argo CD цей ключ (див. рис. 3.22).

```
[serhii@80LT ~]$ kubectl create secret generic sentiment-repo-ssh-key -n argocd
--from-file=sshPrivateKey=/home/serhii/.ssh/id_rsa
secret/sentiment-repo-ssh-key created
```

Рис. 3.22. Збереження приватного ключа всередині Kubernetes

Обидва маніфести додаємо в індекс Git та зберігаємо в репозиторії з описовим повідомленням, після чого вони відправляються у віддалений репозиторій GitLab (див. рис. 3.20). Майже миттєво Argo CD, який моніторить репозиторій, виявляє новий коміт. У веб-інтерфейсі Argo CD статус додатка sentiment-api-app змінюється на OutOfSync. Оскільки, в налаштуваннях sync_policy увімкнено automated, Argo CD застосовує ці маніфести до кластера. Kubernetes намагається створити поди, але,

оскільки, образу :placeholder ще не існує, поди переходять у очікуваний стан помилки ImagePullBackOff. Це є коректною поведінкою на даному етапі, оскільки, платформа тепер повністю налаштована і очікує, поки конвеєр CI/CD надасть їй перший робочий образ (див. рис. 3.21).

```
[serhii@80LT sentiment-config-repo]$ code k8s/deployment.yaml
[serhii@80LT sentiment-config-repo]$ code k8s/service.yaml
[serhii@80LT sentiment-config-repo]$ git add .
[serhii@80LT sentiment-config-repo]$ git commit -m "Add initial K8s manifests"
[main 5db89d2] Add initial K8s manifests
 2 files changed, 35 insertions(+)
 create mode 100644 k8s/deployment.yaml
 create mode 100644 k8s/service.yaml
[serhii@80LT sentiment-config-repo]$ git push
Username for 'https://gitlab.com': sergii_andrunkiv
Password for 'https://sergii_andrunkiv@gitlab.com':
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 4 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 808 bytes | 161.00 KiB/s, done.
Total 5 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://gitlab.com/sergii_andrunkiv/sentiment-config-repo.git
   0b84f72..5db89d2  main -> main
[serhii@80LT sentiment-config-repo]$
```

Рис. 3.20. Відправлення маніфестів у репозиторій GitLab

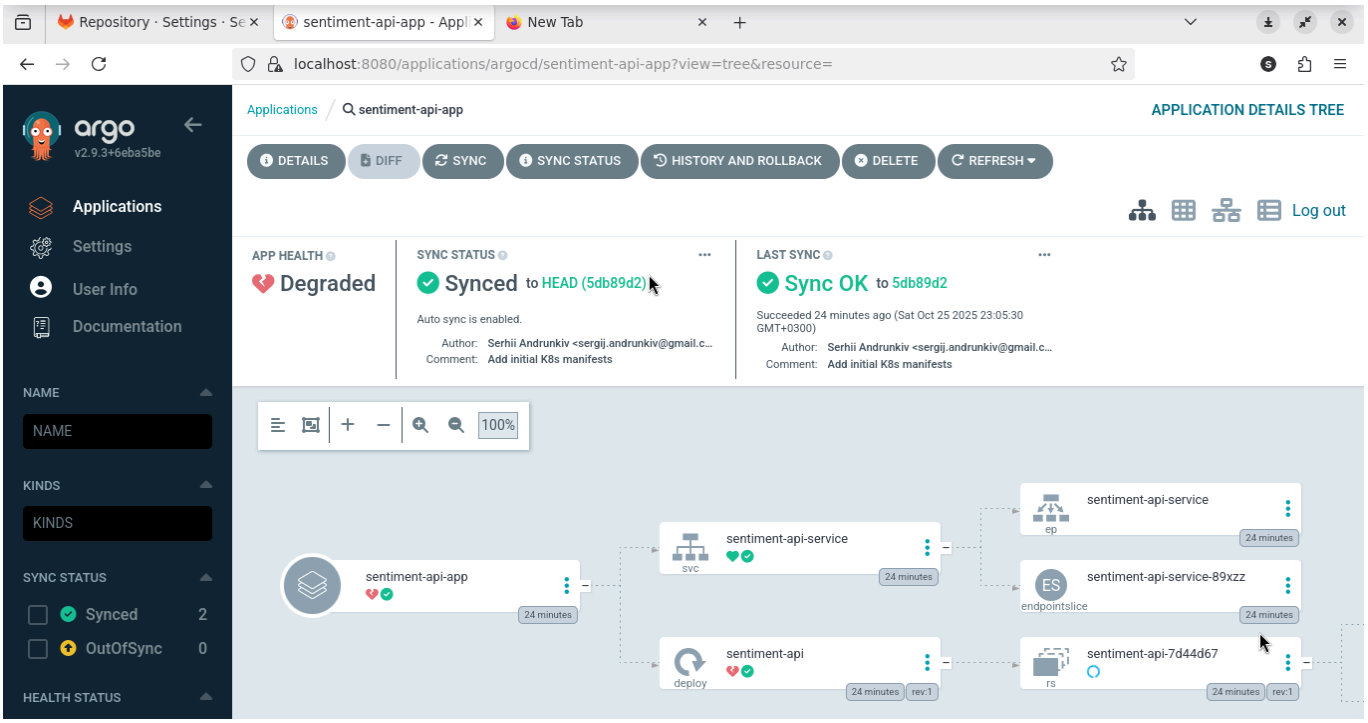

Рис. 3.21. Очікувана поведінка Argo CD при створення подів

3.4. Розробка сервісу моделі та конфігурація додатка

На цьому етапі будемо наповнювати репозиторій sentiment-app-repo, який є ядром додатка. Напишемо всю логіку для тренування моделі, її обслуговування через API, а також файли, що описують залежності, конфігурацію якості коду та інструкції для збірки образу Docker.

Спочатку налаштуємо керування великими файлами даних. Оскільки, датасет data/reviews.csv є завеликим для зберігання у Git, ініціалізуємо DVC у каталозі. Команда “dvc add data/reviews.csv” замінить оригінальний файл на малий текстовий файл-вказівник reviews.csv.dvc, який містить хеш-суму та інформацію про розташування (див. рис. 3.22).

```
(.venv) [serhii@80LT sentiment-app-repo]$ ls -lah
total 40K
drwxr-xr-x 7 serhii serhii 4.0K Oct 26 12:06 .
drwxr-xr-x 4 serhii serhii 4.0K Oct 26 10:36 ..
drwxr-xr-x 2 serhii serhii 4.0K Oct 26 10:36 data
drwxr-xr-x 3 serhii serhii 4.0K Oct 26 12:06 .dvc
-rw-r--r-- 1 serhii serhii  139 Oct 26 12:06 .dvcignore
drwxr-xr-x 7 serhii serhii 4.0K Oct 26 12:06 .git
-rw-r--r-- 1 serhii serhii 6.1K Oct 25 13:59 README.md
drwxr-xr-x 3 serhii serhii 4.0K Oct 26 10:24 terraform
drwxr-xr-x 5 serhii serhii 4.0K Oct 26 11:58 .venv
(.venv) [serhii@80LT sentiment-app-repo]$ dvc add data/reviews.csv
100% Adding...|                                                  |1/1 [00:00,  3.06file/s]

To track the changes with git, run:

        git add data/reviews.csv.dvc data/.gitignore

To enable auto staging, run:

        dvc config core.autostage true
```

Рис. 3.22. Ініціалізація DVC та створення файлу-вказівника даних

Перед вивантаженням даних, налаштуємо віддалене сховище Google Drive, яке буде зберігати вміст великих файлів, тоді як Git зберігатиме лише невеликі файли-вказівники. Для того щоб DVC міг програмно отримати доступ до приватного Google Drive, необхідно налаштувати автентифікацію за протоколом OAuth 2.0. Для цього у консолі Google Cloud потрібно зареєструвати клієнт DVC як “додаток”. У

розділі “APIs & Services” на вкладці “Credentials” створимо новий “OAuth client ID”. Оскільки DVC є локальною командною утилітою, а не веб-сервером, обираємо тип додатка “Desktop app” (див. рис. 3.23).

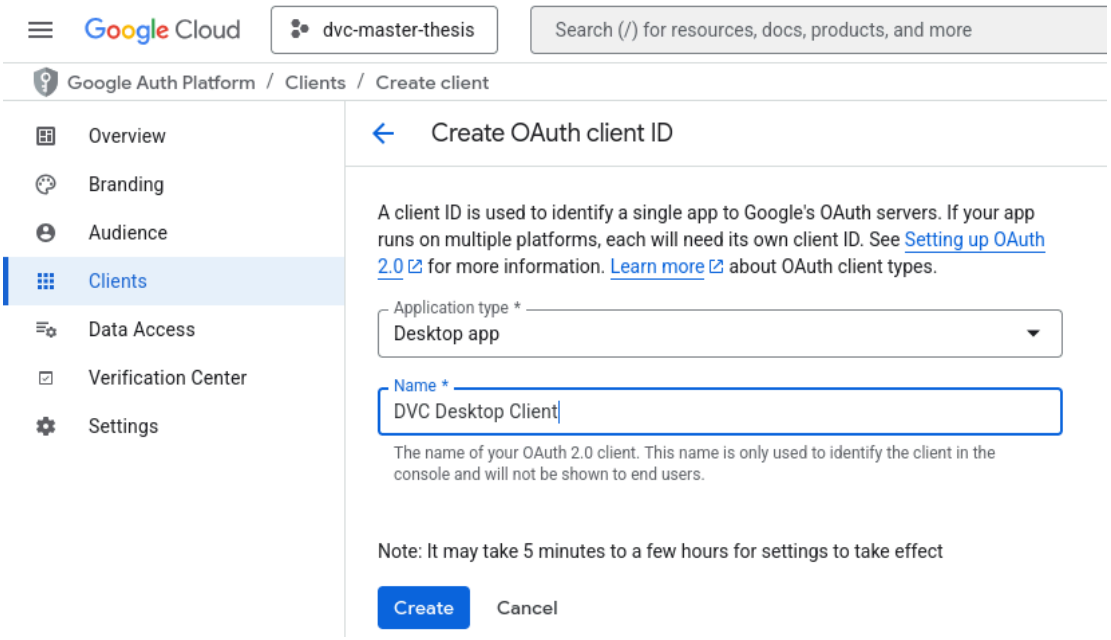

Рис. 3.23. Створення OAuth client ID у Google Cloud

Після створення Google надасть публічний ідентифікатор додатка Client ID та приватний пароль Client Secret (див. рис. 3.24).

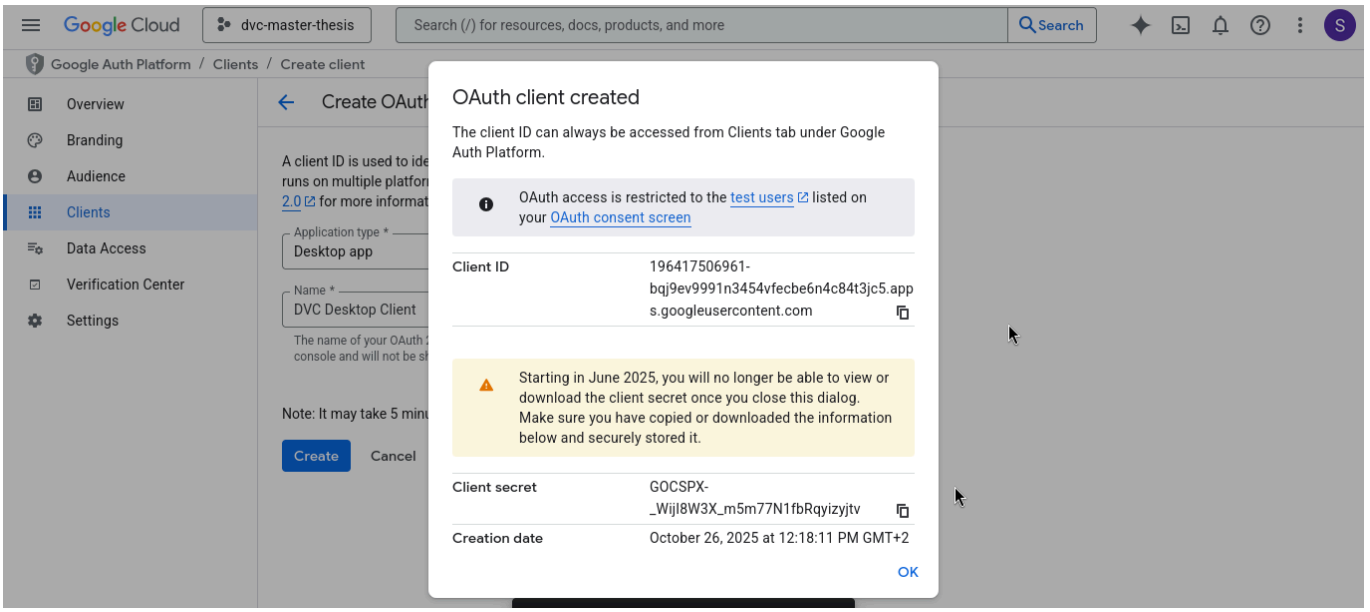

Рис. 3.24. Облікові дані для додатка

Маючи ці облікові дані, повернемося до терміналу, щоб налаштувати локальну конфігурацію DVC. Додаємо посилання на спільний каталог на Google Drive, а також за допомогою команд dvc remote modify ми передаємо отримані значення Client ID та Client Secret у конфігураційний файл .dvc/config. Ці налаштування виконуються лише один раз і повідомляють DVC, як саме він має бути видимим для серверів Google під час запиту на доступ (див. рис. 3.25).

```
(.venv) [serhii@80LT sentiment-app-repo]$ dvc remote list
my-gdrive       gdrive://1sq4wQkzY1-0ws2ZDs3sz9JqTXVfNWbKG      (default)
(.venv) [serhii@80LT sentiment-app-repo]$ dvc remote modify my-gdrive gdrive_cli
ent_id 196417506961-bqj9ev9991n3454vfecbe6n4c84t3jc5.apps.googleusercontent.com
(.venv) [serhii@80LT sentiment-app-repo]$ dvc remote modify my-gdrive gdrive_cli
ent_secret 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
(.venv) [serhii@80LT sentiment-app-repo]$ rm -f /home/serhii/.cache/pydrive2fs/7
10796635688-iivsgbgsb6uv1fap6635dhvuei09o66c.apps.googleusercontent.com/default.
json
(.venv) [serhii@80LT sentiment-app-repo]$ dvc push
Collecting                                      |0.00 [00:00,    ?entry/s]
Pushing                                                                  /
home/serhii/mlops/sentiment-app-repo/.venv/lib/python3.13/site-packages/oauth2cl
ient/_helpers.py:255: UserWarning: Cannot access /home/serhii/.cache/pydrive2fs/
196417506961-bqj9ev9991n3454vfecbe6n4c84t3jc5.apps.googleusercontent.com/default
.json: No such file or directory
  warnings.warn(_MISSING_FILE_MESSAGE.format(filename))
Your browser has been opened to visit:

    https://accounts.google.com/o/oauth2/auth?client_id=196417506961-bqj9ev9991n
3454vfecbe6n4c84t3jc5.apps.googleusercontent.com&redirect_uri=http%3A%2F%2Flocal
host%3A8080%2F&scope=https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fdrive+https%3A%2F
%2Fwww.googleapis.com%2Fauth%2Fdrive.appdata&access_type=offline&response_type=c
ode&approval_prompt=force

Authentication successful.
Pushing
1 file pushed
(.venv) [serhii@80LT sentiment-app-repo]$
```

Рис. 3.25. Налаштування локальної конфігурації DVC

Для завершення “рукостискання” автентифікації, яке відбувається під час першого запуску команди dvc push перейдемо по згенерованій унікальній URL-адресі для автентифікації, яка була виведена у консоль. Входимо у свій акаунт Google та надаємо “DVC Master Thesis App” дозвіл на перегляд та редагування файлів на Google Drive (див. рис. 3.26). Після цього відбудеться перенаправлення на локальну адресу http://localhost:8080. Отримавши підтвердження, DVC зберігає

токен доступу локально для майбутніх сесій, після чого негайно починається завантаження файлу.

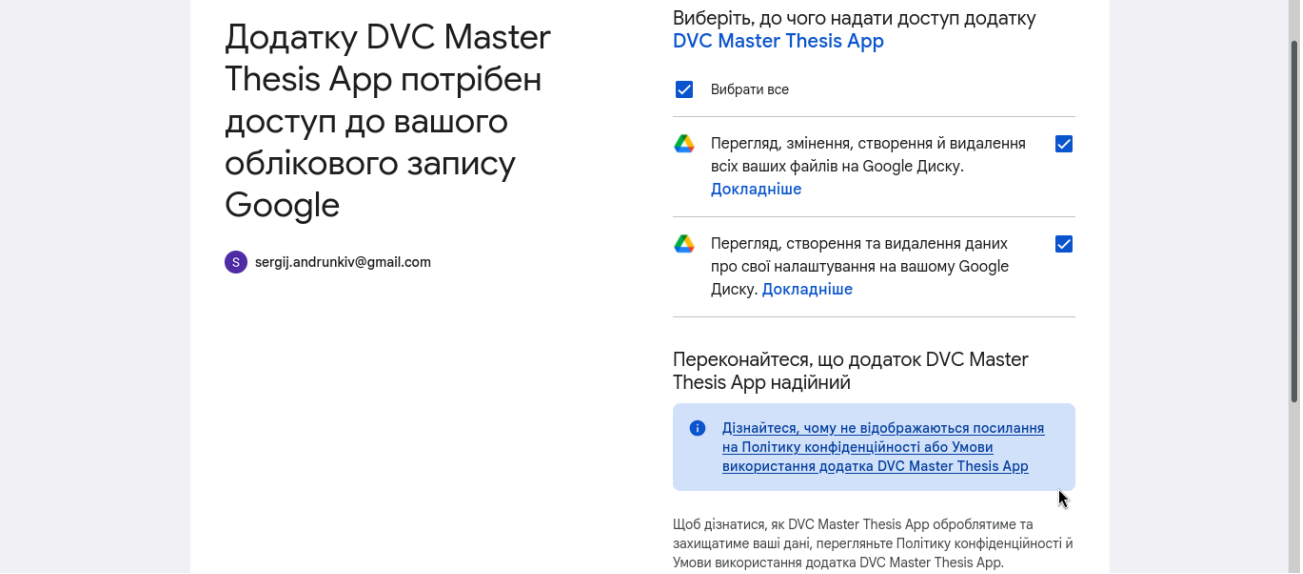

Рис. 3.26. Надання "DVC Master Thesis App" доступу до Google Drive

Створимо файл requirements.txt, що містить повний перелік бібліотек Python, необхідних для всіх етапів життєвого циклу моделі ML. Для обробки даних та машинного навчання використовуються pandas, scikit-learn та nltk, для MLOps-задач – mlflow та dvc[gdrive], для створення високопродуктивного API – fastapi, uvicorn та pydantic, для інтеграції метрик моніторингу – prometheus-fastapi-instrumentator (див. лістинг 3.9). Окремо створимо файл sonar-project.properties, який надає інструкції для статичного аналізатора коду SonarQube, вказуючи унікальний ключ проєкту та директорію з вихідним кодом (див. лістинг 3.10).

Лістинг 3.9 – Файл із залежностями requirements.txt

```
pandas
scikit-learn
mlflow
dvc[gdrive]
fastapi
uvicorn[standard]
pydantic
prometheus-fastapi-instrumentator
nltk # Для NLP
```

Лістинг 3.10 – Файл з інструкціями для аналізатора коду SonarQube

```
sonar.projectKey=sentiment-app
sonar.sources=src
sonar.python.version=3.9
```

Тепер детально розглянемо головний компонент циклу MLOps – файл тренування моделі “train.py”, який відповідає за генерування артефактів моделі. Цей скрипт призначений для автоматизованого виконання на етапі train конвеєра CI/CD (див. додаток Б). Його логіка починається з імпорту необхідних бібліотек, таких як mlflow для відстеження експериментів, pickle для серіалізації артефактів, nltk та sklearn для обробки тексту та машинного навчання. Спочатку скрипт завантажує stopwords з nltk та визначає функції clean_text, яка виконує необхідну попередню обробку “сирих” текстових даних, видаляючи HTML-теги, пунктуацію та стоп-слова, а також приводячи текст до нижнього регістру. Вся логіка тренування обгорнута у контекстний менеджер with mlflow.start_run(), який гарантує, що всі параметри, метрики та артефакти, згенеровані під час цього запуску, будуть автоматично зареєстровані та пов’язані в єдиному експерименті MLflow.

Всередині блоку MLflow послідовно виконуються всі етапи підготовки моделі. Спочатку, за допомогою pandas, відбувається завантаження та семплування даних з reviews.csv для прискорення тренування в середовищі CI, після чого мітки настрою конвертуються у числовий формат. Далі, на очищених даних навчається TfidfVectorizer, який перетворює текст на числові вектори. Цей векторизатор є критичним артефактом, тому він зберігається за допомогою pickle.dump у локальну папку model/vectorizer.pkl для подальшого пакування в Docker-образ, а також за допомогою mlflow.log_artifact для реєстрації в MLflow. Після цього відбувається поділ даних, тренування моделі MultinomialNB, її оцінка на тестовій вибірці та логування отриманих метрик і параметрів в MLflow. Аналогічно до векторизатора, фінальна натренована модель зберігається локально у model/model.pkl та логується в MLflow, забезпечуючи повну відтворюваність та готовність артефактів для наступного етапу – збірки сервісу.

Другий файл app.py слугує “споживачем” артефактів, створених скриптом train.py, і реалізує продуктивний REST API на базі FastAPI для обслуговування моделі (див. додаток Б). На самому початку скрипту визначаються абсолютні шляхи до артефактів model.pkl та vectorizer.pkl. Ключова логіка завантаження моделі відбувається один раз при старті сервера. Тепер великі файли моделі та векторизатора завантажуються в оперативну пам'ять заздалегідь, що дозволяє обробляти вхідні запити на прогнозування з мінімальною затримкою.

Для забезпечення надійності API використовується Pydantic (class TextRequest), який визначає очікувану схему вхідних JSON-запитів і автоматично валідує їх. У сервісі реалізовано два основні ендпоінти:

1) кореневий @app.get("/") – для базової перевірки працездатності;

2) основний @app.post("/predict") – для приймання тексту, проведення його попередньої обробки, і послідовного застосування завантаження з пам'яті vectorizer та model для отримання прогнозу.

Важливою особливістю сервісу є інтеграція моніторингу. Декоратор @app.on_event("startup") використовує prometheus-fastapi-instrumentator для автоматичного створення ендпоінту /metrics. Таким способом система Prometheus збирає кількість запитів, час відповіді, помилки безпосередньо з API та забезпечує детальний моніторинг роботи сервісу.

Останній файл Dockerfile описує процес пакування всього додатка у портативний контейнер. Збірка починається з легкого образу python:3.9-slim. Далі, за найкращими практиками кешування Docker, ми копіюємо requirements.txt і встановлюємо залежності. Командою RUN python -m nltk.downloader stopwords завантажуємо лінгвістичні дані в образ, щоб контейнер не потребував доступу до інтернету під час роботи. Після цього копіюємо вихідний код та каталог з натренованими артефактами. Останньою командою CMD запускаємо в контейнері uvicorn сервер, який обслуговує src.app і робить його доступним на порту 8000 (див. лістинг 3.11).

Лістинг 3.11 – Вміст Dockerfile для пакування додатка

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
RUN python -m nltk.downloader stopwords
COPY ./src /app/src
COPY ./model /app/model
CMD ["uvicorn", "src.app:app", "--host", "0.0.0.0", "--port", "8000"]
```

3.5. Розробка та конфігурація CI/CD конвеєра

Перейдемо до створення автоматизованого CI/CD конвеєра у файлі .gitlab-ci.yml, який розмістимо в корені sentiment-app-repo. Конвеєр буде описувати повний життєвий цикл додатка, від перевірки якості коду до фінального розгортання. Для коректної роботи цього конвеєра спочатку необхідні забезпечення доступу для GitOps-кроку та інтеграція з SonarQube.

Спочатку на локальній машині згенеруємо приватний gitops_key та публічний gitops_key.pub за допомогою команди ssh-keygen. Створимо ключ без парольної фрази, щоб його можна було використовувати в автоматизованих скриптах без інтерактивного втручання (див. рис. 3.27).

```
(.venv) [serhii@80LT sentiment-app-repo]$ ssh-keygen -t ed25519 -C "gitlab-ci-gi
tops" -f ~/.ssh/gitops_key
Generating public/private ed25519 key pair.
Enter passphrase for "/home/serhii/.ssh/gitops_key" (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/serhii/.ssh/gitops_key
Your public key has been saved in /home/serhii/.ssh/gitops_key.pub
The key fingerprint is:
SHA256:JRgx0NN3fT3T7+o1nr0TPToYV7ddJy9MB3bmQfh/Zpg gitlab-ci-gitops
The key's randomart image is:
+--[ED25519 256]--+
|    .o+o     .o.o|
|      o+. . ..++*|
|      ..... ..*=|
|         o    ooB|
|        S    o.BO|
|           . .E+X|
|            + .*=|
|           . ooo+|
|             .o++|
+----[SHA256]-----+
```

Рис. 3.27. Створення ключів для скрипта запуску CI/CD конвеєра

У репозиторії sentiment-config-repo у розділі Deploy Key додамо і налаштуємо публічний ключ gitops_key.pub. Надамо йому назву “GitOps Key” і встановимо прапорець “Grant write access”. Це дозволить будь-кому, хто володіє приватним ключем, робити git push у цей репозиторій (див. рис. 3.28).

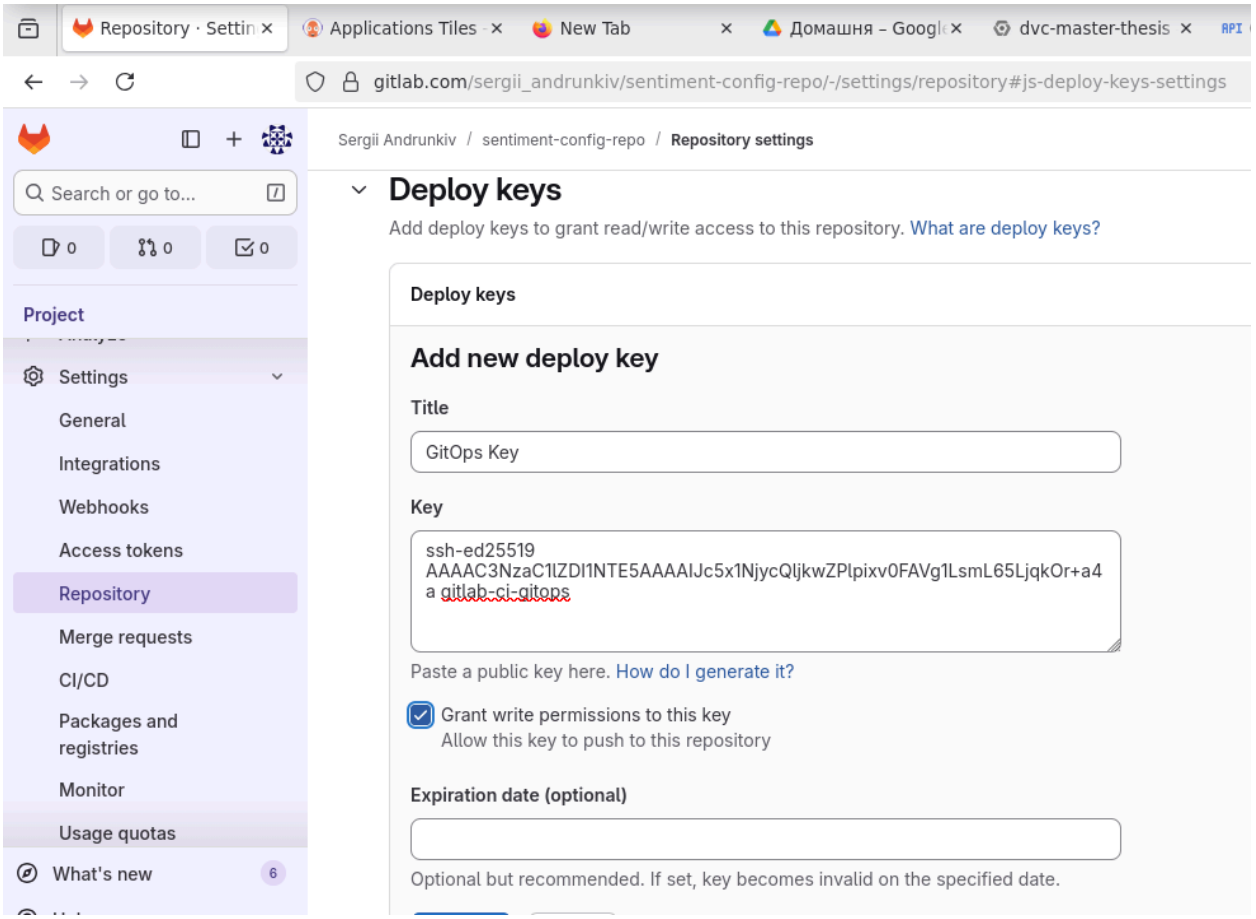

Рис. 3.28. Додавання публічного ключа gitops_key.pub у репозиторій

Тепер налаштуємо приватний ключ CI/CD Variable. Приватний ключ gitops_key скопіюємо і додамо в sentiment-app-repo у розділ Variables зі змінною GIT_SSH_KEY. Оскільки, ключ є багаторядковим, він кодується в base64 перед збереженням. У налаштуваннях змінної знімемо прапорець “Protect variable”, щоб deploy job міг працювати на гілці main, яка може бути не захищеною і знімемо прапорець “Masked”, оскільки Base64-рядок містить символи, що не підтримуються маскуванням (див. рис. 3.29).

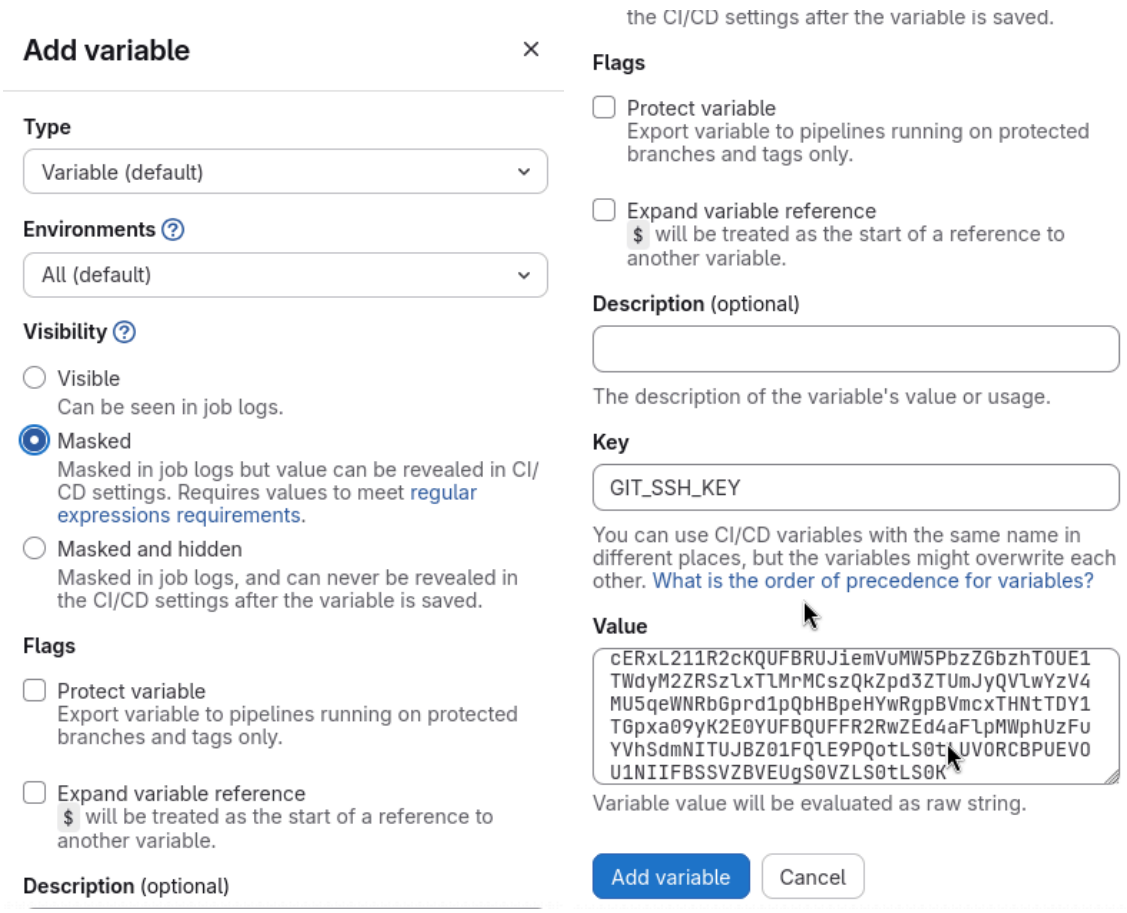

Рис. 3.29. Додавання gitops_key у розділ “Variables” зі змінною GIT_SSH_KEY

Для інтеграції етапу контролю якості коду, пайплайн повинен мати доступ до сервера SonarQube, який був запущений локально в Docker. У веб-інтерфейсі SonarQube за адресою http://localhost:9000 створимо новий пароль для входу (див. рис. 3.30).

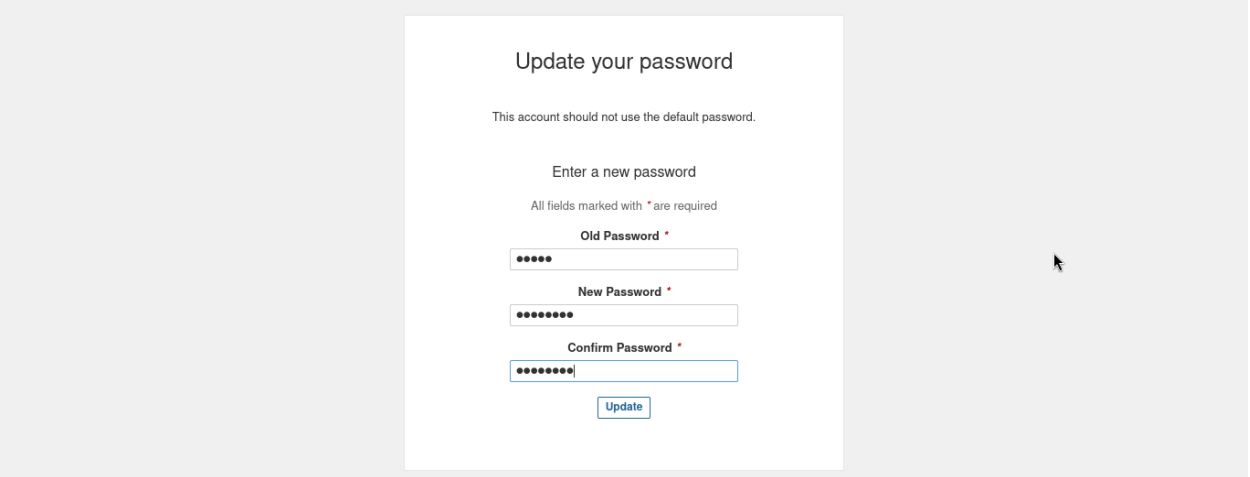

Рис. 3.30. Налаштування входу у SonarQube

Для того, щоб конвеєр CI/CD міг відправляти звіти про якість коду на сервер SonarQube, сканер sonar-scanner повинен пройти автентифікацію. Для цього зайдемо у веб-інтерфейс SonarQube. Створимо новий проєкт з унікальним ключем sentiment-app (див. рис. 3.31).

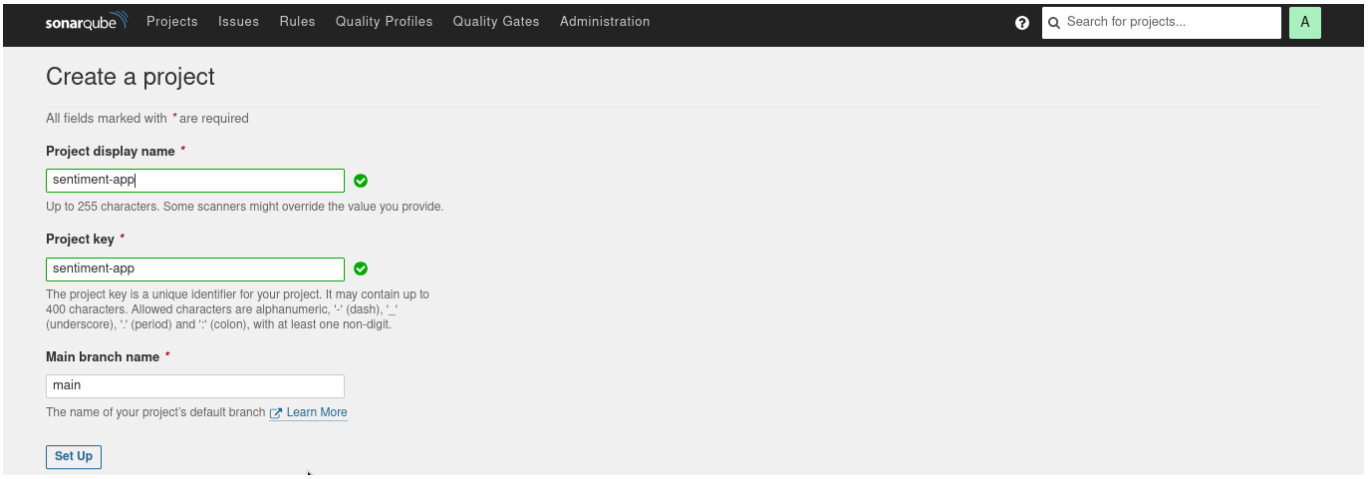

Рис. 3.31. Створення проєкту у SonarQube з унікальним ключем

Після створення проєкту SonarQube генерує токен доступу. Цей токен є довгим, унікальним рядком, який діє як пароль для sonar-scanner. Він дозволить сканеру підключатися до API SonarQube та завантажувати звіти про аналіз коду, пов'язуючи їх із щойно створеним проєктом sentiment-app. Цей належить до категорії чутливих даних і має зберігатися безпечно (див. рис. 3.32).

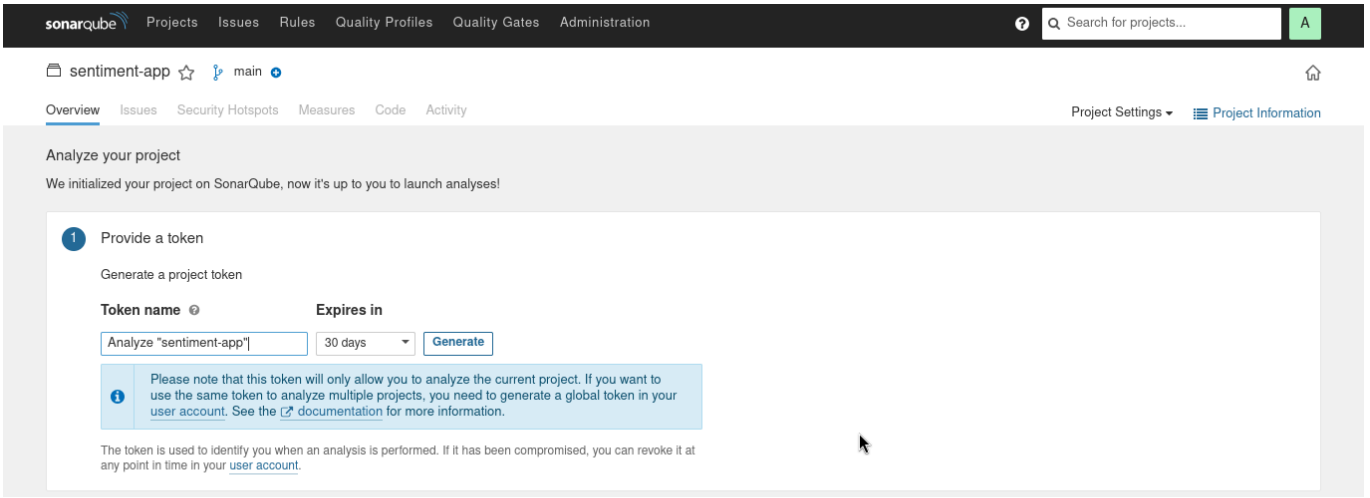

Рис. 3.32. Створення токену у SonarQube

Згенерований токен SonarQube не можна жорстко кодувати у файлі .gitlab-ci.yml, оскільки це є серйозним порушенням безпеки. Замість цього, будемо використовувати вбудований механізм секретів GitLab. Перейдемо до sentiment-app-repo і у розділі Variables і створимо нову змінну. У поле Key введемо SONAR_TOKEN, а у Value вставимо скопійований токен. Також встановимо прапорець “Mask variable”, який дасть команду GitLab CI/CD автоматично приховувати цей токен у логах виконання конвеєра, запобігаючи його випадковому витоку. Тепер sonar-scanner всередині конвеєра зможе отримати доступ до токена через змінну середовища $SONAR_TOKEN (див. рис. 3.33).

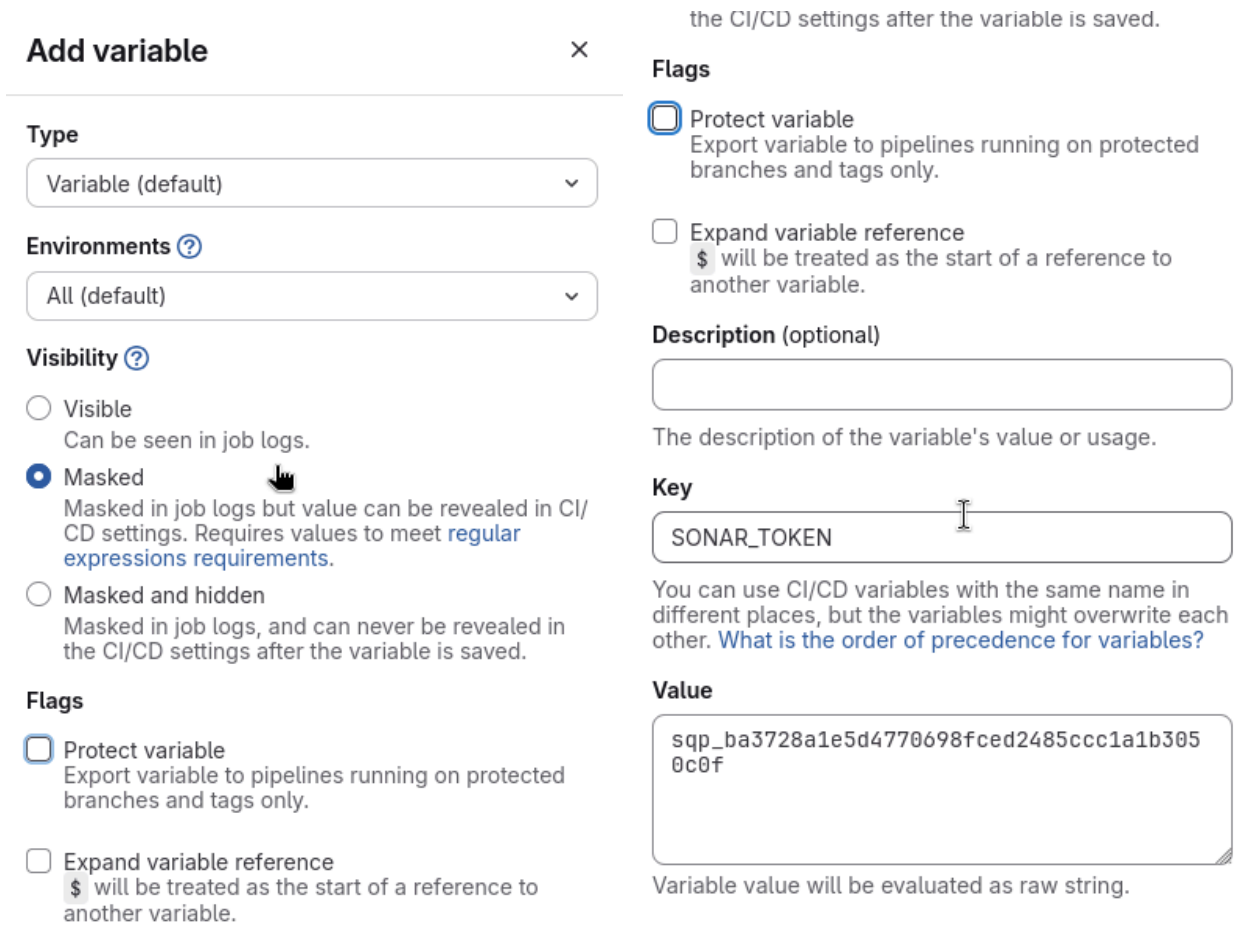

Рис. 3.33. Збереження токену SonarQube у змінній репозиторію GitLab

Оскільки, CI/CD-пайплайн буде виконуватися локальним GitLab Runner у контейнері Docker, йому потрібна IP-адреса хост-машини для доступу до іншого контейнера із SonarQube. За допомогою команди ip a | grep docker0 визначимо

IP-адресу (див. рис. 3.34) і додамо в CI/CD Variables під назвою SONAR_HOST_URL (див. рис. 3.35).

```
[serhii@80LT ~]$ ip a | grep docker0
5: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP gr
oup default
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
6: vetha501e8a@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue mas
ter docker0 state UP group default
[serhii@80LT ~]$
```

Рис. 3.34. Визначення IP-адреси мережевого інтерфейсу Docker

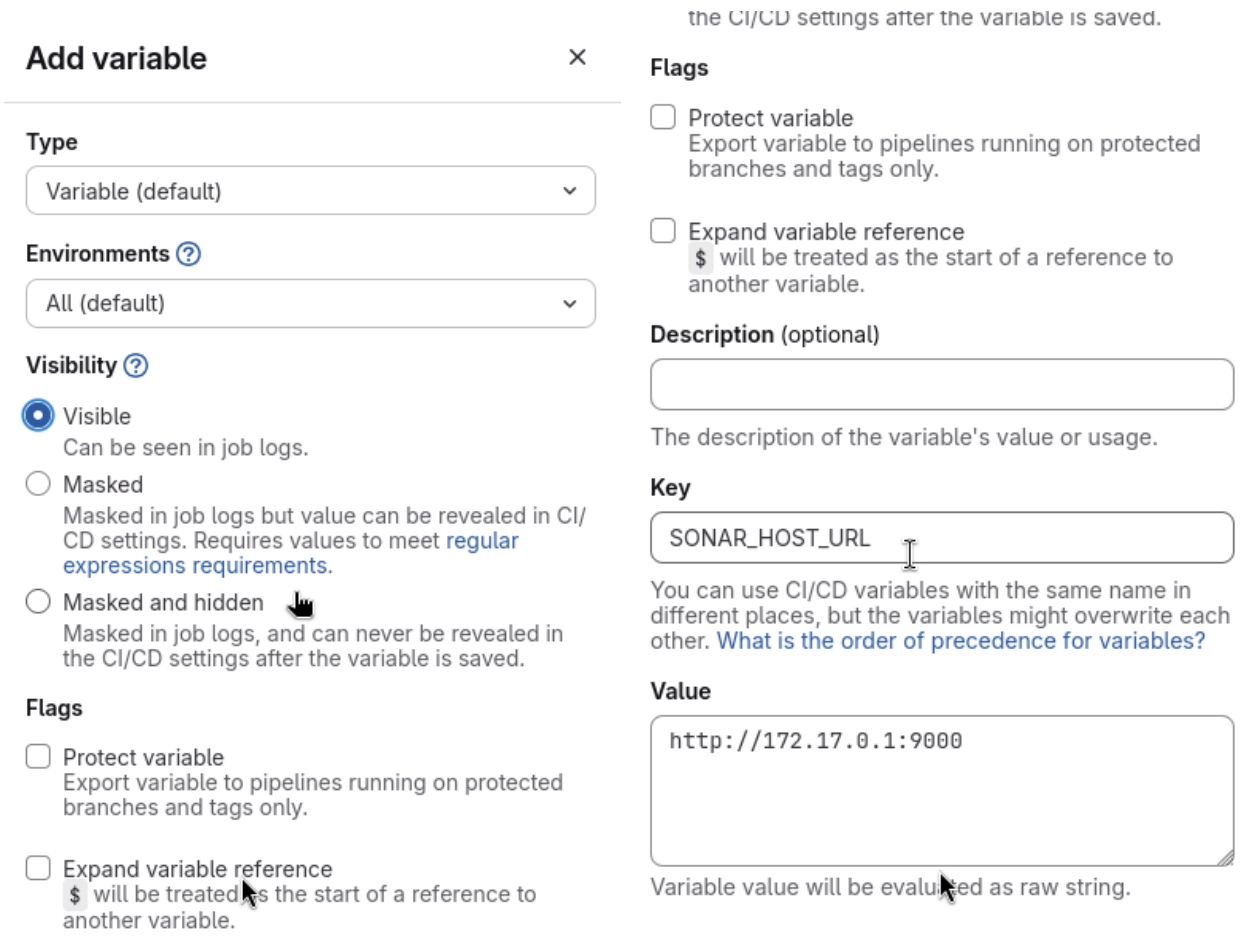

Рис. 3.35. Створення змінної SONAR_HOST_URL із мережевою адресою Docker

В даній роботі хмарний сервіс GitLab ніколи напряму не взаємодіє з локальною IP-адресою http://172.17.0.1:9000, а виступає виключно як координатор завдань. Коли виконується git push, GitLab аналізує .gitlab-ci.yml і бачить, що завдання sonar_scan вимагає тег docker. Оскільки, спільні хмарні виконавці вимкнені

для проєкту, GitLab шукає та передає це завдання зареєстрованому локальному GitLab Runner, який був раніше запущений на машині із тегом docker.

Саме цей локальний ранер отримує із змінних CI/CD проєкту рядок SONAR_HOST_URL зі значенням http://172.17.0.1:9000 та SONAR_TOKEN. Ранер запускає Docker-контейнер sonar-scanner-cli на тій самій хост-машині, де вже працює контейнер sonarqube. Оскільки, обидва контейнери знаходяться в одній внутрішній мережі Docker, sonar-scanner може успішно підключитися до sonarqube за цією адресою.

Нарешті створимо файл .gitlab-ci.yml, в якому опишемо чотири послідовні етапи та глобальні змінні (див. додаток Б).

Розглянемо детальніше код конвеєра. Змінна IMAGE_TAG генерує унікальне ім'я для кожного Docker-образу на основі хешу коміту $CI_COMMIT_SHORT_SHA, а CONFIG_REPO_URL зберігає SSH-адресу репозиторію конфігурацій.

Перший крок конвеєра використовує офіційний образ sonarsource/sonar-scanner-cli. Команда sonar-scanner запускає аналіз коду в каталозі src/, використовуючи SONAR_HOST_URL та SONAR_TOKEN з CI-змінних для підключення та автентифікації на сервері SonarQube. Якщо якість коду не відповідає заданим критеріям, цей етап аварійно завершиться, зупиняючи весь конвеєр.

Другий крок відповідає за MLOps-частину. Він використовує базовий образ Python та тег docker, щоб виконуватися на локальному ранері. Скрипт встановлює залежності Python, завантажує актуальний датасет з віддаленого сховища DVC і запускає скрипт тренування. Після успішного тренування, він зберігає згенеровані артефакти model.pkl та vectorizer.pkl і логи MLflow як артефакти GitLab CI.

На третьому кроці додаток пакується. Він використовує конфігурацію Docker-in-Docker. Завдяки ключовому слову dependencies: [train_model], цей job автоматично завантажує артефакти з попереднього етапу. Скрипт авторизується у GitLab Container Registry, запускає docker build і завантажує фінальний образ з унікальним тегом $IMAGE_TAG.

Фінальний четвертий етап реалізує GitOps. Він виконується на мінімальному образі alpine і також передбачає наявність тегу docker. Спочатку готується

середовище для Git-операцій. Встановлюються git, openssh, sed та base64. Потім декодується змінна $GIT_SSH_KEY у файл ~/.ssh/id_rsa, встановлюються коректні права доступу за допомогою chmod 600 та налаштовується Git. Вкінці скрипт клонує sentiment-config-repo. Потім знаходить рядок, що починається з “image:” у файлі deployment.yaml і замінює його на новий $IMAGE_TAG. Потім виконується git commit та git push з цією зміною назад у sentiment-config-repo. Саме цей push є тригером для Argo CD, який побачить оновлення і автоматично розгорне нову версію додатка в Kubernetes.

3.6. Верифікація наскрізного MLOps-циклу та тестування сервісу

У цьому фінальний підрозділі ми продемонструємо повний, наскрізний робочий процес системи, починаючи від дії розробника і закінчуючи перевіркою роботи оновленого сервісу в Kubernetes.

Практичну верифікацію всієї системи починаємо з команди git push у репозиторій додатку, яка ініціює наскрізний MLOps-цикл. Цією дією ми фіксуємо всі розроблені компоненти – програмний код API, скрипт тренування, інструкції зі збірки, а також сам файл конвеєра (див. рис. 3.36).

```
(.venv) [serhii@80LT sentiment-app-repo]$ git push
Username for 'https://gitlab.com': sergii_andrunkiv
Password for 'https://sergii_andrunkiv@gitlab.com':
Enumerating objects: 56, done.
Counting objects: 100% (56/56), done.
Delta compression using up to 4 threads
Compressing objects: 100% (39/39), done.
Writing objects: 100% (55/55), 64.13 MiB | 1.29 MiB/s, done.
Total 55 (delta 4), reused 0 (delta 0), pack-reused 0 (from 0)
To https://gitlab.com/sergii_andrunkiv/sentiment-app-repo.git
   2981225..d58cc61  main -> main
```

Рис. 3.36. Відправлення файлів на віддалений репозиторій sentiment-app-repo

Система контролю версій GitLab, отримавши цей push, розпізнає .gitlab-ci.yml і запускає конвеєр на раніше зареєстрованому локальному ранері (див. рис. 3.37).

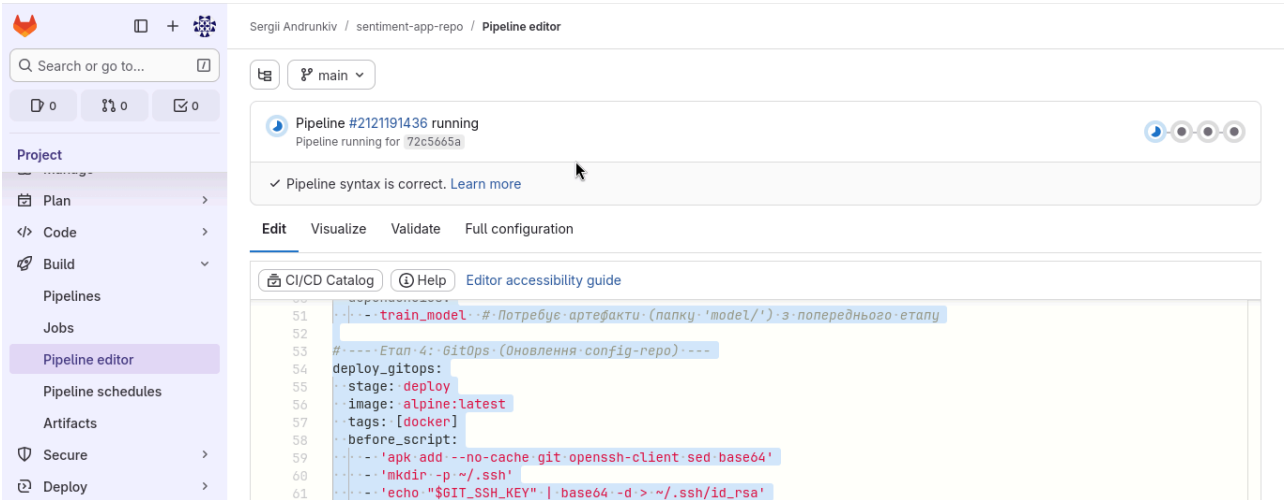

Рис. 3.37. Запуск конвеєра CI/CD

Спостерігати за процесом ми можемо у веб-інтерфейсі GitLab на вкладці Pipelines репозиторію sentiment-app-repo. Тут візуалізується проходження чотирьох послідовних етапів. На етапі validate запускається sonar-scanner, який аналізує якість коду. Успішний train виконує dvc pull та python src/train.py, зберігаючи артефакти. Етап build використовує Docker-in-Docker для збірки образу та його завантаження в GitLab Registry. Кульмінацією є етап deploy_gitops, який виконує git push у sentiment-config-repo, оновлюючи тег образу в deployment.yaml (див. рис. 3.38).

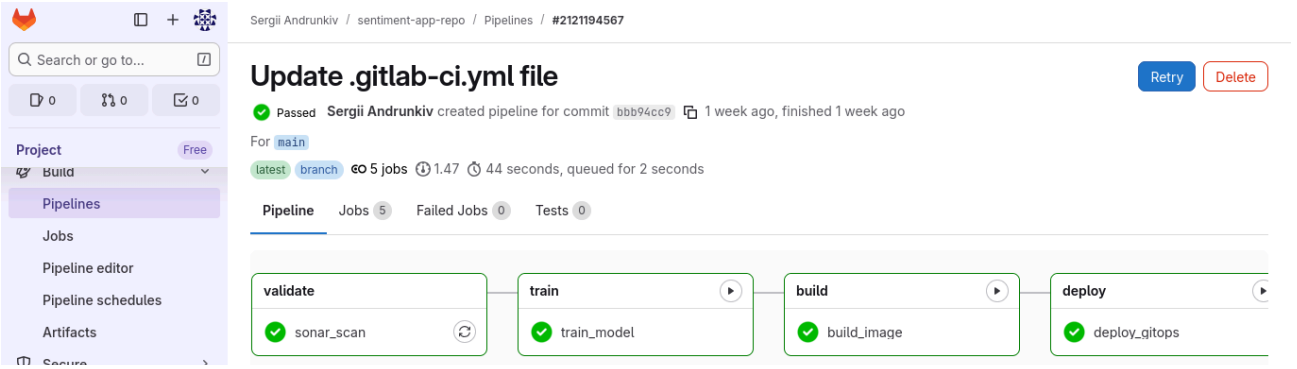

Рис. 3.38. Успішне проходження CI/CD-конвеєра

У цей момент спрацьовує GitOps-механізм. У веб-інтерфейсі Argo CD додаток sentiment-api-app майже миттєво змінює свій статус на OutOfSync, оскільки Argo CD виявляє розбіжність між бажаним станом, що надійшов через новий коміт у config-repo, та реальним станом у кластері (див. рис. 3.39).

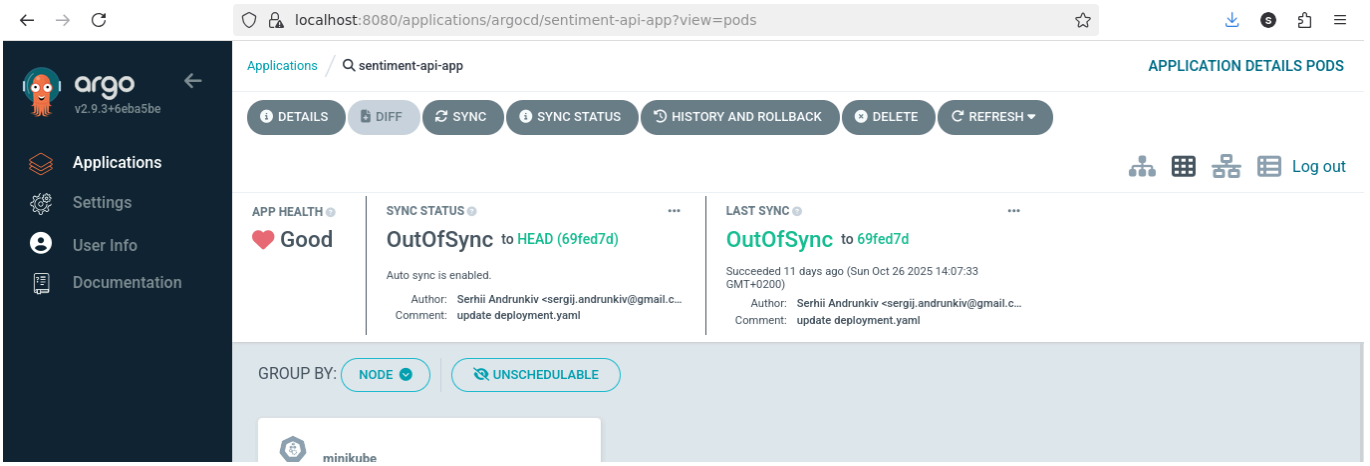

Рис. 3.39. Зміна статусу додатку на OutOfSync у Argo CD

Завдяки політиці automated self_heal, Argo CD негайно ініціює синхронізацію, надсилаючи оновлену конфігурацію Deployment до Kubernetes. За цим процесом можемо спостерігати в реальному часі за допомогою команди kubectl get pods -n sentiment-app -w. Вона демонструє процес “rolling update”. Kubernetes плавно зупиняє старі поди, які до цього могли бути в стані ImagePullBackOff через відсутність образу-заповнювача, та запускає нові поди з оновленим Docker-образом (див. рис. 3.40).

```
[serhii@80LT ~]$ kubectl get pods -n sentiment-app -w
NAME                                READY   STATUS             RESTARTS   AGE
sentiment-api-5c4d8b9f7-k2j4p       1/2     ImagePullBackOff   0          15m
sentiment-api-5c4d8b9f7-z9q7m       1/2     ImagePullBackOff   0          15m
[serhii@80LT ~]$ kubectl get pods -n sentiment-app -w
NAME                                READY   STATUS             RESTARTS   AGE
sentiment-api-65ff8b7989-x5g1r      0/2     Pending            0          0s
sentiment-api-65ff8b7989-x5g1r      0/2     ContainerCreating  0          1s
[serhii@80LT ~]$ kubectl get pods -n sentiment-app -w
NAME                                READY   STATUS             RESTARTS   AGE
sentiment-api-65ff8b7989-x5g1r      2/2     Running            0          16s
[serhii@80LT ~]$ kubectl get pods -n sentiment-app -w
NAME                                READY   STATUS             RESTARTS   AGE
sentiment-api-65ff8b7989-w8h3f      0/2     Pending            0          0s
sentiment-api-65ff8b7989-w8h3f      0/2     ContainerCreating  0          1s
sentiment-api-5c4d8b9f7-k2j4p       0/2     Terminating        0          16m
[serhii@80LT ~]$ kubectl get pods -n sentiment-app -w
NAME                                READY   STATUS             RESTARTS   AGE
sentiment-api-65ff8b7989-x5g1r      2/2     Running            0          1m10s
sentiment-api-65ff8b7989-w8h3f      2/2     Running            0          57s
```

Рис. 3.40. Перезапуск подів через оновлену конфігурацію Deployment

Після того, як поди переходять у стан Running, в окремому терміналі запускається команда linkerd viz dashboard. Вона відкриває веб-інтерфейс Linkerd, який надає глибоку телеметрію service mesh. Оскільки, в deployment.yaml була присутня анотація linkerd.io/inject: enabled, Linkerd автоматично впровадив свої проксі-контейнери у наші поди. Завдяки цьому, у дашборді ми можемо обрати namespace sentiment-app і побачити “золоті сигнали” success rate, RPS та P99 latency для нашого sentiment-api-service, підтверджуючи, що сервіс не лише запущений, але й знаходиться під повним моніторингом (див. рис. 3.41).

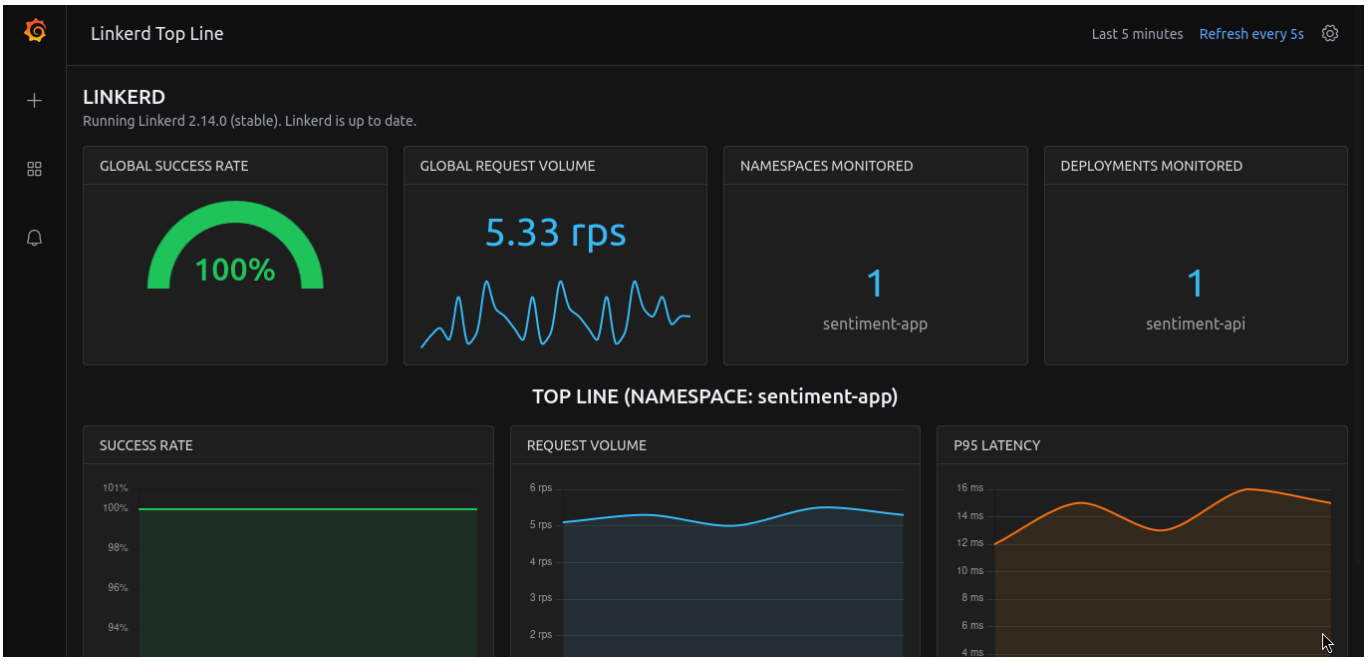

Рис. 3.41. Веб-інтерфейс Linkerd Dashboard

Проведемо верифікацію функціональності самого API. Оскільки, сервіс sentiment-api-service за замовчуванням доступний лише всередині кластера Minikube, для доступу з хост-машини виконаємо команду minikube service sentiment-api-service -n sentiment-app. Ця утиліта автоматично створить тимчасовий тунель до сервісу та відкриє його URL-адресу http://127.0.0.1:51234 у браузері (див. рис. 3.42).

```
[serhii@80LT ~]$ minikube service sentiment-api-service -n sentiment-app
|-------------|-------------------------|-------------|---------------------------|
|  NAMESPACE  |          NAME           | TARGET PORT |            URL            |
|-------------|-------------------------|-------------|---------------------------|
| sentiment-app | sentiment-api-service | http/80     | http://127.0.0.1:51234    |
|-------------|-------------------------|-------------|---------------------------|
🏃  Opening the sentiment-app/sentiment-api-service service in your browser...
[serhii@80LT ~]$
```

Рис. 3.42. Створення тунелю до API-сервісу

Після цього, minikube автоматично відкриє веб-браузер за цією URL. Там можна побачити чистий текстовий JSON – відповідь від нашого GET / ендпоінту (див. рис. 3.43).

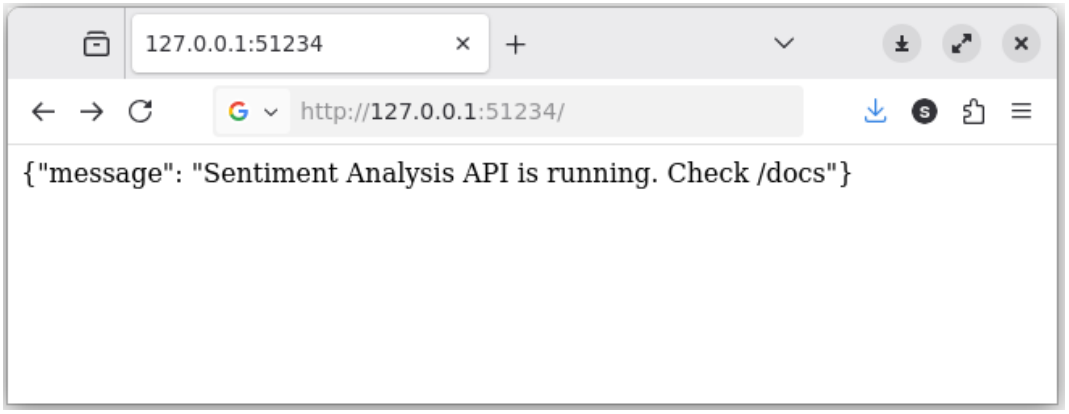

Рис. 3.43 Відповідь від GET-ендпоінту у форматі JSON

Тепер, використовуючи цю отриману адресу, відправимо позитивний тестовий POST-запит за допомогою утиліти curl. У тілі запиту передамо JSON-навантаження у форматі, який очікує наш Pydantic-клас. Успішне отримання у відповідь JSON-об'єкта, що містить коректний прогноз, є остаточним підтвердженням (див. рис. 3.44).

```
[serhii@80LT ~]$ curl -X POST "http://127.0.0.1:51234/predict" \
> -H "Content-Type: application/json" \
> -d '{"text": "This movie was absolutely fantastic, I loved it!"}'

{"text":"This movie was absolutely fantastic, I loved it!","sentiment":"positive
"}
[serhii@80LT ~]$
```

Рис. 3.44. Відправлення позитивного POST-запиту

Тепер відправимо другий запит з очевидно негативним текстом. Успішне отримання відповіді {"sentiment":"negative"} демонструє, що модель коректно розрізняє тональності, а не повертає заздалегідь визначене значення (див. рис. 3.45).

```
[serhii@80LT ~]$ cat /data/CE/Masters\ thesis/POST/POST\ negative
[serhii@80LT ~]$ curl -X POST "http://127.0.0.1:51234/predict" \
> -H "Content-Type: application/json" \
> -d '{"text": "This was the worst film I have ever seen."}'

{"text":"This was the worst film I have ever seen.","sentiment":"negative"}
[serhii@80LT ~]$
```

Рис. 3.45. Відправлення негативного POST-запиту

У результаті ми отримали добре спроєктований та налаштований комплексний MLOps-конвеєр, що поєднує CI/CD, GitOps та Service Mesh, з використанням локального GitLab Runner для зв'язку хмарного GitLab з локальними сервісами Minikube та SonarQube. За допомогою Terraform було автоматизовано розгортання Argo CD та Linkerd, а центральний конвеєр CI/CD повністю автоматизував життєвий цикл, починаючи від аналізу коду в SonarQube, версіонування даних з DVC та тренування з MLflow до збірки Docker-образу та GitOps-розгортання. Фінальною верифікацією ми підтвердили коректність наскрізного процесу, а функціональним тестуванням через curl та моніторингом “золотих сигналів” у Linkerd Viz Dashboard довели, що поставлена мета – створення автоматизованої, безпечної та спостережуваної MLOps-платформи – була досягнута.

РОЗДІЛ 4
ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1. Охорона праці

Розробка та впровадження інтегрованої системи MLOps передбачає тривалу роботу інженера за комп’ютером, що супроводжується значним нервово-емоційним навантаженням та впливом специфічних факторів виробничого середовища. У цьому розділі ми проаналізуємо умови праці розробника програмного забезпечення, ідентифікуємо небезпечні та шкідливі виробничі фактори, а також розробимо заходи щодо їх усунення відповідно до чинного законодавства України, зокрема Закону України «Про охорону праці» та сучасних європейських стандартів, імплементованих в Україні. Об’єктом дослідження є робоче місце інженера-програміста, обладнане персональним комп’ютером, розташоване в офісному приміщенні.

Класифікація небезпечних та шкідливих виробничих факторів здійснюється відповідно до ДСТУ 2293:2014 «Охорона праці. Терміни та визначення понять». Згідно з цим стандартом та результатами гігієнічної оцінки, на інженера можуть впливати фактори різних груп. До фізичних факторів належать підвищений рівень електромагнітних полів від системного блоку, монітора або ДБЖ, ризик ураження електричним струмом у разі замикання в електричному ланцюзі, недостатня або нерівномірна освітленість робочої зони та прямий блиск на екрані. Також враховується шум від систем охолодження серверного обладнання відповідно до ДСН 3.3.6.037-99. До психофізіологічних факторів відносяться нервово-емоційне напруження через високу відповідальність та дедлайни, перенапруження зорового аналізатора, а також статичне фізичне навантаження внаслідок вимушеної робочої пози.

Організація робочого місця користувача ПК регламентується НПАОП 0.00-7.15-18 «Вимоги до безпеки та захисту здоров’я працівників під час роботи з екранними пристроями», який гармонізовано з Директивою 90/270/ЄЕС. Згідно з

цим документом, роботодавець зобов’язаний провести аналіз ризиків робочого місця. Робочий стіл має бути достатнього розміру для розміщення периферії та забезпечувати комфортне розташування рук. Робоче крісло повинно бути стійким, підйомно-поворотним, регулюватися за висотою та кутом нахилу спинки для підтримки попереку.

Ергономічні параметри обладнання повинні відповідати стандартам серії ДСТУ EN ISO 9241 «Ергономіка взаємодії людина-система». Екран монітора розташовується на відстані 600–700 мм від очей, лінія погляду має падати на центр або верхню третину екрана під кутом 15-20°. Клавіатура повинна мати матову поверхню для уникнення відблисків. Режим праці та відпочинку встановлюється відповідно до характеру виконуваної роботи: рекомендовано робити регламентовані перерви тривалістю 10–15 хвилин кожні 1–2 години роботи для профілактики зорової та фізичної втоми.

Мікроклімат у приміщенні має відповідати Санітарним нормам мікроклімату виробничих приміщень ДСН 3.3.6.042-99. Робота програміста відноситься до категорії робіт 1а (легка фізична робота). Оптимальними параметрами мікроклімату є:

– температура повітря 22–24 °С у холодний період року та 23–25 °С у теплий період;

– відносна вологість 40–60 %;

– швидкість руху повітря не більше 0,1 м/с; також рекомендується враховувати положення ДСТУ EN 15251:2011, що визначає параметри мікроклімату для проектування та оцінки енергоефективності будівель.

Освітлення робочого місця проектується згідно з ДБН В.2.5-28:2018 «Природне і штучне освітлення» та з урахуванням ДСТУ EN 12464-1:2016 «Світло та освітлення». Приміщення повинно мати природне освітлення, при якому КПО не менше 1,5 %. Для штучного освітлення використовується система загального рівномірного або комбінованого освітлення. Нормована освітленість на поверхні столу має становити 500 лк. Важливою вимогою є обмеження коефіцієнта пульсації освітленості, що не повинно перевищувати 5 % та показника дискомфорту UGR, що

забезпечується використанням LED-світильників з якісними драйверами та нейтральною колірною температурою 4000 К. Вікна повинні бути обладнані регульованими жалюзі для захисту від прямого сонячного світла.

Електробезпека офісного приміщення регулюється НПАОП 40.1-1.21-98 «Правила безпечної експлуатації електроустановок споживачів» та ПУЕ:2017 «Правила улаштування електроустановок». Оскільки приміщення відноситься до класу без підвищеної небезпеки, захист від ураження струмом забезпечується шляхом використання обладнання І класу захисту із заземленням корпусу через євророзетку та пристроїв захисного вимкнення в електрощитку. Електропроводка має бути стаціонарною, трипровідною. Забороняється використання саморобних подовжувачів та експлуатація пошкоджених кабелів живлення.

Пожежна безпека забезпечується відповідно до НАПБ А.01.001-2014 Кодексу цивільного захисту України та Правил пожежної безпеки в Україні. Приміщення з комп’ютерною технікою зазвичай відноситься до категорії пожежної небезпеки «В». Приміщення обладнується автоматичними пожежними сповіщувачами. В якості первинних засобів пожежогасіння слід використовувати вуглекислотні вогнегасники типу ВВК-2 або ВВК-3.5, які не пошкоджують електронне обладнання та ефективні при гасінні електроустановок під напругою до 1000 В. Шляхи евакуації мають бути вільними та позначеними відповідними знаками згідно з ДСТУ EN ISO 7010:2019.

У підсумку, аналізуючи умови праці, ми змогли ідентифікувати головні фактори ризику для інженера-програміста. Впровадження заходів відповідно до розглянутих нормативів (наказу № 207, ДБН В.2.5-28:2018 та стандартів серії ДСТУ EN) гарантує створення безпечного та ергономічного робочого середовища, що сприяє збереженню здоров’я та високій продуктивності праці.

4.2. Оцінка дії електромагнітного поля, хвиль, імпульсу на людину та апаратуру, способи захисту

У сучасних умовах експлуатації комп’ютерних систем, центрів обробки даних та телекомунікаційного обладнання електромагнітна обстановка є важливим

фактором надійності. Для апаратної частини основною проблемою є забезпечення електромагнітної сумісності, оскільки зростання щільності монтажу транзисторів на кристалах мікропроцесорів та зниження робочої напруги робить їх надзвичайно чутливими до зовнішніх полів. Порушення електромагнітної сумісності може призводити як до катастрофічних відмов, так і до функціональних збоїв. Особливу небезпеку становить дія потужних електромагнітних імпульсів, що виникають внаслідок грозових розрядів, комутаційних процесів у силових мережах або спеціального впливу. Такі імпульси індукують у провідниках друкованих плат та з'єднувальних кабелях високовольтні перенапруги, здатні миттєво зруйнувати p-n переходи напівпровідникових елементів, пошкодити діелектричні шари конденсаторів та розплавити мікроскопічні струмопровідні доріжки. Рівні захисту об'єктів інформатизації від таких загроз суворо регламентуються серією стандартів ДСТУ EN 62305 «Блискавкозахист», яка визначає методи оцінки ризиків та параметри зон блискавкозахисту.

Водночас, менш потужні, але тривалі електромагнітні поля викликають явище електромагнітної інтерференції. Це призводить спотворення бітових послідовностей при передачі пакетів даних, випадкового перемикання логічних станів у комірках пам'яті, хибних спрацювань сенсорів та зависання серверного обладнання. Для забезпечення стабільної роботи IT-інфраструктури в умовах промислових та радіочастотних завад обладнання повинно відповідати вимогам серії стандартів ДСТУ EN 61000 «Електромагнітна сумісність». Зокрема, стандарт ДСТУ EN 61000-4-3 описує вимоги до стійкості технічних засобів до випромінюваного радіочастотного електромагнітного поля, а ДСТУ EN 61000-6-2 встановлює норми завадостійкості для промислового середовища.

Окрім ризиків для техніки, важливим аспектом безпеки є оцінка впливу електромагнітних полів на інженерний та обслуговуючий персонал. Специфіка роботи фахівців комп'ютерної інженерії передбачає тривале перебування поблизу джерел випромінювання, таких як серверні шафи, джерела безперебійного живлення, антени Wi-Fi та радіорелейний зв'язок. Біологічна дія електромагнітних полів на людину є кумулятивною і проявляється через тепловий ефект та

специфічний біотропний вплив на центральну нервову, імунну та серцево-судинну системи. Це може спричинити зниження когнітивних здібностей, розлади сну та професійні захворювання. В Україні контроль за безпекою персоналу здійснюється відповідно до ДСанПіН 3.3.6.096-2002 «Державні санітарні норми і правила при роботі з джерелами електромагнітних полів». Крім того, з метою гармонізації з європейськими директивами застосовується стандарт ДСТУ EN 50499:2012, що визначає єдину процедуру оцінювання впливу електромагнітних полів на працівників безпосередньо на робочих місцях.

Для мінімізації негативного впливу електромагнітних полів на інформаційну інфраструктуру та персонал застосовується комплексний підхід, що поєднує схемотехнічні, конструктивні та організаційні рішення. Організаційні заходи передбачають розробку і планування дій керівного та командно-начальницького складу, підрозділу з питань ЦЗ, служб і формувань ЦЗ щодо захисту робітників і службовців [30]. Основні інженерні методи захисту можна класифікувати наступним чином:

– екранування та використання захищених середовищ передачі передбачає організацію кліток Фарадея та застосування екранованих кабелів відповідно до вимог ДСТУ EN 50173 «Інформаційні технології. Структуровані кабельні системи», що за рахунок відбивання та поглинання електромагнітної енергії дозволяє суттєво знизити рівень наведених синфазних і диференційних завад;

– еквіпотенціальне з’єднання та заземлення створює єдину низькоімпедансну систему заземлення, проектування якої регламентується ДСТУ Б В.2.5-82:2016 та спеціалізованим стандартом ДСТУ EN 50310, шо є головною умовою для ефективного стікання паразитних струмів у землю та вирівнювання потенціалів між корпусами телекомунікаційного обладнання;

– встановлення пристроїв захисту від імпульсних перенапруг забезпечує обмеження амплітуди перенапруг до безпечного рівня, гарантуючи збереження вхідних каскадів мережевих карт та блоків живлення; для стійкого постачання об’єктів енергією необхідно створювати автономні резервні джерела

електропостачання (пересувні ДЕС на залізничних або автомобільних платформах) [30].

Отже, в даному розділі ми визначили, що забезпечення стійкості комп’ютерної системи до електромагнітних впливів – це інтегрований процес, який закладається ще на етапі проектування. Він вимагає вибору сертифікованого обладнання, що пройшло випробування на електромагнітну сумісність, та суворого дотримання нормативних вимог щодо монтажу кабельних систем і заземлення.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було досягнуто головної мети – спроєктовано та практично реалізовано інтегровану систему MLOps, яка забезпечує наскрізну автоматизацію повного життєвого циклу моделі машинного навчання. Для досягнення поставлених цілей було здійснено наступне:

1) в рамках дослідження проведено глибокий аналіз сучасних підходів до операціоналізації ML, виявлено критичну необхідність переходу від традиційних DevOps-практик до спеціалізованої методології MLOps;

2) проаналізовано вимоги до системи, яка використовує ML-засоби у своїй інфраструктурі та сформульовано завдання, які мають бути розв’язані в рамках MLOps-підходу;

3) проаналізовано наявні архітектурні патерни, методології автоматизації, що дало змогу обґрунтувати архітектуру платформи, яка поєднує три ключові парадигми: безперервну інтеграцію CI/CD на базі GitLab для автоматизації збірки та перевірки коду, GitOps на базі Argo CD для декларативного керування інфраструктурою та Service Mesh на базі Linkerd для забезпечення безпеки і спостережності;

4) на основі запропонованих методологій було здійснено практичну реалізацію системи з використання ML-сервісів для чого було розгорнуто локальний кластер Kubernetes, налаштовано інфраструктуру за допомогою Terraform, розроблено повний програмний стек для моделі аналізу тональності, а також, створено автоматизований конвеєр, який охоплює версіонування даних за допомогою DVC, відстеження експериментів у MLflow, статичний аналіз коду в SonarQube, контейнеризацію сервісу та його автоматичне розгортання;

5) на основі проаналізованих практик впроваджено підхід GitOps, який дозволив досягти повної синхронізації між станом репозиторію та кластером, забезпечуючи самовідновлення системи, а також використано архітектурний патерн Service Mesh, що дало можливість забезпечення прозорого шифрування трафіку за допомогою mTLS та збору “золотих сигналів” моніторингу без зміни коду додатку;

6) проведено верифікацію запропонованих підходів шляхом “end-to-end”-тестування, а саме перевірено коректність роботи всіх компонентів, починаючи від успішного проходження конвеєра CI/CD закінчуючи коректною обробкою запитів і розгорнутим API-сервісом.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Луцик Н.С., Луцків А.М., Осухівська Г.М., Тиш Є.В. Програма та методичні рекомендації з проходження практики за тематикою кваліфікаційної роботи для студентів спеціальності 123 «Комп'ютерна інженерія» другого (магістерського) рівня вищої освіти усіх форм навчання. Тернопіль: ТНТУ. 2024. 45 с.

2. Луцик Н.С., Луцків А.М., Осухівська Г.М., Тиш Є.В. Методичні рекомендації до виконання кваліфікаційної роботи магістра для студентів спеціальності 123 «Комп'ютерна інженерія» другого (магістерського) рівня вищої освіти усіх форм навчання. Тернопіль. 2024. 44 с.

3. Варавін А.В., Лещишин Ю.З., Чайковський А.В. Методичні вказівки до виконання курсового проєкту з дисципліни «Дослідження і проєктування комп'ютерних систем та мереж» для здобувачів другого (магістерського) рівня вищої освіти спеціальності 123 «Комп'ютерна інженерія» усіх форм навчання. Тернопіль: ТНТУ, 2024. 32 с.

4. А. Луцків, С. Андруньків. Розробка системи моніторингу семантичної якості та галюцинацій для моделей LLM в MLOps. Актуальні задачі сучасних технологій: Праці XIV наук.-прак. конф. (Тернопіль, 11-12 грудня 2025 р.), Тернопіль, 2025, с. 295.

5. А. Луцків, С. Андруньків. Інтерпретація чорних скриньок. Використання методів SHAP та LIME для візуалізації процесу прийняття рішень у глибоких нейронних мережах. Інформаційні моделі, системи та технології: Праці XIII наук.-техн. конф. (Тернопіль, 17-18 грудня 2025 р.), Тернопіль, 2025, с. 54.

6. Lutskiv A., Popovych N. Big data-based approach to automated linguistic analysis effectiveness. IEEE Third International Conference on Data Stream Mining & Processing. August 21- 25, 2020, Lviv, Ukraine pp.438–443.

7. Lutskiv A. Lutsyshyn R. Corpus-Based Translation Automation in Adaptable Corpus Translation Module. Computational Linguistics and Intelligent Systems. Proc. 5th

Int. Conf. COLINS 2021. Volume I: Workshop. Lviv, Ukraine, April 22-23, 2021, CEUR-WS.org, online. pp.374-395.

8. Yatsyshyn V., Kharchenko O., Lutskiv A. Maturity. Requirements Model for Software Requirements with the Implementation of ISO/IEC 25010 Recommendations. International Journal "Information Models and Analyses" Volume 9, Number 2, 2020. Pp.126-143.

9. Yatsyshyn V., Pastukh O., Lutskiv A., Tsymbalistyy V., Martsenko N. A Risks management method based on the quality requirements communication method in agile approaches / //Information Technologies: Theoretical and Applied Problems 2022 (ITTAP 2022), Ternopil, Ukraine, November 22-24, 2022. pp.1-10.

10. Burkov A. Machine Learning Engineering. Quebec: True Positive Inc., 2020. 270 p.

11. Treveil M. et al. Introducing MLOps: How to Scale Machine Learning in the Enterprise. Sebastopol: O'Reilly Media, 2020. 186 p.

12. Gift N., Deza A. Practical MLOps: Operationalizing Machine Learning Models. Sebastopol: O'Reilly Media, 2021. 458 p.

13. Субботін С. О. Нейронні мережі: теорія та практика: навч. посіб. Житомир: Вид-во О. О. Євенок, 2020. 184 с.

14. Гороховатський В.О., Творошенко І.С. Методи інтелектуального аналізу та оброблення даних: навч. посіб. Харків: ХНУРЕ, 2021. 92 с.

15. Yuen B., Matyushentsev A., Ekenstam T., Suen J. GitOps and Kubernetes: Continuous Deployment with Argo CD, Jenkins X, and Flux. Shelter Island: Manning Publications, 2021. 344 p.

16. Burns B., Beda J., Hightower K. Kubernetes: Up and Running. 2nd ed. Sebastopol: O'Reilly Media, 2019. 278 p.

17. Зінченко О.В., Іщеряков С.М., Прокопов С.В., Сєрих С.О., Василенко В.В. Хмарні технології: навч. посіб. Київ: ФОП Гуляєва В.М., 2020. 74 с.

18. Sayfan G. Mastering Kubernetes: Level up your container orchestration skills with Kubernetes. 3rd ed. Birmingham: Packt Publishing, 2020. 642 p.

19. Brikman Y. Terraform: Up and Running: Writing Infrastructure as Code. 2nd ed. Sebastopol: O'Reilly Media, 2019. 368 p.

20. Morgan J., Flynn. Linkerd: Up and Running. San Francisco: Buoyant, 2025. 534 p.

21. Литвин В. В., Пасічник В.В., Яцишин Ю.В. Інтелектуальні системи: підручник. Львів: Новий Світ-2000, 2020. 406 с.

22. Humble J., Farley D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Boston: Addison-Wesley Professional, 2010. 512 p.

23. Voron M. Building Data Science Applications with FastAPI. Birmingham: Packt Publishing, 2023. 422 p.

24. McKinney W. Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython. 2nd ed. Sebastopol: O'Reilly Media, 2017. 550 p.

25. GitLab CI: Feature Overview, Tutorial and Best Practice. URL: https://codefresh.io/learn/gitlab-ci/ (дата звернення: 25.11.2025).

26. Data Version Control (DVC): An Overview. URL: https://dvc.org/doc (дата звернення: 25.11.2025).

27. Argo CD: Declarative GitOps for Kubernetes. URL: https://argo-cd.readthedocs.io (дата звернення: 25.11.2025).

28. Linkerd: The World's Lightest Service Mesh. URL: https://linkerd.io (дата звернення: 25.11.2025).

29. Стручок В.С. Методичний посібник для здобувачів освітнього ступеня «магістр» всіх спеціальностей денної та заочної (дистанційної) форм навчання «БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ». Тернопіль: ФОП Паляниця В. А. 156 с.

30. Стручок В.С. Навчальний посібник «ТЕХНОЕКОЛОГІЯ ТА ЦИВІЛЬНА БЕЗПЕКА. ЧАСТИНА «ЦИВІЛЬНА БЕЗПЕКА»». Тернопіль: ФОП Паляниця В. А. 156 с.

ДОДАТОК А

Тези конференцій

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Тернопільський національний технічний університет імені Івана Пулюя
Маріборский університет (Словенія)
Технічний університет у Кошице (Словаччина)
Вільнюський технічний університет ім. Гедимінаса (Литва)
Краківський економічний університет (Польща)
Вроцлавський економічний університет (Польща)
Університет «Опольська Політехніка» (Польща)
Національний університет «Полтавська політехніка імені Юрія Кондратюка»
Вінницький національний аграрний університет
Львівський національний університет ім. І. Франка
Головне управління Пенсійного фонду в Тернопільській області
Наукове товариство ім. Шевченка
Тернопільський обласний комунальний інститут післядипломної педагогічної освіти
Сумський державний педагогічний університет
Запорізький національний університет

АКТУАЛЬНІ ЗАДАЧІ СУЧАСНИХ ТЕХНОЛОГІЙ

Збірник

тез доповідей

XIV Міжнародної науково-технічної конференції молодих учених та студентів

11-12 грудня 2025 року

УКРАЇНА
ТЕРНОПІЛЬ – 2025

УДК 004.048
А.Луцків канд. техн. наук, С. Андруньків
Тернопільський національний технічний університет імені Івана Пулюя

РОЗРОБКА СИСТЕМИ МОНІТОРИНГУ СЕМАНТИЧНОЇ ЯКОСТІ ТА ГАЛЮЦИНАЦІЙ ДЛЯ МОДЕЛЕЙ LLM В MLOPS

Andriy Lutskiv PhD., Assoc. Prof., Serhii Andrunkiv

DEVELOPMENT OF A SYSTEM FOR MONITORING SEMANTIC QUALITY AND HALLUCINATIONS FOR LLM MODELS IN MLOPS

Однією із ключових проблем MLOps для великих мовних моделей є відсутність надійних засобів автоматизованого моніторингу семантичної якості відповідей. На відміну від класичних моделей машинного навчання, де моніторинг фокусується на дрифті даних та зміні точності, LLM схильні до так званих "галюцинацій", при яких генерується фактологічно неправдива інформація, а також семантичної деградації, при якій проявляється токсичність та втрата тональності. Така поведінка створює значні ризики при їхньому використанні у виробничому середовищі.

В основі роботи лежить система (див. рис. 1), інтегрована в MLOps-цикл, що призначена для безперервного моніторингу LLM в режимі реального часу. Архітектурно вона складається з таких взаємопов'язаних компонентів:

– асинхронний логгер, який перехоплює та кешує пари "запит-відповідь" безпосередньо з виробничого сервісу LLM, не впливаючи на затримку для кінцевого користувача;

– підсистема оцінки якості, що асинхронно обробляє збережені дані і складається з детектора галюцинацій для валідації фактологічної коректності відповідей, класифікатора семантичної якості для оцінки токсичності та аналізатора безпеки для виявлення зловмисних запитів;

– експортер метрик, який агрегує результати роботи евалюаторів у кількісні показники, такі як відсоток галюцинацій, середній бал токсичності, після чого експортує їх у часову базу даних.

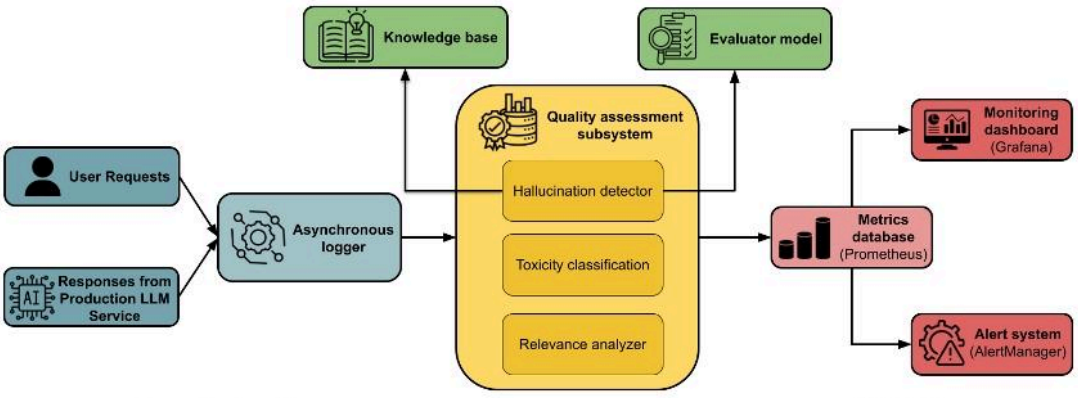

Рис. 1. Архітектурна схема системи моніторингу якості LLM

На основі зібраних даних формуються дашборди, що візуалізують тренди семантичної якості, безпеки та рівня галюцинацій. Система також інтегрована з механізмами сповіщень, які миттєво реагують на раптову деградацію поведінки моделі та виявляють проблеми до того, як вони вплинуть на користувачів.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ТЕРНОПІЛЬСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ІМЕНІ ІВАНА ПУЛЮЯ

М А Т Е Р І А Л И

XIII НАУКОВО-ТЕХНІЧНОЇ КОНФЕРЕНЦІЇ

«ІНФОРМАЦІЙНІ МОДЕЛІ, СИСТЕМИ ТА ТЕХНОЛОГІЇ»

17-18 грудня 2025 року

ТЕРНОПІЛЬ
2025

УДК 004.048

А. Луцків, к. т. н., доц.; С. Андруньків
(Тернопільський національний технічний університет імені Івана Пулюя, Україна)

ІНТЕРПРЕТАЦІЯ 'ЧОРНИХ СКРИНЬОК'. ВИКОРИСТАННЯ МЕТОДІВ SHAP ТА LIME ДЛЯ ВІЗУАЛІЗАЦІЇ ПРОЦЕСУ ПРИЙНЯТТЯ РІШЕНЬ У ГЛИБОКИХ НЕЙРОННИХ МЕРЕЖАХ

UDC 004.048

A. Lutskiv, PhD., Assoc. Prof.; S. Andrunkiv

INTERPRETATION OF 'BLACK BOXES'. USE OF SHAP AND LIME METHODS TO VISUALIZE THE DECISION-MAKING PROCESS IN DEEP NEURAL NETWORKS

Глибокі нейронні мережі досягли надзвичайної точності у складних задачах, таких як розпізнавання зображень, обробка мови та медична діагностика. Однак їхня внутрішня складність, що складається з мільйонів параметрів, перетворює їх на «чорні скриньки». Відсутність прозорості та неможливість пояснити, чому модель прийняла певне рішення, є головною перешкодою для їхнього впровадження у сфери медицини, фінансів, юриспруденції, де довіра та валідація є обов'язковими.

Для досягнення мети було проаналізовано:

– метод LIME, Local Interpretable Model-agnostic Explanations, який будує просту лінійну модель навколо одного прогнозу, щоб пояснити його локально;

– метод SHAP, SHapley Additive exPlanations, який базується на теорії ігор для обчислення точного внеску кожної ознаки у кінцевий прогноз, забезпечуючи локальну та глобальну інтерпретацію.

LIME продемонстрував здатність швидко надавати інтуїтивно зрозумілі візуальні пояснення для окремих прогнозів, виділяючи «суперпікселі» на зображенні, які найбільше вплинули на рішення (див. рис. 1, зліва). На зображенні зеленою маскою виділені суперпікселі, що відповідають за правильну класифікацію «хаскі», а червоною – ті, що заважають, та які модель теж асоціює з «хаскі».

SHAP надав докладніші та стійкіші пояснення. На рисунку 1, праворуч, схожа візуалізація. На зображенні червоні пікселі показують, що очі та морда собаки підвищили ймовірність класу «хаскі», а сині, що вказують на білу шерсть та фон – знизили. Ця візуалізація часто є чіткішою та менш «шумною», ніж LIME.

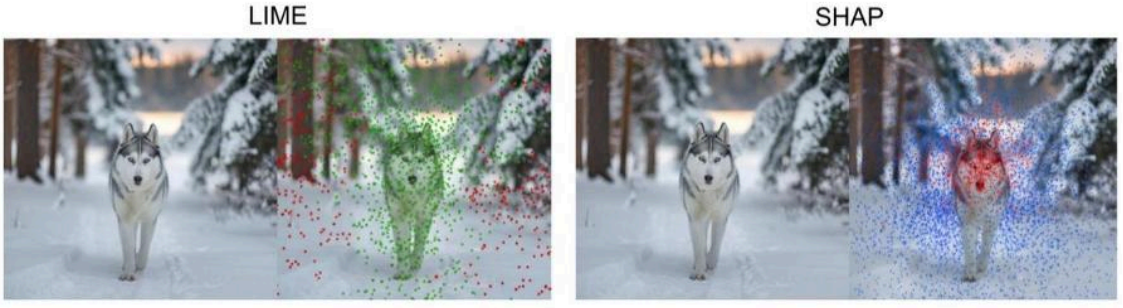

Рисунок 1. Порівняння візуалізацій LIME та SHAP для пояснення класифікації зображення

ДОДАТОК Б

Скрипти тренування моделі, генерування артефактів та опису конвеєра CI/CD

Скрипт тренування моделі та генерування артефактів “train.py”:

```
import pandas as pd
import re
import pickle
import mlflow
import nltk
from nltk.corpus import stopwords
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, f1_score
import os

# Download NLTK stopwords
nltk.download('stopwords')
stop_words = set(stopwords.words('english'))

# Define text cleaning function
def clean_text(text):
    text = re.sub(r'<[^>]+>', '', text) # Remove HTML tags
    text = re.sub(r'[^a-zA-Z]', ' ', text) # Keep only letters
    text = text.lower()
      text = ' '.join([word for word in text.split() if word not in
stop_words])
    return text

# Create artifact directory
os.makedirs('model', exist_ok=True)

# Start MLflow Run
with mlflow.start_run():
    # 1. Load data
    print("Loading data...")
    df = pd.read_csv('data/reviews.csv')
    df = df.sample(10000, random_state=42) # Sample for speed
       df['sentiment'] = df['sentiment'].apply(lambda x: 1 if x ==
'positive' else 0)

    # 2. Preprocessing
    print("Cleaning text...")
    df['cleaned_text'] = df['review'].apply(clean_text)

    # 3. Vectorization (TF-IDF)
    print("Vectorizing...")
    vectorizer = TfidfVectorizer(max_features=5000)
    X = vectorizer.fit_transform(df['cleaned_text'])
    y = df['sentiment']
```

```
    # Save the vectorizer
    pickle.dump(vectorizer, open('model/vectorizer.pkl', 'wb'))
    mlflow.log_artifact('model/vectorizer.pkl')

    # Split data
        X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

    # 4. Train Model (Naive Bayes)
    print("Training model...")
    model = MultinomialNB()
    model.fit(X_train, y_train)

    # 5. Evaluate Model
    preds = model.predict(X_test)
    acc = accuracy_score(y_test, preds)
    f1 = f1_score(y_test, preds)

    print(f"Accuracy: {acc}")
    print(f"F1-Score: {f1}")

    # 6. Log metrics and params to MLflow
    mlflow.log_param("model_type", "MultinomialNB")
    mlflow.log_param("max_features", 5000)
    mlflow.log_metric("accuracy", acc)
    mlflow.log_metric("f1_score", f1)

    # 7. Save the model
    pickle.dump(model, open('model/model.pkl', 'wb'))
    mlflow.sklearn.log_model(model, "sentiment_model")

print("Training complete.")
```

Скрипт реалізації REST API на базі FastAPI “app.py”:

```
import pickle
import os
from fastapi import FastAPI
from pydantic import BaseModel
from prometheus_fastapi_instrumentator import Instrumentator

# Define absolute paths for model files
BASE_DIR = os.path.dirname(os.path.abspath(__file__))
MODEL_DIR = os.path.join(BASE_DIR, "..", "model")
VECTORIZER_PATH = os.path.join(MODEL_DIR, "vectorizer.pkl")
MODEL_PATH = os.path.join(MODEL_DIR, "model.pkl")

app = FastAPI(title="Sentiment Analysis API")

# Load models on application startup
try:
    with open(VECTORIZER_PATH, 'rb') as f:
```

```
        vectorizer = pickle.load(f)
    with open(MODEL_PATH, 'rb') as f:
        model = pickle.load(f)
    print("Model and vectorizer loaded.")
except Exception as e:
    print(f"Error loading models: {e}")
    vectorizer, model = None, None

# Define the expected request data structure
class TextRequest(BaseModel):
    text: str

# Root endpoint for health check
@app.get("/")
def root():
      return {"message": "Sentiment Analysis API is running. Check
/docs"}

# Main prediction endpoint
@app.post("/predict")
def predict(request: TextRequest):
    if not model or not vectorizer:
        return {"error": "Model not loaded"}, 503

    # Preprocessing (must match training)
    text = request.text.lower()

    # Vectorize text and make prediction
    vectorized_text = vectorizer.transform([text])
    prediction = model.predict(vectorized_text)[0]
    sentiment = "positive" if prediction == 1 else "negative"

    return {"text": request.text, "sentiment": sentiment}

# Enable Prometheus monitoring
# This runs once on startup to expose a /metrics endpoint
@app.on_event("startup")
async def startup():
    Instrumentator().instrument(app).expose(app)
```

Скрипт YAML-опису конвеєра CI/CD “.gitlab-ci.yml”:

```
---
stages:
  - validate
  - train
  - build
  - deploy

variables:
  # Define a unique image tag for each commit
  IMAGE_TAG: $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA
  # SSH URL for the GitOps config repository
```

```
                                                                CONFIG_REPO_URL:
"git@gitlab.com:sergii_andrunkiv/sentiment-config-repo.git"

# Stage 1: Code Quality Check
sonar_scan:
  stage: validate
  image: sonarsource/sonar-scanner-cli:latest
  script:
    # Run SonarScanner analysis
                -    sonar-scanner    -D"sonar.projectKey=sentiment-app"
-D"sonar.sources=./src"              -D"sonar.host.url=$SONAR_HOST_URL"
-D"sonar.login=$SONAR_TOKEN"

# Stage 2: Model Training
train_model:
  stage: train
  image: python:3.9-slim
  tags: [docker]
  script:
    - pip install -r requirements.txt
    - dvc pull
    - python src/train.py
  artifacts:
    paths:
      - model/
      - mlruns/

# Stage 3: Docker Image Build
build_image:
  stage: build
  image: docker:20.10
  services:
    - docker:20.10-dind
  tags: [docker]
  script:
    - >
      echo $CI_REGISTRY_PASSWORD |
      docker login -u $CI_REGISTRY_USER $CI_REGISTRY --password-stdin
    - docker build -t $IMAGE_TAG .
    - docker push $IMAGE_TAG
  dependencies:
    - train_model

# Stage 4: GitOps (Update Config Repo)
deploy_gitops:
  stage: deploy
  image: alpine:latest
  tags: [docker]
  before_script:
    - 'apk add --no-cache git openssh-client sed base64'
    - 'mkdir -p ~/.ssh'
    - 'echo "$GIT_SSH_KEY" | base64 -d > ~/.ssh/id_rsa'
    - 'chmod 600 ~/.ssh/id_rsa'
```

```
        - 'echo -e "Host *\n\tStrictHostKeyChecking no\n\n" >
~/.ssh/config'
    - 'git config --global user.email "gitlab-ci-bot@your.domain"'
    - 'git config --global user.name "GitLab CI Bot"'
  script:
    - 'git clone "$CONFIG_REPO_URL" /tmp/config-repo'
    - 'CONFIG_FILE="/tmp/config-repo/k8s/deployment.yaml"'
    - 'sed -i "s|image:.*|image: $IMAGE_TAG|g" "$CONFIG_FILE"'
    - 'cd /tmp/config-repo'
    - 'git add .'
    - 'git commit -m "CI: Update image tag to $CI_COMMIT_SHORT_SHA" ||
echo "No changes to commit"'
    - 'git push'
  dependencies:
    - build_image
```