Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)

Навчально-науковий інститут комп'ютерних наук
та інформаційних технологій
(повне найменування інституту, назва факультету(відділення))

Кафедра комп'ютерних наук
(повна назва кафедри (предметної циклової комісії))

Магістерська кваліфікаційна робота

другий (магістерський)
(рівень вищої освіти)

на тему: «Інфраструктура як код (IaC) та CI/CD для хмарних сервісів на базі Azure та Kubernetes»

Виконав студент 6 курсу, групи КН-63
спеціальності:
122 „Комп'ютерні науки"
(шифр і назва напрямку підготовки спеціальності)

Кирьяков Дмитро Дмитрович
(прізвище, ім'я, по батькові)

Керівники: Борецька І.Б.
(прізвище, ініціали)

Рецензент: Сторожук О.Л.
(прізвище, ініціали)

Львів-2025

Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)

ННІ комп'ютерних наук та інформаційних технологій
Кафедра комп'ютерних наук
Рівень вищої освіти другий (магістерський)
Спеціальність 122 "Комп'ютерні науки"

ЗАТВЕРДЖУЮ:
Завідувачка кафедри КН

Борецька І.Б.
„10" грудня 2025 р.

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Кирьяков Дмитру Дмитровичу
(прізвище, ім'я, по батькові)

1.Тема роботи: "Інфраструктура як код (IaC) та CI/CD для хмарних сервісів на базі Azure та Kubernetes"
керівник роботи *Борецька Ірина Богданівна, к. т. н., доцент*
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від "29" квітня 2025 року № С-288
2.Термін подання студентом проекту(роботи) 10 грудня 2025 р.
3. Вихідні дані до роботи Розробити інфраструктурне та програмне забезпечення для автоматизованого розгортання контейнерних сервісів у хмарному середовищі Microsoft Azure на базі Azure Kubernetes Service.
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
Стан проблемної області
Інформаційне забезпечення
Математичне забезпечення
Програмне забезпечення
Розроблення стартап-проєкту
Висновки
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Підготовка матеріалу до доповіді.
6. Дата видачі завдання 1 травня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Етапи бакалаврської роботи	Термін виконання	Відмітка про виконання
1.	Аналіз предметної області	02.05.25 – 03.05.25	Виконано
2.	Формування вимог, постановка задачі та визначення архітектури системи	14.05.25 – 20.06.25	Виконано
3.	Вибір технологій та інструментарію для реалізації	12.07.25 – 17.07.25	Виконано
4.	Розроблення інформаційного забезпечення	20.07.25 – 01.08.25	Виконано
5.	Розроблення математичного забезпечення	02.08.25 – 10.08.25	Виконано
6.	Розроблення та налагодження Terraform-конфігурацій	11.08.25 – 01.09.25	Виконано
7.	Розроблення CI/CD-конвеєрів у Azure DevOps	02.09.25 – 01.10.25	Виконано
8.	Проведення тестування, експериментальних досліджень	02.10.25 – 20.11.25	Виконано
9.	Оформлення пояснювальної записки	01.12.25 – 09.12.25	Виконано

Студент (підпис) Кирьяков Д.Д. (прізвище та ініціали)

Керівник роботи (підпис) Борецька І.Б. (прізвище та ініціали)

АНОТАЦІЯ

Дипломна робота містить 82 сторінок пояснювальної записки, 9 рисунків, 2 додатки, 18 джерел.

У магістерській кваліфікаційній роботі розглянуто застосування підходів «Інфраструктура як код» (Infrastructure as Code, IaC) та безперервної інтеграції й постачання (CI/CD) для побудови та керування хмарними сервісами на базі платформи Microsoft Azure та Kubernetes (Azure Kubernetes Service, AKS). Метою роботи є розроблення цілісного підходу до опису, розгортання та супроводу хмарної інфраструктури й прикладних сервісів, який забезпечує відтворюваність середовищ, зменшення обсягу ручних операцій і підвищення надійності системи.

У роботі проаналізовано сучасний стан проблемної області, сформовано вимоги до цільової інфраструктури та інформаційних потоків, побудовано модель життєвого циклу декларативної інфраструктури й CI/CD-конвеєрів, визначено ключові DevOps-метрики та підхід до оцінювання якості процесів. Практична частина включає проєктування й реалізацію модульної структури Terraform-конфігурацій для Azure-ресурсів, розроблення YAML-конвеєрів у Azure DevOps для автоматичного розгортання інфраструктури та контейнеризованих застосунків в AKS, а також інтеграцію стеку моніторингу й логування на базі Prometheus, Grafana та Loki.

Запропоноване рішення може бути використане як типове ядро для побудови DevOps-інфраструктури у хмарних проєктах, скорочує час розгортання нових середовищ, зменшує ймовірність конфігураційних помилок і спрощує подальший розвиток системи.

Ключові слова: Infrastructure as Code, Terraform, Azure, Azure DevOps, Kubernetes, AKS, CI/CD, GitOps, Prometheus, Grafana, Loki.

ABSTRACT

The thesis contains 82 pages of explanatory note, 9 figures, 2 appendices, 18 sources.

In this master's thesis, the approaches of Infrastructure as Code (IaC) and Continuous Integration / Continuous Delivery (CI/CD) are applied to the design and operation of cloud services based on Microsoft Azure and Kubernetes (Azure Kubernetes Service, AKS). The aim of the research is to develop an end-to-end approach to describing, deploying and maintaining cloud infrastructure and application services that ensures environment reproducibility, reduces manual operations and increases system reliability.

The thesis analyses the current state of the problem domain, defines the requirements for the target infrastructure and information flows, builds a model of the life cycle of declarative infrastructure and CI/CD pipelines, and identifies key DevOps metrics and methods for evaluating process quality. The practical part includes the design and implementation of a modular structure of Terraform configurations for Azure resources, the development of YAML pipelines in Azure DevOps for automatic deployment of infrastructure and containerized applications to AKS, as well as the integration of a monitoring and logging stack based on Prometheus, Grafana and Loki.

The proposed solution can be used as a reusable core for building DevOps infrastructure in cloud projects, reducing the time required to provision new environments, decreasing the probability of configuration errors and simplifying further system evolution.

Keywords: Infrastructure as Code, Terraform, Azure, Azure DevOps, Kubernetes, AKS, CI/CD, GitOps, Prometheus, Grafana, Loki.

ТЕХНІЧНЕ ЗАВДАННЯ

Мета роботи: розробити повноцінний приклад застосування підходів Infrastructure as Code (IaC) та CI/CD для автоматизованого керування хмарною інфраструктурою й сервісами на базі Microsoft Azure та Azure Kubernetes Service (AKS), що забезпечує відтворюваність середовищ, керованість змін та інтегроване моніторинг-спостереження.

Необхідно виконати такі завдання:

1. Проаналізувати вимоги до цільової хмарної інфраструктури, середовищ розробки/тестування/продакшену та процесів розгортання.
2. Розробити архітектуру рішення для побудови та супроводу інфраструктури й CI/CD-конвеєрів у середовищі Microsoft Azure та AKS.
3. Спроєктувати модульну структуру Terraform-конфігурацій для створення й керування основними ресурсами Azure (ресурсні групи, мережа, ACR, AKS, Key Vault, засоби моніторингу).
4. Налаштувати CI/CD-конвеєри в Azure DevOps для: побудови та публікації Docker-образів у ACR; автоматизованого розгортання інфраструктури за допомогою Terraform; розгортання застосунків у AKS на базі Helm-чартів.
5. Реалізувати розгортання стеку моніторингу та логування (Prometheus, Grafana, Loki) та інтегрувати його з AKS і тестовим застосунком.
6. Визначити й описати ключові DevOps-метрики (Lead Time, Deployment Frequency, MTTR, Change Failure Rate), розробити приклади дашбордів для їх відстеження.
7. Провести функціональне тестування розробленого рішення, перевірити коректність автоматизованих процесів розгортання та можливість відновлення інфраструктури «з нуля» на основі коду.
8. Підготувати документацію з описом архітектури, структури репозиторію, сценаріїв використання та рекомендацій щодо подальшого розвитку рішення.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ 7
ВСТУП 8
РОЗДІЛ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ 10
1.1 DevOps та концепція «Інфраструктура як код» 10
1.2 Платформа Microsoft Azure для побудови хмарних сервісів 12
1.3 Kubernetes та Azure Kubernetes Service як платформа розгортання 14
1.4 Огляд наявних інструментів IaC та CI/CD для хмарних рішень 16
Висновки до розділу 18
РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ 19
2.1 Опис предметної області та вимоги до хмарної інфраструктури 19
2.2 Архітектура цільового рішення в Microsoft Azure 21
2.3 Інформаційні потоки та артефакти (стан інфраструктури, конфігурації, журнали, метрики) 23
2.4 Моделі користувачів, ролей і доступу в системі IaC та CI/CD 25
2.5. Вимоги до надійності, безпеки, масштабованості та спостережуваності 27
Висновки до розділу 30
РОЗДІЛ 3. МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ 31
3.1 Формальна модель життєвого циклу декларативної інфраструктури 31
3.2 Модель графа залежностей ресурсів та ідемпотентність операцій 33
3.3 Моделювання CI/CD-конвеєра як послідовності етапів та станів 34
3.4. DevOps-метрики та методика їх оцінювання в хмарних середовищах 36
3.5 Моделювання показників надійності (SLO, SLA, error budget) для сервісів в AKS 38
Висновки до розділу 41
РОЗДІЛ 4. ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ 42
4.1 Вибір та обґрунтування технологічного стеку (Terraform, Azure DevOps, Helm, Prometheus, Grafana, Loki) 42
4.2 Проєктування та реалізація Terraform-модулів для Azure-ресурсів 43
4.3 Реалізація CI/CD-конвеєрів у Azure DevOps для інфраструктури та прикладних сервісів 46

4.4 Організація GitOps-підходу до розгортання застосунків у AKS на базі Helm-чартів 49
4.5 Інтеграція систем моніторингу, логування та алертингу з AKS 50
4.6 Тестування, верифікація та експериментальні дослідження розробленого рішення 53
Висновки до розділу 56
РОЗДІЛ 5. РОЗРОБЛЕННЯ СТАРТАП-ПРОЄКТУ 57
5.1 Опис ідеї стартап-платформи IaC та CI/CD-як-сервіс (Platform as a Service) 57
5.2 Аналіз ринку, конкурентів та цільової аудиторії 58
5.3 Розроблення ринкової стратегії та моделі монетизації 60
5.4 Вимоги до технічного та програмного забезпечення стартап-рішення 62
5.5 Оцінка ризиків та дорожня карта розвитку проєкту 65
Висновки до розділу 68
ВИСНОВКИ 69
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 72
ДОДАТОК А - Приклади Terraform-конфігурацій для розгортання інфраструктури 74
ДОДАТОК Б - YAML-опис CI/CD-конвеєрів у Azure DevOps 78

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

AKS – Azure Kubernetes Service, керований сервіс Kubernetes у хмарі Microsoft Azure.

API – Application Programming Interface, програмний інтерфейс прикладних програм.

ARM – Azure Resource Manager, платформа керування ресурсами Microsoft Azure.

CI – Continuous Integration, безперервна інтеграція.

CI/CD – Continuous Integration / Continuous Delivery (Deployment), безперервна інтеграція та постачання (розгортання).

CPU – Central Processing Unit, центральний процесор.

DevOps – підхід до організації розробки та експлуатації програмного забезпечення, що поєднує Development та Operations.

DNS – Domain Name System, система доменних імен.

IaC – Infrastructure as Code, інфраструктура як код. IP – Internet Protocol, протокол міжмережевої взаємодії.

K8s – скорочена назва Kubernetes.

KPI – Key Performance Indicator, ключовий показник ефективності.

RBAC – Role-Based Access Control, керування доступом на основі ролей.

SLA – Service Level Agreement, угода про рівень сервісу.

SLO – Service Level Objective, цільовий показник рівня сервісу.

VM – Virtual Machine, віртуальна машина.

VNet – Virtual Network, віртуальна мережа в Microsoft Azure.

YAML – Yet Another Markup Language, формат текстових конфігурацій.

ВСТУП

Розвиток хмарних технологій, мікросервісної архітектури та контейнеризації суттєво підвищив вимоги до швидкості та надійності постачання програмного забезпечення. Сучасні компанії працюють у мультихмарному середовищі, де кількість середовищ (development, testing, staging, production), сервісів та конфігурацій постійно зростає, а помилки в налаштуванні інфраструктури можуть призводити до простоїв, втрати даних та фінансових збитків. У цих умовах ручне керування хмарними ресурсами стає неефективним та ризикованим, а автоматизація інфраструктури перетворюється на необхідну передумову стабільного розвитку бізнесу.

Особливої актуальності набуває застосування керованих Kubernetes-сервісів, зокрема Azure Kubernetes Service (AKS), які дають змогу зосередитись на розробці та експлуатації прикладних сервісів замість підтримки обчислювальної платформи. Водночас, без системного підходу до опису інфраструктури та побудови конвеєрів безперервної інтеграції й постачання (CI/CD) навіть використання AKS не гарантує відтворюваності середовищ та контрольованості змін.

Магістерська робота присвячена розробленню та дослідженню комплексного підходу до побудови інфраструктури як коду й CI/CD для хмарних сервісів на базі Microsoft Azure та Kubernetes (Azure Kubernetes Service). У роботі розглядаються питання проєктування архітектури інфраструктури, організації модульних Terraform-конфігурацій, побудови конвеєрів у Azure DevOps для автоматизованого розгортання інфраструктури та контейнеризованих застосунків, а також інтеграції систем моніторингу та логування.

Об’єктом дослідження є процес побудови та експлуатації хмарної інфраструктури та сервісів на базі платформи Microsoft Azure з використанням Azure Kubernetes Service.

Предметом дослідження є моделі, методи та програмні засоби автоматизованого опису, розгортання і супроводу інфраструктури та прикладних сервісів із використанням підходів IaC, CI/CD та систем спостережуваності.

Метою роботи є розроблення, обґрунтування та практична реалізація комплексного шаблону інфраструктури як коду та CI/CD.

Для досягнення поставленої мети необхідно розв’язати такі завдання:

- проаналізувати сучасні підходи DevOps, IaC та CI/CD;
- сформувати вимоги до цільової хмарної інфраструктури та інформаційних потоків у Microsoft Azure;
- побудувати модель життєвого циклу декларативної інфраструктури та конвеєрів CI/CD;
- спроєктувати та реалізувати модульну структуру Terraform-конфігурацій;
- налаштувати CI/CD-конвеєри;
- інтегрувати стек моніторингу та логування;
- провести експериментальне дослідження розробленого рішення;

Наукова новизна роботи полягає у формуванні комплексного шаблону побудови інфраструктури як коду та CI/CD для Azure-орієнтованих рішень, який поєднує модульну структуру Terraform-конфігурацій, конвеєри Azure DevOps, GitOps-підхід до розгортання застосунків у AKS та інтегрований стек моніторингу й логування, а також у формалізації життєвого циклу декларативної інфраструктури через DevOps-метрики.

Практичне значення роботи полягає у можливості використання розробленого рішення як типового шаблону для корпоративних і стартап-проєктів, що працюють у Microsoft Azure. Запропонований підхід дозволяє скоротити час розгортання нових середовищ, зменшити кількість конфігураційних помилок, спростити супровід та масштабування хмарних сервісів, а також закласти основу для подальшої комерціалізації у форматі DevOps-платформи як сервісу.

РОЗДІЛ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ

1.1 DevOps та концепція «Інфраструктура як код»

Розвиток хмарних платформ, контейнеризації та мікросервісної архітектури призвів до істотного ускладнення процесів розробки й експлуатації програмних систем. Кількість сервісів, середовищ та взаємозв'язків між ними постійно зростає, що робить традиційні підходи до адміністрування інфраструктури на основі ручних операцій малоефективними. У відповідь на ці виклики сформувався підхід DevOps, який поєднує практики розробки (Development) та експлуатації (Operations) і спрямований на скорочення часу постачання змін до продуктивного середовища при одночасному підвищенні стабільності системи [1].

DevOps розглядається не лише як набір інструментів, а насамперед як культура взаємодії між командами, що включає спільну відповідальність за результат, автоматизацію типових операцій, прозорість процесів і безперервне вдосконалення (рис 1.1). Ключовими практиками DevOps є безперервна інтеграція (Continuous Integration, CI), безперервне постачання або розгортання (Continuous Delivery / Deployment, CD), автоматизоване тестування, інфраструктура як код (Infrastructure as Code, IaC), а також розвинуті засоби моніторингу та спостережуваності [2].

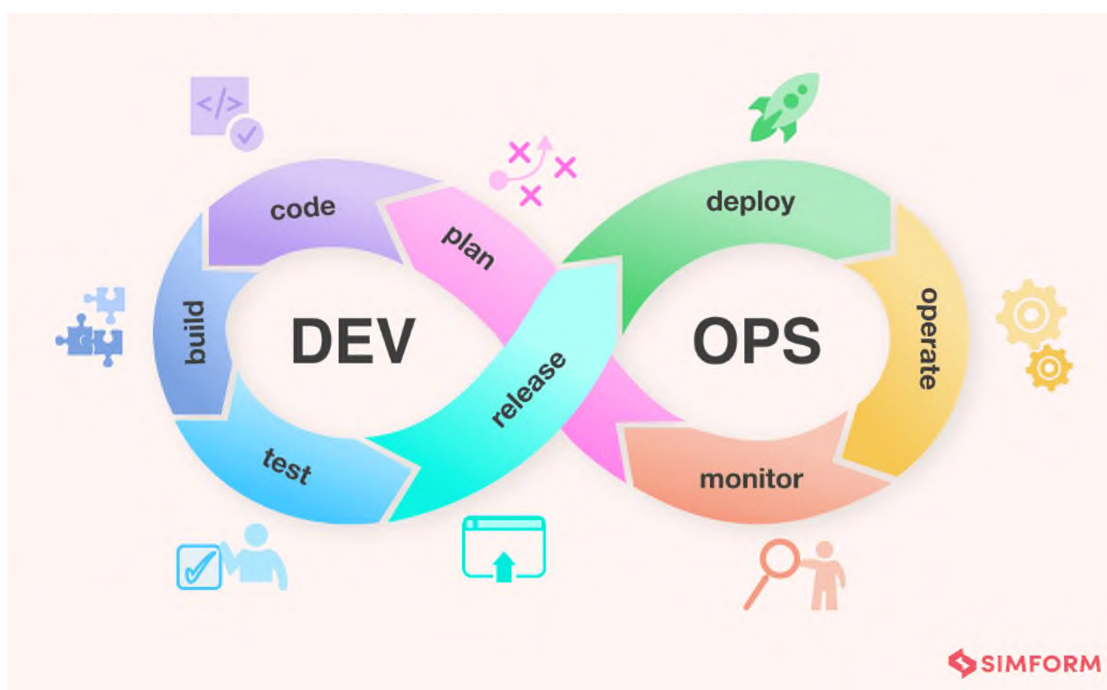

Рисунок 1.1 – Узагальнена схема життєвого циклу DevOps

Концепція «Інфраструктура як код» виникла як відповідь на проблему «дрейфу конфігурацій», коли середовища з часом починають відрізнятися одне від одного через ручні зміни, що складно відслідкувати й відтворити. Згідно з підходом IaC, інфраструктура (віртуальні машини, мережі, балансувальники навантаження, кластери Kubernetes, сховища даних тощо) описується у вигляді декларативних або імперативних конфігурацій, які зберігаються у системі контролю версій нарівні з програмним кодом. Зміни в інфраструктурі виконуються шляхом модифікації цього коду та його застосування за допомогою спеціалізованих інструментів [5].

До переваг підходу IaC належать:

- **відтворюваність** – можливість розгорнути однакові середовища (development, staging, production) на основі тих самих конфігурацій;
- **ідемпотентність** – багаторазове застосування одного й того самого опису інфраструктури не призводить до непередбачуваних змін стану;
- **прозорість та трасованість** – усі зміни фіксуються в історії комітів, що полегшує аудит і аналіз інцидентів;
- **автоматизація та швидкість** – зменшення кількості ручних операцій скорочує час розгортання та знижує ризик людських помилок;
- **узгодженість з практиками DevOps та GitOps** – інфраструктура керується тими самими принципами, що й прикладний код.

На практиці IaC реалізується за допомогою таких інструментів, як Terraform, Ansible, Pulumi, Azure Resource Manager (ARM) / Bicep та інших. У межах даної роботи базовим інструментом є Terraform, який дозволяє декларативно описувати ресурси різних хмарних провайдерів, у тому числі Microsoft Azure, а також спирається на граф залежностей між ресурсами для коректного порядку їх створення й оновлення [6].

Поєднання DevOps-підходу з IaC забезпечує керований життєвий цикл інфраструктури: від проєктування й опису конфігурацій до автоматизованого розгортання та експлуатації. При цьому IaC-код інтегрується в конвеєри CI/CD, що дозволяє застосовувати до інфраструктури ті самі принципи, що й до прикладного коду: код-рев’ю, тестування, автоматичні перевірки якості та поступове

впровадження змін. Це особливо важливо в середовищах на базі Kubernetes і керованих сервісів, таких як Azure Kubernetes Service (AKS), де кількість об'єктів конфігурації (deployment, service, ingress, secrets, configmap тощо) може бути дуже великою.

Таким чином, DevOps та концепція «Інфраструктура як код» утворюють теоретичну та методологічну основу для побудови масштабованих і надійних хмарних рішень. Саме на них базується підхід, який розробляється й досліджується в даній магістерській роботі.

1.2 Платформа Microsoft Azure для побудови хмарних сервісів

Microsoft Azure є однією з провідних публічних хмарних платформ, що надає широкий спектр сервісів для побудови, розгортання та експлуатації програмних систем. Azure підтримує різні моделі надання послуг (Infrastructure as a Service, Platform as a Service, Software as a Service) та пропонує розвинуту екосистему засобів для реалізації DevOps-підходу, включаючи засоби керування інфраструктурою, CI/CD, моніторинг та безпеку [7].

Базовою концепцією в Azure є **ресурс** (resource), який представляє окремий об'єкт хмарної інфраструктури, та **група ресурсів** (Resource Group), що використовується для логічного об'єднання пов'язаних ресурсів і керування ними як єдиним цілим. Ресурси описуються й керуються через Azure Resource Manager (ARM) – шар абстракції, який забезпечує узгоджене керування життєвим циклом об'єктів, застосування політик і контроль доступу.

Для побудови хмарних сервісів у контексті Kubernetes-кластера важливими є такі сервіси Azure:

- **Azure Virtual Network (VNet)** – віртуальна мережа, що забезпечує ізоляцію та маршрутизацію мережевого трафіку між ресурсами;
- **Azure Kubernetes Service (AKS)** – керований сервіс оркестрації контейнерів, який спрощує розгортання, масштабування й оновлення кластерів Kubernetes;
- **Azure Container Registry (ACR)** – приватний реєстр контейнерних образів, інтегрований з Azure та AKS;

- **Azure Key Vault** – сервіс для безпечного зберігання секретів, ключів і сертифікатів;
- **Azure Storage та інші сервіси зберігання** – для зберігання конфігурацій, артефактів і даних;
- **Azure Monitor, Log Analytics та пов'язані сервіси** – для збору метрик, логів і налаштування алертингу [7].

Окремий клас складають інструменти для організації процесів розробки й CI/CD. До них належать:

- **Azure DevOps** – набір сервісів (Repos, Pipelines, Boards, Artifacts), що забезпечують повний цикл керування розробкою та ручними/автоматизованими процесами розгортання [8];
- інтеграція з **GitHub** та GitHub Actions, що дозволяє будувати гібридні сценарії розробки й розгортання;
- підтримка сторонніх інструментів (наприклад, Terraform, Helm, kubectl, Prometheus/Grafana) через розширення та агентів.

Для реалізації підходу «Інфраструктура як код» у середовищі Azure можуть використовуватися як нативні засоби (ARM Templates, Bicep), так і крос-платформні рішення (Terraform, Pulumi). У межах даної роботи основний акцент зроблено на Terraform, який має офіційний провайдер для Azure й добре інтегрується з Azure DevOps. Це дозволяє реалізувати сценарії, коли зміни в репозиторії коду автоматично запускають конвеєри, що перевіряють, планують і застосовують зміни в інфраструктурі [5-7,17].

Важливою перевагою Microsoft Azure є можливість побудови **єдиної екосистеми**: від керування кодом і роботою команд до автоматизованого розгортання інфраструктури, кластерів AKS, контейнеризованих застосунків і систем моніторингу (рис 1.2). Завдяки цьому Azure виступає не лише як набір окремих сервісів, а як платформа для реалізації повного DevOps-циклу.

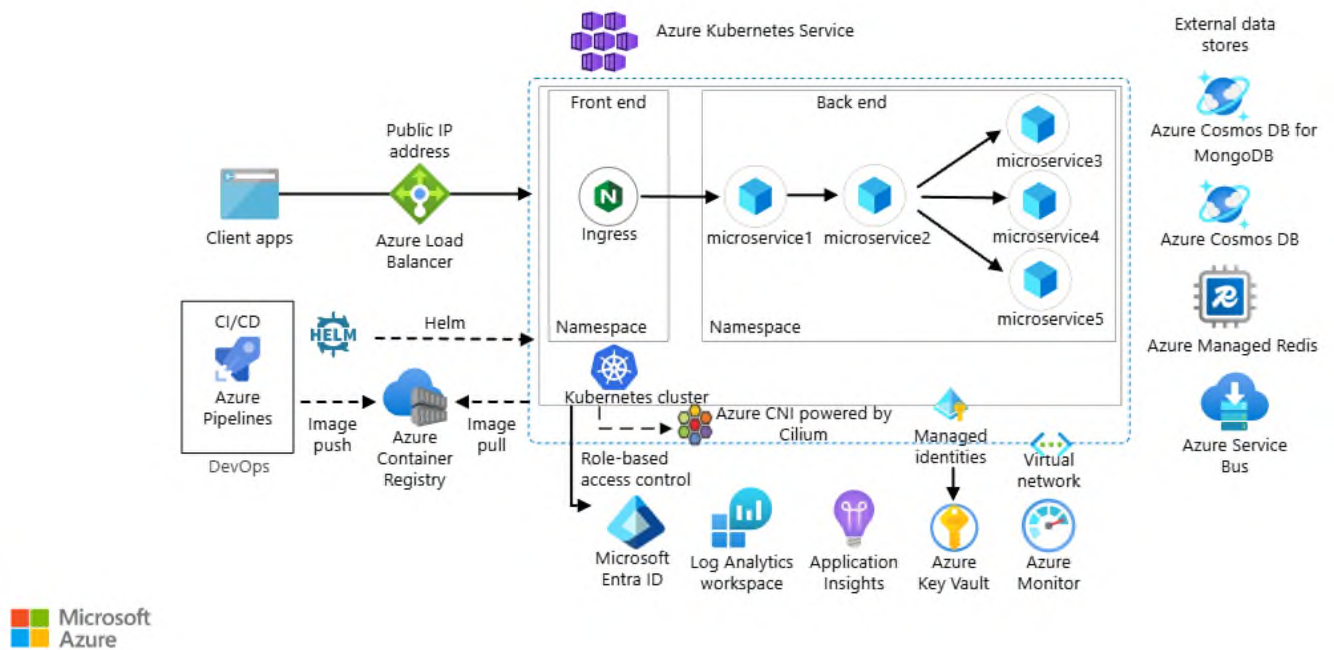

Рисунок 1.2 – Приклад архітектури хмарної інфраструктури в Microsoft Azure з використанням AKS.

Таким чином, Microsoft Azure надає всі необхідні засоби для побудови хмарної інфраструктури, реалізації інфраструктури як коду, організації CI/CD-конвеєрів і інтегрованого моніторингу. Саме це робить Azure доцільною базовою платформою для практичної реалізації рішення, що розробляється в даній магістерській роботі.

1.3 Kubernetes та Azure Kubernetes Service як платформа розгортання

Kubernetes є де-факто стандартом оркестрації контейнеризованих застосунків. Його основна задача – автоматизувати розгортання, масштабування та керування контейнерами в кластері з декількох вузлів. Архітектурно Kubernetes побудований за принципом розподіленої системи з розділенням на **площину керування (control plane)** та **робочі вузли (worker nodes)**. Площина керування відповідає за прийняття рішень щодо стану кластера (планування подів, підтримання декларативно заданого стану, реагування на події), тоді як робочі вузли безпосередньо виконують контейнери (рис 1.3).

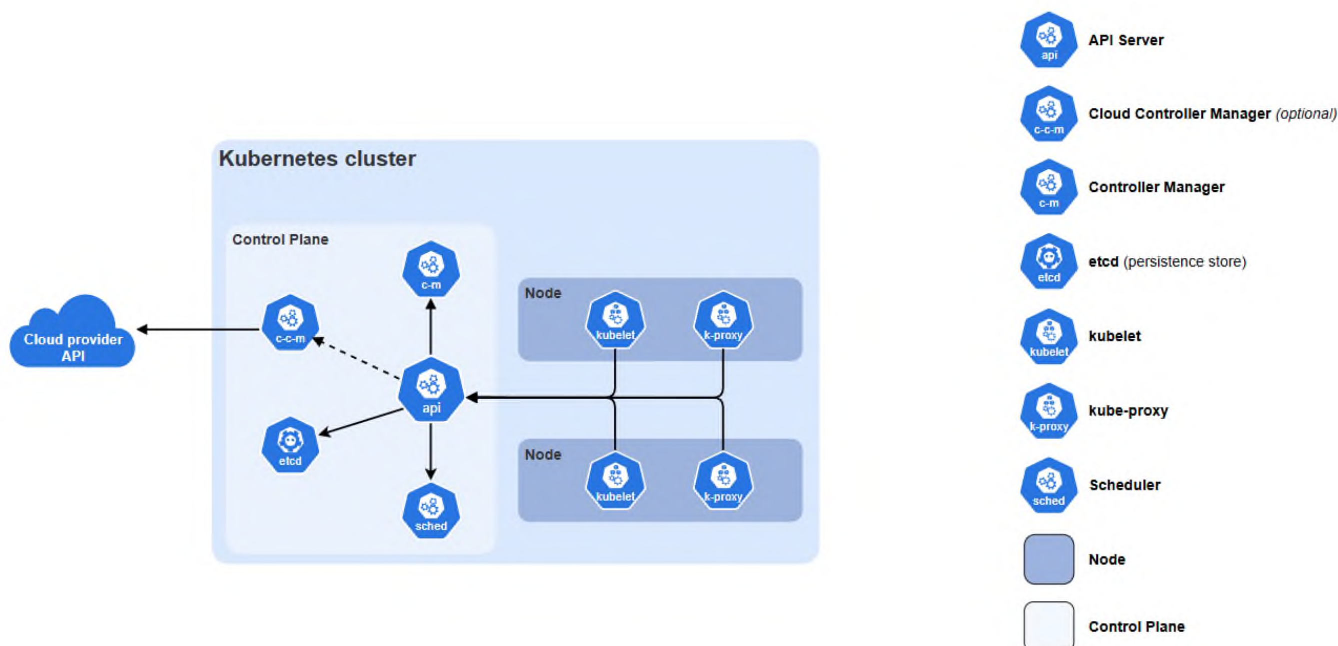

Рисунок 1.3 – Базова архітектура кластера Kubernetes.

До складу control plane входять такі ключові компоненти: API-server, через який усі клієнти взаємодіють із кластером; сховище стану etcd, у якому зберігається поточна конфігурація; контролери, що відслідковують відхилення від бажаного стану та ініціюють коригувальні дії; а також планувальник (scheduler), який визначає, на яких вузлах мають бути запущені нові поди. Робочі вузли містять kubelet, що відповідає за запуск контейнерів відповідно до інструкцій control plane, та компонент мережевої взаємодії (kube-proxy або CNI-плагін), який забезпечує маршрутизацію трафіку між подами та сервісами. Така архітектура дозволяє масштабувати кластер, забезпечувати відмовостійкість і поєднувати різні типи навантажень [3-4].

Azure Kubernetes Service (AKS) є керованою реалізацією Kubernetes у хмарі Microsoft Azure. У моделі AKS площина керування повністю обслуговується провайдером: користувач не керує віртуальними машинами control plane, їх оновленням чи масштабуванням – ці задачі виконує Azure. Замовник відповідає за конфігурацію кластера на рівні параметрів (версія Kubernetes, типи й кількість вузлів, мережеві налаштування, інтеграція з іншими сервісами) та за керування робочими вузлами (agent node pools). Це спрощує експлуатацію, дозволяє зосередитись на застосунках і зменшує операційні ризики.

AKS тісно інтегрований з іншими сервісами Azure. Для зберігання та доставки контейнерних образів використовується Azure Container Registry (ACR), для керування доступом – Azure Active Directory та механізми RBAC, для мережевої взаємодії – Azure Virtual Network та пов'язані ресурси (subnet, load balancer, Application Gateway), для моніторингу – Azure Monitor та Log Analytics. Підтримуються кілька плагінів мережі (Azure CNI, kubenet), різні типи пулів вузлів (Linux/Windows), autoscaling, використання керованих ідентичностей та інтеграція з GitOps-інструментами.

Використання AKS як платформи розгортання в контексті даної роботи обумовлене двома основними факторами. По-перше, AKS поєднує гнучкість «чистого» Kubernetes з перевагами керованого сервісу, знижуючи операційні витрати. По-друге, він органічно вписується в екосистему Azure, що спрощує побудову комплексного рішення на основі IaC та CI/CD із використанням Terraform, Azure DevOps та відкритого стеку моніторингу [7-8].

1.4 Огляд наявних інструментів IaC та CI/CD для хмарних рішень

Побудова відтворюваної хмарної інфраструктури та конвеєрів розгортання неможлива без використання спеціалізованих інструментів. Для реалізації підходу «Інфраструктура як код» існує кілька основних класів рішень. До нативних засобів Azure належать **ARM Templates** та мова **Bicep**, які дозволяють декларативно описувати ресурси платформи та виконувати їх розгортання через Azure Resource Manager. Перевагою цих інструментів є тісна інтеграція з екосистемою Azure, проте вони орієнтовані переважно на один хмарний провайдер [5-6,18].

Крос-платформні інструменти, такі як **Terraform** та **Pulumi**, підтримують різних провайдерів і дозволяють уніфікувати підхід до опису інфраструктури в мультихмарному середовищі. Terraform використовує декларативну мову HCL та модель графа залежностей між ресурсами, що забезпечує ідемпотентне застосування змін і можливість попереднього планування (plan). Pulumi пропонує опис інфраструктури за допомогою загальнопризначених мов програмування. Окрему нішу займають конфігураційні менеджери на кшталт Ansible, Chef, Puppet, які часто

застосовують у поєднанні з Terraform для налаштування програмного оточення поверх уже створених ресурсів.

У сфері CI/CD існує широкий спектр рішень – як хмарних, так і on-premises. До найпоширеніших належать **Azure DevOps Pipelines**, **GitHub Actions**, **GitLab CI/CD**, **Jenkins**, **CircleCI** та інші. Вони забезпечують автоматизацію збірки, тестування та розгортання застосунків, інтегруються з системами контролю версій, дозволяють будувати багатостадійні конвеєри з умовною логікою, паралельними етапами та перевірками якості. Вибір конкретного інструмента залежить від вимог проєкту, існуючої екосистеми й обмежень безпеки.

В екосистемі Azure ключову роль відіграють **Azure DevOps** та **GitHub Actions**, які мають глибоку інтеграцію з сервісами платформи, підтримують керування секретами, роботу з агентами в хмарі та on-premises, а також дозволяють будувати конвеєри для IaC (наприклад, автоматичне виконання Terraform plan/apply при зміні конфігурації). У поєднанні з Helm та GitOps-підходом (Flux, Argo CD) ці інструменти забезпечують повний життєвий цикл розгортання контейнеризованих сервісів у Kubernetes-кластерах.

У рамках цієї магістерської роботи як основні інструменти обрано **Terraform** для опису інфраструктури в Azure та **Azure DevOps Pipelines** для реалізації CI/CD-конвеєрів. Такий вибір дозволяє поєднати крос-платформний підхід до IaC з тісною інтеграцією в Azure-екосистему, а також створити шаблон, який у майбутньому може бути адаптований до інших хмарних провайдерів або CI/CD-платформ.

Висновки до розділу

У першому розділі розглянуто теоретичні та технологічні передумови побудови хмарної інфраструктури на базі підходів DevOps та «Інфраструктура як код». Показано, що традиційні ручні методи керування інфраструктурою не відповідають вимогам до швидкості та надійності сучасних програмних систем, тоді як DevOps-орієнтовані практики та IaC забезпечують відтворюваність середовищ, контрольованість змін і можливість вимірювати ефективність процесів.

Окремо проаналізовано платформу Microsoft Azure як базу для побудови хмарних сервісів, її ключові концепції (ресурси, групи ресурсів, віртуальні мережі, керовані сервіси) та засоби підтримки DevOps-підходу. Показано, що керований сервіс Azure Kubernetes Service поєднує гнучкість Kubernetes із перевагами повністю керованої площини керування, а також тісно інтегрований з іншими сервісами Azure.

Наведено огляд основних інструментів для реалізації IaC та CI/CD у хмарному середовищі й обґрунтовано доцільність вибору Terraform та Azure DevOps Pipelines як базових засобів у цій роботі. Таким чином, у розділі сформовано концептуальну та технологічну основу, на якій ґрунтуватиметься подальший аналіз вимог, моделювання та практична реалізація комплексного рішення IaC + CI/CD для хмарних сервісів на базі Azure та Kubernetes.

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ

2.1 Опис предметної області та вимоги до хмарної інфраструктури

Предметна область даної роботи охоплює процес побудови та експлуатації хмарної інфраструктури для контейнеризованих сервісів із використанням підходів IaC та CI/CD. Типовим сценарієм є продуктова або аутсорс-команда, яка розробляє декілька мікросервісів, що мають бути розгорнуті в хмарі Microsoft Azure у кількох середовищах (наприклад, development, staging, production). Для таких команд характерні часті зміни коду, необхідність швидко піднімати нові середовища для тестування, а також вимоги до передбачуваності та безпеки інфраструктури.

У традиційних підходах адміністратори створюють ресурси вручну через веб-портал або окремі скрипти, що неминуче призводить до «дрейфу конфігурацій», коли середовища починають відрізнятися, а відтворення точної копії інфраструктури стає складним або неможливим. У такій ситуації важко гарантувати коректність релізів, швидко відновлюватися після інцидентів і проводити експерименти з новими конфігураціями без ризику порушити роботу продуктивної системи.

У межах цієї роботи розглядається підхід, за якого всі основні компоненти інфраструктури – мережа, кластер Kubernetes (AKS), реєстр контейнерів (ACR), засоби зберігання секретів, системи моніторингу та логування – описуються декларативно, зберігаються в системі контролю версій і розгортаються автоматично. Такий підхід дозволяє розглядати інфраструктуру як невід’ємну частину програмного продукту, а не як зовнішнє «середовище», про яке згадують лише на етапі релізу.

На основі аналізу предметної області можна сформулювати ключові **функціональні вимоги** до хмарної інфраструктури:

- підтримка запуску контейнеризованих сервісів у кластері Kubernetes з можливістю масштабування;
- наявність окремих середовищ (мінімум: dev, stage, prod), які можна розгортати й оновлювати незалежно;
- інтеграція з реєстром контейнерних образів для зберігання артефактів збірки;

- підтримка механізму безпечного зберігання секретів (паролі, ключі, рядки підключення);
- наявність централізованого моніторингу, логування та алертингу для сервісів та інфраструктури;
- можливість автоматизованого розгортання, оновлення та видалення ресурсів на основі опису у вигляді коду.

Окрім цього, до інфраструктури висуваються **нефункціональні вимоги**, пов'язані з експлуатацією та якістю сервісу:

- **відтворюваність** – можливість розгорнути нове середовище, яке буде еквівалентним наявному (за винятком параметрів, специфічних для середовища, наприклад, розмірів кластерів);
- **масштабованість** – підтримка горизонтального масштабування сервісів і вертикального/горизонтального масштабування вузлів кластера;
- **надійність та відмовостійкість** – здатність інфраструктури витримувати відмови окремих вузлів та компонентів без тривалих простоїв;
- **безпека** – контрольований доступ до ресурсів, сегментація мережі, шифрування даних у стані спокою й під час передавання, мінімізація відкритих публічних інтерфейсів;
- **спостережуваність** – можливість отримувати метрики, логи та трасування, достатні для діагностики проблем і оцінювання DevOps-метрик;
- **керованість змін** – усі зміни мають проходити через систему контролю версій, код-рев'ю та автоматизовані перевірки.

Також важливо врахувати **організаційні вимоги**: інфраструктура має бути зрозумілою не лише для її первинного автора, але й для інших членів команди. Це означає необхідність дотримання єдиних підходів до структури репозиторію, іменування ресурсів, організації Terraform-модулів та CI/CD-конвеєрів.

У сукупності ці вимоги визначають рамки, в яких має бути спроєктована та реалізована інфраструктура на базі Microsoft Azure та AKS, а також формують критерії, за якими можна оцінювати успішність розробленого рішення.

2.2 Архітектура цільового рішення в Microsoft Azure

Відповідно до визначених вимог у попередньому підрозділі, цільове рішення будується як набір взаємопов'язаних сервісів Microsoft Azure, що утворюють єдину хмарну платформу для розгортання контейнеризованих застосунків. Базовим елементом є **Resource Group**, до якої входять усі ресурси конкретного середовища (наприклад, rg-dev-aks, rg-stg-aks, rg-prod-aks). Це дозволяє логічно ізолювати середовища, спрощує операції з ними (резервне копіювання, видалення, міграція) та полегшує облік витрат.

У межах кожної групи ресурсів створюється **Azure Virtual Network (VNet)** з однією або кількома підмережами (subnet), у яких розміщуються вузли кластера AKS та пов'язані сервіси. Використання власної VNet дозволяє застосовувати мережеві політики, network security groups, інтегрувати кластер із внутрішніми ресурсами (наприклад, базами даних або сервісами в інших VNet), а також обмежувати доступ із публічного Інтернету.

Ключовим компонентом є **Azure Kubernetes Service (AKS)**, який розгортається в зазначеній віртуальній мережі та використовує один або кілька пулів вузлів (node pools) для обробки навантаження. Параметри кластера (версія Kubernetes, розмір вузлів, кількість екземплярів, тип дисків тощо) задаються в коді Terraform, що дозволяє задавати відмінності між середовищами на рівні змінних, не змінюючи структури самого рішення.

Для зберігання контейнерних образів використовується **Azure Container Registry (ACR)**, який інтегрується з AKS таким чином, щоб вузли кластера могли автентифікуватися до реєстру за допомогою керованих ідентичностей або відповідних секретів. ACR виступає центральним сховищем артефактів збірки, які створюються на етапі CI й використовуються на етапі CD.

Конфіденційні дані – ключі, токени, рядки підключення до баз даних – зберігаються в **Azure Key Vault**. Доступ до Key Vault організовується через керовані ідентичності (managed identities) та політики доступу, що дозволяє мінімізувати використання секретів у явному вигляді в конфігураціях та змінних середовища. Усі

посилання на секрети інтегруються в IaC та CI/CD-конфігурації таким чином, щоб застосунок отримував необхідні значення вже під час розгортання.

Для моніторингу й логування використовуються **Azure Monitor** разом із **Log Analytics Workspace** або відкритий стек на базі Prometheus, Grafana та Loki, розгорнутий у кластері AKS. У першому випадку компоненти кластера й застосунки відправляють метрики та логи у відповідні служби Azure; у другому – в кластері створюються окремі неймспейси й ресурси, які реалізують функції збору та візуалізації телеметрії. Обидва підходи можуть співіснувати: Azure Monitor використовується для базового спостереження за інфраструктурою, а Prometheus/Grafana/Loki – для гнучкого моніторингу прикладних сервісів і DevOps-метрик.

На рівні процесів розробки та розгортання до архітектури додається сервіс **Azure DevOps**, у якому розміщуються репозиторії коду (IaC, застосунки, Helm-чарти), описуються CI/CD-конвеєри та конфігуруються артефактні сховища. Завдяки цьому формується повноцінна екосистема: розробник робить зміну в репозиторії, запуск CI/CD відбувається автоматично, результати збірки потрапляють до ACR, а нові версії сервісів – до кластера AKS [7,17].

Логічну структуру цільової архітектури можна представити у вигляді кількох шарів:

- **інфраструктурний шар** – Resource Groups, VNet, AKS, ACR, Key Vault, сховища, моніторинг;
- **прикладний шар** – контейнеризовані сервіси, розгорнуті в AKS з використанням Helm;
- **процесний шар** – CI/CD-конвеєри в Azure DevOps, які керують життєвим циклом інфраструктури та застосунків;
- **шар спостережуваності** – системи збору метрик, логів та алертингу.

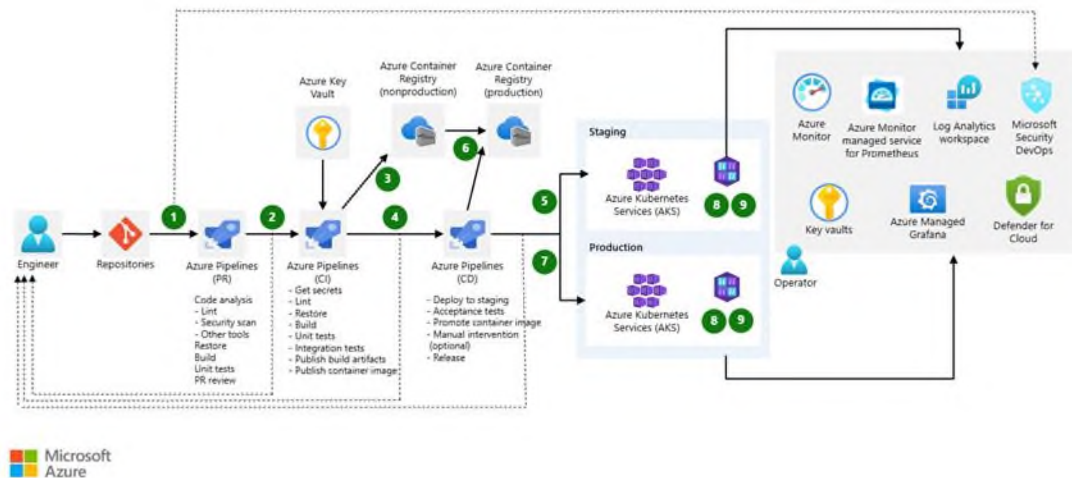

Рисунок 2.1 – Цільова архітектура рішення в Microsoft Azure

2.3 Інформаційні потоки та артефакти (стан інфраструктури, конфігурації, журнали, метрики)

Цільова архітектура визначає набір основних **артефактів**, із якими працює система, та **інформаційні потоки** між ними. Їх чітке розуміння є критичним для побудови коректних конвеєрів CI/CD, організації спостережуваності й забезпечення відтворюваності інфраструктури.

До ключових артефактів інфраструктурного рівня належать:

- **IaC-конфігурації** – файлові структури Terraform (модулі, змінні, файли оточень), які описують усі ресурси в Azure. Вони зберігаються в репозиторії системи контролю версій (наприклад, Azure Repos чи GitHub) і підлягають код-рев’ю, версіонуванню та тестуванню.
- **стан інфраструктури (Terraform state)** – файл чи набір файлів, у яких інструмент IaC зберігає актуальну інформацію про розгорнуті ресурси (ідентифікатори, атрибути, залежності). У випадку Terraform для продакшен-середовищ зазвичай використовується віддалений бекенд, наприклад, контейнер у **Azure Storage Account** із блокуванням стану (state locking) через механізми Terraform.

- **конфігурації Kubernetes/Helm** – маніфести, Helm-чарти, файли values, які описують прикладні сервіси, їхні залежності, параметри ресурсів, змінні середовища тощо.
- **секрети та параметри** – дані, що зберігаються в Azure Key Vault або у вигляді Kubernetes Secret (у зашифрованому вигляді), й використовуються на етапі розгортання.

На рівні процесів CI/CD основними артефактами є:

- **вихідний код застосунків** і Dockerfile'и;
- **контейнерні образи**, що зберігаються в ACR;
- **артефакти побудови** (наприклад, зібрані Helm-чарти, файли релізних конфігурацій);
- **логи виконання конвеєрів**, які дають змогу аналізувати історію розгортань.

Інформаційні потоки можна умовно розділити на три великі групи.

1. **Потік змін конфігурацій інфраструктури.**

   Розробник IaC змінює Terraform-код у репозиторії → створюється pull request → після його затвердження конвеєр CI/CD запускає етап terraform plan, формуючи план змін щодо поточного стану, збереженого в бекенді → план перевіряється (Automated checks / manual approval) → у разі підтвердження виконується terraform apply, що оновлює як реальні ресурси Azure, так і файл стану. Таким чином, стан інфраструктури завжди відображає останню застосовану версію коду.

2. **Потік змін прикладного коду.**

   Розробник змінює код сервісу → CI-конвеєр збирає Docker-образ, запускає тести, штовхає образ у ACR → CD-конвеєр оновлює значення образу в Helm-values або в маніфестах → застосовує зміни до кластера AKS (через helm upgrade або GitOps-оператор). У результаті кожен коміт має чітко пов'язану версію образу й конфігурації, що дозволяє відтворити будь-який реліз.

3. **Потік телеметрії (логи, метрики, події).**

   Компоненти кластера та застосунки генерують метрики (CPU, пам'ять, latency, кількість запитів, статуси HTTP), журнали подій та прикладні логи. Вони

надсилаються до систем моніторингу – Azure Monitor/Log Analytics або Prometheus/Loki. На їх основі будуються дашборди в Grafana чи Azure Portal, налаштовуються алерти, а також розраховуються DevOps-метрики (Lead Time, Deployment Frequency, MTTR тощо). Телеметрія зберігається певний час, що дозволяє аналізувати динаміку роботи системи й виявляти деградації.

Важливо, що всі ці потоки взаємопов’язані: зміни в IaC-коді, прикладному коді та отримані метрики разом утворюють «замкнений цикл зворотного зв’язку». Це дозволяє не лише автоматизувати розгортання, але й обґрунтовано оцінювати вплив змін на стабільність та продуктивність системи [10-12,15].

2.4 Моделі користувачів, ролей і доступу в системі IaC та CI/CD

У системі, де інфраструктура описується як код і зміни проходять через CI/CD-конвеєри, модель доступу стає критично важливою складовою архітектури. Неправильно налаштовані дозволи можуть призвести як до випадкових помилок (наприклад, видалення продуктивних ресурсів під час експериментів у dev), так і до цілеспрямованих атак. Тому доступ до інфраструктури, репозиторіїв та конвеєрів має будуватися на принципах **мінімально необхідних привілеїв (least privilege)** та **розмежування обов’язків (separation of duties**).

У типових DevOps-командах можна виділити кілька основних категорій користувачів (персон):

- **Розробники застосунків** – працюють із прикладним кодом, Dockerfile та, частково, з Helm-конфігураціями. Вони мають повні права на свої репозиторії, можуть запускати CI-конвеєри, але зазвичай не мають прямого доступу до управління хмарною інфраструктурою (Azure Subscription).
- **Інженери з інфраструктури / DevOps** – відповідають за Terraform-конфігурації, налаштування конвеєрів CI/CD, інтеграцію з Azure та AKS. Їхні права в репозиторіях IaC ширші (створення модулів, оновлення бекенду стану), а в Azure вони мають обмежені ролі рівня subscription/resource group (наприклад, Contributor або спеціальні кастомні ролі).

- **SRE / операційні інженери** – фокусуються на експлуатації, моніторингу, реакції на інциденти. Часто мають доступ до систем спостережуваності (Azure Monitor, Grafana, Loki), до читання логів і метрик у продуктивних середовищах, але не обов'язково – до зміни інфраструктури.
- **Безпекові фахівці** – можуть мати права на аудит конфігурацій, доступів, логів безпеки, але обмежені в можливості змінювати інфраструктуру й код.
- **Продакт- і технічні власники** – зазвичай мають обмежений доступ до дашбордів і звітів, можуть затверджувати зміни в критичних середовищах (manual approval у конвеєрах), але не оперують безпосередньо інструментами IaC.

У Microsoft Azure управління доступом здійснюється за допомогою **Azure Role-Based Access Control (RBAC)**, де роль (role definition) прив'язується до суб'єкта безпеки (користувач, група, service principal, managed identity) на певному рівні **scope** – від subscription до окремого ресурсу. Це дозволяє, наприклад, надати DevOps-інженеру роль Contributor лише на певну групу ресурсів, у якій розгортається AKS і пов'язані з ним сервіси, не даючи йому повного доступу до всієї підписки. Для менш критичних задач (перегляд конфігурацій, моніторинг) використовуються ролі Reader або спеціально створені кастомні ролі з обмеженим набором дій.

На рівні **Azure DevOps** моделі доступу включають ролі для організації, проєкту, репозиторіїв і конвеєрів. Розробники мають права на коміти в прикладні репозиторії та запуск конвеєрів, DevOps-інженери – на конфігурацію pipeline та сервісних з'єднань (service connections) до Azure, а безпекові фахівці – на перегляд журналів виконання та налаштувань безпеки. Для критичних етапів (наприклад, terraform apply у prod або деплой на продуктивний namespace) зазвичай додаються **manual approval** із боку окремих відповідальних осіб.

Важливою частиною моделі доступу є використання **сервісних ідентичностей** – service principal та managed identities, через які конвеєри CI/CD автентифікуються до Azure. Замість зберігання ключів доступу в pipeline-конфігураціях, їх розміщують у безпечних сховищах (Azure Key Vault, сховище секретів Azure DevOps), а доступ

надають тільки необхідним конвеєрам. Це мінімізує ризики витоку облікових даних та спрощує ротацію ключів.

Додатковий рівень контролю доступу реалізується в самому Kubernetes через **RBAC API**, де визначаються ролі (Role/ClusterRole) та їх прив’язка до суб’єктів (RoleBinding/ClusterRoleBinding). Наприклад, можна надати розробнику права тільки на читання логів і статусу подів у певному namespace, але заборонити йому виконання операцій kubectl delete чи kubectl apply поза CI/CD-процесом.

У сукупності ці механізми дозволяють побудувати **багаторівневу модель безпеки й доступу**, де кожен користувач або сервіс має рівно той набір прав, який необхідний для виконання його задач, і всі критичні дії проходять через контрольовані конвеєри та системи логування.

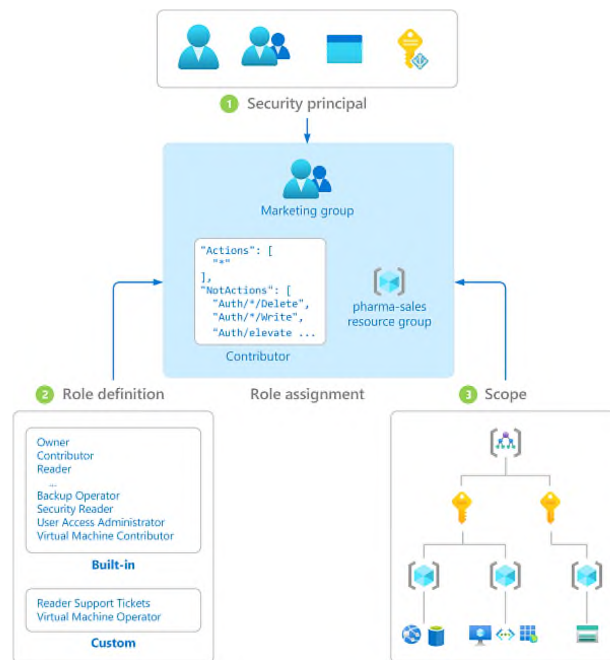

Рисунок 2.2 – Приклад моделі рольового доступу в Azure

2.5. Вимоги до надійності, безпеки, масштабованості та спостережуваності

Проєктування інфраструктури IaC + CI/CD у хмарі неможливе без явного формулювання вимог до **надійності**, **безпеки**, **масштабованості** та **спостережуваності**. Ці характеристики визначають, як система поводиться в умовах зростання навантаження, відмов компонентів, спроб несанкціонованого доступу та інших реальних сценаріїв експлуатації.

З точки зору **надійності** важливо забезпечити відмовостійкість як на рівні інфраструктури Azure, так і на рівні Kubernetes. Для інфраструктури це включає використання **Availability Zones** та розміщення ресурсів у кількох зонах однієї регіональної локації, що зменшує ймовірність одночасної відмови всіх компонентів. Для AKS це означає розгортання кількох вузлів у різних зонах, налаштування autoscaling, використання liveness/readiness-проб для автоматичного перезапуску контейнерів. Додатково слід передбачити механізми резервного копіювання критичних даних і можливість відтворення інфраструктури «з нуля» на основі IaC-коду в разі серйозних інцидентів [15-16].

Безпека включає кілька вимірів: керування ідентичностями й доступом (Azure AD, RBAC, managed identities), захист мережевих периметрів (Network Security Groups, приватні ендпоінти, обмеження публічних IP), шифрування даних у стані спокою та під час передавання, а також безпеку ланцюжка постачання (supply chain security) – перевірка образів, контроль джерел залежностей, підпис артефактів. У контексті IaC і CI/CD особливо важливо, щоб **усі зміни до інфраструктури та конвеєрів були відслідковуваними**: мали автора, дату, коментар і проходили через код-рев'ю. Це дозволяє визначити джерело потенційної вразливості та відкотити небезпечні зміни.

Масштабованість охоплює здатність системи ефективно обробляти зростання навантаження. У Kubernetes це досягається через **горизонтальне масштабування подів** (Horizontal Pod Autoscaler) та **масштабування вузлів** (Cluster Autoscaler). В Azure масштабованість також залежить від правильного вибору SKU вузлів, типів дисків, обмежень ACR, пропускної здатності мережі тощо. З погляду IaC важливо, щоб модульна структура Terraform дозволяла легко додавати нові сервіси, кластери, середовища без повного перепроєктування. Конвеєри CI/CD мають масштабуватися разом із кількістю сервісів, наприклад, за рахунок шаблонів pipeline та повторно використовуваних задач.

Спостережуваність (observability) – це здатність системи надавати достатньо інформації, щоб можна було відповісти на питання «що відбувається всередині» без прямого доступу до кожного компонента. Для хмарної інфраструктури це означає збір

і аналіз **метрик** (ресурсів, продуктивності, бізнес-показників), **логів** (системних і прикладних), **трасувань** (distributed tracing) та подій. Типовим підходом є використання комбінації Azure Monitor/Log Analytics та відкритого стеку Prometheus – Grafana – Loki, які в сукупності забезпечують єдину точку спостереження за станом кластера й сервісів.

Вимоги до спостережуваності включають:

- наявність дашбордів для ключових сервісів та інфраструктурних компонентів;
- налаштування алертів за основними SLO (latency, error rate, доступність);
- зберігання історії метрик і логів достатньої тривалості для аналізу тенденцій;
- можливість кореляції подій: зв’язок між релізами, змінами в IaC, піками навантаження та інцидентами.

Усі ці аспекти мають бути враховані вже на етапі проєктування IaC і CI/CD, а не додаватися «постфактум». Тільки тоді система зможе відповідати вимогам реальної експлуатації, а не лише демонстраційного сценарію.

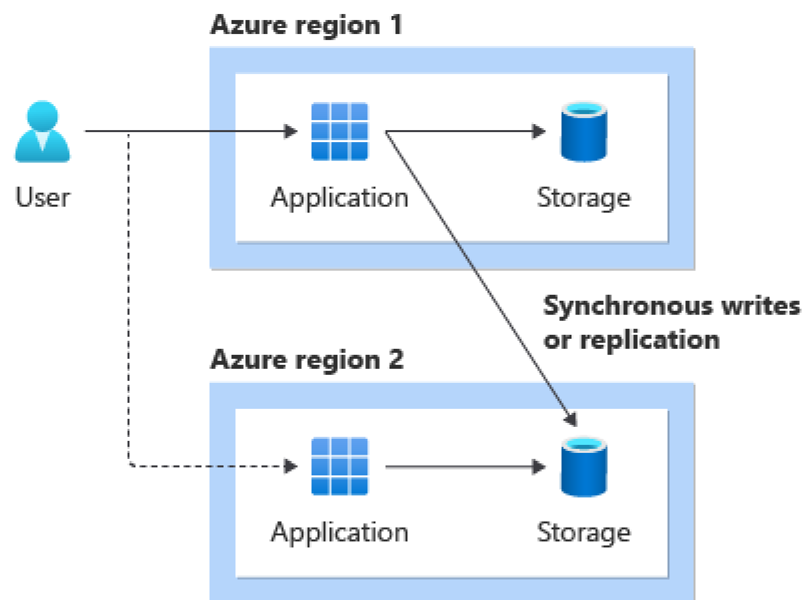

Рисунок 2.3 – Високорівнева схема регіону Azure та зон доступності

Висновки до розділу

У другому розділі було сформульовано вимоги до хмарної інфраструктури та процесів IaC/CI/CD, а також побудовано цільну модель інформаційних потоків і доступу. На основі аналізу предметної області визначено функціональні, нефункціональні та організаційні вимоги, що задають рамки для проєктування рішення в Microsoft Azure та AKS.

Описана цільова архітектура охоплює ключові сервіси Azure – віртуальні мережі, керований кластер Kubernetes, реєстр контейнерів, сховище секретів, системи моніторингу та платформу Azure DevOps. Показано, як ці компоненти взаємодіють між собою через IaC-опис, конвеєри CI/CD та потоки телеметрії, формуючи єдину платформу для розгортання та експлуатації контейнеризованих застосунків.

Особливу увагу приділено моделям користувачів і ролей, а також вимогам до надійності, безпеки, масштабованості та спостережуваності. Сформована модель доступу й набір експлуатаційних вимог визначають, якими саме мають бути Terraform-конфігурації, конвеєри Azure DevOps, політики безпеки та стек моніторингу в практичній частині роботи. Таким чином, розділ 2 створює основу для подальшого математичного й модельного опису життєвого циклу інфраструктури та для реалізації прототипу рішення, що буде розглянуто в наступних розділах.

РОЗДІЛ 3. МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Формальна модель життєвого циклу декларативної інфраструктури

Життєвий цикл декларативної інфраструктури відрізняється від традиційного підходу тим, що **джерелом істини** виступає не поточний стан хмарних ресурсів, а конфігурація у вигляді коду. Інфраструктура розглядається як система, що переходить між станами під дією змін у конфігурації та операцій розгортання.

Позначимо через

- C — множину усіх можливих конфігурацій інфраструктури (Terraform-код, змінні, модулі),
- S — множину можливих станів реальної інфраструктури в Azure,
- f : C → S — відображення, яке описує результат застосування конфігурації до хмари (операція terraform apply та пов'язані дії).

Тоді **цільовий стан** інфраструктури для конфігурації c ∈ C позначимо як s*=f(c). Реальний поточний стан системи в хмарі позначимо s ∈ S. Завдання IaC-підходу можна формалізувати як прагнення забезпечити умову

$$s \approx s^* \qquad (3.1)$$

тобто реальний стан максимально наближений до цільового (відмінності допускаються лише в межах незначних технічних деталей, які не впливають на роботу системи). В ідеальному випадку виконується s=s*.

Життєвий цикл можна подати у вигляді послідовності етапів:

1. **Редагування конфігурації**: перехід від конфігурації c_t до c_{t+1} через зміну коду в репозиторії (коміт).
2. **Аналіз змін**: обчислення дельти між $f(c_t)$ та $f(c_{t+1})$ за допомогою операції plan.
3. **Застосування змін**: виконання операції apply, яка змінює реальний стан інфраструктури із s_t у s_{t+1} так, щоб $s_{t+1} \approx f(c_{t+1})$.
4. **Експлуатація та моніторинг**: збір телеметрії, виявлення відхилень і деградацій.

5. **Виявлення дрейфу конфігурації**: якщо реальний стан sss змінюється не через IaC (ручні зміни, зовнішні фактори), виникає дрейф, який можна формально подати як $s \neq f(c)$. Завдання оператора — повернути систему до контрольованого стану шляхом оновлення коду або повторного застосування конфігурації.

Таким чином, життєвий цикл декларативної інфраструктури можна інтерпретувати як **цикл замкненого керування** із джерелом істини у вигляді коду. На кожному кроці змінюється або конфігурація ccc, або стан sss, а комплекс IaC-інструментів і CI/CD-конвеєрів відповідає за те, щоб тримати ці два об’єкти максимально узгодженими.

У практичних реалізаціях (Terraform + Azure) життєвий цикл доповнюється поняттями **бекенду стану**, блокування (state locking), середовищ (workspaces) та перевірок якості коду (linting, policy as code). Однак навіть з урахуванням цих деталей базова модель залишається незмінною: конфігурація → план → застосування → контроль стану.

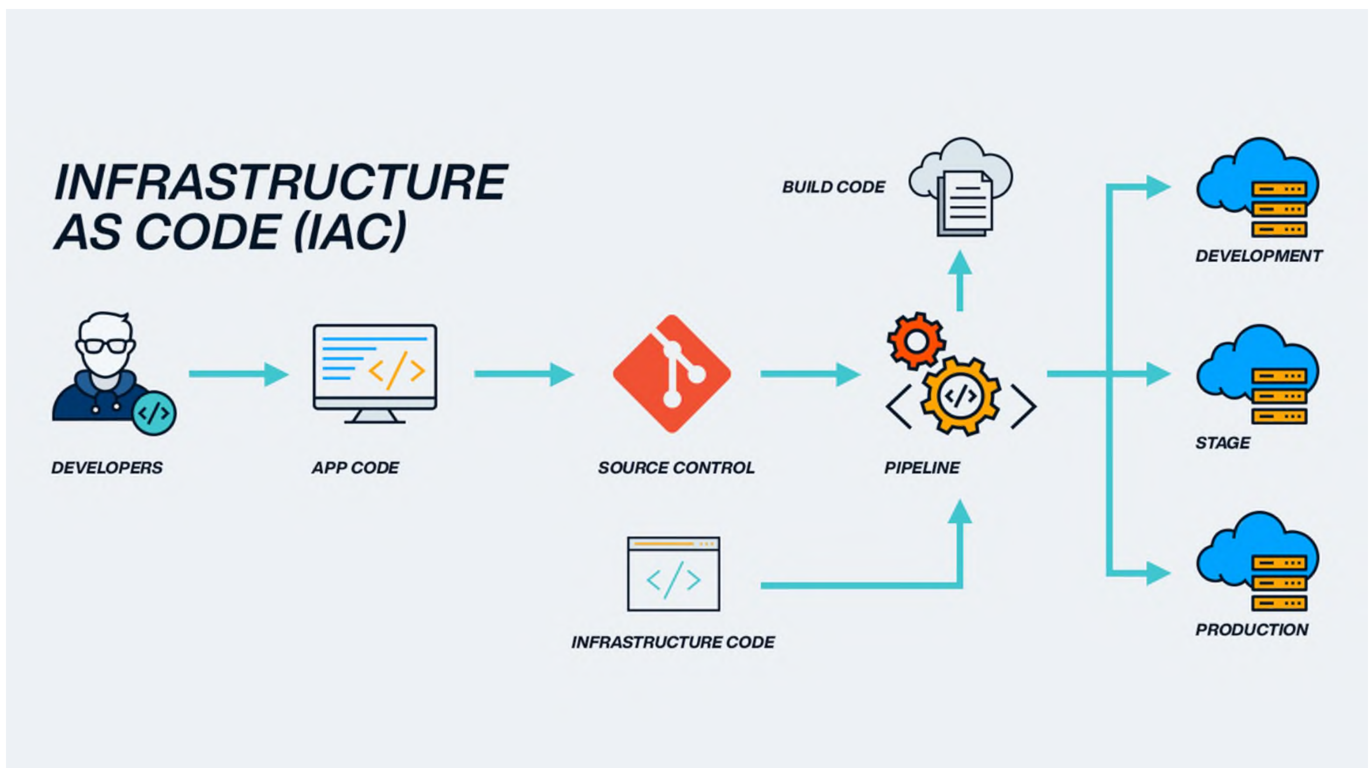

Рисунок 3.1 – Узагальнена схема життєвого циклу декларативної інфраструктури.

3.2 Модель графа залежностей ресурсів та ідемпотентність операцій

Однією з ключових особливостей інструментів IaC, зокрема Terraform, є представлення інфраструктури у вигляді **графа залежностей ресурсів**. Це дозволяє автоматично визначати порядок створення, оновлення та видалення об’єктів, виконуючи операції паралельно там, де це можливо, та послідовно — там, де це необхідно.

Формально ресурсну модель можна представити як орієнтований ациклічний граф

$$G=(V,E), \quad (3.2)$$

де

- V — множина вершин, кожна вершина відповідає окремому ресурсу (наприклад, ресурсна група, віртуальна мережа, кластер AKS, ACR, Key Vault);
- $E \subseteq V \times V$ — множина орієнтованих ребер, де ребро (v_i, v_j) означає, що ресурс v_j залежить від ресурсу v_i (наприклад, AKS залежить від VNet, ACR залежить від ресурсної групи).

Вимога ациклічності (відсутність циклів) гарантує, що існує **топологічний порядок** на V, у якому можна послідовно застосовувати операції створення або оновлення ресурсів без конфліктів. Інструмент IaC будує граф залежностей на основі посилань у конфігурації (references), явних директив залежностей та властивостей ресурсів, і потім виконує операції, дотримуючись цього порядку.

Операцію застосування змін до інфраструктури можна розглядати як функцію

$$\text{apply} : (G,s) \rightarrow s', \quad (3.3)$$

де s — поточний стан, а s′ — новий стан після виконання плану. Важливою властивістю цієї операції є **ідемпотентність**: якщо конфігурація не змінюється, повторне застосування не повинно призводити до додаткових модифікацій стану. Формально це можна записати як

$$\text{apply} : (G,s^*) \rightarrow s^*, \quad (3.4)$$

тобто для стану, що вже відповідає графу конфігурації, операція є нейтральною. Ідемпотентність дозволяє безпечно запускати конвеєри застосування змін

багаторазово (наприклад, у нічних перевірках, при відновленні після збоїв CI/CD тощо), не ризикуючи отримати різні конфігурації від одного й того ж коду.

При оновленні конфігурації з G_t до G_{t+1} інструмент IaC визначає **множину змінних вершин** ΔV⊆V та пов'язаних із ними ребер, після чого формує послідовність операцій створення, оновлення або видалення ресурсів. Цю процедуру можна розглядати як побудову нового плану

$$P_t: (G_t, s_t) \Rightarrow (G_{t+1}, s_{t+1}), \tag{3.5}$$

де P_t містить впорядковану множину операцій над ресурсами. У Terraform це відповідає результату команди terraform plan, який вказує, які ресурси будуть створені (+), змінені (~) або видалені (-).

У контексті хмарних рішень на базі Azure, граф залежностей включає як **логічні зв'язки** (наприклад, AKS використовує певну VNet), так і **обмеження платформи** (певні типи ресурсів можуть створюватися тільки після інших). Графова модель спрощує аналіз наслідків змін: можна заздалегідь оцінити, які ресурси будуть перезібрані, і чи не зачепить це критичні компоненти продуктивного середовища.

Таким чином, інфраструктура у вигляді графа залежностей забезпечує формальний фундамент для коректної послідовності операцій, а ідемпотентність операцій застосування дозволяє перетворити розгортання інфраструктури на стабільний і передбачуваний процес, інтегрований у CI/CD-конвеєри.

3.3 Моделювання CI/CD-конвеєра як послідовності етапів та станів

У CI/CD-конвеєр можна розглядати як кінцевий автомат, що переводить систему (код, артефакти, інфраструктуру) з одного стану в інший через послідовність етапів (stages). Кожен етап має вхідні артефакти, набір операцій та можливі результати (успіх, помилка, частковий успіх із ручним втручанням).

Позначимо через

- X — множину можливих станів конвеєра (наприклад: Idle, Building, Testing, Publishing, Deploying, Verifying, Failed, Succeeded),
- A — множину дій або подій (коміт, запуск пайплайну, успішне завершення етапу, помилка, manual approval),

- $\delta: X \times A \to X$ — функцію переходів, яка описує, у який стан переходить конвеєр при виконанні дії.

Спрощений сценарій CI/CD для контейнерного застосунку в AKS можна описати так:

1. **Подія commit**: перехід із Idle у стан Building.

$$\delta(Idle, commit) = Building \quad (3.6)$$

2. **Етап Build**: збірка Docker-образу; у разі успіху – перехід у Testing, у разі помилки – у Failed.

$$\delta(Building, build_ok) = Testing,\ \delta(Building, build_failed) = Failed \quad (3.7)$$

3. **Етап Test**: запуск модульних/інтеграційних тестів; у разі успіху – перехід у Publishing, у разі помилки – у Failed.
4. **Етап Publishing**: завантаження образу до ACR, публікація артефактів; у разі успіху – перехід у Deploying.
5. **Етап Deploy**: застосування Helm-чартів до AKS (або оновлення Git-репозиторію в GitOps-сценарії); можливий додатковий стан AwaitingApproval між Publishing та Deploying для продуктивного середовища.
6. **Етап Verify**: перевірка стану застосунку (health-check, smoke-тести, перевірка метрик); у разі успіху – перехід у Succeeded, у разі помилки – у Failed або ініціювання rollback.

Таким чином, конвеєр задає **формальну послідовність кроків**, що повинні бути виконані, аби зміна в коді перейшла в продуктивний стан. У термінах DevOps це відповідає скороченню **Lead Time for Changes** — часу від коміту до успішного розгортання.

Якщо розширити цю модель, можна включити в неї й **інфраструктурний конвеєр**:

- зміна IaC-конфігурації → етап Plan (обчислення плану змін) → етап Review (перегляд і затвердження плану) → етап Apply (застосування змін до Azure) → етап Post-Checks (верифікація стану, базові тести).

У цьому випадку CI/CD-система містить два тісно пов’язані автомати: один керує **прикладним кодом**, інший — **інфраструктурним кодом**. Взаємодія між ними

описується додатковими подіями: наприклад, успішне оновлення інфраструктури може бути передумовою для запуску деплою нової версії сервісу.

Окрім дискретних станів, CI/CD-конвеєр можна характеризувати **кількісними показниками**: час виконання кожного етапу, частота помилок, відсоток успішних розгортань, середній час відновлення після невдалого релізу. Ці величини використовуються для розрахунку DevOps-метрик і подальшого вдосконалення процесів (оптимізація етапів, розподіл завдань між агентами, паралелізація тестів тощо).

Формальне моделювання конвеєра як автомата зі станами дозволяє:

- чітко визначити точки, де потрібне ручне втручання (approval, audits);
- описати сценарії обробки помилок (retry, rollback, зупинка конвеєра);
- автоматизувати збір статистики й побудову SLA/SLO для процесу розгортання.

3.4. DevOps-метрики та методика їх оцінювання в хмарних середовищах

Однією з ключових відмінностей DevOps-підходу від класичних підходів до розробки є орієнтація на **кількісні показники** процесів. Це дозволяє не лише суб'єктивно оцінювати «швидко/повільно» або «стабільно/нестабільно», а й чітко вимірювати ефективність змін, відстежувати прогрес та приймати рішення на основі даних. У контексті даної роботи основну увагу буде приділено **DORA-метрикам** та показникам, пов'язаним із експлуатацією сервісів у AKS [2,14].

До базових DevOps-метрик, що застосовуються в роботі, належать:

- **Lead Time for Changes** – час від моменту коміту (або створення pull request) до розгортання зміни в продуктивному середовищі. Характеризує швидкість проходження змін через CI/CD-конвеєр.
- **Deployment Frequency** – частота розгортань у продуктивне середовище (наприклад, кількість релізів за день/тиждень). Відображає, наскільки команда здатна робити малі й часті зміни.

- **Mean Time To Restore (MTTR)** – середній час від моменту виявлення інциденту до повного відновлення працездатності системи. Характеризує здатність швидко реагувати на відмови.
- **Change Failure Rate** – частка розгортань, що приводять до інцидентів (відкат, гарячий фікс, деградація сервісу). Відображає стабільність релізного процесу.

Формально, для заданого періоду спостереження T (наприклад, 30 днів) метрики можуть обчислюватися так:

- Нехай N_{deploy} – кількість розгортань у prod за період T, тоді

$$\text{Deployment Frequency} = \frac{N_{deploy}}{T} \quad (3.8)$$

- Для кожної зміни і нехай $t_{commit}^{(i)}$ – час коміту, а $t_{prod}^{(i)}$ – час успішного розгортання в prod, тоді

$$\text{Lead Time} = \frac{1}{N}\sum_{i=1}^{N}(t_{prod}^{(i)} - t_{commit}^{(i)}) \quad (3.9)$$

- Для MTTR нехай $t_{incident}^{(i)}$ – час реєстрації інциденту j, а $t_{restore}^{(i)}$ – час повного відновлення сервісу, тоді

$$\text{MTTR} = \frac{1}{M}\sum_{j=1}^{M}(t_{restore}^{(j)} - t_{incident}^{(j)}) \quad (3.10)$$

- Для Change Failure Rate нехай N_{fail} – кількість розгортань, що призвели до інцидентів, тоді

$$\text{CFT} = \frac{N_{fail}}{N_{deploy}} \quad (3.11)$$

Практична реалізація збору цих метрик у системі IaC + CI/CD базується на кількох джерелах даних:

1. **Логи CI/CD-конвеєрів (Azure DevOps)** – містять інформацію про початок і завершення етапів, статуси збірки, тестів, деплою. Звідси беруться часові мітки для обчислення Lead Time та Deployment Frequency.
2. **Системи моніторингу й логування (Prometheus, Grafana, Loki, Azure Monitor)** – надають дані про інциденти, деградації, недоступність сервісів, час відновлення. Це основа для MTTR, Change Failure Rate, а також для «класичних» SRE-метрик: доступність, латентність, рівень помилок.

3. **Системи керування інцидентами (наприклад, Azure Boards, Jira)** – використовуються для формального фіксування інцидентів, часу їх відкриття й закриття, категоризації.

Методика оцінювання DevOps-метрик у межах роботи може бути описана так:

- визначається **період спостереження** (наприклад, місяць) та **набір сервісів**, для яких збираються показники;
- конфігуруються теги/лейбли в конвеєрах і моніторингу (service name, environment, version), щоб можна було зв'язати релізи з інцидентами;
- з логів Azure DevOps обчислюються часові інтервали від коміту до деплою (Lead Time) та кількість розгортань (Deployment Frequency);
- із систем моніторингу та трекінгу інцидентів виділяються періоди деградації, на основі яких обчислюються MTTR та Change Failure Rate;
- будується дашборд (наприклад, у Grafana або Azure Dashboard), який відображає ці метрики в динаміці.

Отримані показники дозволяють не лише описати поточний стан процесів, але й задати цільові орієнтири (наприклад, скорочення Lead Time удвічі, зниження Change Failure Rate до певного порогу) і відстежувати ефект від упровадження IaC, автоматизації та покращення тестування [10-12,14].

3.5 Моделювання показників надійності (SLO, SLA, error budget) для сервісів в AKS

DevOps-метрики описують насамперед **процес** доставки змін. Для характеристики **якості самого сервісу** з точки зору користувача застосовують поняття **SLI, SLO та SLA**, запозичені з підходу Site Reliability Engineering (SRE) [15-16].

- **SLI (Service Level Indicator)** – кількісний індикатор рівня сервісу. Наприклад, відсоток успішних HTTP-запитів, середня або 95-й перцентиль латентності, доступність за часом.

- **SLO (Service Level Objective)** – цільовий рівень сервісу, тобто бажане значення SLI за певний період. Наприклад, «доступність не менше 99,9 % за останні 30 днів».
- **SLA (Service Level Agreement)** – формальна угода (зазвичай юридично оформлена) між постачальником і споживачем послуг, яка визначає гарантовані рівні сервісу, санкції за їх порушення тощо.

У практиці AKS-сервісів типовим **SLI** може бути частка успішних HTTP-запитів за певний інтервал:

$$SLI_{availabillity} = \frac{N_{success}}{N_{total}} \quad (3.12)$$

де $N_{success}$ – кількість запитів із кодами 2xx/3xx, а N_{total} – загальна кількість запитів у розглядуваному вікні часу. Метрики для цього SLI збираються з ingress-контролера, API Gateway або безпосередньо з сервісів через Prometheus/NGINX/Ingress-логування.

Приклад SLO:

$$SLI_{availabillity} \geq 99{,}9\% \text{ за останні 30 днів.}$$

У такому випадку **error budget** визначається як частка часу (або частка запитів), яку система може «витратити» на відмови, не порушивши SLO. Якщо мова про доступність у часі, то:

$$\text{Error Budget} = 1 - \text{SLO} \quad (3.12)$$

Для SLO = 99,9 % маємо error budget = 0,1 %, що за 30 днів відповідає приблизно 43,2 хвилини дозволеного простою. Якщо цей бюджет вичерпано (тобто сумарний час недоступності перевищив 0,1 %), вважається, що SLO порушено.

Практичний сенс error budget у DevOps/SRE-підході полягає в **балансі між швидкістю впровадження змін і стабільністю**. Якщо бюджет помилок не використано (сервіс працює значно краще за SLO), команда може дозволити собі більш агресивні експерименти, часті релізи, великі зміни архітектури. Якщо ж бюджет вичерпано або наблизився до нуля, фокус має зміститися на стабілізацію: обмеження нових фіч-релізів, посилення тестування, виправлення технічного боргу.

У контексті AKS SLI, SLO та error budget визначаються й контролюються на основі телеметрії, що збирається з кластеру:

- для **доступності** використовуються показники успішних/неуспішних запитів на рівні ingress-контролера або сервісу (HTTP-коди, timeouts);
- для **латентності** – перцентили часу відповіді (наприклад, 95-й перцентиль не більше 300 мс);
- для **стабільності розгортань** – частка релізів, які не призвели до перевищення допустимого рівня помилок.

Визначивши SLO для ключових сервісів, можна інтегрувати їх у процес CI/CD:

- при плануванні релізів враховується поточний стан error budget;
- у pipeline додаються **quality gates**, які блокують розгортання нових версій, якщо показники SLI погіршилися або error budget вичерпано;
- дашборди в Grafana/Azure Monitor показують текуче значення SLO та залишок бюджету помилок.

Таким чином, SLO/SLA та error budget забезпечують формальний зв’язок між технічними метриками (SLI, DevOps-показники) та бізнес-вимогами до надійності сервісу, дозволяючи приймати рішення щодо частоти релізів, пріоритетів задач і допустимого рівня ризику.

Висновки до розділу

У третьому розділі побудовано формальну та модельну основу для опису процесів інфраструктури як коду та CI/CD, що використовується в подальших розділах для проєктування і реалізації рішення.

Запропоновано модель життєвого циклу декларативної інфраструктури, де конфігурація розглядається як джерело істини, а реальний стан хмарних ресурсів – як її відображення. Показано, що підтримання узгодженості між конфігурацією та станом забезпечується циклом «конфігурація → план → застосування → моніторинг», реалізованим за допомогою інструментів IaC та CI/CD.

Інфраструктура формально описана у вигляді графа залежностей ресурсів, на основі якого визначається коректна послідовність операцій створення, оновлення та видалення. Ідемпотентність операцій застосування конфігурації дозволяє перетворити процес розгортання на передбачувану й стабільну процедуру, придатну для багаторазового виконання і автоматизації.

CI/CD-конвеєр змодельовано як автомат зі станами, що описує послідовність етапів від коміту до розгортання та верифікації змін, а також взаємодію конвеєрів інфраструктурного й прикладного рівнів. На основі цієї моделі визначено DevOps-метрики (Lead Time, Deployment Frequency, MTTR, Change Failure Rate) та запропоновано методику їх оцінювання з використанням логів конвеєрів, систем моніторингу та трекерів інцидентів.

Окремо розглянуто поняття SLI, SLO, SLA та бюджету помилок, що застосовуються для формалізації вимог до якості сервісів у AKS. Показано, як ці показники пов’язуються з DevOps-метриками та як можуть бути інтегровані в процес прийняття рішень щодо частоти релізів та допустимого рівня ризику.

Таким чином, розділ 3 формує теоретико-методологічну основу, на якій у наступному розділі буде побудовано конкретне програмне рішення: модульні Terraform-конфігурації, CI/CD-конвеєри в Azure DevOps, конфігурації Kubernetes та стек спостережуваності для цільової інфраструктури в Microsoft Azure.

РОЗДІЛ 4. ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

4.1 Вибір та обґрунтування технологічного стеку (Terraform, Azure DevOps, Helm, Prometheus, Grafana, Loki)

Вибір технологічного стеку для реалізації підходів IaC та CI/CD є критичним етапом проєктування системи. Використовувані інструменти мають не лише задовольняти технічні вимоги (підтримка Azure, Kubernetes, контейнеризація), але й забезпечувати зрозумілу модель роботи для команди, надійну інтеграцію між собою та наявність зрілої екосистеми.

У ролі основного інструмента **інфраструктури як коду** обрано **Terraform**. На відміну від нативних засобів Azure (ARM Templates, Bicep), Terraform є крос-платформним рішенням, яке дозволяє описувати ресурси різних хмарних провайдерів єдиною декларативною мовою HCL [5-6]. Це робить рішення більш гнучким і незалежним від конкретного вендора, а також спрощує потенційний перехід до мультихмарної архітектури в майбутньому. Terraform має офіційний провайдер для Azure, підтримує більшість сервісів платформи, включно з AKS, ACR, Key Vault, VNet, та надає розвинуті можливості роботи з графом залежностей ресурсів і керування станом.

Як платформу для **CI/CD** обрано **Azure DevOps**, який органічно інтегрується з Azure й дозволяє будувати повністю декларативні конвеєри на основі YAML. Azure DevOps включає сервіси Repos (система контролю версій), Pipelines (конвеєри CI/CD), Artifacts (артефакти збірок) та Boards (управління роботою), що дозволяє реалізувати повний цикл розробки в рамках однієї екосистеми. Важливою перевагою є підтримка **service connections** до Azure Subscription, що дає змогу безпечно виконувати Terraform-операції, збірку контейнерних образів і деплой до AKS без ручного управління обліковими даними [8].

Для **керування розгортанням застосунків у Kubernetes** використовується **Helm** як де-факто стандартний пакетний менеджер для Kubernetes. Замість роботи з великою кількістю окремих YAML-маніфестів Helm дозволяє описати застосунок у вигляді чарту, який містить шаблони ресурсів і файл параметрів (values.yaml). Це

спрощує параметризацію конфігурацій між середовищами (dev, stage, prod), підтримує версіонування релізів (історія helm release) та дає можливість виконувати відкат до попередньої версії. У поєднанні з Azure DevOps Helm дозволяє автоматизувати оновлення застосунків у кластері AKS у рамках конвеєрів CD [9].

Для **моніторингу й спостережуваності** обрано стек **Prometheus – Grafana – Loki**. Prometheus відповідає за збір і збереження метрик (як інфраструктурних, так і прикладних), Grafana — за їх візуалізацію та побудову дашбордів, а Loki — за збір і пошук логів із контейнеризованих сервісів. Перевага цього стеку полягає у його «природній» інтеграції з Kubernetes (eksporters, service discovery, label-based model) та широкій підтримці у спільноті [10-12]. При цьому використання відкритого стеку не виключає паралельного застосування Azure Monitor/Log Analytics, але дозволяє будувати незалежні, гнучко налаштовані дашборди, орієнтовані на потреби конкретної команди.

Обраний стек забезпечує:

- **декларативне керування інфраструктурою** (Terraform);
- **декларативні CI/CD-конвеєри** з тісною інтеграцією з Azure (Azure DevOps);
- **декларативне описання застосунків для AKS** (Helm);
- **спостережуваність** із підтримкою DevOps- та SRE-практик (Prometheus, Grafana, Loki).

Таким чином, комбінація Terraform + Azure DevOps + Helm + Prometheus/Grafana/Loki дозволяє побудувати цілісний, взаємопов’язаний ланцюг від коду інфраструктури й застосунків до їх розгортання, моніторингу й аналізу якості роботи в Azure Kubernetes Service.

4.2 Проєктування та реалізація Terraform-модулів для Azure-ресурсів

Для забезпечення масштабованості та повторного використання IaC-коду інфраструктура в Azure описується у вигляді **модульної структури Terraform**. Замість одного великого набору файлів, у якому змішуються всі ресурси, рішення розділене на логічні модулі, кожен із яких відповідає за певний аспект

інфраструктури: мережу, кластер AKS, реєстр контейнерів, сховище секретів, моніторинг тощо [5-6,17].

Типова структура репозиторію Terraform у межах даної роботи може включати:

- каталог modules/, що містить окремі модулі:
  - modules/resource_group – створення ресурсної групи;
  - modules/network – створення VNet, підмереж, Network Security Groups;
  - modules/acr – конфігурація Azure Container Registry;
  - modules/aks – розгортання кластера Azure Kubernetes Service та пулів вузлів;
  - modules/keyvault – створення Azure Key Vault і базової конфігурації доступу;
  - modules/monitoring – ресурси для інтеграції з Azure Monitor / Log Analytics (за потреби);
- каталог envs/ (або live/), де для кожного середовища (dev, stage, prod) описані окремі конфігурації, які підключають модулі й передають їм параметри;
- файл налаштувань **бекенду стану** (backend.tf) для зберігання Terraform state у Azure Storage Account.

Кожен модуль містить:

- файл main.tf із описом ресурсів;
- файл variables.tf із параметрами модуля;
- файл outputs.tf з вихідними значеннями (наприклад, імена ресурсів, ідентифікатори, DNS-імена), які можуть використовувати інші модулі або CI/CD-конвеєри.

Наприклад, модуль network приймає як вхідні параметри назву ресурсної групи, CIDR-діапазон для VNet, список підмереж, теги, а на виході повертає ідентифікатор VNet і підмереж. Модуль aks отримує ці значення як вхідні змінні, що забезпечує декларативний зв’язок між мережевим рівнем і кластером Kubernetes. Такий підхід відповідає графовій моделі залежностей, описаній у розділі 3, і дозволяє Terraform автоматично визначати правильний порядок створення ресурсів.

Для управління станом інфраструктури використовується **віддалений бекенд** у Azure Storage Account. Це дозволяє:

- зберігати стан централізовано для всієї команди;
- реалізувати блокування стану (state locking), щоб уникнути одночасного apply з різних конвеєрів;
- мати резервну копію файлів стану на випадок збоїв локальних середовищ.

Кожне середовище (наприклад, dev, stage, prod) може мати окремий контейнер або префікс у бекенді, а також власні змінні (.tfvars), де задаються специфічні параметри: розмір кластера, SKU вузлів, кількість реплік, дозволені IP-адреси тощо. При цьому сама структура модулів залишається однаковою, що спрощує супровід і зменшує кількість помилок.

Особливу увагу під час проєктування модулів приділено:

- **іменуванню ресурсів** (узгоджені префікси/суфікси, вказівка середовища, регіону);
- **тагуванню** (теги для обліку вартості, відповідальних осіб, критичності сервісу);
- **параметризації** (мінімум «зашитих» значень, максимум – змінні з розумними дефолтами);
- **можливості розширення** (додавання нових модулів без змін у вже існуючих).

Результатом є набір Terraform-модулів, які можна повторно використовувати в інших проєктах Azure, змінюючи тільки параметри середовищ та складу ресурсів.

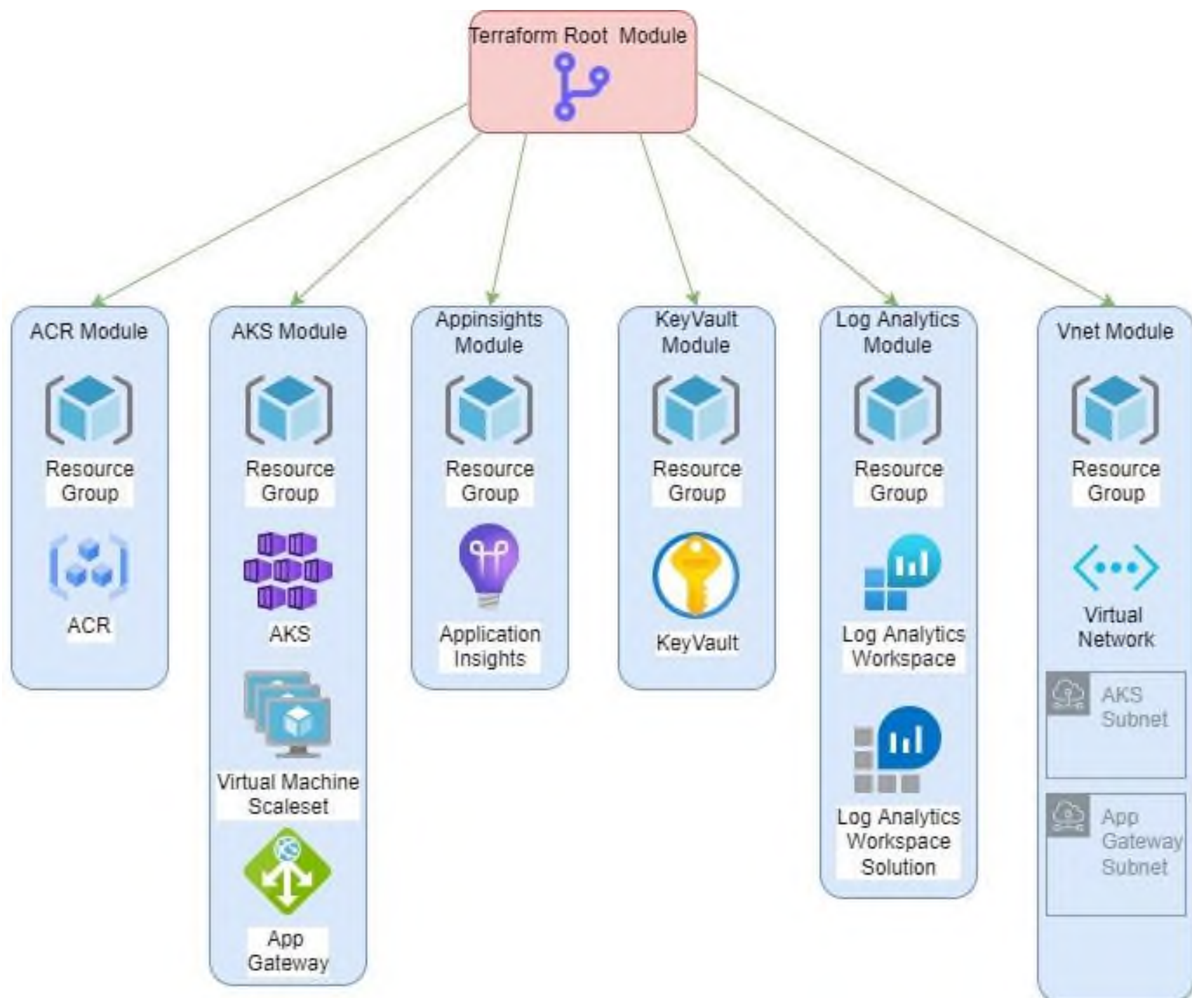

Рисунок 4.1 – Приклад модульної структури Terraform-конфігурацій для ресурсів Microsoft Azure

4.3 Реалізація CI/CD-конвеєрів у Azure DevOps для інфраструктури та прикладних сервісів

Побудова CI/CD-конвеєрів у Azure DevOps виконується на основі YAML-описів, що зберігаються разом із кодом у репозиторіях. Це відповідає принципу «pipeline as code», де логіка збірки й розгортання підлягає версіонуванню, код-рев'ю та перевіркам так само, як і будь-який інший код.

У межах роботи реалізовано **два основні класи конвеєрів**:

1. **Конвеєри для інфраструктури (IaC)** – відповідають за розгортання та оновлення Azure-ресурсів за допомогою Terraform.
2. **Конвеєри для прикладних сервісів** – відповідають за збірку Docker-образів, тестування, публікацію в ACR і розгортання в AKS за допомогою Helm.

Конвеєри для інфраструктури

CI/CD для інфраструктури, як правило, організовано у вигляді двох основних етапів:

- **Етап валідації та планування** (validate/plan):
  - перевірка синтаксису Terraform-конфігурацій (terraform fmt, terraform validate);
  - виконання terraform plan з використанням віддаленого бекенду стану;
  - збереження плану (plan file) як артефакта, а також публікація його текстового представлення для рев'ю.
- **Етап застосування** (apply):
  - запуск terraform apply з використанням затвердженого плану;
  - виконання пост-перевірок (наприклад, простий kubectl get nodes для новоствореного кластера AKS).

Для продуктивних середовищ зазвичай додається **manual approval** між plan та apply: конвеєр зупиняється після генерації плану і чекає підтвердження від відповідальної особи (DevOps-інженера, технічного власника). Це мінімізує ризик неконтрольованих змін у критичних середовищах.

У YAML-конфігурації Azure DevOps описуються [8]:

- тригери (наприклад, запуск при зміні файлів у каталозі envs/prod);
- використання **service connection** до Azure (через service principal або managed identity);
- параметри середовища (ім'я storage account для стану, ім'я resource group, backend config);
- розділення на stages і jobs (наприклад, окремі стадії для dev, stage, prod).

Конвеєри для прикладних сервісів

Для контейнерних застосунків конвеєр зазвичай складається з таких етапів:

1. **Build**:
   - checkout коду із репозиторію;
   - збірка Docker-образу (наприклад, з тегом, що містить номер коміту або версію);

- запуск юніт-тестів (за потреби – інтеграційних тестів);

2. **Publish**:
   - пуш образу до **Azure Container Registry (ACR)**;
   - публікація згенерованих артефактів (наприклад, файлів values.yaml для Helm, якщо вони параметризуються у конвеєрі).
3. **Deploy**:
   - автентифікація до кластера AKS (через az aks get-credentials або Kubernetes service connection);
   - виконання helm upgrade --install із потрібними параметрами (ім’я release, namespace, tag образу, конфігураційні параметри для середовища);
   - базові перевірки після деплою (health-check, smoke-тести).

Аналогічно до інфраструктурних конвеєрів, для продуктивних середовищ може використовуватися окремий stage із manual approval і додатковими перевірками. Тригери деплою можуть бути як автоматичними (наприклад, при мерджі в main), так і ручними (релізи за рішенням команди).

CI/CD-конвеєри тісно інтегровані з Terraform-модулями та Helm-чартами:

- після успішного terraform apply конвеєр прикладних сервісів отримує оновлені значення виходів (наприклад, DNS-імена, назви ресурсів) через змінні середовища або файли конфігурацій;
- Helm-чарти містять посилання на образи в ACR та використовують значення, що відповідають поточному релізу;
- конвеєри моніторингу (наприклад, оновлення дашбордів Grafana) можуть запускатися після інфраструктурних змін, які додають нові сервіси або метрики.

У результаті формується **єдина конвеєрна модель**, де IaC та прикладний код розвиваються узгоджено: будь-яка зміна в інфраструктурі проходить через Terraform-конвеєр, а будь-яка зміна в сервісі — через конвеєр збірки та деплою. Це мінімізує «ручні» дії, зменшує ймовірність помилок і дозволяє відслідковувати весь ланцюжок змін від коміту до продуктивного середовища.

4.4 Організація GitOps-підходу до розгортання застосунків у AKS на базі Helm-чартів

GitOps-підхід розширює класичний CI/CD тим, що робить **Git єдиним джерелом істини** не лише для прикладного коду, а й для усіх Kubernetes-конфігурацій. Замість того, щоб конвеєр «штовхав» зміни в кластер, GitOps-оператор усередині кластера **«тягне» бажаний стан із репозиторію** та синхронізує його з фактичним станом ресурсів. Це особливо добре поєднується з Helm, де весь опис застосунку вже є декларативним [9,13].

У межах розробленого рішення GitOps-підхід організовано навколо таких принципів:

1. **Розділення репозиторіїв**:
   - репозиторій **application code**: містить вихідний код сервісу, Dockerfile та базові Helm-чарти (без прив'язки до конкретного середовища);
   - репозиторій **environment/config**: містить посилання на Helm-чарти (через chart або oci://) та файли values.yaml для середовищ dev, stage, prod. Саме цей репозиторій вважається джерелом істини для конфігурацій кластера AKS.
2. **Опис релізів Helm у Git**:

   Для кожного сервісу й середовища створюється окрема директорія, наприклад:
   - environments/dev/service-a/values.yaml
   - environments/prod/service-a/values.yaml

     Файл values.yaml визначає: версію контейнерного образу (tag із ACR), кількість реплік, ресурси, змінні середовища, налаштування probes тощо. Зміна версії образу або конфігурації відбувається шляхом коміту в Git.
3. **GitOps-оператор у кластері (FluxCD / Argo CD / інший інструмент) [13]**:

   У кластері AKS розгортається GitOps-інструмент, який:
   - періодично опитує Git-репозиторій конфігурацій;
   - при виявленні змін витягує оновлені values.yaml/маніфести;
   - виконує helm upgrade --install (або аналогічну операцію) для відповідних релізів;

- відстежує статус застосунків та сигналізує про розбіжність між станом у Git та станом у кластері.

4. **Взаємодія GitOps із CI-пайплайном**:

   Azure DevOps-пайплайн відповідає за:
   - збірку Docker-образу;
   - пуш образу в ACR;
   - оновлення відповідного values.yaml у Git-репозиторії (наприклад, зміна тега образу);

     Після мерджу змін у main/release-гілку GitOps-оператор автоматично підхоплює нову конфігурацію та оновлює сервіс у AKS. Таким чином, **CI залишається у Azure DevOps, а CD виконується через GitOps-контролер**.

5. **Політики та рев'ю змін**:

   Оскільки будь-яке розгортання в кластер починається з коміту в Git, усі зміни конфігурацій проходять через pull request, код-рев'ю та перевірки (lint, валідація Helm-чартів). Це підвищує прозорість і відтворюваність, а також спрощує аудит (історія змін конфігурацій зберігається в Git).

   Такий підхід дозволяє:

- зменшити кількість прямого доступу операторів до кластера (усі зміни – через Git);
- забезпечити **автоматичний дрифт-менеджмент**: якщо хтось змінить щось вручну в кластері, GitOps-оператор поверне конфігурацію до стану, описаного в репозиторії;
- уніфікувати роботу з кількома середовищами та кластерами AKS, використовуючи ті самі шаблони.

4.5 Інтеграція систем моніторингу, логування та алертингу з AKS

Моніторинг і спостережуваність є обов'язковими компонентами експлуатації хмарних систем, особливо в середовищах, побудованих на Kubernetes. У

розробленому рішенні використовується стек **Prometheus – Grafana – Loki**, інтегрований із кластером AKS [10-11].

Інтеграція Prometheus із AKS

Prometheus розгортається в окремому namespace кластера (наприклад, monitoring) за допомогою офіційного Helm-чарту або kube-prometheus-stack. У конфігурації увімкнено:

- **service discovery для Kubernetes**: Prometheus автоматично знаходить:
  - kubelet, kube-apiserver, kube-proxy та інші системні сервіси;
  - застосунки, що експортують метрики через HTTP-ендпоінти (наприклад, /metrics);
- **scrape-конфіги** для:
  - системних метрик (CPU, пам'ять вузлів, стан подів, використання ресурсів);
  - прикладних метрик (HTTP-запити, час відповіді, бізнес-показники).

Додатково встановлюються **експортери** (node-exporter, kube-state-metrics), які надають детальну інформацію про стан вузлів і об'єктів Kubernetes.

Інтеграція Loki та збір логів

Для централізованого логування використовується **Loki** як система зберігання логів та **Promtail/Fluent Bit** як агент збору [11-12]:

- на кожному вузлі кластера встановлюється агент (DaemonSet), який:
  - читає логи контейнерів із директорій /var/log/containers;
  - додає до них лейбли (namespace, pod, container, app, environment);
  - відправляє потоки логів до Loki через HTTP.
- Loki зберігає логи у вигляді time-series із лейблами, оптимізуючи зберігання та пошук у порівнянні з класичними «важкими» стеком ELK.

Централізація логів дозволяє розробникам та SRE:

- знаходити логи для конкретного релізу/сервісу/namespace;
- корелювати лог-події з метриками (через спільні лейбли й час).

Grafana як єдина точка доступу до телеметрії

Grafana виступає єдиним інтерфейсом для спостереження за станом системи:

- підключаються джерела даних:
    - Prometheus (метрики);
    - Loki (логи);
- створюються дашборди:
    - для загального стану кластера AKS (вузли, поди, namespace, ресурси);
    - для кожного мікросервісу (HTTP-метрики, помилки, латентність);
    - для інфраструктурних компонентів (ingress, ACR, вузли, autoscaler).

Алертинг

Алерти в системі будуються на основі метрик Prometheus (через **Alertmanager**) та правил у Grafana:

- визначаються тригери, пов’язані з SLO [15-16]:
    - доступність сервісу (частка успішних запитів);
    - latency (95-й перцентиль часу відповіді);
    - кількість помилок 5xx, CrashLoopBackOff, падіння кількості реплік;
- налаштовуються канали сповіщення:
    - email, Slack/Teams, вебхуки в системи інцидент-менеджменту.

Таким чином, інтегрований стек моніторингу забезпечує:

- збір і збереження **метрик**, **логів** та (за потреби) **трасувань**;
- побудову дашбордів для різних ролей (Dev, DevOps, SRE, бізнес);
- автоматичні алерти при порушенні SLO або інших критичних показників.

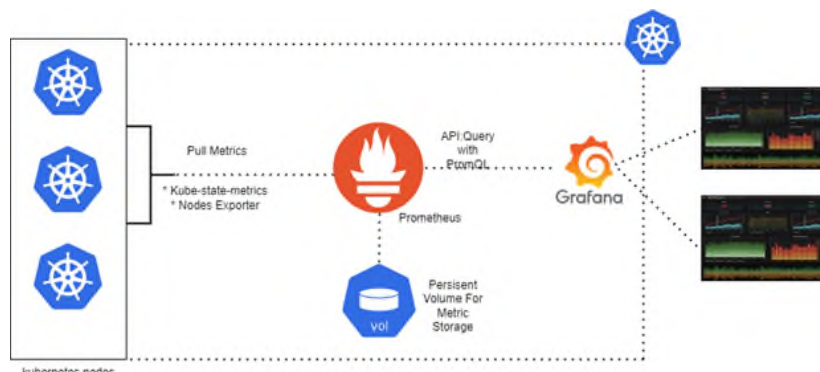

Рисунок 4.2 – Потік метрик від сервісів у AKS до дашбордів Grafana та системи алертингу.

4.6 Тестування, верифікація та експериментальні дослідження розробленого рішення

Після побудови інфраструктури та впровадження CI/CD і GitOps-підходу необхідно перевірити, що система працює коректно, відповідає вимогам та реально покращує процеси розгортання й експлуатації. Для цього проводиться комплекс **тестування та експериментальних досліджень**, які охоплюють кілька рівнів:

Функціональна верифікація Terraform-конфігурацій

На першому етапі перевіряється працездатність IaC:

- виконуються terraform validate та terraform plan для середовищ dev і stage з різними наборами параметрів;
- у dev-середовищі запускається terraform apply із повним розгортанням: ресурсні групи, мережа, ACR, AKS, Key Vault, ресурси моніторингу;
- після створення інфраструктури застосовуються базові перевірки:
  - az aks get-credentials та kubectl get nodes/kubectl get pods -A;
  - перевірка доступності ACR, Key Vault, підключення до кластера з CI-агента.

Результатом є підтвердження того, що модульна структура Terraform дозволяє повністю відтворити хмарне середовище з «чистого» стану та коректно обробляє повторні запуски (ідемпотентність).

Перевірка роботи CI/CD-конвеєрів

Далі тестується поведінка конвеєрів Azure DevOps:

- **для інфраструктури**:
  - імітується зміна параметрів (наприклад, кількості нод у пулі AKS чи типу SKU);
  - конвеєр plan генерує зміну, яка порівнюється з очікуваною;
  - після approval запускається apply, а потім перевіряється, що кластер оновився без простою;
- **для прикладних сервісів**:
  - виконується тестовий коміт у репозиторій застосунку;

    - конвеєр збирає Docker-образ, пушить його в ACR, оновлює Helm-конфігурації та тригерить розгортання в AKS (через GitOps або прямий helm upgrade – залежно від обраної моделі);
    - перевіряється, що нова версія сервісу доступна користувачеві, а попередня може бути відкачана (rollback) у випадку проблем.

Паралельно фіксуються часові характеристики: час збірки, час деплою, повний Lead Time від коміту до доступності сервісу.

Перевірка GitOps-підходу

Для GitOps-сценарію проводяться окремі дослідження:

- змінюється лише values.yaml із новим тегом образу – без прямого втручання у конвеєр;
- фіксується, через який час GitOps-оператор підхоплює зміну й оновлює реліз у кластері;
- імітується «ручне» втручання в кластер (зміна числа реплік або параметрів через kubectl) і спостерігається робота механізму самовідновлення до стану, описаного в Git.

Це демонструє переваги GitOps як механізму захисту від дрейфу конфігурацій і підвищення прозорості змін.

Навантажувальне тестування та аналіз метрик

Для оцінки поведінки системи під навантаженням запускаються прості load-тести цільового сервісу (наприклад, за допомогою k6, JMeter чи інших інструментів):

- формується профіль навантаження (кількість запитів за секунду, тривалість тесту, сценарії запитів);
- під час тесту збираються:
    - метрики використання ресурсів (CPU, RAM, мережа) на рівні pod/node;
    - прикладні метрики (час відповіді, кількість помилок);
    - лог-події (timeouts, помилки 5xx, падіння pod-ів);
- у Grafana аналізуються:
    - відповідність отриманих показників заявленим SLO (латентність, доступність);

- робота autoscaler'ів (додавання/зменшення реплік подів чи вузлів);
- час відновлення після збільшення/зменшення навантаження.

Ці дослідження дозволяють не лише підтвердити, що система «працює», а й оцінити, як вона поводиться при реалістичних сценаріях експлуатації.

Оцінка DevOps-метрик до і після впровадження рішення

Якщо є можливість, порівнюються ключові DevOps-показники до й після впровадження IaC + CI/CD + GitOps:

- середній Lead Time (до – ручні деплої, після – автоматизовані конвеєри);
- частота релізів (до – кілька релізів на місяць, після – декілька на тиждень/день);
- MTTR (зменшення часу відновлення завдяки автоматизованим rollback-ам і спостережуваності);
- Change Failure Rate (зниження завдяки тестам, рев'ю й стандартизації процесу).

Навіть якщо кількісні значення подаються орієнтовно, така оцінка демонструє практичну цінність розробленого рішення.

Висновки до розділу

У четвертому розділі було розроблено та описано практичну реалізацію системи інфраструктури як коду та CI/CD для хмарних сервісів у Microsoft Azure на базі Azure Kubernetes Service.

Обґрунтовано вибір технологічного стеку: Terraform використано для декларативного опису інфраструктури Azure, Azure DevOps – для організації CI/CD-процесів, Helm – для пакетування та розгортання застосунків у AKS, а стек Prometheus–Grafana–Loki – для забезпечення спостережуваності. Показано, що така комбінація інструментів дозволяє побудувати повністю автоматизований ланцюг від коду до продуктивного середовища.

Спроєктовано модульну структуру Terraform-конфігурацій, яка охоплює ресурсні групи, мережу, кластер AKS, реєстр контейнерів, сховище секретів та компоненти моніторингу. На основі Azure DevOps реалізовано окремі конвеєри для інфраструктури й прикладних сервісів, що підтримують принципи «pipeline as code» та забезпечують відтворюваність і контрольованість змін.

На рівні Kubernetes впроваджено GitOps-підхід до розгортання застосунків із використанням Helm-чартів і Git як єдиного джерела істини для конфігурацій кластера. Описано інтеграцію систем моніторингу, логування та алертингу з AKS, що дає змогу відстежувати стан інфраструктури й сервісів, контролювати дотримання SLO та реагувати на інциденти.

Проведено тестування, верифікацію та експериментальні дослідження розробленого рішення: перевірено роботу Terraform-модулів, CI/CD-конвеєрів, GitOps-сценаріїв і стеку спостережуваності під навантаженням. Показано, що запропонований підхід дозволяє скоротити час постачання змін, підвищити стабільність розгортань і забезпечити прозорість експлуатації сервісів у AKS.

Отримані результати створюють основу для подальшого розширення рішення – додавання нових мікросервісів, інтеграції з іншими хмарними сервісами Azure та поглибленого аналізу DevOps-метрик у динаміці.

РОЗДІЛ 5. РОЗРОБЛЕННЯ СТАРТАП-ПРОЄКТУ

5.1 Опис ідеї стартап-платформи IaC та CI/CD-як-сервіс (Platform as a Service)

Ідея стартап-проєкту полягає у створенні спеціалізованої платформи **“IaC & CI/CD as a Service”** для компаній, що використовують або планують використовувати Microsoft Azure та Kubernetes (AKS), але не мають достатньої експертизи в галузі DevOps.

Основна концепція полягає в тому, що замовник отримує **готову платформу для розгортання та експлуатації своїх застосунків**, а не окремий набір скриптів чи рекомендацій. Умовно кажучи, замість того, щоб наймати окрему DevOps-команду, компанія підключається до сервісу, де:

- інфраструктура в Azure описана у вигляді **стандартних Terraform-модулів**, які адаптуються під конкретного клієнта;
- процеси **CI/CD у Azure DevOps** конфігуруються за готовими шаблонами конвеєрів;
- розгортання застосунків у AKS здійснюється через **GitOps-підхід на базі Helm-чартів**;
- моніторинг, логування та алертинг забезпечуються через інтегрований стек **Prometheus – Grafana – Loki** (або гібридно з Azure Monitor).

Платформа пропонується у вигляді **PaaS-рішення**, де ядро системи – це набір:

- перевірених Terraform-модулів для типових лендінг-зон в Azure (VNet, AKS, ACR, Key Vault, стейт, моніторинг тощо);
- шаблонів YAML-конвеєрів Azure DevOps для:
  - збірки й тестування застосунків;
  - розгортання інфраструктури;
  - деплою в AKS (через Helm або GitOps);
- стандартних Helm-чартів для типових сервісів (API, фронтенд, воркери, джоби, cron-и);
- готових дашбордів Grafana і правил алертингу.

Клієнт проходить **онбординг**, під час якого:

1. Визначаються середовища (наприклад, dev/stage/prod), регіон Azure, вимоги до доступності й безпеки.
2. Підбирається або генерується конфігурація Terraform-модулів під цей сценарій.
3. Автоматично розгортаються базові ресурси в Azure.
4. Створюються або підключаються репозиторії в Azure DevOps, додаються шаблонні CI/CD-конвеєри.
5. Налаштовується GitOps-підхід для AKS: створюється конфігураційний репозиторій, підключається GitOps-оператор у кластері.

Таким чином, кінцевий користувач платформи отримує:

- **відтворювану інфраструктуру** в Azure з модульною архітектурою;
- **автоматизовані процеси CI/CD** без необхідності самостійно проєктувати конвеєри;
- **спостережуваність "із коробки"** для сервісів, що працюють у AKS;
- **єдину точку керування** середовищами й релізами через Git та Azure DevOps.

Цінність стартапу полягає в тому, що він перетворює складну для багатьох компаній тему «DevOps + хмара + Kubernetes» на продукт, який можна купити як сервіс, а не будувати з нуля.

5.2 Аналіз ринку, конкурентів та цільової аудиторії

Цільова аудиторія платформи "IaC & CI/CD as a Service" — це насамперед малий та середній бізнес, а також середні/великі продуктові та аутсорсингові компанії, які [17-18]:

- вже використовують Azure або планують міграцію до нього;
- хочуть переходити до мікросервісної архітектури та Kubernetes;
- не мають сильної внутрішньої DevOps/SRE-команди або не готові довго інвестувати в її формування.

Типові приклади таких клієнтів:

- компанії, що мають 3–10 мікросервісів і вже страждають від ручних деплоїв;
- аутсорс-команди, яким потрібно **швидко піднімати стейджинг/прод середовища** в Azure під кожного нового замовника;

- невеликі продуктові стартапи, які хочуть зосередитися на фічах, а не на інфраструктурі.

З погляду ринку, платформа конкурує одночасно з кількома класами рішень:

1. **Нативні сервіси хмарних провайдерів**
   - Azure пропонує власні шаблони (Azure Blueprints, landing zones, Bicep), GitHub Actions, DevOps Starter тощо.
   - Сильна сторона: глибока інтеграція з екосистемою Azure.
   - Слабка сторона: інструменти часто даються як “будівельні блоки”, а не як повністю зібраний продукт.
2. **CI/CD та DevOps-платформи загального призначення**
   - GitLab CI/CD, GitHub Actions, CircleCI, Harness, тощо.
   - Вони дають потужні механізми конвеєрів, але:
     - не завжди дають готові **Azure+AKS+Terraform “рецепти”**;
     - не покривають комплексно весь шлях “IaC + GitOps + observability” саме під Azure.
3. **Консалтинг та інтегратори**
   - Компанії, які “під ключ” будують клієнтові інфраструктуру й CI/CD, але:
     - це довго й дорого;
     - результат часто погано стандартизований, залежить від конкретної команди;
     - повторне використання рішень між клієнтами обмежене.

Позиціонування запропонованої платформи можна описати так:

- **Вужча спеціалізація**: фокус не на “будь-якій хмарі”, а на **Azure + AKS** (це спрощує підтримку і дозволяє робити глибшу інтеграцію).
- **Продукт, а не консалтинг**: клієнт отримує **повторно використану, стандартизовану платформу**, а не разове “ми вам щось налаштували”.
- **Повний ланцюг**: від Terraform-інфраструктури до моніторингу й GitOps, а не лише CI/CD або лише Landing Zone.

Таким чином, ринок уже частково заповнений інструментами та сервісами, але є **ніша готового “DevOps-платформенного продукту під Azure + AKS”**,

орієнтованого на компанії, які хочуть швидкий старт із адекватним рівнем стандартизації й ціни.

5.3 Розроблення ринкової стратегії та моделі монетизації

Ринкова стратегія стартапу базується на моделі SaaS-платформи з консалтинговим онбордингом. Це означає, що:

- основний продукт — це **підписка на платформу**, яка включає:
  - доступ до Terraform-модулів і оновлень;
  - шаблони та апдейти конвеєрів Azure DevOps;
  - Helm-чарти й GitOps-конфігурації;
  - готові дашборди та правила алертингу;
- додатково продаються **послуги впровадження й супроводу**:
  - початковий аудит і міграція існуючої інфраструктури;
  - кастомізація під специфічні вимоги клієнта;
  - підтримка, навчання персоналу.

Модель монетизації

Можливими варіантами монетизації є:

1. **Підписка “per environment / per cluster”**
   - Тарифікація залежно від кількості середовищ (наприклад, dev, stage, prod) або кількості кластерів AKS.
   - Приклад:
     - **Basic**: 1 кластер, 1–2 середовища, обмежене число сервісів.
     - **Pro**: 2–3 кластери, до N сервісів, розширений моніторинг.
     - **Enterprise**: необмежена кількість середовищ/кластерів, кастомні рішення, пріоритетна підтримка.
2. **Плата за онбординг**
   - Одноразова оплата за початкове впровадження платформи: налаштування Terraform, CI/CD, GitOps, моніторингу.

   - Це компенсує витрати на глибоке занурення в контекст конкретного клієнта та знижує бар'єр входу в підписку (після онбордингу використання платформи вже дешевше, ніж свій in-house DevOps).
3. **Додаткові сервіси**
   - аудит DevOps-процесів та оптимізація витрат в Azure;
   - налаштування SLO/SLA, error budget, побудова кастомних дашбордів;
   - підтримка режиму 24/7 для критичних клієнтів.

Go-to-market стратегія

Канали виходу на ринок можуть включати:

- **партнерства з Azure-провайдерами та інтеграторами**:
  - платформа виступає як "швидкий старт" або "стандартна платформа" для їхніх замовників;
- **контент-маркетинг**:
  - публікації технічних статей, гайдів, кейсів "як з нуля підняти AKS із повноцінним CI/CD за X днів";
  - виступи на профільних конференціях і мітапах;
- **цільові продажі (outbound)**:
  - контакт із компаніями, які вже активно використовують Azure, але не мають відпрацьованих DevOps-процесів;
- **free/low-cost пилот**:
  - спрощена версія платформи для одного dev-середовища, щоб показати цінність рішення без великого контракту.

Конкурентні переваги

Основні конкурентні переваги платформи у порівнянні з «чистим консалтингом» чи суто інструментами:

- **стандартизований, але гнучкий шаблон**: є ядро, яке однакове для всіх, і параметри, які налаштовуються для кожного клієнта;
- **продуктова еволюція**: з кожним новим клієнтом платформа вдосконалюється (нові модулі, шаблони), а не створюється щоразу "з нуля";

- **зрозуміла модель витрат** для клієнта — фіксована підписка + прогнозована ціна онбордингу.

5.4 Вимоги до технічного та програмного забезпечення стартап-рішення

Для впровадження платформи «IaC та CI/CD-як-сервіс» необхідно визначити мінімальні та рекомендовані вимоги до технічних і програмних ресурсів як на боці провайдера платформи, так і на боці клієнтів.

Хмарна інфраструктура провайдера

Стартап-рішення базується на сервісах Microsoft Azure, тому основою є **окрема підписка Azure** або виділений **tenant/management group**, у межах якого розгортається інфраструктура платформи та референсні середовища клієнтів.

Базовий набір ресурсів включає:

- **Azure Kubernetes Service (AKS)** – кластер для:
  - розміщення GitOps-оператора (наприклад, FluxCD/Argo CD);
  - розміщення сервісів платформи (панель керування, API, допоміжні сервіси);
  - тестових/демо-середовищ клієнтів (за потреби).
- **Azure Container Registry (ACR)** – для зберігання контейнерних образів:
  - образи компонентів платформи;
  - базові/шаблонні образи клієнтських сервісів.
- **Azure Storage Account** – для:
  - зберігання Terraform state (remote backend);
  - артефактів логів/беккапів конфігурацій.
- **Azure Key Vault** – для безпечного зберігання секретів:
  - облікові дані service principal/managed identities;
  - ключі шифрування, токени доступу до репозиторіїв тощо.
- **Log Analytics Workspace / Azure Monitor** (опційно) – для інтеграції платформеного моніторингу з нативними засобами Azure.

Мінімальна конфігурація кластера AKS для демонстраційного середовища може включати 2–3 вузли середнього класу (наприклад, серії D), однак для продуктивної експлуатації рекомендується планувати:

- окремі node pool'и для:
  - o системних компонентів платформи;
  - o стеку моніторингу (Prometheus, Grafana, Loki);
  - o GitOps-інструментів;
- резерв ресурсу для одночасного обслуговування кількох клієнтів.

Програмне забезпечення платформи

На стороні платформи використовуються:

- **Terraform** (актуальна LTS-версія) – для опису інфраструктури провайдера та референсних конфігурацій клієнтів;
- **Azure DevOps**:
  - o Azure Repos – для зберігання Terraform-модулів, Helm-чартів, GitOps-конфігурацій;
  - o Azure Pipelines – для CI/CD-конвеєрів інфраструктури та сервісів;
  - o Azure Artifacts (опційно) – для зберігання додаткових пакунків;
- **Helm** – як стандартний механізм пакетування й розгортання сервісів у AKS;
- **GitOps-оператор** (FluxCD або Argo CD) – для реалізації підходу GitOps до конфігурацій кластера клієнта;
- **Prometheus, Grafana, Loki** – для збору метрик, логів і візуалізації;
- допоміжні компоненти:
  - o node-exporter, kube-state-metrics та інші експортери для Prometheus;
  - o Promtail/Fluent Bit або інший агент збору логів.

Вимоги до інфраструктури клієнта

З боку клієнта мінімальними технічними вимогами є:

- **Підписка Microsoft Azure** із можливістю:
  - o створювати ресурсні групи у вибраних регіонах;
  - o розгортати AKS, ACR, Key Vault, Storage Account;

    - налаштовувати доступ до Azure DevOps (або GitHub, якщо підтримується альтернативний варіант CI/CD).
- Наявність або готовність створити:
    - **Azure DevOps проєкт** (якщо платформа інтегрується у вже існуючу організацію);
    - або надати доступ до репозиторіїв, де розміщується код сервісів.

На рівні робочих станцій розробників і DevOps-фахівців клієнта рекомендується:

- ОС: Windows 10/11, Linux або macOS;
- встановлені:
    - Git;
    - Terraform (для локального тестування конфігурацій, якщо потрібно);
    - kubectl, Helm (для діагностики та локальної роботи з AKS);
    - Docker Desktop або аналог (для локальної збірки та налагодження контейнерів);
    - сучасне IDE/редактор (Visual Studio Code, JetBrains Rider/IntelliJ та ін.).

Нефункціональні вимоги

До нефункціональних вимог належать:

- **Надійність**: використання Azure Availability Zones для критичних компонентів, регулярний бекап стану Terraform, конфігурацій GitOps та дашбордів моніторингу.
- **Безпека**: шифрування даних у стані спокою й у каналі, використання Azure RBAC, розділення середовищ і subscription за клієнтами, зберігання секретів у Key Vault.
- **Масштабованість**: можливість горизонтального масштабування компонента платформи (GitOps, моніторинг) і додавання нових клієнтів без кардинальної зміни архітектури.
- **Спостережуваність**: наявність дашбордів для самої платформи (стан конвеєрів, статус GitOps-синхронізації, завантаження кластера).

5.5 Оцінка ризиків та дорожня карта розвитку проєкту

Успішність стартап-платформи залежить не лише від технічної якості рішення, але й від здатності передбачати й керувати ризиками, а також планувати розвиток продукту на кілька ітерацій уперед.

Основні ризики проєкту

Технічні ризики:

- **Залежність від одного хмарного провайдера (Azure)**: зміни в API, політиках безпеки, тарифах можуть вплинути на вартість і стабільність рішення. *Мітингуючий захід:* проєктування модулів Terraform та логіки CI/CD таким чином, щоб частину компонентів можна було адаптувати під інші хмари (AWS/GCP) у довгостроковій перспективі.
- **Складність підтримки стандартних шаблонів для різних типів клієнтів**: чим більше варіантів архітектур, тим складніше утримувати платформу в уніфікованому вигляді.

  Мітингуючий захід: жорсткіше визначення «референсних архітектур» і чіткий поділ між «ядром платформи» та «кастомним консалтингом».
- **Безпекові ризики**: помилки в конфігурації доступів (RBAC, Key Vault, service connections) можуть призвести до несанкціонованого доступу до інфраструктури клієнта.

  Мітингуючий захід: застосування принципів least privilege, регулярні аудити доступів, використання policy-as-code (наприклад, Azure Policy, OPA/Gatekeeper).

Організаційні та ринкові ризики:

- **Недостатня зрілість DevOps-практик у клієнтів**: компанія може не бути готовою змінювати процеси та культуру, навіть отримавши технічно якісну платформу.

  Мітингуючий захід: включення до пропозиції навчання, воркшопів і супроводу при переході до нової моделі роботи.
- **Конкуренція з боку великих гравців**: Microsoft та інші постачальники можуть розширити свої нативні DevOps-рішення, частково перекриваючи

функціональність стартапу.

Мітингуючий захід: фокус на нішевих сценаріях (спеціалізовані галузі, глибока кастомізація під AKS, інтеграція зі стеком спостережуваності), швидка реакція на запити реальних клієнтів.

- **Фінансові ризики**: висока вартість початкової розробки та підтримки платформи при невеликій кількості перших клієнтів.

  Мітингуючий захід: поетапний розвиток (MVP → пілот → розширення), максимальне повторне використання напрацьованих рішень і автоматизація онбордингу.

Дорожня карта розвитку проєкту

Дорожню карту можна умовно розділити на кілька етапів.

Етап 1. MVP (мінімально життєздатний продукт)

- Розробка базового набору Terraform-модулів:
  - ресурсна група, VNet, ACR, AKS, Key Vault, Storage Account.
- Створення референсних CI/CD-конвеєрів у Azure DevOps:
  - інфраструктурний pipeline (plan/apply);
  - pipeline для одного типового сервісу (API або web).
- Інтеграція GitOps-підходу для одного кластера AKS.
- Розгортання мінімального стеку моніторингу (Prometheus + Grafana).
- Пілотне впровадження з 1–2 клієнтами/внутрішнім проєктом.

Етап 2. Продуктова стабілізація та розширення функціональності

- Формалізація «референсних архітектур»:
  - типові профілі: малий стартап, середній продукт, аутсорс-проєкт.
- Додавання Loki та повноцінного логування, розширення дашбордів Grafana.
- Підтримка декількох кластерів AKS для одного клієнта (multi-environment/multi-region).
- Поліпшення GitOps-шару (гнучкіші шаблони для Helm, better practices для структури репозиторіїв).
- Автоматизація онбордингу нового клієнта (скрипти, wizard, документація).

Етап 3. Масштабування та оптимізація

- Впровадження multi-tenant-моделі для провайдера платформи:
  - чітке логічне та ресурсне розділення клієнтів;
  - спільні компоненти (моніторинг, артефакти) із ізоляцією даних.
- Оптимізація вартості:
  - типові рекомендації з cost-optimization в Azure;
  - автоматизовані звіти про навантаження й використання ресурсів.
- Поглиблення інтеграції зі SLO/SLA:
  - шаблонні SLO для типових сервісів;
  - вбудовані дашборди з error budget для клієнтів.

Етап 4. Довгостроковий розвиток

- Розгляд підтримки альтернативних CI/CD- та SCM-платформ (GitHub Actions, GitLab CI).
- Поступова підготовка до мультихмарних сценаріїв (AWS EKS / GCP GKE) на базі вже наявної модульної структури Terraform.
- Розвиток аналітичної частини:
  - агрегація DevOps-метрик;
  - рекомендації щодо покращення процесів для клієнтів (DevOps maturity insights).

Висновки до розділу

У п'ятому розділі на основі розробленого технічного рішення було сформовано концепцію стартап-платформи «IaC та CI/CD-як-сервіс» для Microsoft Azure та Azure Kubernetes Service. Описано ідею продукту як платформи, що надає клієнтам відтворювану хмарну інфраструктуру, стандартизовані CI/CD-конвеєри, GitOps-підхід до розгортання та інтегровану систему спостережуваності без необхідності будувати ці рішення з нуля.

Виконано базовий аналіз ринку, конкурентів та цільової аудиторії, що дозволило визначити нішу для платформи: організації, які використовують Azure та прагнуть впровадити сучасні DevOps-практики, але не мають достатньої експертизи або ресурсів для створення власної платформи. Запропоновано модель монетизації на основі підписки та послуг із впровадження, а також описано варіанти позиціонування продукту відносно нативних сервісів Azure, універсальних CI/CD-платформ та консалтингових компаній.

Сформовано вимоги до технічного та програмного забезпечення як для провайдера платформи, так і для клієнтів, включно з переліком необхідних хмарних ресурсів, інструментів та нефункціональних характеристик. Проведено якісну оцінку ризиків (технічних, організаційних, ринкових) і запропоновано заходи їх мінімізації. Запропоновано дорожню карту розвитку стартапу від мінімально життєздатного продукту до масштабованої платформи з підтримкою багатьох клієнтів і розширеним функціоналом.

Таким чином, розділ 5 демонструє, що розроблене в роботі технічне рішення може бути використане не лише як навчальний або внутрішній інженерний кейс, але й як основа для реального комерційного продукту, здатного закривати актуальні потреби ринку у сфері автоматизації хмарної інфраструктури та DevOps-практик.

ВИСНОВКИ

У магістерській роботі розглянуто комплексну задачу побудови та автоматизації хмарної інфраструктури для контейнерних сервісів на базі Microsoft Azure та Azure Kubernetes Service із застосуванням підходів «Інфраструктура як код» (IaC) та безперервної інтеграції/деплою (CI/CD). На основі аналізу сучасних DevOps-практик, інструментів керування інфраструктурою та систем моніторингу розроблено та впроваджено цілісне рішення, яке може бути використане як у навчальних цілях, так і як прототип реальної виробничої платформи.

У першому розділі досліджено стан проблемної області: розглянуто еволюцію DevOps-підходу, місце концепції IaC у життєвому циклі програмного забезпечення, а також роль хмарних платформ, зокрема Microsoft Azure, у побудові сучасних розподілених систем. Окреслено ключові можливості Azure Kubernetes Service як керованого сервісу оркестрації контейнерів і виконано огляд інструментів IaC та CI/CD, що найчастіше застосовуються для хмарних рішень (Terraform, ARM/Bicep, Azure DevOps, GitHub Actions, Helm тощо).

У другому розділі сформовано інформаційну модель системи. Визначено вимоги до хмарної інфраструктури, описано цільову архітектуру рішення в Microsoft Azure, включно з мережевою топологією, кластерами AKS, реєстром контейнерів, сховищем секретів та компонентами спостережуваності. Проаналізовано інформаційні потоки й артефакти (стан інфраструктури, конфігурації, журнали, метрики), а також змодельовано ролі та сценарії доступу до системи IaC та CI/CD. Окрему увагу приділено вимогам до надійності, безпеки, масштабованості й спостережуваності.

У третьому розділі побудовано формальну та модельну основу розробленого рішення. Запропоновано формальну модель життєвого циклу декларативної інфраструктури, де конфігурація у вигляді коду виступає єдиним джерелом істини, а реальний стан хмарних ресурсів розглядається як відображення цієї конфігурації. Інфраструктуру описано у вигляді графа залежностей ресурсів, що дозволяє коректно визначати послідовність операцій створення, оновлення та видалення. CI/CD-конвеєр змодельовано як автомат зі станами, що переходить від коміту до розгортання та

верифікації змін. Розглянуто DevOps-метрики (Lead Time, Deployment Frequency, MTTR, Change Failure Rate) та їх зв'язок із SLI/SLO, SLA й бюджетом помилок (error budget) для сервісів у AKS.

У четвертому розділі реалізовано практичну частину роботи. Обґрунтовано вибір технологічного стеку: Terraform для опису інфраструктури Azure, Azure DevOps для побудови CI/CD-процесів, Helm для пакетування застосунків у Kubernetes, а також стек Prometheus–Grafana–Loki для моніторингу й логування. Розроблено модульну структуру Terraform-конфігурацій для ресурсних груп, мережі, AKS, ACR, Key Vault і компонентів спостережуваності. Реалізовано конвеєри Azure DevOps для інфраструктури та прикладних сервісів, а також організовано GitOps-підхід до розгортання застосунків у AKS на базі Helm-чартів. Інтегровано систему збору метрик, логів та алертингу, що забезпечує спостережуваність за станом як інфраструктури, так і сервісів.

У п'ятому розділі запропоновано концепцію стартап-проєкту на основі розробленого технічного рішення. Сформульовано ідею платформи «IaC та CI/CD-як-сервіс» для Azure та AKS, визначено цільову аудиторію, проаналізовано конкурентне оточення й окреслено можливі моделі монетизації. Наведено вимоги до технічного та програмного забезпечення стартап-рішення, виконано якісну оцінку ризиків і розроблено дорожню карту розвитку проєкту від мінімально життєздатного продукту до масштабованої платформи.

У результаті виконання роботи:

- сформовано та реалізовано шаблон інфраструктури як коду для Microsoft Azure з використанням Terraform;
- побудовано CI/CD-конвеєри в Azure DevOps для автоматизованого розгортання інфраструктури та контейнерних сервісів у AKS;
- впроваджено GitOps-підхід до керування конфігураціями Kubernetes-кластера;
- інтегровано стек спостережуваності Prometheus–Grafana–Loki для моніторингу метрик, логів і показників якості сервісів;
- продемонстровано можливість використання розробленого рішення як основи для комерційної платформи.

Наукова новизна роботи полягає в комплексному поєднанні підходів IaC, GitOps та DevOps-метрик у контексті керованого Kubernetes-сервісу Azure Kubernetes Service, а практичне значення — у можливості безпосереднього застосування розробленого шаблону для створення відтворюваних хмарних середовищ та автоматизації процесів розгортання й експлуатації контейнерних сервісів.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Humble J., Farley D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. – Addison-Wesley, 2011. – 512 p.
2. Kim G., Debois P., Willis J., Humble J. The DevOps Handbook. – IT Revolution Press, 2016. – 480 p.
3. Burns B., Beda J., Hightower K. Kubernetes: Up and Running. – 2nd ed. – O'Reilly Media, 2019. – 368 p.
4. Burns B., Grant B., Oppenheimer D., Brewer E., Wilkes J. Borg, Omega, and Kubernetes. // *Communications of the ACM.* – 2016. – Vol. 59, No. 5. – P. 50–57.
5. Terraform Documentation. HashiCorp. – Режим доступу: https://developer.hashicorp.com/terraform/docs (дата звернення: 20.9.2025).
6. Brikman Y. Terraform: Up & Running. – 3rd ed. – O'Reilly Media, 2022. – 430 p.
7. Azure Kubernetes Service (AKS) Documentation. Microsoft Learn. – Режим доступу: https://learn.microsoft.com/azure/aks/ (дата звернення: 17.10.2025).
8. Azure DevOps Services Documentation. Microsoft Learn. – Режим доступу: https://learn.microsoft.com/azure/devops/ (дата звернення: 17.9.2025).
9. Helm – The Kubernetes Package Manager. – Офіційна документація. – Режим доступу: https://helm.sh/docs/ (дата звернення: 25.9.2025).
10. Prometheus Documentation. – Режим доступу: https://prometheus.io/docs/ (дата звернення: 25.10.2025).
11. Grafana Documentation. – Режим доступу: https://grafana.com/docs/ (дата звернення: 12.11.2025).
12. Grafana Loki Documentation. – Режим доступу: https://grafana.com/docs/loki/latest/ (дата звернення: 15.10.2025).
13. FluxCD Documentation. – Режим доступу: https://fluxcd.io/docs/ (дата звернення: 20.11.2025).
14. Forsgren N., Humble J., Kim G. Accelerate: The Science of Lean Software and DevOps. – IT Revolution Press, 2018. – 288 p.
15. Betts B. та ін. Site Reliability Engineering: How Google Runs Production Systems. – O'Reilly Media, 2016. – 552 p.

16. Beyer B., Jones C., Petoff J., Murphy N. The Site Reliability Workbook. – O'Reilly Media, 2018. – 552 p.
17. Azure Architecture Center – Cloud Design Patterns. Microsoft. – Режим доступу: https://learn.microsoft.com/azure/architecture/ (дата звернення: 2.12.2025).
18. CNCF Cloud Native Interactive Landscape. Cloud Native Computing Foundation. – Режим доступу: https://landscape.cncf.io/ (дата звернення: 2.12.2025).

ДОДАТОК А - Приклади Terraform-конфігурацій для розгортання інфраструктури

```
terraform {
  required_version = ">= 1.5.0"

  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~> 3.100"
    }
  }

  # У реальному середовищі бекенд для стану зберігається в Azure Storage
  backend "azurerm" {
    resource_group_name  = "rg-tfstate"
    storage_account_name = "sttfstateexample"
    container_name       = "tfstate"
    key                  = "demo-dev.terraform.tfstate"
  }
}

provider "azurerm" {
  features {}
}

locals {
  project     = "demo"        # умовна назва проєкту
  env         = "dev"         # середовище
  location    = "westeurope"  # регіон Azure
  name_prefix = "${local.project}-${local.env}"

  tags = {
    project     = local.project
    environment = local.env
    owner       = "devops"
  }
}
```

```
# Ресурсна група
resource "azurerm_resource_group" "rg" {
  name     = "${local.name_prefix}-rg"
  location = local.location
  tags     = local.tags
}

# Віртуальна мережа
resource "azurerm_virtual_network" "vnet" {
  name                = "${local.name_prefix}-vnet"
  location            = azurerm_resource_group.rg.location
  resource_group_name = azurerm_resource_group.rg.name
  address_space       = ["10.10.0.0/16"]

  tags = local.tags
}

# Підмережа для AKS
resource "azurerm_subnet" "aks" {
  name                 = "${local.name_prefix}-snet-aks"
  resource_group_name  = azurerm_resource_group.rg.name
  virtual_network_name = azurerm_virtual_network.vnet.name
  address_prefixes     = ["10.10.1.0/24"]

  # Для Azure CNI / AKS може знадобитися delegation / service endpoints
}

# Log Analytics Workspace для моніторингу
resource "azurerm_log_analytics_workspace" "law" {
  name                = "${local.name_prefix}-law"
  location            = azurerm_resource_group.rg.location
  resource_group_name = azurerm_resource_group.rg.name
  sku                 = "PerGB2018"
  retention_in_days   = 30

  tags = local.tags
```

```
}

# Azure Container Registry
resource "azurerm_container_registry" "acr" {
  name                = replace("${local.name_prefix}acr", "-", "")
  resource_group_name = azurerm_resource_group.rg.name
  location            = azurerm_resource_group.rg.location
  sku                 = "Standard"
  admin_enabled       = false

  tags = local.tags
}

# Кластер Azure Kubernetes Service
resource "azurerm_kubernetes_cluster" "aks" {
  name                = "${local.name_prefix}-aks"
  location            = azurerm_resource_group.rg.location
  resource_group_name = azurerm_resource_group.rg.name
  dns_prefix          = "${local.name_prefix}-aks"

  kubernetes_version = "1.29.0" # приклад версії, у реальному проєкті – змінна

  # Системно призначена identity
  identity {
    type = "SystemAssigned"
  }

  default_node_pool {
    name                 = "system"
    vm_size              = "Standard_D4s_v5"
    node_count           = 3
    vnet_subnet_id       = azurerm_subnet.aks.id
    os_disk_size_gb      = 128
    orchestrator_version = "1.29.0"

    tags = local.tags
  }
```

```
  # Інтеграція з Log Analytics
  oms_agent {
    log_analytics_workspace_id = azurerm_log_analytics_workspace.law.id
  }

  # Профіль мережі (приклад для Azure CNI)
  network_profile {
    network_plugin    = "azure"
    load_balancer_sku = "standard"
    outbound_type     = "loadBalancer"
  }

  # RBAC за замовчуванням увімкнено в нових кластерах
  role_based_access_control_enabled = true

  tags = local.tags

  lifecycle {
    ignore_changes = [
      default_node_pool[0].node_count
    ]
  }
}

# Приклад вихідних змінних
output "resource_group_name" {
  value = azurerm_resource_group.rg.name
}

output "aks_name" {
  value = azurerm_kubernetes_cluster.aks.name
}

output "acr_login_server" {
  value = azurerm_container_registry.acr.login_server
}
```

ДОДАТОК Б - YAML-опис CI/CD-конвеєрів у Azure DevOps

azure-pipelines-infra.yml

```yaml
trigger:
  branches:
    include:
      - main
  paths:
    include:
      - infra/

variables:
  vmImage: 'ubuntu-latest'
  tfVersion: '1.6.0'
  azureServiceConnection: 'sc-azure-demo'    # назва service connection
  tfWorkingDir: 'infra/dev'                  # каталог із Terraform-конфігурацією
  environmentName: 'dev'

stages:
- stage: ValidateAndPlan
  displayName: 'Terraform validate & plan'
  jobs:
  - job: tf_plan
    pool:
      vmImage: $(vmImage)
    steps:
    - task: AzureCLI@2
      displayName: 'Install Terraform'
      inputs:
        azureSubscription: $(azureServiceConnection)
        scriptType: bash
        scriptLocation: inlineScript
        inlineScript: |
          curl -fsSL
https://releases.hashicorp.com/terraform/$(tfVersion)/terraform_$(tfVersion)_linux_amd64.zip -o
terraform.zip
          sudo unzip -o terraform.zip -d /usr/local/bin
          terraform -version
```

```
   - task: AzureCLI@2
     displayName: 'Terraform init & validate'
     inputs:
       azureSubscription: $(azureServiceConnection)
       scriptType: bash
       scriptLocation: inlineScript
       workingDirectory: $(tfWorkingDir)
       inlineScript: |
         terraform init -input=false
         terraform validate

   - task: AzureCLI@2
     displayName: 'Terraform plan'
     inputs:
       azureSubscription: $(azureServiceConnection)
       scriptType: bash
       scriptLocation: inlineScript
       workingDirectory: $(tfWorkingDir)
       inlineScript: |
         terraform plan -input=false -out=tfplan
         terraform show -no-color tfplan > tfplan.txt

   - publish: $(tfWorkingDir)/tfplan
     artifact: tfplan
   - publish: $(tfWorkingDir)/tfplan.txt
     artifact: tfplan-text

- stage: Apply
  displayName: 'Terraform apply'
  dependsOn: ValidateAndPlan
  condition: and(succeeded(), eq(variables['Build.SourceBranch'], 'refs/heads/main'))
  jobs:
  - job: tf_apply
    pool:
      vmImage: $(vmImage)
    steps:
```

```
  - download: current
    artifact: tfplan

  - task: AzureCLI@2
    displayName: 'Terraform apply'
    inputs:
      azureSubscription: $(azureServiceConnection)
      scriptType: bash
      scriptLocation: inlineScript
      workingDirectory: $(tfWorkingDir)
      inlineScript: |
        terraform init -input=false
        terraform apply -input=false tfplan
```

azure-pipelines-app.yml

```
trigger:
  branches:
    include:
      - main

variables:
  vmImage: 'ubuntu-latest'
  azureServiceConnection: 'sc-azure-demo'
  acrName: 'demoDevAcr'            # ім'я ACR
  aksResourceGroup: 'demo-dev-rg'    # RG, де розгорнутий AKS
  aksName: 'demo-dev-aks'           # ім'я кластера AKS
  imageName: 'demo-api'
  helmReleaseName: 'demo-api'
  helmChartPath: 'helm/demo-api'
  namespace: 'demo-api'

stages:
- stage: BuildAndPush
  displayName: 'Build & push Docker image'
  jobs:
  - job: build_push
    pool:
```

```
    vmImage: $(vmImage)
  steps:
  - checkout: self
    clean: true

  - task: Docker@2
    displayName: 'Build and push image'
    inputs:
      containerRegistry: 'sc-acr-demo'   # service connection до ACR
      repository: '$(imageName)'
      command: 'buildAndPush'
      Dockerfile: 'Dockerfile'
      tags: |
        $(Build.BuildId)

- stage: Deploy
  displayName: 'Deploy to AKS with Helm'
  dependsOn: BuildAndPush
  jobs:
  - job: deploy_aks
    pool:
      vmImage: $(vmImage)
    steps:
    - checkout: self

    - task: AzureCLI@2
      displayName: 'Get AKS credentials'
      inputs:
        azureSubscription: $(azureServiceConnection)
        scriptType: bash
        scriptLocation: inlineScript
        inlineScript: |
          az aks get-credentials \
            --resource-group $(aksResourceGroup) \
            --name $(aksName) \
            --overwrite-existing
```

```yaml
- task: HelmInstaller@1
  displayName: 'Install Helm'
  inputs:
    helmVersionToInstall: 'latest'

- script: |
    kubectl create namespace $(namespace) --dry-run=client -o yaml | kubectl apply -f -

    helm upgrade --install $(helmReleaseName) $(helmChartPath) \
      --namespace $(namespace) \
      --set image.repository=$(acrName).azurecr.io/$(imageName) \
      --set image.tag=$(Build.BuildId)
  displayName: 'Helm upgrade/install'
```