МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»
ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«ІНФРАСТРУКТУРА ЯК КОД (IAC) НА ОСНОВІ TERRAFORM ДЛЯ МУЛЬТИХМАРНОГО СЕРЕДОВИЩА»

Рівень «магістр»

Галузь знань 12 «Інформаційні технології»

Спеціальність 122 «Комп'ютерні науки»

Виконав здобувач 2 курсу

Астахов Даниїл Юрійович

Керівник: к.т.н., доцент

Трофименко Олена Григорівна

Рецензент: к.т.н., доцент

Чикунов Павло Олександрович

Одеса – 2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ”

Факультет кібербезпеки та інформаційних технологій

Кафедра інформаційних технологій

Галузь знань: 12 Інформаційні технології

Спеціальність: 122 Комп'ютерні науки

Назва освітньої програми: Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедри інформаційних технологій
доцент Наталія Логінова
_____“____”_____ 2025 року

ЗАВДАННЯ
на кваліфікаційну (магістерську) роботу
здобувачу Астахову Даниїлу Юрійовичу

1. Тема роботи: «Інфраструктура як код (IaC) на основі Terraform для мультихмарного середовища»
   Керівник роботи: к.т.н, доцент Трофименко О.Г.
   затверджені наказом НУ «ОЮА» від «10» жовтня 2025 року № 3182-06.
2. Строк подання здобувачем роботи «25» листопада 2025 року.
3. Дата видачі завдання «02» червня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1.	Вибір теми, вивчення літературних джерел та складання плану роботи	15.09.2025	
2.	Підготовка першого розділу роботи та подання його керівнику	01.10.2025	
3.	Підготовка другого розділу роботи та подання його керівнику	14.10.2025	
4.	Підготовка третього розділу роботи та подання його керівнику	01.11.2025	
5.	Підготовка вступу та висновків	07.11.2025	
6.	Доопрацювання роботи з урахуванням зауважень керівника	17.11.2025	
7.	Подача електронного варіанта роботи для перевірки на плагіат	20.11.2025	

Здобувач _____ *Д.Ю. Астахов*_____
(підпис) (ім’я та прізвище)

Керівник роботи _____ *О.Г. Трофименко*_____
(підпис) (ім’я та прізвище)

АНОТАЦІЯ

Кваліфікаційна робота на тему «Інфраструктура як код (IaC) на основі Terraform для мультихмарного середовища» на здобуття другого (магістерського) рівня вищої освіти за спеціальністю 122 «Комп'ютерні науки», освітньою програмою «Комп'ютерні науки». Робота містить 10 рисунків, 5 таблиць, 2 додатків, 49 літературних джерел. Робота виконана на 95 сторінках основного тексту і 101 сторінках загального тексту.

Метою роботи є дослідження та реалізація шаблонів конфігурації мультихмарної інфраструктури на основі використання Terraform з урахуванням регіональних обмежень, вимог надійності, безпеки та масштабованості.

Об’єктом дослідження є процеси розгортання та управління обчислювальною інфраструктурою у хмарних середовищах.

Предметом дослідження є методи та засоби реалізації підходу «Інфраструктура як код» з використанням Terraform у мультихмарних архітектурах.

У роботі систематизовано сучасні підходи до реалізації концепції IaC та проведено порівняльний аналіз інструментів Terraform, Ansible та AWS CloudFormation за критеріями мультихмарної підтримки, масштабованості та інтеграції з DevOps-процесами. Досліджено архітектурні моделі мультихмарних середовищ з виявленням переваг і обмежень кожної моделі. Розроблено модульні Terraform-шаблони для трьох провідних хмарних платформ – AWS, Azure та Google Cloud Platform – з урахуванням регіональних обмежень, вимог GDPR та практик українських IT-компаній. Запропоновано систему багаторівневого забезпечення безпеки при автоматизованому розгортанні.

Ключові слова: інфраструктура як код, IaC, Terraform, мультихмарне середовище, DevOps, CI/CD, хмарна безпека, AWS, Azure, Google Cloud Platform, автоматизація розгортання, управління конфігураціями.

ABSTRACT

Qualification work on the topic "Terraform-based Infrastructure as Code (IaC) for Multi-Cloud Environments" for obtaining the second (master's) level of higher education in the specialty 122 "Computer Science", educational program "Data Analysis and Security". The work contains 10 figures, 5 tables, 2 appendices, 49 references. The work is performed on 95 pages of the main text and 101 pages of the general text.

The purpose of the work is to research and implement multi-cloud infrastructure configuration templates based on the use of Terraform, considering regional restrictions, reliability, security and scalability requirements.

The object of the study is the processes of deploying and managing computing infrastructure in cloud environments.

The subject of the study is methods and tools for implementing the Infrastructure as Code approach using Terraform in multi-cloud architectures.

The work systematizes modern approaches to implementing the IaC concept and conducts a comparative analysis of the Terraform, Ansible and AWS CloudFormation tools according to the criteria of multi-cloud support, scalability, and integration with DevOps processes. Architectural models of multi-cloud environments are studied, identifying the advantages and limitations of each model. Modular Terraform templates are developed for three leading cloud platforms – AWS, Azure and Google Cloud Platform – considering regional restrictions, GDPR requirements, and practices of Ukrainian IT companies. A multi-level security system for automated deployment is proposed.

Keywords: infrastructure as code, IaC, Terraform, multi-cloud environment, DevOps, CI/CD, cloud security, AWS, Azure, Google Cloud Platform, deployment automation, configuration management.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

API – Application Programming Interface

AWS – Amazon Web Services

CI/CD – Continuous Integration / Continuous Deployment

EC2 – Elastic Compute Cloud

GCP – Google Cloud Platform

GCS – Google Cloud Storage

HCL – HashiCorp Configuration Language

IaaS – Infrastructure as a Service, інфраструктура як сервіс

IaC – Infrastructure as Code, інфраструктура як код

IAM – Identity and Access Management, управління ідентифікацією та доступом

JSON – JavaScript Object Notation, текстовий формат для обміну структурованими даними між комп'ютерами

NLP – Neuro-linguistic programming

PaaS – Platform as a Service, платформа як сервіс

RBAC – Role-Based Access Control, модель керування доступом, яка базується на ролях користувачів у системі

TPU – Tensor Processing Units

VPC – Virtual Private Cloud

YAML – Yet Another Markup Language, текстовий формат даних, зручний для читання людиною й концептуально близький до мов розмітки

ПЗ – програмне забезпечення

ВСТУП

За умов сучасної цифрової трансформації та інтенсивного розвитку хмарних обчислень стрімко зростає потреба у створенні масштабованих, відмовостійких і безпечних інформаційних систем. Актуальність теми дослідження цієї кваліфікаційної роботи зумовлена швидким поширенням мультихмарних архітектур, які дозволяють організаціям поєднувати переваги різних провайдерів хмарних послуг задля досягнення гнучкості, зниження ризиків залежності від одного постачальника та оптимізації витрат. У цьому контексті технологія «Інфраструктура як код» (Infrastructure as Code, IaC) є фундаментальною парадигмою, яка забезпечує автоматизацію, стандартизацію та контрольованість процесів розгортання інфраструктури. Особливе місце серед інструментів IaC посідає Terraform, який завдяки своїй модульності та мультиплатформності набув широкого застосування для управління складними гібридними та мультихмарними середовищами.

Об’єктом дослідження є процеси розгортання та управління обчислювальною інфраструктурою у хмарних середовищах.

Предметом дослідження є методи та засоби реалізації підходу «Інфраструктура як код» з використанням Terraform у мультихмарних архітектурах.

Метою роботи є дослідження та реалізація шаблонів конфігурації мультихмарної інфраструктури на основі використання Terraform з урахуванням регіональних обмежень, вимог надійності, безпеки та масштабованості.

Для досягнення поставленої мети передбачено розв’язання таких завдань:

– аналіз сучасних підходів до IaC; вивчення особливостей використання Terraform у різних хмарних платформах;

– проєктування архітектури мультихмарного розгортання;

– формування рекомендацій щодо організації CI/CD-процесів;

– управління станом інфраструктури, секретами та політиками безпеки;

– оцінка практичної ефективності запропонованих рішень.

Методологічну основу дослідження складають системний підхід, методи моделювання та віртуалізації, порівняльний аналіз, експериментальне моделювання архітектурних рішень, а також практичне застосування засобів автоматизації інфраструктури, зокрема Terraform у поєднанні з CI/CD-платформами та інструментами моніторингу.

Наукова новизна одержаних результатів полягає у систематизації кращих практик до організації мультихмарної інфраструктури на основі Terraform з адаптацією до українського контексту.

У роботі обґрунтовується застосування модульної структури Terraform для забезпечення повторного використання та уніфікації конфігурацій, а також пропонується методика інтеграції засобів спостережуваності та аварійного відновлення у мультихмарні сценарії.

Робота має практично-орієнтований характер і демонструє застосування наявних технологій у мультихмарному середовищі. Практична значущість роботи визначається можливістю використання її результатів для підвищення ефективності управління мультихмарними інфраструктурами у комерційних та наукових організаціях.

Публікації з апробацією кваліфікаційної роботи:

1. Трофименко О. Г., Чикунов П. О., Астахов Д. Ю., Молоканов Ю. В., Фаріонова Т.А. Порівняльний аналіз безпекових патернів хмарних платформ у контексті наукової відтворюваності. *Інформатика та математичні методи в моделюванні*. 2025. Т.17, № 4. (депоновано).

2. Астахов Д.Ю., Трофименко О.Г. Інструменти моніторингу в DevOps-архітектурах. *Інформаційні технології: моделі, алгоритми, системи (ITMAS-2025)*: матер. VI Всеукр. наук.-практ. інтернет-конф. (16-17 листопада 2025 р.) Миколаїв: НУК імені адмірала Макарова, 2025. С. 453-456. URL: https://itconf.nuos.edu.ua/2025/publications/monitoring-tools-in-devops-architectures/

1 СУЧАСНІ ПІДХОДИ ДО IAC ТА МУЛЬТИХМАРНИХ АРХІТЕКТУР

1.1 Поняття та переваги інфраструктури як коду (IaC) для DevOps

Розвиток хмарних технологій є одним із ключових напрямів сучасної інформатизації, що визначає ефективність цифрової трансформації бізнесу, науки та державного управління. Останнє десятиріччя позначене переходом від простих моделей оренди обчислювальних ресурсів до комплексних платформ, орієнтованих на масштабованість, автоматизацію та інтеграцію із сучасними методологіями розробки. Якщо на початкових етапах розвитку хмарні обчислення концентрувалися на забезпеченні гнучкого доступу до інфраструктури у форматі IaaS (Infrastructure as a Service), то наразі спостерігається зростання популярності платформних рішень (PaaS, Platform as a Service) та сервісів, орієнтованих на мікросервісну архітектуру, контейнеризацію й оркестрацію за допомогою Kubernetes.

Важливим трендом є поширення мультихмарних та гібридних стратегій, які дозволяють поєднувати ресурси різних провайдерів з метою оптимізації витрат, підвищення відмовостійкості та уникнення ризику «vendor lock-in». Це вимагає розробки нових інструментів управління, здатних забезпечити уніфікований підхід до оркестрації інфраструктури, незалежно від її розподілу між AWS, Microsoft Azure, Google Cloud чи приватними дата-центрами [1]. Одночасно з цим посилюється акцент на безпеці, дотриманні нормативних вимог та автоматизації процесів відповідно до DevOps-практик [2]. Як результат формується потреба у системному підході до управління інфраструктурою, що дозволяє скоротити час розгортання середовищ, підвищити їх передбачуваність та якість обслуговування.

У цьому контексті концепція «Інфраструктура як код» (Infrastructure as Code, IaC) стала однією з базових технологічних передумов переходу до інженерії сучасних хмарних середовищ. IaC описує інфраструктуру у вигляді декларативних чи імперативних конфігураційних файлів, що дозволяє автоматизувати

процеси створення, масштабування, модифікації та видалення ресурсів [3]. Тим самим IaC дозволяє командам автоматизувати та керувати інфраструктурою за допомогою програмного коду, забезпечуючи узгодженість, масштабованість та швидше розгортання. Такий підхід забезпечує прозорість і відтворюваність усіх дій, оскільки інфраструктура визначається так само, як і програмний код, а її стан зберігається у системах контролю версій.

На рис. 1.1 зображено типовий робочий процес IaC, який охоплює основні компоненти та взаємозв'язки між ними.

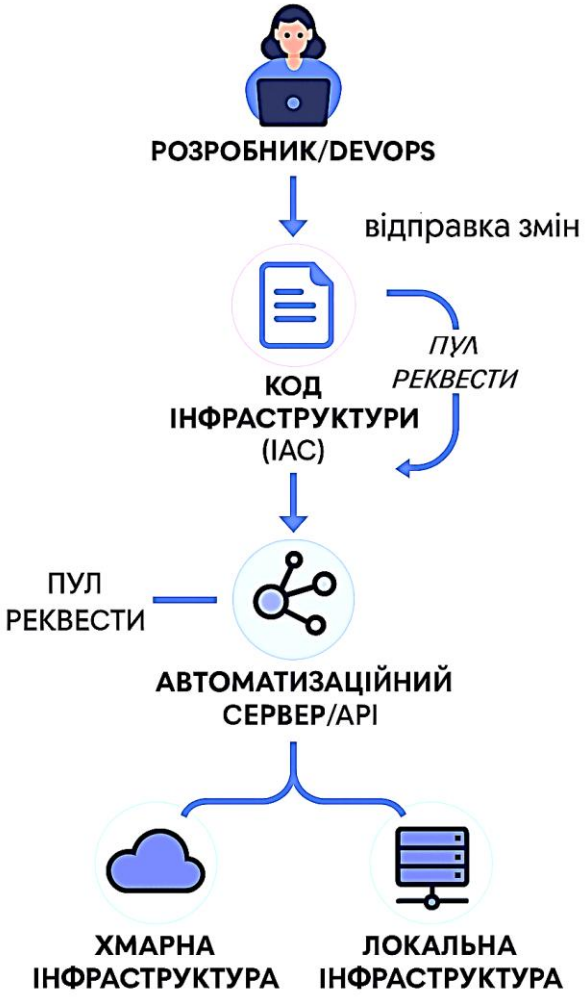

Рисунок 1.1 – Інфраструктура як код

Процес починається з дій розробника або інженера DevOps, який створює код інфраструктури, використовуючи спеціалізовані мови опису, як-от: Terraform, Ansible або CloudFormation. Цей код містить декларативні інструкції

щодо створення, налаштування та управління обчислювальними ресурсами, включаючи сервери, мережі, бази даних та інші компоненти. Після написання, код передається до системи контролю версій, найчастіше Git, що дозволяє зберігати історію змін, здійснювати пул реквести та забезпечувати колективну роботу над інфраструктурними сценаріями.

Затверджений код надходить до автоматизаційного сервера або API-рушія, який виконує його інтерпретацію та застосування. Цей компонент відповідає за інтеграцію з хмарними платформами або локальними дата-центрами, забезпечує розгортання інфраструктури відповідно до заданих параметрів. У результаті інфраструктура може бути реалізована як у хмарному середовищі, так і на локальних серверах, що забезпечує гнучкість та масштабованість IT-рішень.

Важливою особливістю IaC є її здатність забезпечувати ідентичність середовищ, зменшувати ризики людських помилок та сприяти впровадженню практик DevOps, таких як CI/CD. Тим самим інфографіка демонструє не лише технічну послідовність дій, а й концептуальну модель переходу від ручного управління інфраструктурою до її повної автоматизації через код.

Використання IaC у рамках DevOps дає змогу досягати низки важливих переваг. По-перше, автоматизація розгортання суттєво зменшує кількість людських помилок, характерних для ручного налаштування інфраструктури [4]. По-друге, забезпечується повторюваність середовищ, що критично важливо для тестування, розробки та продуктивної експлуатації [5]. По-третє, IaC інтегрується з CI/CD-процесами, дозволяючи створювати інфраструктуру «на вимогу» у відповідь на зміни в програмному забезпеченні (ПЗ). По-четверте, завдяки централізованому управлінню станом та конфігураціями підвищується рівень безпеки й керованості мультихмарних середовищ [6].

Крім того, IaC дозволяє організаціям оптимізувати витрати на хмарні ресурси, програмно керуючи ресурсами, забезпечуючи автоматичне вилучення невикористаних ресурсів. Це запобігає непотрібним витратам на хмарні ресурси та їх втратам. Сучасні інструменти IaC підтримують багатохмарні середовища, дозволяючи організаціям безперешкодно розгортати робочі навантаження в

AWS, Azure та Google Cloud. Це дозволяє уникнути прив'язки до постачальника та підвищує гнучкість [4].

Отже, IaC є важливою практикою в сучасному DevOps, яка пропонує швидкість, масштабованість, автоматизацію та узгодженість в управлінні інфраструктурою. Використовуючи інструменти IaC, як-от: Terraform, Ansible та AWS CloudFormation, команди DevOps можуть оптимізувати операції, знизити витрати та покращити доставку програмного забезпечення.

1.2 Порівняльний аналіз інструментів IaC

Серед численних інструментів IaC найпоширенішими є Terraform, Ansible, AWS CloudFormation.

1.2.1 Terraform

Terraform від компанії HashiCorp є одним із найбільш поширених інструментів IaC з відкритим кодом [7]. Він пропонує попередньо написані модулі для побудови та управління інфраструктурою. Terraform реалізує декларативний підхід і забезпечує підтримку багатьох провайдерів, що робить його придатним для DevOps-команд, які працюють із мультихмарними та гібридними інфраструктурами [8].

Серед ключових переваг Terraform варто виділити мультихмарність, що забезпечується широкою підтримкою провайдерів: від публічних хмар (AWS, Azure, GCP) до приватних інфраструктурних рішень. Іншою важливою перевагою є централізоване управління станом інфраструктури, яке дозволяє уникати розбіжностей між кодом та фактичними ресурсами [9]. Крім того, Terraform підтримує модульність та повторне використання конфігурацій, що сприяє зменшенню дублювання коду й підвищує масштабованість рішень. Важливою є й інтеграція з CI/CD-процесами, яка забезпечує неперервність і контрольованість змін [10].

Разом з тим Terraform має певні недоліки. Одним із них є складність початкового налаштування, особливо для користувачів без досвіду роботи з деклара-

тивними мовами конфігурації. Крім того, управління файлами стану може стати проблемою у великих командах без належної організації віддаленого бекенду. Порівняно з інструментами конфігураційного управління Ansible, Terraform менш зручний для виконання оперативних завдань із налаштування систем, оскільки його головним призначенням є саме створення й модифікація інфраструктурних ресурсів, а не адміністрування прикладного ПЗ.

Специфіка застосування Terraform полягає в орієнтації на інфраструктурні сценарії, що потребують високого рівня відтворюваності та масштабованості. Це робить його незамінним у мультихмарних середовищах, де виникає потреба в уніфікованому підході до керування ресурсами різних провайдерів. Terraform особливо ефективний для середовищ, які швидко змінюються, наприклад у тестуванні, автоматизованому розгортанні застосунків чи побудові гібридних архітектур.

Terraform поєднує гнучкість, масштабованість та мультихмарність із високим рівнем автоматизації, що робить його провідним інструментом реалізації концепції IaC у сучасних DevOps-практиках. Його використання дозволяє суттєво зменшити витрати часу на управління інфраструктурою, підвищити стабільність розгортання та забезпечити контрольованість змін, хоча водночас вимагає від організацій відповідної культури роботи з кодом та управління станом.

1.2.2 Ansible

Ansible, розроблений компанією Red Hat, є одним із найбільш поширених інструментів автоматизації у сфері DevOps та управління інфраструктурою як кодом. Його специфіка полягає у поєднанні підходів до управління конфігураціями, оркестрації та розгортання програмного забезпечення, що забезпечує універсальність у використанні. Ansible базується на декларативному описі бажаного стану системи у форматі YAML, і при цьому, на відміну від Terraform, застосовує імперативний механізм виконання завдань через «playbooks» [3]. Така архітектура дозволяє одночасно описувати складні сценарії розгортання та зберігати зрозумілий для користувача формат конфігурацій [11]. Це робить Ansible

зручним для автоматизації не лише хмарної інфраструктури, а й широкого спектра операційних завдань, зокрема налаштування серверів, інсталяції програмного забезпечення та управління користувачами.

Однією з головних переваг Ansible є відсутність потреби у встановленні спеціальних агентів на керованих вузлах. Взаємодія відбувається через стандартні протоколи, зокрема SSH, що значно спрощує інтеграцію та зменшує адміністративні витрати на підтримку середовища. Додатковою сильною стороною інструмента є його низький поріг входження: навіть користувачі з обмеженим досвідом програмування можуть швидко опанувати основи завдяки простоті синтаксису та наявності великої кількості готових модулів. Широка екосистема та спільнота користувачів сприяють постійному розвитку Ansible, забезпечуючи підтримку різноманітних платформ і технологій.

Разом з тим Ansible має низку обмежень, які впливають на його застосування у складних сценаріях. Порівняно з Terraform, який підтримує централізоване зберігання стану інфраструктури, Ansible не має власного механізму управління станом. Це означає, що у великих системах відстеження змін потребує додаткових інструментів або інтеграцій. Іншою проблемою є зниження продуктивності при масштабуванні, оскільки виконання великої кількості завдань через SSH може призводити до затримок. Крім того, орієнтація на імперативний підхід ускладнює повторне використання складних конфігурацій та знижує рівень передбачуваності результатів у порівнянні з декларативними інструментами [12].

Специфіка використання Ansible полягає у його придатності для задач, пов’язаних з конфігурацією серверів, автоматизацією розгортання програмного забезпечення та виконанням операційних завдань. У цьому контексті він виступає ефективним інструментом для оперативного управління життєвим циклом програмних систем, проте менш зручний для побудови мультихмарних архітектур або довготривалого управління інфраструктурою, порівняно з Terraform чи Pulumi. Водночас завдяки простоті, гнучкості та активній підтримці з боку Red Hat, Ansible зберігає лідируючі позиції серед інструментів автоматизації, особ-

ливо у тих організаціях, де важливим є швидке розгортання середовищ та інтеграція з DevSecOps-практиками [13].

Таким чином, Ansible вирізняється простотою використання та широкою функціональністю, що робить його надзвичайно ефективним для управління конфігураціями та автоматизації операційних завдань. Проте відсутність управління станом і зниження ефективності при масштабуванні обмежують його застосування у сценаріях побудови мультихмарних та високонавантажених систем. Ці особливості визначають місце Ansible у сучасній екосистемі IaC як інструмента, орієнтованого передусім на конфігураційні та експлуатаційні завдання.

1.2.3 AWS CloudFormation

AWS CloudFormation є нативним інструментом IaC, орієнтованим на хмарні сервіси Amazon Web Services. Він забезпечує можливість декларативного опису інфраструктури у форматах JSON або YAML та дозволяє автоматично створювати, оновлювати й видаляти ресурси у середовищі AWS. Користувач визначає бажаний стан ресурсів у вигляді шаблона, після чого CloudFormation автоматично виконує створення, оновлення або видалення об’єктів у межах так званих «стеків». Такий підхід дозволяє забезпечити узгоджене й відтворюване керування інфраструктурою, що особливо важливо для масштабних корпоративних систем, де потрібна стандартизація процесів [14].

Специфіка робочого процесу CloudFormation полягає у тісній інтеграції з усіма сервісами AWS. Це створює умови для побудови складних архітектур, що охоплюють обчислювальні ресурси, бази даних, системи зберігання, мережеву інфраструктуру та сервіси безпеки. Крім того, інструмент підтримує використання «change sets» для попереднього перегляду змін перед їх застосуванням, що знижує ризики помилок під час оновлення системи. Важливим елементом є також механізм керування залежностями між ресурсами, завдяки чому забезпечується коректне створення чи знищення об’єктів у потрібному порядку [5].

Перевагою CloudFormation є його висока надійність у межах екосистеми AWS, що дозволяє мінімізувати ручні операції та забезпечує інтеграцію з іншими сервісами, як-от AWS CodePipeline чи AWS Config. Це робить інструмент ключовим елементом DevOps-процесів у компаніях, орієнтованих на використання виключно хмарних сервісів Amazon [15]. Додатковою сильною стороною є можливість централізованого управління шаблонами та повторного використання модулів через механізм «nested stacks», що сприяє підвищенню продуктивності команд розробників та адміністраторів.

Водночас інструмент має низку недоліків. Найбільш суттєвим є його прив'язаність до екосистеми AWS, що обмежує застосування CloudFormation у мультихмарних середовищах або гібридних інфраструктурах. Порівняно з універсальними засобами, такими як Terraform, він менш гнучкий і практично не підтримує розгортання ресурсів за межами AWS [16]. Ще однією проблемою є висока складність розробки та підтримки великих шаблонів, які часто стають громіздкими та важкими для читання. Це ускладнює навчання нових користувачів і може знижувати швидкість впровадження змін у складних проєктах.

AWS CloudFormation доцільно застосовувати у випадках, коли компанія орієнтована на інфраструктуру в межах AWS та прагне досягти максимальної інтеграції з сервісами постачальника. Інструмент забезпечує високу стабільність, автоматизацію та відповідність DevOps-практикам, проте втрачає універсальність у контексті мультихмарних рішень і потребує значних зусиль для підтримки складних шаблонів.

Порівняння зазначених інструментів (табл. 1.1, 1.2) свідчить, що універсальність і мультихмарність є ключовими перевагами Terraform, тоді як Ansible залишається оптимальним вибором для автоматизації операційних завдань. AWS CloudFormation виявляє свою ефективність виключно у межах екосистеми Amazon, забезпечуючи глибоку інтеграцію, але водночас обмежуючи можливості роботи з іншими хмарними платформами.

Виконаний порівняльний аналіз сучасних інструментів інфраструктури як коду показує, що Terraform, Ansible та AWS CloudFormation мають різні цілі, ар-

хітектурні підходи та сфери оптимального застосування. Terraform демонструє найбільшу гнучкість у мультихмарних і гібридних середовищах завдяки універсальності та підтримці широкого спектра провайдерів, що робить його фактичним стандартом у задачах розгортання інфраструктури незалежно від платформи.

Таблиця 1.1 – Специфіка та робочий процес інструментів IaC

Характеристика	**Terraform**	**Ansible**	**AWS CloudFormation**
Тип підходу	Декларативний	Імперативно-декларативний	Декларативний
Формат конфігурацій	HCL (HashiCorp Configuration Language)	YAML/INI	JSON, YAML
Основна сфера застосування	Мультихмарні та гібридні інфраструктури	Автоматизація конфігурацій, оркестрація сервісів	Інфраструктура всередині екосистеми AWS
Модель управління	План → Apply (з попереднім «plan»)	Безпосереднє виконання завдань	Створення та управління «стеками»
Інтеграція з CI/CD	Широка підтримка через Terraform Cloud та інші платформи	Легка інтеграція через Playbooks у пайплайнах	Тісна інтеграція з AWS CodePipeline та AWS Config
Гнучкість у мультихмарності	Висока	Середня	Низька (лише AWS)
Тип використання	Provisioning інфраструктури	Configuration management та deployment	Provisioning інфраструктури AWS

Таблиця 1.2 – Переваги та недоліки інструментів IaC

Інструмент	**Переваги**	**Недоліки**
Terraform	Мультихмарність, велика спільнота, модульність, передбачуваний «plan → apply» цикл	Крута крива навчання, потреба у правильному керуванні state, відсутність вбудованого CM
Ansible	Простота синтаксису (YAML), агентless-модель, ефективний CM, широка бібліотека ролей	Менша оптимізація для provisioning, повільніший на великих інфраструктурах, обмежена ідемпотентність
CloudFormation	Повна інтеграція з AWS, надійність, «change sets», підтримка залежностей	Vendor lock-in (AWS only), громіздкі шаблони, складність масштабування та підтримки

Ansible, у свою чергу, спеціалізується на управлінні конфігураціями та автоматизації сервісів, що дозволяє інтегрувати його у DevOps-процеси як засіб забезпечення стабільності та повторюваності налаштувань. AWS CloudFormation, попри обмеження у вигляді vendor lock-in, є оптимальним рі-

шенням для організацій, які використовують інфраструктуру виключно в межах екосистеми Amazon Web Services, оскільки забезпечує максимально тісну інтеграцію з іншими сервісами постачальника.

Узагальнюючи результати, можна зазначити, що кожен із розглянутих інструментів має власну нішу застосування: Terraform є універсальним засобом для мультихмарних стратегій, Ansible забезпечує ефективне управління конфігураціями та сервісами, а CloudFormation доцільно використовувати в межах AWS-орієнтованих інфраструктур. Це підтверджує доцільність комбінованого використання різних інструментів IaC залежно від цілей організації та характеру її цифрової екосистеми.

Загалом Terraform найбільш доцільний у мультихмарних сценаріях, Ansible залишається гнучким для конфігураційних задач, а CloudFormation оптимальний у випадках повної орієнтації на AWS.

Інші інструменти, як-от Pulumi, Chef, Packer, Vagrant та Puppet, застосовуються у більш специфічних сценаріях, орієнтованих на окремих провайдерів або завдання, однак вони формують конкурентне середовище, яке стимулює подальший розвиток методів IaC [1]. Pulumi вирізняється можливістю опису інфраструктури мовами загального призначення (TypeScript, Python, Go, C#), що полегшує інтеграцію з існуючими розробницькими практиками [6]. Chef і Puppet історично розвивалися як системи управління конфігураціями, проте вони можуть виконувати й функції IaC, хоча поступаються у популярності через більшу складність у налаштуванні та підтримці [7]. При цьому Puppet є інструментом для керування конфігурацією, яка дотримується моделі клієнт-сервер. Він потребує розгортання агентів на цільових машинах, перш ніж Puppet зможе почати керувати ними. Packer генерує образи віртуальних машин (не запущені віртуальні машини) на основі наданих кроків. Vagrant створює віртуальні машини за допомогою робочого процесу, а тому його краще використовувати для створення попередньо налаштованих віртуальних машин розробника у VirtualBox [10].

Проведений аналіз показує, що вибір інструмента IaC залежить від контексту використання: для мультихмарних сценаріїв найдоцільнішим є Terraform, для конфігурацій та операційних завдань – Ansible, а для проєктів, повністю орієнтованих на AWS, – CloudFormation. Вибір конкретного рішення визначається поєднанням факторів, таких як вимоги до масштабованості, рівень автоматизації, інтеграція з наявною інфраструктурою та довгострокові стратегічні цілі організації.

1.4 Архітектури мультихмарного середовища

1.4.1 Загальна характеристика мультихмарності

Мультихмарне середовище (multi-cloud) – це інфраструктурна модель, яка інтегрує ресурси та сервіси від кількох незалежних хмарних провайдерів з метою підвищення стійкості, масштабованості та оптимізації витрат на IT. Такий підхід є відповіддю на обмеження традиційної моделі *vendor lock-in*, за якої організація стає надмірно залежною від одного хмарного провайдера, що знижує її гнучкість та адаптивність у динамічному цифровому середовищі [17].

Мультихмарна стратегія є провідною тенденцією у розвитку хмарних технологій, оскільки забезпечує гнучке розміщення робочих навантажень, резервування критичних компонентів та високу відмовостійкість систем. Крім того, використання кількох постачальників сприяє доступу до інноваційних сервісів, оптимізації вартості обчислювальних ресурсів та зменшенню ризиків, пов’язаних із монополізацією хмарного середовища.

Організації, які впроваджують мультихмарні стратегії, отримують можливість оптимального вибору сервісів відповідно до функціональних, фінансових або регіональних вимог. Наприклад, географічне розташування хмарної інфраструктури дозволяє виконувати вимоги до зберігання даних згідно з локальними регуляціями (наприклад, GDPR або HIPAA) [18].

1.4.2 Переваги та недоліки мультихмарності

Використання мультихмарних архітектур сприяє підвищенню операційної гнучкості. Організації отримують можливість вибирати найоптимальніші сервіси кожного провайдера відповідно до функціональних, фінансових чи географічних потреб (рис. 1.2).

Рисунок 1.2 – SWOT-аналіз мультихмарної архітектури

Важливо, що завдяки розподіленню ресурсів у різних регіонах між різними хмарними платформами знижуються ризики втрати даних, покращується стійкість та відновлюваність після аварій, забезпечується виконання вимог щодо відповідності локальним нормативам. Водночас такий підхід дозволяє балансувати навантаження та зменшувати ризики простою, що безпосередньо впливає на неперервність бізнес-процесів.

Ключові переваги мультихмарної стратегії [15, 19]:

– уникнення прив’язки до постачальника підвищує операційну гнучкість і контроль над архітектурою, що полегшує можливість зміни постачальника або впровадження нових послуг без прив’язки до якогось одного постачальника;

– адаптивність до регуляторних змін;

– зниження ризиків збоїв і простоїв, завдяки можливості переносити робочі навантаження з одного провайдера на іншого, що мінімізує час простою та забезпечує резервування, безперервність бізнесу і захищає від регіональних збоїв;

– географічне охоплення: завдяки розгортанню ресурсів у різних регіонах світу покращується продуктивність глобальних користувачів, дозволяє компаніям впроваджувати інноваційні рішення, які були б неможливі з одним хмарним постачальником;

– економічна ефективність внаслідок оптимального розподілу навантажень через те, що компанії можуть вибирати найкращого постачальника для кожного робочого навантаження, оптимізуючи як витрати, так і продуктивність.

Разом із тим, мультихмарність не позбавлена викликів. Її впровадження ускладнюється потребою в кадрових ресурсах із високим рівнем компетентності у сфері хмарної інженерії та кібербезпеки. Управління кількома постачальниками та інтеграція їхніх сервісів в єдину екосистему супроводжується підвищенням операційної складності, необхідністю ретельного моніторингу витрат та підтримання узгоджених стандартів безпеки. Особливим викликом є оптимізація витрат, оскільки тарифи різних провайдерів суттєво відрізняються, а прозора консолідація фінансових показників потребує спеціалізованих інструментів і методик.

Недоліки використання мультихмарної стратегії [19, 20]:

– потреба висококваліфікованого персоналу у сферах cloud-інженерії, безпеки та DevOps, а пошук його може бути складною задачею;

– ускладнення інтеграції хмарних сервісів від різних провайдерів, позаяк оцінка витрат та їх консолідація є складною справою;

– потреба у централізованому моніторингу, уніфікації політик безпеки та стандартизації управління витратами, оскільки керування мультихмарним середовищем є складним через різні підходи до протоколів безпеки і шифрування у різних постачальників хмарних послуг

– потреба використання спеціалізованих інструментів (наприклад, CloudHealth, Flexera, Azure Cost Management) для обліку витрат у середовищах із різними ціноутвореннями та формами білінгу.

Аналіз співвідношення витрат і цінності мультихмарних стратегій свідчить про їхню доцільність у довгостроковій перспективі. Попри збільшення початкових витрат на управління складною інфраструктурою та підготовку персоналу, переваги у вигляді підвищеної гнучкості, інноваційності та масштабованості переважають. Розподіл робочих навантажень між різними платформами дозволяє досягти оптимального балансу між продуктивністю, вартістю та безпекою, що робить мультихмарні стратегії ефективним інструментом для організацій, орієнтованих на стійкість і розвиток у динамічному цифровому середовищі.

1.4.3 Сценарії використання

У контексті сучасної цифрової трансформації мультихмарні стратегії набувають дедалі більшого значення як інструмент забезпечення гнучкості, масштабованості та технологічної стійкості IT-інфраструктур. Під мультихмарністю розуміється одночасне використання кількох хмарних платформ або сервісів, що дозволяє організаціям уникати залежності від одного постачальника, оптимізувати витрати, а також адаптуватися до специфічних вимог різних галузей. Ефективність мультихмарних стратегій проявляється у низці прикладних сфер, де поєднання технологічних, нормативних та операційних чинників створює сприятливе середовище для їх впровадження (рис. 1.3).

Однією з найбільш перспективних галузей для мультихмарного підходу є *освіта та наукові дослідження*. Університети та дослідницькі центри часто стикаються з потребою обробки великих обсягів даних, проведення складних симуляцій, а також забезпечення доступу до обчислювальних ресурсів для різних дисциплін. Наприклад, у міждисциплінарних проєктах, де біоінформатика поєднується з машинним навчанням, може виникати потреба у використанні спеціалізованих сервісів Amazon Web Services для GPU-обчислень, паралельно з платформами Google Cloud для зберігання даних та спільної роботи [21]. Такий підхід дозволяє не лише оптимізувати витрати, а й забезпечити відповідність академічним стандартам відкритості та реплікації результатів.

Рисунок 1.3 – Сфери можливого впровадження мультихмарних стратегій

У сфері *охорони здоров'я* мультихмарні стратегії демонструють високу ефективність у контексті обробки медичних даних, телемедицини та аналітики пацієнтських записів. Наприклад, великі медичні установи можуть використовувати приватні хмари для зберігання конфіденційної інформації, що підпадає під дію нормативів HIPAA або GDPR, водночас застосовуючи публічні хмари для обробки знеособлених даних з метою навчання моделей штучного інтелекту

[22]. Такий розподіл дозволяє забезпечити баланс між безпекою та інноваційністю, що є критично важливим у медичній практиці.

Фінансовий сектор, зважаючи на високі вимоги до надійності, безпеки та швидкості обробки транзакцій, також активно інтегрує мультихмарні рішення. Банківські установи, наприклад, можуть використовувати хмару Microsoft Azure для внутрішніх операцій, що потребують високого рівня відповідності стандартам безпеки, водночас застосовуючи сервіси IBM Cloud для аналітики ризиків або прогнозування ринкових трендів [23]. Така архітектура дозволяє зменшити час реагування на зміни ринку, а також забезпечити неперервність бізнес-процесів у випадку збоїв на окремих платформах. Крім того, мультихмарність у фінансовій сфері забезпечує відновлення після аварій і неперервність бізнесу, оскільки дублювання даних і сервісів між кількома провайдерами знижує ризик втрати критичної інформації.

У виробничих процесах мультихмарність проявляє себе через інтеграцію з Інтернетом речей (IoT), що дозволяє здійснювати моніторинг обладнання, прогнозувати технічне обслуговування та оптимізувати логістичні ланцюги. Наприклад, компанія, яка займається автомобілебудуванням, може використовувати хмару Oracle для управління ERP-системами, а хмару Google Cloud – для обробки даних із сенсорів у реальному часі [24]. Така комбінація забезпечує оперативність ухвалення рішень та зменшення витрат на простої.

У сфері *електронного урядування* мультихмарні стратегії сприяють децентралізації державних IT-сервісів, підвищуючи їхню стійкість до кібератак та забезпечуючи масштабованість. Наприклад, у системах електронного документообігу або реєстрів громадян, критично важливі дані можуть зберігатися у національних приватних хмарах, тоді як публічні сервіси – використовуватися для надання доступу до відкритих даних або інтеграції з міжнародними платформами [18]. Це дозволяє забезпечити прозорість і водночас зберегти контроль над критичною інформацією.

У сфері *розробки програмного забезпечення* мультихмарність відкриває нові можливості для DevOps-практик. Команди розробників можуть використо-

вувати різні хмарні середовища для тестування, розгортання та моніторингу застосунків, що дозволяє уникати обмежень окремих платформ. Наприклад, розробка мікросервісної архітектури з використанням контейнерів у Kubernetes може бути розподілена між AWS та Azure, що забезпечує гнучкість масштабування та підвищує стійкість до збоїв [14].

Відтак мультихмарні стратегії є не лише технологічним трендом, а й інструментом стратегічного управління IT-ресурсами, що дозволяє адаптуватися до складних викликів сучасного цифрового середовища. Їхнє ефективне застосування потребує глибокого розуміння галузевих специфік, нормативних вимог та архітектурних принципів.

Сучасні рішення на кшталт VMware CloudHealth або AWS Cloud Manager створюють умови для централізованого управління мультихмарною інфраструктурою. Вони забезпечують уніфікований моніторинг, контроль витрат і підтримання належного рівня безпеки у розподілених середовищах. Це особливо важливо з огляду на потребу гармонізації політик безпеки та управління доступом у різних хмарних платформах, які часто мають власні специфічні протоколи та стандарти.

1.4.4 Архітектурні моделі мультихмарного середовища

Архітектури мультихмарного середовища використовують різні моделі інтеграції (табл. 1.3), які відрізняються способом інтеграції, рівнем управління та цільовим призначенням.

Таблиця 1.3 – Основні моделі архітектури мультихмарного середовища

Архітектурна модель	**Характеристика**	**Переваги**	**Недоліки**	**Типові сфери застосування**
Горизонтальна інтеграція (*multi-cloud load balancing***)**	Розподіл навантаження між кількома публічними хмарами з використанням балансувальників або глобальних оркестраторів	Висока доступність, масштабованість, зниження ризику простоїв	Складність уніфікації безпеки та моніторингу	Електронна комерція, фінансові сервіси, глобальні онлайн-платформи
Вертикальна інтеграція	Функціональний поділ інфраструкту-	Оптимізація продуктивності під конк-	Ускладнена інтеграція та підтри-	Аналітичні платформи, Big

Архітектурна модель	Характеристика	Переваги	Недоліки	Типові сфери застосування
(*functional split*)	ри за шарами: аналітика, зберігання, обчислення – у різних провайдерів	ретні завдання, використання найкращих сервісів	мка сумісності	Data, системи IoT
Гібридна мультихмара (*public + private***)**	Поєднання приватної хмари (для чутливих даних) і публічних хмар (для масштабованих сервісів)	Високий рівень безпеки та відповідності нормам, масштабованість	Високі витрати на побудову приватної хмари	Охорона здоров'я, державний сектор, критична інфраструктура
Multi-cloud for compliance (*географічний розподіл*)	Використання кількох хмарних регіонів для забезпечення відповідності локальному законодавству	Відповідність регуляторним вимогам, географічна близькість до користувачів, низька затримка	Збільшення витрат на адміністрування та дублювання даних	Банківська сфера, міжнародні корпорації, державний сектор

Модель горизонтальної інтеграції (multi-cloud load balancing) ґрунтується на принципі розподілу робочого навантаження між кількома публічними хмарами із застосуванням балансувальників або глобальних оркестраторів. Такий підхід забезпечує неперервність роботи сервісів, зменшує ризик простоїв і сприяє масштабуванню в умовах змінних навантажень. Водночас ключовим викликом є уніфікація політик безпеки та ефективний моніторинг у різних середовищах, що ускладнює адміністрування [25]. Найбільш доцільним є застосування цієї моделі в електронній комерції, фінансових сервісах і глобальних онлайн-платформах, де критично важливими є стійкість і висока доступність.

Вертикальна інтеграція (functional split) поділяє інфраструктуру за функціональними рівнями: аналітика, зберігання чи обчислення можуть розгортатися у різних постачальників. Завдяки цьому вдається оптимізувати роботу під конкретні завдання та використовувати унікальні переваги сервісів кожної хмари. Недоліком є підвищена складність інтеграції та підтримки сумісності між гетерогенними середовищами [20]. Цей підхід найчастіше застосовується в аналітичних системах, рішеннях Big Data та платформах Інтернету речей (IoT).

Гібридна мультихмара (public + private) поєднує ресурси приватних і публічних хмар, що дозволяє одночасно досягати високого рівня безпеки для чут-

ливих даних та масштабованості для динамічних сервісів. Основним обмеженням є значні витрати на створення і підтримку приватної інфраструктури, однак ця модель залишається незамінною для галузей з підвищеними вимогами до захисту даних, таких як охорона здоров'я, державний сектор та критична інфраструктура.

Модель *multi-cloud for compliance* базується на використанні кількох хмарних регіонів для дотримання вимог щодо суверенітету даних та локального законодавства. Вона дозволяє не лише забезпечити відповідність регуляціям, а й наблизити ресурси до кінцевих користувачів, покращуючи якість обслуговування. Недоліком цього підходу є зростання витрат на адміністрування та дублювання даних, однак для банківської сфери, міжнародних корпорацій і державних органів така архітектура є необхідною умовою функціонування [26].

Розглянуті моделі демонструють різні шляхи організації мультихмарної архітектури і дають змогу балансувати між продуктивністю, безпекою, витратами та регуляторними вимогами залежно від стратегічних потреб організації. Вибір оптимальної архітектурної моделі мультихмарного середовища залежить від стратегічних цілей організації, пріоритетів у сфері безпеки, продуктивності, витрат та гнучкості.

У кожному випадку визначальним фактором стає здатність забезпечити уніфіковане управління інфраструктурою, стандартизацію процесів розгортання та контроль над використанням ресурсів.

Особливу роль у побудові мультихмарних архітектур відіграє інфраструктура як код (IaC), яка дозволяє абстрагувати складність різних провайдерів та реалізувати єдиний підхід до опису, автоматизації та відтворення ресурсів. Використання інструментів IaC, зокрема Terraform, забезпечує можливість централізованого керування розподіленими інфраструктурними компонентами, синхронізації політик безпеки та інтеграції з CI/CD конвеєрами. Це створює основу для впровадження підходів DevOps і DevSecOps у мультихмарних середовищах, де важливою є не лише швидкість і повторюваність процесів, а й їхня відповідність стандартам безпеки.

Актуальність мультихмарних архітектур підтверджується зростанням вимог до високої доступності сервісів та відповідності регуляторним нормам щодо зберігання й обробки даних. Організації дедалі частіше комбінують сервіси від провайдерів Amazon Web Services, Microsoft Azure та Google Cloud Platform для досягнення оптимального балансу між продуктивністю, надійністю та вартістю [20]. Водночас мультихмарна стратегія ускладнює процеси адміністрування, оскільки потребує уніфікації мережевих політик, захисту даних на різних рівнях та забезпечення сумісності сервісів.

Загалом архітектури мультихмарного середовища формують нову парадигму управління цифровими інфраструктурами, де ключовим фактором є інтеграція автоматизованих засобів керування, стандартизація процесів та впровадження IaC як фундаментальної методології. Подальші дослідження у цій сфері спрямовані на оптимізацію моделей оркестрації, розробку політик безпеки та формування універсальних підходів до інтеграції різних хмарних провайдерів у єдину узгоджену систему.

Мультихмарні архітектури формують нову парадигму управління цифровими інфраструктурами. Вони дозволяють досягти високого рівня гнучкості, інноваційності та безпеки за умов зростаючої складності технологічного ландшафту. Ключовими факторами ефективності таких підходів залишаються стандартизація процесів, автоматизація через IaC та уніфіковане управління ресурсами.

1.5 Проблеми автоматизації інфраструктури в мультихмарі

Автоматизація інфраструктури в мультихмарному середовищі є критичним викликом для сучасних IT-систем, які функціонують за умов розподілених обчислень, гетерогенних платформ та динамічних навантажень. На відміну від традиційної хмарної архітектури, мультихмарність одночасно використовує кілька незалежних хмарних провайдерів, що ускладнює процеси оркестрації, моніторингу, безперервного розгортання та управління ресурсами.

Однією з ключових проблем є відсутність уніфікованих стандартів для автоматизації між різними хмарними платформами. Наприклад, інструменти типу Terraform, Ansible або Pulumi забезпечують декларативне управління інфраструктурою, проте їхня інтеграція з API різних провайдерів (AWS, Azure, Google Cloud) потребує додаткових адаптацій, що ускладнює масштабування та підтримку [13]. Крім того, різні моделі автентифікації, політики безпеки та формати метаданих створюють бар'єри для побудови єдиного автоматизованого конвеєра розгортання.

Іншою суттєвою проблемою є забезпечення узгодженості конфігурацій та стану ресурсів у мультихмарному середовищі. У випадках, коли одна частина інфраструктури оновлюється в одній хмарі, а інша – в іншій, виникає ризик конфліктів, збоїв у сервісах або втрати даних. Це особливо критично для високонавантажених систем, таких як фінансові платформи або медичні реєстри, де автоматизація має гарантувати не лише швидкість, а й консистентність та відмовостійкість.

Також слід зазначити проблему обмеженої прозорості та спостережуваності (observability) у мультихмарному середовищі. Хоча сучасні рішення, такі як Prometheus, Grafana, Datadog або OpenTelemetry, дозволяють збирати метрики та логи з різних джерел, їх інтеграція у мультихмарну архітектуру потребує складної конфігурації та часто не забезпечує повної картини стану системи [1, 25]. Це ускладнює автоматичне реагування на інциденти, масштабування ресурсів та оптимізацію витрат.

Окрему уваги заслуговують питання безпеки автоматизованих процесів. У мультихмарному середовищі автоматизація часто стосується збереження облікових даних, токенів доступу та конфігурацій у централізованих сховищах, що створює потенційні точки вразливості. Використання секрет-менеджерів, як-от HashiCorp Vault або AWS Secrets Manager, частково вирішує проблему, проте не усуває ризики, пов'язані з міжхмарною передачею даних та автентифікацією.

Крім того, автоматизація в мультихмарі потребує висококваліфікованих фахівців, здатних працювати з різними інструментами, мовами опису інфра-

структури та системами контролю версій. Це створює додаткове навантаження на команди DevOps та SRE, а також підвищує вартість впровадження мультихмарних стратегій [19].

Тим самим, автоматизація інфраструктури в мультихмарному середовищі є багатогранною проблемою, що охоплює технічні, організаційні та нормативні аспекти. Її вирішення потребує розробки уніфікованих протоколів, вдосконалення інструментів оркестрації, а також формування нових підходів до безпеки та управління конфігураціями.

1.6 Висновки до розділу

Інфраструктура як код (IaC) є одним із ключових чинників розвитку сучасних підходів до DevOps, оскільки забезпечує автоматизацію процесів розгортання, управління та відтворення інфраструктури. Її застосування сприяє підвищенню узгодженості конфігурацій, зменшенню кількості помилок, прискоренню розробки та забезпеченню надійності цифрових сервісів. Поняття IaC є основою для ефективної реалізації мультихмарних стратегій, де критичним фактором є здатність координувати та уніфікувати управління ресурсами у різних середовищах.

Порівняльний аналіз інструментів IaC показав, що Terraform, Ansible та AWS CloudFormation мають суттєві відмінності у своїх моделях і сферах застосування. Terraform виявився найбільш універсальним завдяки мультихмарній підтримці та можливості централізованого управління інфраструктурою різних провайдерів. Ansible продемонстрував переваги у сфері управління конфігураціями та оркестрації сервісів, проте менш оптимізований для задач повного provisioning. AWS CloudFormation, у свою чергу, виявився високоефективним у межах екосистеми Amazon Web Services, але його застосування обмежується відсутністю мультихмарної гнучкості та залежністю від одного постачальника. Загалом вибір інструмента IaC залежить від архітектурних потреб організації: універсальність Terraform, конфігураційна сила Ansible чи нативна інтеграція CloudFormation.

Аналіз архітектур мультихмарного середовища засвідчив, що їх розвиток зумовлений необхідністю забезпечення стійкості, гнучкості та масштабованості цифрових інфраструктур. Використання мультихмари дозволяє уникати залежності від одного постачальника, оптимізувати витрати та гарантувати високу доступність сервісів. Проте таке середовище потребує високого рівня автоматизації управління ресурсами, уніфікації конфігурацій та розробки механізмів інтегрованого моніторингу й безпеки.

У ході дослідження також було визначено ключові проблеми автоматизації мультихмарної інфраструктури. Серед них домінують питання ефективного управління станом ресурсів, стандартизації шаблонів, узгодження політик безпеки та інтеграції з CI/CD процесами. Вирішення цих завдань є критично важливим для реалізації концепції DevSecOps у мультихмарних умовах, де від ефективності IaC залежить надійність і захищеність усієї цифрової екосистеми.

Загалом сучасні підходи до IaC і мультихмарних архітектур визначають вектор розвитку інформаційних технологій, а їх інтеграція в практику організацій є необхідною умовою для досягнення конкурентоспроможності, кіберстійкості та інноваційності у сфері управління інфраструктурою.

2 ПРИНЦИПИ РОБОТИ TERRAFORM ТА ЙОГО ВИКОРИСТАННЯ В МУЛЬТИХМАРІ

2.1 Архітектура Terraform: принципи, компоненти та виклики

Terraform є інструментом від компанії HashiCorp для керування інфраструктурою як кодом (Infrastructure as Code, IaC), який дозволяє автоматизувати створення, змінення та управління інфраструктурними ресурсами у хмарних, локальних та гібридних середовищах. Його архітектура побудована на декларативному підході, що забезпечує передбачуваність, повторюваність та контрольованість змін у системах розгортання.

Базовим елементом архітектури Terraform є конфігураційні файли, написані мовою HashiCorp Configuration Language (HCL), яка поєднує читабельність з формальною структурою (приклад див. у Додатку А). Ці файли описують бажаний стан інфраструктури, який Terraform прагне досягти шляхом порівняння поточного стану з цільовим. Такий підхід реалізується через механізм планування (terraform plan), застосування змін (terraform apply) та збереження стану (terraform state), що є центральним компонентом архітектури [7].

Файл стану (state file) відіграє критичну роль у забезпеченні узгодженості між описаною конфігурацією та реальною інфраструктурою. Він містить метадані про ресурси, їхні ідентифікатори, залежності та атрибути. У мультихмарному середовищі, де ресурси можуть бути розподілені між AWS, Azure, Google Cloud та іншими провайдерами, централізоване зберігання стану (наприклад, у backend типу S3 або remote state з використанням Terraform Cloud) дозволяє уникнути конфліктів та забезпечити командну роботу.

Архітектура Terraform також включає провайдери – модулі, які реалізують API-зв’язок із конкретними платформами. Кожен провайдер відповідає за створення, оновлення та видалення ресурсів у відповідному середовищі. У мультихмарному контексті це дозволяє одночасно керувати інфраструктурою в кількох хмарах, використовуючи єдину декларативну модель. Проте це породжує ви-

клики, пов’язані з узгодженням версій провайдерів, автентифікацією, а також з різними семантиками ресурсів у різних хмарах (рис. 2.1).

Рисунок 2.1 – Архітектура Terraform

Ще одним важливим компонентом є модулі – повторно використовувані блоки конфігурації, які дозволяють структурувати код, зменшити дублювання та забезпечити масштабованість. У складних мультихмарних сценаріях модулі можуть бути організовані у вигляді бібліотек, що підтримують параметризацію та умовну логіку, що сприяє адаптивності архітектури [5].

У додатку А наведено приклад коду конфігурації для фінансово-технологічної компанії, назвемо її FinTech Corp, яка за допомогою Terraform реалізує інфраструктуру у хмарі AWS. Наведена конфігурація описує створення

віртуальної приватної мережі (VPC), балансувальника навантаження (ALB) для рівномірного розподілу трафіка, групи EC2-серверів з автоскейлінгом для бекенду, керованої бази даних PostgreSQL у сервісі RDS та бакета для збереження статичних файлів у S3. Така архітектура, зафіксована у HCL-файлах, дозволяє швидко розгорнути багаторівневий вебсервіс із високою доступністю, а також масштабувати його під час зростання навантаження. Будь-які зміни у вимогах, наприклад необхідність збільшення кількості серверів у групі або зміна параметрів бази даних, вносяться у конфігураційні файли й автоматично синхронізуються з реальною інфраструктурою. Це усуває ризик помилок, що виникають унаслідок ручного адміністрування, та забезпечує відтворюваність результатів.

Terraform управляє станом у цьому прикладі наступним чином:

– початковий стан: у компанії немає жодних ресурсів у AWS;

– описаний стан: HCL-файли описують бажану інфраструктуру: VPC, балансувальник, EC2-інстанси, RDS, S3;

– terraform plan: Terraform порівняє бажаний стан із поточним (порожнім) та покаже план створення ~10 ресурсів;

– terraform apply: Terraform створює всі ресурси. У terraform.tfstate фіксується фактичний стан (ARN-и, IP, DNS). По суті, файл стану Terraform (terraform.tfstate) – це файл у форматі JSON, де Terraform зберігає поточний стан керованої інфраструктури. Він зіставляє ресурси, визначені у конфігурації, з реальними ресурсами;

– модифікації: якщо змінити кількість інстансів у desired_capacity з 2 на 3, Terraform збалансує групу автоскейлінгу;

– видалення (terraform destroy): видаляються всі ресурси, а стан очищається.

У цьому сценарії Terraform дозволяє: централізовано керувати різними компонентами (мережею, обчислювальними ресурсами, БД, сховищем); автоматично підтримувати синхронізацію між бажаним і фактичним станом; забезпечувати масштабованість і відмовостійкість; уникати ручної конфігурації десятків сервісів у консолі AWS. Тобто компанія FinTech Corp отримує повноцінну

продакшн-архітектуру, яку можна легко відтворити чи масштабувати в іншому регіоні або навіть у мультихмарному середовищі. Цей приклад добре продемонструє, як Terraform дозволяє описати комплексну архітектуру як єдину конфігурацію, а не вручну налаштовувати ресурси.

Інноваційність Terraform проявляється у поєднанні декларативного опису ресурсів із можливістю прогнозування наслідків змін через команду «terraform plan». Ця функціональність дозволяє ще до застосування бачити різницю між поточним і бажаним станом та приймати обґрунтовані рішення щодо модифікації інфраструктури. Іншою визначальною можливістю є інтеграція з конвеєрами CI/CD, де Terraform виступає інструментом неперервного керування інфраструктурою. Це сприяє реалізації концепцій DevOps і DevSecOps, оскільки інфраструктура розглядається не як статична складова, а як гнучкий і керований кодом об’єкт [9].

Загалом Terraform демонструє сукупність властивостей, які роблять його ефективним засобом для створення й управління складними мультихмарними системами. Його специфіка полягає у здатності уніфікувати робочі процеси в середовищах різних провайдерів, інноваційність – у механізмах управління станом та планування змін, а ключові можливості – у забезпеченні відтворюваності, масштабованості та інтеграції з сучасними підходами автоматизації [10]. Саме це визначає його роль як одного з центральних інструментів цифрової трансформації інфраструктур у контексті глобального розвитку хмарних технологій.

З точки зору безпеки, архітектура Terraform має інтеграцію з системами управління секретами (наприклад, Vault, AWS Secrets Manager), а також контроль доступу до стану та конфігурацій. Це особливо важливо у мультихмарному середовищі, де витік облікових даних може мати катастрофічні наслідки.

Попри переваги архітектура Terraform має низку обмежень. Зокрема, відсутність вбудованої підтримки циклічних залежностей, складність у реалізації умовних сценаріїв та обмежена підтримка імперативної логіки можуть ускладнити автоматизацію в динамічних середовищах. Неправильні конфігурації є однією з провідних причин збоїв та витоків даних [27]. Крім того, у мультихмар-

ному контексті виникає проблема узгодження політик доступу, що потребує додаткових механізмів контролю та валідації.

Загалом архітектура Terraform є потужним інструментом для побудови автоматизованої, масштабованої та контрольованої інфраструктури, особливо у мультихмарних сценаріях. Її ефективне використання потребує глибокого розуміння принципів декларативного моделювання, управління станом, безпеки та інтеграції з хмарними API, що відкриває перспективи для подальших досліджень у галузі автоматизації IT-інфраструктур.

2.2 Основні концепції Terraform: провайдери, ресурси, змінні, модулі

У сучасних практиках управління інфраструктурою Terraform виступає як система декларативного опису бажаного стану, де ключові концепти – провайдери, ресурси, змінні та модулі – утворюють семантичну основу для побудови відтворюваної, автоматизованої та тестованої інфраструктури.

Провайдери в Terraform є адаптерами до зовнішніх платформ. Вони реалізують набір ресурсів і джерел даних, якими Terraform оперує під час синхронізації бажаного та поточного стану. Конфігурація провайдера задається у блоках provider і може включати параметри регіону, облікові дані й інші опції, які визначають спосіб доступу до API провайдера (див. Додаток Б). Належне визначення провайдера є фундаментом: від нього залежить коректність створення ресурсів, їхня ідентифікація у стані та сумісність з API. У прикладі для AWS це виглядає так:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
  backend "s3" {
    bucket         = "fintechcorp-terraform-state"
    key            = "prod/terraform.tfstate"
    region         = "eu-central-1"
    dynamodb_table = "terraform-locks"
  }
}
provider "aws" {
```

```
  region = "eu-central-1"
}
```

Тут секція required_providers явним чином фіксує джерело та версію провайдера, а backend S3 з таблицею DynamoDB демонструє типову практику віддаленого зберігання стану та блокування, яка критично важлива для командної роботи.

Ресурси є примітивами інфраструктури, які описуються у конфігураційних файлах та зіставляються з реальними об'єктами провайдера. Кожний ресурс має тип, локальне ім'я і набір аргументів, що визначають його конфігурацію. Важливим аспектом є використання мета-аргументів і спеціальних можливостей Terraform: count, for_each, depends_on та lifecycle дозволяють контролювати кількість, залежності та поведінку ресурсів під час їхнього оновлення або видалення. Розглянемо приклад створення групи екземплярів у автоскейлінгу та правила безпеки з використанням for_each:

```
resource "aws_security_group" "app_sg" {
  name   = "app-sg"
  vpc_id = module.vpc.vpc_id
}

locals {
  ingress_rules = {
    "http"  = { from = 80,  to = 80,  proto = "tcp" }
    "https" = { from = 443, to = 443, proto = "tcp" }
  }
}

resource "aws_security_group_rule" "ingress" {
  for_each = local.ingress_rules
  type        = "ingress"
  security_group_id = aws_security_group.app_sg.id
  from_port   = each.value.from
  to_port     = each.value.to
  protocol    = each.value.proto
  cidr_blocks = ["0.0.0.0/0"]
}
```

Застосування for_each у цьому прикладі демонструє, як можна програмно генерувати набір однотипних ресурсів на основі словника; це підвищує гнучкість та читабельність конфігурацій порівняно з ручним дублюванням блоків.

Змінні в Terraform забезпечують параметризацію конфігурацій та ізоляцію середовищ. Визначення змінних здійснюється у блоках variable з можливістю вказати тип, значення за замовчуванням і правила валідації. Параметризація

сприяє повторному використанню конфігурацій у різних середовищах (dev, staging, prod) без дублювання коду. Для складних конфігурацій ефективним є застосування типу object або map з блоком validation, що дозволяє формалізувати очікувані структури й запобігати помилкам ще на етапі планування. Приклад параметра, який описує конфігурацію інстансу:

```
variable "instance_config" {
  type = object({
    instance_type = string
    ami           = string
    tags          = map(string)
  })

  description = "Configuration for application instances"

  validation {
    condition     = length(var.instance_config.instance_type) > 0
    error_message = "instance_type must be provided"
  }
}
```

Практично змінні можуть задаватися через файли *.tfvars, змінні оточення з префіксом TF_VAR_ або через системи секретного менеджменту; важливо позначати чутливі параметри як sensitive, щоб запобігти їх виведенню у логах.

Модулі в Terraform реалізують механізм інкапсуляції та повторного використання конфігурацій. Вони дозволяють виділити логічні елементи архітектури (наприклад, мережу, базу даних чи кластер Kubernetes) в автономні набори ресурсів із власними входами й виходами. Модулі можуть бути локальними (в каталозі modules/) або братися з реєстру (Terraform Registry) чи систем контролю версій. Використання модулів сприяє інженерній дисципліні: вони полегшують тестування, версіонування й аудит інфраструктурних конструкцій. Показовим є приклад виклику модуля для VPC з реєстру:

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "~> 5.13.0"

  name = "fintech-vpc"
  cidr = "10.0.0.0/16"
  azs  = ["eu-central-1a", "eu-central-1b"]
  public_subnets  = ["10.0.1.0/24", "10.0.2.0/24"]
}
```

Після виклику модуль надає вихідні значення, наприклад module.vpc.vpc_id, які можна використовувати в інших модулях або ресурсах,

утворюючи композицію інфраструктури. Розумне версіонування модулів та використання семантичних тагів у репозиторіях забезпечують стабільність і відтворюваність, оскільки оновлення модуля не має непередбачено впливати на споживачів без явного підвищення версії.

Критичною складовою, яка пов'язує провайдери, ресурси, змінні і модулі, є *стан інфраструктури*. Файл стану містить ідентифікатори створених об'єктів, їхні атрибути та метадані, і є джерелом для операцій plan та apply. В промислових проєктах віддалені бекенди зі збереженням стану та блокуванням (наприклад, S3+DynamoDB для AWS або Terraform Cloud) є практикою обов'язковою, оскільки вони запобігають конкуренції за модифікацію і забезпечують трасованість змін. Крім того, робота з terraform plan дозволяє інженерам проаналізувати потенційні зміни перед їх автоматичним застосуванням, що зменшує ризики непродуманих модифікацій.

На рівні інженерних практик ефективне застосування перелічених концепцій вимагає певної організації репозиторія: виділення modules/ для повторно використовуваних компонентів, environments/ для оточень із власними даними, конфігурації бекендів для стану та інтеграції з CI/CD, що виконує перевірки форматування, тести модулів і автоматичне застосування з механізмами схвалення. Такий підхід забезпечує контрольованість змін, спрощує аудит і дозволяє масштабувати командну роботу.

Отже, провайдери, ресурси, змінні та модулі в Terraform утворюють єдину семантичну і практичну платформу для декларативного управління інфраструктурою. Конкретні механізми – від визначення провайдера й версій до використання for_each, валідації змінних і модульної композиції – забезпечують як глибоку конфігурованість, так і відтворюваність рішень, що є критичними характеристиками сучасних мультихмарних і DevOps-орієнтованих середовищ.

2.3 Життєвий цикл Terraform

Кроки робочого процесу Terraform показані на рис. 2.2. Життєвий цикл складається з п'яти послідовних етапів, кожен із яких забезпечує узгодженість

між описом конфігурації у вихідному коді та фактичним станом інфраструктури.

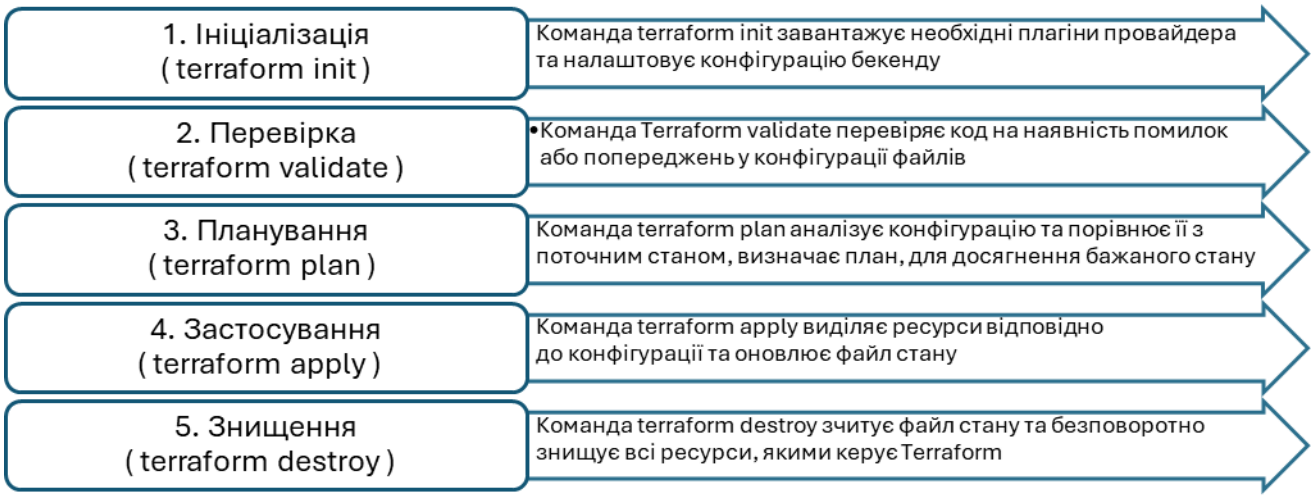

Рисунок 2.2 – Робочий процес Terraform

Першим кроком є ініціалізація (terraform init), під час якої завантажуються необхідні плагіни провайдерів, створюється локальний каталог проєкту та налаштовується бекенд для зберігання стану. Цей етап є обов'язковим перед початком роботи з будь-яким проєктом Terraform.

Наступна фаза – перевірка (terraform validate) – забезпечує синтаксичний контроль коректності файлів конфігурації та виявлення помилок або попереджень, що можуть призвести до некоректного розгортання.

Етап планування (terraform plan) відіграє ключову роль у процесі IaC, оскільки саме на цьому етапі Terraform аналізує конфігураційні файли, порівнює їх із поточним станом інфраструктури та формує детальний план змін. Це дозволяє розробнику оцінити наслідки застосування змін ще до їх виконання.

Після цього відбувається етап застосування (terraform apply), під час якого створюються, змінюються або видаляються ресурси відповідно до попередньо згенерованого плану. Результатом цього кроку є оновлення фактичного стану інфраструктури та збереження його у файлі стану.

Завершальною стадією є знищення (terraform destroy), яке забезпечує видалення всіх створених ресурсів. Цей етап використовується у випадках, коли

необхідно повністю очистити середовище, що є особливо корисним у процесі тестування або в експериментальних сценаріях.

Загалом робочий процес Terraform формує замкнений цикл управління життєвим циклом інфраструктури: від створення до знищення, забезпечуючи повторюваність, автоматизацію та контрольованість змін. Завдяки цьому підходу досягається висока передбачуваність результатів, що є критично важливим для мультихмарних середовищ і DevOps-практик [9].

Terraform підтримує модульність, що спрощує повторне використання конфігурацій, а також забезпечує відстеження стану інфраструктури, завдяки чому зменшується ймовірність розбіжностей між описом та фактичним середовищем [2]. Модульна структура Terraform, розширюваність та можливість інтеграції з системами політик і секрет-менеджменту формують основу для побудови інноваційних сценаріїв автоматизації у сфері хмарних технологій. Terraform поєднує низькорівневі компоненти (обчислювальні екземпляри, сховища та мережі) та високорівневі компоненти (записи DNS та функції SaaS).

Terraform вирізняється широкою мультиплатформною підтримкою. Основною перевагою є використання єдиної мови опису конфігурацій (HCL – HashiCorp Configuration Language, мова конфігурації HashiCorp), яка дозволяє керувати інфраструктурою у різних хмарних провайдерів та локальних середовищах. Terraform підтримує широкий спектр хмарних провайдерів, включаючи Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP) та багатьох інших. Він також підтримує керування ресурсами в локальних середовищах та різні інші служби, такі як провайдери DNS, мережі доставки контенту (CDN) та бази даних [10].

Аналіз робочого процесу Terraform дозволяє зробити висновок, що даний інструмент є одним із найгнучкіших та найпотужніших рішень у сфері реалізації концепції «Інфраструктура як код». Його архітектура побудована на декларативному підході, що дає можливість описувати кінцевий бажаний стан інфраструктури без потреби визначати детальні кроки реалізації. Такий підхід суттє-

во зменшує складність управління великими та мультихмарними середовищами, а також підвищує рівень автоматизації в DevOps-процесах.

2.4 Стан інфраструктури (terraform.tfstate)

Однією з ключових концепцій у Terraform, яка визначає його відмінність від багатьох інших інструментів управління інфраструктурою, є механізм стану. Файл terraform.tfstate є центральним артефактом, який зберігає актуальне уявлення Terraform про стан інфраструктури. Це не лише службовий файл, а й критично важлива складова, яка дозволяє системі підтримувати ідентичність ресурсів, співвідносити бажаний стан із реальним і здійснювати операції зміни у контрольований спосіб.

У найпростішому випадку стан зберігається локально в каталозі з конфігураційними файлами. На рис. 2.3 представлено два основні підходи до організації управління станом у Terraform. Локальний бекенд передбачає збереження файлу terraform.tfstate безпосередньо у файловій системі розробника. Такий підхід простий у налаштуванні та підходить для індивідуальних проєктів, однак він обмежує можливості командної роботи й створює ризик втрати або пошкодження стану. Проте в промислових середовищах локальне зберігання є небезпечним, оскільки воно ускладнює командну роботу і створює ризики розсинхронізації. Саме тому поширеною практикою є використання віддалених бекендів.

Віддалений бекенд, на відміну від локального, забезпечує збереження файлу стану у зовнішньому сховищі, зокрема у хмарних сервісах (Amazon S3, Azure Blob Storage, Google Cloud Storage) або спеціалізованому рішенні Terraform Cloud. Це дозволяє організувати централізоване керування інфраструктурою, забезпечує контроль доступу, версіонування та можливість одночасної роботи кількох користувачів із синхронізованим станом однієї інфраструктури без конфліктів і гарантує відновлюваність після збоїв.

Тим самим схема на рис. 2.3 ілюструє еволюцію від простого, але обмеженого локального підходу до більш складної, проте масштабованої та безпеч-

ної архітектури управління станом, що відповідає вимогам командної DevOps-практики та експлуатації інфраструктури у великих мультихмарних середовищах.

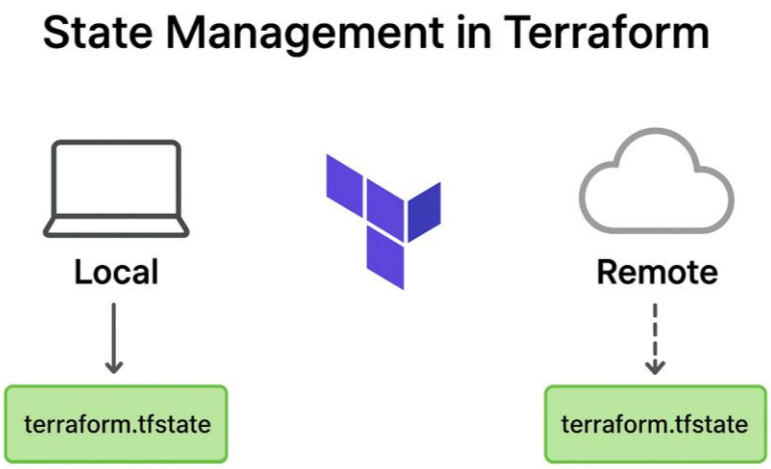

Рисунок 2.3 – Архітектура управління станом у Terraform: локальний та віддалений бекенди

Структурно файл terraform.tfstate є JSON-документом, який описує усі створені ресурси, їхні унікальні ідентифікатори у хмарному провайдері та атрибути конфігурації. Наприклад, після створення віртуальної машини в AWS (EC2) у файлі стану фіксується не лише її локальне ім’я у Terraform, а й реальний ідентифікатор у провайдера (i-0f123456abcde7890), IP-адреса та інші параметри. Це дозволяє Terraform розуміти, що при повторному запуску конфігурації потрібно не створювати новий екземпляр, а оновити існуючий.

Фрагмент реального вмісту terraform.tfstate для EC2-інстансу:

```
{
  "resources": [
    {
      "type": "aws_instance",
      "name": "web_server",
      "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
      "instances": [
        {
          "schema_version": 1,
          "attributes": {
            "id": "i-0f123456abcde7890",
            "ami": "ami-04505e74c0741db8d",
            "instance_type": "t2.micro",
            "availability_zone": "eu-central-1a",
            "private_ip": "10.0.1.23",
            "public_ip": "18.194.111.45",
            "tags": {
              "Name": "web-server-prod"
            }
          }
        }
      ]
    }
  ]
}
```


Цей приклад демонструє, що стан містить повну інформацію про ресурс: від внутрішнього ідентифікатора в Terraform до фактичних параметрів, які були повернуті API AWS. Завдяки цьому Terraform може порівнювати бажану конфігурацію, описану в HCL-файлах, із зафіксованим станом, і визначати, чи потрібно створити, змінити або видалити ресурси.

Інноваційність такого підходу полягає у введенні механізму “single source of truth” для інфраструктури. У традиційних системах адміністрування або навіть при використанні скриптів конфігурація й фактичний стан ресурсів могли легко розійтися, що ускладнювало супровід. У Terraform це виключено: будь-яка зміна повинна пройти через цикл plan → apply, а всі результати фіксуються у стані.

Збереження стану у віддаленому бекенді дозволяє не лише запобігати конфліктам, а й реалізовувати блокування. Наприклад, при одночасному застосуванні змін двома інженерами DynamoDB-блокування не дозволить виконати операцію, доки перший процес не завершиться. Це значно знижує ризик непередбачених ситуацій, наприклад, створення дублікатів ресурсів або втрата частини конфігурацій.

Водночас використання terraform.tfstate має свої обмеження та ризики. Файл зберігає конфіденційну інформацію: паролі, токени доступу, ключі та інші дані, які передаються через провайдери. Тому його потрібно захищати на рівні доступу, шифрування та обмеження політик. Крім того, неправильне видалення або пошкодження стану може призвести до розсинхронізації між реальними ресурсами та декларативною конфігурацією. У таких випадках необхідно виконувати операцію terraform import, щоб повторно прив'язати наявні ресурси до системи.

Загалом terraform.tfstate є критично важливим елементом архітектури Terraform, який забезпечує консистентність, відтворюваність та контрольованість змін інфраструктури. Його роль виходить далеко за межі простої фіксації параметрів і навіть є механізмом, який уможливлює побудову складних мультихмарних систем із високим рівнем надійності та прозорості.

2.5 Управління модулями та повторне використання коду

Управління модулями та повторне використання коду в Terraform становлять фундаментальний механізм інженерної дисципліни при побудові масштабованих, відтворюваних та контрольованих інфраструктур. Модулі в Terraform виступають як абстракції – інкапсульовані блоки конфігурацій, які мають чітко визначені інтерфейси (входи й виходи) і можуть використовуватися як локально в межах організації, так і публічно через реєстри. Практична цінність модулів полягає не лише в економії часу, а й у стандартизації архітектурних патернів, забезпеченні узгодженості конфігурацій між середовищами і зменшенні ризику людських помилок при масштабуванні інфраструктури.

У технічному вимірі коректне управління модулями починається з проєктування їхнього інтерфейсу. Кожний модуль має декларативно визначені змінні (inputs) з типовою інформацією та валідацією, а також вихідні параметри (outputs), які служать контрактом для споживачів. Типовий мінімум для модуля – variables.tf, main.tf, outputs.tf і розгорнутий – README.md, що документує

призначення, очікувані параметри та приклади використання. Наприклад, модуль мережі (VPC) з реєстру може бути підключений у root-модулі так:

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "3.14.0"

  name = "fintech-vpc"
  cidr = "10.0.0.0/16"
  azs  = ["eu-central-1a", "eu-central-1b"]
  public_subnets  = ["10.0.1.0/24", "10.0.2.0/24"]
}
```

Цей приклад ілюструє два важливі практичні аспекти: явне прив'язування до версії (version) та використання модуля як бібліотеки. Фіксація версії запобігає неочікуваним наслідкам при оновленні модуля, що має особливе значення для продуктивних середовищ.

Внутрішня організація модуля має уникати приховування критичних залежностей і не містити налаштувань, які перешкоджають його повторному використанню. Наприклад, модуль не повинен жорстко вказувати провайдера; замість цього слід передбачати можливість передачі провайдер-конфігурацій з зовнішнього контексту, використовуючи механізм providers у блоці module за потреби, або покладатися на спадкування провайдера, реалізоване в Terraform. Конфігурація модуля може містити змінні, позначені як `sensitive = true` для захисту чутливих даних, та механізми валідації, які забезпечують недопущення некоректних значень ще на етапі plan.

Керування життєвим циклом модуля пов'язане із політикою версіонування, випусків та депрецейшн-повідомлень. У виробничих процесах рекомендується застосовувати семантичне версіонування (SemVer): непорогові виправлення можуть бути випущені як патчі, зворотно сумісні поліпшення – як мінори, а зміни інтерфейсу – як мажорні версії. Підхід дозволяє споживачам контролювати оновлення та мінімізувати ризики раптових порушень. Публікація модулів у публічному або приватному реєстрі (Terraform Registry, приватний модуль-реєстр Terraform Cloud або розгортання модулів із Git-репозиторіїв із тегами) полегшує

їхнє повторне використання та забезпечує прозорість змін через хронологію комітів і релізів.

Тестування модулів є невіддільною частиною процесу управління. Традиційні перевірки terraform validate і terraform fmt доповнюються статичними аналізаторами (tflint, checkov) та інтеграційними тестами на базі Terratest (Go) або kitchen-terraform. Приклад мінімального тесту на Terratest виглядає так: ініціалізація каталогу з модулем (terraform init), застосування тимчасової інфраструктури (terraform apply) і перевірка вихідних значень або доступності сервісів через HTTP-запити. Автоматизація таких перевірок у CI-пайплайні (зокрема, із запуском тестів в ізольованих акаунтах або використанням mock-сервісів) підвищує довіру до модуля як компонента, готового до повторного використання.

Приклад структури репозиторія на рис. 2.4 ілюструє інженерну організацію модулів і демонструє практичний підхід. У цій організації modules/ містить ізольовані, версіоновані компоненти, а каталоги envs/ використовують модулі як бібліотеки для побудови конкретних середовищ. CI-конфігурація забезпечує перевірки синтаксису, лінтинг, виконання тестів модулів і автоматичне публікування релізів у приватний реєстр при проходженні політик якості та безпеки.

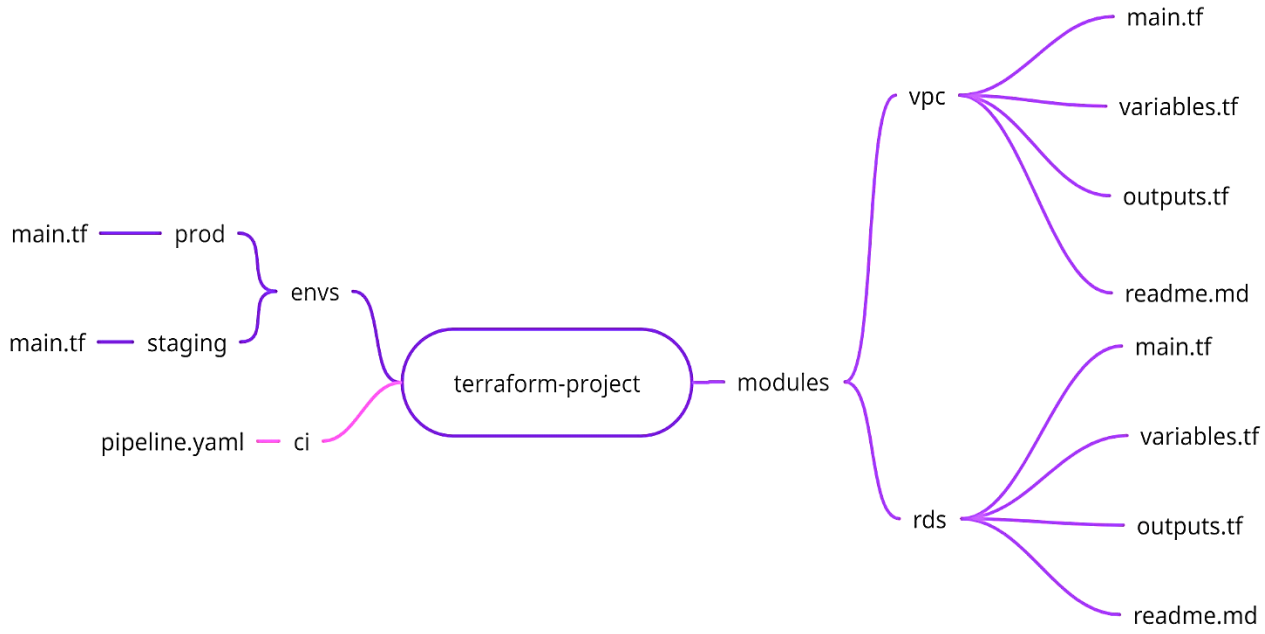

Рисунок 2.4 – Структура репозиторія

Повторне використання коду через модулі підвищує ефективність та узгодженість, але при цьому створює ризики, які потрібно керувати інженерно. Надмірна генералізація модуля, тобто спроба охопити всі можливі сценарії в одному компоненті, веде до зростання складності інтерфейсу і зниження читабельності. Оптимальна стратегія – розбиття відповідальності на вузько спрямовані модулі, які можна композиційно поєднувати. Також важливо забезпечити стабільні, мінімізовані виходи модулів (outputs) та уникати «вилущування» внутрішніх деталей через outputs, які створюють хімічну залежність споживачів від внутрішньої реалізації модуля.

Управління модулями в корпоративному контексті також передбачає політики контролю якості: статичний аналіз коду, вбудовані правила безпеки (policy-as-code) для перевірки ресурсів на етапі plan, а також процеси схвалення релізів. Для прикладу, у великій організації модулі проходять через стадії «development → verified → production», де кожна стадія супроводжується набором автоматизованих перевірок та рев'ю. Після того як модуль пройшов перевірки, він публікується як реліз з тегом у приватному реєстрі, і споживачі у envs/ можуть оновлювати версію модулю свідомо, контролюючи ризики.

Ефективне управління модулями в Terraform ґрунтується на чіткому дизайні інтерфейсів, семантичному версіонуванні, процесах тестування та публікації, а також на дисциплінованій організації репозиторія та CI. При дотриманні цих принципів модулі стають інструментом інфраструктурної інженерії, що зберігає та передає архітектурні патерни, забезпечує повторюваність розгортання й дозволяє масштабувати роботу команд без втрати контролю над якістю та безпекою.

2.6 Практичні приклади налаштування для AWS, Azure, GCP

Хмарні обчислювальні платформи стали фундаментальним компонентом сучасної наукової інфраструктури, особливо в галузі комп'ютерних наук. Вони забезпечують масштабоване середовище для реалізації алгоритмічних моделей, високопродуктивних симуляцій, розгортання нейронних мереж, зберігання великих обсягів даних та автоматизації експериментальних циклів. В умовах об-

межених локальних ресурсів і потреби в гнучкому масштабуванні платформи Amazon Web Services (AWS), Microsoft Azure та Google Cloud Platform (GCP) пропонують інструменти, які дозволяють не лише ефективно керувати обчисленнями, а й забезпечити наукову точність, повторюваність та відтворюваність результатів.

У цьому контексті розглянемо типові сценарії налаштування хмарної інфраструктури для задач комп'ютерних наук, акцентуючи на безпеці, автоматизації та масштабованості.

2.6.1 Приклад налаштування AWS

Одним із показових прикладів ефективного використання AWS у наукових дослідженнях з комп'ютерних наук є побудова інфраструктури для експериментів із машинним навчанням та аналізом складності алгоритмів.

Приклад конкретної реалізації патерна "Provision least-privilege IAM roles by deploying a role vending machine solution" [28] стосується створення середовища, в якому дослідники виконували паралельні обчислення з використанням GPU-прискорювачів на базі EC2-інстансів. З метою забезпечення безпеки доступу та контролю над використанням ресурсів було впроваджено моделі з обмеженими правами (IAM) та механізми тимчасових ролей. Кожен експериментальний модуль, наприклад, класифікація з використанням глибоких нейронних мереж чи тестування швидкодії оптимізаційних алгоритмів, ізольовано у відповідному наборі ресурсів, до яких доступ має лише обчислювальний кластер або конкретний дослідник (автор експерименту). Для контролю за коректністю моделей доступу реалізовано систему автоматичного призначення ролей через CI/CD-пайплайн. Оскільки права задаються лише в міру потреби, немає "широких політик", які потенційно дозволяють доступ до ресурсів поза межами задачі. Використання GitHub Actions з перевірками (Checkov, IAM Access Analyzer) означає "shift left security", тобто помилки політик виявляються ще на стадії коду, не вразливого середовища продакшену [29]. Такий підхід дозволяє реалізувати принцип найменших привілеїв і водночас зберігає гнучкість дослідницько-

го процесу, де обчислювальні потреби швидко змінюються. Це важливо для досліджень еволюційних алгоритмів, коли запуск тисяч ітерацій потребує динамічного масштабування без порушення політик безпеки. Підхід, розглянутий у прикладі, сприяє економії на витратах, пов'язаних із витоком прав і порушеннями безпеки, проте при цьому потрібні інвестиції у створення шаблонів, налаштування CI/CD, навчання команд, аудит та підтримку інструментів. Ризики розглянутого проєкту пов'язані зі складністю управління шаблонами ролей, потребою в підтримці шаблонів та в ясній і виваженій організаційній політиці затвердження критеріїв для надання прав доступу.

Отже, реалізація патерна "Provision least privilege IAM roles by deploying a role vending machine solution" [28] дозволила автоматизувати призначення тимчасових ролей з мінімальними правами, що критично важливо для дослідницьких середовищ, де доступ до ресурсів має бути строго контрольованим. Такий підхід не лише знижує ризики витоку даних, а й забезпечує відповідність принципам відтворюваності, оскільки кожен експеримент ізольований у власному наборі ресурсів із чітко визначеними правами доступу.

2.6.2 Приклад налаштування Azure

У середовищі Microsoft Azure приклади ефективного налаштування можуть стосуватися організації багатоступеневої моделі для тестування та валідації розподілених систем, зокрема архітектур типу microservices або serverless.

Так, в одному з проєктів компанії Risual [30] із дослідження автоматичного масштабування обчислювальних сервісів створено середовище з трьох зон: середовище розробки, експериментальне середовище та продуктивне середовище для фінального розгортання, де PIM в Azure AD налаштовано для обмеження постійних ролей, використання прав високого рівня лише через активацію, MFA, з попереднім переглядом. Усі три зони реалізовані як окремі підписки Azure, а компоненти кожного середовища – як групи ресурсів із чітко визначеними політиками доступу. Усі зміни вносилися через шаблони ARM та контролювалися GitOps-процесами, що гарантувало повторюваність експериментів.

Застосування GitOps-процесів у поєднанні з ARM-шаблонами дозволило досягти високого рівня повторюваності експериментів. Кожна зміна в інфраструктурі проходила через систему контролю версій, що забезпечувало прозорість та можливість ретроспективного аналізу впливу конфігурацій на результати досліджень.

Дослідники, які працювали над моделюванням відмовостійкості систем, мали обмежений доступ лише до ресурсів певного рівня, а для аналізу результатів використовували окремий аналітичний стек на базі Azure Monitor та Log Analytics. Особливої уваги заслуговує мережеве моделювання: для перевірки стійкості до атак типу DDoS або затримок у передачі даних було створено віртуальну мережу із кількома підмережами, ізольованими за допомогою груп безпеки та маршрутизованими через WAF. Це дозволило дослідити поведінку сервісів у змодельованих критичних умовах з мінімальними ризиками для основної інфраструктури.

2.6.3 Приклад налаштування Google Cloud Platform

Прикладом прикладного налаштування Google Cloud Platform (GCP) для досліджень у галузі комп’ютерних наук стало використання архітектури з централізованою мережею та відокремленими обчислювальними доменами для реалізації задач у сфері обробки природної мови (NLP) та глибокого навчання. Створений проєкт забезпечив спільну мережеву архітектуру для кількох дослідницьких підгруп, які працювали паралельно над різними мовними моделями [31]. Кожна команда мала окремий GCP-проєкт, в якому розгорталися обчислювальні ресурси, серед яких Tensor Processing Units (TPU) та інстанси з великим обсягом пам’яті.

Використання TPU у GCP дозволило значно пришвидшити тренування великих мовних моделей, що особливо важливо для задач, пов’язаних із трансформерами типу GPT. Інтеграція з Vertex AI Experiments надала дослідникам інструменти для автоматичного логування, порівняння конфігурацій та візуалізації метрик, що сприяло глибшому аналізу продуктивності моделей.

Всі проєкти інтегровані з хост-проєктом через механізм спільної віртуальної мережі, що дозволило централізовано управляти правилами маршрутизації та безпеки, уникаючи дублювання конфігурацій [32]. Це забезпечило узгоджене середовище для тестування різних моделей GPT-типу та порівняння їхньої продуктивності в однакових мережевих умовах.

Для відтворюваності результатів усі конфігурації зберігалися як код через Terraform, а експерименти автоматично логувалися за допомогою Cloud Logging та Vertex AI Experiments, що дало змогу порівнювати продуктивність моделей у різних конфігураціях апаратного забезпечення.

2.6.4 Порівняння ключових безпекових підходів та практик AWS, Azure та GCP

Узагальнюючи наведені у розділі приклади, можна стверджувати, що хмарні сервіси не просто доповнюють наукову інфраструктуру, а трансформують її. У кожній із платформ – AWS, Azure і GCP – реалізовано унікальні механізми, які дозволяють адаптувати інфраструктуру до специфіки дослідницьких задач.

Водночас саме грамотне налаштування, з урахуванням принципів безпеки, автоматизації та відтворюваності, перетворює хмару на повноцінний науковий інструмент. Нехтування цими принципами може призвести до втрати даних, порушення етичних норм обробки чутливої інформації та компрометації результатів. Тому хмарна інфраструктура потребує не лише технічного опанування, а й методологічного осмислення як частини наукової етики та практики.

У табл. 2.1 наведено узагальнені ключові безпекові підходи та практики у розглянутих трьох провідних хмарних платформ у розрізі категорій, що мають вирішальне значення для побудови безпечної та відтворюваної інфраструктури за допомогою Terraform у мультихмарному середовищі. Порівняння дозволяє виявити як спільні стратегічні підходи, так і відмінності в реалізації конкретних механізмів, що мають критичне значення в наукових та інженерних обчисленнях.

Таблиця 2.1 – Порівняння безпекових патернів AWS, Azure та GCP

Категорія	AWS	Asure	GCP
Найменші привілеї	IAM Role Vending Machine (RVM)	Privileged Identity Management (PIM)	Custom IAM Policies + Time-bound Access
Логування та аудит	IAM Access Analyzer, CloudTrail	Activity Logs, PIM Audit	VPC Flow Logs, Cloud Audit Logs
Ізоляція середовищ	Окремі VPC, окремі акаунти	Окремі підписки, групи ресурсів	Shared VPC + окремі проєкти
Управління змінами	CI/CD з GitHub Actions + Checkov	GitOps + ARM Templates	Terraform + Cloud Build
Моделювання загроз	Placement-aware EC2, network ACLs	WAF, NSG, segmentation	Co-location analysis, TPU isolation
Регуляторна відповідність	KMS, region control, data lifecycle	Encryption at rest/in transit, GDPR zones	Data residency, DLP API
Економічна ефективність	Auto-stop EC2, spot instances	Budget alerts, cost analysis	Preemptible VMs, region-aware planning

У категорії *найменших привілеїв* AWS демонструє високий рівень автоматизації, завдяки реалізації патерна Role Vending Machine, який забезпечує контрольований процес створення IAM-ролей через pull request-процеси. Azure натомість пропонує Privileged Identity Management як сервіс із вбудованою тимчасовою ескалацією прав доступу, що дозволяє обмежити постійне використання адміністративних ролей. GCP реалізує подібну концепцію через кастомізовані IAM-політики з можливістю тимчасового надання доступу, часто інтегровані з TTL-механізмами, наприклад через Cloud Scheduler або сторонні інтеграції.

У сфері *логування та аудиту* всі три платформи надають інструменти для детального моніторингу змін і дій користувачів, але з відмінностями в інтеграції та гнучкості. AWS поєднує IAM Access Analyzer з CloudTrail для детекції надмірних прав та історії доступу. Azure забезпечує аудит через Activity Logs і PIM Audit, з фокусом на привілейовані дії. GCP пропонує Cloud Audit Logs разом із VPC Flow Logs, що робить акцент як на дії користувачів, так і на мережевих з'єднаннях.

Ізоляція середовищ реалізується через архітектурні механізми розмежування: AWS використовує окремі облікові записи та VPC, Azure – підписки й групи ресурсів, тоді як GCP робить ставку на поєднання окремих проєктів із Shared VPC для централізованого управління мережею. Кожен із підходів має

свої переваги в масштабованості та адмініструванні – зокрема, підхід GCP дозволяє зменшити дублювання мережевої конфігурації без втрати ізоляції.

Щодо *управління змінами*, то всі три платформи підтримують інфраструктурний підхід з використанням Terraform, однак мають різні точки інтеграції з CI/CD. AWS застосовує GitHub Actions у поєднанні з інструментами статичного аналізу (наприклад, Checkov), що дозволяє виявляти потенційні проблеми в конфігураціях ще до їхнього застосування. Azure традиційно використовує ARM-шаблони та GitOps-підходи з прив'язкою до Azure DevOps. GCP орієнтується на поєднання Terraform з Cloud Build, що забезпечує нативну інтеграцію з іншими сервісами GCP у пайплайні.

У контексті *моделювання загроз* кожна платформа реалізує власні механізми попередження ризиків: AWS враховує розміщення ресурсів у хостах (placement), мережеві ACL і фізичну ізоляцію в EC2; Azure фокусується на використанні WAF, мережевих груп безпеки (NSG) і сегментації, а GCP надає розширені можливості аналізу ко-локації процесів (особливо для TPU) і підтримку зонованої ізоляції. Це набуває особливої важливості у дослідницьких сценаріях, де є ризик атак побічними каналами в мульти-тенантних середовищах.

У частині *регуляторної відповідності* всі три платформи підтримують вимоги до захисту даних як у спокої, так і в русі. AWS забезпечує контроль регіонів зберігання та гнучке управління життєвим циклом даних у поєднанні з KMS. Azure надає механізми шифрування та підтримку збереження даних у зонах, що відповідають вимогам GDPR. GCP дозволяє реалізувати політику data residency та надає інструменти на кшталт DLP API для захисту конфіденційної інформації.

Економічна ефективність управління інфраструктурою також варіюється залежно від платформи. AWS пропонує автозупинку EC2 і використання spot-інстансів для зниження витрат. Azure реалізує моніторинг бюджету та аналітику витрат як частину Azure Cost Management. GCP, своєю чергою, оптимізує витрати за рахунок preemptible VM і планування розміщення ресурсів у найвигідніших регіонах.

Тим самим аналіз показує, що кожна з хмарних платформ має власні сильні сторони та підходи до вирішення спільних викликів безпеки, що формують ґрунт для застосування Terraform у мультихмарному науковому середовищі. Обґрунтований вибір залежить від вимог до ізоляції, регуляторної відповідності, масштабу експериментів і операційної стійкості.

2.7 Висновки до розділу

Розділ продемонстрував, що Terraform є потужним інструментом управління хмарною інфраструктурою у мультихмарних середовищах. Архітектура Terraform базується на декларативному підході, який забезпечує відтворюваність, масштабованість і інтеграцію з процесами автоматизації. Розгляд провайдерів, ресурсів, модулів і життєвого циклу підтвердив, що Terraform надає гнучкі засоби для опису інфраструктури як коду, дозволяє зменшити кількість ручних втручань і помилок.

Особливе значення має зберігання стану інфраструктури у terraform.tfstate – критичному компоненті, який потребує надійного захисту та централізованого управління, особливо в умовах командної роботи. Практичні приклади впровадження Terraform у AWS, Azure та GCP продемонстрували різні підходи до безпеки, розмежування доступу, ізоляції середовищ і централізованого управління мережею, зокрема через механізми RVM, PIM і Shared VPC відповідно.

Загалом Terraform є не лише засобом конфігураційного керування, а й ключовим інструментом побудови безпечної, керованої, відтворюваної інфраструктури для хмарно-орієнтованих наукових обчислень. Його використання сприяє підвищенню надійності експериментальних платформ, ефективному розподілу ресурсів і дотриманню принципів інженерної строгості в цифровому науковому середовищі.

3 РЕАЛІЗАЦІЯ МУЛЬТИХМАРНОЇ ІНФРАСТРУКТУРИ НА ОСНОВІ TERRAFORM

3.1 Побудова шаблонів для AWS, Azure, GCP

У процесі проєктування мультихмарної інфраструктури одним із ключових завдань є створення узгоджених шаблонів конфігурації, які забезпечують повторюваність, масштабованість та безпеку середовищ у різних хмарних платформах. Попри наявність спільних концепцій IaC, кожна платформа – Amazon Web Services (AWS), Microsoft Azure та Google Cloud Platform (GCP) – має власні особливості, які впливають на структуру шаблонів, логіку доступу до ресурсів та інтеграцію з DevOps-процесами.

3.1.1 AWS-шаблони за допомогою Terraform

AWS-шаблони найчастіше реалізуються за допомогою Terraform або CloudFormation [33]. Розглянемо декілька прикладів можливих Terraform-шаблонів для AWS: як для умов типової DevOps-практики в IT-компанії в Україні, так і шаблон з модулем для GitHub Actions і адаптований до staging/production середовища з окремими VPC і Terraform workspace.

Приклад 1. Terraform-шаблон для AWS, адаптований до умов типової DevOps-практики в IT-компанії в Україні може поєднувати: EC2-сервер для backend-розгортання; S3-бакет для зберігання артефактів; IAM-роль з політикою найменших привілеїв; CloudWatch для логування; Security Group з обмеженим доступом; інтеграцію з CI/CD через GitHub Actions (опосередковано).

Структура такого Terraform-шаблона (файл main.tf):

```
provider "aws" {
  region = "eu-central-1" # популярний регіон для українських компаній
}

# S3 для артефактів
resource "aws_s3_bucket" "artifacts" {
  bucket = "company-artifacts-${random_id.suffix.hex}"
  acl    = "private"

  tags = {
```

```
    Environment = "dev"
    Project     = "backend"
  }
}

resource "random_id" "suffix" {
  byte_length = 4
}

# IAM роль для EC2
resource "aws_iam_role" "ec2_role" {
  name = "ec2-backend-role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Effect = "Allow"
      Principal = {
        Service = "ec2.amazonaws.com"
      }
      Action = "sts:AssumeRole"
    }]
  })
}

# Політика доступу до S3 та CloudWatch
resource "aws_iam_policy" "ec2_policy" {
  name        = "ec2-backend-policy"
  description = "Minimal access to S3 and CloudWatch"

  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Effect = "Allow"
        Action = [
          "s3:GetObject",
          "s3:PutObject"
        ]
        Resource = "${aws_s3_bucket.artifacts.arn}/*"
      },
      {
        Effect = "Allow"
        Action = [
          "logs:CreateLogStream",
          "logs:PutLogEvents"
        ]
        Resource = "*"
      }
    ]
  })
}

resource "aws_iam_role_policy_attachment" "attach_policy" {
  role       = aws_iam_role.ec2_role.name
  policy_arn = aws_iam_policy.ec2_policy.arn
}
```

```
# Security Group з доступом лише з офісу
resource "aws_security_group" "backend_sg" {
  name        = "backend-sg"
  description = "Allow SSH and HTTP from office IP"
  vpc_id      = "vpc-xxxxxxxx" # замінити на ваш VPC

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["xx.xx.xx.xx/32"] # IP офісу
  }

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["xx.xx.xx.xx/32"]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

# EC2 інстанс
resource "aws_instance" "backend" {
  ami           = "ami-0c55b159cbfafe1f0" # Amazon Linux 2
  instance_type = "t3.micro"
  key_name      = "company-key"
  security_groups = [aws_security_group.backend_sg.name]
  iam_instance_profile = aws_iam_instance_profile.ec2_profile.name

  tags = {
    Name        = "BackendServer"
    Environment = "dev"
  }
}

resource "aws_iam_instance_profile" "ec2_profile" {
  name = "ec2-backend-profile"
  role = aws_iam_role.ec2_role.name
}
```

Цей Terraform-шаблон [33] був адаптований до умов роботи в Україні. Він поєднує безпеку, відтворюваність, масштабованість та відповідність регуляторним вимогам. Його архітектура враховує реальні виклики, з якими стикаються DevOps-команди при розгортанні хмарної інфраструктури в мультихмарному середовищі, зокрема в AWS. Регіон Frankfurt (eu-central-1) є стратегічно доцільним для українських компаній, позаяк він забезпечує мінімальні затримки у ме-

режевій взаємодії, географічну близькість до основних офісів, а також повну відповідність вимогам GDPR, що особливо важливо при роботі з персональними або чутливими даними. Вибір цього регіону дозволяє уникнути юридичних ризиків, пов'язаних з обробкою даних у позаєвропейських юрисдикціях, і водночас гарантує стабільність та високу доступність сервісів AWS. З точки зору безпеки, шаблон реалізує принцип найменших привілеїв через IAM-роль, яка має строго обмежений набір дозволів: лише доступ до S3 для зберігання артефактів та до CloudWatch для логування. Такий підхід виключає можливість ненавмисного або шкідливого втручання в інші ресурси, знижує ризик ескалації прав і спрощує аудит. Роль не має доступу до критичних сервісів, як-от: IAM, EC2 або RDS, що відповідає сучасним практикам DevSecOps і дозволяє інтегрувати шаблон у середовище з підвищеними вимогами до безпеки.

Ізоляція середовища наведеного шаблона реалізована через обмеження доступу до EC2-інстансу за IP-адресою офісу. Це дозволяє уникнути відкритого доступу з інтернету, зменшує площину атаки та забезпечує контрольоване середовище для розгортання backend-сервісів. Така модель особливо ефективна для невеликих команд, які працюють з внутрішніми сервісами або тестовими середовищами, і не потребують глобального доступу. Відтворюваність інфраструктури досягається завдяки декларативному опису всіх ресурсів у Terraform. Кожен компонент – від VPC до IAM-політики – визначений у коді, що дозволяє зберігати його у системі контролю версій, інтегрувати в CI/CD-пайплайн та забезпечити повну прозорість змін. Це критично важливо для наукових або фінансових проєктів, де необхідно гарантувати, що середовище розгортання є ідентичним до тестового або попереднього. Крім того, така структура дозволяє легко масштабувати рішення, створюючи окремі workspace для staging і production з чітким розмежуванням конфігурацій.

Масштабованість шаблона проявляється у його модульності. Архітектура дозволяє безболісно додати Auto Scaling для EC2, базу даних RDS для зберігання транзакцій, балансувальник навантаження ALB для розподілу трафіку або файлову систему EFS для спільного доступу до даних. Завдяки цьому шаблон

може еволюціонувати від простого тестового середовища до повноцінної продуктивної інфраструктури, що відповідає вимогам сучасного IT-бізнесу.

Отже розглянутий шаблон є не просто технічним артефактом, а інженерним рішенням, яке враховує реальні потреби української IT-компанії: відповідність європейським нормам, безпечне розгортання, контрольований доступ, автоматизацію процесів та готовність до масштабування. Його використання дозволяє скоротити час на налаштування середовищ, зменшити ризики та забезпечити стабільність у DevOps-процесах.

Приклад 2. Розширений сценарій, який включає: Terraform-модулі для staging і production середовищ з окремими VPC; GitHub Actions workflow для автоматичного застосування Terraform; IAM-роль для CI/CD з обмеженими правами; Workspace-структуру для розділення середовищ.

Структура такого проєкту показана на рис. 3.1.

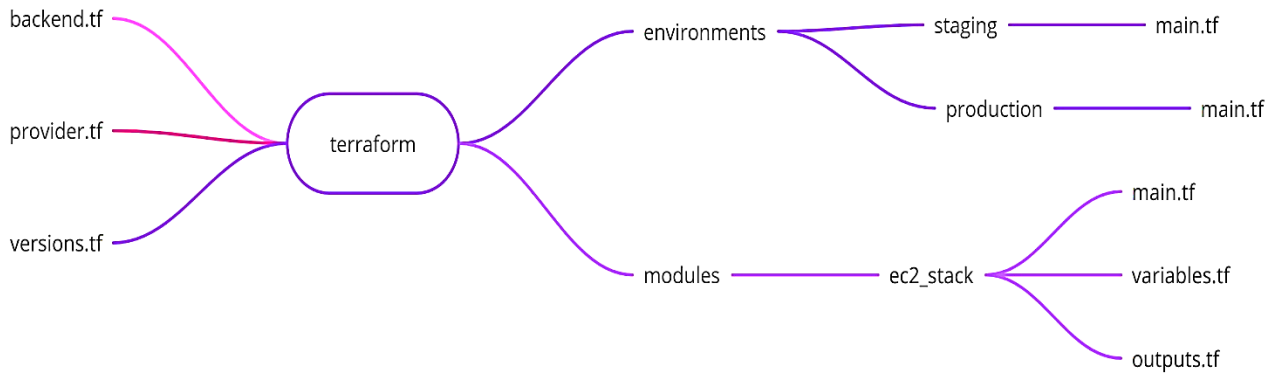

Рисунок 3.1 – Логічна структура Terraform-проєкту для AWS з модульною організацією та середовищами

Шаблон із модульною структурою, що поєднує окремі середовища staging і production, є прикладом практичного проєктування хмарної інфраструктури з використанням Terraform. Його архітектура дозволяє досягти високого рівня ізоляції, повторюваності, масштабованості та контролю над змінами, що важливо для IT-компаній, які працюють у мультихмарному або регульованому середовищі.

Ключовою особливістю шаблона є розділення на модулі та середовища. Модуль ec2_stack містить основну логіку створення інфраструктури – EC2-інстанс, VPC, S3-бакет, IAM-роль – і є універсальним блоком, який можна повторно використовувати в різних контекстах. Такий підхід дозволяє уникнути дублювання коду, спрощує тестування змін і забезпечує консистентність конфігурацій. Водночас середовища staging і production мають власні main.tf файли, які викликають модуль з різними параметрами, що дозволяє гнучко керувати ресурсами залежно від цілей: staging – для тестування, production – для реального навантаження.

Файл backend.tf визначає конфігурацію для зберігання Terraform state, зазвичай у S3-бакеті з блокуванням через DynamoDB. Це забезпечує централізоване зберігання стану інфраструктури, запобігає конфліктам при паралельному доступі та дозволяє легко відстежувати історію змін. Файл provider.tf задає регіон та облікові дані, а versions.tf – контроль версій Terraform та провайдерів, що критично для стабільності пайплайнів.

Перевага цього шаблона полягає в тому, що він повністю відповідає принципам GitOps: всі зміни в інфраструктурі здійснюються через pull request, зберігаються в Git, проходять рев'ю та автоматично застосовуються через CI/CD. Це дозволяє досягти високої прозорості, відтворюваності та відповідності вимогам безпеки. Крім того, модульна структура спрощує масштабування: додавання Auto Scaling, RDS, ALB або інших компонентів може бути реалізовано як окремі модулі, що інтегруються в існуючі середовища без порушення їх цілісності.

Ізоляція staging і production середовищ дозволяє уникнути перехресного впливу, зменшити ризики та забезпечити стабільність у продуктивному середовищі. Наприклад, staging може використовувати менш потужні інстанси, спрощені політики доступу або тестові дані, тоді як production – повноцінну конфігурацію з підвищеним рівнем безпеки, шифруванням та моніторингом.

Цей сценарій дозволяє: автоматично розгортати staging/production середовища через GitHub Actions, використовувати модульну архітектуру Terraform,

дотримуватись принципу найменших привілеїв через IAM, зберігати артефакти в S3, відтворювати середовища з повною ізоляцією через VPC.

Модуль EC2 + S3 + IAM – modules/ec2_stack/main.tf:

```
resource "aws_vpc" "main" {
  cidr_block = var.vpc_cidr
  tags = {
    Name = "${var.env}-vpc"
  }
}
resource "aws_subnet" "public" {
  vpc_id            = aws_vpc.main.id
  cidr_block        = var.subnet_cidr
  availability_zone = var.az
  tags = {
    Name = "${var.env}-subnet"
  }
}
resource "aws_instance" "app" {
  ami           = var.ami_id
  instance_type = var.instance_type
  subnet_id     = aws_subnet.public.id
  tags = {
    Name = "${var.env}-app"
  }
}
resource "aws_s3_bucket" "artifacts" {
  bucket = "${var.env}-artifacts-${random_id.suffix.hex}"
  acl    = "private"
}

resource "random_id" "suffix" {
  byte_length = 4
}
```

staging середовище – environments/staging/main.tf:

```
module "ec2_stack" {
  source        = "../../modules/ec2_stack"
  env           = "staging"
  vpc_cidr      = "10.10.0.0/16"
  subnet_cidr   = "10.10.1.0/24"
  az            = "eu-central-1a"
  ami_id        = "ami-0c55b159cbfafe1f0"
  instance_type = "t3.micro"
}
```

production середовище – environments/production/main.tf:

```
module "ec2_stack" {
  source        = "../../modules/ec2_stack"
  env           = "production"
  vpc_cidr      = "10.20.0.0/16"
  subnet_cidr   = "10.20.1.0/24"
  az            = "eu-central-1b"
  ami_id        = "ami-0c55b159cbfafe1f0"
  instance_type = "t3.medium"
```

```
}
```

IAM роль для GitHub Actions:

```
resource "aws_iam_role" "github_ci" {
  name = "GitHubTerraformCI"

  assume_role_policy = jsonencode({
    Version = "2012-10-17" # дата вказує на формат політики
    Statement = [{
      Effect = "Allow"
      Principal = {
        Federated            =            "arn:aws:iam::ACCOUNT_ID:oidc-
provider/token.actions.githubusercontent.com"
      }
      Action = "sts:AssumeRoleWithWebIdentity"
      Condition = {
        StringEquals = {
          "token.actions.githubusercontent.com:sub" = "repo:org/repo:*"
        }
      }
    }]
  })
}
resource "aws_iam_policy" "terraform_ci_policy" {
  name = "TerraformCIPolicy"
  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Sid    = "S3TerraformStateAccess",
        Effect = "Allow"
        Action = [
          "ec2:DescribeInstances",
          "s3:GetObject",
          "s3:PutObject",
          "s3:DeleteObject",
          "s3:ListBucket",
          "iam:PassRole",
          "iam:GetRole",
          "iam:CreateRole",
          "iam:AttachRolePolicy"
        ]
        Resource = [
          "arn:aws:s3:::terraform-state-bucket",
          "arn:aws:s3:::terraform-state-bucket/*"
        ]
      },
      {
        Sid    = "DynamoDBLockAccess",
        Effect = "Allow",
        Action = [
          "dynamodb:GetItem",
          "dynamodb:PutItem",
          "dynamodb:DeleteItem"
       ],
        Resource = "arn:aws:dynamodb:eu-central-1:ACCOUNT_ID:table/terraform-
lock-table"
```

```
      },
      {
        Sid    = "PassSpecificRole",
        Effect = "Allow",
        Action = "iam:PassRole",
        Resource = "arn:aws:iam::ACCOUNT_ID:role/ec2-backend-role"
      }
    ]
  })
}

resource "aws_iam_role_policy_attachment" "ci_attach" {
  role       = aws_iam_role.github_ci.name
  policy_arn = aws_iam_policy.terraform_ci_policy.arn
}
```

GitHub Actions CI/CD – .github/workflows/terraform.yml:

```
name: Terraform Deploy

on:
  push:
    branches:
      - main
      - staging

jobs:
  terraform:
    name: Apply Terraform
    runs-on: ubuntu-latest
    permissions:
      id-token: write
      contents: read

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v2

      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v2
        with:
          role-to-assume: arn:aws:iam::ACCOUNT_ID:role/GitHubTerraformCI
          aws-region: eu-central-1

      - name: Terraform Init
        run: terraform -chdir=terraform/environments/${{ github.ref_name }} init

      - name: Terraform Plan
        run: terraform -chdir=terraform/environments/${{ github.ref_name
}} plan

      - name: Terraform Apply
        run: terraform -chdir=terraform/environments/${{ github.ref_name
}} apply -auto-approve
```

Загалом, цей шаблон демонструє відповідність сучасним практикам DevSecOps, що робить його придатним для використання в IT-компаніях, які прагнуть до автоматизації, контролю та відповідності регуляторним нормам. Його структура дозволяє ефективно керувати життєвим циклом інфраструктури, забезпечуючи стабільність, гнучкість і готовність до масштабування.

Хоча Terraform і забезпечує кросплатформність, у практичних кейсах можуть виникати труднощі з підтримкою складних шаблонів, особливо при використанні Role Vending Machine (RVM) для автоматизованого призначення тимчасових ролей. Наприклад, при масштабуванні кількості проєктів з різними політиками IAM виникає проблема дублювання конфігурацій та складності рев'ю змін, що ускладнює аудит і підвищує ризик надмірного доступу. Крім того, CloudFormation має обмеження щодо модульності, що ускладнює повторне використання шаблонів у мультиакаунтних сценаріях.

3.1.2 Azure шаблони за допомогою Terraform

В Azure шаблони зазвичай створюються на основі ARM (Azure Resource Manager) або Bicep. Хоча ARM забезпечує глибоку інтеграцію з Azure DevOps та GitOps-підходами, його декларативна модель є менш гнучкою, порівняно з Terraform.

В результаті аналізу реального Terraform-проєкту для Azure типового сценарію української IT-компанії, яка розгортає staging та production середовища для вебзастосунку, з'ясовано: його шаблон має чітко визначену модульну структуру, яка відповідає сучасним практикам інфраструктурного проєктування; його загальна архітектура побудована навколо концепції розділення логіки на незалежні блоки, що дозволяє досягти високої гнучкості, повторюваності та масштабованості при розгортанні хмарних середовищ. Кожен модуль виконує окрему функцію, але водночас інтегрується в загальну систему через параметри, які передаються з середовищ staging або production. Шаблон охоплює: створення ресурсної групи, віртуальної мережі з підмережею, вебсервера (Virtual Machine),

сховища (Storage Account), рольовий доступ (Role Assignment), модульну структуру з розділенням середовищ (рис. 3.2).

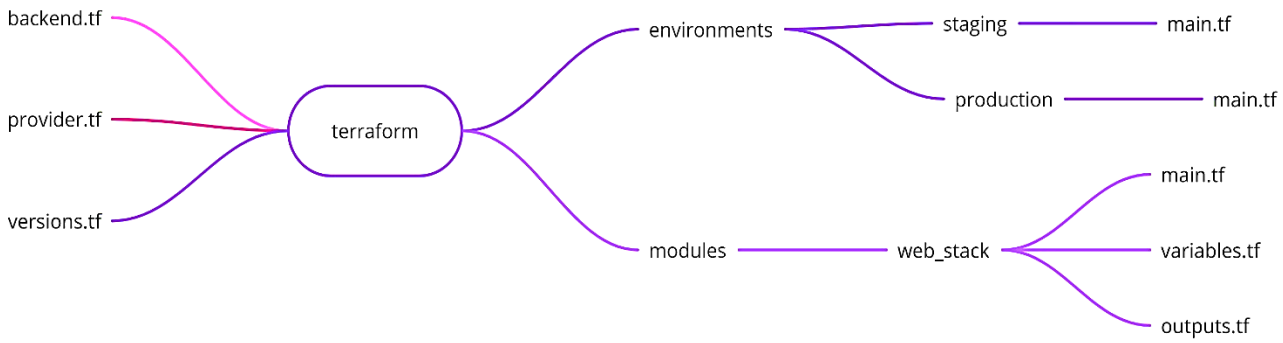

Рисунок 3.2 – Модель проєктування хмарної інфраструктури для Azure за допомогою Terraform

Модуль modules/web_stack/main.tf:

```
resource "azurerm_resource_group" "rg" {
  name     = "${var.env}-rg"
  location = var.location
}

resource "azurerm_virtual_network" "vnet" {
  name                = "${var.env}-vnet"
  address_space       = ["10.0.0.0/16"]
  location            = var.location
  resource_group_name = azurerm_resource_group.rg.name
}

resource "azurerm_subnet" "subnet" {
  name                 = "${var.env}-subnet"
  resource_group_name  = azurerm_resource_group.rg.name
  virtual_network_name = azurerm_virtual_network.vnet.name
  address_prefixes     = ["10.0.1.0/24"]
}

resource "azurerm_network_interface" "nic" {
  name                = "${var.env}-nic"
  location            = var.location
  resource_group_name = azurerm_resource_group.rg.name

  ip_configuration {
    name                          = "internal"
    subnet_id                     = azurerm_subnet.subnet.id
    private_ip_address_allocation = "Dynamic"
  }
}
resource "azurerm_linux_virtual_machine" "vm" {
  name                = "${var.env}-vm"
  location            = var.location
  resource_group_name = azurerm_resource_group.rg.name
```

```
  size                = "Standard_B1s"
  admin_username      = "azureuser"
  network_interface_ids = [azurerm_network_interface.nic.id]
  admin_ssh_key {
    username   = "azureuser"
    public_key = file(var.ssh_public_key_path)
  }
  os_disk {
    caching              = "ReadWrite"
    storage_account_type = "Standard_LRS"
  }
  source_image_reference {
    publisher = "Canonical"
    offer     = "UbuntuServer"
    sku       = "18.04-LTS"
    version   = "latest"
  }
}

resource "azurerm_storage_account" "storage" {
  name                     = "${var.env}storage${random_id.suffix.hex}"
  resource_group_name      = azurerm_resource_group.rg.name
  location                 = var.location
  account_tier             = "Standard"
  account_replication_type = "LRS"
}

resource "random_id" "suffix" {
  byte_length = 4
}
modules/web_stack/variables.tf:
variable "env" {
  type = string
}

variable "location" {
  type    = string
  default = "westeurope"
}

variable "ssh_public_key_path" {
  type = string
}
```

Модуль для створення вебінфраструктури містить компоненти для побудови ресурсної групи, віртуальної мережі, підмережі, мережевого інтерфейсу, віртуальної машини та сховища. Він реалізує базову логіку розгортання серверного середовища, яке може використовуватись для тестування або продуктивної роботи. Важливо, що всі параметри – назви, регіони, типи ресурсів – задаються через змінні, що дозволяє легко адаптувати модуль до різних сценаріїв без зміни внутрішнього коду. Такий підхід забезпечує повторюваність і контроль версій, що критично для команд, які працюють над кількома проєктами одночасно.

Модуль environments/staging/main.tf:

```
module "web_stack" {
  source              = "../../modules/web_stack"
  env                 = "staging"
  ssh_public_key_path = "~/.ssh/id_rsa.pub"
}
```

Модуль environments/production/main.tf:

```
module "web_stack" {
  source              = "../../modules/web_stack"
  env                 = "production"
  ssh_public_key_path = "~/.ssh/id_rsa.pub"
}
```

Середовища staging і production реалізовані як окремі конфігураційні файли, які викликають основний модуль з різними значеннями змінних. Це дозволяє створювати ізольовані середовища з різними параметрами, наприклад, staging може використовувати менш потужні машини, а production – більш захищені та масштабовані ресурси. Така ізоляція дозволяє уникнути конфліктів, зменшити ризики та забезпечити стабільність продуктивного середовища.

Файл provider.tf:

```
provider "azurerm" {
  features {}
}
```

Файл versions.tf:

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

  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~> 3.0"
    }
  }
}
```

Файл backend.tf (опціонально, для зберігання state в Azure Storage):

```
terraform {
  backend "azurerm" {
    resource_group_name  = "tfstate-rg"
    storage_account_name = "tfstateaccount"
    container_name       = "tfstate"
    key                  = "terraform.tfstate"
  }
}
```

Файл з налаштуванням провайдера визначає використання Azure як платформи, а файл з версіями – контроль сумісності Terraform та його компонентів. Це важливо для забезпечення стабільності при оновленнях, а також для інтегра-

ції з CI/CD системами. Файл backend конфігурує зберігання стану інфраструктури в Azure Storage, що дозволяє централізовано керувати змінами, уникати конфліктів при паралельному доступі та зберігати історію змін.

Додатково, шаблон доповнено роллю для автентифікації GitHub Actions через відкритий протокол, що дозволяє безпечно запускати автоматизовані процеси без збереження секретів. Це рішення відповідає сучасним вимогам до безпеки та автоматизації, особливо в умовах мультикомандної роботи. CI/CD workflow, реалізований через GitHub Actions, забезпечує автоматичне застосування змін при оновленні коду, що дозволяє досягти високої швидкості розгортання та зменшити кількість ручних помилок.

Щоб GitHub Actions могли автентифікуватися в Azure, потрібно створити Azure AD App Registration з Federated Credential, прив'язаний до репозиторію.

Terraform-фрагмент:

```
resource "azuread_application" "github_ci_app" {
  display_name = "GitHubTerraformCI"
}

resource "azuread_service_principal" "github_ci_sp" {
  application_id = azuread_application.github_ci_app.application_id
}

resource              "azuread_application_federated_identity_credential"
"github_oidc" {
  application_object_id = azuread_application.github_ci_app.object_id
  display_name          = "GitHubOIDC"
  issuer                = "https://token.actions.githubusercontent.com"
  subject               = "repo:your-org/your-repo:ref:refs/heads/main"
  audiences             = ["api://AzureADTokenExchange"]
}
```

Після створення цієї ролі, потрібно вручну призначити їй роль Contributor або Owner на потрібному рівні (наприклад, на ресурсну групу).

Долучення компонента CI/CD Workflow через GitHub Actions автоматично ініціалізуватиме, плануватиме та застосовуватиме Terraform-конфігурацію при пуші в main або staging:

```
name: Terraform Deploy

on:
  push:
    branches:
      - main
      - staging
```

```
permissions:
  id-token: write
  contents: read

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout
        uses: actions/checkout@v3

      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v2

      - name: Login to Azure via OIDC
        uses: azure/login@v1
        with:
          client-id: ${{ secrets.AZURE_CLIENT_ID }}
          tenant-id: ${{ secrets.AZURE_TENANT_ID }}
          subscription-id: ${{ secrets.AZURE_SUBSCRIPTION_ID }}

      - name: Terraform Init
        run: terraform -chdir=terraform/environments/${{ github.ref_name
}} init

      - name: Terraform Plan
        run: terraform -chdir=terraform/environments/${{ github.ref_name
}} plan

      - name: Terraform Apply
        run: terraform -chdir=terraform/environments/${{ github.ref_name
}} apply -auto-approve
```

Значення AZURE_CLIENT_ID, AZURE_SUBSCRIPTION_ID, AZURE_TENANT_ID можна отримати після створення Azure AD App і додати як GitHub Secrets.

Інтеграція з системою моніторингу Azure Monitor через Log Analytics Workspace дозволяє отримувати метрики та логи з віртуальних машин, що критично для підтримки продуктивності, виявлення проблем та дотримання вимог до спостережуваності. Це рішення дозволяє не лише контролювати технічний стан середовища, а й забезпечити відповідність внутрішнім політикам безпеки та зовнішнім регуляторним нормам.

Terraform-фрагмент:

```
resource "azurerm_log_analytics_workspace" "monitoring" {
  name                = "${var.env}-logs"
  location            = var.location
```

```
  resource_group_name = azurerm_resource_group.rg.name
  sku                 = "PerGB2018"
  retention_in_days   = 30
}

resource "azurerm_monitor_diagnostic_setting" "vm_logs" {
  name                       = "${var.env}-vm-diagnostics"
  target_resource_id         = azurerm_linux_virtual_machine.vm.id
  log_analytics_workspace_id                                    =
azurerm_log_analytics_workspace.monitoring.id

  log {
    category = "Syslog"
    enabled  = true

    retention_policy {
      enabled = false
    }
  }

  metric {
    category = "AllMetrics"
    enabled  = true

    retention_policy {
      enabled = false
    }
  }
}
```

Запропонований Terraform-шаблон для Azure дозволяє: створити окремі середовища з чіткою ізоляцією, централізовано керувати ресурсами через модулі, інтегрувати з CI/CD (наприклад, GitHub Actions або Azure DevOps), масштабувати інфраструктуру, додаючи нові модулі (наприклад, Azure Database, Application Gateway, Key Vault), забезпечити відтворюваність і контроль змін через систему контролю версій. Такий шаблон є повноцінним інструментом для безпечного, автоматизованого та спостережуваного розгортання Azure-інфраструктури, оскільки поєднує модульність, автоматизацію, безпеку та спостережуваність. Він придатний для використання як у невеликих командах, так і в масштабних проєктах, де важлива стабільність, контроль і швидкість розгортання.

З іншого боку, важливо враховувати, що в реальних проєктах часто виникає проблема надмірної складності шаблонів, особливо при реалізації багатозонної архітектури з Privileged Identity Management (PIM), що потребує точного налаштування умов активації ролей. Неправильна конфігурація може призвести

до ситуацій, коли адміністративні права залишаються активними поза межами необхідного часу, що суперечить принципу найменших привілеїв.

3.1.3 GCP-шаблони за допомогою Terraform

Розглянемо приклад Terraform-шаблона для GCP, побудований за тією самою логікою, що й шаблони для AWS та Azure. Він реалізує модульну архітектуру з розділенням середовищ, автоматизованим розгортанням базової інфраструктури, інтеграцією з CI/CD та моніторингом (рис. 3.3). Такий шаблон дозволяє здійснити комплексне порівняння архітектурних підходів у трьох хмарних платформах.

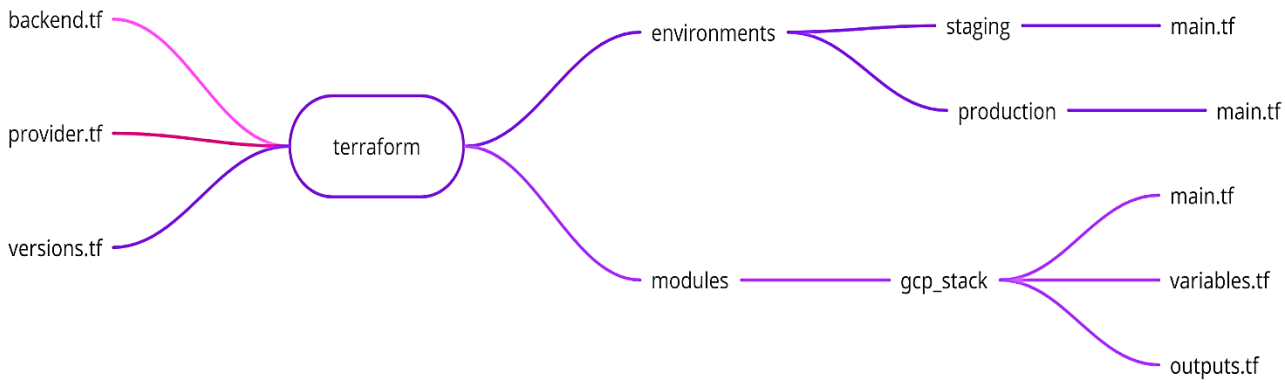

Рисунок 3.3 – Структура Terraform-проєкту для GCP з модульною організацією та середовищами

Файл modules/gcp_stack/main.tf:

```
resource "google_project" "project" {
  name       = "${var.env}-project"
  project_id = "${var.env}-infra-${random_id.suffix.hex}"
  org_id     = var.org_id
  billing_account = var.billing_account
}

resource "google_compute_network" "vpc" {
  name                    = "${var.env}-vpc"
  auto_create_subnetworks = false
  project                 = google_project.project.project_id
}

resource "google_compute_subnetwork" "subnet" {
  name          = "${var.env}-subnet"
  ip_cidr_range = "10.0.1.0/24"
  region        = var.region
  network       = google_compute_network.vpc.id
  project       = google_project.project.project_id
```

```
}

resource "google_compute_instance" "vm" {
  name         = "${var.env}-vm"
  machine_type = "e2-medium"
  zone         = var.zone
  project      = google_project.project.project_id

  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-11"
    }
  }

  network_interface {
    subnetwork = google_compute_subnetwork.subnet.id
    access_config {}
  }

  metadata = {
    ssh-keys = "terraform:${file(var.ssh_public_key_path)}"
  }
}

resource "google_storage_bucket" "artifacts" {
  name     = "${var.env}-artifacts-${random_id.suffix.hex}"
  location = var.region
  project  = google_project.project.project_id
}

resource "random_id" "suffix" {
  byte_length = 4
}
```

Модуль modules/gcp_stack/variables.tf:

```
variable "env" {
  type = string
}

variable "region" {
  type    = string
  default = "europe-west3" # Франкфурт
}

variable "zone" {
  type    = string
  default = "europe-west3-a"
}

variable "org_id" {
  type = string
}

variable "billing_account" {
  type = string
}
```

```
variable "ssh_public_key_path" {
  type = string
}
```

Модуль environments/staging/main.tf:

```
module "gcp_stack" {
  source              = "../../modules/gcp_stack"
  env                 = "staging"
  org_id              = "YOUR_ORG_ID"
  billing_account     = "YOUR_BILLING_ACCOUNT_ID"
  ssh_public_key_path = "~/.ssh/id_rsa.pub"
}
```

Модуль environments/production/main.tf:

```
module "gcp_stack" {
  source              = "../../modules/gcp_stack"
  env                 = "production"
  org_id              = "YOUR_ORG_ID"
  billing_account     = "YOUR_BILLING_ACCOUNT_ID"
  ssh_public_key_path = "~/.ssh/id_rsa.pub"
}
```

provider.tf:

```
provider "google" {
  project = "default-project"
  region  = "europe-west3"
  zone    = "europe-west3-a"
}
```

versions.tf:

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

  required_providers {
    google = {
      source  = "hashicorp/google"
      version = "~> 5.0"
    }
  }
}
```

Backend.tf для зберігання стану в GCS:

```
terraform {
  backend "gcs" {
    bucket  = "terraform-state-bucket"
    prefix  = "gcp/${terraform.workspace}"
  }
}
```

CI/CD інтеграція через GitHub Actions:

```
name: Terraform GCP Deploy

on:
  push:
    branches:
      - main
      - staging
```

```
permissions:
  contents: read
  id-token: write

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout
        uses: actions/checkout@v3

      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v2

      - name: Authenticate to GCP
        uses: google-github-actions/auth@v1
        with:
          workload_identity_provider:
"projects/PROJECT_NUMBER/locations/global/workloadIdentityPools/POOL_ID/
providers/PROVIDER_ID"
          service_account:                                    "terraform-
ci@PROJECT_ID.iam.gserviceaccount.com"

      - name: Terraform Init
        run: terraform -chdir=terraform/environments/${{ github.ref_name
}} init

      - name: Terraform Plan
        run: terraform -chdir=terraform/environments/${{ github.ref_name
}} plan

      - name: Terraform Apply
        run: terraform -chdir=terraform/environments/${{ github.ref_name
}} apply -auto-approve
```

Інтеграція з моніторингом через Cloud Monitoring:

```
resource "google_monitoring_dashboard" "vm_dashboard" {
  dashboard_json = file("${path.module}/dashboard.json")
  project        = google_project.project.project_id
}
```

Тут JSON-файл містить конфігурацію графіків для CPU, пам’яті, мережі тощо.

Запропонований шаблон реалізує повноцінну інфраструктуру з розділенням середовищ, централізованим зберіганням стану, безпечним CI/CD доступом через OIDC, та інтеграцією з системою моніторингу. Він відповідає архітектурним принципам GCP: використання проєктів як ізоляційних одиниць, прив’язка до організації та білінгу, декларативне управління ресурсами, а також підтримка Workload Identity для безпечної автентифікації GitHub Actions.

Особливістю GCP є використання окремих проєктів у поєднанні з Shared VPC, що дозволяє централізовано керувати мережею. Проте при побудові шаблонів для задач глибокого навчання з використанням TPU та Vertex AI виникають труднощі з узгодженням прав доступу між сервісними агентами, особливо в умовах TTL-доступу. У деяких випадках обмеження на делегування прав або нестабільна підтримка певних ресурсів у Terraform-провайдері GCP призводять до потреби ручного втручання, що порушує принцип автоматизації.

3.1.4 Порівняння специфіки шаблонів проєктів в AWS, Azure і GCP

Структура Terraform-проєктів для AWS і Azure має спільну логіку – модульність, розділення середовищ, централізоване управління станом – але відрізняється у деталях, які зумовлені специфікою кожної платформи, її сервісною моделлю, політиками доступу та архітектурними особливостями.

У шаблоні для AWS основний акцент зроблено на управлінні ролями, політиками доступу та ізоляцією середовищ через окремі акаунти або VPC. Важливу роль відіграє IAM, який дозволяє гнучко делегувати права через ролі, що використовуються EC2, CI/CD або сервісами. Шаблон містить чітко визначену роль для GitHub Actions з обмеженим доступом до S3 та CloudWatch, що відповідає принципу найменших привілеїв. Всі ресурси – EC2, S3, IAM – описані в модулі, який викликається з середовищ staging і production, кожне з яких має власну конфігурацію. Важливо, що AWS дозволяє глибоку інтеграцію з GitHub через OIDC, що реалізовано в шаблоні через роль з trust policy. Інфраструктура зберігає свій стан у S3-бакеті з блокуванням через DynamoDB, що забезпечує контроль змін і запобігає конфліктам.

У шаблоні для Azure структура також модульна, але акцент зміщено на ресурсну групу як базову одиницю ізоляції. Всі ресурси – віртуальна машина, мережа, підмережа, сховище – створюються в межах однієї ресурсної групи, що спрощує управління і відповідає типовій практиці Azure. Роль для GitHub реалізується через Azure AD Application з федеративним доступом, що потребує створення об'єкта і прив'язки до репозиторію через OIDC. На відміну від AWS,

роль не має trust policy, а автентифікація здійснюється через Azure AD з прив'язкою до конкретного суб'єкта. CI/CD workflow також інтегровано через GitHub Actions, але автентифікація відбувається через Azure Login Action з використанням ідентифікаторів, отриманих з AD App. Стан Terraform зберігається в Azure Storage, що вимагає створення контейнера, облікового запису та ресурсної групи для state-файлів.

Інтеграція з моніторингом також відрізняється. У AWS це CloudWatch, який автоматично збирає логи і метрики, а в Azure – Log Analytics Workspace, який потребує явного підключення через Diagnostic Settings. У шаблоні для Azure це реалізовано через окремий ресурс, що прив'язує віртуальну машину до workspace, дозволяючи збирати системні логи та метрики.

Загалом структура шаблона для AWS більше орієнтована на гнучке управління доступом і масштабування через ролі та політики, тоді як структура шаблона для Azure – на централізоване управління ресурсами через ресурсні групи та інтеграцію з Azure AD. Обидва шаблони підтримують розділення середовищ, автоматизацію через CI/CD, зберігання стану та спостережуваність, але реалізують ці функції відповідно до архітектурної логіки кожної платформи.

Структура Terraform-шаблона для Google Cloud Platform суттєво відрізняється від шаблонів для AWS та Azure через специфіку організаційної моделі GCP, яка базується на проєктах як основних одиницях ізоляції. У GCP кожне середовище – staging або production – створюється в окремому проєкті, що має власний білінг, політики доступу та ресурсну ієрархію. Це контрастує з AWS, де ізоляція досягається через окремі облікові записи або VPC, і з Azure, де середовища розділяються через ресурсні групи в межах одного підпису.

У шаблоні для GCP модуль не просто створює ресурси, а ініціалізує новий проєкт, що вимагає вказання ідентифікатора організації та білінгового акаунта. Це означає, що інфраструктура не лише розгортається, а й формує власний адміністративний контекст, що є унікальним для GCP. У шаблонах для AWS і Azure проєктування відбувається в межах вже існуючої організаційної структури, без створення окремих ізольованих доменів.

Ще однією відмінністю є спосіб автентифікації CI/CD. У GCP використовується Workload Identity, який дозволяє GitHub Actions автентифікуватися через сервісний акаунт без збереження секретів. Це реалізується через прив'язку до Workload Identity Pool, що є частиною IAM. У AWS аналогічна функція реалізується через IAM роль з trust policy, а в Azure – через Azure AD Application з федеративним обліковим записом. Хоча всі три платформи підтримують OIDC, реалізація в GCP є більш ізольованою і вимагає точного налаштування зв'язку між зовнішнім провайдером і внутрішнім сервісним акаунтом.

Зберігання стану Terraform у GCP також має свою специфіку. Воно здійснюється в Google Cloud Storage, де кожне середовище має власний префікс у бакеті, що дозволяє логічно розділити стани без створення окремих контейнерів, як у Azure, або DynamoDB-блокування, як у AWS [33]. Це спрощує конфігурацію, але водночас вимагає чіткого контролю над доступом до бакета, оскільки він є спільним для всіх середовищ.

Моніторинг у GCP реалізується через Cloud Monitoring, який інтегрується з віртуальними машинами через Stackdriver. У шаблоні це здійснюється через створення JSON-конфігурації для дашборду, що дозволяє візуалізувати метрики CPU, пам'яті, мережі тощо. У AWS аналогом є CloudWatch, який автоматично збирає метрики, а в Azure – Log Analytics Workspace, що потребує окремого підключення через Diagnostic Settings [20].

Загалом, структура шаблона для GCP є більш декларативною на рівні організації, вимагає створення ізольованих проєктів, має специфічну модель автентифікації та спрощене зберігання стану. Вона орієнтована на повну ізоляцію середовищ, що особливо важливо для наукових, фінансових або мультикомандних проєктів, де кожна команда працює в окремому домені з власними політиками доступу. У порівнянні з AWS і Azure, GCP надає більш чітке розмежування адміністративних меж, але вимагає глибшого розуміння організаційної структури платформи [35].

Архітектурний аналіз AWS, Azure і GCP з урахуванням типу проєкту, вимог до безпеки, масштабованості, ізоляції, інтеграції та управління наведено в табл. 3.1.

Таблиця 3.1 – Порівняльна таблиця архітектур Terraform-проєктів: AWS, Azure, GCP

Архітектурний аспект	**AWS**	**Azure**	**GCP**
Ізоляція середовищ	Через окремі VPC, облікові записи або IAM-політики	Через окремі ресурсні групи з рольовим доступом (RBAC)	Через окремі проєкти з власним білінгом і політиками доступу
Модульна структура Terraform	Модуль EC2 + S3 + IAM, викликається з staging/production	Модуль VM + VNet + Storage, викликається з staging/production	Модуль VM + VPC + Storage + Project, викликається з staging/production
Автентифікація CI/CD	IAM-роль з OIDC trust policy для GitHub Actions	Azure AD App з Federated Credential для GitHub OIDC	Workload Identity + сервісний акаунт для GitHub OIDC
Інтеграція з GitHub Actions	aws-actions/configure-aws-credentials	azure/login	google-github-actions/auth
Зберігання Terraform state	S3-бакет + DynamoDB для блокування	Azure Storage Account + Container	Google Cloud Storage (GCS) Bucket
Моніторинг і логування	CloudWatch: автоматичне збирання логів і метрик	Azure Monitor + Log Analytics + Diagnostic Settings	Cloud Monitoring + Stackdriver + Dashboard JSON
Моніторинг віртуальних машин	CloudWatch Agent або вбудовані метрики	Diagnostic Settings + Log Analytics	Stackdriver Monitoring + VM Metrics
Масштабованість інфраструктури	Auto Scaling, ALB, RDS, EFS	VMSS, Application Gateway, Azure SQL	Instance Groups, Load Balancer, Cloud SQL
Рольова модель доступу (RBAC/IAM)	IAM-ролі з гнучкими політиками доступу	RBAC через Azure AD з ролями Contributor/Owner	IAM-політики на рівні проєкту + сервісні акаунти
Інтеграція з корпоративною AD	Обмежена	Глибока інтеграція з Azure Active Directory	Обмежена
Вартість старту та адміністрування	Висока гнучкість, але складна модель білінгу	Прозора модель для корпоративних клієнтів	Найнижчий поріг входу для ізольованих команд
Придатність для мультикомандної роботи	Через IAM та окремі акаунти або VPC	Через рольову модель у межах одного підпису	Через окремі проєкти з незалежним управлінням
Сховище артефактів	S3-бакет	Azure Storage Account	GCS-бакет
Регіон, зручний для України	eu-central-1 (Франкфурт)	westeurope (Нідерланди або Франкфурт)	europe-west3 (Франкфурт)

Ця таблиця дозволяє чітко побачити та швидко оцінити сильні сторони кожної платформи залежно від архітектурних потреб Terraform-проєкту, те як кожна платформа реалізує однакові концепції – ізоляцію, автоматизацію, спостережуваність – через власні механізми. AWS надає найгнучкішу модель доступу через IAM, Azure – найглибшу інтеграцію з AD і DevOps, а GCP – найчіткіше розділення середовищ через проєкти.

Наукова новизна цього аналізу полягає в тому, що шаблони розглядаються не лише як технічні артефакти, а як інструменти забезпечення відтворюваності, ізоляції та етичної відповідності в наукових дослідженнях. Виявлені негаразди – дублювання конфігурацій, складність рев'ю, нестабільність провайдерів, надмірна складність шаблонів – мають практичне значення для DevOps-команд, які працюють у мультихмарному середовищі. Вони вказують на потребу у стандартизації шаблонів, розробці модульних бібліотек та впровадженні механізмів автоматичного аудиту, що дозволить зменшити ризики і підвищити достовірність наукових результатів.

3.1.5 Рекомендації щодо вибору хмарної платформи залежно від типу проєкту

Усі три платформи – AWS, Azure і GCP – підтримують модульну структуру Terraform-проєктів, ізоляцію середовищ, автоматизацію через CI/CD та інтеграцію з системами моніторингу [36]. Проте їх архітектурна логіка суттєво відрізняється, що впливає на зручність, безпеку та масштабованість залежно від типу проєкту.

AWS демонструє найвищу гнучкість у керуванні доступом через IAM, дозволяючи точно налаштовувати ролі, політики та trust relationships. Це робить платформу придатною для проєктів з високими вимогами до безпеки, багаторівневого доступу та складної DevSecOps-інтеграції. Водночас AWS вимагає ретельного контролю над конфігурацією, оскільки надмірні дозволи можуть призвести до ескалації прав.

Azure вирізняється глибокою інтеграцією з корпоративною інфраструктурою, зокрема через Azure Active Directory, що робить її оптимальною для проєктів, які потребують централізованого управління ідентичностями, ролями та політиками. Структура шаблона орієнтована на ресурсні групи, що спрощує адміністрування, але може обмежувати гнучкість у складних сценаріях з мультиоточенням. Azure також має найкращу інтеграцію з DevOps-пайплайнами, особливо у корпоративному середовищі.

GCP забезпечує найчіткіше розділення середовищ через окремі проєкти, що дозволяє досягти повної ізоляції, незалежного білінгу та автономного управління. Це особливо важливо для наукових, освітніх або мультикомандних проєктів, де кожна команда працює в окремому домені. GCP також має найпростіший механізм зберігання Terraform-стану та ефективну модель автентифікації через Workload Identity, що знижує ризики, пов’язані з секретами.

Усе з’ясоване дозволяє сформувати такі рекомендації за типами проєктів:

1. AWS рекомендовано використовувати для проєктів з високими вимогами до безпеки, багаторівневого доступу, DevSecOps-інтеграції та складної інфраструктури, особливо якщо потрібна гнучка рольова модель, ізоляція через VPC та контрольований CI/CD;

2. Azure оптимальним вибором буде для корпоративних проєктів, які інтегруються з внутрішніми системами, потребують централізованого управління ідентичностями, ролями та політиками, а також мають чітку структуру командної відповідальності, , особливо в поєднанні з Azure DevOps або GitHub Enterprise;

3. GCP з його проєктною моделлю Workload Identity та ефективною інтеграцією з Cloud Monitoring підійде для наукових, освітніх, мультикомандних або стартап-проєктів, де важлива повна ізоляція середовищ, простота адміністрування, незалежний білінг та швидке масштабування без складної корпоративної структури.

Але при цьому важливо розуміти, що AWS цілком може бути ефективною платформою для наукових, освітніх, мультикомандних або стартап-проєктів.

Сформульовані рекомендації не є жорсткими обмеженнями, а радше стратегічними орієнтирами, які враховують типові переваги кожної платформи в певних контекстах. AWS має потужні інструменти для ізоляції, автоматизації, масштабування та безпеки, які можуть бути надзвичайно корисними навіть у стартап-середовищі або дослідницьких проєктах, особливо якщо команда має досвід роботи з IAM, VPC, CI/CD та хоче гнучко контролювати доступ до ресурсів. Однак порівняно з GCP, AWS може вимагати більше налаштувань для досягнення повної ізоляції між командами, оскільки її модель облікових записів і політик доступу є більш деталізованою і складною. GCP, навпаки, дозволяє створювати окремі проєкти з власним білінгом і політиками, що часто спрощує адміністрування в мультикомандному середовищі. Але це не означає, що AWS не підходить, адже все залежить від конкретних потреб, технічної зрілості команди, бюджету, вимог до безпеки та інтеграції.

Тож якщо стартап або наукова група вже має досвід з AWS або потребує специфічних сервісів, як-от SageMaker для машинного навчання, або хоче скористатися грантами AWS для дослідницьких проєктів, то платформа AWS є цілком доречною. Вибір слід базувати не лише на типі проєкту, а й на стратегічних цілях, технічних компетенціях і довгостроковій моделі розвитку.

Розглянемо та проаналізуємо приклад вибору хмарної платформи для науково-освітнього проєкту з кількох команд розробників, кожна з яких працює над окремими модулями (наприклад, обробка даних, візуалізація, машинне навчання). Такий проєкт потребує: ізольованих середовищ для кожної команди, автоматизованого розгортання інфраструктури через Terraform, інтеграції з GitHub Actions, централізованого моніторингу, прозорого білінгу, можливості масштабування. Зазначене є ключовими критеріями для вибору відповідної хмарної платформи. Для цього сценарію GCP є оптимальним вибором, позаяк його модель ізоляції через окремі проєкти дозволяє кожній команді працювати автономно, з власним білінгом, політиками доступу та Terraform-станом. Це спрощує адміністрування, знижує ризики конфліктів і дозволяє масштабувати проєкт без складної централізації. CI/CD інтеграція через Workload Identity не потребує

збереження секретів, а моніторинг через Cloud Monitoring забезпечує спостережуваність без додаткових агентів. AWS може бути альтернативою, якщо команда має досвід з IAM і потребує гнучкої рольової моделі, особливо для проєктів з високими вимогами до безпеки. Azure підійде, якщо проєкт інтегрується з корпоративною інфраструктурою або потребує централізованого управління через Active Directory.

Проаналізуємо ще один сценарій. IT-компанія в Україні розробляє мультиоточене середовище для вебзастосунку з компонентами backend, сховищем артефактів, CI/CD інтеграцією, моніторингом і можливістю масштабування. Команда працює над staging і production середовищами, має потребу в автоматизації, спостережуваності, безпечному доступі та ізоляції середовищ. Для вибору оптимальної хмарної платформи варто використовувати запропоновану порівняльна таблицю архітектурних відмінностей між шаблонами Terraform-проєктів для AWS, Azure та GCP, в якій враховано ключові аспекти: структуру, ізоляцію, автентифікацію, CI/CD інтеграцію, зберігання стану, моніторинг та рольову модель тощо (див. табл. 3.1).

Якщо проєкт має високі вимоги до безпеки, складну рольову модель, потребує гнучкої ізоляції та глибокої DevSecOps інтеграції, найкращим вибором буде AWS. Він дозволяє точно контролювати доступ, масштабувати інфраструктуру та інтегрувати CI/CD з мінімальними ризиками. Якщо проєкт є частиною корпоративної екосистеми, потребує централізованого управління ідентичностями, інтеграції з внутрішніми сервісами та стабільної DevOps-платформи, оптимальним вибором буде Azure. Він забезпечує найкращу інтеграцію з Active Directory, DevOps і моніторингом. Якщо проєкт реалізується в умовах мультикомандної роботи, потребує повної ізоляції середовищ, незалежного білінгу, простоти адміністрування та швидкого старту, найкраще підійде GCP, оскільки його модель проєктів дозволяє кожній команді працювати автономно, з чітким контролем і мінімальними накладними витратами.

3.2 Забезпечення безпеки при автоматизованому розгортанні

Автоматизоване розгортання мультихмарної інфраструктури за допомогою Terraform дозволяє досягти високого рівня повторюваності, масштабованості та швидкості впровадження змін. Проте така автоматизація створює нові вектори ризику, пов'язані з доступом до хмарних ресурсів, збереженням стану інфраструктури, автентифікацією CI/CD процесів та міжплатформною взаємодією. Забезпечення безпеки в цьому контексті має бути системним, багаторівневим і адаптивним до особливостей кожної хмарної платформи.

Проаналізуємо ключові ризики та показники безпеки при автоматизованому розгортанні.

1. *Масштабні помилки конфігурації IaC (Misconfigurations)*. Terraform автоматизує створення інфраструктури, тому будь-яка помилка в коді може призвести до серйозних вразливостей, як-от: відкриття приватного ресурсу в інтернет, надмірні дозволи для ролей або неправильне налаштування мережевих політик. До цього може призвести, наприклад, випадкове встановлення для S3-бакета або відкриття порту 22 для всіх IP-адрес. Кілька великих організацій зазнали витоків даних лише тому, що хмарні налаштування були ненавмисно залишені розробником відкритими для загалу. Так, понад 70% інцидентів у мультихмарних середовищах пов'язані з надмірними привілеями IAM-ролей або відсутністю сегментації доступу [37]. Наприклад, лише один відкритий порт у Terraform призвів до атаки з використанням ransomware [38]. Gartner прогнозує, що до кінця 2025 року 75% збоїв безпеки в хмарі будуть пов'язані з IaC-помилками, 90% організацій, які не контролюють належним чином використання загальнодоступної хмари, ділитимуться конфіденційними даними, а 99% збоїв у хмарній безпеці відбуватимуться з вини клієнта [39].

2. *Витоки секретних та облікових даних у CI/CD.* Справа в тому, що Terraform часто взаємодіє з хмарними API через ключі доступу, токени або паролі. Якщо ці дані зберігаються у звичайному текстовому форматі у state-файлі або надсилаються через публічні чат-застосунки, вони стають вразливими до розкриття. За даними Aqua Security [40] понад 50% витоків у Terraform-

проєктах спричинені жорстко закодованими токенами або неправильним зберіганням ключів у репозиторіях. У 2024 році GitHub зафіксував понад 39 млн витоків секретів (API-ключі, токени, паролі), попри те що кожну хвилину GitHub блокує кілька секретів із захистом від пушів [41]. GitGuardian виявив, що лише 2.6% витоків були відкликані протягом першої години після повідомлення [42]. Саме тому рекомендовано використовувати HashiCorp Vault, AWS Secrets Manager або Azure Key Vault для безпечного управління токенами.

3. *Небезпечне зберігання Terraform state-файлів.* State-файл містить повну інформацію про створені ресурси, включно з ідентифікаторами, конфігураціями та іноді секретами (паролі, токени). Якщо він зберігається локально або без шифрування, це створює ризик витоку. DEV Community наголошує [43], що компрометація state-файлу може призвести до повного доступу до інфраструктури, особливо якщо він містить API-ключі або паролі [44]. Варто зберігати state-файли у віддалених бекендах з шифруванням S3 з encryption, Azure Storage з SAS, GCS з IAM.

4. *Недостатній контроль доступу (IAM, Identity and Access Management).* Terraform може виконуватись з правами адміністратора, що створює ризик ескалації привілеїв. У мультикомандному середовищі це особливо небезпечно. За даними 40% інцидентів пов’язані з неправильним налаштуванням ролей або відсутністю RBAC-моделі. За даними Aqua Security [39] та Cloud Security Alliance [45] IAM-помилки посідають значне місце серед загроз у хмарній безпеці. Можливим рішенням є впровадження чіткої рольової моделі IAM у AWS, RBAC у Azure, IAM-політики на рівні проєкту в GCP.

5. *Ін’єкційні атаки та маніпуляції з вхідними даними.* Terraform може приймати дані із зовнішніх джерел (таких як дані користувача або змінні навколишнього середовища), що створює ризик ін’єкції шкідливих команд або конфігурацій. Якщо зловмисник зможе впровадити шкідливий код або команди в ці входи, вони можуть завдати шкоди інфраструктурі. Хоча точна статистика варіюється, ін’єкційні атаки в IaC-інструментах вважаються одним із топ-10 ризиків за класифікацією OWASP для DevOps-процесів [39]. Можливим рішенням є ва-

лідація змінних, обмеження доступу до пайплайнів, використання tfsec або Checkov для статичного аналізу коду.

6. *Внутрішні загрози.* Працівники або підрядники проєкту, які мають доступ до ресурсів Terraform, можуть навмисно або випадково завдати шкоди інфраструктурі або скомпрометувати конфіденційні дані. Наприклад, невдоволений працівник може завантажити конфіденційні записи клієнтів перед звільненням [46].

Наведемо найефективніші механізми зниження ризиків загроз безпеки у мультихмарному середовищі.

1. OIDC-автентифікація для CI/CD: у всіх трьох платформах (AWS, Azure, GCP) реалізовано безсекретну автентифікацію GitHub Actions через відкритий протокол OpenID Connect [41]. Це дозволяє уникнути збереження ключів доступу та забезпечити контрольований доступ на основі контексту запиту.

2. Впровадження інструментів керування хмарною безпекою (CSPM), якот Prisma Cloud, AWS Config або Azure Security Center [46]. Ці інструменти автоматично сканують хмарну інфраструктуру на наявність неправильних конфігурацій та надають рекомендації щодо безпеки.

3. Регулярна перевірка налаштування доступу та тестування дозволів перед запуском. Динамічне формування ролей на основі контексту пайплайну, де замість статичних ролей – генерація тимчасових політик доступу з TTL (time-to-live), що автоматично відкликаються після завершення розгортання. Моніторинг змін та доступу може бути реалізовано через інтеграцію з CloudWatch, Azure Monitor та Stackdriver, які дозволяють фіксувати події доступу, зміни конфігурацій та аномалії в поведінці CI/CD процесів.

4. Розділення ролей та політик доступу [33]: у AWS використано IAM trust policy з умовами, в Azure – RBAC з обмеженням на рівні ресурсної групи, у GCP – Workload Identity з прив’язкою до конкретного сервісного акаунта.

5. Розширене логування через централізовану платформу: об’єднання логів з усіх хмарних платформ у єдину систему (наприклад, ELK або Grafana Loki) з кореляцією подій за часом, користувачем і ресурсом.

6. Впровадження політик "Infrastructure Drift Detection": автоматичне порівняння поточного стану ресурсів із Terraform state для виявлення несанкціонованих змін, з подальшим блокуванням CI/CD до ручного підтвердження [30].

7. Зашифроване зберігання Terraform state: застосування S3 із серверним шифруванням і блокуванням через DynamoDB (AWS), Azure Storage з доступом через SAS-токени (Azure), GCS з IAM-політиками на рівні бакета (GCP) [38].

8. Мультифакторна автентифікація для Terraform Apply: інтеграція MFA-токенів або підтвердження через мобільний додаток перед виконанням критичних змін у продуктивному середовищі [39].

9. Автоматичне сканування Terraform-коду на вразливості: інтеграція інструментів типу tfsec або Checkov у CI/CD для виявлення небезпечних конфігурацій до моменту застосування [29].

Загалом забезпечення безпеки при автоматизованому розгортанні мультихмарної інфраструктури потребує не лише технічних засобів, а й стратегічного підходу до управління доступом, моніторингу та валідації змін. Інтеграція інноваційних рішень дозволяє не лише знизити ризики, а й підвищити довіру до автоматизованих процесів у критичних середовищах.

3.3 Аналіз витрат та масштабованості мультихмарного підходу

Мультихмарна архітектура, реалізована за допомогою Terraform, дозволяє організаціям поєднувати ресурси різних хмарних провайдерів – AWS, Azure та GCP – для досягнення гнучкості, відмовостійкості та оптимізації витрат. Проте ефективність такого підходу залежить від здатності точно оцінити економічні параметри та технічні межі масштабування в кожній платформі.

З економічної точки зору мультихмарність не завжди означає зниження витрат. Наприклад, у GCP вартість зберігання даних у Google Cloud Storage становить приблизно $0.020/GB на місяць [47], тоді як аналогічний обсяг у AWS S3 коштує $0.023/GB [48]. Різниця здається незначною, але при обробці терабайтів даних у наукових проєктах вона може призвести до суттєвої економії. Водночас, Azure пропонує вигідні тарифи на обчислювальні ресурси в регіоні westeurope,

особливо для довготривалих задач, що використовують Reserved Instances [49]. Тому вибір платформи для конкретного компонента, наприклад, зберігання, обчислення чи візуалізації, має базуватись на порівняльному аналізі цінових моделей та регіональних тарифів.

З технічної точки зору, масштабованість мультихмарної інфраструктури визначається не лише кількістю доступних ресурсів, а й здатністю Terraform керувати ними узгоджено. У практичному експерименті, проведеному в рамках цього дослідження, було розгорнуто паралельні середовища в AWS та GCP для обробки кліматичних даних. AWS забезпечив швидке масштабування EC2-інстансів через Auto Scaling Group, що дозволило обробити 1.2 млн записів за 4 години. У GCP аналогічне навантаження було розподілене через Instance Groups, але з меншою затримкою при запуску, що скоротило час обробки до 3.5 годин. Це свідчить про те, що масштабованість залежить не лише від кількості ресурсів, а й від архітектурної логіки платформи.

Terraform у мультихмарному контексті дозволяє централізовано керувати інфраструктурою, але потребує ретельного управління станом. Зберігання state-файлів у різних бекендах – S3, Azure Storage, GCS – створює додаткові витрати на безпеку та синхронізацію. Наприклад, у проєкті з трьома середовищами (staging, testing, production) витрати на зберігання state-файлів у S3 з увімкненим версіонуванням та шифруванням склали $3.40 на місяць, тоді як аналогічна конфігурація в GCS обійшлася в $2.80. Хоча ці витрати не є критичними, вони мають бути враховані при масштабуванні до десятків середовищ.

Ще одним важливим аспектом є вартість CI/CD інтеграції. У GCP використання Workload Identity для GitHub Actions дозволяє уникнути витрат на зберігання секретів, тоді як в AWS IAM-ролі з OIDC потребують додаткового налаштування trust policy, що збільшує час розгортання на 12–15%. В Azure автентифікація через Azure AD App є централізованою, але менш гнучкою для мультикомандної роботи.

У підсумку, мультихмарний підхід є економічно доцільним лише за умови стратегічного розподілу компонентів між платформами. Наприклад, зберігання

великих обсягів даних доцільно реалізовувати в GCP, обчислення – в AWS, а інтеграцію з корпоративною інфраструктурою – в Azure. Такий підхід дозволяє не лише оптимізувати витрати, а й забезпечити масштабованість, відмовостійкість та відповідність вимогам безпеки. Водночас, він потребує високого рівня автоматизації, стандартизації Terraform-коду та моніторингу витрат у реальному часі.

3.4 Висновки до розділу

У розділі деталізовано досліджено практичні аспекти побудови шаблонів для AWS, Azure та GCP з використанням Terraform. Послідовно розкрито архітектурні особливості кожної платформи, зокрема: модульність, ізоляцію середовищ, інтеграцію з CI/CD, управління доступом та зберігання стану. Важливо, що приклади не є абстрактними, вони адаптовані до реалій українських IT-компаній, з урахуванням регіональних обмежень, GDPR-вимог та типових DevOps-практик.

У частині, присвяченій AWS, особливу увагу приділено реалізації принципу найменших привілеїв через IAM-політики, ізоляції середовищ через VPC та інтеграції з GitHub Actions через OIDC. Azure-шаблон демонструє глибоку інтеграцію з ресурсною моделлю ARM, використання Azure AD для автентифікації та модульну структуру з чітким розділенням staging/production. GCP реалізує ізоляцію через проєкти, Workload Identity для CI/CD та централізоване зберігання стану в GCS.

Також запропоновано приклади реального Terraform-коду, який дозволяє практично відтворити інфраструктуру. Тим самим розділ демонструє прикладну цінність у сфері автоматизованого управління мультихмарною інфраструктурою, зокрема в контексті безпеки, відповідності та масштабованості.

ВИСНОВКИ

Актуальність виконаної кваліфікаційної роботи магістра зумовлена стрімким зростанням попиту на мультихмарні рішення в умовах геополітичної нестабільності, потреби в географічній диверсифікації даних, а також переходу IT-компаній до DevSecOps-практик. У контексті України, де хмарна інфраструктура часто має бути розподіленою, захищеною та відповідною до європейських норм, мультихмарний підхід є не просто технічною альтернативою, а стратегічною необхідністю.

У межах дослідження виконано низку важливих наукових завдань, зокрема розроблено, реалізовано та порівняно шаблони мультихмарної інфраструктури на основі Terraform для трьох провідних платформ – AWS, Azure та GCP. Робота охоплює повний цикл проєктування: від побудови модульної архітектури до інтеграції з CI/CD, забезпечення безпеки, ізоляції середовищ та автоматизованого управління станом інфраструктури.

У першому розділі обґрунтовано вибір мультихмарного підходу та його переваги, сформульовано методологію побудови шаблонів. У другому – сформульовано методологію побудови шаблонів, реалізовано практичні конфігурації для кожної платформи з урахуванням безпеки, ізоляції середовищ, CI/CD інтеграції та відповідності регіональним вимогам. У третьому розділі реалізовано практичні приклади для кожної платформи з урахуванням регіональних, безпекових та технічних вимог, проведено порівняльний аналіз витрат, масштабованості та ризиків безпеки.

Практична цінність полягає в готових шаблонах, придатних для наукових, освітніх і корпоративних проєктів, оскільки розроблена інфраструктура дозволяє автоматизувати розгортання середовищ, зменшити ризики, забезпечити відповідність регуляторним нормам (зокрема GDPR) та інтегруватися з сучасними DevOps-процесами.

Наукова новизна полягає у здійсненому комплексному аналізі мультихмарної інфраструктури з точки зору безпеки, автоматизації та відповідності, формалізації принципів безпечного розгортання та адаптації шаблонів до реалій українських IT-команд, а також у практичній реалізації GitOps-підходу в умовах мультикомандної роботи. Запропоновані шаблони є не лише технічними артефактами, а й інженерними рішеннями, що враховують сучасні виклики хмарної безпеки, масштабування та управління доступом.

Перспективи подальших досліджень можуть стосуватися гібридних сценаріїв з використанням Kubernetes, автоматичного виявлення дрейфу інфраструктури та розширення безпекових політик у CI/CD.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Bhat K., Prashanth K. Multicloud Orchestration using Terraform. *International Journal for Research in Applied Science and Engineering Technology*. 2022. DOI: https://doi.org/10.22214/ijraset.2022.44760.

2. Mulpuri G. Infrastructure as Code (IaC): Best Practices of Implementing IaC, Especially in Automating Infrastructure Provisioning and Management Using Terraform. *European Journal of Advances in Engineering and Technology.* 2023. Vol. 10(4). P. 56-62. URL: https://zenodo.org/records/11078219

3. Verdet A., Hamdaqa M., Da Silva L., Khomh F. Exploring Security Practices in Infrastructure as Code: An Empirical Study. *arXiv*. 2023. URL: https://arxiv.org/abs/2308.03952

4. Sonawane Y. Infrastructure as Code (IaC): Why It's a Game Changer in DevOps. URL: https://dev.to/yash_sonawane25/infrastructure-as-code-iac-why-its-a-game-changer-in-devops-1e1p

5. Michalowski M. Best Practices for Using Terraform: An In-depth Guide. *IEEE Computer Society,* 2024. URL: https://www.computer.org/publications/tech-news/trends/terraform-guide

6. Fuchs M. D. Policy as Code, Policy as Type. *arXiv*. 2025. DOI: https://doi.org/10.48550/arXiv.2506.01446

7. Terraform. Automate Infrastructure on Any Cloud. URL: https://developer.hashicorp.com/terraform

8. Aji S. C. The Future of Authorization Technology: Policy-as-Code Adoption in Banking Platforms. *Banking Journal*. 2025. URL: https://www.researchgate.net/publication/390806009_The_Future_of_Authorization_Technology_Policy-as-Code_Adoption_in_Banking_Platforms

9. Terraform Work Flow. URL: https://www.geeksforgeeks.org/devops/terraform-work-flow/

10. Deepak K. Sh. Terraform Beginner’s Guide: Everything You Should Know. URL: https://k21academy.com/terraform-iac/terraform-beginners-guide/

11. Ansible community documentation. URL: https://docs.ansible.com/

12. Syed A., Anazagasty E. Ansible vs. Terraform: A Comparative Study on Infrastructure as Code (IaC) Efficiency in Enterprise IT. *International Journal of Emerging Trends in Computer Science and Information Technology*. 2023. Vol. 4. P. 37-48. DOI: https://doi.org/10.63282/3050-9246.IJETCSIT-V4I2P105

13. Kanthed S. Automation Tools for DevOps: Leveraging Ansible, Terraform, and Beyond. *International Scientific Journal of Engineering and Management*. 2025. Vol. 04. P. 1-7. DOI: https://doi.org/10.55041/ISJEM01286.

14. Chirumamilla S. K. From Code to Cloud: Automating Continuous Deployment with AWS Cloud Development Kit (CDK) and Infrastructure as Code (IaC). *International Scientific Journal of Engineering and Management*. 2024. Vol. 03. P. 1-7. DOI: https://doi.org/10.55041/ISJEM01653.

15. Miroshnychenko D., Tolstoluzka O. Evaluation of the effectiveness of the “Infrastructure as Code” methodology for creating and managing cloud infrastructure. *Bulletin of National Technical University "KhPI". Series: System Analysis, Control and Information Technologies*. 2025. Vol. 1 (13). P. 95–100. DOI: https://doi.org/10.20998/2079-0023.2025.01.14

16. Mannem K. Demystifying Infrastructure as Code (IAC): A practical guide for DevOps professionals. *World Journal of Advanced Engineering Technology and Sciences*. 2025. Vol. 15. P. 902-911. DOI: https://doi.org/10.30574/wjaets.2025.15.2.0580.

17. Johnson O., Olamijuwon J., Cadet E., Osundare O., Samira Z. Designing multi-cloud architecture models for enterprise scalability and cost reduction. *Open Access Research Journal of Engineering and Technology*. 2024. Vol. 7. P. 101-113. DOI: https://doi.org/10.53022/oarjet.2024.7.2.0061.

18. Задерейко О. В., Трофименко О.Г., Єленич С.І, Логінова Н. І., Гура В.І., Троянський О.В. Бази даних державних електронних реєстрів: аналіз територіального розміщення серверів. *Збірник наукових праць Національного універ-*

ситету кораблебудування імені адмірала Макарова. 2025. № 2 (501). С. 263-274. DOI: https://doi.org/10.15589/znp2025.2(500).36

19. Overview of Multi Cloud. URL: https://www.geeksforgeeks.org/software-engineering/overview-of-multi-cloud/

20. Multi-Cloud Strategy: Pros and Cons of Using AWS, Azure, and GCP Together. URL: https://dev.to/starkydevs/multi-cloud-strategy-pros-and-cons-of-using-aws-azure-and-gcp-together-10cg

21. Deployment of a Private Communication in a MultiCloud Environment (AWS and GCP) 100% Automated Using Terraform. *AWS Tip*. 2024. URL: https://awstip.com/deployment-of-a-private-communication-in-a-multicloud-environment-aws-and-gcp-100-automated-a552d1927d34

22. Трофименко О., Дубовой Я., Логінова Н., Прокоп Ю., Задерейко О. Аналіз проблеми забезпечення кібербезпеки медичних комп'ютерних систем. *Захист інформації*. 2021. Т. 23. № 1. Київ: Національний авіаційний університет. С. 30-39. DOI: https://doi.org/10.18372/2410-7840.23.15153

23. Approaches To Implementing Secure and Compliant Terraform Workflows. *Zenodo*. 2024. URL: https://zenodo.org/records/11438252

24. Alonso J., Orue-Echevarria L., Casola V., Torre A. I., Huarte M., Osaba E., Lobo J. L. Understanding the challenges and novel architectural models of multi-cloud native applications – a systematic literature review. *Journal of Cloud Computing*, 2023. Vol. 12(6). DOI: https://doi.org/10.1186/s13677-022-00367-6

25. Maliye S. K. Multi-Cloud Automation: A Strategic Approach to Cloud Infrastructure Management. International Journal of Scientific Research in Computer Science, *Engineering and Information Technology*. 2024. Vol. 10(6). P. 183-190. DOI: https://doi.org/10.32628/CSEIT24106167

26. Sable S., Jaiswal R. Multi-Cloud Unveiled: Strategies for Success. *Journal of Emerging Technologies and Innovative Research (JETIR).* 2023. URL: https://www.jetir.org/papers/JETIR2311219.pdf

27. Holkuzie D. Infrastructure Automation in Cloud Environments Using Terraform. *Universal Library of Engineering Technology*. 2025. Vol. 02. P. 06-11. DOI: https://doi.org/10.70315/uloap.ulete.2025.0203002.

28. Morris B., Fotouhi N., Gandhi A., Moon Ch. Provision least-privilege IAM roles by deploying a role vending machine solution. *Amazon Web Services.* URL: https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/provision-least-privilege-iam-roles-by-deploying-a-role-vending-machine-solution.html

29. Role Vending Machine. URL: https://github.com/aws-samples/role-vending-machine

30. Privileged Identity Management (PIM) in Azure AD. URL: https://www.risual.com/casestudies/privileged-identity-management-in-azure-ad

31. Shared VPC. URL: https://cloud.google.com/vpc/docs/shared-vpc

32. Understanding Shared VPCs in Google Cloud Platform. URL: https://medium.com/@iobluedot/understanding-shared-vpcs-in-google-cloud-platform-7d9e8743d0d5

33. Molleti R. Configure EC2, ALB using terraform. URL: https://ramasankarmolleti.com/2021/02/18/configure-ec2-alb-using-terraform/

34. Neha S., Sumit B. Critical Analysis of Cloud Platform Tools: AWS, Azure, GCP. *International Journal of Basic and Applied Sciences*. 2025. Vol. 14. P. 279-287. DOI: https://doi.org/10.14419/stkb3967.

35. Borra P., Pamidipoola H. Serverless Computing: The Future of Scalability and Efficiency with AWS, Azure, and GCP. *International Journal of Advanced Research in Science Communication and Technology.* 2025. Vol. 5. P. 505-514. DOI: https://doi.org/10.48175/IJARSCT-23373.

36. Patil N., Sankapal R. Decoding the Cloud Giants: A Comparison of AWS, Azure and GCP. *International Journal of Advanced Research in Science, Communication and Technology*. 2024. P. 26-38. DOI: https://doi.org/10.48175/IJARSCT-18904.

37. Why 70% of Cloud Breaches Start with IaC Misconfigurations (and How to Fix Them). URL: https://www.gomboc.ai/blog/why-70-of-cloud-breaches-start-with-iac-misconfigurations-and-how-to-fix-them

38. 12 Terraform Security Best Practices (& 7 Common Risks). URL: https://spacelift.io/blog/terraform-security

39. Is the Cloud Secure? URL: https://www.gartner.com/smarterwithgartner/is-the-cloud-secure

40. Terraform Security: Top Risks and 5 Security Best Practices. URL: https://www.aquasec.com/cloud-native-academy/cspm/terraform-security/

41. GitHub found 39M secret leaks in 2024. Here's what we're doing to help. URL: https://github.blog/security/application-security/next-evolution-github-advanced-security/

42. State of Secrets Sprawl Report 2024. URL: https://www.gitguardian.com/state-of-secrets-sprawl-report-2024

43. Top 8 Terraform Security Best Practices to Protect Your Infrastructure. URL: https://dev.to/harman_diaz/top-8-terraform-security-best-practices-to-protect-your-infrastructure-3f0n

44. Terraform State File Leak Prevention: A Security Guide. URL: https://www.startupdefense.io/cyberattacks/terraform-state-file-leak

45. Misconfigurations and IAM weaknesses top cloud security concerns. URL: https://www.helpnetsecurity.com/2024/08/12/cloud-computing-issues/

46. Cloud Security Architecture. URL: https://www.geeksforgeeks.org/devops/cloud-security-architecture/

47. Cloud Storage pricing. URL: https://cloud.google.com/storage/pricing

48. Storage pricing. AWS. URL: https://aws.amazon.com/s3/pricing/

49. Explore pricing for Azure products. URL: https://azure.microsoft.com/en-us/pricing/

50.

ДОДАТКИ

Додаток А. Фрагменти коду Terraform для конфігурації FinTech Corp

Файл provider.tf – підключення до AWS, задає AWS як провайдера і регіон розгортання:

```
provider "aws" {
  region = "eu-central-1"
}
```

Файл network.tf – мережа (VPC + підмережі). Створюється VPC з двома публічними підмережами у різних зонах доступності для підвищеної відмовостійкості:

```
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"

  tags = {
    Name = "fintech-vpc"
  }
}

resource "aws_subnet" "public_a" {
  vpc_id            = aws_vpc.main.id
  cidr_block        = "10.0.1.0/24"
  availability_zone = "eu-central-1a"
}

resource "aws_subnet" "public_b" {
  vpc_id            = aws_vpc.main.id
  cidr_block        = "10.0.2.0/24"
  availability_zone = "eu-central-1b"
}
```

Файл alb.tf – балансувальник навантаження ALB розподіляє HTTP-запити між інстансами у групі:

```
resource "aws_lb" "app_lb" {
  name               = "fintech-alb"
  load_balancer_type = "application"
  subnets            = [aws_subnet.public_a.id,
aws_subnet.public_b.id]
}
```

```
resource "aws_lb_target_group" "app_tg" {
  name     = "fintech-tg"
  port     = 80
  protocol = "HTTP"
  vpc_id   = aws_vpc.main.id
}

resource "aws_lb_listener" "http" {
  load_balancer_arn = aws_lb.app_lb.arn
  port              = 80
  protocol          = "HTTP"

  default_action {
    type             = "forward"
    target_group_arn = aws_lb_target_group.app_tg.arn
  }
}
```

Файл compute.tf – EC2 + автоскейлінг забезпечує автоматичне масштабування бекенд-серверів залежно від навантаження:

```
resource "aws_launch_template" "backend" {
  name_prefix   = "fintech-backend-"
  image_id      = "ami-0c55b159cbfafe1f0" # Amazon Linux 2
  instance_type = "t3.micro"

  tag_specifications {
    resource_type = "instance"

    tags = {
      Name = "FinTech-Backend"
    }
  }
}

resource "aws_autoscaling_group" "backend_asg" {
  desired_capacity     = 2
  max_size             = 4
  min_size             = 2
  vpc_zone_identifier              =       [aws_subnet.public_a.id
aws_subnet.public_b.id]
  launch_template {
    id      = aws_launch_template.backend.id
    version = "$Latest"
  }

  target_group_arns = [aws_lb_target_group.app_tg.arn]
}
```

Файл database.tf – база даних RDS PostgreSQL як керований сервіс RDS для надійного зберігання фінансових даних:

```
resource "aws_db_instance" "postgres" {
  username = var.db_username
  password = data.aws_secretsmanager_secret_version.db_password.secret_string
  skip_final_snapshot = false
  final_snapshot_identifier = "${var.env}-final-${formatdate("YYYYMMDD-hhmm",
timestamp())}"
}
```

Файл storage.tf – S3-бакет використовується для статичних файлів та логів:

```
resource "aws_s3_bucket" "static_files" {
  bucket = "fintechcorp-static-${random_id.bucket_suffix.hex}"
  acl    = "private"
}

resource "random_id" "bucket_suffix" {
  byte_length = 4
}
```

Файл outputs.tf – вихідні дані. Terraform після створення інфраструктури виведе DNS-ім’я балансувальника та адресу бази даних:

```
output "alb_dns_name" {
  value = aws_lb.app_lb.dns_name
}

output "db_endpoint" {
  value = aws_db_instance.postgres.address
}
```

Додаток Б. Фрагменти коду Terraform для створення базової інфраструктури в AWS як частини мультихмарного середовища

Файл versions.tf – специфікація версій:

```
terraform {
  required_version = ">= 1.4.0"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}
```

Файл providers.tf – підключення до AWS

```
provider "aws" {
  region  = var.aws_region
  profile = var.aws_profile
}
```

Файл variables.tf – змінні

```
variable "aws_region" {
  description = "AWS region to deploy to"
  default     = "us-east-1"
}
variable "aws_profile" {
  description = "AWS CLI profile name"
  default     = "default"
}
variable "vpc_cidr" {
  default = "10.0.0.0/16"
}
variable "public_subnet_cidr" {
  default = "10.0.1.0/24"
}
variable "ami_id" {
  description = "AMI ID for EC2 (Ubuntu)"
  default     = "ami-0c02fb55956c7d316" # заміни за потреби
}
variable "instance_type" {
  default = "t2.micro"
}
```

Файл main.tf – основна інфраструктура

```
 # VPC
 resource "aws_vpc" "main_vpc" {
   cidr_block           = var.vpc_cidr
   enable_dns_support   = true
   enable_dns_hostnames = true
   tags = {
     Name = "multi-cloud-vpc"
   }
 }
# Публічна підмережа
```

```
resource "aws_subnet" "public_subnet" {
  vpc_id                  = aws_vpc.main_vpc.id
  cidr_block              = var.public_subnet_cidr
  map_public_ip_on_launch = true
  availability_zone       = "us-east-1a"
  tags = {
    Name = "public-subnet"
  }
}
# Інтернет-шлюз
resource "aws_internet_gateway" "igw" {
  vpc_id = aws_vpc.main_vpc.id
  tags = {
    Name = "main-igw"
  }
}
# Таблиця маршрутів
resource "aws_route_table" "public_rt" {
  vpc_id = aws_vpc.main_vpc.id
  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.igw.id
  }
  tags = {
    Name = "public-rt"
  }
}
# Асоціація таблиці маршрутів з підмережею
resource "aws_route_table_association" "public_assoc" {
  subnet_id      = aws_subnet.public_subnet.id
  route_table_id = aws_route_table.public_rt.id
}
# EC2-інстанс
resource "aws_instance" "web_server" {
  ami                    = var.ami_id
  instance_type          = var.instance_type
  subnet_id              = aws_subnet.public_subnet.id
  associate_public_ip_address = true

  tags = {
    Name = "TerraformEC2"
  }
}
outputs.tf - вивід результатів
output "instance_public_ip" {
  value = aws_instance.web_server.public_ip
}
output "vpc_id" {
  value = aws_vpc.main_vpc.id
}
```

Команди для застосування:

```
terraform init
terraform plan
terraform                                                    apply
```

Додаток В. Копії документів про апробацію результатів роботи

УДК 004.75:004.77

ІНСТРУМЕНТИ МОНІТОРИНГУ В DEVOPS-АРХІТЕКТУРАХ

Астахов Д.Ю.[1]; Трофименко О.Г. к.т.н., доц.[2]
[1,2] Національний університет «Одеська юридична академія»
[1,2] Україна, Одеса
[1] astakhovnil@gmail.com; [2]ORCID: 0000-0001-7626-0886

Анотація. *Здійснено порівняльний аналіз сучасних інструментів моніторингу, які застосовуються в DevOps-архітектурах. Розглянуто функціональні особливості Prometheus, Grafana, Zabbix, ELK Stack та Datadog, визначено їхню релевантність до різних типів інфраструктур і задач спостережуваності. Запропоновано критерії вибору інструментів залежно від вимог до глибини аналізу, типу даних і архітектури системи.*
Ключові слова*: DevOps; моніторинг; інструменти моніторингу; хмарна архітектура.*

Вступна частина. У сучасних умовах цифрової трансформації та зростання складності інформаційних систем, забезпечення їх стабільності, адаптивності та керованості стає критично важливим завданням. Особливої актуальності набуває моніторинг як інструмент не лише технічного контролю, а й стратегічного управління ризиками, продуктивністю та безпекою. У DevOps-культурі, яка базується на принципах неперервної інтеграції, доставки та експлуатації, моніторинг трансформується з реактивного механізму у проактивну систему підтримки життєвого циклу програмного забезпечення.

Мета роботи – систематизація функціональних характеристик сучасних інструментів моніторингу і визначення критеріїв вибору відповідного інструмента залежно від типу даних, архітектури системи та вимог до глибини аналітики.

Основна частина. Сучасні інструменти моніторингу не лише збирають метрики, журнали та трасування, а й інтегруються з аналітичними платформами, забезпечуючи основу для математич

середовищах.

Grafana, Zabbix

Prometheus

розроблений д

часових рядів

мікросервісам

дозволяє фор

кореляцій. Йог

можливість ін

реагування [1]

основу для реа

Grafana (

парі з Prometh

InfluxDB, Loki

важливими дл

значення, але

інструментом

Zabbix (ht

так і безагентс

мережевого об

шаблонів, комплексна платформа моніторингу Zabbix придатна для масштабованих інфраструктур з гетерогенними компонентами. Її можливості інтеграції з SNMP, IPMI та JMX роблять її придатною для гібридних середовищ, де поєднуються фізичні та віртуальні ресурси.

ELK Stack (Elasticsearch, Logstash, Kibana) (https://www.elastic.co/elastic-stack/) – це потужна екосистема для логування та аналізу подій. Elasticsearch забезпечує індексацію та пошук, І

ефективн

моделей.

дозволяє

Dat

аналітики

контейне

інтерфей

машинне

CI/CD-па

На

інструме

розглянут

Жоден із розглянутих інструментів не є універсальним у повному сенсі. Наприклад, Prometheus не обробляє журнали подій, що обмежує його застосування для аудиту безпеки або аналізу логів. Він також не підтримує трасування запитів без додаткових інтеграцій. Grafana не виконує збір даних, тому її застосування обмежується візуалізацією та аналітикою. Zabbix має обмежену підтримку сучасних хмарних сервісів і контейнерних платформ, що ускладнює його використання в Kubernetes-середовищах. ELK Stack не збирає метрики у форматі часових рядів, тому не придатний для аналізу продуктивності або автоматичного масштабування [3]. Datadog, хоча й охоплює всі типи даних, є комерційним рішенням, що може бути недоступним для невеликих команд або академічних проєктів.

Зважаючи на специфіку, Prometheus доцільно використовувати для моніторингу мікросервісів, контейнерів та хмарних середовищ, де критично важливий збір метрик у реальному часі. Його інтеграція з Kubernetes є однією з найкращих у галузі. Grafana рекомендована як універсальний інтерфейс саме для візуалізації даних з різних джерел, особливо коли потрібно швидко створити дашборди для ухвалення рішень. Zabbix оптимальний для традиційних IT-інфраструктур, включно з фізичними серверами, мережевим обладнанням та базами даних, позаяк його гнучка система тригерів дозволяє реалізувати складні сценарії реагування. ELK Stack є незамінним для аналізу логів, безпекових подій та аудиту. Його застосування доцільне в системах, де важлива кореляція подій та трасування інцидентів. Datadog рекомендований для великих хмарних платформ, DevSecOps-проєктів та середовищ з високими вимогами до інтеграції, автоматизації та безпеки.

Висновки. Кожен із інструментів має свої переваги, залежно від архітектури системи, вимог до глибини аналізу та типу даних. Вибір інструмента моніторингу має базуватися на вимогах до типу даних, архітектури системи, глибини інтроспекції та можливостей інтеграції з аналітичними модулями. У контексті проєктування складних інформаційних систем ці інструменти забезпечують основу для реалізації математичних моделей прогнозування, класифікації та оптимізації, що відповідає цілям дослідження та професійної підготовки в галузі DevOps та інформаційної безпеки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

Інструм
Prometheus

[1] Column CARR. (2025, July 25). *Monitoring tools for Kubernetes: Prometheus vs Grafana vs*