Хмельницький національний університет
Факультет інформаційних технологій
Кафедра комп'ютерної інженерії та інформаційних систем

КВАЛІФІКАЦІЙНА РОБОТА

Узагальнений метод керування життєвим циклом
Terraform-інфраструктури для кількох середовищ
Назва теми

Рівень вищої освіти *другий (магістерський)*

Галузь знань *12 «Інформаційні технології»*
Шифр, назва

Спеціальність *123 «Комп'ютерна інженерія»*
Шифр, назва

Освітня програма *«Комп'ютерна інженерія та програмування»*
Назва

Шифр КвРКІ 240234.24.02.08 ПЗ

Виконав здоб. ІІ курсу, гр. КІ2м-24-2 _____ Денис КОЛОМИЦЬКИЙ
Підпис Ім'я, ПРІЗВИЩЕ

Керівник д.т.н., професор _____ Олег САВЕНКО
Науковий ступінь, учене звання Підпис Ім'я, ПРІЗВИЩЕ

Нормоконтролер д.т.н., професор _____ Сергій ЛИСЕНКО
Науковий ступінь, учене звання Підпис Ім'я, ПРІЗВИЩЕ

До захисту допускаю:
завідувач кафедри КІС _____ Ольга ПАВЛОВА
« 5 » травня 2026 р. Підпис Ім'я, ПРІЗВИЩЕ
дата

Хмельницький 2026

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій

Кафедра Комп'ютерної інженерії та інформаційних систем

Рівень вищої освіти другий (магістерський)

Галузь знань 12 Інформаційні технології

Спеціальність 123 Комп'ютерна інженерія

Освітня програма «Комп'ютерна інженерія та програмування»

ЗАТВЕРДЖУЮ
Завідувачка кафедри КІС
Ольга ПАВЛОВА

« 12 » 01 2026 р.

ЗАВДАННЯ НА КВАЛІКАЦІЙНУ РОБОТУ

Коломицький Денис Євгенійович
Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи): «Узагальнений метод керування життєвим циклом Terraform-інфраструктури для кількох середовищ»

Керівник проекту (роботи): Савенко Олег Станіславович, д.т.н., проф.
Прізвище, ім'я, по-батькові, науковий ступінь, вчене звання

Затверджено наказом університету від 12.01.2026 р. №6

2. Термін подання здобувачем роботи на кафедру 01.05.2026 р.

3. Вихідні дані по проекту Завдання на кваліфікаційну роботу

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)

Аналіз існуючих підходів до автоматизації та управління Terraform-інфраструктурою для кількох середовищ

Узагальнений метод керування життєвим циклом Terraform-інфраструктури для кількох середовищ

Алгоритмічне забезпечення та архітектурне проєктування методу

Практична реалізація та експериментальна оцінка методу

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

Блок-схеми алгоритмів керування життєвим циклом Terraform-інфраструктури

Архітектурна схема компонентів реалізації методу

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали і посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання « 12 » 01 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів (розділів) кваліфікаційної роботи магістра	Термін виконання етапів роботи	Примітка
1	Вибір напряму дослідження та узгодження тематики КвРМ з керівником	12.01.2026	виконано
2	Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження	12.01.2026	виконано
3	Робота над розділом 1 — аналіз існуючих підходів до автоматизації та управління Terraform-інфраструктурою для кількох середовищ	20.01.2026	виконано
4	Робота над розділом 2 — узагальнений метод керування життєвим циклом Terraform-інфраструктури для кількох середовищ	01.02.2026	виконано
5	Робота над науковою статтею	01.03.2026	виконано
6	Робота над розділом 3 — алгоритмічне забезпечення та архітектурне проєктування методу	15.03.2026	виконано
7	Робота над розділом 4 — практична реалізація та експериментальна оцінка методу	01.04.2026	виконано
8	Оформлення пояснювальної записки згідно вимог	18.04.2026	виконано
9	Попередній захист КвРМ	29.04.2026	
10	Захист КвРМ на засіданні ЕК	До 15.05.2026	

Здобувач _____ Денис КОЛОМИЦЬКИЙ
Підпис Ім'я, ПРІЗВИЩЕ

Керівник кваліфікаційної роботи _____ Олег САВЕНКО
Підпис Ім'я, ПРІЗВИЩЕ

РЕФЕРАТ

Тема кваліфікаційної роботи магістра: «Узагальнений метод керування життєвим циклом Terraform-інфраструктури для кількох середовищ».

Автор роботи: Коломицький Денис Євгенійович, ст. гр. КІ2м-24-2.

Керівник роботи: Савенко Олег Станіславович, д.т.н., професор.

Пояснювальна записка: 80 с., 7 рис., 9 табл., 2 дод., 81 джерело.

INFRASTRUCTURE AS CODE, TERRAFORM, ЖИТТЄВИЙ ЦИКЛ ІНФРАСТРУКТУРИ, МУЛЬТИСЕРЕДОВИЩНЕ РОЗГОРТАННЯ, КОНФІГУРАЦІЙНИЙ ДРЕЙФ, БЕЗПЕРЕРВНА ДОСТАВКА, ПОЛІТИКИ ВІДПОВІДНОСТІ.

Об'єктом дослідження є процес керування життєвим циклом Terraform-інфраструктури для кількох середовищ. Предметом дослідження є методи, моделі та алгоритми управління цим циклом. Метою роботи є розроблення узагальненого методу керування життєвим циклом Terraform-інфраструктури для кількох середовищ. Для розв'язання поставлених задач використовувалися методи математичного моделювання, теорія скінченних автоматів, теорія множин та порівняльний аналіз.

Наукова новизна отриманих результатів:

– вперше запропоновано двопроекційну модель життєвого циклу Terraform-інфраструктури для кількох середовищ у вигляді скінченного автомата з восьми стадій та простору середовищ як частково впорядкованої множини, що відрізняється виокремленням перевірки відповідності плану як самостійної стадії та замкненим циклом виявлення і усунення конфігураційного дрейфу;

– отримав подальший розвиток метод просування змін між середовищами, що відрізняється формалізацією передумови готовності середовища-джерела, версіонованими знімками конфігурацій та предикатом безпечності відкату з урахуванням зворотності ресурсів;

– удосконалено підхід до перевірки відповідності інфраструктурних конфігурацій за рахунок багаторівневої моделі політик з ієрархічним успадкуванням та

трикласової класифікації конфігураційного дрейфу.

У першому розділі проведено аналіз існуючих підходів до управління Terraform-інфраструктурою, розглянуто еволюцію парадигм IaC, методи забезпечення якості, безпеки та автоматизації конвеєрів доставки. У другому розділі розроблено формальний апарат та двопроекційну модель життєвого циклу у вигляді скінченного автомата, механізм просування змін, відкату та багаторівневу модель перевірки відповідності. У третьому розділі розроблено алгоритмічне забезпечення методу, сформульовано вимоги до програмних засобів та проведено порівняльний аналіз з існуючими підходами. У четвертому розділі виконано практичну реалізацію на платформі Scalr з інфраструктурою AWS та експериментальну оцінку за шістьма метриками.

Практична значимість полягає у можливості застосування методу при проєктуванні конвеєрів керування мультисередовищною Terraform-інфраструктурою в організаціях, що потребують формалізованого контролю просування змін, багаторівневої перевірки відповідності та автоматизованого виявлення конфігураційного дрейфу.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

AWS — Amazon Web Services, хмарна платформа Amazon.

CI/CD — Continuous Integration / Continuous Delivery, безперервна інтеграція та доставка.

CLI — Command Line Interface, інтерфейс командного рядка.

DevOps — Development Operations, методологія розробки та експлуатації програмного забезпечення.

EaC — Everything as Code, парадигма «все як код».

EC2 — Elastic Compute Cloud, сервіс віртуальних обчислювальних ресурсів AWS.

GCP — Google Cloud Platform, хмарна платформа Google.

HCL — HashiCorp Configuration Language, мова опису конфігурацій HashiCorp.

HCP — HashiCorp Cloud Platform, хмарна платформа HashiCorp.

IAM — Identity and Access Management, управління ідентифікацією та доступом.

IaC — Infrastructure as Code, інфраструктура як код.

IfC — Infrastructure from Code, інфраструктура з коду.

OPA — Open Policy Agent, рушій перевірки політик відповідності.

RDS — Relational Database Service, сервіс реляційних баз даних AWS.

S3 — Simple Storage Service, сервіс об’єктного зберігання даних AWS.

VPC — Virtual Private Cloud, віртуальна приватна мережа.

TF — Terraform.

ВСТУП

Поширення хмарних технологій та методологій DevOps сформувало стійку тенденцію до управління інфраструктурою засобами версіонованого коду — парадигму Infrastructure as Code (IaC). Серед інструментів цього класу Terraform посідає центральне місце завдяки декларативній мові опису HCL, провайдеро-незалежній архітектурі та явному механізму управління станом через файл `terraform.tfstate`. Практика розгортання інфраструктури для кількох середовищ (розробки, тестування та виробництва) є загальноприйнятою, проте управління їхнім життєвим циклом залишається методологічно нерозв'язаною задачею: існуючі підходи (Terraform Workspaces, Terragrunt, GitOps CI/CD) покривають лише окремі аспекти, не забезпечуючи одночасно повної ізоляції середовищ, формалізованого просування змін та автоматизованого виявлення конфігураційного дрейфу. Ця проблема підтверджується систематичними оглядами академічної літератури, що фіксують брак стандартизованих практик управління станом як одну з ключових невирішених проблем галузі.

Актуальність роботи полягає у відсутності єдиного підходу, що одночасно забезпечував би повне покриття стадій життєвого циклу Terraform-інфраструктури для кількох середовищ, формалізовану координацію просування змін та автоматизоване виявлення і усунення конфігураційного дрейфу. Наявні в галузі практики формуються переважно емпірично, без опори на формалізовані моделі, що ускладнює їхнє порівняння, відтворення та систематичне вдосконалення. Це призводить до того, що організації самостійно винаходять патерни управління станом та просуванням змін, часто повторюючи ті самі помилки й не маючи критеріїв для обґрунтованого вибору між альтернативами. Розроблення узагальненого методу, що спирається на формальний апарат та систему кількісних показників, усуває цю прогалину і дає практикам інструмент для обґрунтованих архітектурних рішень.

Метою кваліфікаційної роботи магістра є розроблення узагальненого методу керування життєвим циклом Terraform-інфраструктури для кількох середовищ.

Поставлена мета досягається розв'язанням таких основних завдань:

1. Проаналізувати існуючі патерни ізоляції середовищ у Terraform-інфраструктурі та обґрунтувати критерії вибору між ними.

2. Розробити модель життєвого циклу Terraform-інфраструктури для кількох середовищ, що охоплює стадії від ініціалізації конфігурацій до виявлення та усунення конфігураційного дрейфу.

3. Обґрунтувати метод управління станом та версіонування конфігурацій у мультисередовищних Terraform-проєктах з інтеграцією перевірок якості і безпеки у конвеєр просування змін.

4. Реалізувати запропонований метод на реальній мультисередовищній інфраструктурі та оцінити його ефективність за визначеними показниками.

Об'єктом дослідження є процес керування життєвим циклом Terraform-інфраструктури для кількох середовищ.

Предметом дослідження є методи, моделі та алгоритми управління життєвим циклом Terraform-інфраструктури для кількох середовищ.

Наукова новизна отриманих результатів:

1. Вперше запропоновано двопроекційну модель життєвого циклу Terraform-інфраструктури для кількох середовищ у вигляді скінченного автомата з восьми стадій та простору середовищ, що відрізняється виокремленням перевірки відповідності плану як самостійної стадії та замкненим циклом виявлення і усунення конфігураційного дрейфу;

2. Отримав подальший розвиток метод просування змін між середовищами, що відрізняється формалізацією передумови готовності середовища-джерела, версіонованими знімками конфігурацій та предикатом безпечності відкату з урахуванням зворотності ресурсів;

3. Удосконалено підхід до перевірки відповідності інфраструктурних конфігурацій за рахунок багаторівневої моделі політик з ієрархічним успадкуванням та трикласової класифікації конфігураційного дрейфу з ефективною дельтою.

Практична значимість отриманих результатів полягає у реалізації запропонованого методу на платформі Scalr з інфраструктурою AWS, що підтвердила повне виявлення порушень політик до застосування змін, відсутність хибних сповіщень при моніторингу дрейфу та скорочення кількості ручних операцій при просуванні змін між середовищами до одного кроку.

Для розв'язання поставлених задач використовувалися методи математичного моделювання, теорія скінченних автоматів, теорія множин, порівняльний аналіз та методи експериментального дослідження на реальній хмарній інфраструктурі.

За темою кваліфікаційної роботи опубліковано наукову статтю «Generalized method for managing the lifecycle of Terraform infrastructure accross multiple environments» [81].

1 АНАЛІЗ ІСНУЮЧИХ ПІДХОДІВ ДО АВТОМАТИЗАЦІЇ ТА УПРАВЛІННЯ TERRAFORM-ІНФРАСТРУКТУРОЮ ДЛЯ КІЛЬКОХ СЕРЕДОВИЩ

1.1 Еволюція парадигм автоматизації інфраструктури та концептуальна база управління її життєвим циклом

Сучасні підходи до розробки програмного забезпечення зазнали суттєвої трансформації під впливом концепції автоматизації операційних процесів через версіонований код. Узагальненням цієї тенденції стала парадигма «Everything as Code» (EaC) — систематизований підхід, за якого будь-який артефакт програмної системи, включно з інфраструктурою, конфігурацією, політиками безпеки та процесами доставки, описується та управляється у формі програмного коду [79]. Мультивокальний огляд 128 джерел, проведений Wei зі співавторами, дозволив ідентифікувати 25 окремих практик EaC та побудувати їхню таксономію у вигляді функціональних шарів. Кожен шар відповідає певному рівню абстракції системи — від базової інфраструктури до процесів управління відповідністю — і підтримує принцип єдиного джерела істини для всіх учасників розробки [70].

Центральне місце у цій таксономії займає практика Infrastructure as Code (IaC) — підхід, що передбачає опис ресурсів обчислювального середовища у декларативних або процедурних файлах конфігурації, які зберігаються у системі контролю версій і виконуються автоматизованими засобами [7]. Поява IaC стала відповіддю на обмеженість ручного управління серверами та мережевою інфраструктурою у контексті масштабованих хмарних систем: традиційні методи не забезпечували відтворюваності середовищ, призводили до накопичення конфігураційного боргу та унеможливлювали ефективну взаємодію між командами розробки та експлуатації [43]. Дослідники пов’язують формалізацію поняття IaC з поширенням методологій DevOps на початку 2010-х років, коли потреба в автоматизованому й відтворюваному забезпеченні середовищ стала передумовою для

безперервного розгортання [54].

Теоретичне осмислення місця IaC у програмній інженерії пройшло кілька етапів. На початковому етапі дослідники зосереджувались переважно на розробці інструментів та вивченні їхньої придатності для конкретних задач [56]. Систематичний огляд 31 498 публікацій, здійснений Rahman зі співавторами, показав, що розробка інструментів є найбільш дослідженим напрямом галузі, тоді як питання тестування, безпеки та управління дефектами залишались поза увагою більшості академічних робіт. Подальший розвиток привів до появи класифікаційних фреймворків: Pahl зі співавторами запропонували систематизацію технологій IaC за чотирма вимірами (контекстом застосування, функціональністю, мовою опису та архітектурою) і на її основі побудували спеціалізований DevOps-життєвий цикл для інфраструктурного коду [49,51,52,71]. Цей цикл охоплює стадії від написання та валідації коду до самовідновлення середовища після виявлених відхилень, чим суттєво розширює класичне розуміння управління конфігурацією.

Паралельно з таксономічними дослідженнями розвивались концептуальні альтернативи IaC. Aviv зі співавторами запропонували парадигму «Infrastructure from Code» (IfC), за якої ресурси хмарного середовища генеруються автоматично безпосередньо з вихідного коду застосунку, повністю виключаючи ручне написання інфраструктурних описів [8]. Ця концепція відображає загальну тенденцію до підвищення рівня абстракції: якщо IaC вимагає від інженера явно описати бажаний стан ресурсів, то IfC делегує цей опис інструментам статичного аналізу та кодогенерації. Водночас Sokolowski зі співавторами дослідили децентралізований підхід до IaC, де розгортання координується через явні контракти між незалежними командами без необхідності централізованої синхронізації, що вирішує проблему організаційних залежностей у великих розподілених системах [68].

Попри різні запропоновані концепції, у науковій спільноті зберігається консенсус щодо базових властивостей якісного інфраструктурного коду. До них відносять декларативність як спосіб опису бажаного стану без вказівки послідовності

дій, ідемпотентність як властивість, що гарантує однаковий кінцевий результат незалежно від кількості виконань, та відтворюваність як здатність гарантовано відновити ідентичне середовище у будь-який момент часу [41]. Саме ці властивості визначають придатність IaC-підходу для управління множиною середовищ у промислових умовах і формують теоретичне підґрунтя для розроблення узагальнених методів автоматизації життєвого циклу Terraform-інфраструктури.

Різноманіття інструментів автоматизації інфраструктури зумовлює необхідність їхньої систематизації за чіткими критеріями. Найбільш повну класифікаційну схему запропонували Pahl зі співавторами, які виокремили чотири виміри порівняння: контекст застосування (хмарна, гібридна або локальна інфраструктура), функціональність (забезпечення ресурсів, управління конфігурацією або оркестрація), мова опису (декларативна, процедурна або змішана) та архітектура виконання (агентна або безагентна, pull- або push-модель) [51, 52]. Ця схема дозволяє не лише порівнювати наявні рішення, а й виявляти прогалини у функціональному покритті окремих класів задач.

За критерієм парадигми опису конфігурації інструменти IaC поділяються на дві принципово відмінні групи. Декларативні інструменти, до яких належать Terraform, AWS CloudFormation та TOSCA-сумісні рішення, описують бажаний кінцевий стан інфраструктури, залишаючи вибір послідовності операцій на розсуд виконавчого рушія [61]. Процедурні інструменти, зокрема Ansible, Chef та Puppet, натомість вимагають явного задання кроків виконання, що забезпечує більший контроль над порядком операцій, але ускладнює досягнення ідемпотентності [25]. Контрольоване порівняльне дослідження за участю 67 фахівців підтвердило, що декларативний підхід у поєднанні з модельно-орієнтованим інструментарієм значно підвищує точність опису ресурсів та прискорює процес забезпечення середовищ [61]. Систематичний огляд сірої літератури виявив, що найбільш стійкими практиками незалежно від використовуваного інструменту залишаються декомпозиція конфігурацій на модулі, розділення змінних та логіки, а також мінімізація

стану у виконуваному коді [41].

Terraform займає особливе місце у цій класифікації завдяки поєднанню декларативної парадигми з провайдеро-незалежною архітектурою. Інструмент реалізує власну мову опису HashiCorp Configuration Language (HCL), яка забезпечує описову виразність для ресурсів будь-якого хмарного провайдера через єдину абстракцію [40, 72]. Ключовою архітектурною особливістю є явне управління станом через файл `terraform.tfstate`, який зберігає поточне відображення між описаними ресурсами та реальними об'єктами у хмарі. Цей механізм відрізняє Terraform від інших IaC-інструментів та визначає специфіку управління його життєвим циклом: будь-яка операція над інфраструктурою потребує узгодженості між кодом, станом та фактичним станом хмарних ресурсів [32]. Порівняльний аналіз продуктивності показав, що Terraform завершує операції забезпечення ресурсів у мультихмарних сценаріях на 44% швидше від альтернативних рішень завдяки ефективному обчисленню графу залежностей та дельта-змін між станами [16].

Позиціонування Terraform серед інструментів IaC, однак, не залишається незмінним. У 2023 році компанія HashiCorp змінила ліцензійну модель з відкритої MPL 2.0 на комерційну BSL, що призвело до появи форку OpenTofu під егідою Linux Foundation [47]. Обидва проєкти зберігають функціональну сумісність на рівні мови HCL та провайдерів, проте OpenTofu розвиває низку відсутніх у комерційній версії можливостей, зокрема криптографічне шифрування файлів стану на рівні ядра інструменту. Це розгалуження екосистеми є важливим контекстом для дослідження методів управління життєвим циклом, оскільки узагальнений метод має враховувати можливість застосування до обох реалізацій однієї і тієї ж архітектурної основи.

Wurster зі співавторами дослідили тринадцять провідних технологій автоматизації розгортання та показали, що попри відмінності у синтаксисі більшість інструментів оперують спільним набором концептів, формалізованих у вигляді технологічно-нейтральної метамоделі EDMM [80]. Weerasiri зі співавторами уто-

чнюють розподіл відповідальності: Terraform функціонує на рівні забезпечення платформи, тоді як Kubernetes та Nomad — на рівні оркестрації робочих навантажень поверх неї [60, 78].

Емпіричні дослідження зафіксували системні проблеми, що не вирішуються на рівні конкретного інструменту: фрагментацію екосистеми, відсутність зрілих засобів тестування та брак стандартизованих практик управління станом [25, 55], що підтверджує необхідність методологічного підходу до управління Terraform-інфраструктурою для кількох середовищ.

Поняття життєвого циклу інфраструктури як коду не має єдиного загальноприйнятого визначення, проте більшість дослідників сходяться на тому, що він охоплює послідовні стадії від написання та валідації конфігураційних файлів до їхнього застосування, моніторингу фактичного стану середовища та адаптації у відповідь на виявлені відхилення [43, 51]. На відміну від життєвого циклу програмного забезпечення, де артефактом є виконуваний код, у випадку IaC артефакт є одночасно описом бажаного стану та інструкцією для його досягнення, що породжує специфічні вимоги до кожної стадії.

Центральною властивістю, що пронизує всі стадії цього циклу, є ідемпотентність — гарантія того, що повторне виконання одного й того самого опису конфігурації не спричинить небажаних змін у вже відповідному цільовому середовищі [10,26,41]. Формальний аналіз збіжності декларативних скриптів через графи переходів станів підтвердив, що порушення ідемпотентності є прихованим джерелом помилок, які виявляються лише при повторному застосуванні конфігурацій або при зміні порядку виконання операцій [26]. Тісно пов'язаною властивістю є відтворюваність: здатність гарантовано відновити ідентичне середовище з одного й того самого коду незалежно від моменту часу та цільового провайдера [41]. Саме відтворюваність робить IaC практично корисним для управління кількома середовищами, оскільки дозволяє розглядати середовища розробки, тестування та виробництва як параметризовані екземпляри одного опису.

Емпіричні дослідження еволюції інфраструктурних репозиторіїв засвідчили, що файли конфігурацій зазнають значно частіших змін, ніж прикладний код, і тісно пов'язані з еволюцією тестів та виробничого коду [36]. Це підтверджує, що управління IaC-артефактами не може бути відокремлено від загального процесу розробки: зміни в інфраструктурі мають синхронізуватися зі змінами у коді застосунку, а конвеєр доставки повинен охоплювати обидва типи артефактів [1]. Ozkaya формалізувала цей зв'язок через поняття структур розподілу в програмній архітектурі, де IaC-файли є безпосередньою реалізацією архітектурних рішень щодо розміщення компонентів у хмарному середовищі [48]. Порушення відповідності між архітектурою та її реалізацією в IaC призводить до конфігураційного дрейфу — стану, коли фактичні ресурси у хмарі відхиляються від описаних у коді, що є однією з ключових проблем, які покликаний вирішувати метод, що розробляється у цій роботі [51].

Розвиток хмарних обчислень призвів до формування гібридних та мультихмарних архітектур, у яких організації одночасно використовують ресурси кількох провайдерів з метою уникнення залежності від одного постачальника, оптимізації витрат та підвищення відмовостійкості [6]. За оцінками дослідників, максимізація цінності від гетерогенних хмарних екосистем вимагає централізованих платформ управління та стандартизованого інструментарію, оскільки фрагментація операційних процесів між провайдерами суттєво збільшує операційне навантаження та розширює поверхню атак [6, 13, 17]. Саме у цьому контексті Terraform набуває стратегічної ролі єдиного рівня абстракції над різнорідними API хмарних провайдерів.

Систематичний огляд фреймворків оркестрації хмарних ресурсів виявив, що переважна більшість існуючих рішень обмежена або одним провайдером, або власною мовою опису, яка потребує значних зусиль для адаптації [76]. Terraform вирізняється серед них провайдеро-незалежною архітектурою: емпіричне дослідження розгортання ідентичних робочих навантажень на AWS, Azure та GCP підтвердило

зниження часу забезпечення ресурсів на 30% та скорочення помилок конфігурації на 25% порівняно з вбудованими інструментами кожного провайдера [42]. Водночас мультихмарне застосування загострює проблему управління станом: при розподіленій роботі команд над спільними конфігураціями файл стану стає вузьким місцем, а його пошкодження або десинхронізація між середовищами можуть призвести до некоректного відображення реального стану ресурсів [6, 24, 42].

Контейнеризація та мікросервісна архітектура стали додатковим чинником ускладнення мультихмарних середовищ. Pahl зі співавторами показали, що повністю контейнеризовані мікросервісні архітектури є практично реалізованими навіть на периферійних обчислювальних кластерах з обмеженими ресурсами, проте для цього необхідне поєднання IaC-інструментів забезпечення платформи з контролерами зворотного зв'язку для динамічного управління навантаженням [50]. Безсерверні обчислення, у свою чергу, переміщують межу відповідальності IaC ще далі: у цій парадигмі Terraform управляє не лише мережевою та обчислювальною інфраструктурою, а й конфігурацією подієвих тригерів та дозволів, що робить повноту та точність описів критично важливою [9].

Спільним знаменником усіх розглянутих сценаріїв є потреба у методі, який забезпечував би узгоджене управління конфігураціями в умовах множини середовищ, провайдерів та команд. Таксономія методів оркестрації вказує на відкритість питань семантичної інтероперабельності та автономної адаптації як головних бар'єрів на шляху до повної автоматизації [78]. Ці спостереження безпосередньо мотивують дослідження, що проводиться у даній роботі: узагальнений метод управління життєвим циклом Terraform-інфраструктури має адресувати не лише технічні аспекти забезпечення ресурсів, а й організаційні та архітектурні передумови масштабованої автоматизації у мультисередовищних конфігураціях.

Порівняльний аналіз підходів та інструментів автоматизації інфраструктури наведено у таблиці 1.1 (с. 16).

Таблиця 1.1 – Порівняльний аналіз підходів та інструментів автоматизації інфраструктури

Інструмент	Парадигма	Управління станом	Мульти-хмарна підтримка	Ліцензія	Основне обмеження
Terraform (HashiCorp)	Декларативна	Явний файл стану (`.tfstate`), віддалений бекенд	Повна (AWS, Azure, GCP та ін.)	BSL 1.1 (з 2023)	Файл стану як єдина точка відмови; складність при паралельному доступі [42]
OpenTofu	Декларативна	Явний файл стану, вбудоване шифрування	Повна, сумісна з Terraform	MPL 2.0 (відкрита)	Молода екосистема; менша кількість перевірених провайдерів [47]
Ansible	Процедурна (з декларативними елементами)	Відсутнє (безстановий)	Часткова (через модулі хмарних провайдерів)	Apache 2.0 (відкрита)	Складність досягнення ідемпотентності; не призначений для забезпечення ресурсів [41]
Puppet / Chef	Декларативна (DSL)	Агентне відстеження стану вузлів	Обмежена (переважно управління конфігурацією ОС)	Apache 2.0 / Apache 2.0	Вимагає агентів на цільових вузлах; не охоплює хмарні ресурси [25]

Кінець таблиці 1.1

Інструмент	Парадигма	Управління станом	Мульти-хмарна підтримка	Ліцензія	Основне обмеження
AWS CloudFormation	Декларативна (JSON, YAML)	Стеки як одиниця стану, керовані AWS	Відсутня (лише AWS)	Пропрієтарна	Прив'язка до одного провайдера; обмежена переносимість [16]
TOSCA / EDMM	Декларативна (стандарт)	Залежить від оркестратора	Повна (через абстракцію стандарту)	Відкритий стандарт OASIS	Висока складність моделювання; обмежена підтримка в існуючих інструментах [45, 80]
Infrastructure from Code (IfC)	Автогенерація з коду застосунку	Делегується платформі	Залежить від реалізації	Varies	Рання стадія; не підходить для складних кастомних топологій [8]

1.2 Методи та практики забезпечення якості, безпеки та автоматизації управління Terraform-інфраструктурою для кількох середовищ

Управління кількома середовищами є однією з найбільш практично значущих задач у контексті Terraform-інфраструктури. Під середовищем у даній роботі розуміється ізольований набір хмарних ресурсів, що відповідає певному етапу життєвого циклу програмного продукту — як правило, це середовища розробки (dev), тестування (staging) та виробництво (production), хоча реальні організації часто оперують більшою кількістю оточень [15]. Ключовою вимогою до будь-якого

методу управління є забезпечення достатнього рівня ізоляції між середовищами при одночасному уникненні дублювання інфраструктурного коду.

Вбудованим механізмом Terraform для роботи з кількома середовищами є робочі простори (workspaces), що дозволяють підтримувати окремі файли стану в межах одного кореневого модуля через перемикання контексту виконання. Порівняльний аналіз засвідчив, що workspaces є ефективними для простих конфігурацій з мінімальними відмінностями між середовищами, проте демонструють суттєві обмеження при необхідності різних топологій ресурсів або різних прав доступу для кожного оточення [62]. Зокрема, спільне використання одного кореневого модуля для всіх середовищ підвищує ризик ненавмисного застосування змін до виробничого оточення при помилці оператора. Альтернативний підхід на основі ієрархічної структури каталогів, де кожне середовище має власний незалежний кореневий модуль та окремий файл стану, забезпечує повну ізоляцію та мінімізує так званий «радіус ураження» при помилкових змінах [15, 28, 29]. Недоліком цього підходу є необхідність дублювання викликів модулів між середовищами, що вирішується інструментами-обгортками, зокрема Terragrunt, через механізм успадкування конфігурацій [15].

Управління файлом стану у мультисередовищних конфігураціях є окремою архітектурною задачею. HashiCorp рекомендує зберігати файли стану у віддалених бекендах з підтримкою блокування для запобігання паралельним операціям запису, що може призвести до пошкодження стану. Сегментація стану за середовищами та функціональними компонентами інфраструктури дозволяє зменшити обсяг кожного окремого файлу та обмежити зону впливу потенційних помилок, проте збільшує кількість незалежних станів, що ускладнює відстеження залежностей між компонентами [29]. Емпіричне дослідження стратегій версіонування та відкату IaC-шаблонів у мультихмарних середовищах показало, що ієрархічна модель семантичного версіонування у поєднанні з автоматизованими конвеєрами валідації скорочує середній час відновлення після збійних розгортань з 47 до 4,7

хвилини [37].

GitOps як операційна модель управління інфраструктурою органічно доповнює директорійний підхід до ізоляції середовищ. За цією моделлю Git-репозиторій є єдиним джерелом істини для бажаного стану всіх середовищ, а автоматизований агент безперервно звіряє фактичний стан хмарних ресурсів із задекларованим у коді та усуває виявлені розбіжності [33,75]. Практична реалізація повного GitOps-конвеєра для Terraform-інфраструктури через GitLab CI та FluxCD підтвердила підвищення частоти розгортань на 35% та скорочення часу відновлення на 40% порівняно з традиційним підходом [33]. Важливо, що pull-модель синхронізації, яка є характерною для GitOps, виключає необхідність надання CI/CD-агентам прямого доступу до хмарних облікових даних, що суттєво зменшує поверхню атак у мультисередовищних конфігураціях [38].

Корпоративні платформи управління Terraform-робочими процесами забезпечують централізоване управління доступом до середовищ, ієрархічне успадкування змінних та автоматизовані шлюзи затвердження змін, проте їхня залежність від конкретного постачальника обмежує переносимість розроблених робочих процесів [62].

Забезпечення якості інфраструктурного коду є необхідною умовою надійного управління середовищами, однак тестування IaC залишається однією з найменш опрацьованих тем у академічній літературі попри його критичну роль у запобіганні збоям розгортання [56]. Специфіка полягає в тому, що виконання інфраструктурного коду невіддільне від взаємодії з реальними або емульованими хмарними API, що ускладнює ізоляцію тестованих компонентів [27].

Якісний аналіз практик тестування IaC на основі 50 публічних джерел дозволив виокремити шість основних категорій перевірок: статичний аналіз синтаксису та стилю, валідація плану виконання без застосування змін, тестування у пісочниці з реальними ресурсами, контрактне тестування модулів, інтеграційне тестування та тестування відповідності політикам [27]. У контексті Terraform ці категорії ре-

алізуються через вбудовані команди `terraform validate` та `terraform plan` на ранніх стадіях, а також через спеціалізовані фреймворки на кшталт Terratest для інтеграційного рівня. Важливим спостереженням є те, що статичний аналіз виявляє лише поверхневі дефекти, тоді як помилки логіки розгортання та порушення ідемпотентності виявляються лише при виконанні у реальному або максимально наближеному середовищі [34].

Дослідження дефектів в IaC-скриптах сформувало власну таксономію: Rahman зі співавторами на основі аналізу 1 448 комітів виокремили вісім категорій дефектів, де помилки конфігураційних даних є найпоширенішими, а порушення ідемпотентності — найбільш специфічними для інфраструктурного коду [58]. Sharma зі співавторами каталогізували «запахи» конфігураційного коду, встановивши кореляцію між щільністю архітектурних дефектів та складністю системи [53, 64], тоді як Dalla Palma зі співавторами розробили каталог із 46 метрик якості для кількісної оцінки підтримуваності конфігурацій, досягнувши AUC-PR 0,93 при прогнозуванні дефектів засобами машинного навчання [11, 18, 19]. Окремою проблемою є лінгвістичні антипатерни — розбіжності між назвою ресурсу та його фактичною логікою, виявлення яких через аналіз синтаксичних дерев є застосованим і до HCL-конфігурацій Terraform [14].

Конфігураційний дрейф, тобто відхилення фактичного стану хмарних ресурсів від задекларованого у коді, виникає внаслідок змін у хмарній консолі, неуспішних часткових застосувань або зовнішніх процесів поза межами Terraform [20]. Lima зі співавторами запропонували ML-метод виявлення дрейфу через аналіз ознак конфігурацій та файлів стану, здатний ідентифікувати аномалії з непрямих залежностей між ресурсами, проте такий що потребує значного обсягу розмічених даних [44]. Практичні платформи реалізують спрощений підхід через безперервне порівняння результатів `terraform plan` з очікуваним порожнім планом та автоматичне сповіщення про розбіжності [20].

Вразливості, закладені на етапі написання конфігурацій, безпосередньо ма-

теріалізуються у хмарному середовищі при їхньому застосуванні. Rahman зі співавторами ідентифікували сім типів «запахів безпеки» в IaC-скриптах, серед яких захардкоджені секрети є найпоширенішим дефектом — емпіричний аналіз 15 232 скриптів виявив, що подібні вразливості можуть залишатись невиправленими до восьми років [57]. Opdebeeck зі співавторами розширили статичний аналіз через поведінковий підхід на основі графів залежностей, що дозволяє виявляти складні вразливості з непрямих потоків даних між ресурсами [46].

Концепція «Shift Left» у застосуванні до безпеки IaC передбачає перенесення перевірок якомога ближче до моменту написання коду — у pre-commit хуки, локальні валідатори та ранні стадії CI/CD конвеєру [67]. Це дозволяє виявляти небезпечні конфігурації до того, як вони потраплять до спільного репозиторію або будуть застосовані до будь-якого середовища. Alonso зі співавторами запропонували комплексний фреймворк PIACERE, що реалізує цей принцип через мову моделювання DOML і забезпечує автоматичну статичну перевірку безпеки, самовідновлення середовища та уніфікацію процесів забезпечення і моніторингу в єдиному DevSecOps-циклі [5]. Haverinen зі співавторами розробили відкриту інформаційну модель Cyberismo, що представляє вимоги відповідності у вигляді декларативних логічних правил та дозволяє автоматично збирати докази виконання цих вимог безпосередньо з артефактів CI/CD конвеєру [30].

Підхід «відповідність як код» (Compliance-as-Code) узагальнює принципи безпеки до рівня організаційних та регуляторних вимог. Agarwal зі співавторами запропонували стандартизовану архітектуру на основі формату OSCAL для перетворення регуляторних вимог у машиночитний код, що інтегрується з Git як єдиним джерелом істини та дозволяє перевіряти відповідність Terraform-конфігурацій регуляторним стандартам ще до їхнього розгортання [3]. Thompson зі співавторами реалізували автоматизований фреймворк відповідності на базі Kubernetes, що скорочує цикл від виявлення порушення до його усунення з 48 годин до 5 хвилин через декларативне управління політиками та автоматизоване виправлення дрей-

фу конфігурацій [74]. Sharma зі співавторами продемонстрували, що поєднання Terraform з платформою управління Red Hat Satellite у єдиному шарі policy-as-code дозволяє досягти 78% скорочення конфігураційного дрейфу та 92% покращення показника відповідності в гібридних мультихмарних середовищах [65].

Практична реалізація безпеки у Terraform-конвеєрах потребує інтеграції кількох рівнів перевірок. На рівні вихідного коду застосовуються статичні аналізатори на кшталт tfsec та Checkov, що перевіряють HCL-конфігурації на відповідність базам правил безпеки [67]. На рівні артефактів розгортання актуальним є сканування контейнерних образів на вразливості з інтеграцією у реєстр артефактів, що унеможливлює розгортання невідповідних образів до виробничого середовища [23]. Silva зі співавторами показали, що автоматизований монітор відповідності, побудований на фреймворках CIS та NIST з використанням кількісної метрики верифікації вузлів, здатен задовольнити 90,47% умов безпеки у виробничих середовищах без ручного втручання [66]. Попри високу ефективність окремих інструментів статичного аналізу, порівняльне дослідження дев'яти сканерів вразливостей виявило суттєві розбіжності у виявлених дефектах між ними, що свідчить про необхідність багаторівневого підходу до автоматизованої перевірки безпеки [35].

Автоматизація конвеєрів доставки інфраструктурного коду є організаційним фундаментом, на якому базуються всі розглянуті практики якості та безпеки. Без інтеграції у відтворюваний конвеєр навіть найретельніше розроблені механізми тестування та перевірки відповідності залишаються факультативними і залежними від дисципліни окремих розробників. Parnin зі співавторами на основі аналізу практик безперервного розгортання у великих технологічних компаніях сформулювали десять ключових принципів, серед яких центральне місце займають автоматизована перевірка кожної зміни та ізоляція середовищ як механізм обмеження ризиків при поступовому просуванні змін [54]. Ці принципи безпосередньо застосовні до конвеєрів управління Terraform-інфраструктурою і визначають їхню

типову архітектуру.

Типовий CI/CD конвеєр для Terraform-інфраструктури включає послідовні стадії форматування та синтаксичної валідації, статичного аналізу безпеки, генерації плану змін, його ручного затвердження для виробничого середовища та застосування [22]. Romm зі співавторами продемонстрували, що інтеграція Terraform з Kubernetes через спеціалізований провайдер у межах єдиного конвеєра дозволяє забезпечити узгоджене управління як хмарними ресурсами, так і навантаженнями всередині кластера, скоротивши час розгортання нових середовищ вдвічі та значно зменшивши кількість ручних операцій при масштабуванні [60].

Управління модулями та реєстрами є ключовим аспектом корпоративних Terraform-конвеєрів. Приватний реєстр модулів дозволяє стандартизувати перевірені конфігурації ресурсів та забезпечити їхнє повторне використання між командами без ризику несанкціонованих модифікацій [73]. Шляхи управління рівня підприємства, зокрема governance gates (автоматизовані шлюзи, що перевіряють відповідність плану змін набору організаційних політик перед його застосуванням), забезпечують централізований контроль над усіма середовищами без необхідності ручного перегляду кожної зміни [65, 73]. Дослідження масштабованих підходів до управління витратами на хмарні ресурси виявило, що IaC-конфігурації рідко містять явні анотації вартості, що унеможливлює автоматизований аналіз фінансових наслідків пропонованих змін на стадії планування [21]. Включення оцінки вартості як окремої стадії конвеєру є перспективним напрямом, що доповнює традиційні перевірки якості та безпеки.

Перспективним напрямом є генерація Terraform-конфігурацій засобами обробки природної мови: система Sharma зі співавторами, що перетворює архітектурні діаграми на валідний HCL-код, скоротила час забезпечення нових середовищ на 60%, проте потребує додаткової верифікації для складних топологій [63]. Holkuziev та Adelusi підкреслюють, що зрілість процесу управління Terraform-інфраструктурою визначається насамперед стандартизованими практиками мо-

дульної декомпозиції, версіонування та документування як передумовою довгострокової підтримуваності [2, 32].

Методи управління Terraform-інфраструктурою для кількох середовищ систематизовано у таблиці 1.2 (с. 24).

Таблиця 1.2 – Методи управління Terraform-інфраструктурою для кількох середовищ

Метод / підхід	Ізоляція середовищ	Виявлення дрейфу	CI/CD інтеграція	Security-as-Code	Основне обмеження
Terraform Workspaces	Часткова (спільний модуль)	Вбудований `plan`	Повна	Відсутня нативно	Високий ризик міжсередовищного впливу; не підходить для різних топологій [62]
Директорійна ієрархія (per-env root module)	Повна (окремий стан)	Вбудований `plan`	Повна	Відсутня нативно	Дублювання викликів модулів; потребує інструментів-обгорток [15]

Продовження таблиці 1.2

Метод / підхід	Ізоляція середовищ	Виявлення дрейфу	CI/CD інтеграція	Security-as-Code	Основне обмеження
Terragrunt (ієрархічне успадкування)	Повна	Вбудований `plan`	Повна	Відсутня нативно	Додаткова залежність поза екосистемою Terraform [15]
GitOps (pull-модель)	Повна (per-env репозиторій або гілка)	Безперервна автоматична	Нативна (Git як джерело істини)	Часткова (через policy gates)	Складність налагодження; потребує окремого оператора синхронізації [33, 38]
ML-based drift detection	Не визначає	Розширена (непрямі залежності)	Інтегрується як окрема стадія	Відсутня	Потребує розмічених даних; складна адаптація до нових провайдерів [44]

Продовження таблиці 1.2

Метод / підхід	Ізоляція середовищ	Виявлення дрейфу	CI/CD інтеграція	Security-as-Code	Основне обмеження
Shift Left / статичний аналіз (tfsec, Checkov)	Не визначає	Відсутня	Pre-commit та CI стадії	Повна (синтаксичний рівень)	Не виявляє логічних помилок розгортання та порушень ідемпотентності [67]
Compliance-as-Code (OSCAL, policy gates)	Не визначає	Частково (через drift від політик)	Повна	Повна (регуляторний рівень)	Висока складність моделювання вимог; потребує підтримки стандартів [3, 65]
Приватний реєстр модулів + governance gates	Непряма (через стандартизацію)	Відсутня	Повна	Частково (через затвердження)	Залежність від корпоративної платформи; обмежена переносимість [73]

1.3 Постановка задачі дослідження

Парадигма «Everything as Code» сформувала теоретичне підґрунтя для розгляду інфраструктури як першокласного артефакту розробки, що підлягає версіонуванню, тестуванню та рецензуванню нарівні з прикладним кодом. Terraform посідає центральне місце серед інструментів цього класу завдяки декларативній парадигмі, провайдеро-незалежній архітектурі та явному механізму управління станом. Аналіз існуючих методів виявив відсутність єдиного підходу, що одночасно забезпечував би повну ізоляцію середовищ, автоматизоване виявлення дрейфу та інтеграцію перевірок безпеки у конвеєр доставки.

На підставі виявлених прогалин визначено такі задачі дослідження:

1. Проаналізувати існуючі патерни ізоляції середовищ у Terraform-інфраструктурі та обґрунтувати критерії вибору між ними залежно від вимог до рівня ізоляції, масштабу команди та складності топології ресурсів.
2. Розробити модель життєвого циклу Terraform-інфраструктури для кількох середовищ, що охоплює стадії від ініціалізації конфігурацій до виявлення та усунення конфігураційного дрейфу.
3. Обґрунтувати метод управління станом та версіонування конфігурацій у мультисередовищних Terraform-проєктах, що мінімізує радіус ураження при помилкових змінах, забезпечує відкат до попереднього стану та інтегрує перевірки якості і безпеки у конвеєр просування змін між середовищами.
4. Реалізувати запропонований узагальнений метод на реальній мультисередовищній Terraform-інфраструктурі та оцінити його ефективність за визначеними показниками якості.

2 УЗАГАЛЬНЕНИЙ МЕТОД КЕРУВАННЯ ЖИТТЄВИМ ЦИКЛОМ TERRAFORM-ІНФРАСТРУКТУРИ ДЛЯ КІЛЬКОХ СЕРЕДОВИЩ

2.1 Концептуальна модель та архітектурні принципи методу

Інженерні дисципліни, що досягли зрілості, відрізняються наявністю усталеного понятійного апарату та формалізованих моделей, які дозволяють однозначно описувати предмет діяльності, відокремлювати сутнісні властивості від випадкових і передавати знання між поколіннями фахівців без втрати сенсу. Управління інфраструктурою як код перебуває в ранній фазі становлення такої дисципліни: значна частина практичних рішень формується емпірично, в межах окремих організацій, і транслюється у спільноту переважно через публічні технічні блоги та доповіді на галузевих конференціях. Хоча такий спосіб накопичення знань забезпечує швидкий обмін досвідом, він має суттєві обмеження для побудови узагальнених методів, оскільки кожне описане рішення пов'язане з конкретним інструментом, конкретною хмарною платформою та конкретною організаційною структурою, в якій воно народилося.

Перехід від опису окремих інженерних рішень до узагальненого методу вимагає формалізації, що абстрагується від специфіки конкретних реалізацій. Така формалізація має описувати об'єкти управління, відношення між ними та операції, які над ними виконуються, мовою, незалежною від конкретного інструменту чи платформи. Важливою властивістю формального апарату є його здатність зберігати сенс при застосуванні до різних середовищ виконання: якщо модель описує життєвий цикл інфраструктури в термінах стадій, переходів та умов відповідності, ці поняття мають однаково застосовуватись до інфраструктури на AWS, Azure, Google Cloud або приватній хмарі, побудованій на базі внутрішні середовища віртуалізації. Інваріантність моделі щодо обчислювальної платформи є одним з критеріїв її придатності як основи для узагальненого методу.

Другою важливою властивістю формального апарату є повнота покриття життєвого циклу. Історично в академічній літературі з управління інфраструктурою як код основна увага приділялась стадіям створення та застосування конфігурацій, тоді як питання еволюції конфігурацій у часі, просування змін між середовищами та відновлення після невдалих застосувань залишались поза межами моделей. Наслідком такого фокусу стала ситуація, коли інструменти забезпечують початкове розгортання інфраструктури, але подальше управління її життєвим циклом відбувається без системної методологічної підтримки. Розроблення узагальненого методу має долати цю асиметрію, охоплюючи весь спектр операцій від першого застосування конфігурації до виведення середовища з експлуатації.

Третім аспектом є розмежування рівнів абстракції. Практика показує, що ефективне управління інфраструктурою вимагає одночасного оперування кількома рівнями: на рівні окремого ресурсу розглядаються його атрибути та залежності, на рівні модуля — групування ресурсів з єдиним інтерфейсом, на рівні середовища — сукупність модулів, що формує ізольований простір виконання, на рівні організації — множина середовищ з відношеннями підпорядкування та просування. Формальний апарат має надавати виразні засоби для опису кожного з цих рівнів та переходів між ними, не змішуючи поняття різних рівнів в одному описі. Відсутність такого розмежування у наявних підходах призводить до того, що одні й ті самі механізми застосовуються для управління об'єктами різної природи, що ускладнює аналіз властивостей системи в цілому.

Розроблення узагальненого методу потребує формального апарату, що однозначно описує об'єкти управління та відношення між ними. Аналіз у розділі 1 засвідчив, що існуючі підходи описують окремі аспекти (ізоляцію середовищ, виявлення дрейфу, перевірку відповідності) без єдиної понятійної бази. Нижче вводяться означення, на яких ґрунтуватиметься модель життєвого циклу та архітектурні принципи методу.

Нехай $\mathcal{M}$ — скінченна множина модулів Terraform, де кожен модуль $m =$

$\langle R_m, V_m, O_m \rangle$ визначається множиною описів ресурсів R_m, вхідних змінних V_m та вихідних значень O_m. Кореневий модуль $m_{\text{root}} \in \mathcal{M}$ є точкою входу для виконання та агрегує виклики інших модулів. Інфраструктурна конфігурація:

$$C = \langle m_{\text{root}},\ \Theta \rangle, \tag{2.1}$$

де $\Theta = \{\theta_1, \theta_2, \ldots, \theta_k\}$ — набір значень вхідних змінних кореневого модуля, що параметризує конкретний екземпляр інфраструктури. Саме Θ визначає відмінності між середовищами при спільному кореневому модулі.

Середовище як центральна одиниця управління визначається четвіркою:

$$e = \langle \text{id}_e,\ C_e,\ S_e,\ P_e \rangle, \tag{2.2}$$

де id_e — унікальний ідентифікатор;
C_e — інфраструктурна конфігурація згідно з (2.1);
S_e — поточний стан;
P_e — набір політик відповідності, що визначають допустимі конфігурації.

Включення P_e у визначення середовища відображає принцип вбудованої відповідності: політики є невід'ємним атрибутом середовища, а не факультативним зовнішнім обмеженням.

Поняття стану потребує розмежування трьох сутностей. Бажаний стан $S_e^d = \text{eval}(C_e)$ — множина ресурсів та атрибутів, що визначається інтерпретацією конфігурації виконавчим рушієм Terraform. Записаний стан S_e^r — вміст файлу `terraform.tfstate`, що оновлюється після кожної успішної операції застосування. Фактичний стан S_e^a — множина ресурсів, що реально існують у хмарному середовищі на момент часу t і можуть відрізнятись від S_e^r внаслідок змін поза Terraform.

Ця тричленна модель дозволяє формально визначити ключові операції.

Дельта планування відповідає набору змін операції `plan`:

$$\Delta_e^{\text{plan}} = S_e^d \setminus S_e^r. \tag{2.3}$$

Дельта дрейфу відповідає конфігураційному дрейфу, виявленому операцією `refresh`:

$$\Delta_e^{\text{drift}} = S_e^r \setminus S_e^a. \tag{2.4}$$

Середовище є консистентним, якщо $\Delta_e^{\text{plan}} = \varnothing$ та $\Delta_e^{\text{drift}} = \varnothing$, тобто всі три стани тотожні. Це узгоджується з формальним визначенням ідемпотентності [26]: повторне виконання конфігурації у консистентному середовищі не породжує жодних змін.

Множина середовищ утворює простір середовищ $\mathcal{E} = \{e_1, e_2, \ldots, e_n\}$ з відношенням часткового порядку просування $\preceq$. Для типової конфігурації $e_{\text{dev}} \preceq e_{\text{staging}} \preceq e_{\text{prod}}$ зміна потрапляє до виробничого середовища лише після проходження через усі проміжні. Відношення не обов'язково є лінійним: паралельні гілки для різних регіонів або клієнтів описуються частково впорядкованою множиною $(\mathcal{E}, \preceq)$.

Операція просування між суміжними середовищами $e_i \preceq e_j$ визначається як:

$$\text{promote}(e_i, e_j) : C_{e_j} \leftarrow \langle m_{\text{root}},\ \Theta_{e_j} \rangle, \tag{2.5}$$

де кореневий модуль залишається спільним, а Θ_{e_j} відображає специфіку цільового середовища. Просування не є копіюванням стану: кожне середовище має незалежний S_{e_j}, і застосування конфігурації проходить повний цикл планування, валідації та затвердження, що забезпечує мінімізацію радіуса ураження відповідно до задачі 3.

У підрозділі 1.2 було ідентифіковано три базові патерни ізоляції середовищ у Terraform-інфраструктурі: на основі робочих просторів (workspace-based), на основі директорійної ієрархії (directory-based) та на основі гілок репозиторію

(branch-based). Кожен з них реалізує різний компроміс між рівнем ізоляції та складністю супроводу. Для обґрунтованого вибору базового патерну методу необхідно формалізувати критерії порівняння та оцінити кожен патерн за ними.

У термінах введеного формалізму патерни відрізняються способом відображення простору середовищ $(\mathcal{E}, \preceq)$ на структуру артефактів Terraform. Визначимо п'ять критеріїв оцінки.

Ізоляція стану I_S — ступінь незалежності файлів стану між середовищами. Повна ізоляція ($I_S = 1$) означає, що кожне $e \in \mathcal{E}$ має окремий файл стану S_e^r з незалежним бекендом та окремими обліковими даними доступу; часткова ізоляція ($I_S = 0{,}5$) — окремі файли стану у спільному бекенді; відсутня ($I_S = 0$) — спільний файл стану. Радіус ураження B_R — максимальна кількість середовищ, на які може вплинути помилкова операція `apply`. Формально $B_R = |\{e \in \mathcal{E} : S_e^a \text{ може бути змінений}\}|$, де менше значення є кращим. Конфігураційна ентропія H_C — міра дублювання коду між середовищами, визначена як відношення обсягу середовищно-специфічного коду до загального обсягу конфігурації; менше значення свідчить про вищий рівень повторного використання. Складність просування K_P — кількість ручних або автоматизованих кроків, необхідних для виконання операції promote(e_i, e_j) згідно з (2.5). Масштабованість M — залежність операційного навантаження від кількості середовищ $|\mathcal{E}|$: лінійна ($M = O(n)$), сублінійна ($M = O(\log n)$) або константна ($M = O(1)$).

Workspace-based патерн використовує вбудований механізм робочих просторів Terraform, де єдиний кореневий модуль m_{root} обслуговує всі середовища через перемикання контексту командою `terraform workspace select`. Кожен робочий простір отримує окремий файл стану, проте в межах спільного бекенду та з єдиним набором облікових даних. За введеними критеріями: $I_S = 0{,}5$, оскільки стани ізольовані логічно, але не фізично; $B_R = 1$ за нормальних умов, проте помилка у спільному бекенді або обліковому записі може вплинути на всі середовища одночасно; H_C є мінімальною, оскільки код не дублюється; K_P є низькою

— просування зводиться до перемикання простору та застосування. Критичним обмеженням є неможливість забезпечити різні топології ресурсів або різні рівні доступу для окремих середовищ без умовної логіки в коді, що погіршує читабельність та підвищує ймовірність помилок [62].

Directory-based патерн виділяє кожному середовищу окремий кореневий модуль у власному каталозі файлової системи, що викликає спільні дочірні модулі з $\mathcal{M}$ із середовищно-специфічними параметрами Θ_e. Формально кожне середовище отримує незалежну конфігурацію $C_e = \langle m^e_{\text{root}},\ \Theta_e \rangle$, де m^e_{root} є тонкою обгорткою над спільними модулями. За критеріями: $I_S = 1$, оскільки кожне середовище має фізично окремий стан, бекенд та облікові дані; $B_R = 1$ у строгому сенсі — помилкова операція впливає лише на одне середовище; H_C є помірною — дублюються виклики модулів, але не їхня логіка; K_P є вищою, оскільки просування потребує оновлення версії модуля у цільовому каталозі. Інструменти-обгортки на кшталт Terragrunt зменшують H_C через успадкування конфігурацій, проте вводять зовнішню залежність [15].

Branch-based патерн відображає середовища на гілки Git-репозиторію: гілка `main` відповідає виробничому середовищу, `staging` — передвиробничому тощо. Просування реалізується через операцію злиття гілок (merge). За критеріями: $I_S = 1$, оскільки кожна гілка має незалежний стан; $B_R = 1$; проте H_C є найвищою — весь код дублюється між гілками, а конфлікти злиття при розходженні гілок суттєво ускладнюють супровід. K_P формально низька (одна операція merge), але практична складність розв'язання конфліктів робить цей патерн непридатним для інфраструктур зі значною кількістю середовищ [15, 33].

Аналіз таблиці 2.1 обґрунтовує вибір directory-based патерну як базового для запропонованого методу. Цей патерн є єдиним, що забезпечує повну ізоляцію стану ($I_S = 1$) та гарантований одиничний радіус ураження ($B_R = 1$) без додаткових умов. Основним недоліком патерну є помірна конфігураційна ентропія та лінійна масштабованість. Цей недолік компенсується архітектурним рішенням,

Таблиця 2.1 – Порівняльна оцінка патернів ізоляції середовищ

Патерн	I_S	B_R	H_C	K_P	M
Workspace-based	0,5	1*	Мін.	Низька	$O(1)$
Directory-based	1	1	Помірна	Середня	$O(n)$
Branch-based	1	1	Висока	Низька**	$O(n)$

* — за умови ізоляції бекенду; за спільного бекенду B_R може дорівнювати $|\mathcal{E}|$.
** — без урахування складності розв'язання конфліктів злиття.

закладеним у формалізації: спільні дочірні модулі з $\mathcal{M}$ інкапсулюють усю логіку ресурсів, а кореневі модулі середовищ зводяться до параметризованих викликів, що мінімізує дублювання. Workspace-based патерн, попри мінімальну ентропію, не задовольняє вимогу повної ізоляції, критичну для виробничих середовищ. Branch-based патерн має найвищу ентропію та практично непридатний для масштабування понад два-три середовища.

На основі формалізованих понять та обраного патерну ізоляції пропонується модель життєвого циклу Terraform-інфраструктури для кількох середовищ, що є відповіддю на задачу дослідження 2. Модель описує дві ортогональні проекції: горизонтальну — послідовність стадій у межах окремого середовища $e \in \mathcal{E}$, та вертикальну — просування змін між середовищами через $(\mathcal{E}, \preceq)$. Формалізація виконана у вигляді скінченного автомата з явними умовами переходів.

2.1.1 Горизонтальна проекція: життєвий цикл окремого середовища

Нехай для середовища e множина стадій $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_8\}$ визначається таким чином:

– σ_1 — ініціалізація (Init): створення кореневого модуля m^e_{root}, налаштування віддаленого бекенду для S^r_e та ініціалізація провайдерів командою `terraform init`. Вхідний артефакт — шаблон конфігурації; вихідний — ініціалізований робочий каталог із завантаженими залежностями;

– σ_2 — авторство (Author): внесення змін до конфігурації C_e, що модифікує бажаний стан S_e^d. Зміни відбуваються у системі контролю версій через механізм pull request, що забезпечує рецензування коду до злиття;

– σ_3 — валідація (Validate): автоматизована перевірка синтаксичної коректності та внутрішньої узгодженості конфігурації засобами `terraform validate`, а також статичний аналіз безпеки та відповідності політикам P_e. Формально стадія перевіряє предикат $\text{valid}(C_e, P_e) = \text{true}$;

– σ_4 — планування (Plan): обчислення дельти $\Delta_e^{\text{plan}} = S_e^d \setminus S_e^r$ згідно з (2.3) без застосування змін до хмарного середовища. Результатом є артефакт плану — серіалізований набір операцій створення, модифікації та видалення ресурсів;

– σ_5 — перевірка відповідності (Comply): оцінка артефакту плану на відповідність політикам P_e на рівні конкретних змін ресурсів, а не лише синтаксису конфігурації. На відміну від σ_3, де перевіряється код, на цій стадії перевіряється семантика запланованих змін: наприклад, заборона видалення ресурсів у виробничому середовищі або обмеження типів екземплярів. Формально перевіряється предикат $\text{comply}(\Delta_e^{\text{plan}}, P_e) = \text{true}$;

– σ_6 — затвердження (Approve): ручне або автоматизоване підтвердження застосування плану. Для середовищ із вищою критичністю (наприклад, e_{prod}) затвердження є обов'язково ручним, тоді як для e_{dev} може бути автоматичним. Рівень затвердження визначається атрибутом середовища і є частиною P_e;

– σ_7 — застосування (Apply): виконання операцій з артефакту плану, що переводить фактичний стан S_e^a у відповідність до бажаного S_e^d. Після успішного виконання записаний стан оновлюється: $S_e^r \leftarrow S_e^a$. Інваріант стадії — ідемпотентність: за умови $\Delta_e^{\text{plan}} = \varnothing$ жодних змін не відбувається;

– σ_8 — моніторинг (Monitor): періодичне обчислення дельти дрейфу $\Delta_e^{\text{drift}} = S_e^r \setminus S_e^a$ згідно з (2.4) через операцію `refresh` та порівняння з очікуваним порожнім планом. Виявлення непорожньої дельти ініціює зворотний перехід до стадії планування для усунення дрейфу.

Життєвий цикл окремого середовища формалізується як скінченний автомат $A_e = \langle \Sigma, \delta, \sigma_1, F \rangle$, де $\delta : \Sigma \times \{0,1\} \to \Sigma$ — функція переходів, що залежить від результату перевірки на кожній стадії (успіх або невдача), а $F = \{\sigma_8\}$ — стадія стаціонарного стану, з якої автомат виходить лише при виявленні дрейфу або надходженні нової зміни конфігурації. Основні переходи утворюють послідовний ланцюг $\sigma_1 \to \sigma_2 \to \sigma_3 \to \sigma_4 \to \sigma_5 \to \sigma_6 \to \sigma_7 \to \sigma_8$, з такими зворотними переходами: невдача на σ_3 або σ_5 повертає автомат до σ_2 для виправлення конфігурації; відхилення на σ_6 повертає до σ_2; виявлення непорожньої Δ_e^{drift} на σ_8 ініціює перехід до σ_4 для обчислення плану усунення дрейфу.

2.1.2 Вертикальна проекція: просування змін між середовищами

Для простору середовищ $(\mathcal{E}, \preceq)$ просування визначається як координована активація горизонтальних циклів послідовних середовищ. Нехай $e_i \preceq e_j$ — пара суміжних середовищ. Операція $\text{promote}(e_i, e_j)$ згідно з (2.5) дозволена лише за виконання умови готовності:

$$\text{ready}(e_i) \iff \sigma(e_i) = \sigma_8 \ \wedge \ \Delta_{e_i}^{\text{plan}} = \varnothing \ \wedge \ \Delta_{e_i}^{\text{drift}} = \varnothing, \tag{2.6}$$

тобто середовище-джерело перебуває у стадії моніторингу та є консистентним. Ця умова гарантує, що до цільового середовища просувається лише перевірена та успішно застосована конфігурація.

Після виконання $\text{promote}(e_i, e_j)$ у цільовому середовищі e_j активується горизонтальний цикл починаючи зі стадії σ_3 (валідація), оскільки ініціалізація (σ_1) вже виконана, а авторство (σ_2) замінено операцією просування. Цільове середовище проходить повний ланцюг перевірок $\sigma_3 \to \sigma_4 \to \sigma_5 \to \sigma_6 \to \sigma_7 \to \sigma_8$ з власними політиками P_{e_j}, які можуть бути суворішими за P_{e_i}. Це є ключовою відмінністю від підходів, де просування зводиться до копіювання артефактів: у запропонованій моделі кожне середовище є повноцінним шлюзом якості зі зростаючим рівнем

вимог.

Для ланцюга з n середовищ $e_1 \preceq e_2 \preceq \ldots \preceq e_n$ повний цикл просування визначається як послідовна композиція:

$$\text{Promote}(e_1, e_n) = \text{promote}(e_{n-1}, e_n) \circ \ldots \circ \text{promote}(e_1, e_2), \tag{2.7}$$

де кожна наступна операція виконується лише після досягнення попереднім середовищем стану $\text{ready}(e_i)$. Для нелінійних топологій $(\mathcal{E}, \preceq)$ з паралельними гілками просування незалежних ланцюгів відбувається конкурентно, що скорочує загальний час доставки змін.

Запропонована двопроекційна модель відрізняється від існуючих підходів у двох аспектах. По-перше, стадія перевірки відповідності σ_5 виокремлена як самостійний крок між плануванням та затвердженням, тоді як у типових конвеєрах перевірка політик або відсутня, або поєднана зі стадією валідації [22]. Це розмежування є принциповим: валідація σ_3 перевіряє коректність коду, тоді як σ_5 перевіряє допустимість запланованих змін у контексті конкретного середовища. По-друге, моніторинг σ_8 із зворотним переходом до σ_4 формалізує замкнений цикл зворотного зв'язку для виявлення та усунення дрейфу, який у більшості існуючих моделей залишається зовнішнім процесом [20,51]. Детальний опис механізму просування змін та забезпечення відповідності подається у підрозділі 2.2.

На основі формалізованих понять та моделі життєвого циклу сформульовано шість архітектурних принципів, що визначають обмеження на допустимі операції над простором середовищ $(\mathcal{E}, \preceq)$ та є інваріантами запропонованого методу.

Принцип 1: повна ізоляція стану. Кожне середовище $e \in \mathcal{E}$ має фізично окремий файл стану S_e^r, бекенд зберігання та облікові дані доступу до хмарного провайдера. Жодна операція над e_i не може модифікувати $S_{e_j}^r$ при $i \neq j$. Принцип безпосередньо випливає з обраного directory-based патерну та забезпечує гарантований радіус ураження $B_R = 1$.

Принцип 2: єдине джерело конфігурацій. Логіка ресурсів інкапсулюється

у спільних модулях $\mathcal{M}$, а кореневі модулі середовищ m^{e}_{root} зводяться до параметризованих викликів згідно з (2.1). Відмінності між середовищами виражаються виключно через набори параметрів Θ_e, що мінімізує конфігураційну ентропію H_C.

Принцип 3: мінімізація радіуса ураження. Стан кожного середовища сегментується за функціональними компонентами інфраструктури на незалежні одиниці управління зі власними файлами стану. Помилкова операція `apply` впливає лише на один сегмент одного середовища.

Принцип 4: вбудована відповідність. Перевірка політик P_e є обов'язковою стадією (σ_3, σ_5) автомата A_e, а не факультативним доповненням. Перехід $\sigma_5 \to \sigma_6$ неможливий за $\text{comply}(\Delta_e^{\text{plan}}, P_e) = \text{false}$. Суворість політик зростає вздовж $\preceq$: $P_{e_i} \subseteq P_{e_j}$ для $e_i \preceq e_j$.

Принцип 5: безперервна верифікація стану. Стадія моніторингу σ_8 виконується періодично та автоматично. Виявлення $\Delta_e^{\text{drift}} \neq \varnothing$ ініціює зворотний перехід до σ_4 без ручного втручання, замикаючи цикл зворотного зв'язку.

Принцип 6: провайдеро-незалежність. Метод оперує абстракціями конфігурації C, стану S та дельти Δ, які не залежать від конкретної реалізації виконавчого рушія. Це забезпечує застосовність як до Terraform, так і до OpenTofu та інших HCL-сумісних інструментів.

Сукупність принципів 1–6 визначає зв'язок «модель – метод – засіб»: модель життєвого циклу (автомат A_e та простір $(\mathcal{E}, \preceq)$) задає структуру процесу; метод (деталізований у підрозділі 2.2) визначає конкретні процедури просування та забезпечення відповідності в межах цієї структури; засіб реалізації (розділ 4) імплементує метод через програмні механізми управління робочими просторами, політиками та конвеєрами.

2.2 Метод автоматизації просування змін між середовищами та забезпечення відповідності

У підрозділі 2.1 операцію просування $\text{promote}(e_i, e_j)$ визначено як перенесення конфігурації між суміжними середовищами з адаптацією параметрів. У цьому підрозділі деталізується конкретний механізм виконання цієї операції як послідовності атомарних кроків із формальними передумовами та гарантіями.

Вибір атомарних кроків як одиниці побудови механізму просування має принципове значення для властивостей результуючої системи. Атомарність означає, що кожен крок або виконується повністю, або не виконується зовсім, без проміжних станів, видимих зовнішньому спостерігачеві. У контексті управління інфраструктурою це властивість особливо важлива через те, що частково виконані операції над хмарними ресурсами породжують стан, який складно автоматично відновити: створення половини ресурсів, зміна частини атрибутів або ініціалізація одного з кількох пов'язаних об'єктів залишають інфраструктуру в невизначеному положенні, що потребує ручної діагностики та виправлення. Побудова просування як послідовності атомарних кроків зміщує цю складність з рівня окремих ресурсів на рівень координації кроків, що є значно простішою задачею з огляду на обмежену кількість кроків і їхню передбачувану поведінку.

Альтернативою атомарним крокам є безперервне застосування змін, за якого кожна модифікація конфігурації негайно транслюється в цільове середовище без проміжної фіксації стану. Такий підхід мінімізує затримку між написанням коду і появою змін у робочому середовищі, проте позбавляє систему можливості для проміжного контролю якості, перевірки відповідності та отримання схвалення від відповідальних осіб. Для виробничих середовищ, де наслідки помилкового застосування можуть бути значними, безперервне застосування є неприйнятним підходом, і практика галузі фактично сформувала консенсус щодо необхідності етапного просування з контрольними точками між середовищами різного рівня

критичності.

Ще однією властивістю, яку забезпечує розбиття на атомарні кроки, є можливість паралельного виконання незалежних операцій. Коли просування складається з послідовних кроків із явно визначеними вхідними та вихідними артефактами, легко встановити, які кроки можуть виконуватись паралельно, а які потребують послідовного виконання через спільні залежності. Для організацій, що управляють десятками або сотнями середовищ, паралельність просування є важливим фактором скорочення загального часу розгортання змін, і формалізація кроків закладає передумови для реалізації такої паралельності без втрати гарантій коректності.

Організаційним аспектом атомарності кроків є чітке розмежування зон відповідальності між учасниками процесу. Коли просування розкладено на послідовність кроків, кожен крок може бути віднесений до компетенції конкретної ролі: автор зміни відповідає за написання та локальну валідацію конфігурації, спеціаліст з безпеки — за перевірку відповідності політикам, керівник команди — за затвердження застосування у виробничому середовищі, оператор — за моніторинг після застосування. Така декомпозиція дає можливість будувати робочі процеси, що відповідають організаційній структурі підприємства, та автоматизовано перевіряти дотримання принципу розподілу обов'язків, що є вимогою багатьох галузевих стандартів безпеки та регуляторних норм для критичних інфраструктур.

Ключовим поняттям є версіонований знімок конфігурації $\hat{C}_e^v$ — зафіксована версія конфігурації середовища e, що відповідає конкретному комітові у системі контролю версій.

$$\hat{C}_e^v = \langle m_{\text{root}},\ \Theta_e,\ v \rangle, \tag{2.8}$$

де $v \in \mathbb{N}$ — монотонно зростаючий номер версії, що присвоюється після успішного проходження повного циклу $\sigma_3 \to \ldots \to \sigma_7$ у середовищі e. Знімок $\hat{C}_e^v$ є незмінним (immutable) артефактом: після фіксації його вміст не може бути модифіковано, що забезпечує відтворюваність та аудитованість кожної зміни.

Операція просування між суміжними середовищами $e_i \preceq e_j$ формалізується як процедура з шести кроків.

1. Перевірка готовності джерела. Згідно з (2.6), верифікується умова $\text{ready}(e_i)$: середовище e_i перебуває у стадії σ_8, дельти планування та дрейфу дорівнюють $\varnothing$. Невиконання умови блокує просування.

2. Фіксація знімку. Визначається версія $\hat{C}^v_{e_i}$, що просувається. Кореневий модуль m_{root} та версії дочірніх модулів фіксуються за конкретними тегами у реєстрі модулів, що унеможливлює неявну зміну залежностей між моментом тестування в e_i та застосування в e_j.

3. Адаптація параметрів. Формується конфігурація цільового середовища $C_{e_j} = \langle m_{\text{root}},\ \Theta_{e_j} \rangle$, де m_{root} ідентичний знімку $\hat{C}^v_{e_i}$, а Θ_{e_j} містить середовищно-специфічні значення: облікові дані провайдера, регіон розміщення, параметри масштабування тощо. Підстановка параметрів є єдиним допустимим видом модифікації при просуванні; будь-яка зміна логіки модулів вимагає повторного проходження циклу починаючи з σ_2 у середовищі-джерелі.

4. Активація циклу у цільовому середовищі. У середовищі e_j послідовно виконуються стадії $\sigma_3 \to \sigma_4 \to \sigma_5 \to \sigma_6 \to \sigma_7$ з політиками P_{e_j}. Стадії σ_1 (ініціалізація) та σ_2 (авторство) пропускаються, оскільки конфігурація надходить через операцію просування, а не через ручне редагування. Кожна стадія може завершитись невдачею: невдача на σ_3 або σ_5 блокує просування та повертає керування автору зміни для виправлення; відхилення на σ_6 вимагає перегляду плану.

5. Фіксація результату. Після успішного виконання у цільовому середовищі σ_7 (Apply) створюється власний знімок $\hat{C}^{v'}_{e_j}$ з новим номером версії v'. Записаний стан оновлюється: $S^r_{e_j} \leftarrow S^a_{e_j}$. Середовище e_j переходить до стадії σ_8 (Monitor).

6. Каскадне просування. Якщо $\text{ready}(e_j)$ досягнуто і існує наступне середовище $e_j \preceq e_k$ у ланцюзі $(\mathcal{E}, \preceq)$, процедура повторюється для пари (e_j, e_k). Для повного ланцюга $e_1 \preceq \ldots \preceq e_n$ каскадне просування відповідає композиції (2.7).

Критичною властивістю описаної процедури є атомарність: просування або завершується повним циклом до σ_8 у цільовому середовищі, або не змінює його фактичний стан $S^a_{e_j}$. Атомарність досягається через механізм артефактів плану: стадія σ_7 виконує лише ті операції, що були зафіксовані в артефакті на стадії σ_4, а невдача на будь-якій проміжній стадії (σ_3, σ_5, σ_6) відбувається до початку модифікації хмарних ресурсів. Часткове застосування (partial apply), коли частина ресурсів створена, а частина — ні, є єдиним сценарієм порушення атомарності; його обробка делегується механізму відкату, що описується далі.

Для нелінійних топологій $(\mathcal{E}, \preceq)$ з паралельними гілками просування незалежних ланцюгів відбувається конкурентно. Нехай $e_1 \preceq e_2$ та $e_1 \preceq e_3$, де e_2 і e_3 не пов'язані відношенням $\preceq$. Після досягнення ready(e_1) операції promote(e_1, e_2) та promote(e_1, e_3) можуть виконуватись паралельно, оскільки принцип повної ізоляції стану гарантує відсутність конфліктів: $S^r_{e_2}$ та $S^r_{e_3}$ є фізично незалежними, і операції над ними не потребують координації.

Атомарність просування, визначена у попередньому блоці, спирається на можливість повернення середовища до попереднього відомого стану у разі збою. Механізм відкату є складовою задачі дослідження 3: забезпечення відкату до попереднього стану при помилкових змінах.

Кожна успішна операція Apply (σ_7) породжує версіонований знімок стану:

$$S^{r,v}_e = \text{snapshot}(S^r_e, v), \tag{2.9}$$

де v — номер версії, що збігається з версією знімку конфігурації $\hat{C}^v_e$, застосованого на цій ітерації. Множина знімків утворює історію стану $\mathcal{H}_e = \{S^{r,1}_e, S^{r,2}_e, \ldots, S^{r,v}_e\}$ з повним упорядкуванням за часом створення. Зберігання історії у віддаленому бекенді з підтримкою версіонування (наприклад, S3 з увімкненим object versioning) забезпечує доступність будь-якого попереднього знімку.

Операція відкату визначається як повернення середовища до стану, що від-

повідає версії $v' < v$:

$$\text{rollback}(e, v') : S_e^r \leftarrow S_e^{r,v'}. \tag{2.10}$$

Після відновлення записаного стану необхідно привести фактичний стан у відповідність, оскільки S_e^a вже містить ресурси, створені або змінені пізнішими версіями. Для цього активується скорочений цикл $\sigma_4 \to \sigma_5 \to \sigma_6 \to \sigma_7$: операція Plan обчислює дельту між відновленим $S_e^{r,v'}$ та поточним S_e^a, що зазвичай включає видалення нових ресурсів та відновлення модифікованих атрибутів. Стадії Comply та Approve залишаються обов'язковими, оскільки відкат також має задовольняти політики P_e.

Безпека відкату залежить від зворотності операцій, що були виконані між версіями v' та v. Визначимо для кожного ресурсу $r \in R_m$ атрибут зворотності:

$$\text{rev}(r) = \begin{cases} 1, & \text{якщо перестворення r відновлює його попередній стан;} \\ 0, & \text{інакше.} \end{cases} \tag{2.11}$$

Ресурси з $\text{rev}(r) = 1$ (наприклад, обчислювальні екземпляри, мережеві правила, DNS-записи) допускають повний автоматичний відкат. Ресурси з $\text{rev}(r) = 0$ (наприклад, бази даних із накопиченими транзакціями, сховища з видаленими об'єктами) потребують додаткових механізмів захисту: позначки `prevent_destroy` у конфігурації, окремих процедур резервного копіювання або ручного підтвердження деструктивних операцій.

На основі атрибута зворотності визначається безпечність відкату для переходу від версії v до v':

$$\text{safe}(e, v, v') \iff \forall\, r \in (S_e^{r,v} \setminus S_e^{r,v'}) : \text{rev}(r) = 1, \tag{2.12}$$

тобто відкат є безпечним, якщо всі ресурси, що підлягають видаленню або зміні, є зворотними. Якщо $\text{safe}(e, v, v') = \text{false}$, метод вимагає ручного затвердження на стадії σ_6 незалежно від рівня автоматизації середовища, що запобігає ненавмисній

втраті даних.

Механізм відкату інтегрується з конвеєром просування таким чином: якщо операція Apply (σ_7) у цільовому середовищі e_j завершується частковим застосуванням (partial apply), коли частина ресурсів створена, а частина — ні, записаний стан $S^r_{e_j}$ фіксує цей проміжний стан. Відновлення виконується через rollback(e_j, v'_j), де v'_j — остання успішна версія, з подальшим повним циклом $\sigma_4 \to \sigma_7$, що видаляє частково створені ресурси та повертає середовище до консистентного стану.

Принцип 4 (вбудована відповідність) визначає, що перевірка політик P_e є обов'язковою стадією автомата A_e. У цьому блоці деталізується структура множини політик та механізм їхнього застосування на трьох рівнях життєвого циклу.

Множина політик середовища P_e декомпозується на три підмножини за рівнем перевірки:

$$P_e = P_e^{\text{code}} \cup P_e^{\text{plan}} \cup P_e^{\text{state}}, \tag{2.13}$$

де P_e^{code} — політики рівня коду, що застосовуються на стадії σ_3;
P_e^{plan} — політики рівня плану, що застосовуються на стадії σ_5;
P_e^{state} — політики рівня стану, що застосовуються на стадії σ_8.

Кожна підмножина оперує різним артефактом і виявляє різний клас порушень.

Рівень коду (σ_3, предикат valid(C_e, P_e^{code})). Вхідний артефакт — файли конфігурації C_e. Політики цього рівня перевіряють синтаксичну коректність через `terraform validate`, дотримання стандартів іменування та структури модулів, а також статичну відсутність відомих вразливостей безпеки (захардкоджені секрети, відкриті мережеві порти, незашифровані сховища). Перевірки виконуються без звернення до хмарного провайдера, що забезпечує швидкий зворотний зв'язок на ранніх стадіях розробки відповідно до принципу Shift Left [67].

Рівень плану (σ_5, предикат comply$(\Delta_e^{\text{plan}}, P_e^{\text{plan}})$). Вхідний артефакт — серіалізований план змін, що містить конкретні операції створення, модифікації та видалення ресурсів з обчисленими атрибутами. На відміну від рівня коду, тут

перевіряється семантика запланованих змін у контексті конкретного середовища: заборона видалення критичних ресурсів у виробничому середовищі, обмеження допустимих типів екземплярів, контроль оцінки вартості змін, обов'язковість шифрування для певних класів ресурсів. Цей рівень є принциповою відмінністю запропонованого методу від типових конвеєрів, де перевірка політик або обмежена рівнем коду, або поєднана з ним [22].

Рівень стану (σ_8, предикат drift_comply$(S_e^a, P_e^{\text{state}})$). Вхідний артефакт — фактичний стан хмарних ресурсів S_e^a, отриманий через операцію `refresh`. Політики цього рівня перевіряють відповідність поточного стану вимогам безпеки у реальному часі: наприклад, відсутність публічного доступу до сховищ або коректність мережевих правил, навіть якщо ці атрибути були змінені поза Terraform. Порушення на цьому рівні ініціює перехід $\sigma_8 \to \sigma_4$ для усунення дрейфу.

Політики організовані у ієрархію успадкування з трьома рівнями: глобальні P_{global}, що застосовуються до всіх середовищ організації; середовищні P_{env}, що визначають додаткові обмеження для конкретного e; компонентні P_{comp}, що є специфічними для окремого сегменту інфраструктури. Результуюча множина формується як об'єднання:

$$P_e = P_{\text{global}} \cup P_{\text{env}(e)} \cup P_{\text{comp}(e)}. \tag{2.14}$$

Ієрархія забезпечує принцип зростання суворості вздовж $(\mathcal{E}, \preceq)$: глобальні політики є спільним мінімумом, а середовищні додають обмеження пропорційно критичності середовища. Наприклад, $P_{\text{env(prod)}}$ може вимагати обов'язкового ручного затвердження та заборони деструктивних операцій, тоді як $P_{\text{env(dev)}}$ допускає повну автоматизацію.

Кожна окрема політика $p \in P_e$ визначається як правило з рівнем застосування (enforcement level):

– advisory — порушення фіксується, але не блокує конвеєр;

– soft-mandatory — блокує за замовчуванням, проте може бути перевизна-

чене уповноваженим користувачем;

– hard-mandatory — блокує без можливості перевизначення. Розподіл рівнів за середовищами типово відповідає $(\mathcal{E}, \preceq)$: одна й та сама політика може бути advisory у e_{dev} та hard-mandatory у e_{prod}, що дозволяє поступово виявляти порушення без блокування розробки на ранніх стадіях.

Стадія моніторингу σ_8 із зворотним переходом до σ_4 замикає цикл зворотного зв'язку автомата A_e. У цьому блоці деталізується алгоритм виявлення дрейфу та механізм прийняття рішення щодо його усунення.

Виявлення дрейфу виконується періодично з інтервалом τ, що є параметром середовища. На кожній ітерації виконуються два кроки: операція `refresh`, що оновлює записаний стан S_e^r відповідно до фактичного S_e^a, та операція `plan`, що обчислює дельту $\Delta_e^{\text{drift}} = S_e^r \setminus S_e^a$ згідно з (2.4). Якщо $\Delta_e^{\text{drift}} = \varnothing$, середовище залишається у σ_8. Інакше кожен елемент дельти класифікується для визначення подальших дій.

Функція класифікації зіставляє змінений ресурс із політиками P_e^{state} та присвоює йому один із трьох класів:

$$\text{class}(r, \Delta_e^{\text{drift}}, P_e^{\text{state}}) \in \{\text{critical},\ \text{actionable},\ \text{informational}\}. \tag{2.15}$$

Критичний дрейф — зміна порушує політику з рівнем hard-mandatory (наприклад, вимкнення шифрування сховища або відкриття порту в мережевому правилі). Усунення ініціюється автоматично через перехід $\sigma_8 \to \sigma_4$: обчислений план повертає ресурс до бажаного стану S_e^d, проходить перевірку відповідності σ_5 та застосовується після затвердження σ_6. Операбельний дрейф — зміна не порушує hard-mandatory політик, проте створює розбіжність між S_e^r та S_e^a, що потребує уваги (наприклад, зміна тегів ресурсу або параметрів масштабування). Усунення вимагає ручного підтвердження. Інформаційний дрейф — зміна атрибутів, що очікувано змінюються поза Terraform (динамічні IP-адреси, автоматично ротовані сертифікати, лічильники стану). Такі атрибути позначаються в конфігурації

директивою `ignore_changes`, і їхня зміна не ініціює усунення.

Формально поняття допустимого дрейфу визначається через множину ігнорованих атрибутів $I_e \subset \text{attr}(R_m)$:

$$\Delta_e^{\text{eff}} = \Delta_e^{\text{drift}} \setminus \{(r, a) : a \in I_e\}, \tag{2.16}$$

де Δ_e^{eff} — ефективна дельта дрейфу після виключення допустимих змін. Середовище вважається консистентним за розширеним означенням, якщо $\Delta_e^{\text{plan}} = \varnothing$ та $\Delta_e^{\text{eff}} = \varnothing$.

Інтервал виявлення τ визначається рівнем критичності середовища: для e_{prod} типовим є $\tau = 1$ година, для e_{staging} — $\tau = 6$ годин, для e_{dev} — $\tau = 24$ години або виявлення за запитом. Цей параметр є частиною P_e і може бути змінений через ієрархію успадкування (2.14).

Поєднання трирівневої класифікації з ефективною дельтою (2.16) забезпечує баланс між автоматизацією та контролем: критичні порушення усуваються негайно, операбельні — за підтвердженням оператора, інформаційні — не створюють хибних сповіщень.

2.3 Висновки

У розділі розроблено узагальнений метод управління життєвим циклом Terraform-інфраструктури для кількох середовищ. Введено формальний апарат, що включає тричленну модель стану (бажаний, записаний, фактичний), простір середовищ як частково впорядковану множину $(\mathcal{E}, \preceq)$ та операцію просування з формальними передумовами. На основі порівняльного аналізу трьох патернів ізоляції за п'ятьма критеріями обґрунтовано вибір directory-based патерну як базового. Запропоновано двопроекційну модель життєвого циклу у вигляді скінченного автомата A_e з восьми стадій, де виокремлення перевірки відповідності (σ_5) як самостійної стадії та замкнений цикл виявлення дрейфу ($\sigma_8 \to \sigma_4$) є ключовими

відмінностями від існуючих підходів. Деталізовано шестикрокову процедуру просування з гарантією атомарності, механізм відкату на основі версіонованих знімків стану з формальним предикатом безпечності, багаторівневу модель перевірки відповідності з ієрархічним успадкуванням політик та трикласову класифікацію конфігураційного дрейфу. Запропонована формалізація дозволяє застосовувати метод до різнотипних хмарних середовищ без модифікації його концептуальної основи, оскільки математичні об'єкти моделі описані незалежно від конкретного постачальника або інструменту оркестрації. Сумісність методу з наявними інструментами Terraform забезпечується тим, що запропоновані абстракції надбудовуються над стандартним виконавчим рушієм і не вимагають його заміни або глибокої модифікації. Сукупність отриманих результатів адресує задачі дослідження 1–3; практична реалізація та експериментальна оцінка методу подаються у розділі 4.

3 АЛГОРИТМІЧНЕ ЗАБЕЗПЕЧЕННЯ ТА АРХІТЕКТУРНЕ ПРОЄКТУВАННЯ МЕТОДУ

3.1 Алгоритми управління життєвим циклом середовища

Перехід від формальної моделі до алгоритмічного забезпечення є необхідним етапом розроблення методу, що перетворює декларативний опис властивостей системи на виконавчу специфікацію, придатну для реалізації програмними засобами. Формальна модель описує об'єкти управління та відношення між ними в статичному вигляді, визначаючи, які стани системи є допустимими і які переходи між ними мають місце. Алгоритми, у свою чергу, фіксують динамічний аспект: послідовність дій, що забезпечує досягнення цільового стану з вихідного, умови прийняття рішень у кожній точці процесу та реакцію на нештатні ситуації. Без алгоритмічної деталізації формальна модель залишається корисною лише для концептуального аналізу, але не може бути безпосередньо втілена в програмну систему.

При проєктуванні алгоритмів управління життєвим циклом інфраструктури важливим принципом є мінімізація кількості неявних припущень про виконавче середовище. Алгоритм, що спирається на специфічні особливості конкретного інструменту або хмарної платформи, втрачає властивість узагальненості і стає непереносним на інші реалізації. Тому в наступних алгоритмах навмисно уникаються посилання на інструментальні деталі, а кроки формулюються в термінах об'єктів формальної моделі: конфігурацій, станів, множин політик та операцій над ними. Це дозволяє реалізовувати ті самі алгоритми на різних платформах оркестрації, зберігаючи незмінними їхні властивості коректності та завершеності.

Другим принципом проєктування є явне розділення функціональних обов'язків між алгоритмами. Замість єдиного монолітного алгоритму, що охоплює весь спектр операцій, пропонується набір спеціалізованих алгоритмів, кожен з яких відповідає за окрему логічну одиницю процесу: основний цикл середовища, проце-

дуру просування змін між середовищами, механізм відкату та моніторинг дрейфу. Такий підхід відповідає принципу єдиної відповідальності, відомому в інженерії програмного забезпечення, і має низку переваг з погляду подальшої еволюції методу. Зокрема, модифікація логіки одного з алгоритмів не вимагає перегляду інших, що спрощує внесення змін при уточненні вимог або при адаптації методу до нових умов застосування.

Третім принципом є явне визначення вхідних і вихідних артефактів для кожного алгоритму. Артефакти слугують контрактом між алгоритмом та його оточенням і дозволяють композувати алгоритми в ширші процеси без необхідності знати деталі їхньої внутрішньої реалізації. Коли алгоритм просування споживає на вході артефакт плану, побудований алгоритмом основного циклу, і виробляє на виході оновлений записаний стан, що стає входом для алгоритму моніторингу, формується конвеєр з чіткими межами між ланками. Така композиція зменшує когнітивне навантаження на інженера, що супроводжує інфраструктуру, оскільки дозволяє аналізувати поведінку системи на рівні взаємодії алгоритмів, не занурюючись щоразу в їхні внутрішні кроки.

На основі формальної моделі розділу 2 деталізуються три алгоритми, що операціоналізують стадії автомата A_e, процедуру просування та механізм відкату. Кожен алгоритм задається у вигляді нумерованої послідовності кроків із явними умовами переходів.

Алгоритм 1. Виконання основного циклу середовища A_e.

Вхід: конфігурація C_e, записаний стан S_e^r, множина політик $P_e = P_e^{\text{code}} \cup P_e^{\text{plan}} \cup P_e^{\text{state}}$, параметри Θ_e.

Вихід: оновлений S_e^r або діагностичне повідомлення про блокування.

1. (σ_1, Init) Завантажити записаний стан S_e^r з віддаленого бекенду. Якщо бекенд недоступний — зупинитись та повідомити оператора.

2. (σ_2, Author) Прийняти зміну ΔC_e через pull request; оновити $C_e \leftarrow C_e \cup \Delta C_e$.

3. (σ_3, Validate) Виконати `terraform validate`; застосувати до C_e правила P_e^{code} (статичний аналіз безпеки, стандарти іменування). Якщо $\text{valid}(C_e, P_e^{\text{code}}) = \text{false}$ — повернутись до кроку 2 з переліком порушень.

4. (σ_4, Plan) Виконати `terraform plan`; обчислити $\Delta_e^{\text{plan}} = S_e^d \setminus S_e^r$ (2.3); серіалізувати результат у JSON-артефакт плану.

5. (σ_5, Comply) Застосувати до JSON-артефакту правила P_e^{plan} (заборона деструктивних операцій у prod, обмеження типів ресурсів, оцінка вартості). Якщо $\text{comply}(\Delta_e^{\text{plan}}, P_e^{\text{plan}}) = \text{false}$ — повернутись до кроку 2.

6. (σ_6, Approve) Якщо P_e вимагає ручного затвердження (виробниче середовище або план містить деструктивні операції) — очікувати підтвердження уповноваженої особи. Якщо відхилено — повернутись до кроку 2.

7. (σ_7, Apply) Виконати `terraform apply` з серіалізованого артефакту. Якщо $\Delta_e^{\text{plan}} = \varnothing$ — жодних дій (ідемпотентність). Після успіху: $S_e^r \leftarrow S_e^a$; зафіксувати знімок $S_e^{r,v}$ (2.9).

8. (σ_8, Monitor) Перейти до Алгоритму 2 (моніторинг дрейфу). При виявленні $\Delta_e^{\text{eff}} \neq \varnothing$ критичного класу — повернутись до кроку 4. При надходженні нової зміни конфігурації — повернутись до кроку 2.

Блок-схему основного циклу автомата A_e наведено на рисунку 3.1 (с. 52).

Алгоритм 2. Просування змін між середовищами $\text{promote}(e_i, e_j)$.

Вхід: середовище-джерело e_i, цільове середовище e_j, версія v.

Вихід: запущений цикл A_{e_j} починаючи з σ_3, або відмова з причиною.

1. Перевірити передумову готовності (2.6): стадія $\sigma(e_i) = \sigma_8$; $\Delta_{e_i}^{\text{plan}} = \varnothing$; $\Delta_{e_i}^{\text{eff}} = \varnothing$. Якщо хоча б одна умова не виконана — повернути помилку «джерело не готове».

2. Отримати знімок конфігурації $\hat{C}_{e_i}^v$ (2.8) з зафіксованими тегами всіх дочірніх модулів $\mathcal{M}$.

3. Сформувати конфігурацію цільового середовища: замінити Θ_{e_i} на Θ_{e_j} у знімку; результат — $C_{e_j}^v$.

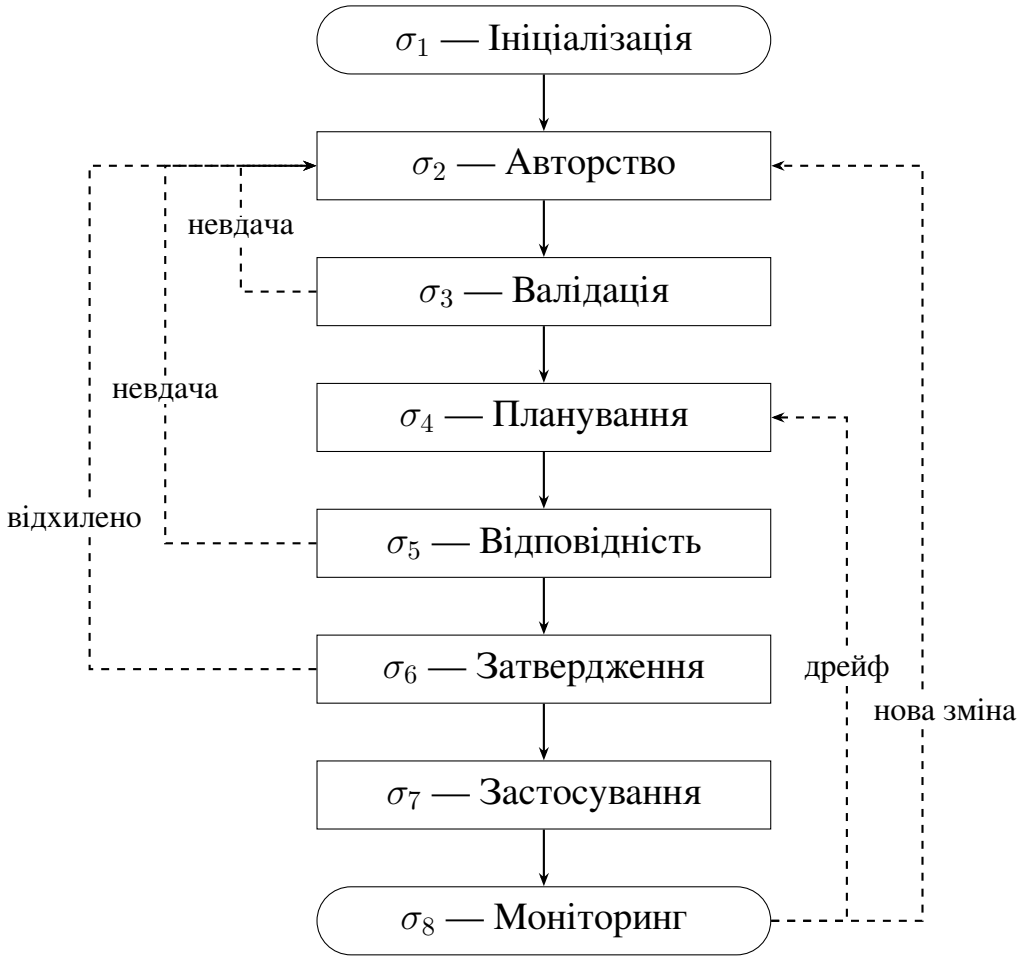


Рисунок 3.1 – Блок-схема Алгоритму 1 — основний цикл автомата A_e

4. Ініціювати A_{e_j} з кроку 3 Алгоритму 1 (стадія σ_3) з конфігурацією $C^v_{e_j}$ та політиками P_{e_j}.

5. Для каскадного просування через повний ланцюг $e_1 \preceq \ldots \preceq e_n$: після досягнення ready$(e_j)$ повторити кроки 1–4 для пари (e_j, e_{j+1}) згідно з (2.7). Незалежні гілки $(\mathcal{E}, \preceq)$ виконуються паралельно.

Блок-схему процедури просування наведено на рисунку 3.2 (с. 53).

Алгоритм 3. Відкат до попередньої версії rollback(e, v').

Вхід: середовище e, цільова версія $v' < v$.

Вихід: S^a_e приведено у відповідність до $S^{r,v'}_e$, або запит на ручне затвердження при незворотних ресурсах.

1. Відновити $S^{r,v'}_e$ (2.9) з версіонованого сховища стану; встановити $S^r_e \leftarrow S^{r,v'}_e$ (2.10).

2. Виконати `terraform plan` відносно відновленого $S^{r,v'}_e$; отримати JSON-

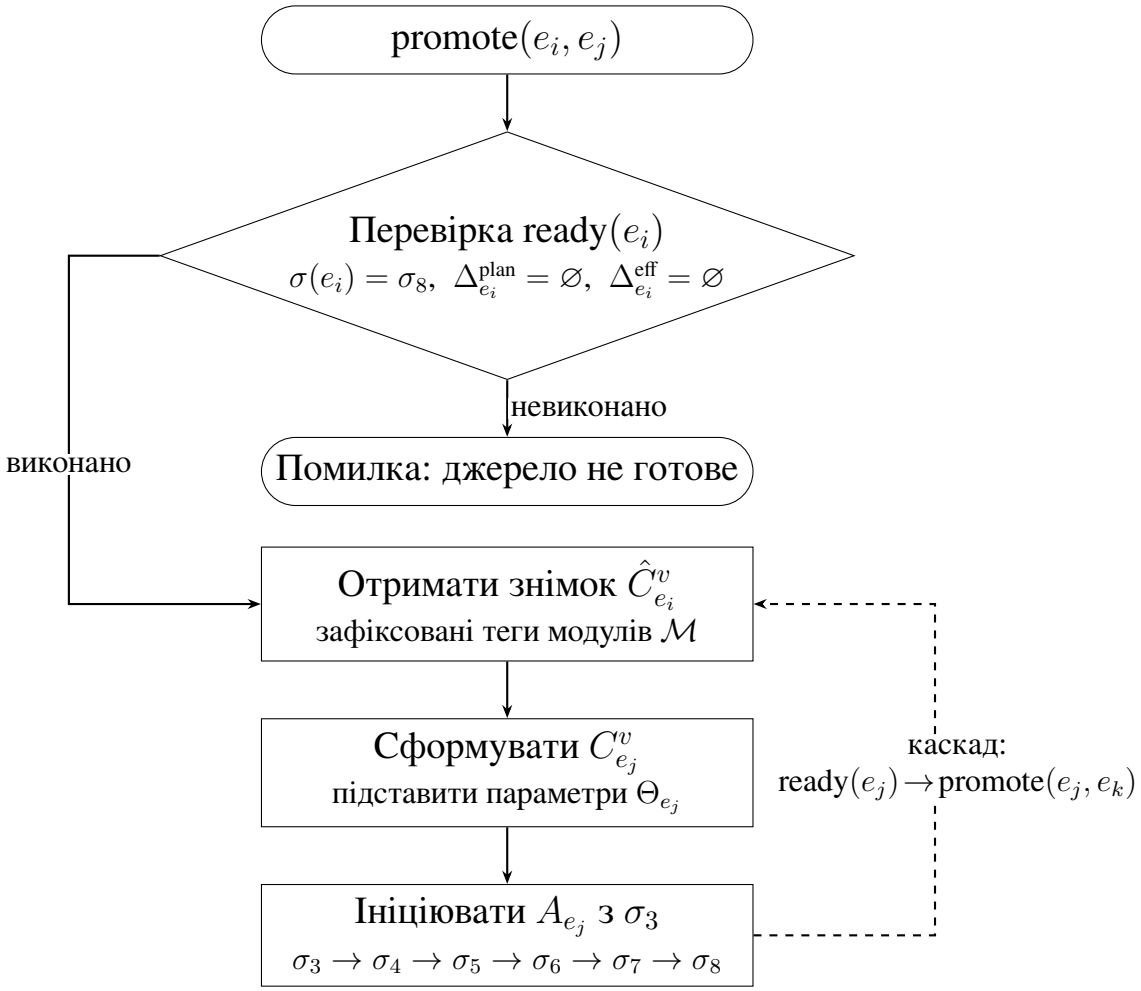


Рисунок 3.2 – Блок-схема Алгоритму 2 — просування змін між середовищами

план операцій відкату (видалення нових ресурсів, відновлення змінених атрибутів).

3. Обчислити предикат безпечності (2.12): перевірити, чи всі ресурси у $(S_e^{r,v} \setminus S_e^{r,v'})$ мають $\text{rev}(r) = 1$ (2.11). Якщо $\text{safe}(e, v, v') = \text{false}$ — примусово встановити σ_6 у ручний режим незалежно від рівня автоматизації середовища; відобразити оператору перелік незворотних ресурсів із попередженням про втрату даних.

4. Виконати кроки 5–7 Алгоритму 1 ($\sigma_5 \to \sigma_6 \to \sigma_7$): перевірити план відкату на відповідність P_e, отримати затвердження, застосувати.

5. Після успіху: $S_e^r \leftarrow S_e^a$; зафіксувати новий знімок $S_e^{r,v''}$ з позначкою відкату.

Блок-схему процедури відкату наведено на рисунку 3.3 (с. 54).

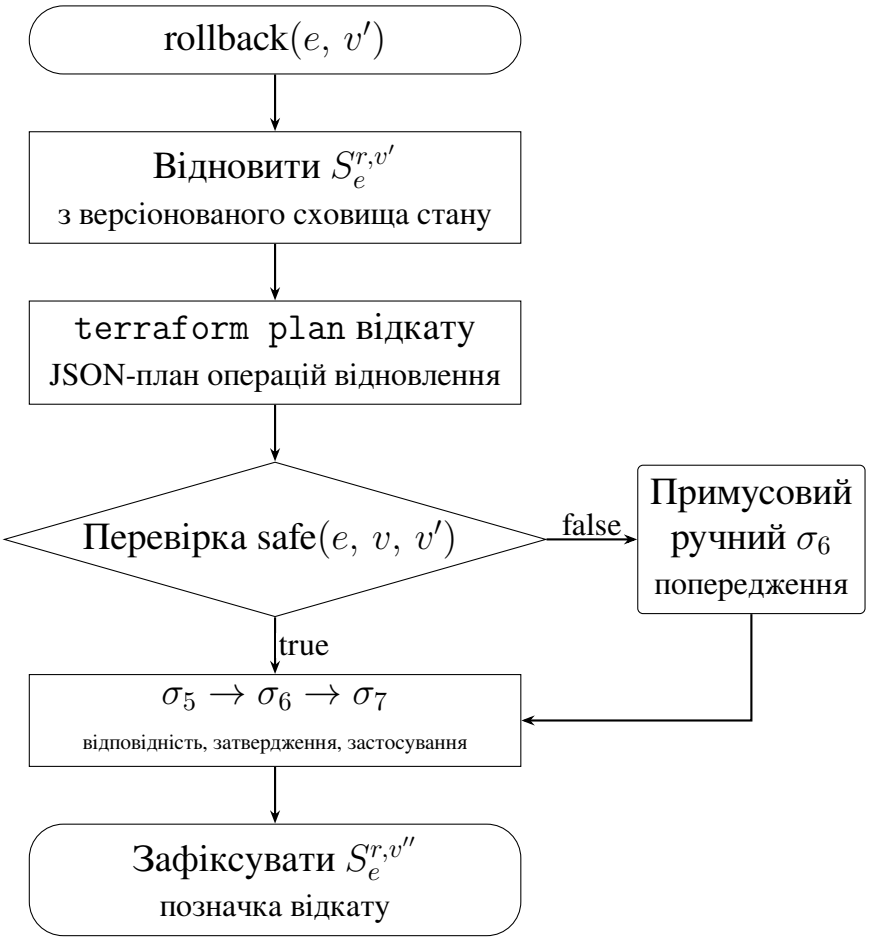


Рисунок 3.3 – Блок-схема Алгоритму 3 — відкат до попередньої версії

3.2 Алгоритм виявлення та класифікації конфігураційного дрейфу

Алгоритм реалізує стадію σ_8 автомата A_e та замикає цикл зворотного зв'язку через перехід $\sigma_8 \to \sigma_4$.

Алгоритм 4. Моніторинг та класифікація дрейфу.

Вхід: середовище e, інтервал τ, множина ігнорованих атрибутів I_e, політики P_e^{state}.

Вихід: усунений дрейф, сповіщення оператора або відсутність дій залежно від класу виявленої розбіжності.

1. Очікувати інтервал τ. Для e_{prod} типово $\tau = 1$ год; для e_{staging} — 6 год; для e_{dev} — 24 год або за запитом.

2. Виконати `terraform plan -refresh-only`; обчислити $\Delta_e^{\text{drift}} = S_e^r \setminus S_e^a$ (2.4).

3. Обчислити ефективну дельту (2.16), виключивши атрибути з I_e: $\Delta_e^{\text{eff}} = \Delta_e^{\text{drift}} \setminus \{(r, a) : a \in I_e\}$. Якщо $\Delta_e^{\text{eff}} = \varnothing$ — повернутись до кроку 1.

4. Для кожного ресурсу $r \in \Delta_e^{\text{eff}}$ застосувати функцію класифікації (2.15):

– критичний (порушення hard-mandatory політики P_e^{state}) — ініціювати перехід $\sigma_8 \to \sigma_4$: виконати кроки 4–7 Алгоритму 1 для автоматичного відновлення S_e^d;

– операбельний (розбіжність із S_e^d без порушення hard-mandatory) — сформувати план відновлення та сповістити оператора; виконати σ_7 лише після ручного підтвердження на σ_6;

– інформаційний (атрибут належить I_e або зміна є очікуваною поведінкою поза Terraform) — зафіксувати у журналі; жодних дій не ініціювати.

5. Повернутись до кроку 1.

Блок-схему алгоритму моніторингу та класифікації дрейфу наведено на рисунку 3.4 (с. 56).

3.3 Вимоги до програмних засобів та архітектурне проєктування

Алгоритмічний опис методу визначає послідовність дій, необхідних для управління життєвим циклом інфраструктури, проте не містить вказівок щодо того, якими програмними засобами ці дії мають виконуватись. Такий поділ між алгоритмом та реалізацією є свідомим методологічним вибором: він зберігає незалежність методу від конкретного інструменту і створює можливість оцінювати придатність різних платформ для реалізації методу за об'єктивними критеріями. Платформа вважається придатною, якщо вона забезпечує виконання всіх алгоритмів без порушення їхніх гарантій, і непридатною у протилежному випадку. Така постановка задачі дозволяє проводити аргументовані порівняння між наявними на ринку рішеннями та робити обґрунтовані технологічні вибори при впровадженні методу в конкретній організації.

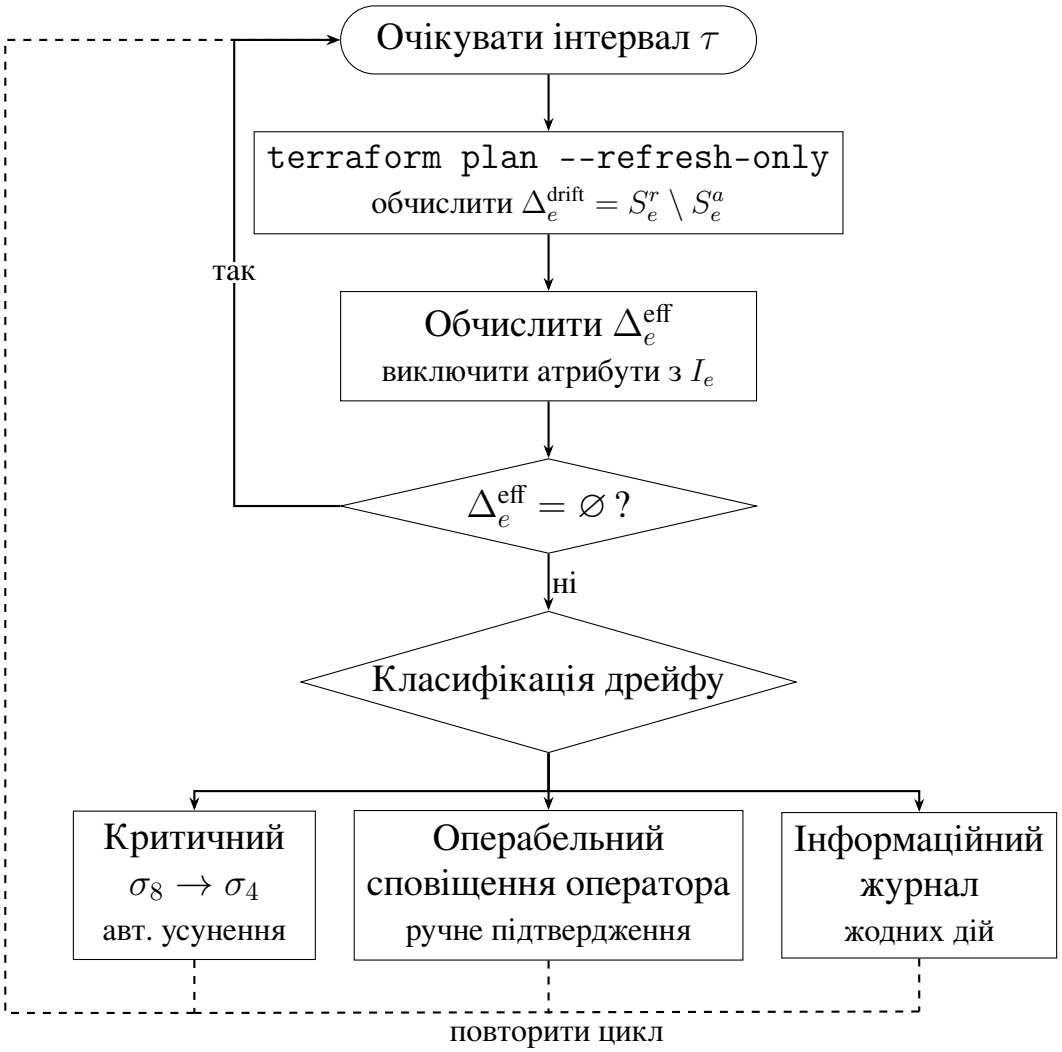


Рисунок 3.4 – Блок-схема Алгоритму 4 — моніторинг та класифікація дрейфу

Вимоги до програмних засобів формулюються на рівні, що абстрагується від конкретних API та синтаксичних особливостей інструментів. Кожна вимога описує функціональну можливість, яка має бути доступною в платформі, але не визначає, як саме ця можливість реалізована технічно. Наприклад, вимога ізоляції стану може бути виконана через окремі бекенди зберігання, через механізм робочих просторів з незалежними обліковими даними або через сегментацію за проєктами в межах однієї облікової системи. Усі три реалізації задовольняють вимогу, попри відмінності в технічному втіленні. Такий рівень абстракції вимог є наслідком того самого принципу, що лежить в основі формалізації алгоритмів: метод має зберігати властивості при зміні реалізації, і вимоги до ПЗ мають виражати саме ті аспекти, від яких ці властивості залежать.

Додатковим аспектом формулювання вимог є розрізнення обов'язкових та бажаних можливостей. Обов'язкові вимоги відповідають функціям, без яких метод не може бути реалізований у повному обсязі. Бажані вимоги стосуються функцій, що підвищують зручність використання, продуктивність або надійність, але не є критичними для базової функціональності. Це розрізнення важливе при оцінці наявних платформ, оскільки рідко яка з них задовольняє всі бажані вимоги одночасно, і практичний вибір часто зводиться до пошуку компромісу між різними аспектами. Платформа, що задовольняє всі обов'язкові вимоги і більшу частину бажаних, може бути придатною для реалізації методу навіть за відсутності окремих функцій, що компенсуються зовнішніми інструментами або ручними процедурами.

Алгоритми 1–4 визначають функціональні вимоги до платформи реалізації незалежно від конкретного інструменту.

B1. Ізоляція стану. Кожному $e \in \mathcal{E}$ відповідає окремий віддалений бекенд із незалежними обліковими даними ($I_S = 1$). Спільний стан між середовищами не допускається.

B2. Дворівнева перевірка відповідності. Конвеєр містить дві незалежні точки застосування політик: до планування (артефакт C_e, стадія σ_3) та після планування (JSON-артефакт Δ_e^{plan}, стадія σ_5). Їх поєднання в єдиній стадії не відповідає вимогам методу.

B3. Версіонований реєстр модулів. Дочірні модулі $\mathcal{M}$ зберігаються з семантичним версіонуванням. Кожне середовище прив'язується до конкретного тегу, що формує незмінний знімок $\hat{C}_e^v$ (2.8).

B4. Автоматизований моніторинг дрейфу. Платформа підтримує конфігурований інтервал τ для періодичного виконання refresh-only планування без застосування змін (крок 2 Алгоритму 4).

B5. Ієрархічне успадкування конфігурацій. Змінні та політики успадковуються за трирівневою ієрархією (глобальний $\rightarrow$ середовища $\rightarrow$ компонента) згідно

з (2.14).

B6. Версіонування стану. Платформа зберігає повну історію $\mathcal{H}_e$ (2.9) для кожного середовища — необхідна умова виконання Алгоритму 3.

B7. Диференційовані рівні затвердження. Стадія σ_6 підтримує три режими: автоматичний, ручний та примусово ручний при $\text{safe}(e, v, v') = \text{false}$.

Архітектура компонентів системи, що задовольняє вимоги B1–B7, зображена на рисунку 3.5.

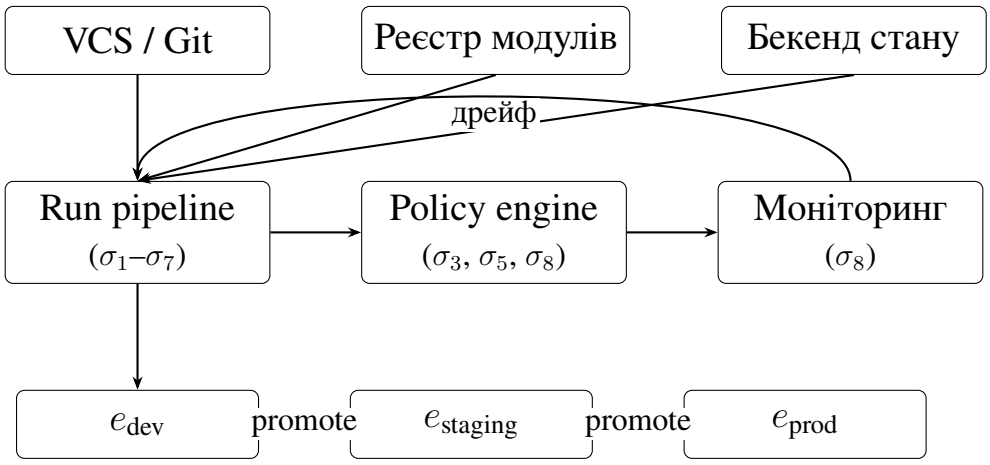


Рисунок 3.5 – Архітектура компонентів реалізації методу

3.4 Порівняльний аналіз запропонованого методу з існуючими підходами

Критерії порівняння поділяються на дві групи: структурні, що характеризують архітектурну організацію підходу, та операційні, що оцінюють його поведінку на окремих стадіях життєвого циклу.

До структурних критеріїв належать:

– повнота покриття стадій — які зі стадій $A_e = \{\sigma_1, \ldots, \sigma_8\}$ автомата явно підтримуються підходом як окремі кроки конвеєру;

– рівень ізоляції середовищ — ступінь незалежності стану, бекенду та облікових даних між середовищами $e \in \mathcal{E}$, що відповідає критерію I_S з підрозділу 2.1;

– гранулярність управління станом — можливість сегментації стану окремого середовища за функціональними компонентами інфраструктури;

– формалізація просування — наявність явно визначеної операції promote(e_i, e_j) з формальними передумовами, на відміну від ad hoc підходів через ручне копіювання конфігурацій або злиття гілок.

До операційних критеріїв належать:

– рівні перевірки відповідності — скільки з трьох рівнів (P^{code}, P^{plan}, P^{state}) підтримуються як окремі стадії з власними артефактами перевірки, що відображає розмежування σ_3 та σ_5 [22];

– механізм виявлення дрейфу — наявність автоматизованого моніторингу Δ_e^{drift} та спосіб реагування на виявлені розбіжності ($\sigma_8 \to \sigma_4$);

– підтримка версіонованого відкату — можливість повернення до $S_e^{r,v'}$ з урахуванням зворотності ресурсів;

– ієрархія успадкування конфігурацій — здатність організувати змінні та політики через ієрархічну структуру з глобальним, середовищним та компонентним рівнями згідно з (2.14).

Як об'єкти порівняння обрано чотири підходи, що представляють різні рівні зрілості управління Terraform-інфраструктурою — від мінімального вбудованого інструментарію до повнофункціональної корпоративної платформи.

Перший об'єкт — нативний Terraform CLI з робочими просторами (далі TF Workspaces). Цей підхід використовує вбудований механізм `terraform workspace`, де єдиний кореневий модуль обслуговує всі середовища через перемикання контексту виконання. Він є базовим для всіх інших підходів і демонструє можливості, доступні без зовнішніх інструментів [72].

Другий об'єкт — Terragrunt-based підхід (далі Terragrunt). Інструмент-обгортка Terragrunt реалізує directory-based патерн із механізмом успадкування конфігурацій через файли `terragrunt.hcl`, що зменшує дублювання коду між середовищами при збереженні повної ізоляції стану [15]. Цей підхід є найбільш поширеним

серед команд, що потребують мультисередовищного управління без централізованої платформи.

Третій об'єкт — типовий GitOps-конвеєр (далі GitOps CI/CD). Під цим підходом розуміється конвеєр безперервної доставки, побудований на основі CI/CD-системи загального призначення (GitLab CI, GitHub Actions або аналогічної) з інтеграцією Terraform через CLI-команди у стадіях конвеєру [22, 33]. До цієї категорії належать також спеціалізовані інструменти автоматизації pull request-орієнтованого процесу, зокрема Atlantis та tf-controller для Kubernetes, що покращують базовий конвеєр у частині автоматизації стадій планування та застосування, але не виходять за межі стадій σ_4–σ_7 [75].

Четвертий об'єкт — HCP Terraform (раніше Terraform Cloud / Enterprise). Комерційна платформа HashiCorp, що є найбільш функціональним серед існуючих рішень і водночас є комерційним продуктом із пропрієтарною ліцензією, що обмежує його застосовність для організацій з вимогами до відкритості інструментарію.

Таблиця 3.1 (с. 61) узагальнює порівняння за структурними критеріями. Нижче коментуються лише ті аспекти, що потребують пояснення понад зміст таблиці.

За повнотою покриття стадій ключовою відмінністю є те, що жоден із розглянутих підходів не розмежовує валідацію коду (σ_3) та перевірку відповідності плану (σ_5) як окремі стадії. HCP Terraform виконує перевірку політик через Sentinel або OPA після планування, проте об'єднує правила рівня коду та рівня плану в єдиному policy evaluation [72]. У запропонованому методі ці стадії є архітектурно незалежними: σ_3 перевіряє предикат $\mathrm{valid}(C_e, P_e^{\mathrm{code}})$ без звернення до провайдера, а σ_5 — предикат $\mathrm{comply}(\Delta_e^{\mathrm{plan}}, P_e^{\mathrm{plan}})$ над артефактом плану з обчисленими атрибутами ресурсів. GitOps CI/CD може інтегрувати статичні аналізатори безпеки як стадію валідації, проте перевірка семантики плану зазвичай відсутня [22]. TF Workspaces та Terragrunt обмежені `terraform validate` без перевірок політик.

Таблиця 3.1 – Порівняння підходів за структурними критеріями

Критерій	TF Workspaces	Terragrunt	GitOps CI/CD	HCP Terraform
Покриття стадій	$\sigma_1, \sigma_4, \sigma_6, \sigma_7$	$\sigma_1, \sigma_4, \sigma_6, \sigma_7$	σ_1–$\sigma_4, \sigma_6, \sigma_7$	σ_1–σ_2, σ_4–σ_8
Ізоляція середовищ	Часткова ($I_S = 0{,}5$)	Повна ($I_S = 1$)	Повна (за умови конфігурування)	Повна ($I_S = 1$)
Гранулярність стану	Один стан на простір	Повна сегментація через dependency	Залежить від конвеєру	Проєкти, run triggers, Stacks
Формалізація просування	Відсутня	Відсутня	Часткова (merge / trigger)	Часткова (run triggers)

За формалізацією просування жоден підхід не визначає операцію promote(e_i, e_j) з формальними передумовами. HCP Terraform підтримує run triggers, що ініціюють виконання у залежних робочих просторах при змінах стану, проте реагують на будь-яку зміну, а не лише на досягнення ready(e_i) [72]. GitOps CI/CD реалізує просування через злиття коду або ручний запуск конвеєру без перевірки консистентності джерела [33]. Terragrunt та TF Workspaces не мають вбудованого механізму просування.

Таблиця 3.2 (с. 62) узагальнює порівняння за операційними критеріями.

За рівнями перевірки відповідності TF Workspaces та Terragrunt не мають вбудованих механізмів перевірки політик: будь-які перевірки потребують зовнішніх інструментів, інтегрованих вручну. GitOps CI/CD дозволяє додати статичний аналіз коду (P^{code}) як стадію конвеєру, проте перевірка на рівні плану (P^{plan}) рідко реалізується, оскільки потребує парсингу серіалізованого плану та написання специфічних правил [22]. HCP Terraform підтримує два рівні: політики Sentinel/OPA можуть оперувати як кодом конфігурації, так і артефактом плану, а health assessments перевіряють стан (P^{state}) [72]. Проте, як зазначено у блоці стру-

Таблиця 3.2 – Порівняння підходів за операційними критеріями

Критерій	TF Workspaces	Terragrunt	GitOps CI/CD	HCP Terraform
Рівні перевірки відповідності	Відсутні	Відсутні	P^{code} (через зовнішні інструменти)	P^{code}, P^{plan} (Sentinel/OPA); P^{state} (health assessments)
Виявлення дрейфу	Ручне	Ручне	Ручне або cron	Автоматичне (periodic refresh-only) без класифікації
Версіонований відкат	Ручне відновлення бекенду	Ручне відновлення бекенду	git revert без аналізу зворотності	Історія станів без предикату безпечності
Ієрархія успадкування	Відсутня	Змінні та бекенд (через include)	Шаблони CI (не стандартизовано)	Variable sets та policy sets за проєктами

ктурних критеріїв, рівні коду та плану не розмежовані в окремі стадії конвеєру — політики обох рівнів оцінюються в єдиному policy evaluation після планування. Запропонований метод є єдиним, що архітектурно виокремлює всі три рівні як стадії σ_3, σ_5 та σ_8 з різними вхідними артефактами, моментами зворотного зв'язку та предикатами перевірки.

За механізмом виявлення дрейфу TF Workspaces, Terragrunt та GitOps CI/CD не мають вбудованого моніторингу: виявлення дрейфу потребує ручного виконання `terraform plan` або побудови окремого cron-конвеєру, що періодично порівнює стани [20]. HCP Terraform реалізує автоматичні health assessments — періодичні refresh-only плани, що виявляють розбіжності між S_e^r та S_e^a і відображають результати у інтерфейсі платформи [72]. Однак HCP Terraform не класифікує виявлений дрейф за рівнем критичності та не ініціює автоматичне усунення: оператор

обирає між відновленням бажаного стану (queue plan) та прийняттям змін (оновлення конфігурації) вручну. Запропонований метод доповнює виявлення трикласовою класифікацією (2.15) та ефективною дельтою (2.16), де критичний дрейф усувається автоматично через перехід $\sigma_8 \to \sigma_4$, операбельний потребує підтвердження оператора, а інформаційний ігнорується для уникнення хибних сповіщень.

За підтримкою версіонованого відкату жоден із розглянутих підходів не формалізує процедуру повернення до попереднього стану. TF Workspaces та Terragrunt покладаються на версіонування бекенду (наприклад, S3 object versioning), проте відновлення є ручною операцією без перевірки зворотності ресурсів. GitOps CI/CD реалізує відкат через `git revert` з повторним застосуванням, що не враховує ресурси з $\text{rev}(r) = 0$ (2.11). HCP Terraform зберігає історію станів робочого простору та дозволяє відновити попередню версію, проте не оцінює безпечність відкату згідно з предикатом (2.12) і не вимагає затвердження для незворотних ресурсів.

За ієрархією успадкування конфігурацій TF Workspaces не має механізму успадкування: змінні визначаються на рівні окремого робочого простору без можливості каскадного визначення. Terragrunt забезпечує успадкування конфігурацій через функцію `include` та `read_terragrunt_config`, що дозволяє визначати спільні змінні на рівні каталогу з перевизначенням на рівні середовища. Проте успадкування у Terragrunt охоплює лише змінні та параметри бекенду, а не політики відповідності. GitOps CI/CD може реалізувати успадкування через шаблони конвеєру та спільні змінні середовища CI-системи, що є гнучким, але не стандартизованим підходом. HCP Terraform підтримує ієрархію через variable sets, що можуть бути призначені на рівні організації, проєкту або окремого робочого простору, а policy sets дозволяють призначати набори політик за аналогічною ієрархією. Це найближчий до запропонованого методу підхід, проте у HCP Terraform ієрархії змінних та політик є незалежними механізмами, тоді як у запропонованому методі вони об'єднані єдиною формулою успадкування (2.14).

3.5 Наукова новизна та теоретичний внесок

Порівняльний аналіз у підрозділі 3.4 засвідчив, що існуючі підходи до управління Terraform-інфраструктурою для кількох середовищ покривають окремі аспекти життєвого циклу, проте жоден із них не забезпечує повного покриття стадій від ініціалізації до усунення дрейфу, не формалізує операцію просування між середовищами з явними передумовами та не розмежовує перевірку відповідності коду і семантики плану як архітектурно незалежні стадії. Теоретичний внесок дисертаційної роботи полягає у розробленні узагальненого методу, що усуває ці прогалини через єдину формальну базу. На основі отриманих результатів формулюються три пункти наукової новизни.

1. Вперше запропоновано двопроекційну модель життєвого циклу Terraform-інфраструктури для кількох середовищ у вигляді скінченного автомата A_e з восьми стадій та простору середовищ $(\mathcal{E}, \preceq)$, що відрізняється від існуючих виокремленням перевірки відповідності плану (σ_5) як самостійної стадії між плануванням та затвердженням і замкненим циклом виявлення та усунення конфігураційного дрейфу ($\sigma_8 \to \sigma_4$), що дозволяє охопити повний життєвий цикл від ініціалізації конфігурацій до автоматичного відновлення консистентності середовищ.

2. Отримав подальший розвиток метод просування змін між середовищами Terraform-інфраструктури, що відрізняється від існуючих формалізацією передумови готовності середовища-джерела $\mathrm{ready}(e_i)$, використанням версіонованих знімків конфігурацій $\hat{C}_e^v$ та введенням предикату безпечності відкату $\mathrm{safe}(e, v, v')$ з урахуванням зворотності ресурсів, що забезпечує атомарність просування, мінімізацію радіуса ураження та контрольований відкат до попереднього стану.

3. Удосконалено підхід до перевірки відповідності інфраструктурних конфігурацій за рахунок багаторівневої моделі (P_e^{code}, P_e^{plan}, P_e^{state}) з ієрархічним успадкуванням політик та трикласовою класифікацією конфігураційного дрейфу з ефективною дельтою, що дозволило забезпечити розмежування перевірок

за артефактами та моментами зворотного зв'язку, зростання суворості політик вздовж ланцюга просування та зменшення хибних сповіщень при моніторингу стану.

Обґрунтування першого пункту новизни спирається на відсутність формалізованої моделі життєвого циклу мультисередовищної Terraform-інфраструктури. Pahl зі співавторами запропонували концептуальний DevOps-цикл для інфраструктурного коду, проте без формалізації переходів та без урахування координації між середовищами [51]. Типові CI/CD конвеєри реалізують лінійну послідовність validate–plan–approve–apply без зворотних переходів [22,33]. HCP Terraform є найбільш повним (таблиця 3.2, с. 62), проте його робочий процес описаний на рівні документації продукту, а не теоретичної моделі з явними умовами переходів [72].

Архітектурне розмежування σ_3 та σ_5 обґрунтовується різницею вхідних артефактів та класів виявлених порушень. Конфігурація може пройти валідацію σ_3 (синтаксично коректний код без відомих вразливостей [57, 67]), але порушити політику σ_5, коли обчислені атрибути плану виявляють недопустимий тип екземпляра або заплановане видалення захищеного ресурсу. Як показано у таблиці 3.1 (с. 61), жоден із розглянутих підходів не виокремлює ці два класи перевірок як окремі стадії конвеєру.

Замкнений цикл $\sigma_8 \to \sigma_4$ відрізняється від існуючих підходів тим, що усунення дрейфу є частиною функції переходів автомата, а не зовнішнім ручним процесом. HCP Terraform автоматизує виявлення через health assessments, проте пропонує оператору два варіанти усунення без автоматичного ініціювання планування [72]. Диференціація реагування за класами дрейфу (2.15) — автоматичне усунення критичного, підтвердження для операбельного, ігнорування інформаційного — є внеском, що відсутній у всіх розглянутих підходах.

Вертикальна проекція, простір $(\mathcal{E}, \preceq)$, також не має прямих аналогів: існуючі підходи передбачають або неформалізований лінійний ланцюг, або довільну топологію без обмежень на порядок просування [15, 33]. Формалізація через частково

впорядковану множину описує як лінійні, так і розгалужені топології в єдиному апараті, де конкурентне просування незалежних ланцюгів є наслідком принципу повної ізоляції стану.

Другий пункт новизни стосується формалізації просування змін між середовищами. Як засвідчив аналіз у таблиці 3.1, жоден із розглянутих підходів не визначає операцію promote(e_i, e_j) як атомарну процедуру з явними передумовами. Run triggers HCP Terraform реагують на будь-яку зміну стану залежного робочого простору, не перевіряючи консистентність джерела [72]; GitOps-конвеєри реалізують просування через злиття коду без верифікації порожності дельт Δ_e^{plan} та Δ_e^{drift} [33]. На практиці це означає, що зміна може бути просунута до виробничого середовища з джерела, яке містить невиявлений дрейф або незавершене застосування.

Введення формальної передумови ready(e_i) згідно з (2.6) усуває цю прогалину: просування дозволяється лише за умови, що середовище-джерело перебуває у стадії σ_8 з порожніми дельтами планування та дрейфу. Це є необхідною умовою, яку не перевіряє жоден із розглянутих підходів автоматично.

Версіоновані знімки конфігурацій $\hat{C}_e^v$ (2.8) забезпечують властивість, яку існуючі підходи реалізують лише частково. Terragrunt та GitOps CI/CD покладаються на Git-коміти як джерело версіонування, проте не фіксують версії дочірніх модулів із реєстру на момент тестування: між успішним застосуванням у e_i та просуванням до e_j версія модуля у реєстрі може змінитись. HCP Terraform вирішує це через приватний реєстр із семантичним версіонуванням, проте прив'язка робочого простору до конкретної версії модуля залишається відповідальністю оператора [72]. У запропонованому методі знімок $\hat{C}_e^v$ є незмінним артефактом, що включає як кореневий модуль, так і зафіксовані теги дочірніх модулів, що гарантує ідентичність коду між середовищами.

Предикат безпечності відкату safe(e, v, v') (2.12) є теоретичним внеском, що не має аналогів серед розглянутих підходів. Як зазначено у таблиці 3.2,

TF Workspaces та Terragrunt покладаються на ручне відновлення бекенду, GitOps CI/CD — на `git revert`, HCP Terraform — на історію станів. Жоден із цих механізмів не оцінює зворотність окремих ресурсів (2.11): відкат, що включає видалення бази даних із накопиченими транзакціями ($\text{rev}(r) = 0$), виконується так само, як відкат обчислювального екземпляра ($\text{rev}(r) = 1$). Запропонований метод вимагає ручного затвердження на σ_6 при $\text{safe}(e, v, v') = \text{false}$ незалежно від рівня автоматизації середовища, що запобігає ненавмисній втраті даних.

Третій пункт новизни стосується багаторівневої моделі перевірки відповідності. Як показано у таблиці 3.2, HCP Terraform є єдиним серед розглянутих підходів, що підтримує перевірку політик на кількох рівнях, проте об'єднує правила рівня коду та рівня плану в єдиному policy evaluation. Запропонована декомпозиція $P_e = P_e^{\text{code}} \cup P_e^{\text{plan}} \cup P_e^{\text{state}}$ (2.13) є узагальненням, що формалізує три незалежні точки перевірки з різними вхідними артефактами, предикатами та моментами зворотного зв'язку: невдача на σ_3 повертає автомат до σ_2 для виправлення коду ще до звернення до провайдера; невдача на σ_5 блокує застосування конкретного плану; порушення на σ_8 ініціює усунення дрейфу. Розмежування моментів зворотного зв'язку є практично значущим: чим раніше виявлено порушення, тим менша вартість його усунення [67].

Ієрархія успадкування (2.14) узагальнює механізми, що в існуючих підходах реалізовані як незалежні інструменти. У HCP Terraform variable sets та policy sets призначаються за окремими ієрархіями: змінні — через організацію, проєкт та робочий простір, політики — через policy sets [72]. У Terragrunt успадкування охоплює лише змінні та параметри бекенду [15]. Запропонована формула $P_e = P_{\text{global}} \cup P_{\text{env}(e)} \cup P_{\text{comp}(e)}$ об'єднує змінні та політики в єдину ієрархічну структуру з властивістю зростання суворості вздовж $(\mathcal{E}, \preceq)$: одна й та сама політика може бути advisory у e_{dev} та hard-mandatory у e_{prod}, що дозволяє поступово виявляти порушення без блокування розробки на ранніх стадіях.

Трикласова класифікація дрейфу (2.15) та ефективна дельта (2.16) вирішу-

ють практичну проблему хибних сповіщень, що не адресується жодним із розглянутих підходів. HCP Terraform відображає всі виявлені розбіжності без диференціації [72]; cron-based рішення у GitOps CI/CD сповіщують про будь-яку непорожню дельту [20]. На практиці значна частина виявлених розбіжностей є очікуваними змінами поза Terraform (динамічні IP-адреси, автоматично ротовані сертифікати, лічильники стану), що генерує потік сповіщень без практичної цінності. Виключення таких атрибутів через множину I_e та обчислення ефективної дельти Δ_e^{eff} зменшує кількість сповіщень до тих, що потребують реагування, а класифікація визначає спосіб реагування без участі оператора для критичних порушень.

Запропонований метод має визначені межі застосовності, що випливають з прийнятих архітектурних рішень та обсягу дослідження. По-перше, метод оперує абстракціями конфігурації C, стану S та дельти Δ, що спираються на семантику HCL та механізм явного управління станом через файл `terraform.tfstate`. Це забезпечує застосовність до Terraform, OpenTofu та інших HCL-сумісних рушіїв (принцип 6), проте виключає інструменти з принципово іншою архітектурою: безстанові, агентні або засновані на власних мовах моделювання. По-друге, модель ґрунтується на directory-based патерні ізоляції, обраному за результатами порівняльного аналізу у підрозділі 2.1. Для організацій, що використовують інші патерни, окремі компоненти методу залишаються застосовними, проте повна модель життєвого циклу потребує адаптації. По-третє, метод не включає оцінку вартості інфраструктурних змін як формалізовану стадію автомата A_e. Практична реалізація може інтегрувати зовнішні інструменти аналізу вартості на стадіях σ_3 або σ_5 як додаткові правила у відповідних множинах політик, проте формалізація такої інтеграції виходить за межі поточного дослідження.

4 ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНА ОЦІНКА МЕТОДУ

4.1 Реалізація методу на базі платформи Scalr

4.1.1 Експериментальна інфраструктура та відображення моделі

Для верифікації запропонованого узагальненого методу обрано платформу Scalr [62] — промислову систему оркестрації Terraform/OpenTofu-інфраструктури, що реалізує directory-based патерн ізоляції, ієрархічне успадкування конфігурацій та вбудовану перевірку відповідності через Open Policy Agent (OPA). Архітектура платформи забезпечує пряме відображення кожного компонента формальної моделі з розділу 2 на конкретний механізм реалізації.

Експериментальна інфраструктура розгорнута у хмарному середовищі AWS і складається з типового веб-сервісу: мережевий рівень (VPC, підмережі, security groups), обчислювальний рівень (EC2), рівень зберігання (RDS PostgreSQL, S3) та управління доступом (IAM). Конфігурація організована як бібліотека з п'яти дочірніх модулів $\mathcal{M} = \{m_{\text{net}},\ m_{\text{compute}},\ m_{\text{db}},\ m_{\text{storage}},\ m_{\text{iam}}\}$, що викликаються спільним кореневим модулем m_{root}. Загальний обсяг — близько 1 200 рядків HCL, 27 керованих ресурсів. Простір середовищ: $e_{\text{dev}} \preceq e_{\text{staging}} \preceq e_{\text{prod}}$, де середовища відрізняються параметрами масштабування, обліковими даними та рівнем суворості політик.

Таблиця 4.1 (с. 70) узагальнює відображення компонентів формальної моделі на механізми платформи.

Ключовою архітектурною відмінністю від типових CI/CD конвеєрів є наявність двох незалежних точок перевірки OPA-політик у run pipeline. Перша точка відповідає стадії σ_3 та оперує файлами конфігурації C_e, перевіряючи предикат $\text{valid}(C_e, P_e^{\text{code}})$ без звернення до хмарного провайдера. Друга точка відповідає σ_5 та оперує JSON-представленням плану, перевіряючи предикат $\text{comply}(\Delta_e^{\text{plan}}, P_e^{\text{plan}})$. Таке розмежування дозволяє виявляти різні класи порушень

Таблиця 4.1 – Відображення компонентів методу на архітектуру Scalr

Компонент моделі	Реалізація у Scalr
Простір середовищ $(\mathcal{E}, \preceq)$	Ієрархія Account → Environment → Workspace; кожен Environment відповідає $e \in \mathcal{E}$
Ізоляція стану $(I_S = 1)$	Вбудована ізоляція для кожного Workspace
Автомат A_e (σ_1–σ_8)	Run pipeline: init → validate+OPA → plan → OPA → approve → apply → drift detection
P_e^{code} (стадія σ_3)	OPA-політики, що оперують конфігурацією до планування
P_e^{plan} (стадія σ_5)	OPA-політики, що оперують JSON-артефактом плану після планування
P_e^{state} (стадія σ_8)	OPA-політики при drift detection
Ієрархія (2.14)	Успадкування змінних та політик: Account → Environment → Workspace
Знімки $\hat{C}_e^v$ (2.8)	Приватний реєстр модулів із семантичним версіонуванням
Рівні застосування політик	advisory / soft-mandatory / hard-mandatory
Моніторинг σ_8	Drift detection — періодичний refresh-only plan з інтервалом τ

на різних стадіях: перша точка фіксує захардкоджені секрети та незашифровані ресурси у коді, друга — заборонені типи екземплярів або видалення захищених ресурсів у семантиці плану. Рівень застосування визначається атрибутом середовища: типова конфігурація передбачає advisory для e_{dev} та hard-mandatory для e_{prod} за однакового набору правил.

Моніторинг (σ_8) реалізується через drift detection — періодичне виконання `terraform plan` у режимі refresh-only з конфігурованим інтервалом τ. При виявленні непорожньої Δ_e^{drift} платформа ініціює зворотний перехід $\sigma_8 \to \sigma_4$. Класифікація дрейфу за трьома класами (2.15) реалізується через OPA-політики рівня стану P_e^{state}, а ефективна дельта (2.16) — через конфігурацію `ignore_changes` для атрибутів, що очікувано змінюються поза Terraform.

Технічний механізм передачі JSON-артефакту плану до OPA-рушія реалізується через систему користувацьких хуків платформи, що дозволяє виконувати довільні команди або скрипти у визначених точках конвеєру: до та після ініціалізації, планування та застосування [62]. Оскільки хуки та основні команди Terraform виконуються в одному контейнері, артефакт плану, збережений командою `terraform show --json`, доступний скрипту хука після стадії σ_4 без додаткової передачі між процесами. Хук типу «after-plan» зчитує JSON-файл плану та передає його до OPA-рушія як вхідний документ, після чого повертає результат оцінки платформі для прийняття рішення про продовження або блокування конвеєру. Така архітектура дозволяє писати правила OPA на мові Rego безпосередньо проти атрибутів конкретних ресурсів у плані (наприклад, перевіряти тип екземпляра EC2 або наявність флагу видалення на захищеному ресурсі), що є принциповою відмінністю від статичного аналізу коду, де обчислені значення атрибутів ще невідомі. Рівень застосування правила (advisory, soft-mandatory або hard-mandatory) визначається метаданими, анотованими безпосередньо в тілі Rego-правила, і успадковується через ієрархію Account $\rightarrow$ Environment $\rightarrow$ Workspace відповідно до (2.14).

4.1.2 Реалізація просування змін та механізму відкату

Операція просування promote(e_i, e_j) (2.5) реалізується через оновлення версії модуля у Workspace цільового середовища. Приватний реєстр модулів платформи зберігає кожну версію дочірнього модуля під семантичним тегом; Workspace прив'язується до конкретного тегу, що формує незмінний знімок $\hat{C}_e^v$ (2.8). Просування зводиться до підвищення тегу у цільовому Workspace до версії, успішно застосованої у середовищі-джерелі, з підстановкою середовищно-специфічних параметрів Θ_{e_j}. Ця дія автоматично ініціює run pipeline у e_j, що проходить повний ланцюг $\sigma_3 \rightarrow \sigma_5 \rightarrow \sigma_6 \rightarrow \sigma_7$ з політиками P_{e_j}.

Передумова готовності ready(e_i) (2.6) перевіряється через стан останнього run у Workspace середовища-джерела: просування дозволяється лише за умови,

що останній run завершився успішно (статус `applied`), drift detection не зафіксував розбіжностей, і жодних незавершених run не існує. Це гарантує, що до цільового середовища просувається лише перевірена та консистентна конфігурація. Для каскадного просування через повний ланцюг $e_{\text{dev}} \preceq e_{\text{staging}} \preceq e_{\text{prod}}$ кожний наступний крок ініціюється лише після досягнення $\text{ready}(e_j)$ у проміжному середовищі, що відповідає композиції (2.7).

Механізм відкату спирається на історію станів (2.9), яку платформа зберігає для кожного Workspace. Операція $\text{rollback}(e, v')$ (2.10) виконується через відновлення записаного стану $S_e^{r,v'}$ з історії та запуск скороченого циклу $\sigma_4 \to \sigma_5 \to \sigma_6 \to \sigma_7$, що приводить фактичний стан S_e^a у відповідність до відновленого. Предикат безпечності $\text{safe}(e, v, v')$ (2.12) реалізується через аналіз JSON-плану на стадії σ_4: якщо план містить видалення ресурсів із $\text{rev}(r) = 0$ (2.11), зокрема баз даних або сховищ із накопиченими даними, стадія затвердження σ_6 примусово переводиться у ручний режим незалежно від налаштувань автоматизації Workspace, що запобігає ненавмисній втраті даних.

Налаштування drift detection у реалізованій інфраструктурі відображає ієрархію критичності середовищ: для e_{prod} встановлено інтервал $\tau = 1$ год, для e_{staging} — $\tau = 6$ год, для e_{dev} — виявлення за запитом без автоматичного циклічного моніторингу. Такий розподіл обґрунтовується балансом між операційними витратами на виконання refresh-only планування та ризиком тривалого перебування виробничого середовища у стані дрейфу. Кожен запуск drift detection споживає обчислювальні ресурси агента та генерує звернення до API хмарного провайдера, тому надто короткий інтервал для некритичних середовищ створює непотрібне навантаження без практичної користі.

Множина ігнорованих атрибутів I_e у реалізованій конфігурації охоплює три категорії атрибутів. До першої належать динамічні мережеві атрибути: приватні та публічні IP-адреси EC2-екземплярів, що змінюються при кожному перезапуску, та DNS-імена балансувальника навантаження. До другої — автоматично керовані

атрибути безпеки: ідентифікатори ротованих сертифікатів та токени доступу з обмеженим терміном дії, що оновлюються зовнішніми процесами поза Terraform. До третьої — лічильники та метадані стану: кількість з'єднань до RDS-інстансу та мітки часу останнього резервного копіювання. Всі три категорії позначені директивою `ignore_changes` у відповідних ресурсних блоках HCL-конфігурації та виключаються з ефективної дельти (2.16) до початку класифікації, що унеможливлює генерацію хибних сповіщень для очікуваної динаміки середовища.

4.2 Експериментальна оцінка ефективності методу

4.2.1 Методика експерименту та система метрик

Експериментальна верифікація методу керування життєвим циклом інфраструктури як код має низку особливостей порівняно з типовими експериментами в галузі програмної інженерії. Перша особливість полягає в тому, що об'єктом дослідження є не алгоритм з чітко визначеною обчислювальною складністю, а методологічний каркас, який охоплює взаємодію програмних, процесних та організаційних компонентів. Тому кількісні показники мають поєднуватись із якісним аналізом структурних властивостей, що не зводяться до числових характеристик. Друга особливість пов'язана з тим, що інфраструктура як об'єкт управління має стохастичну природу виконання: той самий план застосування може тривати різний час залежно від завантаженості API хмарного провайдера, мережевих затримок та поточного стану регіонального сервісу. Тому для усіх часових показників виконувалося кілька повторень з фіксацією діапазону значень, а не одиничне вимірювання.

Третьою особливістю є те, що частина ключових властивостей методу проявляється лише на масштабі, що перевищує можливості лабораторного експерименту. Наприклад, переваги багаторівневої перевірки відповідності повною мірою розкриваються в організаціях із десятками команд та сотнями середовищ, де ручне адміністрування політик стає практично неможливим. У межах цього дослідже-

ння такі властивості оцінювалися непрямо, через перевірку того, що механізми методу підтримують необхідний рівень абстракції та ієрархічної композиції, достатній для масштабування без принципових змін. Повна емпірична верифікація масштабованості залишається предметом подальших досліджень, що потребують доступу до промислових інфраструктур відповідного розміру.

Побудова системи метрик здійснювалась із урахуванням принципу повноти покриття, згідно з яким кожна заявлена властивість методу має бути пов'язана щонайменше з одним вимірюваним показником. Цей принцип дозволяє уникнути ситуації, коли теоретично обґрунтована перевага методу не знаходить підтвердження в емпіричних даних через відсутність відповідного інструменту вимірювання. Зворотний принцип, який також дотримано, полягає в уникненні показників, що не відображають жодної цільової властивості і додають складності оцінки без приросту інформативності. Відбір шести метрик, описаних нижче, є результатом балансу між цими двома вимогами і відповідає мінімальному набору, достатньому для обґрунтованого висновку щодо досягнення цілей дослідження.

Для кількісної оцінки ефективності запропонованого методу визначено шість метрик, що відповідають ключовим властивостям, обґрунтованим у розділі 3: повноті покриття стадій, розмежуванню рівнів перевірки відповідності, ефективності виявлення дрейфу та автоматизації просування.

Метрика $R_{\text{pre-apply}}$ (частка порушень політик, виявлених до стадії застосування σ_7) визначається як відношення кількості порушень, зафіксованих на стадіях σ_3 або σ_5, до загальної кількості внесених порушень. Метрика T_{detect} фіксує час від внесення порушення до його виявлення відповідною стадією конвеєру. Метрика T_{drift} визначає час від зовнішньої зміни ресурсу до завершення усунення дрейфу, включаючи виявлення, класифікацію та застосування плану відновлення. Метрика F_{false}, частка хибних сповіщень дрейфу, визначається як відношення сповіщень про очікувані зміни (інформаційний клас) до загальної кількості виявлених розбіжностей. Метрика N_{manual} фіксує кількість ручних кроків, необхідних для повного

циклу просування $e_{\text{dev}} \rightarrow e_{\text{staging}} \rightarrow e_{\text{prod}}$. Повнота покриття стадій оцінюється якісно — як перелік стадій σ_1–σ_8, що реально виконуються у кожному підході.

Як об'єкти порівняння обрано три підходи: (1) ручне виконання Terraform CLI без автоматизації конвеєру; (2) типовий GitOps CI/CD-конвеєр на базі GitLab CI з інтеграцією статичного аналізатора безпеки на стадії валідації; (3) запропонований метод, реалізований на платформі Scalr згідно з підрозділом 4.1. Для кожного підходу експерименти виконуються на ідентичній інфраструктурі та однаковому наборі сценаріїв.

4.2.2 Експеримент 1: ефективність багаторівневої перевірки відповідності

Для оцінки розмежування стадій σ_3 та σ_5 визначено чотири сценарії порушень, що належать до різних рівнів перевірки. До порушень рівня коду (P_e^{code}) належать: (C1) наявність захардкодженого секрету AWS у файлі конфігурації та (C2) визначення security group із вхідним правилом `0.0.0.0/0` на порті 22. До порушень рівня плану (P_e^{plan}) належать: (C3) використання типу екземпляра `m5.24xlarge`, забороненого політикою для e_{staging} та e_{prod}, та (C4) запланована операція видалення RDS-інстансу, позначеного як захищений ресурс. Кожне порушення вноситься окремим комітом у конфігурацію середовища e_{staging} та обробляється кожним із трьох порівнюваних підходів.

Результати (таблиця 4.2) підтверджують ключову гіпотезу третього пункту наукової новизни: архітектурне розмежування σ_3 та σ_5 забезпечує $R_{\text{pre-apply}} = 1{,}0$ — усі чотири порушення виявлено до стадії застосування σ_7. GitOps CI/CD-конвеєр виявляє лише порушення рівня коду ($R_{\text{pre-apply}} = 0{,}5$), оскільки інтегрований статичний аналізатор не оперує артефактом плану і не має доступу до семантики запланованих змін. Ручний Terraform CLI не має вбудованих механізмів перевірки політик ($R_{\text{pre-apply}} = 0{,}0$); порушення виявляються лише візуально оператором під час перегляду виводу `terraform plan` або постфактум у виро-

Таблиця 4.2 – Стадія виявлення порушень за підходами

Сценарій	Рівень	Terraform CLI	GitOps CI/CD	Метод (Scalr)
C1	P^{code}	Не виявлено	σ_3 (статичний аналіз)	σ_3 (OPA)
C2	P^{code}	Не виявлено	σ_3 (статичний аналіз)	σ_3 (OPA)
C3	P^{plan}	Не виявлено	Не виявлено	σ_5 (OPA над планом)
C4	P^{plan}	Не виявлено	Не виявлено	σ_5 (OPA над планом)

бничому середовищі.

Архітектурна причина неможливості виявлення порушень рівня плану в типовому GitOps CI/CD-конвеєрі полягає у відсутності стандартизованого механізму передачі JSON-артефакту між стадіями конвеєру. Статичні аналізатори на кшталт tfsec та Checkov працюють безпосередньо з HCL-файлами конфігурації до виконання планування і не мають доступу до обчислених значень атрибутів ресурсів, що з'являються лише у результаті `terraform plan`. Додавання перевірки на рівні плану до GitOps CI/CD потребує ручної розробки скрипту, що зберігає артефакт плану у JSON-форматі, передає його до OPA або аналогічного рушія та інтерпретує результат оцінки як умову блокування подальших стадій конвеєру. Такий підхід є технічно можливим, проте виходить за межі стандартної конфігурації та вимагає підтримки нестандартного коду конвеєру для кожного проєкту. Саме ця складність пояснює, чому перевірка рівня плану залишається відсутньою в більшості реальних GitOps-конвеєрів попри теоретичну можливість її реалізації [22]. Запропонований метод усуває цю прогалину через архітектурне виокремлення стадії σ_5 як обов'язкового елемента автомата A_e, що унеможливлює пропуск перевірки рівня плану незалежно від конфігурації конкретного проєкту.

Час виявлення T_{detect} для запропонованого методу становить 15–40 с для порушень рівня коду (залежно від обсягу конфігурації) та 45–90 с для порушень рівня плану (включає час виконання `terraform plan` та OPA-оцінку JSON-артефакту).

У GitOps CI/CD час виявлення порушень рівня коду є аналогічним (20–50 с), проте порушення рівня плану залишаються невиявленими незалежно від часу виконання конвеєру.

Графічне порівняння часу виявлення за підходами подано на рисунку 4.1: ручний Terraform CLI не фіксує жодного з чотирьох сценаріїв, GitOps CI/CD виявляє лише порушення рівня коду (C1, C2), а запропонований метод забезпечує виявлення всіх чотирьох сценаріїв — як рівня коду на стадії σ_3, так і рівня плану на стадії σ_5.

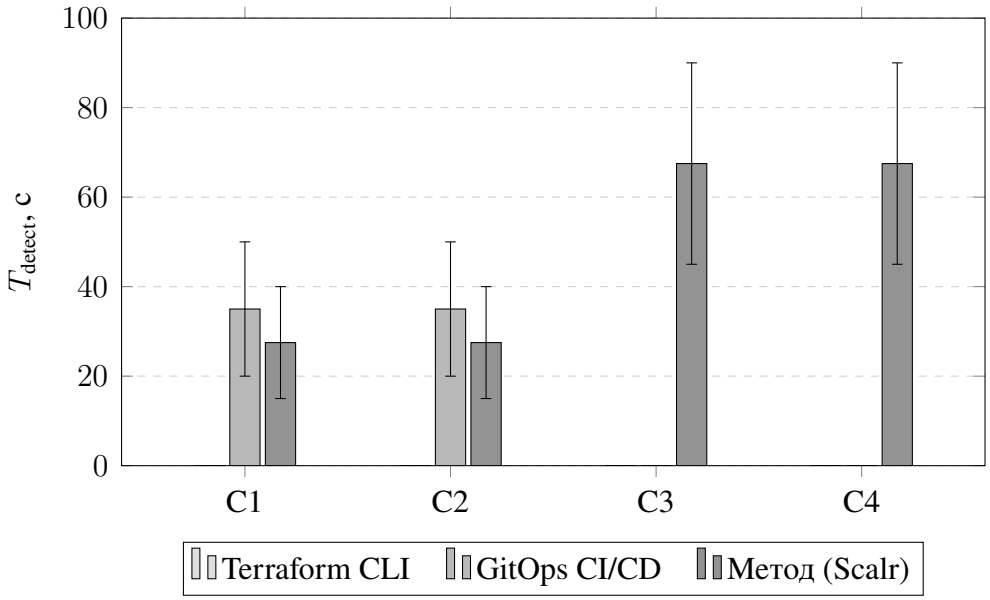


Рисунок 4.1 – Час виявлення порушень T_{detect} за сценаріями C1–C4 (стовпці — середина діапазону, планки похибок — межі діапазону; $T_{\text{detect}} = 0$ позначає «не виявлено»)

4.2.3 Експеримент 2: виявлення та класифікація конфігураційного дрейфу

Для оцінки механізму виявлення дрейфу та ефективності трикласової класифікації (2.15) імітовано три типи зовнішніх змін, внесених вручну через консоль AWS поза Terraform. (Д1) Критичний дрейф: додавання вхідного правила `0.0.0.0/0:443` до security group виробничого середовища, що порушує hard-mandatory політику P_e^{state}. (Д2) Операбельний дрейф: зміна тегу `Environment` на

EC2-екземплярі зі значення `staging` на `test`, що не порушує hard-mandatory політик, проте створює розбіжність між S_e^r та S_e^a. (Д3) Інформаційний дрейф: зміна приватної IP-адреси EC2-екземпляра після його перезапуску — атрибут, позначений директивою `ignore_changes` у конфігурації та виключений з ефективної дельти (2.16).

Інтервал drift detection встановлено $\tau = 5$ хв для прискорення збору даних. Для кожного сценарію вимірюються T_{drift} (від моменту зовнішньої зміни до завершення усунення або фіксації), коректність класифікації та наявність хибних сповіщень.

Таблиця 4.3 – Результати виявлення та класифікації дрейфу

Сценарій	Клас	Terraform CLI	GitOps CI/CD	Метод (Scalr)
Д1	Критичний	Не виявлено	Не виявлено*	Виявлено, усунено автоматично
Д2	Операбельний	Не виявлено	Не виявлено*	Виявлено, очікує підтвердження
Д3	Інформаційний	Хибне сповіщення**	Хибне сповіщення**	Виключено з Δ_e^{eff}

* — за наявності cron-конвеєру виявлення можливе, проте без класифікації та автоматичного усунення.
** — при ручному виконанні `terraform plan` дрейф відображається без диференціації за класами.

Для запропонованого методу T_{drift} становить $\tau + T_{\text{run}}$, де T_{run} — час виконання повного циклу $\sigma_4 \to \sigma_5 \to \sigma_6 \to \sigma_7$. За умов експерименту $T_{\text{drift}} \approx 7$ хв для сценарію Д1 (автоматичне усунення) та ≈ 6 хв для Д2 (до моменту сповіщення оператора). Для Terraform CLI та GitOps CI/CD без спеціалізованого cron-конвеєру дрейф залишається невиявленим до наступного ручного виконання `terraform plan`.

Частка хибних сповіщень F_{false} для запропонованого методу дорівнює $0{,}0$: сценарій Д3 виключається ефективною дельтою (2.16) і не генерує сповіщення. Для ручного CLI та GitOps CI/CD $F_{\text{false}} = 0{,}33$ (одне хибне сповіщення з трьох виявлених розбіжностей), оскільки ці підходи відображають усі зміни без диференціації. На практиці в інфраструктурах із десятками динамічних атрибутів частка хибних сповіщень без ефективної дельти є значно вищою, що підтверджує практичну значущість трикласової класифікації.

Результати виявлення та класифікації дрейфу за трьома сценаріями наведено у таблиці 4.3 (с. 78).

4.2.4 Експеримент 3: просування та відкат

Для верифікації просування виконано повний цикл $e_{\text{dev}} \to e_{\text{staging}} \to e_{\text{prod}}$ з однаковою зміною конфігурації — додаванням нового S3-сховища з політикою шифрування. Для кожного підходу зафіксовано кількість ручних кроків N_{manual} та перевірено спрацювання передумови ready(e_i).

Для Terraform CLI просування потребує ручного виконання `terraform plan` та `terraform apply` у кожному середовищі окремо, перемикання контексту облікових даних та візуальної верифікації результату — загалом $N_{\text{manual}} = 9$ (по три кроки на середовище). GitOps CI/CD зменшує кількість ручних дій до $N_{\text{manual}} = 4$ (ініціювання merge або trigger для кожного наступного середовища та ручне затвердження для e_{prod}), проте не перевіряє консистентність середовища-джерела перед просуванням. Запропонований метод забезпечує $N_{\text{manual}} = 1$ — ручне затвердження лише на стадії σ_6 для e_{prod}; просування до e_{staging} та перевірка ready(e_{dev}) виконуються автоматично.

Розподіл ручних кроків за середовищами для трьох підходів показано на рисунку 4.2 (с. 80): у Terraform CLI ручна праця рівномірно розподілена по всіх середовищах, у GitOps CI/CD зосереджена переважно в e_{prod} (merge і затвердження), а запропонований метод залишає єдиний ручний крок — затвердження на

стадії σ_6 у e_{prod} відповідно до принципу 4 (вбудована відповідність) для критичних середовищ.

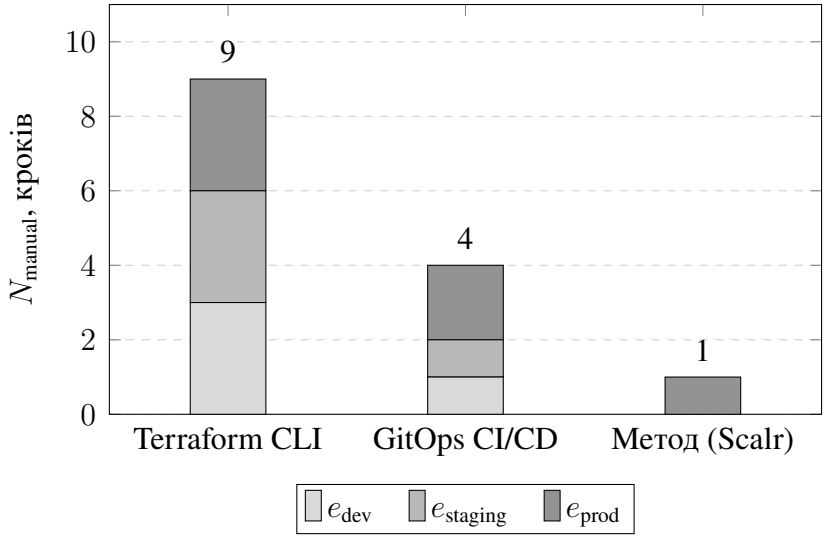


Рисунок 4.2 – Кількість ручних кроків N_{manual} при просуванні через ланцюг $e_{\text{dev}} \preceq e_{\text{staging}} \preceq e_{\text{prod}}$ з розбивкою за середовищами (підпис над стовпцем — сумарне значення)

Для верифікації механізму відкату внесено зміну, що включає модифікацію RDS-інстансу (ресурс із $\text{rev}(r) = 0$) та створення нового EC2-екземпляра ($\text{rev}(r) = 1$). Після успішного застосування в e_{staging} ініційовано $\text{rollback}(e_{\text{staging}}, v')$. Аналіз JSON-плану на стадії σ_4 зафіксував заплановане видалення RDS-інстансу, внаслідок чого предикат $\text{safe}(e, v, v') = \text{false}$ (2.12) примусово перевів стадію σ_6 у ручний режим. У Terraform CLI та GitOps CI/CD аналогічний відкат через `git revert` виконується без аналізу зворотності — оператор не отримує попередження про ризик втрати даних.

4.2.5 Узагальнення результатів

Таблиця 4.4 (с. 81) узагальнює результати трьох експериментів за визначеними метриками.

Результати підтверджують кожен із трьох пунктів наукової новизни, сфор-

Таблиця 4.4 – Зведене порівняння підходів за метриками ефективності

Метрика	Terraform CLI	GitOps CI/CD	Метод (Scalr)
Покриття стадій	$\sigma_1, \sigma_4, \sigma_6, \sigma_7$	σ_1–$\sigma_4, \sigma_6, \sigma_7$	σ_1–σ_8
$R_{\text{pre-apply}}$	0,0	0,5	1,0
T_{detect} (код / план)	— / —	20–50 с / —	15–40 с / 45–90 с
T_{drift}	Не виявляється	Не виявляється	$\tau + T_{\text{run}} \approx 7$ хв
F_{false}	0,33	0,33	0,0
N_{manual}	9	4	1

мульованих у підрозділі 3.4. Повне покриття стадій σ_1–σ_8 із замкненим циклом $\sigma_8 \to \sigma_4$ підтверджує першу новизну — двопроекційну модель життєвого циклу. Автоматична перевірка ready(e_i), каскадне просування з $N_{\text{manual}} = 1$ та спрацювання предикату safe(e, v, v') для незворотних ресурсів підтверджують другу новизну — формалізацію просування та відкату. Значення $R_{\text{pre-apply}} = 1{,}0$ за рахунок розмежування σ_3 і σ_5 та $F_{\text{false}} = 0{,}0$ за рахунок трикласової класифікації дрейфу підтверджують третю новизну — багаторівневу модель перевірки відповідності.

Експеримент має визначені обмеження. Інфраструктура розгорнута в одному хмарному провайдері (AWS) з 27 керованими ресурсами, що не відображає масштабу промислових середовищ із сотнями ресурсів та кількома провайдерами. Значення T_{detect} та T_{drift} залежать від конкретної конфігурації та потужності виконавчого середовища. GitOps CI/CD-конвеєр представлений базовою конфігурацією; додаткова інтеграція інструментів аналізу плану могла б підвищити $R_{\text{pre-apply}}$, проте потребувала б ручної розробки парсерів JSON-плану, що виходить за межі типової конфігурації конвеєру.

Експериментальна оцінка виконана на інфраструктурі фіксованого масштабу в одному хмарному провайдері, що визначає межі узагальнення отриманих результатів. У цьому підрозділі аналізуються умови, за яких висновки зберігають силу, та характер змін, що потребує адаптація методу до відмінних сценаріїв розгортання.

Результати щодо $R_{\text{pre-apply}}$ та F_{false} є структурними властивостями методу, а не артефактами конкретної інфраструктури. Значення $R_{\text{pre-apply}} = 1{,}0$ випливає з архітектурного виокремлення стадії σ_5 як обов'язкового кроку автомата A_e: поки OPA-правила коректно описують заборонені операції, жодне порушення рівня плану не може оминути цю стадію незалежно від кількості ресурсів чи провайдерів. Аналогічно, $F_{\text{false}} = 0{,}0$ є наслідком явного визначення множини I_e для кожного середовища, а не результатом характеристик конкретної інфраструктури. Обидва показники залишаються стабільними при масштабуванні за умови актуальності OPA-правил та множини I_e.

Значення T_{detect} та T_{drift}, навпаки, залежать від масштабу інфраструктури та потужності виконавчого середовища. Зі зростанням кількості керованих ресурсів час виконання `terraform plan` лінійно збільшується через необхідність звернення до API хмарного провайдера для кожного ресурсу. Для інфраструктур із сотнями ресурсів реалістичним є $T_{\text{drift}} = \tau + 15$–$30$ хв замість зафіксованих 7 хв. Це не змінює якісного висновку про перевагу методу над альтернативами, що не виявляють дрейф взагалі, проте є важливим для визначення прийнятного інтервалу τ у промислових умовах.

Показник $N_{\text{manual}} = 1$ зберігається незалежно від кількості середовищ у ланцюзі $(\mathcal{E}, \preceq)$, оскільки ручне затвердження вимагається лише для e_{prod}, тоді як усі проміжні кроки автоматизовані. Для нелінійних топологій із паралельними гілками просування N_{manual} зростає пропорційно кількості виробничих середовищ у ланцюзі, що є закономірним наслідком вимоги обов'язкового ручного затвердження для критичних середовищ.

Адаптація методу до мультихмарних конфігурацій із кількома провайдерами не потребує змін у формальній моделі, оскільки абстракції конфігурації C, стану S та дельти Δ є провайдеро-незалежними на рівні HCL та механізму `terraform.tfstate`. Практична адаптація зводиться до розширення множини I_e атрибутами, специфічними для кожного провайдера, та розробки OPA-правил,

що враховують міжпровайдерні залежності у плані змін. Атрибут зворотності $\mathrm{rev}(r)$ (2.11) також потребує уточнення для ресурсів зберігання даних різних провайдерів, оскільки семантика деструктивних операцій відрізняється між AWS, GCP та Azure.

4.3 Висновки

У четвертому розділі виконано практичну реалізацію та експериментальну оцінку запропонованого узагальненого методу керування життєвим циклом Terraform-інфраструктури для кількох середовищ, чим вирішено четверту задачу дослідження 4.

Реалізація на платформі Scalr продемонструвала пряме відображення кожного компонента формальної моделі на конкретний механізм платформи: ієрархія Account $\to$ Environment $\to$ Workspace реалізує простір $(\mathcal{E}, \preceq)$ з ізоляцією стану $I_S = 1$; run pipeline реалізує автомат A_e з повним покриттям стадій σ_1–σ_8; дві незалежні точки OPA-політик реалізують розмежування σ_3 та σ_5; drift detection реалізує замкнений цикл $\sigma_8 \to \sigma_4$.

Експериментальна оцінка на інфраструктурі з 27 ресурсів та трьох середовищ підтвердила ефективність методу за шістьма метриками. Запропонований метод забезпечив $R_{\text{pre-apply}} = 1{,}0$ проти $0{,}5$ для GitOps CI/CD та $0{,}0$ для ручного CLI; $F_{\text{false}} = 0{,}0$ проти $0{,}33$ для обох альтернатив; $N_{\text{manual}} = 1$ проти 4 та 9 відповідно. Механізм drift detection забезпечив автоматичне виявлення та класифікацію дрейфу з усуненням критичних порушень протягом ≈ 7 хв, тоді як альтернативні підходи не виявляють дрейф без додаткових ручних дій.

ВИСНОВКИ

У кваліфікаційній роботі вирішено актуальну науково-технічну задачу розроблення узагальненого методу керування життєвим циклом Terraform-інфраструктури для кількох середовищ, що має теоретичне значення та практичну цінність для галузі. За результатами дослідження сформульовано наступні висновки.

1. Проаналізовано існуючі патерни ізоляції середовищ у Terraform-інфраструктурі (workspace-based, directory-based та branch-based) та обґрунтовано критерії вибору між ними за п'ятьма показниками: ізоляція стану I_S, радіус ураження B_R, конфігураційна ентропія H_C, складність просування K_P та масштабованість M. Доведено, що directory-based патерн є єдиним, що забезпечує повну ізоляцію стану та гарантований одиничний радіус ураження без додаткових умов, що обґрунтовує його вибір як базового для запропонованого методу.

2. Розроблено двопроекційну модель життєвого циклу Terraform-інфраструктури для кількох середовищ у вигляді скінченного автомата A_e з восьми стадій (σ_1–σ_8) та простору середовищ $(\mathcal{E}, \preceq)$. Модель відрізняється від існуючих виокремленням перевірки відповідності плану (σ_5) як самостійної стадії та замкненим циклом виявлення і усунення конфігураційного дрейфу ($\sigma_8 \to \sigma_4$), що забезпечує повне покриття життєвого циклу від ініціалізації до автоматичного відновлення консистентності.

3. Обґрунтовано метод управління станом та просування змін між середовищами, що включає: формалізовану передумову готовності $\text{ready}(e_i)$; версіоновані знімки конфігурацій $\hat{C}_e^v$; предикат безпечності відкату $\text{safe}(e, v, v')$ з урахуванням зворотності ресурсів; багаторівневу модель перевірки відповідності (P_e^{code}, P_e^{plan}, P_e^{state}) з ієрархічним успадкуванням та трикласову класифікацію конфігураційного дрейфу з ефективною дельтою Δ_e^{eff}. Порівняльний аналіз із чотирма існуючими підходами підтвердив, що запропонований метод є єдиним, який архітектурно виокремлює всі три рівні перевірки як незалежні стадії конвеєру.

4. Реалізовано запропонований метод на платформі Scalr та проведено експериментальну оцінку на інфраструктурі з 27 ресурсів у трьох середовищах. Результати підтвердили ефективність методу: виявлена частка порушень, становить $R_{\text{pre-apply}} = 1{,}0$ проти $0{,}5$ для GitOps CI/CD; частка хибних сповіщень дрейфу $F_{\text{false}} = 0{,}0$ проти $0{,}33$; кількість ручних кроків при просуванні $N_{\text{manual}} = 1$ проти 4 та 9 для альтернативних підходів.

Отримані результати можуть бути використані при проєктуванні конвеєрів керування мультисередовищною Terraform-інфраструктурою в організаціях, що потребують формалізованого контролю просування змін, багаторівневої перевірки відповідності та автоматизованого виявлення конфігураційного дрейфу. Перспективами подальших досліджень є адаптація методу до мультихмарних конфігурацій із кількома провайдерами, формалізація оцінки вартості змін як стадії автомата та розширення класифікації дрейфу методами машинного навчання для автоматичного визначення класу на основі історичних даних.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Adams B., McIntosh S. Modern Release Engineering in a Nutshell: Why Researchers should Care. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER) (Klagenfurt, Austria, March 14–18, 2016)*. IEEE, 2016. P. 78–90. DOI: 10.1109/SANER.2016.108

2. Adelusi J. B., Edwards F. Infrastructure as Code (IaC): Best Practices in Multi-Cloud Environments. 2024. 5 p.

3. Agarwal V., Butlers C., Degenaro L., Kumar A., Sailer A., Steinder G. Compliance-as-Code for Cybersecurity Automation in Hybrid Cloud. *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. 2022. P. 427–436. DOI: 10.1109/CLOUD55607.2022.00066.

4. Almonte L., Guerra E., Cantador I., de Lara J. Recommender systems in model-driven engineering: A systematic mapping review. *Software and Systems Modeling*. 2022. Vol. 21, iss. 1. P. 249–280. DOI: 10.1007/s10270-021-00905-x.

5. Alonso J., Piliszek R., Cankar M. Embracing IaC through the DevSecOps philosophy: Concepts, challenges, and a reference framework. *IEEE Software*. 2023. Vol. 40, no. 1. P. 56–62. DOI: 10.1109/MS.2022.3212194

6. Anderson M. J., Collins R. L., Whitman T. A., Morales J. S. Hybrid and Multi-Cloud Strategies for Enterprise Organizations. *ResearchGate*. 2026. 7 p. URL: `https://www.researchgate.net/publication/400156628_Hybrid_and_Multi-Cloud_Strategies_for_Enterprise_Organizations` (дата звернення: 03.03.2026).

7. Artač M., Borovšak T., Di Nitto E., Guerriero M., Tamburri D. A. DevOps: Introducing Infrastructure-as-Code. *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. Buenos Aires, 2017. P. 497–498. DOI: 10.1109/ICSE-C.2017.162.

8. Aviv L., Gafni R., Sherman S., Aviv B., Sterkin A., Bega E. Infrastructure from Code: The Next Generation of Cloud Lifecycle Automation. *IEEE Software*. 2023.

Vol. 40, no. 1. P. 42–49. DOI: 10.1109/MS.2022.3209958

9. Baldini I. et al. Serverless Computing: Current Trends and Open Problems. Research Advances in Cloud Computing / ed. by S. Chaudhary et al. Singapore : Springer Nature Singapore Pte Ltd, 2017. P. 1–20. DOI: 10.1007/978-981-10-5026-8_1

10. Baset S., Suneja S., Bila N., Tuncer O., Isci C. Usable Declarative Configuration Specification and Validation for Applications, Systems, and Cloud. *Middleware Industry '17: Proceedings of the 2017 International Middleware Conference Industrial Track (Las Vegas, NV, USA, December 11–15, 2017)*. New York: ACM, 2017. P. 29–35. DOI: 10.1145/3154448.3154453

11. Van der Bent E., Hage J., Visser J., Gousios G. How Good Is Your Puppet? An Empirically Defined and Validated Quality Model for Puppet. *2018 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Campobasso, Italy : IEEE, 2018. P. 164–174. DOI: 10.1109/SANER.2018.8330202.

12. Beylerian M., Maurel Y. Concerto-SCP: A Framework for Consistent and Parallel Deployment of Cloud Applications. *Proceedings of the 2022 IEEE International Conference on Cloud Engineering (IC2E)*. 2022. P. 191–197. DOI: 10.1109/IC2E55432.2022.00032

13. Bhattacharjee A., Barve Y., Gokhale A., Kuroda T. CloudCAMP: Automating the Deployment and Management of Cloud Services. *2018 IEEE International Conference on Services Computing (SCC)*. 2018. P. 237–240. DOI: 10.1109/SCC.2018.00038

14. Borovits N., Kumara I., Di Nucci D., Krishnan P., Dalla Palma S., Palomba F., Tamburri D. A., van den Heuvel W. J. FindICI: Using machine learning to detect linguistic inconsistencies between code and natural language descriptions in infrastructure-as-code. *Empirical Software Engineering*. 2022. Vol. 27, no. 6. Art. 178. DOI: 10.1007/s10664-022-10215-5

15. Brikman Y. How to manage multiple environments with Terraform. *Gruntwork Blog*. 2022. URL: `https://medium.com/gruntwork/how-to-manag`

e-multiple-environments-with-terraform-32c7bc5d692 (дата звернення: 05.03.2026).

16. de Carvalho L. R., de Araujo A. P. F. Performance Comparison of Terraform and Cloudify as Multicloud Orchestrators. *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. Melbourne, VIC, Australia, 2020. P. 380–389. DOI: 10.1109/CCGrid49817.2020.00-55.

17. Chauhan Muhammad Aufeef, Babar Muhammad Ali, Benatallah Boualem. Architecting cloud-enabled systems: a systematic survey of challenges and solutions. *Software: Practice and Experience*. 2016. Vol. 46, No. 11. P. 1–41. DOI: 10.1002/spe.2409.

18. Dalla Palma S., Di Nucci D., Palomba F., Tamburri D. A. Within-project Defect Prediction of Infrastructure-as-Code Using Product and Process Metrics. *IEEE Transactions on Software Engineering*. 2020. Vol. 14, no. 8. DOI: 10.1109/TSE.2021.3051492.

19. Dalla Palma S., Di Nucci D., Palomba F., Tamburri D. A. Toward a catalog of software quality metrics for infrastructure code. *The Journal of Systems & Software*. 2020. Vol. 170. Art. 110726. DOI: 10.1016/j.jss.2020.110726

20. Dinu F., Fontaine J.-M. Infrastructure Drift Detection — How to Fix It With IaC Tool. *Spacelift Blog*. 2024. URL: https://spacelift.io/blog/drift-detection (дата звернення: 03.03.2026).

21. Feitosa D., Penca M.-T., Berardi M., Boza R.-D., Andrikopoulos V. Mining for Cost Awareness in the Infrastructure as Code Artifacts of Cloud-based Applications: an Exploratory Study. arXiv:2304.07531 [cs.SE]. 2023. 13 p. DOI: 10.48550/arXiv.2304.07531

22. What is CI/CD?. *GitLab*. 2025. URL: https://about.gitlab.com/topics/ci-cd/ (дата звернення: 03.03.2026).

23. Container scanning overview. *Google Cloud Documentation*. 2025. URL: https://cloud.google.com/artifact-analysis/docs/container-scannin

g-overview (дата звернення: 03.03.2026).

24. Gudelli Venkata Ramana. Cloud Formation and Terraform: Advancing Multi-Cloud Automation Strategies. *International Journal of Innovative Research in Management, Pharmacy and Sciences*. 2023. Vol. 11, issue 2. P. 1–10. DOI: 10.37082/IJIRMPS.v11.i2.232164

25. Guerriero M., Garriga M., Tamburri D. A., Palomba F. Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Cleveland, OH, USA, 2019. P. 580–589. DOI: 10.1109/ICSME.2019.00092

26. Hanappi O., Hummer W., Dustdar S. Asserting Reliable Convergence for Configuration Management Scripts. *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. 2016. P. 328–343. DOI: 10.1145/2983990.2984000

27. Hasan M. M., Bhuiyan F. A., Rahman A. Testing practices for infrastructure as code. *LANGETI 2020: Proceedings of the 1st ACM SIGSOFT International Workshop on Languages and Tools for Next-Generation Testing*. 2020. P. 7–12. DOI: 10.1145/3416504.3424334

28. Maturing your Terraform workflow. *HashiCorp Blog*. 2023. URL: https://www.hashicorp.com/en/blog/maturing-your-terraform-workflow (дата звернення: 03.03.2026).

29. Terraform workflow best practices at scale. *HashiCorp Blog*. 2019. URL: https://www.hashicorp.com/en/resources/terraform-workflow-best-practices-at-scale (дата звернення: 06.03.2026).

30. Haverinen H., Janhunen T., Päivärinta T., Kaartinen S., Lempinen S., Merilä S. Automating Cybersecurity Compliance in DevSecOps with Open Information Model for Security as Code. *eSAAM 24: Proceedings of the 4th Eclipse Security, AI, Architecture and Modelling Conference on Data Space (October 2024)*. New York : ACM, 2024. P. 93–102. DOI: 10.1145/3685651.3686700

31. Hernández R., Moros B., Nicolás J. Requirements management in DevOps environments: a multivocal mapping study. *Requirements Engineering*. 2023. Vol. 28. P. 317–346. DOI: 10.1007/s00766-023-00396-w

32. Holkuziev Diyorjon. Infrastructure Automation in Cloud Environments Using Terraform. *Universal Library of Engineering Technology*. 2025. Vol. 2, issue 3. P. 06–11. DOI: 10.70315/uloap.ulete.2025.0203002.

33. Hughes L., Webb J., Morgan H., Song M. GitOps for Continuous Deployment in Cloud-Native Infrastructure. 2024. 5 p. URL: https://www.researchgate.net/publication/392163663_GitOps_for_Continuous_Deployment_in_Cloud-Native_Infrastructure (дата звернення: 03.03.2026).

34. Hummer W., Rosenberg F., Oliveira F., Eilam T. Automated Testing of Chef Automation Scripts. Proceedings of the 14th ACM/IFIP/USENIX International Middleware Conference (Middleware DPT '16). Beijing, 2016. P. 4:1–4:2. DOI: 10.1145/2541614.2541632.

35. Imtiaz N., Thorn S., Williams L. A comparative study of vulnerability reporting by software composition analysis tools. *ESEM '21: ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Bari, Italy, October 11–15, 2021). New York : ACM, 2021. 11 p. DOI: 10.1145/3475716.3475769.

36. Jiang Y., Adams B. Co-evolution of Infrastructure and Source Code – An Empirical Study. *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. Florence, 2015. P. 45–55. DOI: 10.1109/MSR.2015.12.

37. Karanam R. Multi-cloud IAC template versioning and rollback strategies: An empirical study with terraform and GITOPS. *World Journal of Advanced Research and Reviews*. 2024. Vol. 22, no. 02. P. 2354–2363. DOI: 10.30574/wjarr.2024.22.2.1357

38. Köstem B. From Code to Kubernetes: Building a Full GitOps Pipeline with GitLab CI and FluxCD. *Medium*. Aug. 23, 2024. URL: https://medium.com/@fenari.kostem/from-code-to-kubernetes-building-a-full-gitops-pipeline-with-gitlab-ci-and-fluxed-aa6188ca517f (дата звернення: 03.03.2026).

39. Kovacs J., Kacsuk P. Occopus: a Multi-Cloud Orchestrator to Deploy and Manage Complex Scientific Infrastructures. *Journal of Grid Computing*. 2018. Vol. 16, iss. 1. P. 19–37. DOI: 10.1007/s10723-017-9421-3

40. Krishnan P. Terraform - Automating Infrastructure as a Service. *International Journal of Science and Research (IJSR)*. 2024. Vol. 13, iss. 10. P. 199–203. DOI: 10.21275/SR24930224444

41. Kumara I., Garriga M., Romeu A. U., Di Nucci D., Palomba F., Tamburri D. A., van den Heuvel W. J. The do's and don't's of infrastructure code: A systematic gray literature review. *Information and Software Technology*. 2021. Vol. 137. Art. 106593. DOI: 10.1016/j.infsof.2021.106593

42. Kyadasu R., Dave A., Arulkumaran R., Goel O., Kumar D. L., Jain P. A. Exploring Infrastructure as Code Using Terraform in Multi-Cloud Deployments. *Journal of Quantum Science and Technology*. 2024. Vol. 1, no. 4. P. 1–24. DOI: 10.63345/jqst.v1i4.94

43. Leite L., Rocha C., Kon F., Milojicic D., Meirelles P. A Survey of DevOps Concepts from a Software Architecture Perspective. *Journal of Systems and Software*. 2019. Vol. 157. P. 110351. DOI: 10.1016/j.jss.2019.110351

44. Lima C. G. S., Santana C. S., Santos J. L. C., Junior J. B. R. Machine Learning-based Drift Detection in Infrastructure as Code. *2023 IEEE International Conference on Big Data (Big Data)*. 2023. P. 2483–2492. DOI: 10.1109/BigData59044.2023.10386685

45. Lipton P., Palma D., Rutkowski M., Tamburri D. A. TOSCA Solves Big Problems in the Cloud and Beyond. *IEEE Cloud Computing*. 2018. Vol. 5, no. 2. P. 37–47. DOI: 10.1109/MCC.2018.022171666.

46. Opdebeeck R., Zerouali A., De Roover C. Behaviour-aware Security Smell Detection for Infrastructure as Code. *BENEVOL 2023: 22nd Belgium-Netherlands Software Evolution Workshop*. 2023. DOI: 10.1145/3408897

47. OpenTofu Documentation / OpenTofu. 2026. URL: `https://opentofu`

.org/docs/ (дата звернення: 03.03.2026).

48. Ozkaya I. Infrastructure as Code and Software Architecture Conformance Checking. *IEEE Software*. 2023. Vol. 40, no. 1. P. 4–8. DOI: 10.1109/MS.2022.3213880

49. Pahl C., Jamshidi P., Weyns D. Cloud architecture continuity: Change models and change rules for sustainable cloud software architectures. *Journal of Software: Evolution and Process*. 2017. Vol. 29, no. 2. e1849. DOI: 10.1002/smr.1849

50. Pahl C., Jamshidi P., Zimmermann O. Microservices and Containers – Architectural Patterns for Cloud and Edge. *Software Engineering 2020*. Bonn: Gesellschaft für Informatik, 2020. P. 11–12. (Lecture Notes in Informatics (LNI); vol. P-300). URL: https://dl.gi.de/handle/20.500.12116/29906

51. Pahl C., Gunduz N. G., Sezen O. C., Ghamgosar A., El Ioini N. Infrastructure as Code - Technology Review and Research Challenges. *Proceedings of the 20th International Conference on Cloud Computing and Services Science (CLOSER 2025)*. 2025. P. 1–8. DOI: 10.5220/0012625200003851.

52. Pahl C., Gunduz N. G., Sezen Ö. C., Ghamgosar A., Hofer F., El Ioini N. A Systematic Review of Infrastructure-as-Code Technologies. *Proceedings of the 15th International Conference on Cloud Computing and Services Science (CLOSER 2025)*. 2025. Vol. 1. P. 151–158.

53. Palomba F., Bavota G., Di Penta M., Fasano F., Oliveto R., De Lucia A. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*. 2018. Vol. 23, no. 3. P. 1188–1250. DOI: 10.1007/s10664-017-9535-z.

54. Parnin C., Helms E., Atlee C., Boughton H., Ghattas M., Glover A., Holman J., Micco J., Murphy B., Savor T., Stumm M., Whitaker S., Williams L. The Top 10 Adages in Continuous Deployment. *IEEE Software*. 2017. Vol. 34, no. 3. P. 86–95. DOI: 10.1109/MS.2017.33

55. Rahman A., Partho A., Morrison P., Williams L. What Questions Do Programmers Ask About Configuration as Code?. *RCOSE '18: Proceedings of the 4th*

International Workshop on Rapid Continuous Software Engineering. Gothenburg, 2018. P. 16–22. DOI: 10.1145/3194760.3194769.

56. Rahman A., Mahdavi-Hezaveh R., Williams L. A Systematic Mapping Study of Infrastructure as Code Research. *Information and Software Technology*. 2019. Vol. 108. P. 65–77. DOI: 10.1016/j.infsof.2018.12.004.

57. Rahman A., Parnin C., Williams L. The Seven Sins: Security Smells in Infrastructure as Code Scripts. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE) (Montreal, QC, Canada, May 25–31, 2019)*. New York : IEEE, 2019. P. 164–175. DOI: 10.1109/ICSE.2019.00033.

58. Rahman A., Farhana E., Parnin C., Williams L. Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts. *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. 2020. P. 752–764. DOI: 10.1145/3377811.3380409

59. Rangnau T., Buijtenen R. v., Fransen F., Turkmen F. Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines. *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*. 2020. P. 145–154. DOI: 10.1109/EDOC49727.2020.00026.

60. Romm N. Efficiency of Terraform and Kubernetes Integration in DevOps Practices. *The American Journal of Engineering and Technology*. 2025. Vol. 07, no. 07. P. 88–95. DOI: 10.37547/tajet/Volume07Issue07-10.

61. Sandobalin J., Insfran E., Abrahão S. On the Effectiveness of Tools to Support Infrastructure as Code: Model-Driven versus Code-Centric. *IEEE Access*. 2020. Vol. 8. P. 24961–24979. DOI: 10.1109/ACCESS.2020.2966597.

62. Terraform Module Registry — Hierarchical Inheritance. *Scalr Blog*. 2021. URL: `https://scalr.com/blog/hierarchical-terraform-module-registry` (дата звернення: 03.03.2026).

63. Sharma A., Nayak C., Khandelwal S., Raj U., Almas M. Automated Terraform Generation using NLP and Graph-Based Cloud Architecture Visualizati-

on. *International Journal for Research in Applied Science & Engineering Technology (IJRASET)*. 2025. Vol. 13, issue V. P. 5855–5861. DOI: 10.22214/ijraset.2025.71537

64. Sharma T., Fragkoulis M., Spinellis D. Does Your Configuration Code Smell? *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. Austin, 2016. P. 189–200. DOI: 10.1145/2901739.2901761.

65. Sharma A., Rodriguez E., Tanaka K., Johnson M. Scaling Infrastructure Governance: A Policy-Driven Framework Combining Terraform Automation and Red Hat Satellite. *ResearchGate*. 2026. URL: https://www.researchgate.net/publication/399734521_Scaling_Infrastructure_Governance_A_Policy-Driven_Framework_Combining_Terraform_Automation_and_Red_Hat_Satellite (дата звернення: 02.03.2026).

66. Silva R., Brito A., Lima Filho J. P. Automation of Security Controls for Continuous Compliance in Vulnerability Management. *LADC 2024: Proceedings of the 13th Latin-American Symposium on Dependable and Secure Computing (Recife, Brazil, November 26–29, 2024)*. New York : ACM, 2024. 10 p. DOI: 10.1145/3697090.3697107.

67. Shifting Terraform Configuration Security Left. *HashiCorp Blog*. 2020. URL: https://www.hashicorp.com/en/resources/shifting-terraform-configuration-security-left (дата звернення: 06.03.2026).

68. Sokolowski D., Weisenburger P., Salvaneschi G. Decentralizing Infrastructure as Code. *IEEE Software*. 2023. Vol. 40, no. 1. P. 50–55. DOI: 10.1109/MS.2022.3192968.

69. Steglich C., Marczak S., de Souza C. R. B., Filho F. F., Guerra L. P., Mosmann L. H., Perin M. Social Aspects and How They Influence MSECO Developers. *2019 IEEE/ACM 12th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. 2019. P. 99–106. DOI: 10.1109/CHASE.2019.00032

70. Stirbu V., Raatikainen M., Röntynen J., Sokolov V., Lehtonen T., Mikkonen T. Toward Multiconcern Software Development with Everything as Code. *IEEE*

Software. 2022. Vol. 39, no. 4. P. 27–33. DOI: 10.1109/MS.2022.3167481.

71. Teppan H., Flå L. H., Jaatun M. G. A Survey on Infrastructure-as-Code Solutions for Cloud Development. *2022 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. Bangkok, Thailand, 2022. P. 60–65. DOI: 10.1109/CloudCom55334.2022.00019.

72. Terraform Documentation / HashiCorp. 2026. URL: `https://developer.hashicorp.com/terraform/docs` (дата звернення: 03.03.2026).

73. The Four Stages of Terraform Automation. *env0 Blog*. 2025. URL: `https://www.env0.com/blog/expert-guide-the-four-stages-of-terraform-automation` (дата звернення: 06.03.2026).

74. Thompson M. J., Rivera A. L., Morgan D. K., Allen R. S., William E. Regulatory Compliance Automation in Scalable Cloud Infrastructure. *ResearchGate*. 2024. 9 p. URL: `https://www.researchgate.net/publication/394473640_Regulatory_Compliance_Automation_in_Scalable_Cloud_Infrastructure` (дата звернення: 03.03.2026).

75. Why GitOps is the Future of Infrastructure Management. *The New Stack*. 2025. URL: `https://thenewstack.io/why-gitops-is-the-future-of-infrastructure-management/` (дата звернення: 03.03.2026).

76. Tomarchio O., Calcaterra D., Di Modica G. Cloud resource orchestration in the multi-cloud landscape: a systematic review of existing frameworks. *Journal of Cloud Computing*. 2020. Vol. 9, Iss. 1. Art. 49. 24 p. DOI: 10.1186/s13677-020-00194-7

77. Vasileiou Z. et al. A knowledge-based approach for guided development of Infrastructure as Code. *Software and Systems Modeling*. 2025. DOI: 10.1007/s10270-025-01294-1

78. Weerasiri D. et al. A Taxonomy and Survey of Cloud Resource Orchestration Techniques. *ACM Computing Surveys*. 2017. Vol. 50, No. 2. Article 26. 41 p. DOI: 10.1145/3054177

79. Wei H., Madhavji N., Steinbacher J. Understanding Everything as Code: A

Taxonomy and Conceptual Model. 2025. 12 p. (Препринт. arXiv ; 2507.05100). DOI: 10.48550/arXiv.2507.05100.

80. Wurster M., Breitenbücher U., Falkenthal M., Krieger C., Leymann F., Saatkamp K., Soldani J. The essential deployment metamodel: a systematic review of deployment automation technologies. *SICS Software-Intensive Cyber-Physical Systems*. 2020. Vol. 35. P. 63–75. DOI: 10.1007/s00450-019-00412-x

81. Kolomytskyi D., Rehida P., Onyshko O., Drozd A. Generalized Method for Managing the Lifecycle of Terraform Infrastructure Across Multiple Environments. *Herald of Khmelnytskyi National University. Technical Sciences*. 2026. Vol. 363, no. 2. P. 117–125.

ДОДАТОК А

(обов'язковий)

Стаття

УДК 004.75

Узагальнений метод керування життєвим циклом Terraform-інфраструктури для кількох середовищ

Коломицький Д.Є., студент гр. КІ2м-24-2

Співавтори: Регіда П.Г., Онишко О.Г., Дрозд А.І.

Науковий керівник: Савенко Олег Станіславович, д.т.н., проф.

Хмельницький національний університет

UDC 004.75

DENYS KOLOMYTSKYI, PAVLO REHIDA, OKSANA ONYSHKO,
ANDRIY DROZD
Khmelnytskyi National University, Khmelnytskyi, Ukraine

GENERALIZED METHOD FOR MANAGING THE LIFECYCLE OF TERRAFORM INFRASTRUCTURE ACROSS MULTIPLE ENVIRONMENTS

The article examines the challenge of managing the lifecycle of cloud infrastructure, as described using Terraform, across multiple environments (development, staging, and production). In industrial settings, multi-environment infrastructure management gives rise to a set of interrelated challenges: the absence of formalized procedures for promoting changes between environments with explicit source readiness verification, fragmented compliance checking that fails to distinguish code-level syntax validation from plan-level semantic verification, and configuration drift detection that lacks classification by severity, generating false alerts for expected changes such as dynamic IP addresses and rotated certificates. An analysis of existing approaches to environmental isolation, configuration compliance verification, and drift detection reveals that none of the current methods address the full lifecycle in a unified manner. A generalized method is proposed, based on a formalized two-projection lifecycle model: the horizontal projection describes an eight-stage finite automaton of an individual environment (Init, Author, Validate, Plan, Comply, Approve, Apply, Monitor), while the vertical projection defines a partially ordered environment space with a formalized promotion operation. The method introduces a source readiness precondition requiring the source environment to be in the monitoring stage with empty planning and drift deltas, versioned configuration snapshots ensuring code identity across environments, multi-level compliance verification across code, plan, and state levels using hierarchical policy inheritance, and a three-class drift classification (critical, actionable, informational) with an effective delta mechanism that filters expected changes. Experimental verification on a real multi-environment AWS infrastructure (27 managed resources, 3 environments) using the Scalr platform confirms that the proposed method ensures automated detection of 100% of policy violations before the apply stage (compared to 50% for GitOps CI/CD and 0% for Terraform CLI), reduces false drift alerts to zero, and decreases the number of manual promotion steps to a single approval for the production environment.

Keywords: Infrastructure as Code, Terraform, lifecycle management, multi-environment infrastructure, configuration drift, compliance verification, DevOps, cloud computing.

Д.Є. КОЛОМИЦЬКИЙ, П.Г. РЕГІДА, О.Г. ОНИШКО, А.І. ДРОЗД
Хмельницький національний університет, Хмельницький, Україна

УЗАГАЛЬНЕНИЙ МЕТОД КЕРУВАННЯ ЖИТТЄВИМ ЦИКЛОМ TERRAFORM-ІНФРАСТРУКТУРИ ДЛЯ КІЛЬКОХ СЕРЕДОВИЩ

У статті досліджується проблема управління життєвим циклом хмарної інфраструктури, описаної засобами Terraform, для кількох середовищ (розробки, тестування, виробництва). Здійснено аналіз існуючих підходів до ізоляції середовищ, перевірки відповідності конфігурацій та виявлення конфігураційного дрейфу. Запропоновано узагальнений метод, що ґрунтується на формалізованій двопроекційній моделі життєвого циклу у вигляді восьмистадійного скінченного автомата та частково впорядкованого простору середовищ. Метод включає формалізовану процедуру просування змін між середовищами з передумовою готовності джерела, багаторівневу перевірку відповідності на рівнях коду, плану та стану, а також трикласову класифікацію конфігураційного дрейфу з механізмом версіонованого відкату. Експериментальна верифікація на реальній мультисередовищній інфраструктурі AWS із використанням платформи Scalr підтверджує, що запропонований метод забезпечує виявлення 100% порушень політик до стадії застосування та зведення хибних сповіщень дрейфу до нуля через ефективну дельту.

Ключові слова: Infrastructure as Code, Terraform, управління життєвим циклом, мультисередовищна інфраструктура, конфігураційний дрейф, перевірка відповідності, DevOps, хмарні обчислення.

Overview of the Problem and Its Connection to Major Scientific and Practical Challenges

Modern approaches to software development have experienced a significant shift under the influence of the “Everything as Code” paradigm—a systematic method in which any artifact of a software system, including infrastructure, configuration, and security policies, is described as versioned code [1]. At the core of this paradigm is the practice of Infrastructure as Code (IaC), which involves declaratively describing cloud environment resources in configuration files

stored in a version control system and executed by automated tools. Terraform is the leading tool in this category because of its provider-agnostic design and explicit state management through the terraform.tfstate file, which maps the described resources to the actual entities in the cloud [2].

A typical industrial infrastructure spans multiple environments—development, testing, and production—each consisting of isolated cloud resources at different stages of a software product's lifecycle. Managing this multi-environment setup introduces a complex set of interconnected issues that cannot be addressed by individual tools alone.

Current methods for environment isolation—based on workspaces, directory structures, or repository branches—balance different trade-offs between isolation levels and maintenance complexity [3]. However, none of these methods offers a formal process for promoting changes across environments with clear prerequisites for source readiness. In practice, this can allow a change to reach the production environment from a source that may have undetected configuration drift or an incomplete application, raising the risk of failed deployments. An empirical study of versioning and rollback strategies for IaC templates confirmed that formalizing these procedures can significantly decrease the average recovery time after failures by an order of magnitude [4].

The verification of infrastructure configuration compliance remains scattered. Typical CI/CD pipelines include static security analysis during code validation but do not separate the checking of configuration syntax from verifying the semantics of planned changes as distinct stages. A configuration that is syntactically correct and has no known vulnerabilities can still break organizational policies only at deployment—such as using a forbidden instance type or planning to delete a protected resource. Additionally, studies have shown that adding declarative security policies directly into the resource provisioning pipeline can achieve a 92% improvement in compliance metrics [5].

Configuration drift—the gap between the actual state of cloud resources and what is defined in the code—is another key issue caused by manual changes in the cloud console, partial applications, or external processes outside of Terraform. Existing tools either lack a built-in way to monitor drift or identify discrepancies without prioritizing their severity, leading to a flood of false alerts about expected changes—such as dynamic IP addresses or automatically rotated certificates—which diminishes the response to truly critical issues.

Systematic reviews of IaC technologies show that, despite a considerable amount of research—from tool taxonomies to defect cataloging and formal analyses of idempotency—the comprehensive management of multi-environment infrastructure throughout its entire lifecycle remains neglected in academic studies [6]. There is no universal approach that covers the complete process from configuration setup to automatic drift correction, formalizes change propagation, and guarantees multi-level compliance verification at different stages of the pipeline. Creating such a method is an urgent scientific and practical challenge directly related to enhancing the reliability and security of cloud infrastructures in industrial settings.

Analysis of Recent Research and Publications

Pahl and his co-authors conducted the most thorough systematization of IaC technologies, proposing a classification based on four dimensions—application context, functionality, description language, and execution architecture—and developing a specialized DevOps lifecycle for infrastructure code based on it [7]. This cycle includes stages from code writing and validation to environment self-recovery; however, it is described at a conceptual level without formalizing transitions between stages and without considering coordination across multiple environments.

Based on a systematic review of the gray literature, Kumara and his co-authors identified the fundamental properties of high-quality infrastructure code: declarativeness, idempotence, and reproducibility [8]. Hanappi provided formal confirmation of these properties, demonstrating through state transition graphs that violations of idempotence are a hidden source of errors that only become apparent upon repeated application of configurations [9]. These findings serve as the theoretical foundation for formalizing the lifecycle but do not translate into a specific method for managing environments.

Empirical studies have uncovered systemic gaps in the IaC ecosystem, including fragmentation of tools, a lack of mature testing methods, and a shortage of standardized state management practices [10]. A qualitative analysis of testing practices identified six categories of checks—ranging from static analysis to policy compliance testing—yet none of the existing pipelines implement them as architecturally independent stages with distinct input artifacts [11].

Based on an analysis of 1,448 commits, Rahman and his co-authors developed a taxonomy of eight categories of infrastructure code defects, with configuration data errors being the most common and violations of idempotency being the most specific to IaC [12]. Another area of focus is the detection of configuration drift: Dinu and Fontaine proposed a method for continuously comparing Terraform plan results with an expected empty plan and automatically notifying of discrepancies, although without classifying the detected drift by severity level [13].

In the realm of state management, the fork of the Terraform ecosystem following HashiCorp's license model change is important: the open-source fork OpenTofu, supported by the Linux Foundation, maintains compatibility at the HCL language and provider levels [14], which requires provider independence in any general method.

Research on GitOps pipelines for Terraform shows that deployment frequency increases and recovery time decreases compared to traditional methods; however, the pull synchronization model is limited to planning and application stages without formalizing progress between environments [15].

Thus, existing studies focus on specific aspects—tool classification, formalization of idempotency, testing, drift detection, and GitOps automation—but do not provide a comprehensive method that covers the entire lifecycle of a multi-environment Terraform infrastructure, including formalized change propagation, multi-level compliance verification, and classification of configuration drift.

Statement of the article's objectives

The purpose of this article is to create a comprehensive approach for managing the lifecycle of Terraform infrastructure across multiple environments. To achieve this, three tasks are outlined: (1) formalize a two-projection lifecycle model as a finite state machine and a partially ordered environment space, covering stages from configuration setup to detecting and fixing configuration drift; (2) justify a method for propagating changes between environments with source readiness requirements, multi-level compliance checks, and a versioned rollback system; (3) test the approach on a real multi-environment infrastructure and evaluate its effectiveness using specific metrics.

Conceptual Model and Formalization of the Method

Formalization of Concepts and the Lifecycle Model

Let M be a finite set of Terraform modules, where each module $m = \langle R_m, V_m, O_m \rangle$ is defined by a set of resource descriptions R_m, input variables V_m and output values O_m. An infrastructure configuration is defined as a pair:

$$C = \langle m_{root}, \Theta \rangle, \quad (1)$$

where $m_{root} \in M$ — is the root module (entry point), and, $\Theta = \{\theta_1, \dots, \theta_k\}$ — is the set of input variable values that parameterizes a specific instance of the infrastructure. It is Θ that determines the differences between environments when the root module is shared.

The environment is defined by four factors:

$$e = \langle id_e, C_e, S_e, P_e \rangle, \quad (2)$$

where id_e — is a unique identifier; C_e — is the configuration; S_e — is the current state; P_e — is the set of compliance policies. The inclusion of P_e in the definition reflects the principle of built-in compliance: policies are an integral part of the environment.

The concept of state involves distinguishing three entities. The desired state $S_e^d = eval(C_e)$ is set by interpreting the configuration. The recorded state S_e^r — is the content of terraform.tfstate, which updates after each successful operation. The actual state S_e^a — includes resources that presently exist in the cloud and may differ from S_e^r due to changes outside of Terraform. This three-part model allows us to formally define the *planning delta* $\Delta_e^{plan} = S_e^d \setminus S_e^r$ and the drift delta as a symmetric difference:

$$\Delta_e^{drift} = S_e^r \triangle S_e^a, \quad (3)$$

which covers both resources that are modified or deleted outside of Terraform and resources added to the cloud without a matching description in the configuration. The environment is *consistent* if both deltas are empty.

A set of environments forms the environment space $E = \{e_1, \dots, e_n\}$ with a partial order relation $\preceq$. For a typical configuration such as $e_{dev} \preceq e_{staging} \preceq e_{prod}$ a change reaches the production environment only after passing through all intermediate environments.

The lifecycle of a single environment is modeled as a finite state machine $A_e = \langle \Sigma, \delta, \sigma_1, F \rangle$ with eight stages: σ_1 (Init) — initializing the backend and providers; σ_2 (Author) — making configuration changes; σ_3 (Validate) — performing syntax checks and static security analysis; σ_4 (Plan) — calculating Δ_e^{plan} without applying it; σ_5 (Comply) — verifying the plan's semantics for policy compliance; σ_6 (Approve) — obtaining approval (manual for production, automatic for development); σ_7 (Apply) — implementing the plan, which updates $S_e^r \leftarrow S_e^a$; σ_8 (Monitor) — periodically calculating Δ_e^{drift}. A failure at stages σ_3, σ_5 or σ_6 causes the automaton to return to σ_2 to correct the configuration, while detecting drift at σ_8 triggers a transition back to σ_4 , completing the feedback loop. The automaton's structure and feedback transitions are illustrated in Fig. 1.

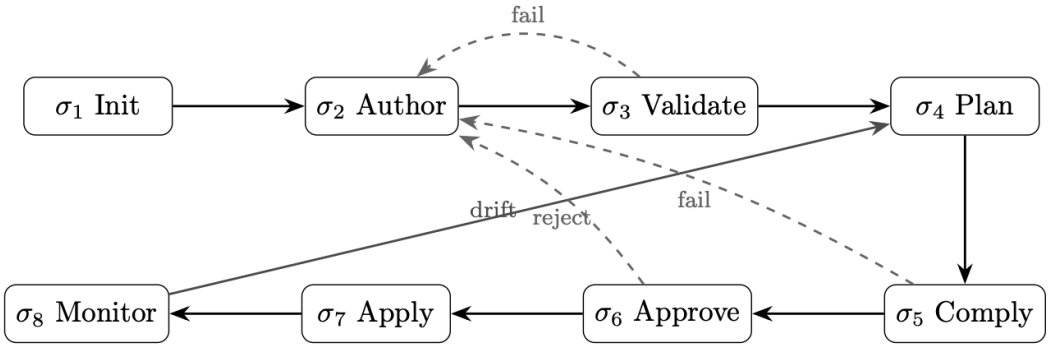


Fig. 1. – Finite-state machine A_e depicts the life cycle of a single environment. Solid arrows show the primary transition sequence; dashed arrows indicate backtransitions in case of failure; the transition $\sigma_8 \rightarrow \sigma_4$ – illustrates the drift elimination cycle.

A key difference from existing conveyors is the architectural separation of σ_3 and σ_5: the former verifies the predicate $valid(C_e, P_e^{code})$ without consulting the provider, while the latter verifies$comply\left(\Delta_e^{plan}, P_e^{plan}\right)$ on the plan artifact with computed resource attributes.

Promotion and Rollback Mechanism

The promotion operation between adjacent environments $e_i \preceq e_j$ is defined as:

$$promote(e_i, e_j): C_{e_j} \leftarrow \left\langle m_{root}, \Theta_{e_j} \right\rangle, \tag{4}$$

where the root module remains shared, and Θ_{e_j} reflects the specifics of the target environment:

$$ready(e_i) \Leftrightarrow stage(e_i) = \sigma_8 \wedge \Delta_{e_i}^{plan} = \emptyset \wedge \Delta_{e_i}^{drift} = \emptyset, \tag{5}$$

in other words, the source environment is in the monitoring phase with empty deltas. None of the existing approaches automatically verify this condition.

Code identity between environments is ensured through *versioned configuration snapshots* $\widehat{C_e^v} = \langle m_{root}, \Theta_e, v \rangle$ where v — is the version number assigned after successfully completing a full cycle in the source environment. A snapshot is an immutable artifact that prevents implicit changes in the dependencies between testing in e_i and deployment in e_j.

After advancing to the target environment, stages $\sigma_3 \rightarrow \sigma_5 \rightarrow \sigma_6 \rightarrow \sigma_7$ are executed sequentially with policies P_{e_j}, ensuring independent compliance verification for each environment.

The rollback mechanism relies on the history of state snapshots $H_e = \{S_e^{r,1}, \ldots, S_e^{r,v}\}$. The $rollback(e, v')$ operation restores the saved state$S_e^{r,v'}$ followed by the cycle $\sigma_4 \rightarrow \sigma_7$ to bring the actual state into compliance. The safety of the rollback is determined by the predicate:

$$safe(e, v, v') \Leftrightarrow \forall r \in \left(S_e^{r,v} \setminus S_e^{r,v'}\right) : rev(r) = 1, \quad (6)$$

where $rev(r) = 1$ for reversible resources (computational instances, network rules) and $rev(r) = 0$ for irreversible ones (databases with accumulated transactions). If $safe = false$ the σ_6 approval stage is forcibly switched to manual mode.

Multi-level compliance checking and drift classification

The set of environment policies is divided according to compliance levels:

$$P_e = P_e^{code} \cup P_e^{plan} \cup P_e^{state}, \quad (7)$$

where P_e^{code} applies to σ_3 (syntax, hardcoded secrets, open ports), P_e^{plan} – to σ_5 (prohibited instance types, deletion of protected resources, cost estimation), and P_e^{state} — to σ_8 (compliance of the actual state with security requirements). Policies are organized into an inheritance hierarchy: $P_e = P_{global} \cup P_{env(e)} \cup P_{comp(e)}$, with strictness increasing along $\preceq$ — the same policy may be advisory in e_{dev} and hard-mandatory in e_{prod}.

Detected drift is categorized into three types:

$$class\left(r, \Delta_e^{drift}, P_e^{state}\right) \in \{\text{critical, actionable, informational}\}, \quad (8)$$

Critical drift (a violation of a strict mandatory policy) is automatically resolved by $\sigma_8 \rightarrow \sigma_4$. Actionable drift (a discrepancy that does not breach critical policies) requires manual confirmation. Informational drift (expected changes such as dynamic IP addresses or rotated certificates) is excluded from the effective delta through the set of ignored attributes I_e:

$$\Delta_e^{eff} = \Delta_e^{drift} \setminus \{r : attr(r) \in I_e\}, \quad (9)$$

which reduces false alerts without compromising oversight of critical changes.

Implementation and Experimental Evaluation of the Method

To validate the method, we selected platform [16]—an industrial-grade Terraform/OpenTofu infrastructure orchestration system that features a directory-based isolation pattern, hierarchical configuration inheritance, and built-in compliance checking via Open Policy Agent (OPA). The experimental infrastructure is deployed in Amazon Web Services (AWS) cloud and includes a typical web service: network layer (VPC, subnets, security groups), compute layer (EC2), storage layer (RDS PostgreSQL, S3), and access management (IAM). The configuration is organized into a library of five submodules, totaling approximately 1,200 lines of HCL and 27 managed resources. Environment space: $e_{dev} \preceq e_{staging} \preceq e_{prod}$.

The platform's Run pipeline directly maps to the A_e: state machine: two independent OPA policy checkpoints correspond to stages σ_3 (configuration check before planning) and σ_5 (check of the JSON plan artifact after planning). Monitoring of σ_8 is carried out through drift detection—periodic execution of `terraform plan` in refresh-only mode. Manual execution of the Terraform CLI and a typical GitOps CI/CD pipeline using GitLab CI, integrated with a static security analyzer, were chosen as comparison methods.

Experiment 1: Multi-level compliance verification.

Four violation scenarios were identified: two code-level violations—(C1) a hardcoded AWS secret and (C2) a security group with a 0.0.0.0/0 rule on port 22; and two plan-level violations—(C3) a prohibited m5.24xlarge instance type and (C4) deletion of a protected RDS instance. Each violation was added as a separate commit to the $e_{staging}$ configuration.

Table 1 – The stage of detecting violations by approach

Scenario	Level	Terraform CLI	GitOps CI/CD	Method (Scalr)
C1	P^{code}	Not automated	σ_3	σ_3(OPA)
C2	P^{code}	Not automated	σ_3	σ_3(OPA)
C3	P^{plan}	Not automated	Not automated	σ_5(OPA)
C4	P^{plan}	Not automated	Not automated	σ_5(OPA)

The proposed method achieves $R_{pre-apply} = 1{,}0$ – all four violations are detected automatically before the apply stage. GitOps CI/CD detects only code-level violations ($R_{pre-apply} = 0{,}5$), because the static analyzer does not process the plan artifact. The Terraform CLI has no built-in mechanisms for automated policy verification ($R_{pre-apply} = 0{,}0$): violations may be noticed by the operator during a manual review of the terraform plan output, but this depends on attentiveness and experience, and it is not a systematic check. The method's detection time is 15–40 seconds for code-level violations and 45–90 seconds for plan-level violations.

Experiment 2: drift detection and classification.

Three types of external changes were simulated via the AWS console: (D1) critical – adding the rule 0.0.0.0/0:443 to the prod-environment security group; (D2) operational – changing the EC2 instance tag; (D3) informational – changing the private IP address after a restart. The drift detection interval was set to $\tau = 5$ minutes to accelerate data collection (the typical value for production environments is 1–24 hours, depending on organizational requirements).

Table 2 – Drift Detection and Classification Results

Scenario	Class	Terraform CLI	GitOps CI/CD	Метод (Scalr)
D1	Critical	Not automated	Not automated	Auto-resolution, 6.2 min.
D2	Operational	Not automated	Not automated	Notification, awaiting confirmation
D3	Informational	Not automated	Not automated	Excluded from Δ_e^{eff}

The Terraform CLI and GitOps CI/CD lack built-in automated drift monitoring; detection requires manually running `terraform plan` or configuring a separate cron job. The proposed method accurately identified all three types of drift. Critical drift (D1) was automatically addressed in 6.2 minutes, including the detection cycle wait and the full $\sigma_4 \rightarrow \sigma_7$. Informational drift (D3) was filtered out from the effective delta via the I_e set, ensuring $F_{false} = 0$ – no false positives.

Experiment 3: Automation of the promotion process.

For the full promotion cycle $e_{dev} \preceq e_{staging} \preceq e_{prod}$ we recorded the number of manual steps N_{manual} (each CLI command or operator action requiring manual input counts as one step) and whether there was an automated check for the readiness of the source environment.

Table 3 – Comparison of promotion automation

Characteristics	Terraform CLI	GitOps CI/CD	Method (Scalr)
N_{manual}	9	3	1
Validation of $ready(e_i)$	None	None	Automatic
Policy validation during deployment	None	Partial (P^{code})	Full (P^{code}, P^{plan})

For Terraform CLI, $N_{manual} = 9$: the operator executes a plan + apply sequence for each of the three environments (6 commands) and reviews the plan output before each apply (3 reviews). For GitOps CI/CD $N_{manual} = 3$: manually launching the pipeline or merging for each environment without automatic readiness verification $ready(e_i)$. The proposed method requires $N_{manual} = 1$ – approval of the application only for the production environment; the remaining stages, including source readiness verification and cascading propagation, are automated.

Conclusions from This Research and Prospects for Further Study

This article introduces a generalized approach for managing the lifecycle of Terraform infrastructure across multiple environments. It relies on a formalized two-projection model: the horizontal projection outlines an eight-state finite automaton for a single environment, while the vertical projection represents a partially ordered space of environments with a formal transition operation.

A key theoretical contribution is the architectural separation of the verification of code compliance σ_3 and plan semantics σ_5 into independent stages with different input artifacts—something none of the approaches considered achieve. The formalization of the source readiness predicate $ready(e_i)$ or versioned configuration snapshots and the rollback safety predicate$safe(e, v, v')$ ensures atomicity of progression and controlled rollback, taking into account resource

reversibility. A three-class drift classification with an effective delta enables differentiated responses: automatic remediation of critical violations, confirmation for operational violations, and filtering of informational violations.

Experimental verification on a real AWS infrastructure (27 resources, 3 environments) confirmed that the method provides automated detection of 100% of policy violations prior to the deployment stage (compared to 50% in GitOps CI/CD and 0% in Terraform CLI), reduces false drift alerts to zero, and reduces manual deployment steps to a single approval for the production environment.

The method has certain limitations. First, it depends on a directory-based isolation pattern and HCL semantics, which guarantees compatibility with Terraform and OpenTofu but excludes tools with fundamentally different architectures (stateless or agent-based). Second, experimental validation was conducted on a single platform (Scalr) that natively implements the required mechanisms; applying it to other platforms necessitates adapting the OPA integration and the drift detection mechanism. Third, the method does not include automated cost estimation of infrastructure changes as a formalized stage.

Prospects for future research include incorporating cost estimation as part of the automaton, adapting the method to nonlinear topologies of environment spaces with parallel propagation chains, and testing on alternative orchestration platforms.

Declaration on the use of generative artificial intelligence tools

During the preparation of this work, the authors used Grammarly in order to: grammar and spelling check; DeepL Translate in order to: some phrases translation into English. After using these tools and services, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

Author Contributions

Conceptualization, Denys Kolomytskyi, Pavlo Rehida and Oksana Onyshko; methodology, Denys Kolomytskyi and Pavlo Rehida; software, Denys Kolomytskyi and Oksana Onyshko; validation, Pavlo Rehida and Andriy Drozd; formal analysis, Andriy Drozd; investigation, Pavlo Rehida and Andriy Drozd; resources, Oksana Onyshko; data curation, Denys Kolomytskyi and Oksana Onyshko; writing – original draft preparation, Denys Kolomytskyi and Oksana Onyshko; writing – review and editing, Pavlo Rehida and Andriy Drozd.

References

1. Wei H., Madhavji N., Steinbacher J. *Understanding Everything as Code: A Taxonomy and Conceptual Model*. 2025. 12 p. (Preprint. arXiv; 2507.05100). DOI: 10.48550/arXiv.2507.05100.

2. *Terraform Documentation* / HashiCorp. 2026. Available at: https://developer.hashicorp.com/terraform/docs (accessed: March 3, 2026).

3. Brikman Y. *How to manage multiple environments with Terraform*. Gruntwork Blog. 2022. Available at: https://medium.com/gruntwork/how-to-manage-multiple-environments-with-terraform-32c7bc5d692 (accessed: March 5, 2026).

4. Karanam R. *Multi-cloud IaC template versioning and rollback strategies: An empirical study with Terraform and GitOps*. World Journal of Advanced Research and Reviews. 2024. Vol. 22, no. 02. P. 2354–2363. DOI: 10.30574/wjarr.2024.22.2.1357.

5. Sharma A., Rodriguez E., Tanaka K., Johnson M. *Scaling Infrastructure Governance: A Policy-Driven Framework Combining Terraform Automation and Red Hat Satellite*. ResearchGate. 2026. Available at: https://www.researchgate.net/publication/399734521_Scaling_Infrastructure_Governance_A_Policy-Driven_Framework_Combining_Terraform_Automation_and_Red_Hat_Satellite (accessed: March 2, 2026).

6. Pahl C., Gunduz N. G., Sezen O. C., Ghamgosar A., El Ioini N. *Infrastructure as Code – Technology Review and Research Challenges*. Proceedings of the 20th International Conference on Cloud Computing and Services Science (CLOSER 2025). 2025. P. 1–8. DOI: 10.5220/0012625200003851.

7. Pahl C., Gunduz N. G., Sezen Ö. C., Ghamgosar A., Hofer F., El Ioini N. *A Systematic Review of Infrastructure-as-Code Technologies*. Proceedings of the 15th International Conference on Cloud Computing and Services Science (CLOSER 2025). 2025. Vol. 1. P. 151–158.

8. Kumara I., Garriga M., Romeu A. U., Di Nucci D., Palomba F., Tamburri D. A., van den Heuvel W. J. *The Do's and Don'ts of Infrastructure Code: A Systematic Gray Literature Review*. Information and Software Technology. 2021. Vol. 137. Art. 106593. DOI: 10.1016/j.infsof.2021.106593.

9. Hanappi O., Hummer W., Dustdar S. *Asserting Reliable Convergence for Configuration Management Scripts*. Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016). 2016. P. 328–343. DOI: 10.1145/2983990.2984000.

10. Guerriero M., Garriga M., Tamburri D. A., Palomba F. *Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry*. 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). Cleveland, OH, USA, 2019. P. 580–589. DOI: 10.1109/ICSME.2019.00092.

11. Hasan Mohammed Mehedi, Bhuiyan Farzana Ahamed, Rahman Akond. *Testing Practices for Infrastructure as Code*. LANGETI 2020: Proceedings of the 1st ACM SIGSOFT International Workshop on Languages and Tools for Next-Generation Testing. 2020. P. 7–12. DOI: 10.1145/3416504.3424334.

12. Rahman A., Farhana E., Parnin C., Williams L. *Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts*. Proceedings of the 42nd International Conference on Software Engineering (ICSE). 2020. P. 752–764. DOI: 10.1145/3377811.3380409.

13. Dinu F., Fontaine J.-M. *Infrastructure Drift Detection — How to Fix It with IaC Tools*. Spacelift Blog. 2024. Available at: https://spacelift.io/blog/drift-detection (accessed: March 3, 2026).

14. *OpenTofu Documentation* / OpenTofu. 2026. Available at: https://opentofu.org/docs/ (accessed: March 3, 2026).

15. Hughes L., Webb J., Morgan H., Song M. *GitOps for Continuous Deployment in Cloud-Native Infrastructure*. 2024. 5 p. Available at: https://www.researchgate.net/publication/392163663_GitOps_for_Continuous_Deployment_in_Cloud-Native_Infrastructure (accessed: March 3, 2026).

16. *Terraform Module Registry — Hierarchical Inheritance*. Scalr Blog. 2021. Available at: https://scalr.com/blog/hierarchical-terraform-module-registry (accessed: March 3, 2026).

Kolomytskyi Denys Коломицький Денис	Master student majoring in Information Systems and Technologies, Khmelnytskyi National University ORCID: 0009-0008-4335-1982 E-mail: kolomitskiy@gmail.com	Магістр спеціальності «Комп'ютерна інженерія», Хмельницький національний університет
Rehida Pavlo Регіда Павло	PhD in Computer Engineering, Associate Professor of the Department of Computer Engineering & Information Systems Department, Khmelnytskyi National University ORCID: 0000-0002-6591-7069 E-mail: pavlo.rehida@gmail.com	Доктор філософії комп'ютерної інженерії, доцент кафедри комп'ютерної інженерії та інформаційних систем, Хмельницький національний університет
Oksana Onyshko Оксана Онишко	Candidate of Pedagogical Sciences, Associate Professor of the Department of Software Engineering, Khmelnytskyi National University ORCID: 0000-0002-2125-4160 E-mail: Van4o@ukr.net	Кандидат педагогічних наук, доцент кафедри інженерії програмного забезпечення, Хмельницький національний університет
Andriy Drozd Андрій Дрозд	PhD Student majoring in Computer Engineering, Khmelnytskyi National University ORCID: 0009-0008-1049-1911 E-mail: andriydrozdit@gmail.com	Аспірант спеціальності «Комп'ютерна інженерія», Хмельницький національний університет

ДОДАТОК Б

(обов'язковий)

Презентація

Хмельницький національний університет
Факультет інформаційних технологій
Кафедра комп'ютерної інженерії та інформаційних систем

Кваліфікаційна робота магістра

«Узагальнений метод керування життєвим циклом Terraform-інфраструктури для кількох середовищ»

Виконав: здобувач групи KI2м-24-2
Коломицький Денис Євгенійович

Науковий керівник: д.т.н., професор
Савенко Олег Станіславович

Хмельницький, 2026

Актуальність

- Terraform — провідний інструмент Infrastructure as Code з декларативною мовою HCL та провайдеро-незалежною архітектурою.
- Управління кількома середовищами (dev, staging, prod) — поширена практика, проте методологічно не розв'язана.
- Існуючі підходи (Workspaces, Terragrunt, GitOps, HCP Terraform) покривають **лише окремі аспекти**.
- Немає єдиного підходу, що одночасно забезпечує:
  - повну ізоляцію середовищ;
  - формалізоване просування змін;
  - автоматизоване виявлення конфігураційного дрейфу.

Мета, об'єкт, предмет, задачі

Мета: розроблення узагальненого методу керування життєвим циклом Terraform-інфраструктури для кількох середовищ.

Об'єкт: процес керування життєвим циклом Terraform-інфраструктури для кількох середовищ.

Предмет: методи, моделі та алгоритми управління цим циклом.

Задачі:

1. Проаналізувати патерни ізоляції середовищ.
2. Розробити модель життєвого циклу.
3. Обґрунтувати метод управління станом та просування змін.
4. Реалізувати метод та оцінити ефективність.

Аналіз існуючих підходів

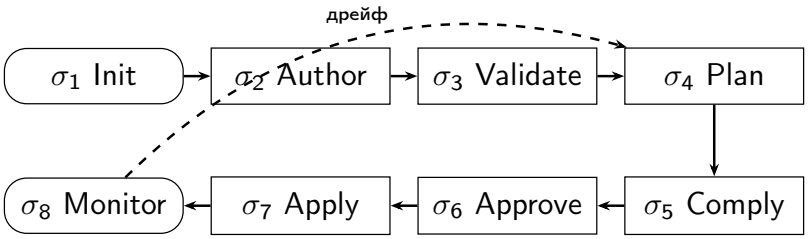
Критерій	**TF Workspaces**	**Terragrunt**	**GitOps**	**HCP TF**
Покриття стадій	4 з 8	4 з 8	5 з 8	7 з 8
Ізоляція стану	Часткова	Повна	Повна	Повна
Формалізація просування	Відсутня	Відсутня	Часткова	Часткова
Рівні перевірки	0	0	1	2
Виявлення дрейфу	Ручне	Ручне	Cron	Автоматичне
Відкат з безпечністю	Ручний	Ручний	`git revert`	Історія

Жоден підхід:

- не розмежовує валідацію коду (σ_3) і перевірку плану (σ_5);
- не формалізує передумову готовності ready(e_i);
- не класифікує дрейф для уникнення хибних сповіщень.

Двопроекційна модель життєвого циклу

Горизонтальна проекція — автомат A_e з 8 стадій:



Вертикальна проекція — простір середовищ $(\mathcal{E}, \preceq)$:

$$e_{\text{dev}} \preceq e_{\text{staging}} \preceq e_{\text{prod}}$$

Відмінності: (1) σ_5 як самостійна стадія; (2) замкнений цикл $\sigma_8 \to \sigma_4$.

Формальний апарат

Тричленна модель стану:

- S_e^d — бажаний (desired) стан, $S_e^d = \text{eval}(C_e)$;
- S_e^r — записаний (recorded) стан, файл `tfstate`;
- S_e^a — фактичний (actual) стан у хмарі.

Ключові дельти:

$$\Delta_e^{\text{plan}} = S_e^d \setminus S_e^r \quad \text{(операція } \texttt{plan}\text{)}$$
$$\Delta_e^{\text{drift}} = S_e^r \setminus S_e^a \quad \text{(виявлення дрейфу)}$$
$$\Delta_e^{\text{eff}} = \Delta_e^{\text{drift}} \setminus I_e \quad \text{(ефективна дельта)}$$

Передумова просування:

$$\text{ready}(e_i) \iff \sigma(e_i) = \sigma_8 \wedge \Delta_{e_i}^{\text{plan}} = \varnothing \wedge \Delta_{e_i}^{\text{eff}} = \varnothing$$

Багаторівнева модель перевірки відповідності

$$P_e = P_e^{\text{code}} \cup P_e^{\text{plan}} \cup P_e^{\text{state}}$$

Рівень	Стадія	Артефакт	Приклади правил
P_e^{code}	σ_3 Validate	Код C_e	Захардкоджені секрети, відкриті порти
P_e^{plan}	σ_5 Comply	JSON-план Δ_e^{plan}	Заборона типів, видалення захищених ресурсів
P_e^{state}	σ_8 Monitor	Стан S_e^a	Відсутність публічного доступу

Ієрархія успадкування:

$$P_e = P_{\text{global}} \cup P_{\text{env}(e)} \cup P_{\text{comp}(e)}$$

Зростання суворості вздовж $(\mathcal{E}, \preceq)$: одна й та сама політика може бути *advisory* у e_{dev} і *hard-mandatory* у e_{prod}.

Трикласова класифікація дрейфу

$$\text{class}(r, \Delta_e^{\text{drift}}, P_e^{\text{state}}) \in \{\text{critical}, \text{actionable}, \text{informational}\}$$

- **Критичний** — порушення hard-mandatory політики
  $\Rightarrow$ автоматичне усунення через $\sigma_8 \to \sigma_4$.
- **Операбельний** — розбіжність без порушення політик
  $\Rightarrow$ сповіщення оператора, ручне підтвердження.
- **Інформаційний** — атрибут з I_e (динамічні IP, ротовані сертифікати)
  $\Rightarrow$ виключається з ефективної дельти Δ_e^{eff}.

Результат: частка хибних сповіщень $F_{\text{false}} = 0$.

Механізм відкату з предикатом безпечності

Предикат безпечності відкату:

$$\text{safe}(e, v, v') \iff \forall r \in (S_e^{r,v} \setminus S_e^{r,v'}) : \text{rev}(r) = 1$$

Зворотність ресурсу:

- $\text{rev}(r) = 1$ — ресурс зворотний (EC2, SG, DNS): безпечний відкат.
- $\text{rev}(r) = 0$ — ресурс незворотний (RDS з даними, S3 з об'єктами): примусовий ручний σ_6.

Інтеграція з просуванням:

- Версіоновані знімки $\hat{C}_e^v = \langle m_{\text{root}}, \Theta_e, v \rangle$.
- Історія станів $\mathcal{H}_e$ у віддаленому бекенді.
- Повний цикл перевірок при відкаті: $\sigma_4 \to \sigma_5 \to \sigma_6 \to \sigma_7$.

Реалізація на Scalr + AWS

Платформа: Scalr — промислова система оркестрації Terraform/OpenTofu з підтримкою OPA.

Інфраструктура: веб-сервіс на AWS
VPC, EC2, RDS PostgreSQL, S3, IAM — 27 керованих ресурсів, ∼1200 рядків HCL.

Відображення моделі на платформу:

Компонент моделі	**Реалізація у Scalr**
Простір $(\mathcal{E}, \preceq)$	Account → Environment → Workspace
Ізоляція стану $I_S = 1$	Окремий стан для кожного Workspace
Автомат A_e	Run pipeline з hook-ами OPA
P_e^{code}, P_e^{plan}, P_e^{state}	Три незалежні точки OPA
Моніторинг σ_8	Drift detection (refresh-only)
Знімки $\hat{C}_e^v$	Приватний реєстр модулів з тегами

Результати експериментів

Метрика	Terraform CLI	GitOps CI/CD	Метод (Scalr)
Покриття стадій	$\sigma_1, \sigma_4, \sigma_6, \sigma_7$	σ_1–$\sigma_4, \sigma_6, \sigma_7$	σ_1–σ_8
$R_{\text{pre-apply}}$ (частка виявлення)	0,0	0,5	**1,0**
T_{detect} код / план	— / —	20–50 с / —	15–40 с / 45–90 с
T_{drift}	Не виявляється	Не виявляється	$\sim$ 7 хв
F_{false} (хибні сповіщення)	0,33	0,33	**0,0**
N_{manual} (ручні кроки)	9	4	**1**

Підтверджено кожен із трьох пунктів наукової новизни:

- Повне покриття σ_1–σ_8 (новизна 1).
- $N_{\text{manual}} = 1$ + спрацювання safe (новизна 2).
- $R_{\text{pre-apply}} = 1$ + $F_{\text{false}} = 0$ (новизна 3).

Наукова новизна

1. Вперше запропоновано двопроекційну модель життєвого циклу у вигляді скінченного автомата A_e з 8 стадій та простору $(\mathcal{E}, \preceq)$, що відрізняється виокремленням σ_5 та замкненим циклом $\sigma_8 \to \sigma_4$.

2. Отримав подальший розвиток метод просування змін — формалізовано ready(e_i), версіоновані знімки $\hat{C}_e^v$, предикат safe(e, v, v') з урахуванням зворотності ресурсів.

3. Удосконалено підхід до перевірки відповідності — багаторівнева модель $(P^{\text{code}}, P^{\text{plan}}, P^{\text{state}})$ з ієрархічним успадкуванням та трикласовою класифікацією дрейфу з ефективною дельтою.

Висновки

1. Обґрунтовано вибір directory-based патерну ізоляції за 5 критеріями: $I_S = 1$, $B_R = 1$ без додаткових умов.
2. Розроблено двопроекційну модель життєвого циклу (автомат A_e + простір $(\mathcal{E}, \preceq)$).
3. Обґрунтовано метод з формалізованою передумовою ready, предикатом безпечності safe, трирівневою моделлю політик.
4. Реалізовано на Scalr + AWS, підтверджено: $R_{\text{pre-apply}} = 1$, $F_{\text{false}} = 0$, $N_{\text{manual}} = 1$.

Перспективи: мультихмарні конфігурації, формалізація оцінки вартості, класифікація дрейфу методами ML.

Апробація: опубліковано наукову статтю у фаховому виданні України.

Доповідь завершено.

Дякую за увагу!

Коломицький Денис Євгенійович
гр. КІ2м-24-2
Хмельницький національний університет

Протокол аналізу звіту подібності експертом

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Денис КОЛОМИЦЬКИЙ

Співавтор:

Назва: Узагальнений метод керування життєвим циклом Terraform-інфраструктури для кількох середовищ

Експерт: Олег САВЕНКО

Підрозділ: Кафедра комп'ютерної інженерії та інформаційних систем

Коефіцієнт подібності 1:5.6%

Коефіцієнт подібності 2:1.67%

Мікропробіли: 0

Заміна букв: 5215

Інтервали: 0

Білі знаки: 0

Дата створення звіту: 2026-04-24 06:46:05.0

Після аналізу Звіту подібності констатую наступне:

☒ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

Дата / Мічепорук Л.О. експерт

Fri Apr 24 07:32:21 EEST 2026, Медзатий Дмитро Миколайович, Хмельницький національний університет, ХНУ

Anti-Plagiarism (http://ap.km.ua) v-15.701

Максимальне співпадіння з одним документом 6.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 47%**

	Документ		Сумарний збіг по Базі Даних	
ID: 270669 Назва: МКР Узагальнений метод керування життєвим циклом Terraform-інфраструктури для кількох середовищ Додано в БД: 2026-04-24 Автора: Денис КОЛОМИЦЬКИЙ Керівники: Олег САВЕНКО Консультанти: Опоненти:	Символи	Лексеми	Символи	Лексеми
	137489	1063	8961 (7%)	75 (7%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ МАГІСТРА

Здобувач: Денис КОЛОМИЦЬКИЙ

Тема: «Узагальнений метод керування життєвим циклом Terraform-інфраструктури для кількох середовищ»

Спеціальність: 123 «Комп'ютерна інженерія»

Обсяг кваліфікаційної роботи магістра:

Кількість листів креслень – ; кількість сторінок записки 80 с.

1. Короткий зміст роботи та прийнятих рішень У роботі запропоновано узагальнений метод керування життєвим циклом Terraform-інфраструктури для кількох середовищ.

2. Висновок про відповідність роботи дипломному завданню Кваліфікаційна робота магістра відповідає виданому завданню.

3. Характеристика виконання кожного розділу, ступінь використання останніх досягнень науки і техніки і передових методів роботи: У першому розділі проведено аналіз парадигми Infrastructure as Code, патернів ізоляції середовищ та практик забезпечення якості й безпеки. У другому розділі запропоновано двопроекційну модель життєвого циклу у вигляді скінченного автомата та обґрунтовано метод просування змін. У третьому розділі розроблено алгоритми управління життєвим циклом та класифікації конфігураційного дрейфу. У четвертому розділі реалізовано метод на платформі Scalr та проведено експериментальну оцінку ефективності.

4. Позитивні сторони роботи: Робота спирається на формалізований апарат теорії скінченних автоматів та теорії множин. Запропонована багаторівнева модель перевірки відповідності та трикласова класифікація конфігураційного дрейфу мають наукову новизну. За темою роботи опубліковано наукову статтю.

5. Негативні сторони роботи: Експериментальне дослідження проведено на одній хмарній платформі, що не повною мірою розкриває поведінку методу в мультихмарних сценаріях з кількома провайдерами.

6. Оцінка графічного оформлення та пояснювальної записки роботи: –

7. Відгук про роботу в цілому: Робота виконана на високому науково-технічному рівні, має теоретичне значення та практичну цінність для галузі DevOps та хмарної інженерії.

8. Інші зауваження: –

9. Оцінка кваліфікаційної роботи магістра:

Розглянувши позитивні та негативні сторони представленої кваліфікаційної роботи магістра вважаю, що робота заслуговує оцінки «відмінно» 90 (А).

Рецензент (прізвище, ім'я, по батькові, посада, місце роботи)

д.т.н., професор, Мартинюк В.В., професор кафедри автоматизації, комп'ютерно-інтегрованих технологій та робототехніки

«1» грудня 2026 р.

Зав. кафедри КІІС
д-р. філософії Ользі ПАВЛОВІЙ

Денис КОЛОМИЦЬКИЙ

ПІБ здобувача вищої освіти

ФІТ, 2 курсу, групи КІ2м-24-2

ЗАЯВА

З правилами чинного Положення про систему забезпечення академічної доброчесності у Хмельницькому національному університеті, згідно з яким виявлення академічного плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів академічної відповідальності, ознайомлений (а). Про використання спеціалізованих програмних засобів (СПЗ) StrikePlagiarism та Anti-Plagiarism для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність академічного плагіату оповіщений (а). Надаю університету право на передачу моєї роботи для обробки та збереження в базах даних СПЗ і використання роботи для виявлення академічного плагіату в інших роботах, які перевіряються СПЗ.

Також надаю свою згоду на обробку й збереження університетом моєї роботи в Інституційному репозитарії Хмельницького національного університету.

Робота надається для перевірки в електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

1 травня 2026 року

РІШЕННЯ ЕКСПЕРТНОЇ КОМІСІЇ
КАФЕДРИ КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Назва кваліфікаційної роботи Узагальнений метод керування життєвим циклом Terraform-інфраструктури для кількох середовищ
Автор Денис КОЛОМИЦЬКИЙ
Освітня програма Комп'ютерна інженерія та програмування
Рівень вищої освіти другий (магістерський)
Спеціальність 123 Комп'ютерна інженерія
Науковий керівник: д.т.н., професор Олег САВЕНКО

На основі аналізу кваліфікаційної роботи на дотримання вимог академічної доброчесності (у т.ч. відсутності ознак академічного плагіату) з урахуванням результатів перевірки роботи спеціалізованим програмним засобом(ами) комісія зробила такий висновок:

№	Висновок	Позначка про відповідність
1	Ознаки академічного плагіату	
1.1	Запозичення, виявлені в роботі, є законними і не є академічним плагіатом (далі – зазначаються підстави віднесення запозичень до правомірних, якщо потрібно). Робота приймається до захисту.	відповідає
1.2	Виявлені запозичення не є академічним плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи (далі – зазначаються детальні та аргументовані підстави віднесення запозичень до правомірних). Робота приймається до захисту, але має бути відкоригована.	
1.3	Виявлені запозичення не є академічним плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота може бути допущена до захисту після того як буде відкоригована та доопрацьована і успішно пройде повторну перевірку на академічний плагіат.	
1.4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття текстових запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
2	Інші види порушень академічної доброчесності	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) усі запозичення фрагментарні, або мають належним чином оформленні посилання;
2) окремі виявлені збіги є загальновживаними фразами або виразами, про що свідчить посилання системи на збіг з джерелами на один фрагмент речення;
3) всі зафіксовані системою ознаки модифікації тексту відносяться до комбінування латинських символів зі україномовними скороченнями індексів в формулах, що не є модифікацією тексту.
4) значна частина знайденого плагіату відноситься до списку використаних джерел

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів/ ідентичності/схожості StrikePlagiarism, складає 5.6% і адресується до 0 першоджерел; та системою Anti-Plagiarism складає 6%, що, з урахуванням наведених обґрунтувань, відповідає характеру наукового дослідження і свідчить на користь кваліфікаційної роботи.

24.04.2026

Завідувач кафедри _____ Підпис — Ольга ПАВЛОВА Ім'я, ПРІЗВИЩЕ

Гарант освітньої програми _____ Підпис — Олег САВЕНКО Ім'я, ПРІЗВИЩЕ

Керівник кваліфікаційної роботи _____ Підпис — Олег САВЕНКО Ім'я, ПРІЗВИЩЕ