Міністерство освіти і науки України
Харківський національний університет імені В. Н. Каразіна
Навчально-науковий інститут комп'ютерних наук та штучного інтелекту
Кафедра комп'ютерних систем та робототехніки

До захисту допущено
Кафедрою комп'ютерних систем та робототехніки
протокол № __ від __ грудня 2025р.

завідувач кафедри _____ Максим ХРУСЛОВ
(підпис)

«___»_____ 2025 р.

Кваліфікаційна робота

здобувача другого (магістерського) рівня вищої освіти

«КОМП'ЮТЕРНА МОДЕЛЬ САМОНАЛАШТУВАННЯ VIRTUAL MACHINES ENVIRONMENT З ВИКОРИСТАННЯМ ANSIBLE PULL APPROACH»

Спеціальність ***123 – Комп'ютерна інженерія.***
Освітня програма ***Комп'ютерна інженерія***

Виконавець _____ **Владислав САМОЙЛЕНКО**
(підпис)

Науковий керівник _____ **Ольга МОРОЗ**
(підпис)

Харків – 2025

АНОТАЦІЯ

Пояснювальна записка до кваліфікаційної роботи магістра складається зі вступу, трьох розділів, висновків, списку використаних джерел і десятьох додатків. Загальний обсяг роботи складає 104 сторінки, із яких 65 сторінок основної частини з 35 рисунками, 26 найменувань списку використаних джерел та десятьма додатками.

Мета роботи - підвищення рівня автономності, масштабованості та безпеки управління конфігурацією серверної інфраструктури порівняно з класичною push-моделлю шляхом створення та оцінки комп'ютерної моделі самоналаштування середовища віртуальних машин на основі підходу Ansible Pull з інтеграцією HashiCorp Vault та AWS IAM. Досягнення мети передбачає експериментальне підтвердження переваг запропонованої моделі за допомогою кількісних метрик, таких як швидкість збіжності конфігурації, стійкість до відмов, безпечність роботи з секретами та ефективність масштабування при зростанні кількості вузлів.

Об'єкт дослідження. Процеси автоматизованого управління конфігурацією серверної інфраструктури у середовищі віртуальних машин.

Предмет дослідження. Pull-based модель управління конфігурацією на основі Ansible Pull із інтеграцією механізмів динамічного керування секретами (HashiCorp Vault) та контролю доступу на основі ролей (AWS IAM).

Область застосування. Рекомендації та результати роботи призначені для застосування при проєктуванні та впровадженні систем автоматизації інфраструктури в хмарних і гібридних середовищах, де потрібні: децентралізоване керування конфігурацією, зниження операційних витрат, підвищення безпеки через динамічне управління секретами, автономність вузлів та просте масштабування.

Ключові слова: *комп'ютерна модель, управління конфігурацією, Ansible Pull, Infrastructure As Code, DevOps, автоматизація інфраструктури, Hashicorp Vault, Terraform, масштабованість, відмовостійкість, AWS*

ABSTRACT

The explanatory note to the master's thesis consists of an introduction, three chapters, conclusions, a list of references, and ten appendices. The total volume of the thesis is 104 pages, of which 65 pages are the main part with 35 figures, 26 titles in the list of references, and ten appendices.

The goal of this work is to increase the autonomy, scalability, and security of server infrastructure configuration management compared to the classic push model by creating and evaluating a computer model of self-configuration of virtual machine environments based on the Ansible Pull approach with integration of HashiCorp Vault and AWS IAM. Achieving this goal involves experimentally confirming the advantages of the proposed model using quantitative metrics such as configuration convergence speed, fault tolerance, security of working with secrets, and scaling efficiency as the number of nodes increases.

Object of study. Processes of automated configuration management of server infrastructure in a virtual machine environment.

Subject of research. Pull-based configuration management model based on Ansible Pull with integration of dynamic secret management mechanisms (HashiCorp Vault) and role-based access control (AWS IAM).

Scope of application. Recommendations and results of the work are intended for use in the design and implementation of infrastructure automation systems in cloud and hybrid environments where the following are required: decentralized configuration management, reduced operating costs, increased security through dynamic secret management, node autonomy, and easy scalability.

Keywords: *computer model,* configuration management, Ansible Pull, Infrastructure As Code, DevOps, infrastructure automation, Hashicorp Vault, Terraform, scalability, fault tolerance, AWS

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ

AWS AMI (Amazon Machine Image) - образ віртуальної машини в AWS.

AWS (Amazon Web Services) - хмарний провайдер обчислювальних ресурсів.

AWS ARN (Amazon Resource Names) - унікальний ідентифікатор AWS ресурсу.

AWS STS (Security Token Service) - сервіс, що дозволяє запитувати тимчасові облікові дані з обмеженими правами для користувачів AWS.

CLI (Command Line Interface) - командний рядок для взаємодії з ОС.

DNS (Domain Name System) - система перетворення доменного імені у IP.

EBS (Elastic Block Store) - сервіс блокових сховищ даних AWS.

EC2 (Elastic Compute Cloud) - сервіс віртуальних машин AWS.

IaC (Infrastructure as Code) - інфраструктура як код.

IAM (Identity and Access Management) - управління ідентифікацією та доступом в AWS.

JSON (JavaScript Object Notation) - текстовий формат обміну даними.

RDS (Relational Database Service) - сервіс, що спрощує налаштування, експлуатацію та масштабування реляційних баз даних у хмарі.

SSH (Secure Shell) - протокол безпечного віддаленого доступу до систем.

URL (Uniform Resource Locator) - стандартизована адреса певного ресурсу в інтернеті (чи деінде).

VPC (Virtual Private Cloud) - ізольована віртуальна мережа в AWS.

S3 (Simple Storage Service) - сервіс-сховище даних, один з Amazon Web Services.

YAML (YAML Ain't Markup Language) - мова серіалізації даних, що використовується для конфігураційних файлів.

ВМ - віртуальна машина.

ОС - операційна система.

ПЗ - програмне забезпечення.

ВСТУП

Актуальність теми дослідження: У сучасних IT-системах зростання масштабів і складності розподілених та хмарних середовищ вимагає автоматизації конфігурування та підтримки великої кількості ВМ. Традиційні централізовані push-підходи до управління конфігураціями стикаються з обмеженнями масштабованості, надійності та безпеки, оскільки залежать від постійного підключення до control node, який стає єдиною точкою відмови.

Методологія Infrastructure as Code (IaC) у поєднанні з підходом Ansible Pull пропонує альтернативу: віртуальні машини самостійно отримують та застосовують актуальні конфігурації з централізованого репозиторію. Така pull-модель підвищує відмовостійкість, покращує горизонтальне масштабування та узгоджується з принципами DevOps. Актуальність роботи визначається потребою підприємств у автоматизованих, масштабованих і безпечних рішеннях для управління хмарною інфраструктурою на основі pull-based конфігураційного менеджменту.

Мета і завдання дослідження: Метою квалфікаційної роботи є розробка та впровадження комп'ютерної моделі самоналаштування ВМ у хмарному середовищі на основі Ansible Pull підходу з автоматизованим управлінням конфігураціями та секретами.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

1) Провести аналіз існуючих систем управління конфігураціями та обґрунтувати доцільність використання pull-моделі для розподілених хмарних середовищ.
2) Розробити архітектуру системи самоналаштування ВМ з інтеграцією інструментів Infrastructure as Code (Terraform, Packer, Ansible).
3) Спроектувати механізм безпечної автентифікації та авторизації на базі AWS IAM та HashiCorp Vault для автоматичного отримання секретів без їх зберігання на цільових вузлах.
4) Реалізувати систему автоматизованого створення базових образів ВМ за допомогою Packer з вбудованою підтримкою Ansible Pull.

5) Розробити Ansible роль та скрипти для забезпечення регулярного самоконфігурування ВМ через SystemD таймери.
6) Імплементувати динамічний inventory механізм на базі AWS EC2 plugin для автоматичної категоризації та групування хостів.

Об'єкт дослідження - процеси автоматизованого управління конфігураціями ВМ у хмарних обчислювальних середовищах.

Предмет дослідження - методи та засоби самоналаштування ВМ на основі pull-моделі з використанням технологій Infrastructure as Code та автоматизованого управління секретами.

Методи дослідження - Метод системного аналізу, метод структурного моделювання, метод декларативного програмування, метод порівняльного аналізу.

Практичне значення одержаних результатів.

Практична цінність кваліфікаційної роботи полягає в розробці повнофункціональної системи самоналаштування ВМ, яка може бути впроваджена для:

- Автоматизації управління великою кількістю ВМ у хмарній інфраструктурі AWS без необхідності централізованого control node.
- Підвищення відмовостійкості системи управління конфігураціями через децентралізацію процесу виконання та усунення єдиної точки відмови.
- Покращення безпеки шляхом уникнення довгострокового зберігання credentials на серверах та використання динамічних токенів з обмеженим терміном дії.
- Спрощення масштабування інфраструктури, де нові ВМ автоматично інтегруються в систему управління конфігураціями без ручного втручання.
- Забезпечення configuration compliance через регулярне автоматичне приведення стану серверів до бажаної конфігурації та виявлення configuration drift.

Розроблена система впроваджена у реальному середовищі та використовується для управління десятками ВМ.

РОЗДІЛ 1.
ТЕОРЕТИЧНІ ОСНОВИ УПРАВЛІННЯ КОНФІГУРАЦІЯМИ

Управління конфігураціями є фундаментальною складовою сучасних підходів до експлуатації IT-інфраструктури. Зростання складності та масштабів розподілених систем вимагає застосування автоматизованих методів для забезпечення консистентності конфігурацій, швидкого розгортання змін та підтримки бажаного стану сотень або тисяч серверів.

Концепція Infrastructure as Code (IaC) трансформувала традиційні підходи до управління інфраструктурою, пропонуючи декларативний опис систем у вигляді версіонованого коду. Це забезпечує можливість застосування практик розробки програмного забезпечення до управління інфраструктурою, включаючи контроль версій, код-рев'ю та автоматизоване тестування змін перед їх впровадженням у production-середовище.

Серед інструментів управління конфігураціями Ansible займає особливе місце завдяки простоті використання, архітектурі без агентів та потужним можливостям автоматизації. Традиційна push-модель Ansible передбачає централізоване виконання конфігурацій з control node, проте для великих розподілених середовищ альтернативна pull-модель пропонує переваги в масштабованості, відмовостійкості та безпеці.

Даний розділ присвячено огляду теоретичних основ, необхідних для розуміння принципів побудови системи самоналаштування ВМ на базі pull-моделі управління конфігураціями. Матеріал розділу формує теоретичний фундамент для подальшого проектування та реалізації системи самоналаштування ВМ, представленої в наступних розділах кваліфікаційної роботи.

1.1. Infrastructure as Code та системи управління конфігураціями

1.1.1. Принципи Infrastructure as Code

Infrastructure as Code (IaC) — це підхід до управління та провізіонінгу IT-інфраструктури за допомогою текстових конфігураційних файлів замість ручного

налаштування. Він з’явився як відповідь на потребу масштабування інфраструктури та забезпечення її послідовності в умовах зростання розподілених систем.

Традиційне ручне налаштування серверів призводило до розбіжностей у конфігураціях (configuration drift), складності відтворення середовищ, проблем з аудитом змін та сильної залежності від окремих адміністраторів. З розвитком віртуалізації, хмарних платформ (AWS, Azure, Google Cloud), DevOps-культури, інструментів автоматизації (Puppet, Chef) та гнучких методологій розробки (Agile) [1] стало можливим застосувати практики розробки ПЗ до інфраструктури: зберігати конфігурації в Git, тестувати їх та розгортати через CI/CD.

Переваги IaC:

- швидке розгортання нової інфраструктури (хвилини/години замість днів/тижнів);
- послідовність і стандартизація конфігурацій між середовищами;
- зменшення кількості ручних помилок завдяки автоматизації, код-рев’ю та тестам;
- суттєве спрощення відновлення після інцидентів завдяки можливості швидко відтворити середовище з коду.

Водночас впровадження IaC потребує навчання команд, організації процесів контролю якості, правильної роботи з секретами та продуманого дизайну репозиторіїв. Найкращі практики включають використання спеціалізованих сховищ секретів (HashiCorp Vault, AWS Secrets Manager) замість зберігання їх у Git, застосування лінтерів та статичного аналізу, автоматизовані тести для критичних змін, принцип найменших привілеїв, регулярний аудит через pull/merge requests [5], а також поділ конфігурацій за середовищами й модульну організацію коду.

Отже, IaC є фундаментальною методологією сучасного управління інфраструктурою, що переводить її з площини ручного конфігурування у площину керованого, версіонованого та тестованого коду. Розуміння цих принципів є базою для побудови системи самоналаштування віртуальних машин, яка розглядається у цій кваліфікаційній роботі; далі буде детально проаналізовано системи управління

конфігураціями, з особливим акцентом на Ansible як інструмент реалізації підходу IaC на практиці

1.1.2. Ansible. Архітектура та принципи роботи

Ansible — це відкрита платформа автоматизації IT-операцій від Red Hat, яка забезпечує управління конфігураціями, оркестрацію та розгортання додатків. Вона відрізняється простотою, агентлес-архітектурою (через SSH без агентів) і мінімальними вимогами до цільових хостів (SSH + Python), що робить її придатною як для малих команд, так і великих корпоративних середовищ.

Інструмент був створений у 2012 році Майклом ДеХааном для спрощення автоматизації шляхом використання зрозумілих YAML-файлів замість складних DSL-мов [6]. У 2015 році Ansible придбала компанія Red Hat, після чого з’явилися корпоративні надбудови, зокрема Ansible Tower / Automation Platform.

Архітектура Ansible побудована за моделлю control node - managed nodes. Control node - сервер, де зберігаються playbooks, інвентар та конфігурація, managed nodes - цільові системи, до яких Ansible підключається по SSH, тимчасово копіює та виконує необхідні модулі Python, а потім видаляє залишки. Агентів на хостах встановлювати не потрібно, що спрощує розгортання та зменшує операційні й безпекові ризики.

Основним будівельним блоком Ansible є модулі - невеликі програми, що виконують конкретні дії (управління пакетами, файлами, сервісами, хмарною інфраструктурою, БД, мережевим обладнанням тощо) [7]. Більшість із них ідемпотентні: повторний запуск змінює стан лише за потреби. Модулі об’єднуються в tasks, які групуються в playbooks — YAML-файли з послідовністю операцій та описом бажаного стану системи.

Для повторного використання коду застосовуються ролі (roles) [8] - стандартизована структура каталогів (tasks, templates, files, defaults, handlers, meta тощо), яка інкапсулює логіку налаштування певного компоненту (наприклад, Tomcat). Готові ролі та Collections публікуються у репозиторії Ansible Galaxy, що дозволяє використовувати вже відпрацьовані рішення замість розробки “з нуля”.

Інвентар (inventory) визначає список керованих хостів і їх групи. Він може бути статичним (INI/YAML) або динамічним (inventory plugins, які автоматично отримують список інстансів із хмарних провайдерів чи систем віртуалізації) [9]. Приклад простого YAML-інвентаря наведено на рис. 1.1. Групи можуть бути вкладеними, що дозволяє гнучко задавати конфігурації для різних рівнів (тип середовища, роль сервера тощо).

```
---
all:
  children:
    kafka:
      hosts:
        kafka-01.test.com: {}
    app:
      children:
        api:
          hosts:
            app-01.test.com: {}
            app-02.test.com: {}
        ui:
          hosts:
            ui-01.test.com: {}
            ui-02.test.com: {}
        job:
          hosts:
            job-01.test.com: {}
            job-02.test.com: {}
```

Рисунок 1.1 - Приклад найпростішого інвентарю в Ansible.

Ansible має ієрархію змінних. Більш специфічні значення перевизначають загальні (рис. 1.2), що дає змогу будувати спільні базові конфігурації з можливістю тонкого налаштування окремих груп і хостів.

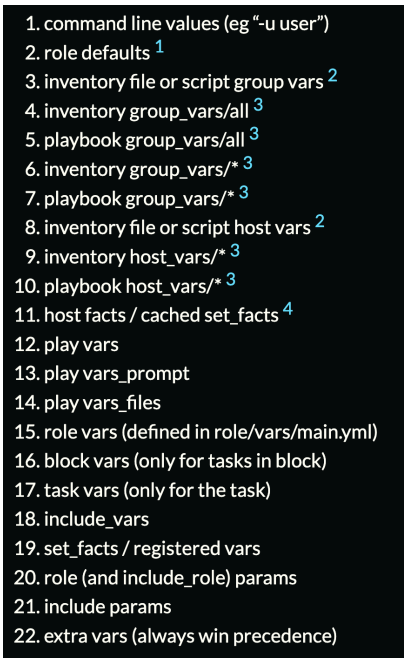

Рисунок 1.2 - Ієрархія змінних в Ansible.

Для генерації конфігурацій використовується шаблонізація Jinja2. Один шаблон може динамічно підлаштовуватися під конкретний хост: підставляти значення змінних, використовувати умови, цикли, фільтри. Наприклад, конфігурація MySQL може автоматично обирати параметри пам'яті на основі обсягу RAM, типу ролі (master/slave) або групи інвентаря.

1.1.3. Ansible Pull. Архітектура. Відмінності від Ansible Push

Ansible Pull — це альтернативний режим роботи Ansible, який реалізує pull-модель управління конфігураціями [10]. Замість того щоб центральний control node по SSH застосовував конфігурацію на всі хости (push-модель), кожна ВМ самостійно періодично витягує актуальний код з Git-репозиторію та виконує його локально через ansible-pull. Такий підхід особливо актуальний для розподілених та хмарних інфраструктур з великою кількістю динамічних ВМ, де централізований control node стає вузьким місцем продуктивності та єдиною точкою відмови [11]. Приклад push архітектури з використанням Jenkins як вузла керування показаний на Рис. 1.3.

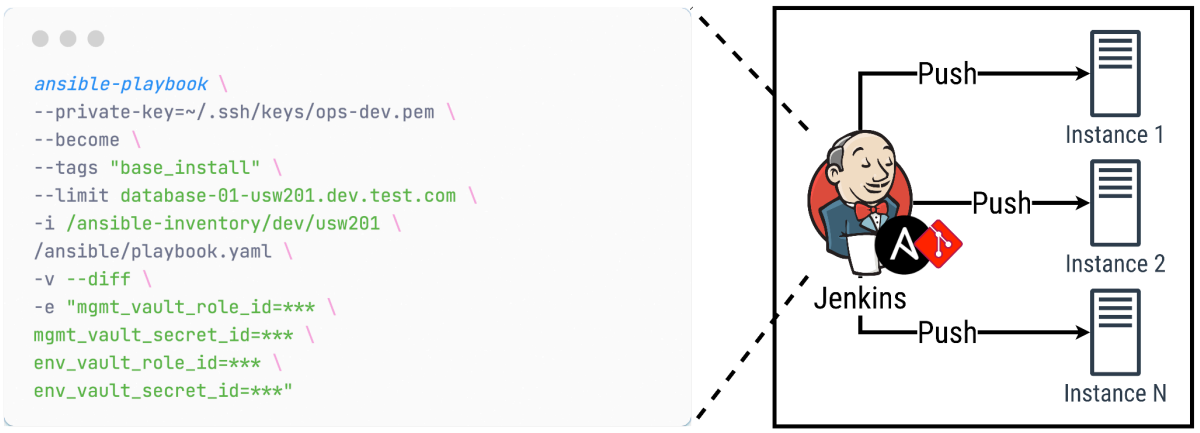

Рисунок 1.3 - Приклад push архітектури з використанням Jenkins як вузла керування.

Порівняно з push-моделлю, у якій control node активно керує виконанням автоматизації (ініціює з'єднання, розподіляє задачі, збирає результати), у pull-підході кожен хост стає автономним агентом. Це забезпечує кращу масштабованість: навантаження розподіляється по вузлах, а не концентрується на одному control node. Також pull-модель природно підтримує самовідновлення та усунення configuration drift — періодичний запуск ansible-pull повертає систему до бажаного стану після ручних або випадкових змін. Приклад pull архітектури показаний на Рис. 1.4.

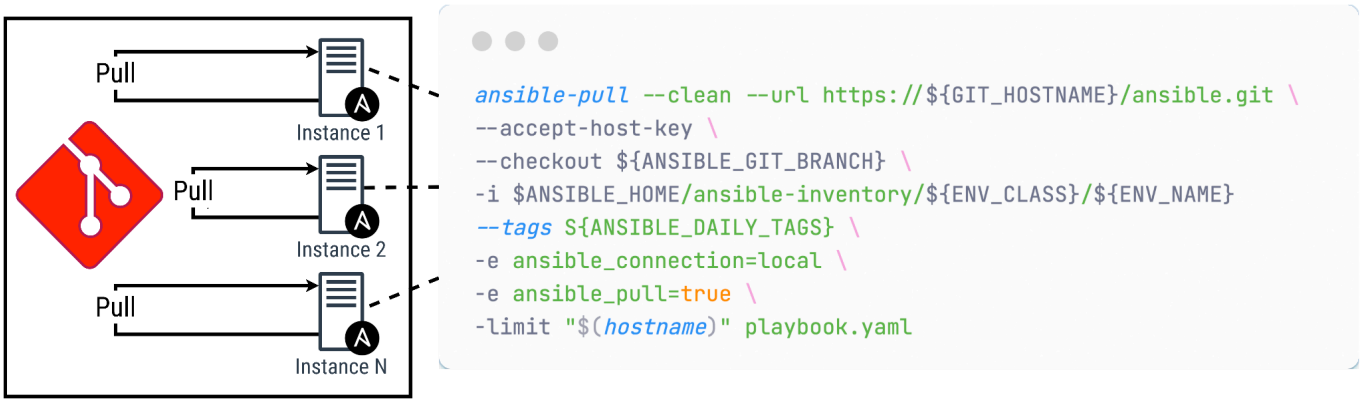

Рисунок 1.4 - Приклад pull архітектури.

В основі архітектури Ansible Pull лежить Git-репозиторій, що містить playbooks, ролі, змінні та інші артефакти конфігурації. Керовані вузли клонують або оновлюють локальну копію репозиторію та запускають ansible-pull із зазначенням URL репозиторію, гілки, цільового playbook і додаткових параметрів (теги тощо.

Виконання відбувається в локальному режимі (`--limit $(hostname)`, `-e ansible_connection=local`), тобто цільовим хостом є сама ВМ. Для коректної роботи ім’я вузла має бути присутнє в inventory.

Разом з тим, у pull-підходу є обмеження. Затримка застосування змін унеможливлює миттєве оновлення всіх вузлів, оскільки конфігурації застосовуються з певним інтервалом. Це ускладнює сценарії, які потребують строго синхронізованих оновлень (наприклад, поетапний апгрейд кластерів). Управління секретами є більш складним, оскільки playbooks лежать у Git. Це забороняє зберігати там відкриті паролі, ключі чи сертифікати. Потрібна інтеграція з системами на кшталт HashiCorp Vault для динамічного отримання секретів на основі ідентичності вузла, що ускладнює конфігурацію та вимагає високої доступності сервісу секретів.

У рамках даної кваліфікаційної роботи Ansible Pull використовується як базовий механізм самоналаштування віртуальних машин, а далі розглядається його інтеграція з Terraform, Packer, HashiCorp Vault та іншими елементами технологічного стеку для забезпечення безпеки, надійності та ефективної автоматизації.

1.2. Технологічний стек для побудови самоналаштовуваних систем

1.2.1. Terraform для управління інфраструктурою

Terraform — це відкритий інструмент Infrastructure as Code від HashiCorp, який дозволяє декларативно описувати та керувати хмарною інфраструктурою за допомогою мови HCL (HashiCorp Configuration Language) [12]. Конфігураційні файли .tf описують ресурси, змінні, вихідні значення та залежності, фокусуючись на тому, що має існувати, а не як це створити. Завдяки цьому конфігурації є самодокументованими та зрозумілими навіть без глибоких знань конкретного хмарного провайдера.

Ключовим елементом Terraform є провайдери (providers) — плагіни для інтеграції з API різних платформ: AWS, Azure, Google Cloud тощо. Кожен провайдер надає типи ресурсів (ВМ, мережі, БД, моніторинг, SaaS-сервіси та навіть on-prem системи). Підтримка понад 1700 провайдерів дозволяє керувати гетерогенною інфраструктурою з єдиної точки входу.

Ресурси (resources) — базові блоки, що описують окремі компоненти: EC2 інстанси, VPC, IAM ролі, RDS, S3 тощо. Terraform автоматично виводить залежності між ресурсами на основі посилань у конфігурації та забезпечує коректний порядок створення/видалення.

Критично важливою частиною є стан (state). State-файл зберігає поточний стан інфраструктури та використовується для обчислення різниці між бажаним описом у .tf та реальною інфраструктурою. Під час terraform apply генерується план змін із переліком ресурсів, що будуть створені, змінені або видалені. Для командної роботи Terraform підтримує віддалені бекенди (наприклад, S3 з блокуванням) [13], що забезпечує синхронізацію, запобігає конфліктам і дає аудит змін. У продакшні використання remote state є обов’язковою практикою.

Модулі Terraform дозволяють інкапсулювати типові набори ресурсів у повторно використовувані компоненти (наприклад, стандартний EC2 інстанс з security groups, IAM роллю, дисками тощо). Публічний Terraform Registry містить тисячі готових модулів, які можна напряму використовувати чи адаптувати. У контексті системи самоналаштовуваних ВМ Terraform виконує роль інструменту провізіонінгу. Створює мережеву інфраструктуру (VPC, subnet, security groups), налаштовує IAM ролі та політики для доступу до, наприклад, Vault, розгортає EC2-інстанси на базі образів Packer, передає скрипт через user_data, який запускає процес самоконфігурації, встановлює та реєструє ansible-pull як systemd-сервіс з таймером і стартує початкове налаштування.

1.2.2. Packer для створення образів

Packer є інструментом від HashiCorp для автоматизованого створення ідентичних образів ВМ для різних платформ з єдиної конфігурації [14]. У контексті хмарних інфраструктур підхід до управління образами операційних систем є критично важливим фактором, який впливає на швидкість розгортання нових екземплярів, консистентність конфігурацій та загальну надійність систем. Замість ручного налаштування базових образів або використання стандартних образів від дистриб'юторів з наступним тривалим конфігуруванням при кожному створенні ВМ, Packer дозволяє автоматизувати процес створення golden images - заздалегідь

підготовлених образів з встановленим базовим програмним забезпеченням та налаштуваннями.

Концепція golden images передбачає створення стандартизованого базового образу, який містить операційну систему, оновлення безпеки, необхідні системні пакети, агенти моніторингу та базові конфігурації [15]. Packer використовує JSON або HCL конфігураційні файли для опису процесу створення образів. Конфігурація складається з декількох ключових секцій, кожна з яких відповідає за певний етап процесу створення образу. Source визначають цільову платформу та спосіб створення образу: amazon-ebs для AWS AMI, azure-arm для Azure, googlecompute для GCP, virtualbox-iso для локальних VirtualBox образів. Кожен source налаштовується специфічними параметрами: базовий образ для використання, тип інстансу для збірки, регіон розміщення, credentials для автентифікації.

Provisioners виконують налаштування образу після створення базового інстансу [16]. Типовими provisioners є shell для виконання bash-скриптів, ansible для застосування Ansible playbooks, file для копіювання файлів на ВМ. Типовий робочий процес Packer складається з чітко визначених етапів. Спочатку Packer валідує конфігураційний файл через команду `packer validate`, перевіряючи синтаксичну коректність та узгодженість параметрів. При виконанні `packer build` Packer створює тимчасову ВМ на цільовій платформі, використовуючи параметри з секції build. Наприклад, для AWS запускається EC2 інстанс з базового AMI, до якого Packer підключається через SSH. Далі послідовно виконуються всі визначені provisioners: спочатку можуть встановлюватися системні пакети через shell-скрипти, потім застосовуватися Ansible playbooks для конфігурування, копіюватися необхідні файли.

Після успішного виконання всіх provisioners Packer створює snapshot або AMI з тимчасової ВМ, зберігаючи весь стан диска включно з встановленим програмним забезпеченням та конфігураціями. Тимчасова ВМ автоматично видаляється, залишаючи лише згенерований образ. Якщо визначені post-processors, вони виконуються на фінальному етапі для додаткової обробки образу. Весь процес

повністю автоматизований та детерміністичний: повторне виконання з тими ж вхідними параметрами гарантовано створить ідентичний образ.

У контексті архітектури самоналаштування Packer виконує роль підготовчого етапу, створюючи універсальні образи з мінімальним набором необхідних компонентів, тоді як специфічне конфігурування для конкретних ролей серверів делегується ansible-pull. Інтеграція Packer з Terraform забезпечує цілісний робочий процес. Packer створює базові образи, Terraform використовує ці образи для провізіонінгу ВМ з user_data скриптами, які ініціюють процес самоконфігурування.

1.2.3. HashiCorp Vault для управління секретами

HashiCorp Vault є централізованою системою управління секретами, яка забезпечує безпечне зберігання, доступ та розповсюдження чутливих даних, таких як паролі, API ключі, сертифікати та токени [17]. У сучасних розподілених системах управління секретами є критично важливою проблемою: зберігання паролів у конфігураційних файлах створює ризики безпеки, ручне розповсюдження секретів не масштабується, відсутність централізованого аудиту ускладнює відстеження доступу до чутливих даних. Vault вирішує ці виклики через централізоване сховище з гранульованим контролем доступу, шифруванням у спокої та під час передачі, детальним аудитом всіх операцій та підтримкою автоматичної ротації секретів.

Фундаментальний принцип Vault полягає у динамічній генерації короткотермінових секретів замість статичних та довготривалих. Замість зберігання пароля бази даних у конфігураційному файлі, додаток запитує Vault для отримання тимчасового пароля з обмеженим терміном дії (lease time), який автоматично інвалідується після закінчення lease або може бути відкликаний достроково. Це значно зменшує вікно вразливості у випадку компрометації credentials та забезпечує автоматичну ротацію без ручного втручання. Для статичних секретів, які не можуть бути динамічно згенеровані, Vault забезпечує шифроване зберігання з версіонуванням та контрольованим доступом на основі політик.

Методи автентифікації (auth methods) визначають способи ідентифікації клієнтів при доступі до Vault. Token auth є базовим методом, де клієнт надає токен для автентифікації. AWS IAM auth дозволяє EC2 інстансам та Lambda функціям

автентифікуватися використовуючи свою IAM роль без необхідності зберігання статичних облікових даних. Політики доступу (policies) визначають, які шляхи у Vault доступні для читання, запису або видалення конкретним токенам або ролям. Політики пишуться у декларативному форматі HCL, де явно вказуються дозволені операції для кожного path. Принцип найменших привілеїв реалізується через створення специфічних політик для кожної ролі або сервісу, надаючи доступ лише до мінімально необхідного набору секретів. Наприклад, веб-сервер може мати доступ лише до секретів database credentials та API keys для сторонніх сервісів, але не має доступу до SSH ключів або сертифікатів інших сервісів. Приклад policie у Vault показаний на Рис. 1.6.

```
{
   "path":{
      "dev/test/app/*":{
         "capabilities":[
            "read",
            "list"
         ]
      },
   }
}
```

Рисунок 1.6 - Приклад Vault policie.

У контексті архітектури самоналаштовуваних систем Vault виконує критично важливу роль забезпечення безпечного доступу до секретів під час виконання ansible-pull. ВМ, створена Terraform з прикріпленою IAM роллю, використовує Vault CLI для автентифікації через AWS IAM метод. Після успішної автентифікації отримується Vault токен, який дає доступ до специфічних секретів відповідно до політик, визначених для IAM ролі цього інстансу. Типові секрети включають Git credentials для клонування приватних репозиторіїв з Ansible конфігураціями, паролі баз даних, API ключі для сторонніх сервісів.

1.2.4. AWS EC2, IAM, Cloud-init

Amazon Web Services надає фундаментальні сервіси для побудови хмарної інфраструктури, серед яких EC2 та IAM відіграють ключову роль у реалізації

самоналаштовуваної системи. EC2 забезпечує обчислювальні ресурси у вигляді ВМ, а IAM управляє ідентичністю та правами доступу. Інтеграція цих компонентів створює основу для автоматизованого розгортання та самоконфігурування ВМ без ручного втручання. EC2 є сервісом ВМ AWS, який надає масштабовані обчислювальні ресурси в хмарі [19]. Екземпляри EC2 створюються на основі Amazon Machine Images (AMI) - попередньо налаштованих образів операційних систем, які можуть бути стандартними від AWS, marketplace образами від сторонніх постачальників або кастомними образами, створеними через Packer. Типи інстансів визначають конфігурацію апаратних ресурсів: кількість vCPU, обсяг оперативної пам'яті, мережеву пропускну здатність, тип сховища. Сімейства типів оптимізовані для різних workloads: загального призначення (t3, m5), обчислювально інтенсивні (c5), оптимізовані для пам'яті (r5), для сховищ даних (i3).

Мережева ізоляція забезпечується через Virtual Private Cloud (VPC), де екземпляри розміщуються в subnet з контрольованим доступом через Security Groups [20]. Security Groups функціонують як stateful firewall, визначаючи дозволені вхідні та вихідні підключення на основі протоколів, портів та джерельних IP адрес. Типово дозволяється вихідний HTTPS трафік для доступу до Git репозиторіїв та Vault, вхідний SSH для адміністративного доступу, специфічні порти налаштовуються відповідно до ролі сервера. Elastic Block Store (EBS) надає постійне блокове сховище для EC2 інстансів з підтримкою snapshots для backup та відновлення. Томи можуть динамічно збільшуватися без простою, шифруватися at-rest через AWS KMS, налаштовуватися з різними performance характеристиками (gp3 для balanced workloads, io2 для high IOPS баз даних).

IAM є сервісом управління ідентичністю та доступом, який контролює аутентифікацію та авторизацію до AWS ресурсів [21]. У контексті EC2 інстансів критично важливим є концепція IAM ролей. Доступ до ролі визначається через trust policy, яка вказує, хто може використовувати цю роль (в даному випадку - EC2 service), та прикріплені policies, які визначають дозволені операції з AWS ресурсами. Принцип найменших привілеїв реалізується через створення специфічних IAM політик для кожного типу серверів. Наприклад, веб-сервери можуть отримати доступ

лише до S3 бакетів зі статичними файлами та CloudWatch для логування, бази даних матимуть права на читання RDS snapshots, worker nodes можуть взаємодіяти з SQS чергами. Такий гранульований контроль мінімізує потенційні наслідки компрометації окремого інстансу, оскільки зловмисник отримує доступ лише до обмеженого набору ресурсів, визначених IAM політикою цього сервера.

Cloud-init є індустрійним стандартом для ініціалізації хмарних ВМ при першому завантаженні [22]. Підтримується всіма основними хмарними провайдерами (AWS, Azure, GCP, OpenStack) та дистрибутивами Linux, cloud-init читає конфігурацію або скрипти з metadata service та виконує визначені операції до того, як система стане повністю операційною. У AWS конфігурація передається через параметр user_data при створенні EC2 інстансу, який може містити shell скрипт, cloud-config YAML або multipart archive з декількома конфігураціями. User data виконується один раз при першому завантаженні інстансу з привілеями root. Типовий сценарій для використання включає встановлення або оновлення критичних пакетів, яких немає в базовому AMI, налаштування hostname на основі instance metadata, конфігурування системних параметрів (timezone, NTP, DNS), створення користувачів та SSH ключів тощо. Команда `cloud-init status` показує поточний стан виконання та успішність завершення всіх модулів. Для production систем рекомендується налаштування автоматичної передачі цих логів до централізованої системи логування (CloudWatch Logs, ELK) для моніторингу успішності ініціалізації нових інстансів та швидкого виявлення проблем у auto-scaling сценаріях.

У цілісній архітектурі самоналаштовуваних систем ці три компоненти тісно інтегровані. Terraform створює EC2 інстанс з визначеним AMI (створеним Packer), типом інстансу, subnet розміщенням та security groups. До інстансу прикріплюється IAM instance profile з роллю, яка надає права на автентифікацію до Vault через AWS IAM auth method. User data скрипт, згенерований з Terraform template, виконується при першому завантаженні через cloud-init: автентифікується до Vault використовуючи IAM роль інстансу, отримує Git credentials для клонування приватного репозиторію з Ansible конфігураціями, виконує початковий запуск

ansible-pull для базового конфігурування системи, встановлює SystemD timer для регулярного виконання ansible-pull кожний день. Після завершення cloud-init інстанс стає повністю самоналаштовуваним: Така багаторівнева інтеграція створює відмовостійку систему, здатну автоматично розгортати та підтримувати великі обсяги ВМ з мінімальним ручним втручанням.

1.2.5. SystemD та Git

SystemD та Git є фундаментальними технологіями сучасних Linux-систем та розробки програмного забезпечення, які знаходять широке застосування в автоматизації управління інфраструктурою. SystemD як сучасна система ініціалізації та управління сервісами Linux замінила традиційні init системи, надаючи потужні можливості для контролю життєвого циклу процесів, планування задач та моніторингу стану системи. Git як розподілена система контролю версій зробила революцію у підході до управління кодом та конфігураціями, забезпечуючи надійне версіонування, колаборацію між розробниками та цілісність даних. Розуміння цих технологій є необхідним для побудови сучасних автоматизованих систем управління інфраструктурою.

SystemD є системою ініціалізації та менеджером сервісів для операційних систем Linux, розробленою з метою заміни традиційної System V init системи [23]. Проект було розпочато у 2010 році Леннартом Поттерінгом та Кеєм Зіверсом з метою створення більш ефективної, паралелізованої та функціонально багатої системи управління процесами. SystemD швидко набула поширення та станом на 2025 рік є стандартом де-факто для більшості популярних дистрибутивів Linux, включаючи Red Hat Enterprise Linux, CentOS, Fedora, Ubuntu, Debian, SUSE та їх похідні. Попри деякі суперечки в спільноті щодо філософії її дизайну, SystemD забезпечує суттєві переваги в продуктивності та функціональності.

SystemD units є базовими об'єктами конфігурації, які описують ресурси, керовані SystemD. Service units (файли з розширенням `.service`) визначають демони (програми, які працюють у фоновому режимі, без прямої взаємодії з користувачем) та процеси, їх параметри запуску, залежності від інших сервісів, політики перезапуску при збоях. Timer units (`.timer`) реалізують планування

періодичного виконання, прив'язуючись до відповідного service unit. Target units групують інші units для досягнення певного системного стану (multi-user.target, graphical.target). Journald є централізованою системою логування SystemD, яка збирає та зберігає логи від усіх компонентів системи. На відміну від традиційного syslog, який оперує текстовими файлами, journald використовує бінарний формат зберігання з індексацією за множиною полів: timestamp, priority, service unit, process ID, користувач, hostname. Це забезпечує ефективні запити та фільтрацію логів через команду journalctl. Структурований логінг дозволяє додавати довільні метадані які можуть використовуватися для розширеного аналізу. Автоматична ротація логів базується на розмірі та часі зберігання, запобігаючи переповненню дискового простору.

Переваги SystemD timers над традиційним cron є суттєвим. Централізоване логування через journald автоматично зберігає весь output сервісу з timestamp та метаданими, доступний через `journalctl -u service`. Явні залежності дозволяють гарантувати, що мережа доступна перед запуском сервісу через `After=network-online.target`. Інтеграція з systemctl надає уніфікований інтерфейс для управління: `systemctl status timer` показує стан та час наступного виконання, `systemctl list-timers` відображає всі активні timers з часом до наступного запуску. Resource limits через cgroups обмежують споживання CPU та пам'яті сервісу, запобігаючи негативному впливу на основні додатки.

Git є розподіленою системою контролю версій, створеною Лінусом Торвальдсом у 2005 році для управління розробкою ядра Linux [24]. На відміну від централізованих систем контролю версій (CVS, Subversion), де існує єдиний центральний сервер з повною історією проєкту, Git надає кожному розробнику повну копію репозиторію з усією історією змін. Це забезпечує можливість роботи offline, швидкість операцій (оскільки більшість виконується локально без мережевих викликів), відмовостійкість (втрата центрального сервера не призводить до втрати історії) та гнучкість у виборі workflow. Git зробив революцію в індустрії розробки програмного забезпечення, ставши стандартом для версіонування коду та надихнувши створення платформ для колаборації, такі як GitHub, GitLab та

Bitbucket. Розподілена природа Git передбачає наявність локального та віддалених (remote) репозиторіїв. Команда `git clone` створює локальну копію віддаленого репозиторію з усією історією. `git fetch` завантажує нові commits з remote без інтеграції їх у локальні гілки. `git pull` комбінує fetch та merge, інтегруючи віддалені зміни у поточну гілку. `git push` відправляє локальні commits до віддаленого репозиторію. Така модель дозволяє розробникам працювати автономно, створюючи локальні commits offline, та синхронізуватися з командою періодично через push/pull операції. Multiple remotes підтримка дозволяє взаємодіяти з декількома віддаленими репозиторіями, що корисно для fork-based workflows або синхронізації між різними Git серверами.

Висновок до розділу 1

У першому розділі роботи розглянуто теоретичні основи управління конфігураціями та технологічний стек інструментів для побудови самоналаштовуваних систем на базі pull-моделі автоматизації. Проаналізовано фундаментальні концепції Infrastructure as Code, архітектурні особливості систем управління конфігураціями та специфічні технології, які формують основу для реалізації автономних самоконфігурувальних віртуальних машин.

Технологічний стек формує цілісну екосистему, де кожен компонент виконує специфічну роль у життєвому циклі віртуальної машини. Terraform забезпечує декларативне управління хмарними ресурсами через версіоновані конфігураційні файли. Packer автоматизує створення стандартизованих golden images з попередньо встановленим базовим програмним забезпеченням.

Теоретичні основи та технологічний огляд, представлені в цьому розділі, формують необхідний фундамент для розуміння проектування та реалізації системи самоналаштування віртуальних машин. Наступний розділ присвячено детальному проектуванню архітектури такої системи з розглядом компонентної діаграми, механізмів безпечної автентифікації, організації конфігураційних репозиторіїв та практичної реалізації спроектованої системи з конкретними прикладами конфігурацій, скриптів та інтеграції всіх компонентів технологічного стеку.

РОЗДІЛ 2.
ПРОЄКТУВАННЯ СИСТЕМИ САМОНАЛАШТУВАННЯ

У першому розділі було розглянуто теоретичні основи Infrastructure as Code, архітектуру систем управління конфігураціями на базі Ansible та компоненти технологічного стеку, необхідні для побудови автономних самоналаштовуваних систем. Було показано, що pull-модель автоматизації природно вирішує проблеми масштабованості, відмовостійкості та безпеки, притаманні централізованому push-підходу в динамічних хмарних середовищах. Однак перехід від теоретичних концепцій до практичної реалізації вимагає детального проєктування архітектури системи, яка інтегрує всі розглянуті технології у цілісне рішення. Цей розділ присвячено проєктуванню системи самоналаштування віртуальних машин на базі Ansible Pull у хмарному середовищі AWS. Основна мета полягає у створенні архітектурного рішення, яке забезпечує автоматичне розгортання, конфігурування та підтримку актуального стану серверів без централізованого управління. Система повинна гарантувати автономність віртуальних машин у прийнятті рішень про оновлення власних конфігурацій, безпечну автентифікацію без зберігання статичних секретів, автоматичне усунення конфігураційних відхилень та можливість відновлення після збоїв.

2.1. Аналіз вимог та постановка задачі

2.1.1. Проблеми централізованої push-моделі управління

Централізована push-модель управління конфігураціями, яка використовується у традиційних системах автоматизації, характеризується наявністю центрального control node, що ініціює підключення до керованих вузлів та виконує на них конфігураційні зміни. Цей архітектурний підхід демонструє критичні обмеження при масштабуванні та експлуатації у динамічних хмарних середовищах.

Control node становить єдину точку відмови для всієї системи управління конфігураціями. Відмова центрального сервера унеможливлює виконання будь-яких конфігураційних змін на всіх керованих вузлах. Зростання кількості керованих

вузлів створює пропорційне навантаження на control node, оскільки всі підключення та виконання playbooks відбуваються з центрального сервера. При досягненні сотень або тисяч серверів виникає необхідність горизонтального масштабування через впровадження декількох control nodes з балансуванням навантаження, що значно ускладнює архітектуру. У push-моделі керовані вузли пасивно очікують на ініціювання конфігураційних змін з боку control node. Configuration drift, що виникає через ручні втручання адміністраторів або нештатні зміни, усувається лише під час наступного запуску playbooks з центрального сервера. Між плановими виконаннями автоматизації сервери можуть перебувати у невідповідному стані з непередбачуваними наслідками для функціонування додатків.

Зазначені обмеження централізованої push-моделі створюють необхідність проектування альтернативного архітектурного підходу, який усуває єдину точку відмови, забезпечує природне масштабування через розподілене виконання, спрощує мережеву топологію та надає керованим вузлам автономність у підтримці актуального стану конфігурацій.

2.1.2. Функціональні та нефункціональні вимоги до системи

На основі аналізу обмежень централізованої push-моделі формулюються вимоги до проєктованої системи самоналаштування, які поділяються на функціональні вимоги, що визначають поведінку системи, та нефункціональні вимоги, що визначають якісні характеристики архітектури. Система повинна забезпечувати автоматичне виконання конфігураційних змін на віртуальних машинах на регулярній основі без зовнішнього ініціювання. Кожен керований вузол має самостійно отримувати актуальні конфігурації з централізованих репозиторіїв та застосовувати їх згідно з визначеним розкладом. Система повинна підтримувати динамічну автентифікацію до систем управління секретами на основі ідентичності віртуальної машини без зберігання статичних credentials. Необхідно забезпечити підтримку різноманітних типів серверів (бази даних, сервери додатків, балансувальники навантаження) з можливістю застосування специфічних наборів конфігурацій відповідно до призначення вузла. Система має підтримувати автоматичне оновлення власних компонентів без перезбудови базових образів

віртуальних машин. Повинна забезпечуватися валідація конфігурацій перед застосуванням з детальним логуванням результатів виконання для подальшого аудиту та діагностики. Кожна віртуальна машина повинна функціонувати як незалежна одиниця, здатна приймати рішення про оновлення власних конфігурацій без залежності від доступності централізованих control nodes. сі конфігураційні операції повинні бути ідемпотентними, тобто багаторазове виконання має призводити до того самого результату без непередбачуваних побічних ефектів. Система повинна мінімізувати зберігання конфіденційних даних на віртуальних машинах через використання короткотермінових динамічно згенерованих credentials. рхітектура системи повинна забезпечувати горизонтальне масштабування без деградації продуктивності при зростанні кількості керованих вузлів від десятків до тисяч віртуальних машин. ас від створення нової віртуальної машини до повного конфігурування та готовності до експлуатації повинен бути мінімальним. Система має забезпечувати автоматичне виконання всіх необхідних конфігураційних кроків одразу після першого завантаження інстансу без очікування на планові інтервали виконання.

Сформульовані функціональні та нефункціональні вимоги формують основу для проєктування архітектури системи самоналаштування, яка реалізує pull-модель управління конфігураціями з усуненням обмежень централізованого підходу. Наступні підрозділи деталізують архітектурні компоненти та механізми, що забезпечують виконання зазначених вимог.

2.2. Компонентна діаграма системи.

Архітектура системи самоналаштування представлена компонентною діаграмою (Рис. 2.1), яка відображає основні компоненти інфраструктури, їх взаємозв'язки та потоки даних. Система побудована на принципі розподіленого виконання, де кожна віртуальна машина функціонує як автономна одиниця, здатна самостійно отримувати та застосовувати конфігурації. Фундаментом архітектури є набір Git репозиторіїв, що зберігають версіоновані конфігураційні артефакти різних рівнів інфраструктури. Репозиторій `terraform-modules.git` містить модулі

Terraform для провізіонінгу інфраструктури, `terraform.git` зберігає конфігурації для створення конкретних середовищ, `packer.git` містить визначення для побудови golden images. Репозиторій `ansible.git` зберігає playbooks, ролі та глобальні конфігурації для управління серверами, а `ansible-inventory.git` містить динамічні inventory плагіни та ієрархію змінних для різних середовищ і груп хостів. Така модульна організація забезпечує можливість незалежного версіонування різних рівнів конфігурацій та гнучкість в управлінні змінами.

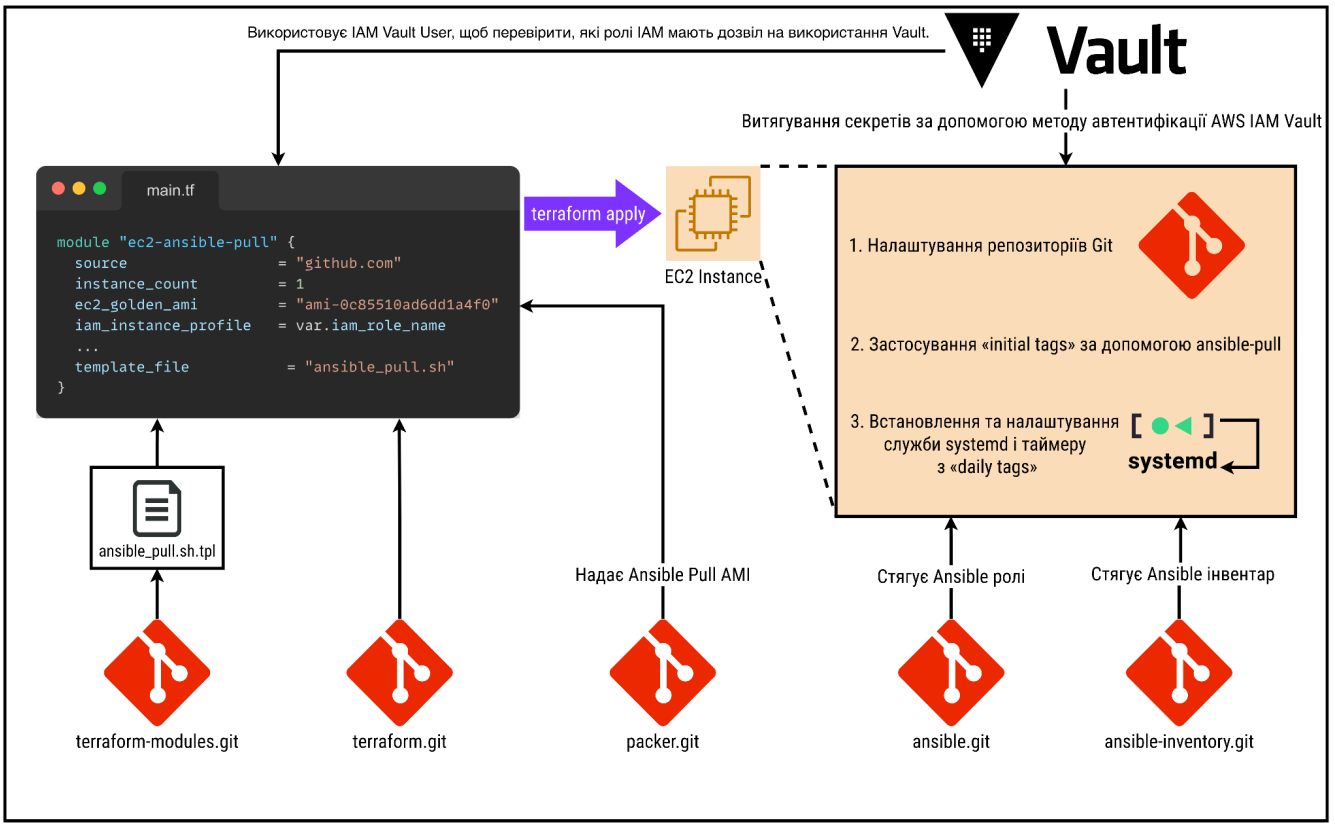

Рисунок 2.1 - Компонентна діаграма архітектури системи самоналаштування

Взаємодія компонентів у повному життєвому циклі формує два основні потоки виконання. Під час провізіонінгу нової віртуальної машини Terraform створює EC2 інстанс на базі Packer образу з IAM Instance Profile та Cloud-init скриптом, який при першому завантаженні автентифікується до Vault, отримує Git credentials, клонує репозиторії та виконує ansible-pull з тегами `initial` для встановлення ролі `ansible_pull`, що налаштовує SystemD timer для подальшого періодичного

виконання. Під час регулярного виконання SystemD timer запускає `run-ansible-pull.sh`, який повторює цикл автентифікації до Vault, оновлення репозиторіїв та застосування конфігурацій з тегами `daily`, забезпечуючи постійну актуальність стану віртуальних машин та автоматичне усунення configuration drift. Наступні підрозділи деталізують окремі аспекти архітектури: механізм безпечної автентифікації, структуру Ansible конфігурацій та процеси самовідновлення системи.

2.3. Механізм безпечної автентифікації

2.3.1. AWS IAM автентифікація до HashiCorp Vault

Механізм автентифікації AWS IAM Auth у системі HashiCorp Vault забезпечує безпечну ідентифікацію та авторизацію ресурсів AWS без необхідності зберігання статичних облікових даних. Принцип роботи базується на взаємодії між клієнтом, що ініціює автентифікацію, та Vault-сервером через перевірку підписаного запиту до AWS STS. На початковому етапі клієнт формує, але не надсилає, запит до методу GetCallerIdentity сервісу STS. Цей метод повертає інформацію про того, хто здійснює запит, тобто визначає ідентичність. Клієнт підписує сформований запит своїми тимчасовими або постійними обліковими даними AWS, після чого надсилає цей підписаний запит як частину запиту на автентифікацію до Vault. Vault перевіряє цю інформацію відповідно до заздалегідь створених Vault Roles, що пов’язують дозволені ARN-ідентифікатори з політиками доступу у Vault. Якщо отриманий ARN збігається з дозволеним, автентифікація завершується успішно. На Рис. 2.2 показано діаграму автентифікації за допомогою AWS IAM Auth.

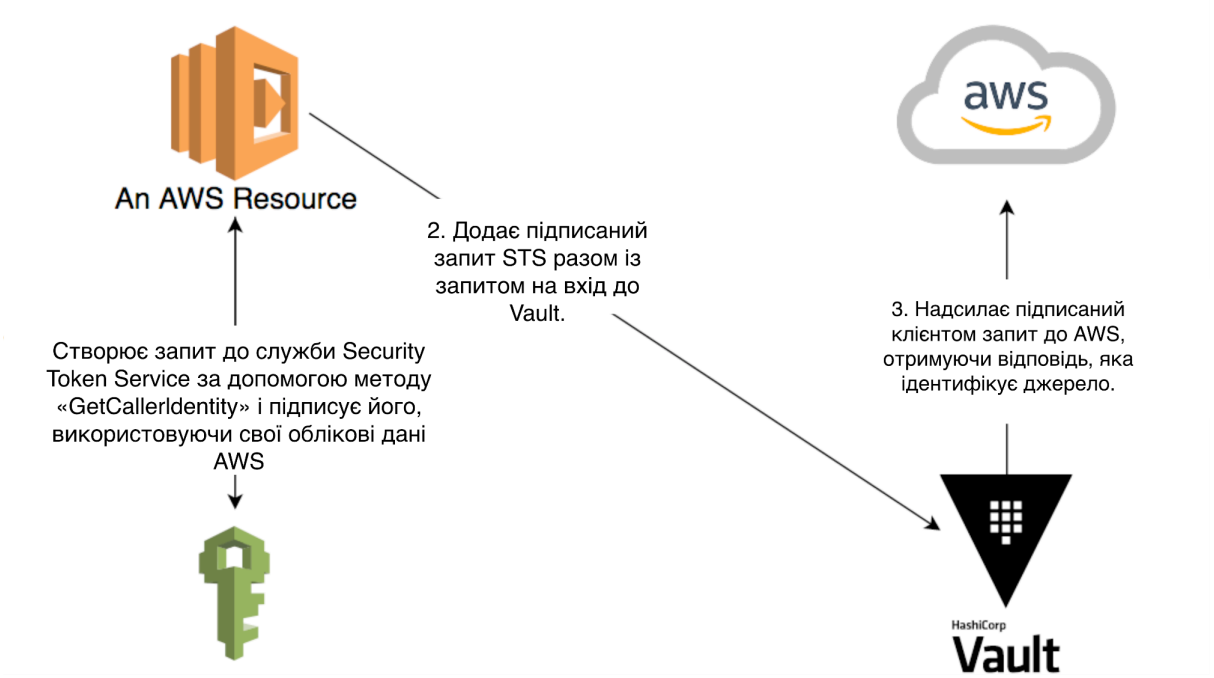

Рисунок 2.2 - Діаграма автентифікації за допомогою AWS IAM Auth [25].

Для коректного функціонування цього механізму Vault-кластер повинен мати відповідні дозволи у своїх AWS-політиках. Зокрема, необхідно надати права на виконання таких дій:

- **iam:GetRole** або **iam:GetUser** — для отримання метаданих про IAM-принципала;
- **sts:GetCallerIdentity** — для перевірки автентичності запиту.

Таким чином, AWS IAM Auth у Vault реалізує надійний механізм безпарольної автентифікації шляхом делегування процесу ідентифікації до AWS STS, що забезпечує високий рівень безпеки та централізоване управління доступом до секретів і конфіденційних даних у хмарному середовищі.

2.3.2. Управління секретами та політики доступу у Vault

Після успішної автентифікації до Vault через AWS IAM метод EC2 інстанс отримує короткотерміновий токен, який надає доступ до секретів відповідно до асоційованих політик. Архітектура управління секретами у проєктованій системі реалізує дворівневий підхід, де IAM автентифікація використовується для отримання Ansible Role credentials, які в свою чергу надають доступ до конфігураційних

секретів та Git credentials. Такий підхід забезпечує гнучкість у управлінні правами доступу та дозволяє централізовано контролювати ротацію AppRole секретів без необхідності оновлення IAM ролей.

Vault політики визначають гранульовані права доступу до різних шляхів у key-value backend, де зберігаються секрети різних типів та призначень. Політика, асоційована з IAM роллю через Vault AWS auth, надає доступ до специфічного шляху для читання Ansible Role credentials: `{{env_class}}/{{env_name}}//ansible-pull`. За цим шляхом зберігаються `env_vault_role_id` та `env_vault_secret_id` для доступу до конфігураційних секретів середовища, а також Git credentials (`git_username`, `git_password`, `git_hostname`) для клонування приватних репозиторіїв з Ansible конфігураціями. Отримані Ansible Role credentials мають більш широкі права доступу для читання конфігураційних секретів різних компонентів ізанианфраструктури відповідно до типу сервера.

Початкова політика, прикріплена до AWS IAM Auth методу є обмежена до невеликої кількості секретів. На Рис. 2.3 показана сама політика доступу.

```
path "dev/test/ansible-pull"
{
     capabilities = ["read", "list"]
}
```

Рисунок 2.3 - Початкова політика доступу ansible-pull

На Рис. 2.4 показані приклади секретів, які знаходяться по шляху `dev/test/ansible-pull.`

```
{
  "env_vault_role_id": "***",
  "env_vault_secret_id": "***",
  "git_hostname": "local-gitlab.test.com ",
  "git_password": "***",
  "git_username": "ansible_pull",
}
```

Рисунок 2.4 - Приклад секретів, які знаходяться по шляху `dev/test/ansible-pull`

Після отримання AppRole credentials віртуальна машина виконує повторну автентифікацію до Vault через AppRole auth method, отримуючи токен з ширшими правами для читання конфігураційних секретів різних компонентів `env_vault_role_id` та `env_vault_secret_id`. Політки, яка прикріплена до цього AppRole auth методу показана на Рис. 2.5.

```
path "dev/*"
{
     capabilities = ["create", "read", "update", "delete", "list"]
}
```

Рисунок 2.5 - Загальна політика доступу до секретів середовища

2.4. Структура Ansible конфігурацій

2.4.1. Організація репозиторіїв (ansible, ansible-inventory) та розподіл обов'язків.

Архітектура проєктованої системи самоналаштування базується на розділенні Ansible конфігурацій між двома Git репозиторіями для забезпечення модульності та незалежного версіонування. Репозиторій ansible містить виконуваний код автоматизації у вигляді playbooks та ролей, тоді як репозиторій ansible-inventory зберігає метадані про інфраструктуру та конфігураційні змінні. Така організація дозволяє незалежно оновлювати логіку конфігурування без змін у визначеннях інфраструктури та навпаки.

Репозиторій ansible структуровано навколо центрального playbook main.yaml, який імпортує vars.yaml для ієрархії змінних та vault_auth.yaml для автентифікації до HashiCorp Vault. Playbook складається з послідовності plays, де кожен play застосовує специфічні ролі до відповідних груп хостів. Фінальний play орієнтований на групу all і застосовує роль ansible_pull до всіх серверів інфраструктури,

забезпечуючи встановлення механізму періодичного самооновлення на кожній віртуальній машині. На Рис. 2.6 показано вміст main.yaml.

```yaml
---
- import_playbook: vars.yaml
- import_playbook: vault_auth.yaml

- hosts: web
  gather_facts: false
  become: yes
  roles:
    - role: role1
    - role: role2
    - role: tomcat

- hosts: all
  become: true
  gather_facts: false
  roles:
    - role: ansible_pull
```

Рисунок 2.6 - Вміст файлу main.yaml

Директорія `roles` містить модульні компоненти конфігурацій, де кожна роль інкапсулює логіку налаштування специфічного сервісу. Роль ansible_pull відповідає за підготовку середовища для регулярного виконання ansible-pull, включаючи встановлення залежностей, налаштування Git репозиторіїв, конфігурацію SystemD сервісу та таймера, експорт змінних середовища для гранульованого виконання через систему тегів. Інші ролі реалізують конфігурацію конкретних компонентів інфраструктури залежно від потреб користувача.

Репозиторій `ansible-inventory` організовано за принципом ієрархічного зберігання конфігураційних даних. Директорія `shared_resources/global_group_vars` містить глобальні змінні для кожного типу серверів, які застосовуються незалежно від середовища. Директорія `shared_resources/groups_hierarchy` визначає ієрархію груп хостів з наслідуванням змінних, що дозволяє уникнути дублювання конфігурацій. Директорії специфічних середовищ містять aws_ec2.yaml для динамічного inventory плагіна, `common_vars.yaml` зі змінними спільними для середовища та `group_vars` зі

змінними специфічними для кожного типу серверів. На Рис. 2.7 показано приклад вмісту groups.yaml.

```
---
all:
  children:
    webservers:
      children:
        nginx_servers: {}
        apache_servers: {}
    databases:
      children:
        mysql_primary: {}
        mysql_secondary: {}
```

Рисунок 2.7 - Приклад вмісту файлу groups.yaml

Ієрархія змінних у проєктованій системі реалізує принцип від загального до специфічного, де змінні більш специфічних рівнів перевизначають значення глобальних змінних. Ansible застосовує змінні у порядку зростання пріоритету, але потрібна додаткова логіка, яка б підійшла під дану архітектуру. Вміст файлу vars.yaml показаний на Рис. 2.8.

```
---
- hosts: all
  become: false
  gather_facts: false
  pre_tasks:
    - name: Include Global variables
      include_vars:
        file: "inventory/global_vars.yaml"
      no_log: "{{ ansible_verbosity < 2 }}"

    - name: Include Class variables
      include_vars:
        file: "inventory/{{ env_class }}/common_vars.yaml"
      no_log: "{{ ansible_verbosity < 2 }}"

    - name: Include Evironment variables
      include_vars:
        file: "inventory/{{ env_class }}/{{ env_name }}/group_vars/all.yaml"
      no_log: "{{ ansible_verbosity < 2 }}"

  tags: always
```

Рисунок 2.8 - Вміст файлу vars.yaml

2.4.2. Дизайн ролі ansible pull.

Роль `ansible_pull` займає центральне місце у проєктованій системі самоналаштування, оскільки вона відповідає за підготовку середовища на віртуальній машині для автономного виконання конфігураційних змін без централізованого управління. Основне призначення ролі полягає у встановленні та налаштуванні інфраструктури, необхідної для періодичного виконання ansible-pull, включаючи залежності, Git репозиторії, змінні середовища та SystemD таймер для регулярного запуску. Роль проєктується для виконання у двох режимах: початкове встановлення при першому завантаженні віртуальної машини через Cloud-init та щоденне оновлення для підтримки актуальності середовища та застосування конфігураційних змін.

Архітектура ролі побудована на модульному підході з розділенням відповідальності між окремими tasks (файлами завдань), що забезпечує гранульований контроль виконання через систему Ansible тегів. Головний файл `tasks/main.yaml` виступає як оркестратор, який послідовно викликає спеціалізовані task файли з відповідними тегами для диференціації поведінки між початковим встановленням та щоденним виконанням. Використання тегу `never` у комбінації з специфічними тегами (`ansible_pull_initial_install`, `ansible_pull_daily_install`) забезпечує явний контроль виконання, де tasks не запускаються автоматично, а лише при явному вказуванні відповідних тегів під час виклику ansible-pull. Така архітектура дозволяє Cloud-init скрипту викликати ansible-pull з тегами `initial` для початкового налаштування, тоді як SystemD timer використовує теги `daily` для регулярних оновлень конфігурацій. Воний вміст main.yaml ansible_pull ролі показаний на Додатку Г.

Перший етап виконання ролі включає попередні перевірки валідності середовища через `tasks/checks.yaml`, де верифікується наявність критичних змінних конфігурації, другий етап забезпечує встановлення та оновлення залежностей через `tasks/install.yaml`, включаючи системні пакети через DNF

package manager, третій етап виконує оновлення Git репозиторіїв через окремі task файли `git_refresh_ansible_inventory.yaml` та `git_refresh_ansible.yaml`, які клонують або оновлюють відповідні репозиторії у визначених директоріях з використанням Git credentials, отриманих з Vault, Четвертий етап генерує файл змінних середовища `.env_var` через tasks `export_env_initial.yaml` або `export_env_daily.yaml` залежно від тегів виконання, п'ятий етап налаштовує SystemD сервіс та таймер через `tasks/install_systemd_service.yaml` для автоматизації періодичного виконання ansible-pull.

Дизайн ролі `ansible_pull` реалізує принцип самодостатності, де після успішного виконання початкового встановлення віртуальна машина отримує всі необхідні компоненти для автономного функціонування без залежності від централізованих control nodes. Модульна архітектура з гранульованими тегами забезпечує гнучкість у виборі компонентів для виконання, дозволяючи селективно застосовувати оновлення залежностей, Git репозиторіїв або SystemD конфігурацій без повного перевиконання всієї ролі. Ідемпотентність всіх tasks гарантує безпечність повторного виконання ролі на регулярній основі без ризику створення дублікатів або конфліктів конфігурацій.

2.4.3. Динамічний inventory на базі AWS EC2.

Динамічний inventory на основі плагіна `amazon.aws.aws_ec2` усуває необхідність ручної підтримки статичних списків хостів та забезпечує автоматичне виявлення віртуальних машин у хмарному середовищі. На відміну від традиційного статичного inventory, динамічний підхід запитує AWS EC2 API під час кожного виконання ansible-pull для отримання актуального списку інстансів з їх характеристиками та тегами. Така автоматизація критична для pull-моделі, де новостворені віртуальні машини повинні автоматично інтегруватися у процес управління конфігураціями без ручної реєстрації.

Конфігурація плагіна `aws_ec2` визначається у YAML файлі `aws_ec2.yaml` у кореневій директорії inventory конкретного середовища. Параметр `plugin:`

`amazon.aws.aws_ec2` ідентифікує використання динамічного плагіна з колекції amazon.aws, який інтегрується з Ansible inventory системою. Параметр `regions` визначає AWS регіони для сканування інстансів, обмежуючи пошук специфічними географічними локаціями для оптимізації швидкості запитів. Типова конфігурація показана на Рис. 2.9.

```
plugin: amazon.aws.aws_ec2

regions:
  - us-west-2

exclude_filters:
  - tag:ansible-enabled:
      - 'false'

filters:
  instance-state-name:
    - running
  tag:Environment:
    - dev.test

hostnames:
  - tag:Name

keyed_groups:
  - key: tags['ansible-group']
    separator: ''
  - key: tags['ansible-group-extra-01']
    separator: ''
  - key: tags['ansible-group-extra-02']
    separator: ''
  - key: architecture
    prefix: arch

vars:
  ansible_user: ec2-user

cache: true
cache_plugin: memory
cache_timeout: 3600
```

Рисунок 2.9 - Вміст файлу aws_ec2.yaml

Секція `filters` визначає критерії відбору EC2 інстансів для включення у inventory. Фільтр `instance-state-name: running` обмежує результати активними

віртуальними машинами, виключаючи зупинені інстанси. Фільтрація за тегом сегментує inventory за середовищами, забезпечуючи отримання лише релевантних хостів без змішування з інстансами інших середовищ. Параметр `exclude_filters` додатково виключає інстанси з тегом `ansible-enabled: false`, надаючи механізм для тимчасового виведення серверів з управління конфігураціями.

Секція keyed_groups реалізує динамічне групування інстансів на основі їх тегів для застосування специфічних конфігурацій до різних типів серверів. Конфігурація `key: tags['ansible-group']` створює групи inventory на основі значення тегу `ansible-group`, де інстанс з тегом `ansible-group: database` автоматично додається до групи `database`. Додаткові теги `ansible-group-extra-01` та `ansible-group-extra-02` дозволяють віртуальній машині належати до декількох груп одночасно для застосування множинних ролей. Terraform забезпечує встановлення необхідних тегів під час провізіонінгу EC2 інстансів через параметри модуля створення віртуальних машин. IAM Instance Profile повинен включати політику з правами `ec2:DescribeInstances` та `ec2:DescribeTags` для дозволу плагіну `aws_ec2` запитувати інформацію про інстанси, що автоматично налаштовується Terraform під час створення IAM ролі. Комбінація динамічного inventory зі статичними групами з файлу groups.yaml дозволяє створювати ієрархічні структури, де динамічно виявлені інстанси автоматично включаються у відповідні батьківські групи.

2.4.4. Система тегів для гранульованого виконання.

Система тегів у проєктованій архітектурі реалізує механізм гранульованого контролю виконання конфігураційних tasks, дозволяючи різним типам серверів застосовувати специфічні набори налаштувань відповідно до їх ролі в інфраструктурі. На відміну від монолітного підходу, де всі tasks виконуються на всіх серверах незалежно від типу, система тегів забезпечує селективне виконання лише релевантних конфігурацій для кожної віртуальної машини. Це критично важливо у середовищах з гетерогенною інфраструктурою, де сервери баз даних, додатків,

балансувальники навантаження та сервіси моніторингу вимагають принципово різних конфігурацій та залежностей.

Ansible теги прив'язуються до tasks, ролей або plays у playbook конфігураціях, дозволяючи виконувати лише ті компоненти, які явно вказані через параметр `--tags` при запуску ansible-pull. Роль `ansible_pull` експортує змінні `ANSIBLE_INITIAL_TAGS` та `ANSIBLE_DAILY_TAGS` у файл `.env_vars`, які потім використовуються скриптом `run-ansible-pull.sh` для визначення набору тегів при виконанні. Початкові теги (`env_initial_tags`) включають встановлення програмного забезпечення та базову конфігурацію компонентів при першому завантаженні віртуальної машини, тоді як щоденні теги (`env_daily_tags`) орієнтовані на оновлення конфігураційних файлів, перевірку стану сервісів та застосування змін налаштувань без повного переінсталювання компонентів.

Сервери групи `database` (бази даних) мають змінну `env_initial_tags` зі значенням, що включає базові теги `env_common_initial_tags` (спільні для всіх серверів), додаючи специфічні теги `java_install` для встановлення Java Runtime Environment та `mariadb_install` для встановлення та конфігурації MariaDB сервера. Вміст файлу databases.yaml показаний на Рис. 2.10.

```
---
env_initial_tags: "{{ env_common_initial_tags }},java_install,mariadb_install"
```

Рисунок 2.10 - Вміст файлу databases.yaml у директорії group_vars.

Приклад вмісту `env_common_initial_tags` показаний на Рис. 2.11

```
---
env_common_initial_tags: "base_install,..."
```

Рисунок 2.11 - Приклад вмісту змінної `env_common_initial_tags`.

Використання змінної з базовими тегами через механізм інтерполяції Jinja2 (`{{ env_common_initial_tags }}`) у group_vars файлах забезпечує

консистентність спільних конфігурацій без дублювання визначень для кожної групи серверів.

2.5. SystemD, автооновлення та механізми валідації/обробки помилок

SystemD сервіс та таймер формують технологічну основу для автоматичного періодичного виконання ansible-pull, забезпечуючи регулярну синхронізацію стану віртуальної машини з бажаними конфігураціями без ручного втручання або залежності від централізованих планувальників. На відміну від традиційного підходу з cron jobs, SystemD таймери надають розширені можливості для керування розкладом виконання, інтеграцію з системою логування через journald, автоматичне відстежування статусу виконання та гранульований контроль ресурсів через cgroups. Використання SystemD як нативного init system забезпечує надійність механізму автоматизації з можливістю автоматичного перезапуску при збоях та централізованого моніторингу стану сервісів.

Механізм автоматичного оновлення скрипта run-ansible-pull.sh через порівняння MD5 хешів забезпечує актуальність виконуваного коду без ручного втручання. Скрипт реалізує самоперевірку власної версії, автоматично виявляючи зміни у віддаленому Git репозиторії та перезапускаючись з оновленим кодом під час виконання.

Алгоритм автооновлення обчислює MD5 хеш поточного файла через команду `md5sum "$0"` на початку роботи, зберігаючи результат у змінній `SCRIPT_ORIG_MD5`. Після оновлення Git репозиторію ansible через виклик ansible-playbook з тегами `ansible_pull_refresh_git_ansible` скрипт повторно обчислює хеш у змінній `SCRIPT_NEW_MD5`. Оскільки файл run-ansible-pull.sh зберігається у репозиторії ansible у директорії `roles/ansible_pull/files/`, оновлення репозиторію автоматично замінює файл скрипта новою версією. При виявленні розбіжності хешів скрипт експортує змінну `SCRIPT_RESTART=1` для запобігання нескінченному циклу та викликає сам себе через `"${0}"`. На Рис. 2.12 показано код, відповідальний за самооновлення скрипту.

```
SCRIPT_ORIG_MD5=$(md5sum "$0")
SCRIPT_RESTART=${SCRIPT_RESTART:-""}

...

SCRIPT_NEW_MD5=$(md5sum "$0")
if [[ -z $SCRIPT_RESTART && "$SCRIPT_ORIG_MD5" != "$SCRIPT_NEW_MD5" ]]; then
  echo "Script changes detected, restarting"
  export SCRIPT_RESTART=1
  "${0}"; exit
fi

...
```

Рисунок 2.12 - Частина коду, який відповідає за самооновлення скрипту run-ansible-pull.sh.

Використання MD5 хешу забезпечує криптографічну гарантію виявлення будь-яких змін у вмісті файла, оскільки навіть зміна одного байта призводить до повністю іншого хешу. Хоча MD5 вважається криптографічно слабким для захисту, для виявлення змін у файлах він достатній та ефективний з погляду продуктивності.

Механізм попередніх перевірок забезпечує раннє виявлення проблем конфігурації перед виконанням основних tasks, запобігаючи застосуванню часткових або некоректних налаштувань. Роль ansible_pull реалізує валідацію через task файл checks.yaml з використанням модуля ansible.builtin.assert для перевірки наявності та коректності критичних змінних. Кожна перевірка специфікує умови валідації через директиву that, де змінна повинна бути визначена (is defined), мати відповідний тип даних (is string) та непорожнє значення (length > 0). При провалі будь-якої перевірки Ansible зупиняє виконання з детальним повідомленням про причину збою. На Рис. 2.13 показано приклад такої валідації.

```
---
- name: Check if ansible_pull_env_name is set correctly
  ansible.builtin.assert:
    that:
      - ansible_pull_git_hostname is defined
      - ansible_pull_git_hostname is string
      - ansible_pull_git_hostname | length > 0
    quiet: true
  tags:
    - ansible_pull_check
    - ansible_pull_export_initial_tags
    - ansible_pull_export_daily_tags
    - ansible_pull_initial_install
    - ansible_pull_daily_install
...
```

Рисунок 2.13 - Одна з багатьох валідацій змінних у ролі ansible_pull.

Скрипт `run-ansible-pull.sh` реалізує обробку помилок через bash директиву `set -euo pipefail`, яка забезпечує автоматичне завершення виконання при помилці команди (опція `-e`), використанні невизначених змінних (опція `-u`) або помилці у будь-якому елементі pipeline (опція `-o pipefail`). Комбінація попередніх перевірок Ansible з bash error handling створює багаторівневий механізм обробки помилок, де проблеми з конфігурацією виявляються на етапі валідації, помилки виконання команд негайно зупиняють процес, а всі збої логуються через journald для діагностики. SystemD автоматично відстежує стан виконання сервісу, дозволяючи зовнішнім системам моніторингу генерувати алерти при послідовних збоях ansible-pull, вказуючи на проблеми з доступністю Vault, Git репозиторіїв або некоректністю конфігураційних змінних у inventory.

Висновки до розділу 2

У другому розділі здійснено проєктування архітектури системи самоналаштування віртуальних машин на базі Ansible Pull з інтеграцією технологічного стека, розглянутого у першому розділі. Розроблено архітектурну специфікацію, яка трансформує концепції Infrastructure as Code та pull-моделі управління конфігураціями у практичне рішення для хмарного середовища AWS.

Аналіз push-моделі виявив ключові обмеження: єдину точку відмови control node, низьку масштабованість, складну мережеву топологію, відсутність самовідновлення та ризики при централізованому зберіганні credentials. Це зумовило перехід до децентралізованої pull-архітектури з функціональними вимогами до автоматичного оновлення конфігурацій, динамічної автентифікації, підтримки різних типів серверів та високої безпеки.

Розроблено компонентну діаграму системи, де Packer створює стандартизовані образи з попередньо встановленими інструментами, Terraform провізіонує EC2 інстанси з IAM ролями та Cloud-init для початкової ініціалізації, HashiCorp Vault забезпечує динамічну автентифікацію через AWS IAM метод, Git репозиторії зберігають конфігураційний код і метадані інфраструктури, а SystemD виконує періодичне оновлення з випадковою затримкою для рівномірного розподілу навантаження. Безпечна автентифікація реалізована на основі інтеграції AWS IAM з Vault, що усуває потребу у статичних секретах. EC2 інстанс проходить перевірку ідентичності через STS API, після чого Vault видає короткотермінові AppRole credentials з гранульованим контролем доступу до секретів відповідно до ролі сервера.

Ansible конфігурації поділені між репозиторіями ansible та ansible-inventory для незалежного версіонування та гнучкого управління доступом. Використання динамічного inventory на базі плагіну amazon.aws.aws_ec2 усуває ручне оновлення списків хостів, автоматично групуючи інстанси за тегами. Механізм автооновлення через SystemD таймер та перевірку MD5-хешів забезпечує постійну актуальність скриптів з детальним логуванням через journald для відстежуваності виконання.

Спроєктована архітектура усуває єдині точки відмови, забезпечує автономність віртуальних машин, безпечну автентифікацію, масштабованість і модульність. Вона формує цілісну екосистему інтегрованих компонентів, закладаючи основу для практичної реалізації, описаної у наступному розділі.

РОЗДІЛ 3.
РЕАЛІЗАЦІЯ СИСТЕМИ САМОНАЛАШТУВАННЯ

3.1. Підготовка інфраструктури

3.1.1. Налаштування HashiCorp Vault

Реалізація системи самоналаштування розпочинається з конфігурації HashiCorp Vault як централізованої системи управління секретами з підтримкою AWS IAM authentication method. Налаштування Vault включає активацію AWS auth backend, конфігурацію інтеграції з AWS IAM, створення ролей з політиками доступу та структурування секретів у key-value storage.

Першим кроком є активація AWS authentication method у Vault через CLI команду `vault auth enable aws`, яка реєструє новий auth backend для обробки запитів автентифікації від EC2 інстансів на основі їх IAM ідентичності. Конфігурація AWS auth backend вимагає надання credentials IAM користувача, створеного спеціально для Vault з правами `iam:GetUser` та `iam:GetRole` для верифікації IAM ролей EC2 інстансів через AWS API. Приклад активації AWS authentication method показано на Рис. 3.1.

```
vault write auth/aws/config/client \
    secret_key=${VAULT_IAM_SECRET_KEY} \
    access_key=${VAULT_IAM_ACCESS_KEY}
```

Рисунок 3.1 - Активація AWS authentication method.

Додатковий рівень безпеки забезпечується через параметр `iam_server_id_header_value`, який вимагає, щоб клієнти включали специфічний header у запити автентифікації. В цьому випадку ми додамо посилання на Vault в параметр. Команду встановлення `iam_server_id_header_value` показано на Рис. 3.2.

```
vault write auth/aws/config/client \
    iam_server_id_header_value=https://vault-dev.test.com
```

Рисунок 3.2 - Команда встановлення `iam_server_id_header_value`.

Створення ролі у AWS auth backend визначає відображення між IAM ролями EC2 інстансів та Vault політиками доступу. Параметр `bound_iam_principal_arn` специфікує список дозволених IAM ролей ARN, де лише інстанси з цими ролями можуть автентифікуватися через дану Vault роль. Параметр `policies` асоціює Vault політики з роллю, визначаючи набір дозволів для токенів, виданих після успішної автентифікації. Створення Vault ролі показано на Рис. 3.3

```
vault write auth/aws/role/test-test-role-iam \
    auth_type=iam \
bound_iam_principal_arn=arn:aws:iam::123456789012:role/general-role,arn:aws:iam::123456789
012:role/ansible-pull-app-role \
    policies=general-policy \
    max_ttl=1h
```

Рисунок 3.3 - Створення Vault ролі.

Тестування конфігурації AWS auth method показано на Рис. 3.4. Воно виконується з EC2 інстансу з асоційованою IAM роллю через Vault CLI

```
export VAULT_ADDR=https://vault-dev.test.com
export VAULT_SKIP_VERIFY=1

vault login -method=aws \
    header_value=https://vault-dev.test.com \
    role=test-test-role-iam

vault read dev/test/ansible-pull
```

Рисунок 3.4 - Тестування конфігурації AWS auth method.

3.1.2. IAM ролі та політики в AWS

Реалізація механізму безпечної автентифікації вимагає створення IAM ролей та політик у AWS для забезпечення ідентичності EC2 інстансів та надання необхідних

дозволів для автентифікації до Vault і виконання динамічного inventory. Terraform використовується для декларативного визначення IAM компонентів, забезпечуючи версіонування та відтворюваність конфігурації безпеки. Для спрощення конфігурацій та зменшення кількості коду у кваліфікаційній роботі будуть використані абстрактні Terraform модулі.

HashiCorp Vault потребує credentials IAM користувача для верифікації автентичності IAM ролей EC2 інстансів через AWS API. Створюється спеціалізований API-only користувач без console доступу з мінімально необхідними правами `iam:GetUser` та `iam:GetRole` для запитів інформації про IAM entities. Terraform policy document визначає дозволені операції. На Рис. 3.5 показано AWS IAM policy для майбутньої AWS IAM ролі.

```
data "aws_iam_policy_document" "iam_vault_policy" {
  statement {
    actions = [
      "iam:GetUser",
      "iam:GetRole"
    ]
    resources = ["*"]
  }
}
```

Рисунок 3.5 - AWS IAM policy для майбутньої AWS IAM ролі.

Модуль створення IAM користувача показаний на Рис. 3.6. Він застосовує політику та генерує access key і secret key, які конфігуруються у Vault AWS auth backend.

```
module "iam_vault_user" {
  source = "git::ssh://git@example.com/terraform-modules.git//aws/iam/iam-api-user"

  user_name            = "${var.env_name}-${var.env_class}-iam-vault-user"
  policy_name          = "${var.env_name}-${var.env_class}-iam-vault-user-policy"
  policy_document_json = data.aws_iam_policy_document.iam_vault_policy.json
  common_tags          = local.common_tags
}
```

Рисунок 3.6 - Модуль створення IAM користувача.

EC2 інстанси потребують дозволу `ec2:DescribeTags` для виконання динамічного inventory плагіна amazon.aws.aws_ec2, який запитує AWS API для отримання списку інстансів та їх тегів. Політика обмежує доступ лише до операцій читання метаданих без можливості модифікації ресурсів. Політика `ec2:DescribeTags` показана на Рис. 3.7.

```
data "aws_iam_policy_document" "ec2_describe_tags_policy" {
  statement {
    sid = "AllowDescribeEC2Tags"
    actions = [
      "ec2:DescribeTags",
      "ec2:DescribeInstances"
    ]
    resources = ["*"]
  }
}
```

Рисунок 3.7 - Політика `ec2:DescribeTags`.

Створення managed policy через Terraform дозволяє прикріплювати її до множини IAM ролей для різних типів серверів. Ресурс для `ec2_describe_tags_policy` policy показаний на Рис. 3.8.

```
resource "aws_iam_policy" "ec2_describe_tags_policy" {
  name        = "gtv-${var.env_name}-${var.env_class}-ec2-tags"
  description = "Allow EC2 instances to describe tags for dynamic inventory"
  policy      = data.aws_iam_policy_document.ec2_describe_tags_policy.json
}
```

Рисунок 3.8 - Ресурс для `ec2_describe_tags_policy`.

Таку policy можна використовувати як і в окремих ролях, так і в загальних. В цьому випадку ми створимо загальну AWS IAM роль, яку зможемо прикріпити до всіх майбутніх EC2. Загальна роль показана на Рис. 3.9.

```
module "general_iam_role" {
  role_name                     = "${var.env_name}-${var.env_class}-general-role"
  policy_arns_count             = "1"
  policy_arns                   = [module.ec2_describe_tags_policy_from_data_source.arn]
  create_instance_profile       = true
  iam_role_policy_document_json = data.aws_iam_policy_document.ec2_assume_role.json
  common_tags                   = local.common_tags
}
```

Рисунок 3.9 - Загальна AWS IAM роль.

Налаштування IAM компонентів завершує підготовку інфраструктури для безпечної автентифікації EC2 інстансів до Vault. ARN створених IAM ролей використовуються у конфігурації Vault AWS auth roles через параметр `bound_iam_principal_arn`, встановлюючи відображення між AWS ідентичністю та Vault політиками доступу. Instance Profiles автоматично надають EC2 інстансам тимчасові credentials для підпису запитів автентифікації до Vault без необхідності зберігання статичних токенів на віртуальних машинах.

3.2. Створення базового образу з Packer

3.2.1. Структура Packer конфігурації.

Створення базового образу операційної системи через Packer реалізується за допомогою HCL конфігураційних файлів, організованих у модульну структуру з чітким розподілом відповідальності між компонентами. Конфігурація Packer для даної системи створює golden AMI (Amazon Machine Image) на основі офіційних образів AlmaLinux OS 9, встановлюючи необхідні залежності для подальшого самоналаштування через ansible-pull. Структура проекту Packer організована у директорії packer/almalinux/ та включає чотири категорії файлів: конфігураційні HCL файли, скрипти provisioning, допоміжні файли залежностей та файли для копіювання на образ.

Основний конфігураційний файл `config.auto.pkr.hcl` містить визначення блоків `packer`, `data`, `source` та `build`, які відповідають за декларацію вимог до інструменту, пошук базових образів, налаштування параметрів збірки та

послідовність provisioning кроків. Повний вміст файлу `config.auto.pkr.hcl` показано на додатку І. Файл змінних `defaults.auto.pkr.hcl` визначає параметризовані значення з валідацією для AWS credentials, регіонів та облікових записів. Файл значень змінних `variables.auto.pkrvars.hcl` містить конкретні значення для змінних, специфічні для організації, такі як списки цільових AWS регіонів та облікових записів для копіювання AMI. Така організація забезпечує відділення параметризації від логіки конфігурації та дозволяє легко адаптувати процес збірки для різних середовищ та організацій.

Конфігурація розпочинається з блоку `packer`, який специфікує вимоги до версії самого Packer та необхідних плагінів для взаємодії з цільовими платформами. Декларація `required_version = "~> 1.8"` забезпечує сумісність конфігурації з версією Packer 1.8.x, використовуючи оператор обмеження версій, який дозволяє minor версії оновлення, але запобігає breaking changes від major версій.

Блок `locals` визначає обчислювані змінні, які використовуються для генерації унікальних назв AMI та управління життєвим циклом образів. Змінна `timestamp` генерує поточну дату та час у форматі `YYYY-MM-DD-hhmm` за допомогою функції `formatdate()`, що забезпечує унікальність та хронологічну впорядкованість назв створюваних образів. Змінна `deprecate_at` обчислює дату застарівання образу через 35040 годин (приблизно 4 роки) від часу створення за допомогою функції `timeadd()`, що дозволяє автоматизувати політики retention та деактивації застарілих образів у майбутньому.

Packer використовує data source `amazon-ami` для динамічного пошуку офіційних образів AlmaLinux OS 9 у AWS Marketplace замість жорсткого оголошення AMI ID, які різняться між регіонами та оновлюються з випуском нових версій дистрибутива. Фільтри `filters` специфікують критерії пошуку: патерн назви `AlmaLinux OS 9.6*` для точкової версії операційної системи, архітектура `x86_64` або `arm64` для різних апаратних платформ, тип кореневого пристрою `ebs` для використання elastic block storage, тип віртуалізації `hvm` для повної апаратної віртуалізації. Параметр `most_recent = true` гарантує вибір найновішого образу з

доступних, що відповідають фільтрам. Аналогічний data source визначається для архітектури arm64, що дозволяє створювати golden AMI для обох типів EC2 інстансів з єдиної конфігурації. Така стратегія забезпечує гнучкість у виборі апаратних платформ без дублювання логіки конфігурації.

Блоки `source` типу `amazon-ebs` визначають параметри створення AMI через тимчасові EC2 інстанси з EBS-backed storage. Параметр `ami_name` використовує локальну змінну `timestamp` для генерації унікальних назв: `Golden AMI AlmaLinux OS 9.6 x86_64 ${local.timestamp}`. Параметр `ami_description` надає людино-читаємий опис призначення образу. Параметр `ami_users` специфікує список AWS облікових записів, з якими спільно використовується створений AMI, дозволяючи іншим підрозділам організації використовувати образ без його копіювання. Параметр `ami_regions` визначає список AWS регіонів, до яких автоматично копіюється створений AMI після успішної збірки, забезпечуючи доступність образу у мультирегіональних розгортаннях.

Блок `build` об'єднує раніше визначені sources та специфікує послідовність provisioners для конфігурування створюваного образу. Параметр `sources` приймає список source блоків для паралельного виконання збірки образів для обох архітектур.

Provisioners виконуються послідовно для кожного source у списку після створення та запуску тимчасового EC2 інстансу. Перший provisioner типу `shell` виконує скрипт `scripts/configuration.sh` з привілеями root через параметр `execute_command = "sudo {{ .Path }}"`. Цей скрипт оновлює системні пакети до найновіших версій за допомогою менеджера пакетів `yum -y update` та вимикає SELinux через модифікацію `/etc/selinux/config`, що спрощує подальше конфігурування через Ansible без конфліктів з політиками безпеки.

Два наступні provisioners типу `file` копіюють локальні файли на тимчасовий інстанс. Перший копіює скрипт `files/motd.sh` до `/tmp/motd.sh`, який генерує динамічний банер входу з інформацією про систему, hostname, uptime, використання ресурсів. Другий копіює всю директорію `files` з Ansible ролі `ansible_pull` до

/tmp/files, що містить списки залежностей: requirements.dnf для системних пакетів, requirements.txt для Python пакетів, requirements.yaml для Ansible collections. Фінальний provisioner scripts/configure_ansible.sh виконує критично важливу підготовку образу для самоналаштування, яка детально розглядається у наступному підрозділі, включаючи встановлення Python 3.12, Ansible Core, HashiCorp Vault CLI та створення системного користувача ansible з налаштуванням sudo привілеїв. Повний вміст файлу configure_ansible.sh показаний на Додатку Д.

Модульна структура Packer конфігурації з розділенням на декларативні блоки, параметризацію через змінні та екстерналізацію provisioning логіки у скрипти забезпечує читабельність, підтримуваність та можливість повторного використання компонентів. Використання HCL синтаксису замість JSON надає можливості для умовної логіки, циклів та функцій обробки даних, що робить конфігурації більш виразними та зменшує дублювання коду. Підтримка двох архітектур з єдиної конфігурації демонструє гнучкість підходу та спрощує майбутню міграцію між типами інстансів без переробки всієї системи збірки образів.

3.2.2. Встановлення залежностей.

Критичним етапом підготовки golden AMI є встановлення базових залежностей, необхідних для функціонування системи самоналаштування через ansible-pull. Цей процес реалізується через provisioning скрипт configure_ansible.sh, який виконується на етапі збірки образу з привілеями root та забезпечує встановлення системних пакетів, Python залежностей, Ansible collections та створення спеціалізованого системного користувача для виконання автоматизації. Скрипт розпочинається з визначення змінних середовища та налаштування режиму суворої обробки помилок через set -e, що призводить до негайного завершення виконання при виникненні будь-якої помилки у командах. Змінна PYTHON_VERSION фіксує використання Python 3.12 як стандартної версії інтерпретатора для забезпечення сумісності з Ansible Core 2.19.2. Змінна ANSIBLE_HOME визначає /opt/ansible як домашню директорію для системного

користувача ansible. Змінна FILES_DIR вказує на /tmp/files, куди попередній provisioner скопіював файли списків залежностей з Ansible ролі ansible_pull.

Першим кроком є додавання офіційного HashiCorp RPM репозиторію для дистрибутивів сімейства RHEL через команду dnf config-manager --add-repo, що забезпечує доступ до актуальних версій HashiCorp інструментів, зокрема Vault CLI. Встановлення системних пакетів виконується через менеджер пакетів dnf install -y з читанням списку з файлу requirements.dnf, який містить три критичні компоненти: python3.12 як базовий інтерпретатор для виконання Ansible, vault-1.18.0-1 як CLI інструмент для автентифікації до HashiCorp Vault та отримання секретів, git як клієнт системи контролю версій для клонування репозиторіїв з конфігураціями.

Наступним етапом є ініціалізація pip через модуль ensurepip, який створює або оновлює pip для вказаної версії Python за допомогою команди /usr/bin/python3.12 -m ensurepip. Встановлення Python бібліотек виконується з файлу requirements.txt, який містить фіксовані версії десяти пакетів з точною специфікацією через оператор == для забезпечення детерміністичності середовища. Команда /usr/local/bin/ansible-galaxy collection install встановлює Ansible Collections з файлу requirements.yaml до глобальної директорії /usr/share/ansible/collections, забезпечуючи доступність collections для всіх користувачів системи без необхідності повторної установки. Файл requirements.yaml специфікує вісім collections з фіксованими версіями для гарантування сумісності та стабільності виконання playbooks.

Створення спеціалізованого системного користувача ansible виконується командою useradd з параметрами: -r для створення системного облікового запису з UID менше 1000, -s /sbin/nologin для заборони інтерактивного входу через shell, -d /opt/ansible для визначення домашньої директорії, -g wheel для додавання до групи wheel з правами sudo. Налаштування sudo привілеїв без запиту пароля реалізується через створення файлу

`/etc/sudoers.d/ansible-pull-service` з директивою `ansible ALL=(ALL) NOPASSWD:ALL`, що дозволяє користувачу ansible виконувати будь-які команди з привілеями root без інтерактивної автентифікації. Права доступу до файлу встановлюються через `chmod 440` для захисту від несанкціонованої модифікації. Створення директорії `/opt/ansible` з встановленням власника та групи через `chown -R ansible:wheel` забезпечує робоче середовище для користувача ansible, де зберігатимуться клоновані Git репозиторії, тимчасові файли виконання playbooks та логи ansible-pull. Фінальним кроком скрипту є видалення тимчасової директорії `/tmp/files` через `rm -rf`, очищаючи образ від проміжних файлів встановлення та зменшуючи розмір результуючого AMI.

3.3. Cloud-init через user data (ansible pull.sh.tpl)

Механізм cloud-init у поєднанні з AWS user_data реалізує критичну фазу bootstrap нових EC2 інстансів, виконуючи початкову конфігурацію системи до активації механізму самоналаштування через ansible-pull. Template файл ansible_pull.sh.tpl інкапсулює логіку першого запуску, включаючи встановлення hostname, автентифікацію до HashiCorp Vault через AWS IAM, отримання Git credentials, клонування репозиторіїв та триетапне виконання Ansible playbooks для повної підготовки сервера. Скрипт виконується один раз при створенні інстансу з привілеями root та формує фундамент для подальшого автоматичного управління конфігурацією. Повний вміст файлу ansible_pull.sh.tpl показаний на Додатку Е.

Terraform модуль EC2 обробляє template файл через data source template_file, який виконує інтерполяцію змінних у шаблон перед передачею як user_data параметр ресурсу aws_instance. Змінні `${hostname}` та `${node_name}` замінюються на конкретні значення, забезпечуючи унікальність конфігурації для кожного створюваного інстансу. Результуючий скрипт зберігається у метаданих інстансу та залишається доступним через metadata service протягом всього життєвого циклу віртуальної машини.

Центральною частиною скрипту є динамічна генерація Ansible playbook `ansible-pull-init.yaml` через bash, який записує YAML контент у файл `/opt/ansible/ansible-pull-init.yaml`. Цей playbook інкапсулює логіку першого запуску, яка не може бути включена до Git репозиторіїв, оскільки самі репозиторії ще не клоновані на новостворений інстанс. Playbook виконується локально через connection: local та переключається на користувача ansible через become_user: ansible для забезпечення правильної ownership клонованих репозиторіїв.

Перший блок tasks збирає метадані EC2 інстансу через модуль `amazon.aws.ec2_metadata_facts`, який запитує instance metadata service та популює Ansible facts. Модуль `amazon.aws.ec2_tag_info` запитує теги інстансу через AWS API, використовуючи зібрані instance ID та region. Парсинг тегу Environment виконується через Jinja2 фільтр `split('.')`, який розділяє значення типу test.dev на компоненти `env_name (dev)` та `env_class (test)`. Ці змінні використовуються для формування URL Vault у форматі `https://vault-{env_name}.{env_class}.com` та зберігаються у файл `/opt/ansible/.env_vars` для використання наступними етапами виконання та systemd сервісом ansible-pull.

Модуль `community.hashi_vault.vault_login` виконує AWS IAM автентифікацію до Vault, використовуючи instance metadata для підпису запиту. Параметр `auth_method: aws_iam` інструктує модуль використати AWS IAM authentication method, який верифікує ідентичність інстансу через його IAM роль. Успішна автентифікація повертає токен, який використовується для подальших запитів до Vault API. Модуль `community.hashi_vault.vault_read` читає секрети з шляху `{env_class}/{env_name}/ansible-pull`, який містить Git credentials та інші параметри для клонування репозиторіїв.

Модуль `ansible.builtin.git_config` налаштовує глобальний Git credential helper як store, який зберігає credentials для автоматичної автентифікації при Git операціях через HTTPS. Модуль `ansible.builtin.copy` створює файл

`/opt/ansible/.git-credentials` з контентом у форматі `https://{username}:{password}@{hostname}` з обмеженими правами доступу 0600 для захисту sensitive даних. Клонування Git репозиторіїв виконується через модуль ansible.builtin.git у циклі для двох репозиторіїв: `ansible.git` до `/opt/ansible/ansible` та `ansible-inventory.git` до `/opt/ansible/ansible-inventory`. Git автоматично використовує credentials з .git-credentials для автентифікації до приватного репозиторію. Блок always у Ansible playbook гарантує виконання task видалення .git-credentials через ansible.builtin.file незалежно від успіху або помилки попередніх tasks.

Перший етап конфігурації виконує основний playbook main.yaml через ansible-playbook з тегом `ansible_pull_initial_install`, який застосовує роль ansible_pull для встановлення залежностей, створення systemd service та timer, налаштування Git refresh механізму. Параметр `--limit $(hostname)` обмежує виконання лише поточним хостом з inventory, а параметр `-e ansible_connection=local` інструктує Ansible виконувати tasks локально без SSH з'єднань.

Другий етап виконує ansible-pull від імені користувача ansible, який клонує свіжу копію репозиторію ansible.git через `--url` параметр, виконує playbook з тегами `$${ANSIBLE_INITIAL_TAGS}` для застосування конфігурацій конкретної ролі сервера. Третій етап застосовує тег `ansible_pull_daily_install` через локальний ansible-playbook, який встановлює systemd timer для щоденного виконання ansible-pull та фіналізує конфігурацію системи. Розділення на три етапи забезпечує правильний порядок операцій: встановлення інфраструктури ansible-pull, застосування специфічної конфігурації ролі сервера з свіжим клоном репозиторію, активація автоматичного механізму оновлень.

Фінальна команда `dnf needs-restarting -r || reboot` перевіряє необхідність перезавантаження системи після оновлення пакетів та kernel. Після перезавантаження systemd timer активується та починає щоденне виконання ansible-pull для підтримки системи в актуальному стані. Template файл

`ansible_pull.sh.tpl` реалізує повний bootstrap lifecycle нового EC2 інстансу від golden AMI до повністю сконфігурованого та самокерованого сервера з інтеграцією HashiCorp Vault через AWS IAM authentication для динамічного отримання секретів з централізованого сховища.

3.4. Ansible роль ansible pull

3.4.1. Структура ролі та попередні перевірки.

Роль `ansible_pull` організована згідно зі стандартною структурою Ansible ролей з директоріями `tasks`, `defaults`, `files`, `templates` та `meta`. Директорія `tasks` містить модульні task файли, кожен з яких відповідає за специфічну фазу конфігурування: `checks.yaml` для валідації вхідних параметрів, `install.yaml` для встановлення залежностей, `git_refresh_ansible.yaml` та `git_refresh_ansible_inventory.yaml` для оновлення репозиторіїв, `export_env_initial.yaml` та `export_env_daily.yaml` для генерації змінних середовища, `install_systemd_service.yaml` для налаштування автоматичного виконання.

Директорія `defaults` містить файл `main.yaml` з визначенням змінних ролі та їх значень за замовчуванням. Змінні на кшталт `ansible_pull_env_name`, `ansible_pull_env_class` та `ansible_pull_daily_tags` приймають значення з глобальних змінних inventory, які специфікують параметри середовища. Критично важливі змінні `ansible_pull_git_password`, `ansible_pull_git_username` та `ansible_pull_git_hostname` використовують Ansible lookup плагін `hashi_vault` для динамічного отримання Git credentials з HashiCorp Vault через AppRole authentication method при виконанні ролі. Параметри lookup включають шлях до секрету, метод автентифікації, role_id та secret_id з змінних середовища, URL Vault сервера.

Файл `checks.yaml` реалізує попередні перевірки через модуль ansible.builtin.assert, який валідує наявність та коректність критичних змінних перед виконанням основної логіки ролі. Перевірки включають умови is defined для

існування змінної, `is string` для типу даних, `length > 0` для непорожніх значень. Для змінної `ansible_pull_git_password` додатково перевіряється мінімальна довжина `length > 8` для забезпечення мінімальної складності паролів. Параметр `quiet: true` приховує деталі перевірки при успіху, виводячи інформацію лише при помилках валідації. Такий підхід забезпечує fail-fast поведінку з чіткими повідомленнями про відсутні або некоректні параметри, запобігаючи виконанню ролі з неповною конфігурацією та генерації cryptic помилок на пізніх етапах.

3.4.2. Встановлення залежностей.

Файл `install.yaml` інкапсулює логіку встановлення залежностей через три послідовні tasks з різними модулями для системних пакетів, Python бібліотек та Ansible collections. Task `Gather facts` виконує вибіркове збирання Ansible facts через модуль `ansible.builtin.setup` з параметром `gather_subset`, який обмежує збирання лише змінними середовища через `env`, виключаючи всі інші категорії facts через `!all`. Це оптимізує швидкість виконання, оскільки повне збирання facts включає численні запити до системи для мережевих інтерфейсів, дисків, процесора та інших підсистем.

Встановлення системних пакетів виконується через модуль `ansible.builtin.package`, який читає список з файлу `requirements.dnf` за допомогою lookup плагіна `file` з методом `splitlines()` для парсингу рядків у список. Файл містить три пакети: `python3.12`, `vault-1.18.0-1` та `git`. Модуль `package` є абстракцією над специфічними менеджерами пакетів (dnf для RHEL/AlmaLinux, apt для Debian/Ubuntu), автоматично визначаючи відповідний backend на основі дистрибутива. Параметр `state: present` забезпечує ідемпотентність, встановлюючи пакети лише якщо вони відсутні.

Python залежності встановлюються через модуль `ansible.builtin.pip` з параметром `requirements`, який вказує на файл `requirements.txt` у директорії ролі через змінну `role_path`. Модуль автоматично використовує pip для встановлення всіх пакетів з фіксованими версіями, забезпечуючи детерміністичність

середовища. Ansible collections встановлюються через модуль `ansible.builtin.shell`, який виконує команду `ansible-galaxy collection install` з явним встановленням змінних середовища `HOME` та `PATH` для забезпечення коректної роботи під cloud-init. Параметр `-r` специфікує файл `requirements.yaml`, параметр `-p` визначає глобальну директорію `/usr/share/ansible/collections` для доступності collections всім користувачам системи. Повний вміст файлу install.yaml показаний на Рис. 3.10.

```
---
- name: Gather facts
  ansible.builtin.setup:
    gather_subset:
      - '!all'
      - env

- name: Read file contents into a list
  ansible.builtin.set_fact:
    package_list: "{{ lookup('file', 'requirements.dnf').splitlines() }}"

- name: Install DNF packages
  ansible.builtin.package:
    name: "{{ package_list }}"
    state: present

- name: Install PIP packages
  ansible.builtin.pip:
    requirements: "{{ role_path }}/files/requirements.txt"

- name: Install Ansible collections (safe under cloud-init)
  ansible.builtin.shell: |
    export HOME=/root
    export PATH="/usr/local/bin:/usr/bin:/bin:{{ ansible_env.PATH }}"
    ansible-galaxy collection install \
      -r {{ role_path }}/files/requirements.yaml \
      -p /usr/share/ansible/collections
  args:
    executable: /bin/bash
```

Рисунок 3.10 - Вміст файлу install.yaml.

3.4.3. Клонування та оновлення Git репозиторіїв.

Оновлення Git репозиторіїв реалізується через окремі task файли `git_refresh_ansible.yaml` та `git_refresh_ansible_inventory.yaml` з

ідентичною структурою, що відрізняється лише цільовим репозиторієм. Використання блоку `block` з секцією `always` забезпечує гарантоване видалення Git credentials після завершення операцій незалежно від успіху або помилки виконання tasks. Вміст файлу git_refresh_ansible_inventory.yaml показаний на Додатку Є.

Перший task налаштовує глобальну Git конфігурацію через модуль `community.general.git_config`, встановлюючи `credential.helper` у значення `store` для використання plaintext файлу `.git-credentials` при автентифікації. Другий task додає директорії `/opt/ansible/ansible` та `/opt/ansible/ansible-inventory` до глобального списку `safe.directory` через параметр `add_mode: add`.

Task `Set git-credentials` створює файл `/opt/ansible/.git-credentials` через модуль `ansible.builtin.copy` з контентом у форматі `https://{username}:{password}@{hostname}`, де змінні `ansible_pull_git_username` та `ansible_pull_git_password` отримані з Vault через lookup плагін у defaults та URL-encoded через Jinja2 фільтр `urlencode`. Файл має строгі права доступу 0600 та ownership користувача ansible для захисту sensitive даних.

Клонування або оновлення репозиторію виконується через модуль `ansible.builtin.git` з параметрами `force: true` для перезапису локальних змін при конфліктах та `update: true` для виконання git pull при існуючому репозиторії. Параметр `version` специфікує branch для checkout: змінна `ansible_pull_git_branch_ansible` для репозиторію ansible та hardcoded значення `master` для ansible-inventory. Git автоматично читає credentials з файлу `.git-credentials`, налаштованого у попередньому task, для автентифікації до приватного репозиторію без інтерактивних prompt.

Секція `always` гарантує виконання task видалення `.git-credentials` через модуль `ansible.builtin.file` з `state: absent` після завершення блоку. Умова `when: "'ansible_pull_initial_install' not in ansible_run_tags"` запобігає видаленню credentials при першому запуску, оскільки файл необхідний для

наступних етапів bootstrap процесу у cloud-init скрипті. При щоденному виконанні через systemd service credentials видаляються одразу після клонування, мінімізуючи exposure sensitive даних на файловій системі. Такий підхід балансує між необхідністю автентифікації для Git операцій та принципом мінімізації збереження секретів у незашифрованому вигляді.

3.4.4. Експорт змінних середовища та система тегів.

Генерація змінних середовища реалізується через два окремі task файли `export_env_initial.yaml` та `export_env_daily.yaml`, які створюють файл `/opt/ansible/.env_vars` з різним набором змінних залежно від фази життєвого циклу сервера. Файл `.env_vars` використовується bash скриптом `run-ansible-pull.sh` через команду `source` для імпорту змінних у середовище виконання перед запуском ansible-pull.

Task файл `export_env_initial.yaml` виконується під час першого запуску з тегом `ansible_pull_initial_install` та генерує розширений набір змінних, включаючи Git credentials. Модуль `ansible.builtin.copy` створює файл з контентом у форматі `KEY="value"` через heredoc у параметрі `content`. Змінна `ANSIBLE_INITIAL_TAGS` містить список тегів для застосування при першому виконанні ansible-pull, які визначають конфігурації конкретної ролі сервера (database, application, load balancer). Змінні `GIT_USERNAME` та `GIT_PASSWORD` включаються для автентифікації при клонуванні репозиторію ansible-pull командою, що дозволяє bootstrap скрипту отримати свіжу версію playbooks.

Task файл `export_env_daily.yaml` виконується при активації systemd service з тегом `ansible_pull_daily_install` та генерує мінімізований набір змінних без Git credentials. Основна відмінність полягає у заміні `ANSIBLE_INITIAL_TAGS` на `ANSIBLE_DAILY_TAGS`, яка містить список тегів для щоденного виконання, що зазвичай обмежується оновленням конфігураційних файлів, перезапуском сервісів та застосуванням security патчів без повної реконфігурації сервера. Відсутність Git credentials у daily версії обумовлена тим, що скрипт `run-ansible-pull.sh` отримує їх безпосередньо з Vault при кожному

виконанні через AWS IAM автентифікацію, уникаючи збереження sensitive даних у файловій системі між запусками. Вмісти вайлів export_env_initial.yaml та export_env_daily.yaml показано на Рис. 3.11 та 3.12 відповідно.

```
---
- name: Create .env_vars for initial install tags
  ansible.builtin.copy:
    content: |
      ANSIBLE_GIT_BRANCH="{{ ansible_pull_git_branch_ansible }}"
      ANSIBLE_INITIAL_TAGS="{{ ansible_pull_initial_tags }}"
      ENV_CLASS="{{ ansible_pull_env_class }}"
      ENV_NAME="{{ ansible_pull_env_name }}"
      GIT_HOSTNAME="{{ ansible_pull_git_hostname }}"
      GIT_PASSWORD="{{ ansible_pull_git_password }}"
      GIT_USERNAME="{{ ansible_pull_git_username }}"
    dest: "/opt/ansible/.env_vars"
    owner: ansible
    group: wheel
    mode: "0640"
```

Рисунок 3.11 - Вміст файлу export_env_initial.yaml.

```
---
- name: Create .env_vars for daily tags
  ansible.builtin.copy:
    content: |
      ANSIBLE_DAILY_TAGS="{{ ansible_pull_daily_tags }}"
      ANSIBLE_GIT_BRANCH="{{ ansible_pull_git_branch_ansible }}"
      ENV_CLASS="{{ ansible_pull_env_class }}"
      ENV_NAME="{{ ansible_pull_env_name }}"
      GIT_HOSTNAME="{{ ansible_pull_git_hostname }}"
    dest: "/opt/ansible/.env_vars"
    owner: ansible
    group: wheel
    mode: "0640"
```

Рисунок 3.12 - Вміст файлу export_env_daily.yaml.

3.4.5. SystemD сервіс та таймер.

Автоматизація щоденного виконання ansible-pull реалізується через systemd service та timer units, які створюються task файлом `install_systemd_service.yaml`. Перший task створює директорію `/opt/ansible/bin/` через модуль `ansible.builtin.file` з ownership

користувача ansible для зберігання виконуваного скрипту. Другий task копіює скрипт `run-ansible-pull.sh` з директорії `files` ролі до `/opt/ansible/bin/` з правами виконання 0755, який інкапсулює логіку автентифікації до Vault, отримання Git credentials, виконання ansible-pull та очищення sensitive даних. Вміст файлу install_systemd_service.yaml показано на Рис. 3.13.

```
---
- name: Ensure /opt/ansible/bin/ exists
  ansible.builtin.file:
    path: /opt/ansible/bin/
    state: directory
    owner: ansible
    group: wheel
    mode: "0755"

- name: Copy Ansible run bin script
  ansible.builtin.copy:
    src: run-ansible-pull.sh
    dest: /opt/ansible/bin/run-ansible-pull.sh
    owner: ansible
    group: wheel
    mode: "0755"

- name: Create Ansible pull systemd service
  ansible.builtin.template:
    src: ansible-pull.service.j2
    dest: /etc/systemd/system/ansible-pull.service
    owner: root
    group: root
    mode: "0644"

- name: Create Ansible pull systemd timer
  ansible.builtin.template:
    src: ansible-pull.timer.j2
    dest: /etc/systemd/system/ansible-pull.timer
    owner: root
    group: root
    mode: "0644"

- name: Start Ansible-pull timer
  ansible.builtin.systemd:
    name: ansible-pull.timer
    state: started
    enabled: true
    daemon_reload: true
```

Рисунок 3.13 - Вміст файлу install_systemd_service.yaml.

Третій та четвертий tasks створюють systemd unit файли через модуль `ansible.builtin.template`, який обробляє Jinja2 templates `ansible-pull.service.j2` та `ansible-pull.timer.j2`, генеруючи фінальні конфігурації у директорії `/etc/systemd/system/`. Файли мають ownership root та права доступу 0644 для захисту від несанкціонованої модифікації користувачами без root привілеїв. Вміст файлів ansible_pull.service.j2 та ansible_pull.timer.j2 показано на Рис. 3.14 та 3.15 відповідно.

```
[Unit]
Description=Run ansible-pull
After=network.target

[Service]
Type=oneshot
User=ansible
ExecStart=/opt/ansible/bin/run-ansible-pull.sh

[Install]
WantedBy=multi-user.target
```

Рисунок 3.14 - Вміст файлу ansible_pull.service.j2.

```
[Unit]
Description=Run ansible-pull daily
Requires=ansible-pull.service

[Timer]
Unit=ansible-pull.service
OnBootSec=5min
OnCalendar=Mon..Sun *-*-* 4:00:00
Persistent=true
RandomizedDelaySec=3600

[Install]
WantedBy=timers.target
```

Рисунок 3.15 - Вміст файлу ansible_pull.timer.j2.

3.5. Скрипт run-ansible-pull.sh.

3.5.1. Структура та призначення.

Скрипт `run-ansible-pull.sh` є центральним компонентом системи самоналаштування, який виконується щоденно через systemd timer та інкапсулює повний цикл синхронізації конфігурації сервера з Git репозиторіями. Скрипт організований у чітку послідовність етапів: ініціалізація середовища та перевірка цілісності, оновлення Git репозиторіїв через Ansible playbook з тегами, автоматичне перезавантаження при зміні самого скрипту, автентифікація до HashiCorp Vault для отримання Git credentials, виконання ansible-pull з daily тегами та очищення sensitive даних після завершення. Повний вміст файлу run-ansible-pull.sh показаний на додатку Ж.

Скрипт розпочинається з shebang `#!/usr/bin/env bash` для портативності між різними Unix-подібними системами та директиви `set -euo pipefail` для суворої обробки помилок. Критично важливим механізмом є обчислення MD5 хешу самого скрипту через `md5sum "$0"` та збереження у змінну `SCRIPT_ORIG_MD5` перед виконанням основної логіки. Змінна `SCRIPT_RESTART` ініціалізується з environment або порожнім рядком через parameter expansion `${SCRIPT_RESTART:-""}`, зберігаючи значення при рекурсивному виклику скрипту. Завантаження змінних середовища з файлу `/opt/ansible/.env_vars` через команду `source` імпортує параметри `ENV_CLASS`, `ENV_NAME`, `ANSIBLE_CALLBACK`, `ANSIBLE_DAILY_TAGS`, `ANSIBLE_GIT_BRANCH`, `GIT_HOSTNAME` та `SUMOLOGIC_ENV_URL`, які використовуються протягом виконання скрипту.

Перші три команди виконують ansible-playbook з різними наборами тегів для підготовки середовища перед основним виконанням ansible-pull. Перша команда застосовує теги `ansible_pull_check` та `ansible_pull_refresh_git_ansible_inventory` для валідації змінних та оновлення inventory репозиторію. Друга команда застосовує тег `ansible_pull_refresh_git_ansible` для оновлення основного ansible

репозиторію, який може містити модифікований скрипт `run-ansible-pull.sh`. Третя команда застосовує тег `ansible_pull_daily_install` для оновлення файлу `.env_vars` з актуальними значеннями змінних, після чого виконується повторний source для завантаження можливих змін

3.5.2. Автентифікація до Vault та отримання Git credentials.

Автентифікація до HashiCorp Vault виконується через Vault CLI з AWS IAM authentication method. Параметр `header_value` передає значення для `X-Vault-AWS-IAM-Server-ID` header, який повинен співпадати з налаштуванням `iam_server_id_header_value` на Vault сервері. Параметр `role` специфікує назву ролі у Vault AWS auth backend, сформовану з змінних середовища як `${ENV_NAME}-${ENV_CLASS}-role-iam`, яка визначає набір Vault політик для токена.

Отриманий токен зберігається у environment змінній `VAULT_TOKEN` через `export`, роблячи його доступним для подальших команд `vault read` без явної специфікації через параметр `-token`. Дві наступні команди читають Git credentials з Vault key-value storage через `vault read` з параметром `-field` для екстракції конкретного ключа з секрету. Шлях до секрету формується як `${ENV_CLASS}/${ENV_NAME}/ansible-pull`, відповідаючи ієрархічній структурі організації секретів у Vault. Поля `git_username` та `git_password` екстрагуються у окремі змінні через command substitution `$(...)` та використовуються для формування Git credentials URL.

Генерація файлу `.git-credentials` виконується через append operation `>>`, записуючи рядок у форматі `https://{username}:{password}@{hostname}` до файлу `/opt/ansible/.git-credentials`. Використання append замість overwrite дозволяє потенційно зберігати credentials для декількох Git hostnames, хоча у типовій конфігурації файл містить лише один рядок. Змінна `GIT_HOSTNAME` завантажена з `.env_vars` забезпечує гнучкість у зміні Git сервера без модифікації скрипту. Після створення credentials файлу скрипт виконує `unset VAULT_TOKEN` для видалення токена з environment.

3.5.3. Виконання ansible-pull з тегами.

Виконання ansible-pull реалізується через пряму команду `/usr/local/bin/ansible-pull` з множиною параметрів для контролю поведінки клонування, checkout та виконання playbook.

Параметр `--clean` інструктує ansible-pull видалити існуючу локальну копію репозиторію та виконати fresh clone, гарантуючи відсутність залишкових локальних змін або corrupted Git state. Параметр `--url https://${GIT_HOSTNAME}/ansible.git` специфікує URL Git репозиторію для клонування з динамічною інтерполяцією hostname з environment змінної. Git автоматично використовує credentials з файлу `.git-credentials`, створеного на попередньому етапі, для автентифікації до приватного репозиторію. Параметр `--accept-host-key` автоматично приймає SSH host key Git сервера без інтерактивного prompt, що критично важливо для неінтерактивного виконання через systemd service.

Параметр `--checkout ${ANSIBLE_GIT_BRANCH}` специфікує Git branch для checkout з environment змінної, дозволяючи різним середовищам використовувати різні branches репозиторію. Development середовища можуть використовувати `develop` branch для тестування нових конфігурацій, staging використовує `staging` для pre-production валідації, production використовує стабільний `master` або `production` branch. Така стратегія дозволяє поступове просування змін через середовища з валідацією на кожному рівні перед deployment до критичних production систем.

Параметр `-i $ANSIBLE_HOME/ansible-inventory/${ENV_CLASS}/${ENV_NAME}` специфікує шлях до inventory директорії з змінними та групами для конкретного середовища. Параметри `-e ansible_connection=local` та `-e ansible_pull=true` передають extra vars для модифікації поведінки playbooks, визначаючи локальне виконання без SSH та режим pull замість push. Параметр `--tags ${ANSIBLE_DAILY_TAGS}` застосовує лише tasks з відповідними тегами з

environment змінної, яка зазвичай містить значення на кшталт `database_config,monitoring` для оновлення конфігураційних файлів, перезапуску сервісів та синхронізації моніторингу без повної переінсталяції програмного забезпечення.

Параметр `--limit "$(hostname)"` обмежує виконання playbook лише поточним хостом з inventory, навіть якщо inventory містить десятки або сотні серверів. Command `$(hostname)` динамічно визначає FQDN сервера для точного matching з inventory. Фінальний аргумент `main.yaml` специфікує entry point playbook для виконання. Команда завершується через semicolon з наступною командою `rm -f $ANSIBLE_HOME/.git-credentials`, яка виконується незалежно від exit code ansible-pull, забезпечуючи очищення credentials навіть при помилках виконання playbook.

3.5.4. Очищення sensitive даних.

Очищення sensitive даних реалізовано на декількох рівнях для мінімізації exposure credentials у файловій системі та process environment. Перше очищення виконується через `unset VAULT_TOKEN` одразу після використання токена для читання Git credentials з Vault, видаляючи токен з environment змінних поточного bash process. Це запобігає передачі VAULT_TOKEN до дочірніх процесів та його випадковому логуванню при виведенні environment через debugging команди на кшталт `printenv` або `set`.

Друге очищення виконується через `rm -f $ANSIBLE_HOME/.git-credentials` як фінальна команда скрипту, об'єднана з ansible-pull через semicolon для гарантованого виконання. Використання `-f` flag запобігає помилкам при відсутності файлу, забезпечуючи ідемпотентність команди. Розміщення команди видалення на тому ж рядку через semicolon замість окремого рядка є критичним, тому що це гарантує виконання очищення навіть якщо ansible-pull завершується з non-zero exit code через помилки у playbook. Якби команда була на окремому рядку, директива `set -e` припинила б виконання скрипту після помилки ansible-pull, залишивши credentials файл на диску.

Час між створенням .git-credentials та його видаленням мінімізовано через послідовне виконання команд без затримок. Git читає credentials файл при ініціації HTTPS з'єднання до віддаленого репозиторію, що відбувається на початку виконання ansible-pull при клонуванні. Після завершення клонування та виконання playbook файл більше не потрібен та негайно видаляється. У типовому сценарії час існування файлу становить кілька хвилин, протягом яких файл доступний лише користувачу ansible через обмежені права доступу, встановлені при створенні файлу у попередніх етапах.

Відсутність Git credentials у `.env_vars` файлі для daily режиму є додатковим шаром безпеки, який відділяє короткочасні credentials (отримані з Vault при кожному виконанні) від довготривалих конфігураційних параметрів (середовище, теги, hostnames). Якби credentials зберігалися у `.env_vars` з правами доступу 0640, вони були б доступні будь-якому процесу користувача ansible або членам групи wheel протягом всього часу між запусками ansible-pull. Динамічне отримання credentials з Vault через AWS IAM автентифікацію усуває необхідність збереження статичних секретів у файловій системі, реалізуючи принцип “just-in-time access” до сенсативних даних.

Архітектура скрипту з автоматичним перезавантаженням при зміні MD5 хешу забезпечує self-healing capability, де будь-які зміни у логіці автентифікації, очищення або параметрах виконання автоматично застосовуються при наступному запуску systemd timer без необхідності manual intervention. Це критично важливо для масштабованості системи, де сотні серверів автоматично отримують оновлення скрипту через Git та негайно починають використовувати нову версію, забезпечуючи консистентність поведінки по всій інфраструктурі. Загальна архітектура самооновлення скрипту показана на Рис. 3.16.

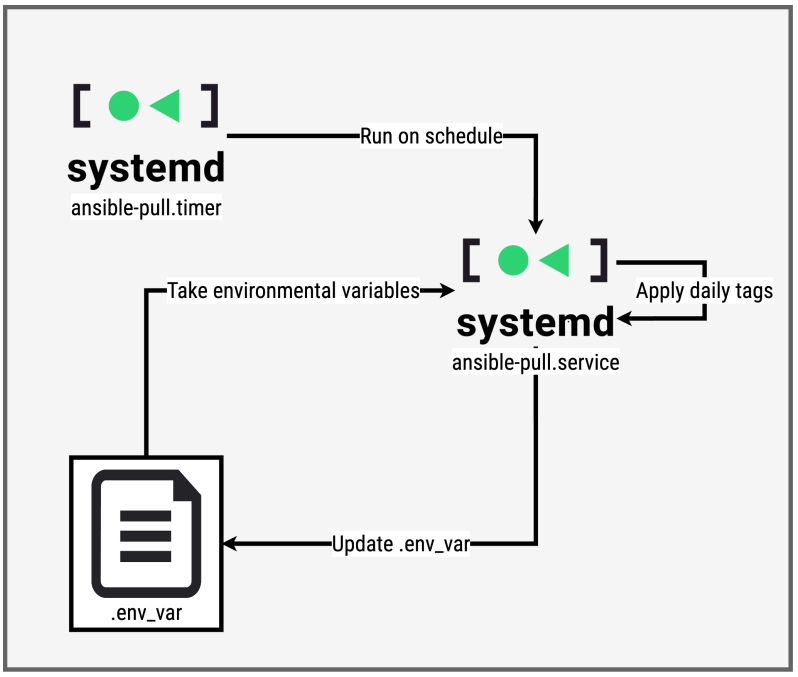

Рисунок 3.16 - Загальна архітектура самооновлення скрипту run-ansible-pull.sh.

Висновок до розділу 3

У третьому розділі здійснено практичну реалізацію спроектованої системи самоналаштування віртуальних машин з детальною специфікацією всіх компонентів архітектури.

Налаштовано HashiCorp Vault з AWS IAM authentication method для динамічної автентифікації на основі ідентичності EC2 інстансів. Створено IAM ролі з гранульованими permissions та Vault політики, які реалізують принцип найменших привілеїв. Організовано Git репозиторії ansible та ansible-inventory з розділенням між виконуваним кодом та метаданими інфраструктури. Розроблено Packer конфігурації для автоматизованого створення golden AMI з попередньо встановленими залежностями (Python 3.12, Ansible Core, Vault CLI, Git) для архітектур x86_64 та arm64. Конфігурації використовують HCL синтаксис з модульною структурою, параметризацією через змінні та версіонуванням модулів через Git теги.

Створено Ansible роль ansible_pull з модульною архітектурою tasks, системою валідації вхідних параметрів через assert та гранулярним контролем виконання через теги. Роль встановлює залежності, оновлює Git репозиторії з динамічним

отриманням credentials з Vault, генерує змінні середовища для initial та daily режимів, встановлює SystemD service та timer для періодичного виконання.

Імплементовано bash скрипт run-ansible-pull.sh з механізмом автоматичного самооновлення через порівняння MD5 хешів, що дозволяє застосовувати зміни у логіці автоматизації без перезбудови базових образів. Скрипт виконує автентифікацію до Vault через AWS IAM, отримує Git credentials, виконує ansible-pull з daily тегами та автоматично очищує sensitive дані після завершення операцій.

Реалізована система демонструє повну інтеграцію всіх компонентів технологічного стеку для забезпечення автоматизованого життєвого циклу віртуальних машин від створення до періодичного самооновлення конфігурацій. Практична імплементація підтверджує застосовність спроектованої архітектури та можливість deployment у production середовищах з вимогами до високої доступності, масштабованості та безпеки.

ВИСНОВКИ

У кваліфікаційній роботі досліджено проблематику управління конфігураціями у хмарних інфраструктурах та розроблено архітектуру системи самоналаштування віртуальних машин на основі Ansible Pull з інтеграцією HashiCorp Vault для динамічного управління секретами. Виконане дослідження охоплює теоретичні основи Infrastructure as Code, детальне проєктування архітектури системи та практичну реалізацію всіх компонентів з використанням сучасного технологічного стеку DevOps інструментів.

Проведено аналіз обмежень централізованої push-моделі управління конфігураціями, який виявив єдину точку відмови, проблеми масштабування, складність мережевої топології та ризики безпеки при централізованому зберіганні credentials. Сформульовано функціональні та нефункціональні вимоги до альтернативної архітектури з акцентом на автономність, ідемпотентність, безпеку та відмовостійкість. Розроблено архітектуру системи самоналаштування з pull-моделлю виконання, де кожна віртуальна машина автономно оновлює власні конфігурації, автентифікуючись до Vault на основі IAM ідентичності та отримуючи playbooks з Git репозиторіїв. Архітектура інтегрує Packer для створення golden images, Terraform для провізіонінгу інфраструктури, HashiCorp Vault для управління секретами, AWS IAM для автентифікації та SystemD для періодичного виконання. Здійснено практичну реалізацію системи з налаштуванням Vault AWS authentication, створенням IAM ролей, організацією Git репозиторіїв, розробкою Packer конфігурацій для x86_64 та arm64 архітектур, імплементацією Terraform модулів з cloud-init інтеграцією, створенням Ansible ролі ansible_pull та скрипту run-ansible-pull.sh з механізмом автоматичного самооновлення.

Розроблена система усуває обмеження централізованої архітектури через децентралізацію прийняття рішень та розподілене виконання. Відсутність control node забезпечує природну відмовостійкість та масштабованість для управління тисячами віртуальних машин без деградації продуктивності. Інтеграція AWS IAM з HashiCorp Vault реалізує динамічну автентифікацію на основі ідентичності без

зберігання статичних секретів. Дворівневий підхід з IAM автентифікацією для отримання AppRole credentials забезпечує гранульований контроль доступу до конфігураційних секретів з реалізацією принципу найменших привілеїв. Механізм автоматичного самовідновлення через SystemD timer з випадковою затримкою та автооновлення скрипту через MD5 хеші забезпечує self-healing capability без перезбудови базових образів. Система тегів Ansible дозволяє диференціювати між початковим встановленням та щоденним оновленням з ієрархічною організацією змінних для різних типів серверів.

Практична цінність проявляється у застосовності для production інфраструктур з вимогами до високої доступності та масштабованості. Усунення залежності від control node підвищує надійність, автоматизація життєвого циклу зменшує operational overhead. Динамічне управління секретами з короткотерміновими credentials та гранульовані політики доступу забезпечують дотримання compliance вимог з повною трасованістю через Vault logs та SystemD journald. Система демонструє переваги у масштабованості через природне горизонтальне масштабування без додаткових control nodes, відмовостійкості через відсутність єдиної точки відмови та operational простоті через зменшення кількості компонентів для підтримки. Спрощення мережевої топології через інверсію напрямку підключень усуває необхідність вхідних SSH портів, зменшує поверхню атаки та дозволяє розміщення у приватних підмережах. Автоматизація оновлення скриптів через Git та централізоване логування забезпечують ефективний моніторинг та аналіз конфігурацій.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Spinellis, D. (2012). "Don't Install Software by Hand." IEEE Software, 29(4), pp. 86-87.
2. Terraform Documentation (2023). Introduction to Infrastructure as Code with Terraform. HashiCorp. Retrieved from https://www.terraform.io/intro
3. Loeliger, J., & McCullough, M. (2012). Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development. O'Reilly Media.
4. Rahman, A., Parnin, C., & Williams, L. (2019). "The Seven Sins: Security Smells in Infrastructure as Code Scripts." In Proceedings of the 41st International Conference on Software Engineering (ICSE), pp. 164-175.
5. HashiCorp Vault Documentation (2023). Security and Best Practices. HashiCorp. Retrieved from https://www.vaultproject.io/docs/internals/security
6. DeHaan, M. (2013). "Ansible: The Inside Story." Ansible Blog. Retrieved from https://www.ansible.com/blog/
7. Shah, G. (2015). Ansible playbook essentials: Design automation blueprints using Ansible’s Playbooks to orchestrate and manage your multitier infrastructure. Packt Publishing.
8. Community, A. Ansible documentation. Ansible Documentation. https://docs.ansible.com/
9. AWS Dynamic Inventory Plugin (2023). Ansible Documentation. Red Hat, Inc. Retrieved from https://docs.ansible.com/ansible/latest/collections/amazon/aws/aws_ec2_inventory.html
10. Ansible Pull Documentation (2023). Pull-mode Playbook Execution. Red Hat, Inc. Retrieved from https://docs.ansible.com/ansible/latest/cli/ansible-pull.html
11. Geerling, J. (2020). Ansible for devops server and configuration management for humans Jeff Geerling. Leanpub.
12. Yevgeniy Brikman. Terraform: Up and Running. “O’Reilly Media, Inc.,” 2017.

13. “Backend Block Configuration Overview | Terraform | HashiCorp Developer.” Backend Block Configuration Overview | Terraform | HashiCorp Developer, 2025, developer.hashicorp.com/terraform/language/backend. Accessed 2 Nov. 2025.
14. “Documentation | Packer | HashiCorp Developer.” Documentation | Packer | HashiCorp Developer, developer.hashicorp.com/packer/docs.
15. “What Is a Golden Image?” www.redhat.com, www.redhat.com/en/topics/linux/what-is-a-golden-image.
16. Provisioners overview | packer | Hashicorp developer. https://developer.hashicorp.com/packer/docs/provisioners
17. “What Is Vault? | Vault | HashiCorp Developer.” What Is Vault? | Vault | HashiCorp Developer, 2024, developer.hashicorp.com/vault/docs/about-vault/what-is-vault.
18. “Database Secrets Engine | Vault | HashiCorp Developer.” Database Secrets Engine | Vault | HashiCorp Developer, 2025, developer.hashicorp.com/vault/docs/secrets/databases. Accessed 3 Nov. 2025.
19. Amazon Web Services. “What Is Amazon EC2? - Amazon Elastic Compute Cloud.” Amazon.com, 2025, docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html.
20. AWS. “What Is Amazon VPC? - Amazon Virtual Private Cloud.” Amazon.com, 2023, docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html.
21. AWS. “What Is IAM? - AWS Identity and Access Management.” Amazon.com, 2025, docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html.
22. “Cloud-Init 24.4 Documentation.” Readthedocs.io, 2017, cloudinit.readthedocs.io/en/latest/.
23. “Systemd.” Www.freedesktop.org, www.freedesktop.org/wiki/Software/systemd/.
24. Silverman, Richard E., et al. Git Pocket Guide. O’reilly, 2013.
25. “Vault Authentication Using AWS IAM Role Example.” Terraform.io, 2025, registry.terraform.io/modules/hashicorp/vault/aws/latest/examples/vault-iam-auth. Accessed 8 Nov. 2025.

ДОДАТКИ

Додаток А

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет імені В. Н. Каразіна

Навчально-науковий інститут комп'ютерних наук та штучного інтелекту
Кафедра комп'ютерних систем та робототехніки
Рівень вищої освіти (освітньо-кваліфікаційний рівень) ***Магістр***
Галузь знань: 12 – Інформаційні технології
Спеціальність: 123 «Комп'ютерна інженерія»
Освітня програма «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ
Завідувач кафедри комп'ютерних
систем та робототехніки
к. ф.-м. н., доц. ХРУСЛОВ М. М.
«02» жовтня 2024року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

САМОЙЛЕНКО Владислав Васильович
(прізвище, ім'я, по батькові студента)

1. Тема роботи: **«КОМП'ЮТЕРНА МОДЕЛЬ САМОНАЛАШТУВАННЯ VIRTUAL MACHINES ENVIRONMENT З ВИКОРИСТАННЯМ ANSIBLE PULL APPROACH»**

керівник роботи **МОРОЗ Ольга Юріївна, : PhD, доцент кафедри комп'ютерних систем та робототехніки,**
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету ***від 30 вересня 2025 року № 4101-5/3554***

2. Строк подання студентом роботи ***30 листопада 2025 року***

3. Перелік питань, які потрібно розробити)
1) Аналіз сучасних методів автоматизації конфігурації віртуальних машин у хмарних середовищах.
2) Порівняння push- та pull-моделей керування конфігураціями та визначення їхніх переваг і обмежень.
3) Дослідження архітектурних принципів Ansible Pull та можливостей інтеграції з Git-орієнтованими робочими процесами.
4) Розробка моделі самоконфігурації, заснованої на тегуванні ролей Ansible і періодичних оновленнях конфігурацій.
5) Аналіз безпекових ризиків, пов'язаних із розподіленою конфігурацією, та визначення методів їх мінімізації.
6) Тестування працездатності та ідемпотентності застосованих конфігураційних ролей у різних сценаріях розгортання.

4. План роботи

№ з/п	Назви етапів роботи	Термін виконання етапів роботи
1	Затвердження теми роботи та керівника	02.09.2024 -
2	Розробка індивідуального графіку проходження практики. Узгодження його з науковим керівником дипломної роботи.	02.09.2024 - 02.10.2024
3	Вивчення нормативних документів, що регулюють питання науково-дослідної діяльності в Україні *щодо*	03.09.2024 - 24.10.2024
4	Самостійний аналіз сучасних підходів до автоматизації конфігурації віртуальних машин та DevOps-інструментів	25.10.2024 - 09.11.2024
5	Систематизація та впорядкування зібраного фактичного матеріалу щодо Ansible Pull, Terraform, Vault, Git	10.11.2024 - 24.11.2024
6	Висвітлення результатів власного наукового дослідження: написання перших розділів роботи, формулювання проблеми та методів	25.11.2024 - 08.12.2024
7	Коректне й аргументоване викладення власної думки на основі самостійної науково-дослідної діяльності; захист спостережень перед науковим керівником	09.12.2024 - 29.01.2025
8	Підготовка і оформлення звітних матеріалів.	30.01.2025- 31.02.2025
9	Оформлення звіту про науково-дослідну практику Написання статті за матеріалами кваліфікаційної роботи.	01.03.2025- 01.04.2025
10	Підготовка і оформлення звітних матеріалів кваліфікаційної роботи. Оформлення списку літератури	01.05.2025- 30.08.2025-
11	Оформлення пояснювальної записки кваліфікаційної роботи відповідно вимогам до звітів про НДР.	01.09.2025- 30.10.2025-
12	Підготовка і оформлення звітних матеріалів та додатків кваліфікаційної роботи.	10.10.2025- 30.10.2025-
13	Оформлення звіту про переддипломну практику	01.11.2025- 30.11.2025
14	Представлення кваліфікаційної роботи керівнику та рецензенту	24.11.2025 - 30.11.2025

5. Дата видачі завдання ***02 жовтня 2024 року.***

Студент Самойленко В. В.
ініціали, прізвище
підпис

Керівник роботи Мороз О. Ю.
ініціали, прізвище
підпис

Додаток Б

Затверджую

_____

«_____»_____2025 р.

Технічне завдання
на розробку програмного виробу «Комп'ютерна модель самоналаштування virtual machines environment з використанням Ansible Pull approach»

1.	Введення	1.1. Назва: Pull-based система управління конфігурацією на базі Ansible Pull з інтеграцією HashiCorp Vault та AWS IAM. 1.2. Галузь застосування: DevOps, Infrastructure as Code, автоматизація інфраструктури, хмарні обчислення.
2.	Підстава для розробки	2.1. Навчальний план за спеціальністю 123 - Комп’ютерна інженерія. 2.2. Завдання на кваліфікаційну роботу магістра № 4101-5/3554 від 30 вересня 2025 (представити як Додаток Б до пояснювальної записки до кваліфікаційної роботи магістра).
3.	Призначення розробки	3.1. Мета розробки: створення децентралізованої pull-based системи автоматизованого управління конфігурацією серверної інфраструктури з підвищеним рівнем безпеки, автономності та масштабованості. 3.2. Призначення розробки надає можливість: - забезпечити автономне управління конфігурацією серверних вузлів без централізованого контролера; - реалізувати динамічне отримання секретів з HashiCorp Vault через AWS IAM аутентифікацію; - автоматизувати процес розгортання та оновлення конфігурацій серверів у хмарному середовищі AWS; - знизити операційні витрати на підтримку інфраструктури через усунення залежності від централізованого керуючого сервера; - підвищити відмовостійкість системи управління конфігурацією завдяки децентралізованій архітектурі; - спростити масштабування інфраструктури через автоматичне самоконфігурування нових вузлів. 3.3. Вихідні дані розробки: - документація Ansible, HashiCorp Vault, Terraform, Packer;

		- архітектурні рішення існуючих систем управління конфігурацією; - вимоги до безпеки та надійності інфраструктурних систем; - специфікації AWS EC2, IAM та інших хмарних сервісів.
4.	Технічні вимоги до програмного виробу	4.1. Вимоги до функціональних характеристик: - автоматичне отримання конфігурацій з Git-репозиторіїв; - динамічна аутентифікація у HashiCorp Vault через AWS IAM ролі; - періодичне виконання конфігураційних завдань через systemd таймери; - валідація конфігурацій перед застосуванням; - логування та моніторинг процесу виконання конфігурацій; - обробка помилок та механізми відновлення. 4.2. Вимоги до надійності: - забезпечувати ідемпотентність операцій конфігурування; - гарантувати автономність роботи вузлів при недоступності центральних компонентів; - підтримувати механізми автоматичного відновлення при збоях; - забезпечувати консистентність конфігурацій між вузлами. 4.3. Вимоги до умов експлуатації: - робота в хмарному середовищі AWS (EC2 instances); - підтримка Linux-based операційних систем (AlmaLinux, RHEL-сумісні дистрибутиви); - наявність мережевого з'єднання для доступу до Git-репозиторіїв та HashiCorp Vault. 4.4. Вимоги до складу і параметрів технічних засобів: - AWS EC2 instances з IAM ролями; - HashiCorp Vault кластер для зберігання секретів; - Git-репозиторії для зберігання Ansible playbooks та інвентарю; - мінімальні системні вимоги: 512 MB RAM, 1 CPU core, 5 GB дискового простору. 4.5. Вимоги до інформаційної та програмної сумісності: - Ansible версії 2.13 або вище; - HashiCorp Vault версії 1.8 або вище; - Python 3.9 або вище;

		- systemd для управління службами; - Git для отримання конфігурацій. 4.6. Вимоги до маркування та упаковки: інфраструктурний код зберігається у Git-репозиторіях з версіонуванням та тегуванням релізів. 4.7. Вимоги до транспортування і зберігання: зберігання у Git-репозиторіях з системою контролю версій, резервне копіювання репозиторіїв. 4.8. Спеціальні вимоги: - дотримання принципів Infrastructure as Code; - застосування принципу найменших привілеїв для IAM політик; - шифрування секретів у HashiCorp Vault; - аудит доступу до конфіденційних даних.
5.	Вимоги до програмної документації	Програмною документацією до виробу «Pull-based система управління конфігурацією на базі Ansible Pull з інтеграцією HashiCorp Vault та AWS IAM» вважати: 1) Справжнє Технічне завдання на розробку виробу (представити у вигляді Додатку Б до пояснювальної записки до кваліфікаційної роботи). 2) Архітектуру системи та опис компонентів (розділ 2 пояснювальної записки до кваліфікаційної роботи). 3) Методику розгортання та налаштування системи (розділ 3 пояснювальної записки до кваліфікаційної роботи). 4) Опис Ansible ролей, Terraform модулів та Packer конфігурацій (розділ 3 пояснювальної записки до кваліфікаційної роботи). 5) Інструкцію з експлуатації системи (як частину розділу 3 пояснювальної записки до кваліфікаційної роботи).
6.	Вимоги до техніко-економічних показників	6.1. Зниження операційних витрат на управління інфраструктурою за рахунок: - усунення необхідності підтримки централізованого Ansible control node; - автоматизації процесів конфігурування та оновлення; - зменшення часу на виявлення та виправлення конфігураційних дрейфів. 6.2. Підвищення масштабованості системи: - лінійне зростання продуктивності при додаванні нових вузлів;

<table>
<tr><td></td><td></td><td colspan="2">- відсутність bottleneck у вигляді централізованого контролера.

6.3. Покращення показників безпеки:
- динамічне отримання секретів без їх зберігання на диску;
- використання тимчасових токенів Vault з обмеженим терміном дії;
- аудит всіх операцій з конфіденційними даними.</td></tr>
<tr><td rowspan="2">7.</td><td rowspan="2">Стадії і етапи розробки</td><td>Дата</td><td>Назва етапу</td></tr>
<tr><td>від 1 вересня 2024
до 15 вересня 2024

від 16 вересня 2024
до 30 вересня 2024

від 1 жовтня 2024
до 31 жовтня 2024

від 1 листопада 2024
до 30 листопада 2024

від 1 грудня 2024
до 31 грудня 2024

від 1 січня 2025
до 31 січня 2025

від 1 лютого 2025
до 28 лютого 2025

від 1 березня 2025</td><td>Аналіз існуючих систем управління конфігурацією та обґрунтування актуальності pull-based моделі.

Вивчення теоретичних основ Infrastructure as Code, Ansible, HashiCorp Vault та AWS сервісів.

Проектування архітектури pull-based системи управління конфігурацією.

Розробка методики інтеграції Ansible Pull з HashiCorp Vault через AWS IAM аутентифікацію.

Імплементація інфраструктурних компонентів: налаштування</td></tr>
</table>

		до 31 березня 2025 від 1 квітня 2025 до 30 квітня 2025 від 1 травня 2025 до 15 травня 2025 від 16 травня 2025 до 31 травня 2025 від 1 червня 2025 до 20 червня 2025 від 21 червня 2025 до 30 червня 2025	HashiCorp Vault, створення IAM ролей та політик. Розробка Packer образів з попередньо встановленими залежностями для Ansible Pull. Розробка Terraform модулів для автоматичного розгортання EC2 instances з cloud-init конфігурацією. Створення Ansible ролей для автоматичного налаштування systemd служб та таймерів для Ansible Pull. Тестування системи в тестовому середовищі, валідація функціональних вимог. Апробація системи в умовах, наближених до продакшену, оцінка продуктивності та надійності.

			Аналіз результатів, порівняння з традиційною push-based моделлю. Оформлення результатів. Написання пояснювальної записки. Представлення кваліфікаційної роботи магістра керівнику та рецензенту.
8.	Порядок контролю і приймання програмного продукту (моделі)	1. Перевірку ходу розробки системи виконувати раз на 3 тижні на зустрічах з науковим керівником. 2. Проміжний контроль результатів провести на кафедральному семінарі. 3. Захист розробленої системи провести на засіданні Екзаменаційної комісії. 4. Пояснювальну записку подати на паперових носіях в 1 примірнику і в електронному вигляді. 5. Інфраструктурний код (Ansible playbooks, Terraform модулі, Packer конфігурації) надати у вигляді Git-репозиторіїв на електронному носії.	

Виконавець
студент групи КІ-61
Самойленко В. В.
_____

Замовник
PhD
Мороз О. Ю.
_____

Додаток В

Програма і методика випробувань
програмного виробу
«Комп'ютерна модель самоналаштування
virtual machines environment з використанням
Ansible Pull approach»

1 Об'єкт випробувань

1. Назва програмного виробу : «Комп'ютерна модель самоналаштування virtual machines environment з використанням Ansible Pull approach»
2. Галузь застосування : DevOps, Infrastructure as Code, автоматизація інфраструктури.

2. Мета випробувань

Перевірка відповідності функціональності розробленої системи заявленим вимогам у технічному завданні та оцінка переваг децентралізованої моделі порівняно з традиійним push-based підходом.

3. Загальні положення

1. Підстави для проведення випробувань

Підставою для проведення випробувань є наказ про призначення атестаційної комісії.

2. Місце і тривалість випробувань

Приймальні (приймально-здавальні) випробування проводяться на базі комп'ютерного класу кафедри в період роботи атестаційної комісії.

3. Обсяг випробувань

Приймальні випробування програмного вироби проводяться в обсязі відповідному цієї програми і методики випробувань.

4. Організації, які беруть участь у випробуваннях

Приймальні випробування проводяться атестаційною комісією напередодні засідання (або в процесі засідання) за участю Замовника, Виконавця та інших осіб, присутніх на засіданні.

4. Вимоги до програми або програмного виробу

Модель повинна задовольняти наступним вимогам:

1. автоматичне отримання конфігурацій з Git-репозиторіїв;
2. аутентифікація у HashiCorp Vault через AWS IAM без статичних облікових даних;
3. динамічне отримання секретів з Vault;
4. періодичне виконання через systemd таймери;
5. валідація playbook перед застосуванням;
6. логування операцій та обробка помилок.
7. ідемпотентність операцій;
8. автономність роботи вузлів;
9. автоматичне відновлення при збоях.

Спеціальні вимоги (не пред’являються).

5. Вимоги до програмної документації

Програмною документацією до виробу «Комп'ютерна модель самоналаштування virtual machines environment з використанням Ansible Pull approach» вважати:

1. Програмною документацією щодо розроблюваного програмного продукту вважати:
2. справжнє технічне завдання на розробку програми (представити як Додаток Б до пояснювальної записки до кваліфікаційної роботи);
3. Програму і методику випробувань розробленої програми (представити як Додаток В до пояснювальної записки до кваліфікаційної роботи);
4. рекомендацій щодо застосування створеної програмної стандартизації у проектах (представити в Розділі 3 пояснювальної записки до кваліфікаційної роботи).
5. Текст програми(представити як інші додатки до пояснювальної записки до кваліфікаційної роботи).

6. Засоби і порядок випробувань

6.1 Засоби випробувань

- AWS акаунт з IAM ролями;
- HashiCorp Vault кластер (версія 1.8+);
- Git-репозиторії;
- мінімум 2 EC2 instances (t3.micro).
- Terraform 1.0+, Packer 1.7+, Ansible 2.13+, Git 2.0+, AWS CLI 2.0+.

6.2 Порядок проведення випробувань

Тест 1.1. Перевірка комплектності документації

Методика: перевірити наявність всіх додатків та розділів.

Критерій: наявність документації згідно з ТЗ.

Тест 1.2. Перевірка інфраструктури

Методика: перевірити доступність AWS, Vault, Git-репозиторіїв.

Критерій: всі компоненти доступні та функціональні.

Тест 2.1. Розгортання EC2 instance через Terraform

Методика:

```
terraform init
terraform plan
terraform apply -auto-approve
ssh ec2-user@test-instance.example.com
sudo systemctl status ansible-pull.service
○ ansible-pull.service - Run ansible-pull
     Loaded: loaded (/etc/systemd/system/ansible-pull.service; disabled; preset: disabled)
```

```
     Active: inactive (dead) since Wed 2025-11-26 07:09:20 CST; 10h ago
TriggeredBy: ● ansible-pull.timer
    Process: 6708 ExecStart=/opt/ansible/bin/run-ansible-pull.sh (code=exited, status=0/SUCCESS)
   Main PID: 6708 (code=exited, status=0/SUCCESS)
        CPU: 1min 27.062s

Nov 26 07:09:20 test-instance.example.com run-ansible-pull.sh[7674]: PLAY [filebeat]
Nov 26 07:09:20 test-instance.example.com run-ansible-pull.sh[7674]: skipping: no hosts matched
Nov 26 07:09:20 test-instance.example.com run-ansible-pull.sh[7674]: PLAY [all]
Nov 26 07:09:20 test-instance.example.com run-ansible-pull.sh[7674]: PLAY RECAP
Nov 26 07:09:20 test-instance.example.com run-ansible-pull.sh[7674]: test-instance.example.com : ok=90
changed=1    unreachable=0    failed=0    skipped=16   rescued=0    ignored=0
Nov 26 07:09:20 test-instance.example.com run-ansible-pull.sh[7674]: Starting Ansible Pull at 2025-11-26 07:07:38
Nov 26 07:09:20 test-instance.example.com run-ansible-pull.sh[7674]: /usr/local/bin/ansible-pull --clean --url
https://gitlab.com/ansible.git --accept-host-key --checkout m>
Nov 26 07:09:20 test-instance.example.com systemd[1]: ansible-pull.service: Deactivated successfully.
Nov 26 07:09:20 test-instance.example.com systemd[1]: Finished Run ansible-pull.
Nov 26 07:09:20 test-instance.example.com systemd[1]: ansible-pull.service: Consumed 1min 27.062s CPU time.
```

Критерій: instance створений, systemd служба активна.

Тест 2.2. Отримання конфігурацій з Git

Методика:

```
ls -la /opt/ansible/
total 12
drwxr-xr-x 11 ansible wheel 4096 Nov 10 04:41 ansible
drwxr-xr-x  9 ansible wheel 4096 Nov  1 04:15 ansible-inventory
drwxr-xr-x  2 ansible wheel   33 Jun 27 07:48 bin
-rw-r-----  1 ansible wheel 3522 Feb 12  2025 pull-init.yaml
```

Тест 2.3. Динамічне отримання секретів з Vault

Методика: виконати playbook з використанням Vault lookup плагіна, перевірити відсутність секретів на диску.

Критерій: секрети отримані динамічно, не збережені у файлах.

Тест 2.5. Застосування конфігурацій через Ansible Pull

Методика:

```
sudo systemctl start ansible-pull.service
sudo journalctl -u ansible-pull.service -f
```

Критерій: playbook виконується успішно, зміни застосовані ідемпотентно.

Тест 2.6. Періодичне виконання через systemd timer

Методика:

```
sudo systemctl status ansible-pull.timer
sudo systemctl list-timers ansible-pull.timer
NEXT                         LEFT     LAST                         PASSED
UNIT               ACTIVATES
Thu 2025-11-27 04:37:54 CST 10h left Wed 2025-11-26 07:06:34 CST 10h ago
ansible-pull.timer ansible-pull.service
```

Критерій: таймер активний, виконується кожень день.

Висновки: Всі функціональні тести (2.1-2.6) пройдені успішно. Випробування пройшло успішно.

Виконавець: студент групи КІ-61, Самойленко В. В.

Додаток Г.

```
---
- name: Checks
  ansible.builtin.import_tasks: checks.yaml
  tags:
    - never
    - ansible_pull_check
    - ansible_pull_initial_install
    - ansible_pull_daily_install

- name: Install dependencies for Ansible
  become: true
  ansible.builtin.import_tasks: install.yaml
  tags:
    - never
    - ansible_pull_install_dependencies
    - ansible_pull_initial_install
    - ansible_pull_daily_install

- name: Refresh Ansible-inventory git repository
  become: true
  become_user: ansible
  ansible.builtin.import_tasks: git_refresh_ansible_inventory.yaml
  tags:
    - never
    - ansible_pull_refresh_git_ansible_inventory
    - ansible_pull_initial_install
    - ansible_pull_daily_install

- name: Refresh Ansible git repository
  become: true
  become_user: ansible
  ansible.builtin.import_tasks: git_refresh_ansible.yaml
  tags:
    - never
    - ansible_pull_refresh_git_ansible
    - ansible_pull_initial_install
    - ansible_pull_daily_install

- name: Export initial env vars
  ansible.builtin.import_tasks: export_env_initial.yaml
  tags:
    - never
    - ansible_pull_export_initial_tags
    - ansible_pull_initial_install

- name: Export daily env vars
  ansible.builtin.import_tasks: export_env_daily.yaml
  tags:
    - never
    - ansible_pull_export_daily_tags
    - ansible_pull_daily_install

- name: Install systemd services
  ansible.builtin.import_tasks: install_systemd_service.yaml
  tags:
    - never
    - ansible_pull_install_systemd
    - ansible_pull_daily_install
```

Додаток Д.

```
packer {
  required_version = "~> 1.8"
  required_plugins {
    amazon = {
      version = "~> v1.0"
      source  = "github.com/hashicorp/amazon"
    }
  }
}

locals {
  timestamp    = formatdate("YYYY-MM-DD-hhmm", timestamp())
  deprecate_at = formatdate("YYYY-MM-DD-hhmm", timeadd(timestamp(), "35040h"))
}

#########################
# AlmaLinux OS 9 x86_64 #
#########################

data "amazon-ami" "almalinux_9_x86_64" {
  filters = {
    name                = "AlmaLinux OS 9.6*"
    architecture        = "x86_64"
    root-device-type    = "ebs"
    virtualization-type = "hvm"
  }
  most_recent = true
  owners      = ["123456789012"]
}

source "amazon-ebs" "almalinux_9_x86_64" {
  ami_name        = "Golden AMI AlmaLinux OS 9.6 x86_64 ${local.timestamp}"
  ami_description = "Golden AMI AlmaLinux 9.6 x86_64"
  ami_users       = var.aws_destination_accounts
  ami_regions     = var.aws_destination_regions
  instance_type   = "t3.small"
  access_key      = var.aws_access_key
  region          = var.aws_region
  secret_key      = var.aws_secret_key
  token           = var.aws_session_token
  source_ami      = data.amazon-ami.almalinux_9_x86_64.id
  ssh_username    = "ec2-user"
  tags = {
    Base_AMI_Name = "{{ .SourceAMIName }}"
    Build         = local.timestamp
    DeprecateAt   = local.deprecate_at
    Environment   = "test.dev"
    Name          = "Golden AMI AlmaLinux OS 9.6 x86_64 ${local.timestamp}"
    OS            = "AlmaLinux"
    OS_Version    = "9"
  }
}

########################
# AlmaLinux OS 9 arm64 #
########################

data "amazon-ami" "almalinux_9_arm64" {
  filters = {
    name                = "AlmaLinux OS 9.6*"
    architecture        = "arm64"
    root-device-type    = "ebs"
    virtualization-type = "hvm"
```

```
  }
  most_recent = true
  owners      = ["764336703387"]
}

source "amazon-ebs" "almalinux_9_arm64" {
  ami_name        = "Golden AMI AlmaLinux OS 9.6 arm64 ${local.timestamp}"
  ami_description = "Golden AMI AlmaLinux 9.6 arm64"
  ami_users       = var.aws_destination_accounts
  ami_regions     = var.aws_destination_regions
  instance_type   = "t4g.small"
  access_key      = var.aws_access_key
  region          = var.aws_region
  secret_key      = var.aws_secret_key
  token           = var.aws_session_token
  source_ami      = data.amazon-ami.almalinux_9_arm64.id
  ssh_username    = "ec2-user"
  tags = {
    Base_AMI_Name = "{{ .SourceAMIName }}"
    Build         = local.timestamp
    DeprecateAt   = local.deprecate_at
    Environment   = "test.dev"
    Name          = "Golden AMI AlmaLinux OS 9.6 arm64 ${local.timestamp}"
    OS            = "AlmaLinux"
    OS_Version    = "9"
  }
}

build {
  name = "almalinux_9"
  sources = [
    "source.amazon-ebs.almalinux_9_x86_64",
    "source.amazon-ebs.almalinux_9_arm64",
  ]

  provisioner "shell" {
    execute_command = "sudo {{ .Path }}"
    script          = "${path.root}/scripts/configuration.sh"
  }

  provisioner "file" {
    source      = "${path.root}/files/motd.sh"
    destination = "/tmp/motd.sh"
  }

  provisioner "file" {
    source      = "${path.root}/../../ansible//roles/ansible_pull/files"
    destination = "/tmp"
  }

  provisioner "shell" {
    execute_command = "sudo {{ .Path }}"
    script          = "${path.root}/scripts/configure_motd.sh"
  }

  provisioner "shell" {
    execute_command = "sudo {{ .Path }}"
    script          = "${path.root}/scripts/configure_ansible.sh"
  }

}
```

Додаток Е.

```
#!/bin/bash

set -e
PYTHON_VERSION="3.12"
ANSIBLE_HOME="/opt/ansible"
FILES_DIR="/tmp/files"

# Install required dependencies
dnf config-manager --add-repo
https://rpm.releases.hashicorp.com/RHEL/hashicorp.repo
dnf install -y $(cat $FILES_DIR/requirements.dnf)
/usr/bin/python${PYTHON_VERSION} -m ensurepip
/usr/bin/python${PYTHON_VERSION} -m pip install -r
$FILES_DIR/requirements.txt
/usr/local/bin/ansible-galaxy collection install -p
/usr/share/ansible/collections -r $FILES_DIR/requirements.yaml

# Disable SELINUX
sed -c -i "s/\SELINUX=.*/SELINUX=disabled/" /etc/sysconfig/selinux
setenforce 0

# Create 'ansible' user with specified properties
useradd -r -s /sbin/nologin -d $ANSIBLE_HOME -g wheel ansible

# Set up sudo for the 'ansible' user without requiring a password
echo "ansible ALL=(ALL) NOPASSWD:ALL" >
/etc/sudoers.d/ansible-pull-service
chmod 440 /etc/sudoers.d/ansible-pull-service

# Ensure the $ANSIBLE_HOME directory exists and set ownership
mkdir -p $ANSIBLE_HOME
chown -R ansible:wheel $ANSIBLE_HOME

rm -rf $FILES_DIR
```

Додаток Ж.

```
#!/usr/bin/env bash

set -euo pipefail

# set hostname to host
cat /dev/null > /var/log/cloud-init-output.log
echo -e "########## \n\n ${node_name} host ${hostname}\n\n##########" > /etc/motd
hostnamectl set-hostname "${hostname}"

cat <<EOF > /opt/ansible/ansible-pull-init.yaml
- hosts: localhost
  connection: local
  become: yes
  become_user: ansible
  tasks:
    - name: Refresh git repositories
      block:
        - name: Gather EC2 metadata facts
          amazon.aws.ec2_metadata_facts: {}

        - name: Gather EC2 instance tags
          amazon.aws.ec2_tag_info:
            resource: "{{ansible_ec2_instance_id}}"
            region: "{{ansible_ec2_placement_region}}"
          register: ec2_instance_tags

        - name: Set env variables
          ansible.builtin.set_fact:
            env_name: "{{ ec2_instance_tags.tags['Environment'].split('.')[0] }}"
            env_class: "{{ ec2_instance_tags.tags['Environment'].split('.')[1] }}"

        - name: Set env variables
          ansible.builtin.set_fact:
            env_vault_url: "https://vault-{{env_name}}.{{env_class}}.com"

        - name: Create .env_vars
          ansible.builtin.copy:
            content: |
              ENV_CLASS="{{ env_class }}"
              ENV_NAME="{{ env_name }}"
            dest: "/opt/ansible/.env_vars"
            owner: ansible
            group: wheel
            mode: 0644

        - name: Retrieve login token from env Vault
          community.hashi_vault.vault_login:
            url: "{{ env_vault_url }}"
            auth_method: aws_iam
            aws_iam_server_id: "{{ env_vault_url }}"
            role_id: "{{env_name}}-{{env_class}}-role-iam"
            validate_certs: false
          register: vault_login_data
          no_log: true

        - name: Gather git secrets
          community.hashi_vault.vault_read:
            url: "{{ env_vault_url }}"
            auth_method: token
            token: "{{ vault_login_data.login.auth.client_token }}"
            path: "{{env_class}}/{{env_name}}/ansible-pull"
          register: vault_approles_data
          no_log: true
```

```
        - name: Set git secrets
          ansible.builtin.set_fact:
            git_password: "{{ vault_approles_data.data.data.git_password | default('undefined') }}"
            git_username: "{{ vault_approles_data.data.data.git_username | default('undefined') }}"
            git_hostname: "{{ vault_approles_data.data.data.git_hostname | default('undefined') }}"
          no_log: true

        - name: Set git config
          ansible.builtin.git_config:
            name: credential.helper
            value: store
            scope: global

        - name: Set git-credentials
          become: true
          become_user: ansible
          ansible.builtin.copy:
            content: "https://{{ git_username | urlencode }}:{{ git_password | urlencode }}@{{ git_hostname }}"
            dest: /opt/ansible/.git-credentials
            owner: ansible
            group: wheel
            mode: 0600

        - name: Clone git repositories
          ansible.builtin.git:
            repo: "https://{{ git_hostname }}/{{ item.git_path }}"
            dest: "{{ item.dest }}"
            force: true
            update: yes
            version: "{{ item.git_branch }}"
          with_items:
            - { git_path: "ansible.git", dest: "/opt/ansible/ansible", git_branch: "master" }
            - { git_path: "ansible-inventory.git", dest: "/opt/ansible/ansible-inventory", git_branch: "master" }

      always:
        - name: Remove git-credentials
          ansible.builtin.file:
            path: /opt/ansible/.git-credentials
            state: absent
EOF

chown ansible:wheel /opt/ansible/ansible-pull-init.yaml && chmod 0640 /opt/ansible/ansible-pull-init.yaml

/usr/local/bin/ansible-playbook /opt/ansible/gtv-pull-init.yaml
. /opt/ansible/.env_vars && /usr/local/bin/ansible-playbook -vv /opt/ansible/ansible/main.yaml -i
/opt/ansible/ansible-inventory/$${ENV_CLASS}/$${ENV_NAME} -e ansible_connection=local -e ansible_pull=true
--limit $(hostname) --tags 'ansible_pull_initial_install'
sudo -u ansible bash -c '. /opt/ansible/.env_vars && /usr/local/bin/ansible-pull -vv --clean --accept-host-key
--url https://$${GIT_HOSTNAME}/ansible.git --checkout $${ANSIBLE_GIT_BRANCH} -i
/opt/ansible/ansible-inventory/$${ENV_CLASS}/$${ENV_NAME} --tags $${ANSIBLE_INITIAL_TAGS} -e
ansible_connection=local -e ansible_pull=true --limit $(hostname) main.yaml; rm -f /opt/ansible/.git-credentials'
. /opt/ansible/.env_vars && /usr/local/bin/ansible-playbook -vv /opt/ansible/ansible/main.yaml -i
/opt/ansible/ansible-inventory/$${ENV_CLASS}/$${ENV_NAME} -e ansible_connection=local -e ansible_pull=true
--limit $(hostname) --tags 'ansible_pull_daily_install'
# Restart only if needs-restarting exits with 1
dnf needs-restarting -r || reboot
```

Додаток 3.

```
---
- name: Refresh git repositories
  block:
    - name: Set credential.helper git config to 'store' globally
      community.general.git_config:
        name: credential.helper
        value: store
        scope: global

    - name: Add a safe.directory to git config globally
      community.general.git_config:
        name: safe.directory
        value: "{{ item.dest }}"
        scope: global
        add_mode: add
      with_items:
        - { dest: "/opt/ansible/ansible" }
        - { dest: "/opt/ansible/ansible-inventory" }

    - name: Set git-credentials
      ansible.builtin.copy:
        content: "https://{{ ansible_pull_git_username | urlencode }}:{{
ansible_pull_git_password | urlencode }}@{{ ansible_pull_git_hostname }}"
        dest: /opt/ansible/.git-credentials
        owner: ansible
        group: wheel
        mode: "0600"

    - name: Clone git repositories
      ansible.builtin.git:
        repo: "https://{{ ansible_pull_git_hostname }}/{{ item.git_path }}"
        dest: "{{ item.dest }}"
        force: true
        update: true
        version: "{{ item.git_branch }}"
      with_items:
        - { git_path: "ansible-inventory.git", dest: "/opt/ansible/ansible-inventory",
git_branch: "master" }

  always:
    - name: Remove git-credentials
      ansible.builtin.file:
        path: /opt/ansible/.git-credentials
        state: absent
      when: "'ansible_pull_initial_install' not in ansible_run_tags"
```

Додаток І.

```
#!/usr/bin/env bash

set -euo pipefail

# Get parent directory path
cd "${0%/*}/.." || exit
BASE_DIR=$(pwd)

SCRIPT_ORIG_MD5=$(md5sum "$0")
SCRIPT_RESTART=${SCRIPT_RESTART:-""} # Retain SCRIPT_RESTART if already set
ANSIBLE_HOME="/opt/ansible"

# set ENV_CLASS and ENV_NAME. This file is created at startup by cloud-init
. $ANSIBLE_HOME/.env_vars
# Run setup in-case something changed

/usr/local/bin/ansible-playbook $ANSIBLE_HOME/ansible/main.yaml -i
$ANSIBLE_HOME/ansible-inventory/${ENV_CLASS}/${ENV_NAME} -e ansible_connection=local -e ansible_pull=true --limit
"$(hostname)" --tags 'ansible_pull_check,ansible_pull_refresh_git_ansible_inventory'
/usr/local/bin/ansible-playbook $ANSIBLE_HOME/ansible/main.yaml -i
$ANSIBLE_HOME/ansible-inventory/${ENV_CLASS}/${ENV_NAME} -e ansible_connection=local -e ansible_pull=true --limit
"$(hostname)" --tags 'ansible_pull_refresh_git_ansible'
/usr/local/bin/ansible-playbook $ANSIBLE_HOME/ansible/main.yaml -i
$ANSIBLE_HOME/ansible-inventory/${ENV_CLASS}/${ENV_NAME} -e ansible_connection=local -e ansible_pull=true --limit
"$(hostname)" --tags 'ansible_pull_daily_install'
# set new env
. $ANSIBLE_HOME/.env_vars

# If script changed, restart
SCRIPT_NEW_MD5=$(md5sum "$0")
if [[ -z $SCRIPT_RESTART && "$SCRIPT_ORIG_MD5" != "$SCRIPT_NEW_MD5" ]]; then
  echo "Script changes detected,e restarting"
  export SCRIPT_RESTART=1
  "${0}"; exit
fi

export VAULT_TOKEN=$(vault login -method=aws -token-only -address https://vault-${ENV_NAME}.${ENV_CLASS}.test.com
header_value=https://vault-${ENV_NAME}.${ENV_CLASS}.test.com role=${ENV_NAME}-${ENV_CLASS}-role-iam)
GIT_USERNAME=$(vault read -field=git_username -address https://vault-${ENV_NAME}.${ENV_CLASS}.test.com
${ENV_CLASS}/${ENV_NAME}/ansible-pull )
GIT_PASSWORD=$(vault read -field=git_password -address https://vault-${ENV_NAME}.${ENV_CLASS}.test.com
${ENV_CLASS}/${ENV_NAME}/ansible-pull )
echo "https://$GIT_USERNAME:$GIT_PASSWORD@${GIT_HOSTNAME}" >> $ANSIBLE_HOME/.git-credentials
unset VAULT_TOKEN

ANSIBLE_CALLBACKS_ENABLED=${ANSIBLE_CALLBACK} SUMOLOGIC_URL=${SUMOLOGIC_ENV_URL} \
/usr/local/bin/ansible-pull --clean --url https://${GIT_HOSTNAME}/ansible.git \
--accept-host-key \
--checkout ${ANSIBLE_GIT_BRANCH} \
-i $ANSIBLE_HOME/ansible-inventory/${ENV_CLASS}/${ENV_NAME} \
--tags ${ANSIBLE_DAILY_TAGS} \
-e ansible_connection=local \
-e ansible_pull=true \
--limit "$(hostname)" main.yaml; rm -f $ANSIBLE_HOME/.git-credentials
```

Додаток К.

УДК (UDC) 004.415.2

Dmitry Bulavin *PhD, associate professor*
Kharkiv National University named V. N. Karazin 61000
e-mail: d.bulavin@karazin.ua
ORCID: 0000-0002-4840-4763
Vladyslav Samoilenko *Student*
Kharkiv National University named V. N. Karazin 61000
e-mail: vladyslav.samoilenko@student.karazin.ua;
ORCID: 0009-0001-3217-4473

Self-Configuring Virtual Machine Environments Using Ansible Pull: A Review of Approaches and Challenges

The automation of virtual machine (VM) configuration has become an integral part of modern IT infrastructure, ensuring compliance with the high demands of scalability, reliability, and security. With the increasing adoption of cloud technologies, there is a growing need for standardized approaches to infrastructure management that eliminate the limitations of traditional methods, such as manual configuration or individual scripts. The goal of this research is to design an architecture for automated VM self-configuration based on Ansible Pull in combination with tools like Terraform, Hashicorp Vault, and Git to ensure flexibility, autonomy, and security.

To build the system, an analytical approach was used to evaluate existing configuration tools, model the architecture using Git repositories, Terraform, and Vault, and experimentally implement cloud-init to initialize Ansible Pull at the VM level. The proposed architecture automates the VM configuration process, ensuring scalability and reducing dependence on centralized management servers. The implementation of Ansible Pull with a tagging system provides idempotent task execution, regular updates, and maintenance of the desired system state.

It has been proven that combining Ansible Pull with Terraform, Git, and Vault ensures simplicity of implementation, flexibility in scaling, and secure configuration management. The proposed approach is effective for dynamic environments requiring frequent updates and aligns with modern DevOps practices.

Keywords: *Ansible Pull, automation of the configuration, Terraform, Vault, self-configuring, DevOps, Ansible*

Як цитувати: Булавін Д. О., Самойленко В. В., «Самостійне конфігурування середовищ віртуальних машин за допомогою Ansible Pull: огляд підходів та проблем». *Вісник Харківського національного університету імені В.Н.Каразіна, сер. «Математичне моделювання. Інформаційні технології. Автоматизовані системи управління»*. 2025. т. XX. С.X-XX. https://doi.org/10.26565/2304-6201-2022-53-01

How to quote: D.O. Bulavin, V.V. Samoilenko, “Self-Configuring Virtual Machine Environments Using Ansible Pull: A Review of Approaches and Challenges” *Bulletin of V.N. Karazin Kharkiv National University, series “Mathematical modelling. Information technology. Automated control systems*, vol. XX, pp. XX-XX, 2025. https://doi.org/10.26565/2304-6201-2022-53-01

1 Introduction

In the modern world of information technology, process automation is a key factor in enhancing efficiency and reducing operational costs. One of the primary tasks in this context is the automation of virtual machine (VM) configuration, which constitutes a vital part of the infrastructure for most organizations. Traditional methods of manual configuration are not only labor-intensive but are also prone to human error, potentially leading to severe system failures.

Automation tools such as Ansible, Puppet, and Chef provide a means to simplify this process by employing a declarative approach. Among these, Ansible stands out for its simplicity, idempotency (ensuring consistent execution results), support for YAML files, and agentless architecture. These features make Ansible an effective tool for managing large-scale server infrastructures in cloud environments, such as AWS, Google Cloud, and Azure.

This study focuses on exploring the capabilities of Ansible Pull as a method for self-organizing virtual machine environments. The emphasis is placed on configuration management, performance optimization, and cost reduction when employing this approach. In addition to analyzing the advantages of Ansible Pull, the paper examines key aspects of deployment process management, including comparisons with similar solutions. The primary challenges associated with integrating Ansible into scalable environments are identified, and strategies for overcoming these challenges are proposed.

Thus, this work aims to establish a foundation for further research and the implementation of automated solutions for virtual machine configuration in modern IT environments.

2 Analysis of the Problem of Self-Configuring Virtual Environments

2.1 Analysis of Modern Methods and Their Limitations

Traditionally, self-configuration is an automated process through which a system configures itself to operate by following predefined instructions without requiring manual intervention. This process includes:

- Installation and management of software dependencies.
- Creation and configuration of user accounts with appropriate access rights.
- Configuration of network settings and ensuring their security.
- Monitoring and ensuring compliance with predefined standards.

The goal of self-configuration is to minimize the risk of human error, accelerate deployment cycles, and ensure the scalability and repeatability of the infrastructure configuration process. This is particularly important for environments that require frequent updates or have complex distributed configurations.

Today, there are several tools and approaches to configuration automation, each with its own advantages and disadvantages:

1. **Custom Scripts:**
   - These are easy to create but poorly scalable and challenging to maintain in dynamic environments. Updates often require manual intervention, increasing the risk of errors.
2. **Configuration Management Tools (e.g., Puppet, Chef):**
   - These tools are powerful but often require the installation of specialized agents on each node, complicating their use and increasing resource consumption.
3. **Ansible Push:**
   - Ansible's push model eliminates the need for agents, simplifying configuration management. However, this approach can create bottlenecks when managing a large number of nodes due to the centralized execution of commands.
4. **Ansible Pull:**
   - Ansible Pull offers a decentralized approach where managed nodes retrieve configurations from a Git repository. This approach ensures scalability and reduces dependency on a central control node but comes with its own challenges:
   - **Dependency on Git:** Any issues with the repository can halt configuration updates.
   - **Network Stability:** Requires a stable SSH connection for initial setup and updates.
   - **Version Management:** Ensuring consistent configurations across all nodes can be challenging without proper control.

Thus, Ansible is not the only tool in the configuration automation space. Several other leading tools predate Ansible's release in 2012: SaltStack in 2011, Puppet in 2005, and Chef in 2008. A comparison of the popularity of search queries among these tools is shown in Figure 2.1.

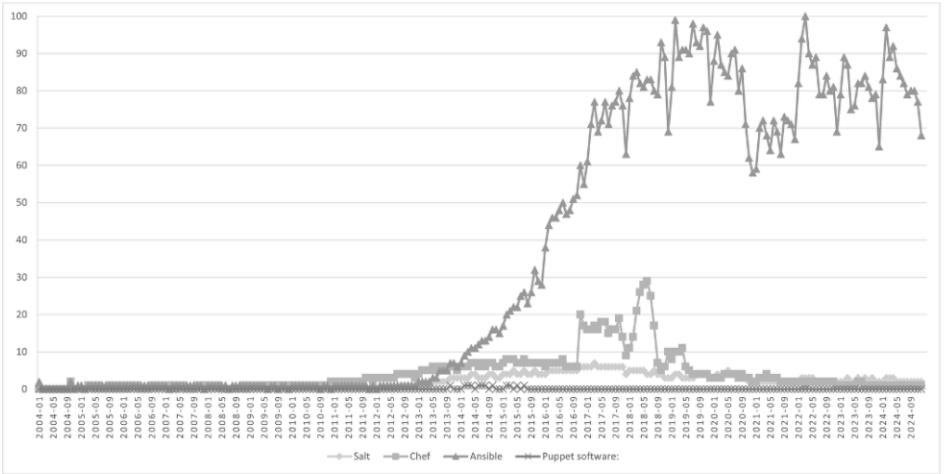

Fig.2.1 Comparing of the search requests Salt, Chef, Ansible ma Puppet popularity in Google.

The choice of tools and methods for self-configuration depends on specific requirements such as scalability, ease of use, and integration capabilities. With Ansible Pull, self-configuration becomes a robust, scalable, and repeatable process that ensures the consistent provisioning of systems according to desired specifications described in code. Storing configurations as code provides significant flexibility and simplifies maintenance.

2.2 Ansible Pull Analysis: Advantages and Disadvantages

Ansible is a popular tool in the category of Infrastructure-as-Code (IaC) that enables automation of deployment and configuration of IT infrastructure. Its agentless architecture, intuitive YAML syntax, and scalability have made Ansible a top choice for organizations seeking to improve process efficiency. Ansible developers create playbooks containing a series of tasks that can be automatically executed on a set of hosts to achieve the desired infrastructure state.

As an automation technology, Ansible is based on the following principles:

- **Agentless Architecture.** Ansible does not require installing software on managed nodes (except in the pull model).
- **Simplicity.** YAML-based playbooks allow easy creation and comprehension of automation scripts. Ansible is also decentralized, using SSH with existing OS credentials to access remote machines.
- **Scalability and Flexibility.** Its modular design allows for rapid scaling, supporting a wide range of platforms and environments.
- **Error-Free, Predictable, and Idempotent.** Ansible files describe infrastructure as code, ensuring the same outcome regardless of how many times the playbooks are executed. [1]

Ansible Pull offers a unique model for self-configuring environments, where managed nodes independently fetch and apply configurations from a repository. This approach enables:

- Avoiding dependence on a central control node.
- Reducing risks associated with single points of failure.
- Efficient operation in distributed infrastructure environments.

Ansible Pull is ideal for dynamic environments that require regular updates, meeting the demands of modern agile systems. [2]

Ansible Pull utilizes a decentralized configuration management method where each node independently fetches and applies updates from a central Git repository. This approach eliminates the need for a centralized control server, enhancing system fault tolerance. Nodes periodically execute the ansible-pull command to download and apply the latest playbooks. Additionally, playbooks can include tasks to update Ansible Pull itself, increasing the autonomy of the process.

Red Hat, the developer of Ansible, provides tools to implement this model but not out-of-the-box solutions. Automating the execution of ansible-pull requires the use of task schedulers such as cron jobs, systemd timers, or similar tools [3]. This modular approach is characteristic of Ansible, offering flexible tools rather than rigid solutions.

Advantages:

1. Autonomy: Each node fetches and applies its configuration independently, minimizing reliance on a central control server and ensuring resilience against network disruptions or server downtime.
2. Scalability: Adding new nodes is simplified since the self-configuration process is integrated into deployment.
3. Flexibility: Changes in the repository are automatically applied during the next execution cycle of ansible-pull, ensuring prompt and consistent updates across all nodes.
4. Version Control: Git integration provides robust version management, allowing effective collaboration, tracking changes, and reverting to previous configurations when needed.
5. No Single Point of Failure: Eliminating reliance on a central server reduces risks associated with potential unavailability.

Disadvantages:

1. Network Dependency: Nodes require connectivity to the Git repository for updates. This can be mitigated by setting up local repository mirrors or using caching mechanisms.
2. Lack of Centralized Management: Unlike the push model, task execution control is more complex, requiring additional monitoring tools such as Ansible callback modules or third-party systems like Prometheus or ELK Stack.

3. Error Debugging: Error logs are stored locally on each node, complicating centralized troubleshooting, especially in large-scale environments. Integration with centralized log collection systems can address this issue.
4. Initial Node Configuration: For ansible-pull to work, each node must be pre-configured with access to the Git repository and necessary dependencies. While this adds steps to configuration, it is a common practice in decentralized systems.
5. Security Challenges: Granting nodes access to the Git repository can pose risks related to data leakage or unauthorized access. Recommendations include restricting access rights, using SSH keys, and secure communication channels (e.g., VPN).

Ansible Pull is a powerful solution for decentralized configuration management, offering autonomy and flexibility. However, its implementation requires additional setup for monitoring, security, and debugging. When properly configured, the system demonstrates high scalability and reliability.

2.3 Comparative Analysis of Existing Solutions

We compare popular configuration management systems such as Chef, Puppet, SaltStack, and Ansible Pull based on key criteria: autonomy, flexibility, security, ease of updates, and scalability to highlight their strengths and weaknesses.

Chef: Uses a pull model where nodes fetch configurations (cookbooks) from a central server. While effective for managing complex configurations, its reliance on the Ruby programming language and the need for a dedicated central server complicates setup and usage.

Puppet: Operates similarly, with nodes fetching manifests from a Puppet master. It supports scalability and consistency but requires installing the Puppet agent and master, increasing complexity. Its domain-specific language (Puppet DSL) reduces accessibility for teams unfamiliar with it.

SaltStack: Primarily uses a push model with a master-minion architecture. While SaltStack offers significant scalability, its setup and usage are complex, though it is compatible with YAML.

Ansible Pull: Unlike the others, Ansible Pull eliminates the need for centralized servers or agents. Nodes independently fetch configurations from a Git repository, making it lightweight and straightforward. Its YAML-based approach and Git integration align with modern DevOps practices, ensuring an intuitive process.

Key Comparative Insights:

- Autonomy: Ansible Pull ensures node autonomy, as each node independently manages updates. Chef and Puppet also support autonomous operation but rely on central servers, creating a potential single point of failure.
- Flexibility: Ansible Pull offers high flexibility with its Git-based workflow, facilitating seamless integration with existing tools and processes. Salt is versatile but more complex to configure, whereas Chef and Puppet require specialized languages or infrastructure, limiting flexibility in some environments.
- Security: Chef and Puppet provide robust authentication and access control through master-agent models. Ansible Pull leverages Git for security, requiring careful management of credentials and repository access control. Salt also provides high security but is harder to configure in pull mode.
- Ease of Updates: Periodic Ansible Pull synchronization with Git ensures quick and consistent updates. Puppet and Chef support automatic updates but with potential delays. Salt's push model provides real-time updates but loses this advantage in pull mode.
- Scalability: Puppet and Chef scale well due to their master-agent architecture. Ansible Pull is suitable for distributed systems as nodes operate independently. However, large-scale deployments can strain Git traffic during updates, which can be mitigated through update scheduling. Salt is efficient in push scenarios but less performant in pull mode.

Ansible Pull is an excellent choice for environments favoring lightweight, decentralized, and straightforward self-configuration processes. It is especially effective where teams use Git, which was adopted by approximately 93% of developers in 2023 [4]. However, environments requiring highly centralized control or low-latency updates may benefit from solutions like Puppet, Chef, or Salt. Ultimately, Ansible Pull is ideal for organizations seeking scalability, flexibility, efficiency, and seamless integration with modern DevOps workflows.

3 Description of the Self-Configuring Virtual Machine Environments solution

3.1 Architecture

After analyzing all possible approaches to architecture design, we propose a solution distinguished by enhanced flexibility and efficiency.

The proposed architecture utilizes several Git repositories to organize and manage different aspects of the system:

- Ansible Repository: Contains roles and corresponding configurations for automation.
- Ansible-Inventory Repository: Includes inventory files necessary for accurate system configuration.
- Terraform Repository: Stores infrastructure-as-code (IaC) definitions, enabling automated infrastructure creation and management.
- Terraform-Modules Repository: Used for reusable modules to enhance scalability and code reusability.

To manage secrets, Hashicorp Vault is integrated, providing a high level of security. EC2 instances are configured to authenticate with Vault using AWS IAM roles, ensuring secure retrieval of secrets and facts required for configuration processes.

This architecture ensures:

- Separation of responsibilities through specialized repositories.
- Secure handling of sensitive data.
- Scalability and flexibility through the use of Terraform modules.

An example of the proposed architecture is shown in Fig. 3.1.

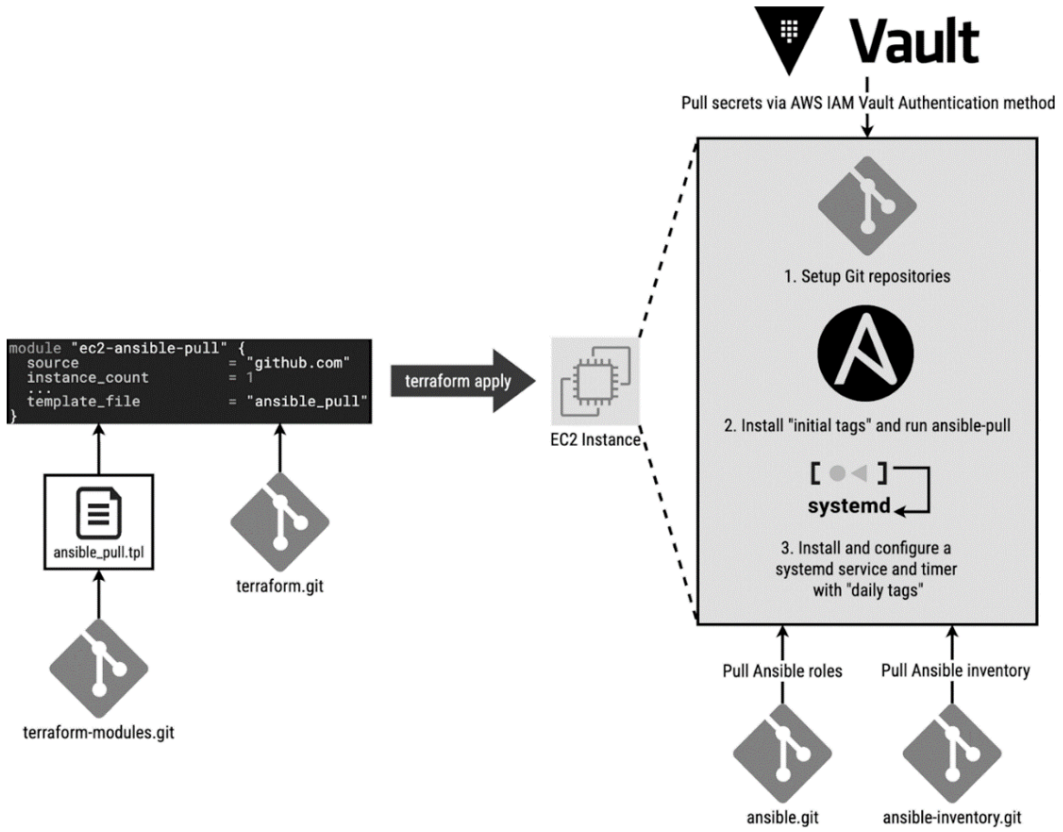

Fig.3.1. The overall architecture of the Ansible Pull solution using Terraform, Ansible, Git, and Vault

When an EC2 virtual machine instance is created using Terraform, a cloud-init script embedded in the configuration is executed during its launch. Cloud-init facilitates the connection between VM deployment and configuration without requiring additional user intervention [5]. This script plays a pivotal role in initializing the Ansible Pull model, retrieving secrets, and preparing the VM environment for automated configuration. The architecture seamlessly integrates with AWS services, delivering a scalable and secure self-configuring solution.

Environment Configuration:

- **Git Repository Setup:** The ansible repository contains playbooks and roles, while the ansible-inventory repository holds inventory configurations. The terraform and terraform-modules repositories organize infrastructure code.
- **Vault Setup:** Vault is configured to securely store secrets, ensuring EC2 instances can authenticate using AWS IAM roles [6].

Provisioning and Initialization:

- **Terraform Deployment:** Terraform creates EC2 instances and configures them using a cloud-init script.
- **Dependency Installation:** Dependencies such as Git, Python, Ansible, Ansible Collections, and Pip modules are installed.
- **Repository Cloning:** The ansible and ansible-inventory repositories are cloned onto the VM.
- **Environment Preparation:** The ansible-pull role prepares the environment by fetching the latest updates and applying the necessary configurations.

Self-Configuration:

- The ansible-pull command is executed with specific tags to configure the VM based on its purpose (e.g., database server).
- Following initial configuration, the Ansible Pull role sets up a systemd service to execute the ansible-pull command daily with "daily" tags, ensuring configurations remain up to date.

3.2 Use Cases

Database Server Configuration: using database-specific tags, Ansible Pull installs and configures database software, such as MySQL. Credentials and access permissions are securely retrieved from Vault, and monitoring and logging systems are initialized. Daily pulls ensure configurations remain intact, and security settings are consistently reapplied.

Application Server Configuration: application servers are configured to install runtime dependencies, deploy application code, and set environment variables using secrets from Vault. Daily execution of Ansible Pull ensures the application environment remains consistent and dependencies stay updated.

General Maintenance: a daily system service verifies critical configurations such as SSH settings and dependency versions, applies updates from the Ansible repository, and ensures the system maintains its desired state without manual intervention.

This architecture allows organizations to establish a fault-tolerant and automated environment for managing virtual machines, ensuring consistency, scalability, and security across their infrastructure.

3.3 Key metrics

Ansible Pull can be evaluated using several key metrics, including setup time, failure rate, and maintenance costs. The setup time for Ansible Pull is minimal compared to traditional push-based configuration management systems, as it enables nodes to autonomously fetch configurations without requiring centralized orchestration. The failure rate is typically low, provided proper error-handling mechanisms are included in the playbooks.

Ansible Pull avoids the network bottlenecks and scalability issues inherent in push models. Maintenance costs are reduced because nodes are self-sufficient, decreasing the need for continuous monitoring or administrator intervention. In contrast, Ansible Push requires a control node and is less efficient in large-scale environments, where connectivity issues can disrupt processes. Such a control node may lack centralization, complicating dependency unification and management.

Comparatively, tools like Puppet or Chef offer similar performance but often require more complex setup and incur ongoing licensing or support costs. Research comparing Ansible, Chef, SaltStack, and Puppet in configuration management highlights metrics such as deployment speed, reliability, and ease of use.

Roman Kostromin conducted a detailed analysis of these tools in heterogeneous distributed computing environments, emphasizing that Ansible outperforms others in simplicity and agentless architecture, reducing setup complexity and human errors. While Chef and Puppet excel in state management and scalability, they demand higher configuration and maintenance effort [7]. Moreover, a 2022 study noted that Ansible's reliance on SSH infrastructure and its agentless nature make it a secure and efficient choice for configuration management, particularly for small to medium-sized deployments, while Chef and Puppet are more suitable for complex infrastructures with frequent configuration changes [8].

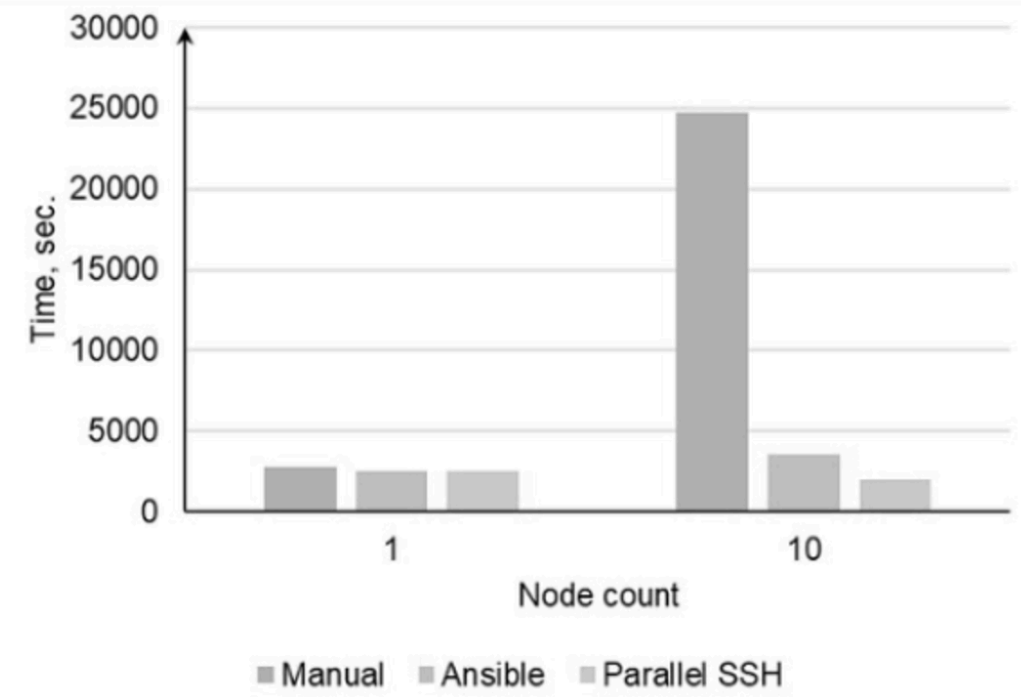

Fig.3.2. Comparison of Node Configuration Time: Manual Setup, Ansible, and Parallel SSH Connections [8]

Ansible Pull is a streamlined, autonomous solution for self-configuring virtual environments, particularly in scenarios where simplicity, scalability, and cost-efficiency are key priorities. Its decentralized model reduces dependence on a central control node, providing enhanced resilience and deployment flexibility.

ЛІТЕРАТУРА

1. Introduction to ansible. Introduction to Ansible - Ansible Community Documentation. (2024, November 20). https://docs.ansible.com/ansible/latest/getting_started/introduction.html
2. Ansible concepts. Ansible concepts - Ansible Community Documentation. (2024, November 20). https://docs.ansible.com/ansible/latest/getting_started/basic_concepts.html#id1
3. ansible-pull – Ansible Community Documentation. Ansible Documentation. URL: https://docs.ansible.com/ansible/latest/cli/ansible-pull.html (date of access: 08.12.2024)
4. Beyond Git: The other version control systems developers use - Stack Overflow. The Stack Overflow Blog - Stack Overflow. URL: https://stackoverflow.blog/2023/01/09/beyond-git-the-other-version-control-systems-developers-use/ (date of access: 15.12.2024).
5. cloud-init 24.4 documentation. cloud-init 24.4 documentation. URL: https://cloudinit.readthedocs.io/en/latest/ (date of access: 18.12.2024).
6. AWS - Auth Methods | Vault | HashiCorp Developer. AWS - Auth Methods | Vault | HashiCorp Developer. URL: https://developer.hashicorp.com/vault/docs/auth/aws (date of access: 18.12.2024).
7. Kostromin, R. O. (2020). Survey of software configuration management tools of nodes in heterogeneous distributed computing environment. The International Workshop on Information, Computation, and Control Systems for Distributed Environments. https://doi.org/10.47350/iccs-de.2020.15
8. S, L. (2022). Automation of server configuration using Ansible. International Journal for Research in Applied Science and Engineering Technology, 10(6), 4109–4113. https://doi.org/10.22214/ijraset.2022.44840

REFERENCES

1. Introduction to ansible. Introduction to Ansible - Ansible Community Documentation. (2024, November 20). https://docs.ansible.com/ansible/latest/getting_started/introduction.html
2. Ansible concepts. Ansible concepts - Ansible Community Documentation. (2024, November 20). https://docs.ansible.com/ansible/latest/getting_started/basic_concepts.html#id1
3. ansible-pull – Ansible Community Documentation. Ansible Documentation. URL: https://docs.ansible.com/ansible/latest/cli/ansible-pull.html (date of access: 08.12.2024)
4. Beyond Git: The other version control systems developers use - Stack Overflow. The Stack Overflow Blog - Stack Overflow. URL: https://stackoverflow.blog/2023/01/09/beyond-git-the-other-version-control-systems-developers-use/ (date of access: 15.12.2024).
5. cloud-init 24.4 documentation. cloud-init 24.4 documentation. URL: https://cloudinit.readthedocs.io/en/latest/ (date of access: 18.12.2024).
6. AWS - Auth Methods | Vault | HashiCorp Developer. AWS - Auth Methods | Vault | HashiCorp Developer. URL: https://developer.hashicorp.com/vault/docs/auth/aws (date of access: 18.12.2024).
7. Kostromin, R. O. (2020). Survey of software configuration management tools of nodes in heterogeneous distributed computing environment. The International Workshop on Information, Computation, and Control Systems for Distributed Environments. https://doi.org/10.47350/iccs-de.2020.15
8. S, L. (2022). Automation of server configuration using Ansible. International Journal for Research in Applied Science and Engineering Technology, 10(6), 4109–4113. https://doi.org/10.22214/ijraset.2022.44840

Dmitry Bulavin *PhD, associate professor*
Kharkiv National University named V. N. Karazin 61000
e-mail: d.bulavin@karazin.ua
ORCID: 0000-0002-4840-4763

Vladyslav Samoilenko *Student*
Kharkiv National University named V. N. Karazin 61000
e-mail: vladyslav.samoilenko@student.karazin.ua;
ORCID: 0009-0001-3217-4473

Self-Configuring Virtual Machine Environments Using Ansible Pull: A Review of Approaches and Challenges

Relevance. Automation of virtual machine (VM) configuration is a key component of modern IT infrastructure, ensuring scalability, reliability, and security. With the increasing adoption of cloud technologies, there is a growing demand for standardized approaches to infrastructure management that overcome the limitations of traditional methods such as manual configuration or standalone scripts.
Goal. To design an architecture for automating VM self-configuration based on Ansible Pull, integrated with tools like Terraform, Hashicorp Vault, and Git, to ensure flexibility, autonomy, and security.
Research methods. The proposed system utilizes an analytical approach to evaluate existing configuration tools, models architecture using Git repositories, Terraform, and Vault, and implements cloud-init to initialize Ansible Pull on VMs.
Results. The proposed architecture automates VM configuration processes, ensuring scalability and reducing dependency on centralized management servers. Ansible Pull implementation with tagging supports idempotent task execution, regular updates, and maintaining the desired state of the system.
Conclusions. Combining Ansible Pull with Terraform, Git, and Vault provides ease of implementation, scalability, and secure configuration handling. The approach is effective for dynamic environments requiring regular updates and aligns with modern DevOps practices.

Keywords: *Ansible Pull, configuration automation, Terraform, Vault, self-configuration, DevOps, Ansible*