МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ВОДНОГО ГОСПОДАРСТВА ТА ПРИРОДОКОРИСТУВАННЯ
Навчально-науковий інститут автоматики, кібернетики та обчислювальної техніки

«До захисту допущена»
Зав. кафедри прикладної математики
д.т.н., професор Турбал Ю.В.
«____» _____ 20__ р.

КВАЛІФІКАЦІЙНА РОБОТА

« РОЗРОБКА PIPELINE CI/CD З ВИКОРИСТАННЯМ TERRAFORM ТА AWS »

Виконав: **Татусь Олександр Русланович** _____
(прізвище, ім'я, по батькові) (підпис)

група КН-21інт

Керівник: **доцент, к.т.н. Ярощак С.В.** _____
(науковий ступінь, вчене звання, посада, призвище та ініціали) (підпис)

Рівне – 2023

ЗМІСТ

РЕФЕРАТ

Об’єкт дослідження: системи автоматизації процесів розробки, тестування та розгортання програмного забезпечення.

Предмет дослідження: системи Terraform і Amazon Web Services (AWS) у поєднанні з пайплайнами.

Мета дослідження: розробка системи CI/CD-пайплайнів (Continuous Integration/Continuous Deployment) на основі Terraform і Amazon Web Services (AWS).

Актуальність. Застосування Terraform, AWS та CI/CD-пайплайнів в сучасному програмуванні є надзвичайно актуальним, оскільки вони сприяють автоматизації та прискоренню процесів розробки, забезпечують більшу надійність та ефективність розгортання інфраструктури та програмного забезпечення. Вони дозволяють командам розробників працювати з більшою гнучкістю, швидкістю та впевненістю в якості своєї роботи.

Ключові слова: пайплайн, Terraform, AWS, CI/CD.

ВСТУП

Terraform і Amazon Web Services (AWS) є дуже актуальними і популярними інструментами у сучасному світі розробки та управління інфраструктурою. Використання їх у поєднанні зі створенням CI/CD-пайплайнів (Continuous Integration/Continuous Deployment) дозволяє організаціям автоматизувати процеси розробки, тестування та розгортання програмного забезпечення.

Terraform є інструментом для створення Infrastructure as Code (IaC), що дозволяє керувати інфраструктурою за допомогою коду. Його актуальність полягає в тому, що він забезпечує можливість декларативно описувати всю інфраструктуру, включаючи віртуальні машини, мережеві ресурси, контейнери та інші складові системи. Terraform дозволяє розгортати і керувати інфраструктурою на різних хмарних платформах, включаючи AWS, забезпечуючи швидкість, надійність та повторюваність процесу.

AWS, з свого боку, є однією з найбільш популярних та розширених хмарних платформ. Вона надає широкий спектр послуг, таких як обчислення, зберігання даних, бази даних, мережі, аналітика та багато іншого. Актуальність AWS полягає в її гнучкості, масштабованості та надійності. AWS забезпечує безпеку, доступність та швидкість для розгортання та управління інфраструктурою, а також надає різноманітні інструменти та сервіси для автоматизації процесів розробки та впровадження програмного забезпечення.

CI/CD-пайплайн (Continuous Integration/Continuous Deployment) дозволяє автоматизувати процеси розробки та впровадження програмного забезпечення. Використання пайплайнів дозволяє розробникам швидко та ефективно вносити зміни до кодової бази, автоматично проводити тести та розгортати нові версії програмного забезпечення. Це забезпечує більшу якість, швидкість та стабільність розробки.

Застосування Terraform, AWS та CI/CD-пайплайнів в сучасному програмуванні є надзвичайно актуальним, оскільки вони сприяють автоматизації та прискоренню процесів розробки, забезпечують більшу надійність та ефективність розгортання інфраструктури та програмного забезпечення. Вони дозволяють командам розробників працювати з більшою гнучкістю, швидкістю та впевненістю в якості своєї роботи.

Terraform сьогодні можна вважати одним з найпопулярніших інструментів для створення Infrastructure as a Code.

Об’єкт дослідження: системи автоматизації процесів розробки, тестування та розгортання програмного забезпечення.

Предмет дослідження: системи Terraform і Amazon Web Services (AWS) у поєднанні з пайплайнами.

РОЗДІЛ 1
ХМАРНІ ПЛАТФОРМИ ТА СЕРВІСИ

1.1. Обгрунтування вибору теми

Amazon Web Services (AWS) розпочав свою роботу як технологічна інфраструктура для користувачів у 2006 році. На сьогоднішній день AWS використовує понад мільйоном активних користувачів для вирішення різноманітних завдань. Його перевагою є те, що він може обслуговувати організації будь-якого розміру. Навіть такі великі компанії, як Netflix і Expedia, спираються на AWS для надання своїх послуг у всьому світі. Малий бізнес також знаходить в AWS все необхідне.

Серед переваг Amazon можна виділити наступне:

- Легка масштабованість – AWS може підтримувати практично необмежену кількість користувачів.

- Гнучкість налаштування та підтримки сторонніх інтеграцій - практично будь-яка організація може користуватися AWS.

- Можливості аналітики в реальному часі та рішень для великих обсягів даних, доступні через програми Amazon Kinesis Streams та Firehose. Необхідно невелике команду розробників для налаштування цих функцій.

- Регулярні оновлення та надання нових функцій, оскільки AWS продовжує враховувати потреби своїх клієнтів.

Таким чином, AWS і Terraform разом надають потужний набір інструментів для створення і керування інфраструктурою у хмарному середовищі.

Terraform дозволяє визначити інфраструктуру як код, що дозволяє забезпечити її повторюваність та автоматизацію. За допомогою Terraform можна створювати, керувати та модифікувати інфраструктурні ресурси AWS, такі як віртуальні машини, мережеві налаштування, бази даних тощо.

AWS надає широкий спектр послуг, що дозволяють розгортати та керувати інфраструктурою в хмарному середовищі. Використання AWS у поєднанні з Terraform надає багато можливостей для автоматизації та управління інфраструктурою.

CI/CD pipeline дозволяє забезпечити автоматизовану поставку програмного забезпечення в середовище виробництва. В рамках pipeline можна виконувати автоматичну перевірку коду, збирання, тестування, розгортання та моніторинг програмного продукту.

Використання Terraform у CI/CD pipeline дозволяє автоматизувати процес розгортання та управління інфраструктурою в AWS. Це забезпечує повторюваність та консистентність інфраструктурних змін, зменшує ризик помилок та спрощує процес розгортання.

Управління конфігурацією за допомогою інструментів, таких як Terraform, дозволяє забезпечити контроль над змінами в інфраструктурі. Це дозволяє відстежувати та аудитувати зміни, швидко відновлювати попередні стану, а також спрощує спільну роботу в команді.

AWS надає різноманітні інструменти для автоматизації CI/CD pipeline, такі як AWS CodePipeline, AWS CodeBuild та AWS CodeDeploy. Ці інструменти допомагають створювати, тестувати та розгортати програмне забезпечення автоматично.

Правильна конфігурація та налаштування pipeline CI/CD забезпечує швидку поставку нових функцій та покращення якості програмного продукту. За допомогою автоматизованого pipeline можна забезпечити безперервну інтеграцію, тестування та розгортання змін, що допомагає знизити час від ідеї до виробництва.

Слід враховувати принципи безпеки та захисту даних при розробці та налаштуванні CI/CD pipeline. AWS надає різноманітні засоби для забезпечення безпеки, такі як управління доступом, шифрування даних, моніторинг тощо.

Загальною метою розробки pipeline CI/CD з використанням Terraform та AWS є автоматизація процесу розгортання та керування інфраструктурою, забезпечення безперервної поставки програмного забезпечення та покращення ефективності розробки. Правильна конфігурація та налаштування pipeline дозволяє забезпечити швидку, надійну та безпечну поставку програмного продукту у середовище виробництва.

1.2. Концепція Infrastructure as a Code

Ідея, що покладена в основі IaC (Інфраструктура як код), полягає в тому, що для визначення, розгортання, оновлення та видалення інфраструктури потрібно писати та виконувати код. Це є досить важливим, коли всі аспекти системного адміністрування розглядаються як програмне забезпечення — навіть ті, що стосуються обладнання (наприклад, налаштування фізичних серверів). Ключовим аспектом DevOps є те, що практично всім можна управляти за допомогою коду, включаючи сервери, бази даних, мережі, журнальні файли, програмну конфігурацію, документацію, автоматичні тести, процеси розгортання та інше. Інструменти IaC можна поділити на п'ять загальних категорій: Спеціалізовані скрипти; Засоби керування конфігурацією; Засоби шаблонізації серверів; Засоби оркестрації; Засоби ініціалізації ресурсів.

Спеціалізовані скрипти є найпростішим та зрозумілим способом автоматизувати різноманітні задачі. Вони включають у себе розбиття завдання на окремі кроки, опис кожного кроку у вигляді програмного коду, написаного на обраній скриптовій мові, та виконання отриманого скрипту на відповідному сервері.

Загальна мета Інфраструктури як код полягає в досягненні автоматизованого та повторюваного процесу розгортання та керування інфраструктурою, забезпечуючи ефективність, масштабованість та надійність

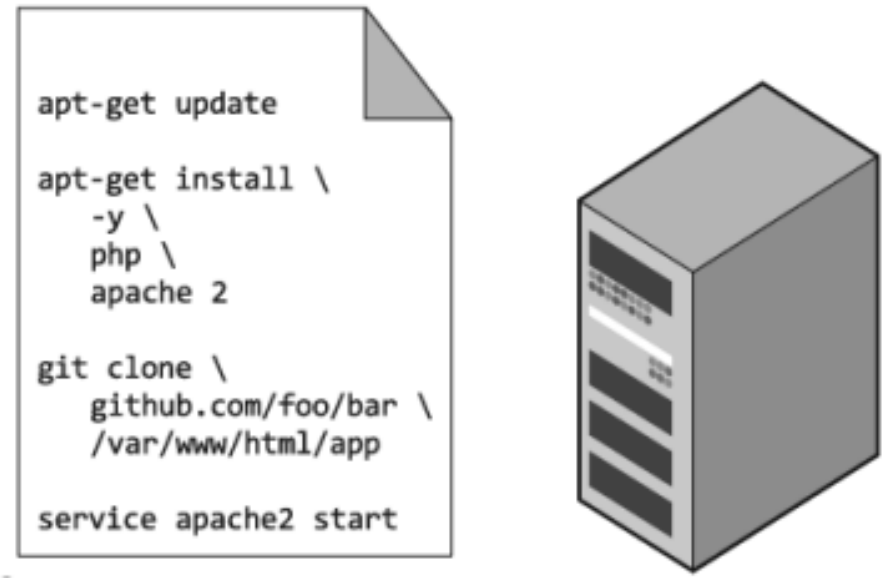


Рис. 1.1. Налаштування веб-сервера, завантаження коду з Git-репозиторію та запуск Apache

Наприклад, нижче наведено bash-скрипт setup-webserver.sh, який налаштовує веб-сервер, встановлює залежності, завантажує код з Git-репозиторію та запускає Apache:

```bash
# Оновлюємо кеш apt-get
sudo apt-get update
# Встановлюємо PHP та Apache
sudo apt-get install -y php apache2
# Копіюємо код з репозиторію
sudo git clone https://github.com/brikis98/php-app.git /var/www/html/app
# Запускаємо Apache
sudo service apache2 start
```

Вкрай зручною особливістю спеціалізованих скриптів (а також їх великим недоліком) є те, що код можна писати будь-яким зручним способом, використовуючи популярні мови загального призначення. Якщо інструменти, спеціально створені для IaC, надають лаконічний API для виконання складних завдань, то мови програмування загального призначення передбачають написання власного коду в кожному окремому випадку. Більше того, засоби IaC зазвичай нав'язують певну структуру коду, тоді як у спеціалізованих скриптах кожен розробник використовує власний стиль та робить речі по-своєму. Якщо мова йде про скрипт з восьми рядків, який встановлює Apache, обидва проблеми можна вважати незначними. Однак, якщо ви спробуєте застосувати той же підхід до управління десятками серверів, базами даних, балансувальниками навантаження та мережевою конфігурацією, все може піти шкереберть.

Якщо вам коли-небудь доводилося підтримувати великий репозиторій bash-скриптів, ви знаєте, що це майже завжди перетворюється на "безлад" погано структурованого коду. Спеціалізовані скрипти чудово підходять для невеликих одноразових завдань, але якщо ви плануєте керувати всією своєю інфраструктурою у вигляді коду, слід використовувати спеціально призначені для цього інструменти IaC.

Засоби управління конфігурацією, такі як Chef, Puppet, Ansible і SaltStack, є інструментами для управління конфігурацією. Це означає, що вони призначені для встановлення та адміністрування програмного забезпечення на існуючих серверах.

Використання спеціально призначеного інструменту для управління конфігурацією, має значні переваги порівняно з простими спеціалізованими скриптами, особливо коли ви працюєте з великим обсягом інфраструктури.

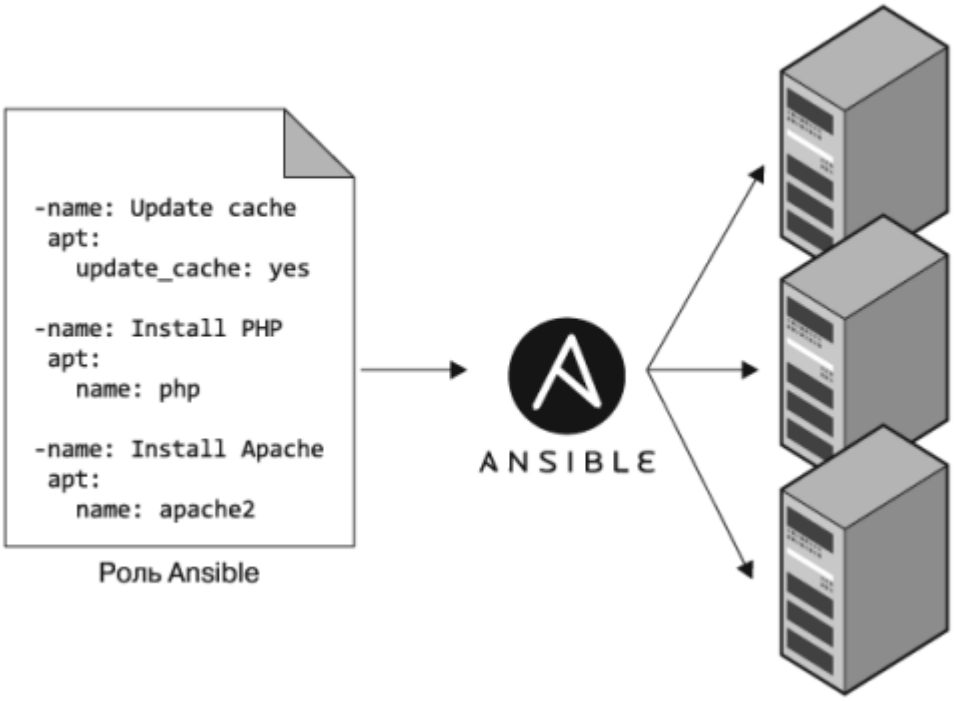


Рис. 1.2. Архітектура розподіленого управління конфігурацією з використанням Ansible

Концепція Infrastructure as a Code (IaC) – це підхід до управління інфраструктурою, в якому її конфігурація, розгортання, оновлення і видалення виконуються шляхом написання та виконання коду. IaC дозволяє розглядати всі аспекти системного адміністрування як програмне забезпечення, включаючи налаштування фізичних серверів, мережі, баз даних, журнальних файлів, процеси розгортання тощо.

Ця концепція базується на ідеї, що інфраструктура повинна бути керована кодом, а не ручними процедурами адміністратора. Замість традиційних ручних дій, розгортання та управління інфраструктурою здійснюється автоматично і повторювано за допомогою конфігураційних файлів або скриптів. Це дозволяє забезпечити консистентність, надійність і швидкість управління інфраструктурою.

Одним із найпоширеніших інструментів для реалізації IaC є Terraform. Terraform - це інфраструктурний оркестратор, який дозволяє описувати бажану стан інфраструктури за допомогою декларативної мови опису ресурсів. За допомогою Terraform можна визначити сервери, мережі, бази даних, навантажувачі, ролі доступу та багато іншого. Коли конфігурація описана, Terraform автоматично розгортає та управляє інфраструктурою, забезпечуючи її потрібний стан.

Іншими популярними інструментами IaC є Ansible, Chef, Puppet і SaltStack. Вони надають засоби для автоматизації конфігурації і управління серверами та програмним забезпеченням. Ці інструменти дозволяють вам описувати бажаний стан системи, а потім автоматично розгортати та керувати нею за допомогою коду.

Переваги використання IaC включають наступні.

Повторюваність та консистентність. За допомогою коду ви можете повторювати розгортання та керування інфраструктурою, що забезпечує консистентність результатів і уникнення помилок.

Швидкість та ефективність. Автоматизоване розгортання дозволяє швидко реагувати на зміни вимог та розмір інфраструктури. Ви можете легко масштабувати, змінювати та оновлювати ресурси, що дозволяє скоротити час та зусилля, необхідні для управління інфраструктурою.

Версіонування та контроль. Код інфраструктури може бути збережений у системах контролю версій, що дозволяє відстежувати зміни, повертатися до попередніх станів і спільно працювати над розробкою інфраструктури.

Безпека. IaC дозволяє визначати правила безпеки та стандарти за допомогою коду, що спрощує впровадження та забезпечення безпеки інфраструктури.

Зменшення людського фактору. Менше залежності від ручного втручання адміністратора знижує ризик помилок та забезпечує більшу автоматизацію в управлінні інфраструктурою.

Загалом, концепція Infrastructure as a Code дозволяє розглядати інфраструктуру як програмне забезпечення, що дозволяє автоматизувати та керувати інфраструктурою шляхом написання коду. Це покращує ефективність, повторюваність та безпеку управління інфраструктурою, особливо в контексті масштабованих та складних середовищ.

1.3. Огляд Terraform

Terraform – це проект з відкритим вихідним кодом, розроблений компанією HashiCorp у 2014 році. Він виступає чудовим інструментом для створення Infrastructure as a Code. Фактично, Terraform використовується як мова програмування в хмарному середовищі. Terraform взаємодіє з такими провайдерами хмарних послуг, як Amazon Web Services, Google Cloud Platform, Microsoft Azure, Digital Ocean, AliCloud та багатьма іншими, з якими можна ознайомитися на офіційному веб-сайті Terraform. У нашій роботі ми будемо працювати з AWS.

Синтаксис коду Terraform написаний мовою HashiCorp Configuration Language (HCL). Файли коду Terraform мають структуру текстових файлів і можуть бути відкриті в будь-якому текстовому редакторі. Зазвичай для популярних інтегрованих середовищ розробки існують плагіни для роботи з Terraform. Файли Terraform не потребують компіляції і працюють на операційних системах Windows, Mac і Linux.

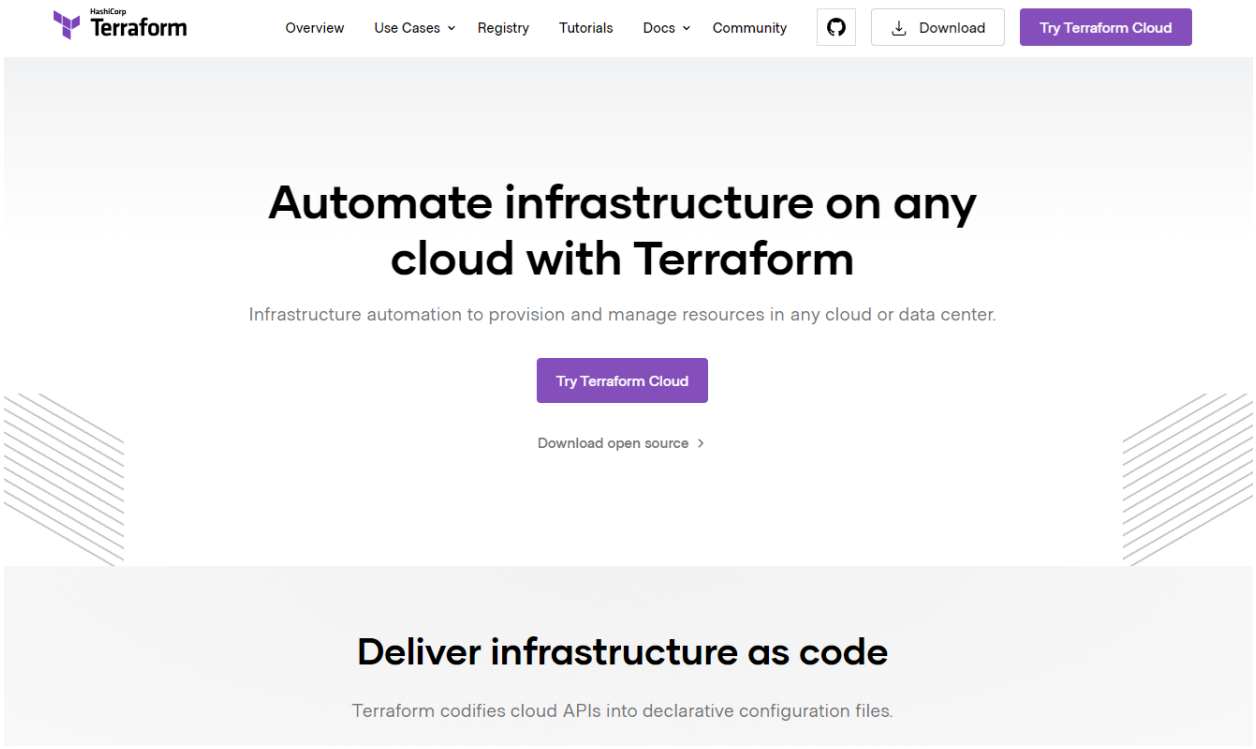


Рис. 1.3. Офіційний сайт Terraform – terraform.io

Terraform виконує функцію Infrastructure as a Code (IaaC). Користувачі можуть описати та створити інфраструктуру за допомогою декларативної мови конфігурації, HashiCorp Configuration Language (HCL), або, за бажанням, JSON. За допомогою Terraform ми можемо визначити всі необхідні ресурси для створення кластера та іншої інфраструктури як код, що дозволить нам повторно використовувати його для інших проектів або завдань, змінюючи лише значення змінних.

Terraform створює та керує ресурсами на хмарних платформах та інших сервісах, використовуючи свої програмні інтерфейси (API). Завдяки провайдерам, Terraform може взаємодіяти з практично будь-якою платформою або сервісом, які надають доступ до свого API.

Основний робочий процес Terraform складається з трьох етапів: "Write", "Plan" та "Apply". На етапі "Write" ви визначаєте ресурси, які можуть існувати на різних хмарних постачальниках та службах. Наприклад, ви можете створити

конфігурацію для розгортання програми на віртуальних машинах у віртуальній приватній хмарі (VPC) з групами безпеки та балансуванням навантаження.

На етапі "Plan" Terraform генерує план виконання, який описує, яку інфраструктуру він створить, оновить або видалить на основі поточного стану і вашої конфігурації.

На етапі "Apply", після підтвердження, Terraform виконує запропоновані операції в правильному порядку з урахуванням залежностей від ресурсів. Наприклад, якщо ви оновлюєте властивості VPC і змінюєте кількість віртуальних машин у цій VPC, Terraform спочатку відтворить VPC, а потім масштабуватиме віртуальні машини.

Terraform підтримує використання багаторазових компонентів конфігурації, які називаються модулями. Вони дозволяють визначати настроювані колекції інфраструктури, що зберігають час та сприяють використанню найкращих практик. Ви можете використовувати готові модулі з реєстру Terraform або створити свої власні.

Terragrunt, з свого боку, є обгорткою для Terraform і надає додаткові інструменти для зберігання конфігурацій Terraform, роботи з кількома модулями Terraform та керування віддаленим станом.

Terraform – це інструмент з відкритим вихідним кодом від компанії HashiCorp, написаний мовою програмування Go. Код на мові Go компілюється в один об'єднаний двійковий файл (точніше, по одному файлу для кожної підтримуваної операційної системи) з передбачуваним назвою terraform. Цей файл дозволяє розгортати інфраструктуру безпосередньо з вашого ноутбука, сервера (або будь-якого іншого комп'ютера) без необхідності додаткової інфраструктури. Все це стає можливим завдяки тому, що виконуваний файл terraform виконує API-виклики від вашого імені до одного або декількох провайдерів, таких як AWS, Azure, Google Cloud, DigitalOcean, OpenStack тощо. Це означає, що Terraform використовує інфраструктуру, яку ці провайдери

надають для своїх серверів API, а також їхні механізми аутентифікації, наприклад, ваші API-ключі для AWS.

В концепції "інфраструктура як код" ці файли виконують роль коду. Ось приклад конфігурації Terraform:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
resource "google_dns_record_set" "a" {
  name         = "demo.google-example.com"
  managed_zone = "example-zone"
  type         = "A"
  ttl          = 300
  rrdatas      = [aws_instance.example.public_ip]
}
```

Цей фрагмент змушує Terraform виконувати API-виклики до двох провайдерів: до AWS, щоб розгорнути там сервер, і до Google Cloud, щоб створити DNS-запис, який вказує на IP-адресу сервера з AWS. Terraform дозволяє використовувати єдиний простий синтаксис для розгортання взаємопов'язаних ресурсів у кількох різних хмарах. Ви можете описати всю вашу інфраструктуру (сервери, бази даних, балансувальники навантаження, топологію мережі і т.д.) у конфігураційних файлах Terraform і зберігати їх у системі керування версіями. Потім цю інфраструктуру можна буде розгорнути за допомогою визначених команд, таких як terraform apply. Утиліта terraform аналізує ваш код, перетворює його у послідовність API-викликів до хмарних провайдерів, що в ньому вказані, та виконує ці API-виклики від вашого імені максимально ефективним способом.

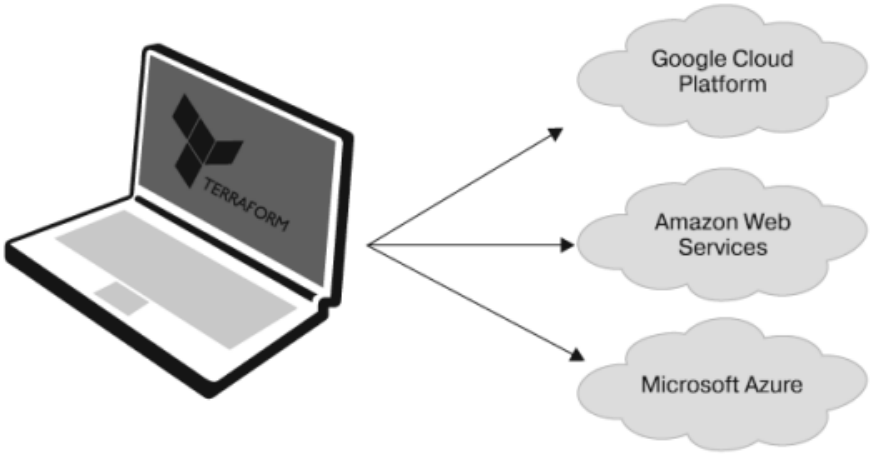


Рис. 1.4. Ілюстрація роботи Terraform

Якщо хтось у вашій команді бажає змінити інфраструктуру, замість того, щоб робити це вручну безпосередньо на серверах, вони редагують конфігураційні файли Terraform. Після цього вони перевіряють ці файли за допомогою автоматичних тестів і аналізу коду, фіксують оновлений код у системі керування версіями і потім виконують команду terraform apply, щоб здійснити необхідні API-виклики для розгортання змін.

Цей процес дозволяє забезпечити контроль над змінами інфраструктури, забезпечується консистентність серед учасників команди і сприяє автоматизації розгортання. Шляхом внесення змін у конфігураційні файли Terraform і їхнього застосування за допомогою команди terraform apply, зміни будуть застосовуватися до інфраструктури у відповідності до нового коду.

Цей процес також дозволяє команді переглядати, аналізувати і коментувати зміни в коді перед їхнім застосуванням, забезпечуючи більшу прозорість і співпрацю. Крім того, завдяки системі керування версіями, ви зможете відстежувати всі зміни, переглядати попередні версії і відновлювати попередні стани інфраструктури при необхідності.

Загалом, процес роботи з Terraform дозволяє команді ефективно керувати змінами інфраструктури, забезпечуючи автоматизацію, прозорість і контроль.

1.4. Аналіз альтернатив та огляд наявних досліджень

Розробка pipeline CI/CD (Continuous Integration/Continuous Deployment) з використанням Terraform та AWS є актуальною темою, оскільки ці інструменти дозволяють автоматизувати процеси розгортання та керування інфраструктурою. Такий пайплайн допомагає забезпечити швидке та надійне внесення змін у програмне забезпечення та інфраструктуру.

Наукові дослідження в цій галузі зазвичай розглядають наступні аспекти.

Архітектура pipeline CI/CD: Дослідження вивчають різні архітектурні підходи до побудови pipeline CI/CD з використанням Terraform та AWS. Вони досліджують, як організувати етапи автоматизованого розгортання, тестування, моніторингу та управління інфраструктурою.

Інтеграція Terraform та AWS: Дослідники досліджують методи та практики інтеграції Terraform та AWS для автоматизації розгортання та управління інфраструктурою.

Якщо розглядати альтернативні інструменти Infrastructure as a Code , то можна вказати наступні: AWS CloudFormation, Ansible, Puppet, Chef.

Інструменти управління конфігурацією, такі як Chef, Puppet, Ansible та SaltStack, а також інструменти такі як CloudFormation, Terraform і OpenStack Heat, представляють собою різні засоби для автоматизації розгортання та конфігурації інфраструктури [2].

Хоча деяка межа між цими двома категоріями інструментів неоднозначна, оскільки інструменти управління конфігурацією можуть включати певний рівень провіжинінгу (наприклад, розгортання сервера за допомогою Ansible), а інструменти провіжинінгу можуть виконувати певну конфігурацію (наприклад, виконання сценаріїв конфігурації на серверах, які надає Terraform), вибір підходящого інструменту залежить від вашої конкретної ситуації використання [2].

Зокрема, якщо ви використовуєте шаблонні засоби для створення серверів, такі як Docker або Packer вибір зрозумілий. Після створення образу зі шаблону Dockerfile або Packer вам лише потрібно надати інфраструктуру для запуску цих образів.

У випадку, якщо ви не використовуєте шаблонні засоби для серверів, ефективною альтернативою є комбінування інструментів управління конфігурацією та провіжинінгу. Наприклад, ви можете використовувати Terraform для створення серверів і виконувати налаштування кожного сервера з допомогою Ansible. Крім того, Terraform має пряму інтеграцію з інструментами управління конфігурацією [2].

Інструменти управління конфігурацією, такі як Chef, Puppet, Ansible та SaltStack, мають властивість змінювати парадигму інфраструктури шляхом внесення змін безпосередньо на існуючі сервери. Наприклад, при використанні Chef для встановлення нової версії OpenSSL, він автоматично оновлює програмне забезпечення на існуючих серверах, що призводить до унікальних змін на кожному сервері. З часом, коли накопичується все більше оновлень, кожен сервер стає відмінним від інших, що призводить до тонких помилок конфігурації, складних у діагностиці та відтворенні (це відома проблема дрейфу конфігурації, яка виникає при ручному керуванні серверами, хоча використання інструментів управління конфігурацією робить цю проблему менш проблематичною) [2]. Навіть з автоматизованими тестами, важко уникнути таких помилок.

У разі використання інструментів - наприклад Terraform, для розгортання машинних образів, створених за допомогою Docker або Packer, більшість "змін" фактично означає розгортання абсолютно нового сервера. Наприклад, для оновлення версії OpenSSL ви можете використати Packer для створення нового образу з оновленою версією OpenSSL, розгорнути цей образ на нових серверах і вимкнути старі сервери. Завдяки використанню незмінних образів при кожному

розгортанні на нових серверах, цей підхід зменшує ризик помилок дрейфу конфігурації, дозволяє точно знати, яке програмне забезпечення присутнє на кожному сервері і легко розгортати попередні версії (за допомогою попередніх образів) у будь-який момент. Це також полегшує автоматизоване тестування, оскільки стабільний образ, який пройшов тести у тестовому середовищі, має велику ймовірність поводитись так само у виробничих умовах [2].

Варто зазначити, що хоча інструменти управління конфігурацією можуть бути налаштовані для незмінних розгортань, це не є типовим підходом до їх використання, тоді як для інструментів провіжинінгу це є природним. Незмінність триває лише до моменту запуску сервера і його роботи, оскільки після цього можуть відбуватись зміни на жорсткому диску і конфігурація може дещо відрізнятись [2].

Інструменти, такі як Chef та Ansible, пропонують процедурний стиль, де ви пишете код, який визначає послідовність кроків для досягнення бажаного стану інфраструктури. Ви конкретно описуєте, що робити на кожному кроці. Однак, цей підхід має свої обмеження. Процедурний код не завжди повністю відображає поточний стан інфраструктури, а також ускладнює повторне використання коду через постійні зміни стану.

У свою чергу, декларативний підхід, який використовують інструменти, такі як Terraform, CloudFormation, SaltStack, Puppet і OpenStack Heat, дозволяє описати бажаний кінцевий стан інфраструктури, а інструмент самостійно вирішує, як досягти цього стану. Ви просто описуєте, що ви хочете мати, а не як цього досягти. Це спрощує розуміння поточного стану інфраструктури і полегшує повторне використання коду, оскільки вам не потрібно враховувати поточний стан вручну. Крім того, декларативний підхід забезпечує стабільні та зрозумілі бази даних, оскільки код відображає останній стан інфраструктури.

Отже, декларативний підхід, яким керують інструменти IaC, зазвичай є більш простим, зрозумілим та легко керованим у порівнянні з процедурним

підходом, який має тенденцію до зростання та ускладнення з часом.

За замовчуванням інструменти Chef, Puppet та SaltStack вимагають використання головного сервера для зберігання стану інфраструктури та розповсюдження оновлень [2]. Для оновлення будь-якого елементу інфраструктури необхідно використовувати клієнтську програму, яка надсилає команди на головний сервер. Головний сервер потім розповсюджує оновлення на всі інші сервери або ці сервери регулярно отримують оновлення з головного сервера. Головний сервер має декілька переваг. По-перше, це централізоване сховище, де можна бачити та керувати станом інфраструктури. Багато інструментів управління конфігурацією навіть надають веб-інтерфейс для головного сервера, що полегшує розуміння ситуації. По-друге, деякі головні сервери можуть постійно працювати в фоновому режимі та застосовувати конфігурацію. Таким чином, якщо будь-хто вносить ручні зміни на сервері, головний сервер може скасувати ці зміни для запобігання порушенню конфігурації. Проте запуск головного сервера має наступні недолікию

Додаткова інфраструктура. Для використання необхідно розгорнути додатковий сервер або навіть кластер серверів для забезпечення високої доступності та масштабованості [2].

Технічне обслуговування. Головний сервер потребує постійного підтримування, оновлення, створення резервних копій, контролю та масштабування [2].

Безпека. Необхідно забезпечити засоби комунікації клієнта з головним сервером або серверами, а також спосіб взаємодії головного сервера або серверів з усіма іншими серверами. Це часто вимагає відкриття додаткових портів та налаштування додаткових систем аутентифікації, що збільшує потенційну можливість атаки [2].

Інструменти Ansible, CloudFormation, Heat і Terraform за замовчуванням працюють без головного сервера [2]. Деякі з них можуть використовувати

головний сервер як частину внутрішньої інфраструктури, але це вже є складовою їхньої функціональності, а не необхідним елементом, який потребує окремого керування. Наприклад, Terraform взаємодіє з постачальниками хмарних послуг через їх API, тому можна вважати, що сервери API є головними серверами. Проте, це не потребує додаткової інфраструктури або механізмів аутентифікації, крім використання API-ключів [2]. Аналогічно, SSH-з'єднання безпосередньо з кожним сервером використовується Ansible, що також не потребує додаткової інфраструктури або механізмів аутентифікації (просто використовується SSH-ключі) [2].

У таблиці 1.1 показано порівняння популярних інструментів IaC з даними, які були зібрані у травні 2019 року, включаючи, чи інструмент IaC з відкритим або закритим кодом, які хмарні провайдери він підтримує, загальну кількість учасників та зірок на GitHub, скільки комісій та активних питань було протягом одного місяця з середини квітня до середини травня, скільки бібліотек з відкритим кодом доступно для цього інструмента, кількість запитань, вказаних для цього інструменту в StackOverflow, і кількість завдань, які згадуються інструмент на веб-сайті Indeed.com

Таблиця 1.1

Статистика для порівняння IaaC засобів

	Вихідний код	Хмара	Вкладники	Коміти (GitHub)	Баги	Бібліотеки	Stack Overflow	Завдання
Chef	Open	All	562	435	86	3832	5982	4378
Pupet	Open	All	515	94	314	6110	3585	4200
Ansible	Open	All	4386	506	523	20677	11746	8787
SaltStack	Open	All	2237	608	441	318	1062	1622

Продовження таблиці 1.1

Cloud Formation	Closed	AWS	?	?	?	?	3315	2318
Heat(Open Stack)	Open	All	361	12	600	0	88	2201
Terraform	Open	All	1261	173	204	1462	2730	3641

Знову ж таки, дані тут не є ідеальними, але їх досить, щоб помітити чітку тенденцію: Terraform і Ansible відчувають стрімке зростання. Збільшення кількості учасників, бібліотек з відкритим кодом, публікацій StackOverflow та робочих місць. Обидва ці інструменти сьогодні мають великі активні спільноти, і, судячи з цих тенденцій, цілком ймовірно, що вони в майбутньому стануть ще більшими [2].

Ще одним ключовим фактором, який слід враховувати при виборі будь-якої технології, є зрілість.

У таблиці 1.2 наведені початкові дати випуску та поточний номер версії (станом на квітень 2020 року) для кожного з інструментів IaC [2].

Таблиця 1.2

Порівняння зрілості вибраних IaaC рішень

	Initial Release	Curent Version
Pupet	2005	6.15.0
Chef	2009	16.0.290
CloudFormation	2011	???
SaltStack	2011	3000.2
Ansible	2012	2.9.7
Heat	2012	13.0.1
Terraform	2014	0.12.24

РОЗДІЛ 2
ОСОБЛИВОСТІ AWS

2.1. Хмара AWS

Amazon Web Services (AWS) є набором хмарних сервісів, який надається компанією Amazon. Ця платформа дозволяє користувачам замовляти обчислювальні ресурси, сховища, інфраструктуру та готові до використання інструменти на єдиній платформі. AWS має понад двісті повнофункціональних сервісів, які можуть бути використані в різних сценаріях роботи.

Набір сервісів, що часто використовуються в розробці інфраструктури та пайплайнів CI/CD з використанням Terraform та AWS, включає наступні.

EC2-Instances – це сервіс, що дозволяє користувачеві орендувати віртуальні сервери, відомі як "інстанси". За допомогою попередньо налаштованих образів можна запускати віртуальні сервери з потрібними конфігураціями.

EC2-ELB – сервіс, відомий як балансувальник навантаження, допомагає розподіляти мережевий трафік для покращення масштабованості додатків. Він забезпечує розподіл навантаження між різними серверами для забезпечення високої доступності та надійності.

Route53 – високодоступний хмарний веб-сервіс системи доменних імен (DNS), який масштабується. Він надсилає запити користувачів до різних інфраструктурних ресурсів AWS, таких як EC2 інстанси, балансувальники навантаження Elastic Load Balancing або сховища S3.

Relational Database Service (RDS) – керований сервіс, який спрощує налаштування, використання та масштабування реляційних баз даних у хмарі. Він надає економічні та масштабовані ресурси, водночас автоматично керуючи складними задачами адміністрування баз даних.

Secrets Manager – сервіс дозволяє захищати конфіденційні дані, що використовуються для доступу до програм, сервісів та IT-ресурсів. Він забезпечує просту ротацію та управління конфіденційними даними, такими як дані для доступу до баз даних, ключі API тощо, протягом усього життєвого циклу.

S3 – протокол передачі, розроблений Amazon, який використовується для зберігання об'єктів у хмарі. Він надає масштабоване та надійне об'єктне сховище для зберігання даних.

DynamoDB – повністю керована безсерверна база даних NoSQL, яка працює на основі пар "ключ-значення" і розроблена для високопродуктивних програм у будь-якому масштабі. DynamoDB пропонує вбудований захист, автоматичну реплікацію у кількох регіонах, кешування в пам'яті та інструменти експорту даних.

Identity Access Management (IAM) – сервіс забезпечує точний контроль доступу до всіх сервісів AWS. За допомогою IAM можна визначати, хто має доступ до певних сервісів та ресурсів і за яких умов. Завдяки політикам IAM можна керувати дозволами для співробітників та систем, надаючи найменші необхідні привілеї.

Elastic Container Registry (ECR) – повністю автоматизований реєстр контейнерів, який дозволяє розробникам легко розгортати та ділитися образами контейнерів та артефактами.

Elastic Kubernetes Service (EKS) – керований сервіс Kubernetes, який дозволяє запускати Kubernetes на AWS та в локальному середовищі. Amazon EKS сумісний з відкритою версією Kubernetes, тому програми, які працюють на відкритій версії Kubernetes, можуть бути використані з Amazon EKS.

Використання цих сервісів AWS у поєднанні з Terraform дозволяє розробникам створювати та управляти інфраструктурою як кодом, а також автоматизувати процеси розгортання (CI/CD). Такий підхід забезпечує швидке та

ефективне впровадження програмного забезпечення у хмарному середовищі AWS.

2.2. Базові налаштування

Перед початком роботи з нашим проекотм нам треба мати аккаунт в AWS, також треба створити User з ACCES_KEY та SECRET_KEY.

В AWS console в IAM створюємо користувача з паравами адміністратора (Рис. 2.1.).

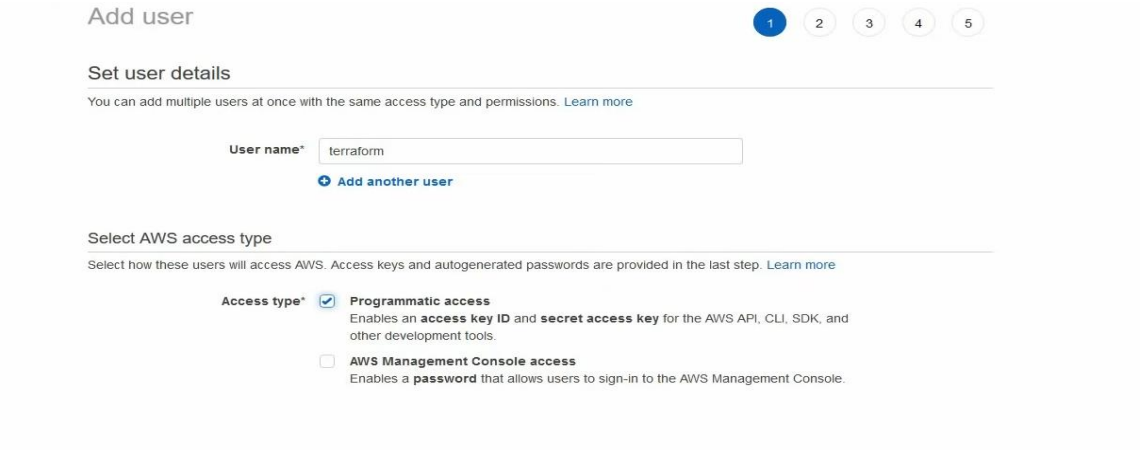


Рис. 2.1. Створення користувача з правами адміністратора

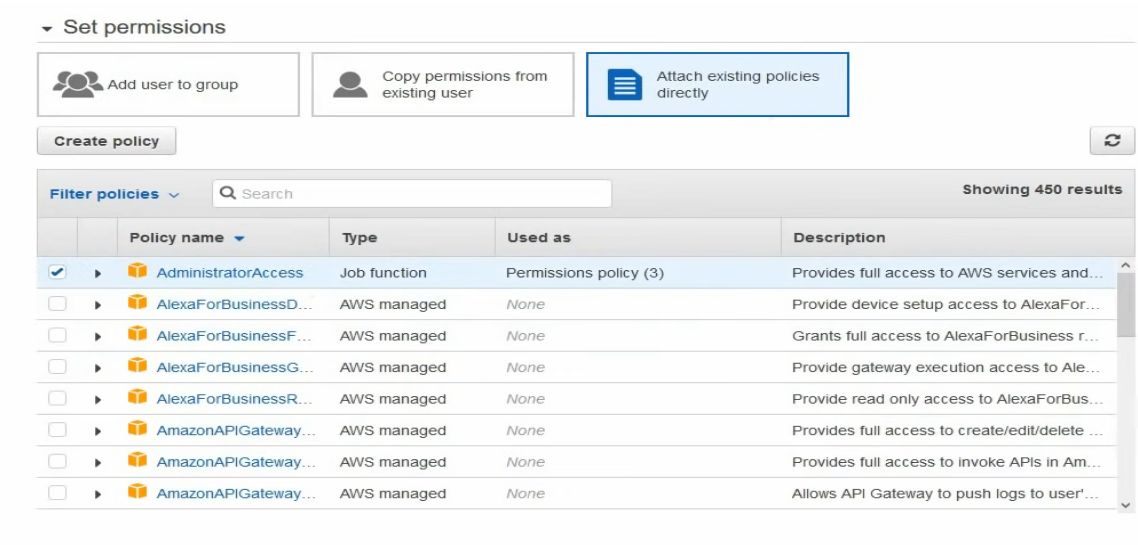


Рис. 2.2. Надання прав доступу

Для надання прав доступу користувачеві IAM потрібно зв'язати його обліковий запис з однією або кількома політиками IAM. Політика IAM - це документ у форматі JSON, який визначає, що дозволено або заборонено користувачеві. Ви можете створювати власні політики або використовувати готові, відомі як керовані політики. Для роботи з конвеєром в нашійй роботі потрібно призначити наступні керовані політики користувачеві IAM :

- AmazonEC2FullAccess
- AmazonS3FullAccess
- AmazonRDSFullAccess
- IAMFullAccess

Після надання цих політик ваш користувач IAM матиме відповідні права доступу, що дозволять виконувати необхідні операції у межах вашого облікового запису AWS.

2.3. Концепція pipeline CI/CD

Концепція CI/CD (Continuous Integration/Continuous Delivery) pipeline відноситься до автоматизованого процесу розробки та доставки програмного забезпечення. Вона включає в себе набір практик, інструментів та процедур, що допомагають розробникам ефективно здійснювати постійну інтеграцію змін у коді, автоматизоване тестування, збирання, пакування та доставку програмного забезпечення виробничому середовищу.

Основна ідея CI/CD pipeline полягає у тому, щоб зменшити ризики та покращити ефективність процесу розробки програмного забезпечення шляхом автоматизації та стандартизації кроків, які відбуваються від написання коду до його впровадження в продуктивне середовище.

Основні компоненти CI/CD pipeline включають:

1. Continuous Integration (постійна інтеграція). Цей етап включає автоматичне злиття змін коду з репозиторієм і виконання автоматичної компіляції, збирання та запуску тестів для перевірки цілісності коду. Якщо тести успішні, зміни інтегруються у загальну кодову базу.

2. Continuous Delivery (постійна доставка). На цьому етапі програмне забезпечення автоматично пакується та готується до випуску. Це може включати створення виконуваних файлів, контейнерів або інших форматів доставки, які готові до розгортання.

3. Continuous Deployment (постійне розгортання). Цей етап включає автоматичне розгортання пакетів програмного забезпечення виробничому середовищу. Зміни автоматично впроваджуються без необхідності вручного втручання, що дозволяє швидко виводити нові функції або виправлення помилок до користувачів.

4. Continuous Monitoring (постійний моніторинг). Після розгортання програмного забезпечення виробниче середовище перевіряється для забезпечення його працездатності та виявлення можливих проблем. Це може включати моніторинг показників продуктивності, реагування на помилки або автоматичне масштабування системи.

Використання CI/CD pipeline допомагає покращити швидкість розробки, забезпечити стабільність програмного забезпечення та знизити ризик впровадження нових функцій або змін. Вона сприяє автоматизації рутинних процесів та полегшує спільну роботу розробників, тестувальників та операторів систем.

Організації, які використовують CI/CD pipeline, зазвичай мають короткі цикли розробки, часті випуски програмного забезпечення та високий рівень автоматизації в своїх процесах розробки.

Основні етапи CI/CD pipeline включають:

- Кодування (Coding). Розробники написують новий код або вносять зміни до існуючого коду. Цей код зберігається у версійному контролі для контролю версій.
- Збирання (Building). Код збирається використовуючи засоби збирання (наприклад, компіляція, збірка пакетів тощо). Результатом цього етапу є виконуваний файл або інший артефакт, готовий для наступного етапу.
- Тестування (Testing). Автоматизовані тестові сценарії запускаються для перевірки якості коду. Це можуть бути модульні тести, функціональні тести, інтеграційні тести тощо. Результати тестування допомагають виявити помилки та недоліки.
- Розгортання (Deployment). Успішний код передається у виробниче середовище або відповідне середовище для демонстрації або реального використання. Це може включати автоматичне розгортання на серверах або контейнерах.
- Тестування виробничого середовища (Production Testing). У виробничому середовищі проводяться додаткові тести, щоб переконатися, що розгорнутий код працює належним чином. Це можуть бути тести на продуктивність, навантаження або безпеку.
- Моніторинг та забезпечення (Monitoring and Maintenance). У виробничому середовищі відбувається постійне моніторингу за роботою програмного забезпечення. Виявлені помилки або проблеми виправляються швидко для забезпечення безперебійної роботи системи.

Використання CI/CD pipeline дозволяє автоматизувати процеси розробки, тестування та доставки, забезпечуючи постійну інтеграцію і швидке

впровадження змін. Це сприяє покращенню якості програмного забезпечення, зниженню ризику помилок та підвищенню ефективності розробних команд.

2.4. CodeDeploy

2.4.1. Основні характеристики

AWS CodeDeploy – це сервіс управління розгортанням програмного забезпечення, який дозволяє автоматизувати процес доставки програмного коду на різні середовища, такі як Amazon EC2 і AWS Lambda. Він допомагає забезпечити безперебійні та швидкі розгортання, дозволяючи розробникам легко керувати версіями свого коду та контролювати процес розгортання.

AWS CodeDeploy може брати код із S3 bucket, GitHub. Сервіс може deploy код в AWS ECS, AWS Lambda, On-Premises Server (для цього потрібний CodeDeploy Agent), AWS EC2.

Працює це так. На наш акаунт наприклад, в Github, окрім основного коду треба закинути ще і файли appspec.yaml або appspec.yml.

```
os: linux
files:
  - source: /
    destination: /var/www/html/WordPress
hooks:
  BeforeInstall:
    - location: scripts/install_dependencies.sh
      timeout: 300
      runas: root
  AfterInstall:
    - location: scripts/change_permissions.sh
      timeout: 300
      runas: root
  ApplicationStart:
    - location: scripts/start_server.sh
    - location: scripts/create_test_db.sh
      timeout: 300
      runas: root
  ApplicationStop:
    - location: scripts/stop_server.sh
      timeout: 300
      runas: root
```

Рис. 2.3. Приклад файлу

Для прикладу можна описати таку реалізацію. В нас є GitHub репозиторій, CodeDeploy, EC2 Webserver with CodeDeploy Agent.

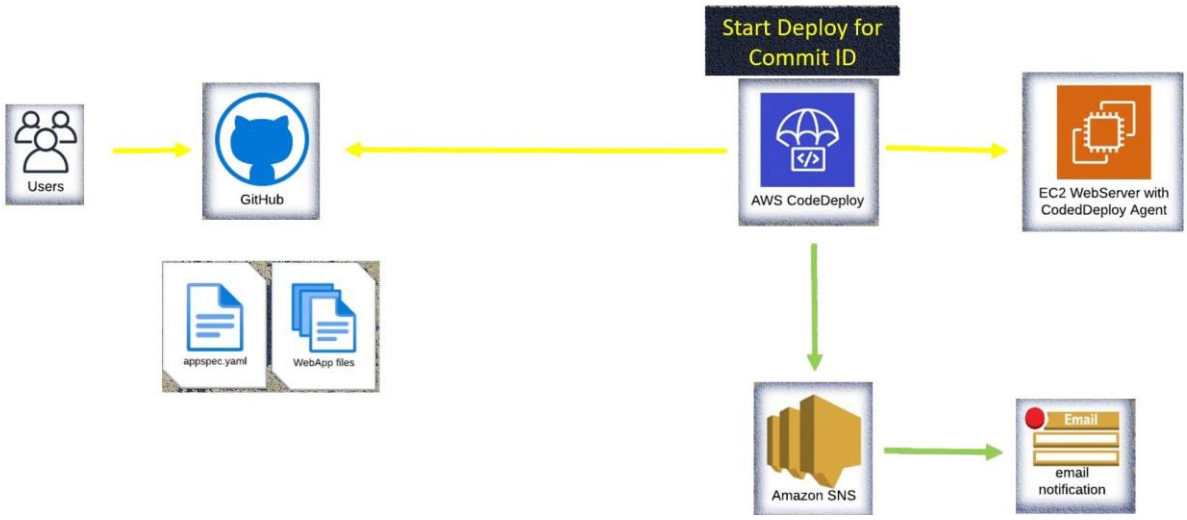


Рис. 2.4. Приклад реалізації

На GitHub розмістимо appspec.yaml та WebApp файли. Користувачі будуть виконувати push якісь зміни в WebApp , на CodeDeploy нам треба виконати Start Deploy for Commit ID вказавши який коміт брати з Github-a. Після цього CodeDeploy візьме код, який нам потрібен і розгорне його на веб-сервер .

Основні концепції та складові CodeDeploy:

1. Додаток (Application): Це контейнер для вашого програмного коду, який буде розгортатися на ваших серверах або в середовищі AWS Lambda. Ви можете створити один або кілька додатків в рамках вашого AWS-акаунту.
2. Група розгортання (Deployment Group): Це група серверів або середовищ AWS Lambda, на які буде розгортуватися ваш програмний код. Ви можете налаштовувати різні параметри групи розгортання, такі як кількість інстансів, політики трафіку, ролі доступу тощо.
3. Конфігурація розгортання (Deployment Configuration): Визначає швидкість та стратегію розгортання вашого коду. Ви можете вибрати одну з наявних конфігурацій або налаштувати свою власну стратегію.

4. Ревізія розгортання (Deployment Revision): Це версія вашого програмного коду, яку ви хочете розгорнути. Ревізія може бути представлена у вигляді архіву ZIP, образу Docker, S3-бакету тощо.
5. Хід розгортання (Deployment Lifecycle): CodeDeploy надає можливість контролювати етапи розгортання, такі як передрозгортання, розгортання, пост-розгортання тощо. Ви можете вказати власні сценарії, які виконуються на кожному етапі розгортання.
6. Моніторинг розгортання (Deployment Monitoring): Ви можете використовувати AWS CloudWatch для відстеження стану розгортання, журналів подій та метрик продуктивності.
7. Уведення/виведення (Input/Output): CodeDeploy надає можливість використовувати різні джерела коду та вихідні пункти, такі як GitHub, S3-бакети, Docker-контейнери тощо.

AWS CodeDeploy дозволяє автоматизувати процес розгортання програмного забезпечення, забезпечуючи безперебійність, швидкість та контроль над процесом. Він інтегрується з іншими сервісами AWS та популярними інструментами розробки, що дозволяє зручно використовувати його в різних сценаріях розробки програмного забезпечення.

2.4.2. CodeDeploy + CodePipeline

Комбінація сервісів AWS CodeDeploy і CodePipeline є потужним інструментом для автоматизації процесу постачання програмного забезпечення (CI/CD) в хмарному середовищі Amazon Web Services.

При використанні CodeDeploy разом з CodePipeline, розробники можуть створювати повністю автоматизовані конвеєри CI/CD. Процес починається з оновлення коду в репозиторії, що викликає спрацювання конвеєру CodePipeline. CodePipeline здійснює послідовні кроки збирання, тестування та розгортання, використовуючи CodeDeploy для керування розгортанням коду на різних

середовищах. Кожен етап проходить автоматично, і результати кожної операції можуть бути моніторингом та аналізовані.

Ця комбінація CodeDeploy і CodePipeline дозволяє побудувати надійні, швидкі і повністю автоматизовані конвеєри CI/CD, що забезпечують безперебійне розгортання програмного забезпечення в хмарному середовищі AWS. Вона дозволяє розробникам прискорити процес розробки та забезпечити швидку і безпечну доставку нового функціоналу в продакшн.

Для автоматизації можна вдосконалити схему із пункта 2.4.1 наступним чином. Ми можемо автоматизувати роботу CodeDeploy що б не вказувати йому який репозиторій і коміт брати, а щоб при кожному коміті це виконувалося автоматично. Тобто треба виконати повний DevOps CI/CD Pipeline. Схема зображена на Рис. 2.5.

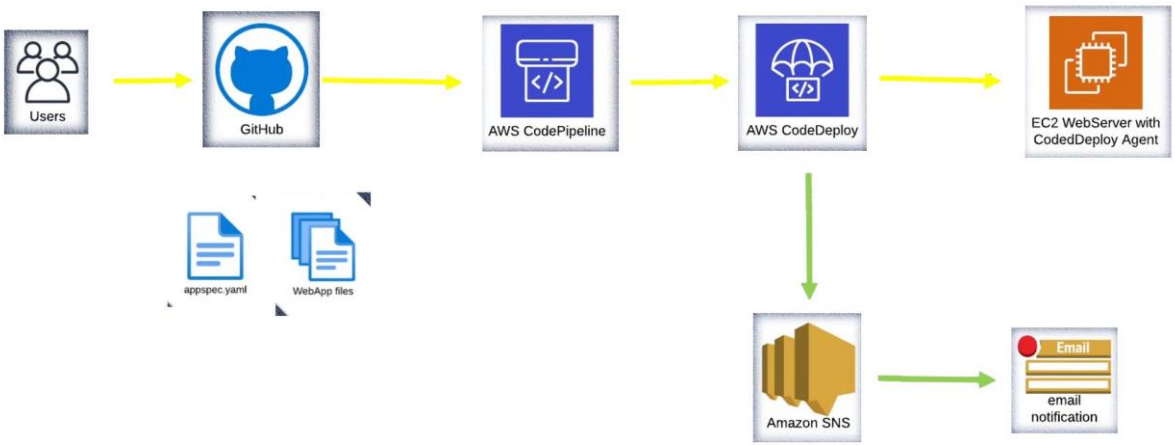


Рис. 2.5. Схема використання

2.5. CodeBuild

2.5.1. Базові характеристики

AWS CodeBuild є сервісом для автоматичної збірки, тестування програмного забезпечення. Він дозволяє створювати збірочні проекти, які

виконуються в хмарному середовищі AWS і можуть бути інтегровані в процес CI/CD.

CodeBuild дозволяє розробникам налаштовувати збірочні проекти з використанням конфігураційних файлів, написаних на мові YAML або JSON. Ці файли містять інструкції для збирання, тестування та пакування коду. Збірочні проекти можуть бути налаштовані для різних платформ та мов програмування, і CodeBuild автоматично обробляє налаштування оточення для виконання проектів.

Одним із головних переваг CodeBuild є його масштабованість. Він може працювати з різними рівнями обсягу ресурсів, що дозволяє легко масштабувати його в залежності від потреб розробки. Крім того, CodeBuild інтегрується з іншими сервісами AWS, такими як CodePipeline і CodeDeploy, що дозволяє створювати повністю автоматизовані конвеєри CI/CD.

При використанні CodeBuild разом з CodePipeline, розробники можуть створювати кроки збирання та тестування коду в рамках конвеєра CI/CD. CodeBuild автоматично стягує вихідний код з репозиторію, виконує зазначені кроки збірки та тестування, і генерує артефакти, які можуть бути передані для розгортання за допомогою CodeDeploy.

CodeBuild дозволяє ефективно керувати процесом збирання програмного забезпечення, спрощує його автоматизацію та полегшує інтеграцію з іншими сервісами AWS. Він допомагає забезпечити швидке, стабільне та надійне розгортання програмного забезпечення в хмарному середовищі.

Прикладом можна навести наступну схему. AWS CodeBuild використовує Docker images для того, щоб запускати білди. Можна використовувати docker image які містить Amazon, а можна брати будь-який свій. CodeBuild може брати код з S3, GitHub, AWS CodeCommit та із Bitbucket. Коли ми виконуєм build and test , build передбачає створення артефакту, тобто ми скомпілювали щось.

CodeBuild може покласти результат в S3, в DockerHub, ECR registry, в будь-яке інше місце.

У нас є GitHub. DockerBuild використовує файл buildspec.yml. Цей файл має бути в репозиторії git.

```
6
7    parameter-store:    # Get Value from SSM Parameter Store
8      TOKEN_ACCESS_KEY  : /shared/TOKEN_ACCESS_KEY
9      TOKEN_SECRET_KEY  : /shared/TOKEN_SECRET_KEY
10
11   secrets-manager:    # Get Value from Secrets Manager
12     PRIVATE_KEY       : /shared/SSH_PRIVATE_KEY
13     PUBLIC_KEY        : /shared/SSH_PUBLIC_KEY
14
15 phases:
16   install:
17     commands:
18       - echo "Start of INSTALL Phase"
19       - mkdir ~/.ssh/
20       - echo "$TOKEN_ACCESS_KEY"   > ~/.ssh/access_token.txt
21       - echo "$TOKEN_SECRET_KEY"  >> ~/.ssh/secret_token.txt
22       - echo "$PRIVATE_KEY"        > ~/.ssh/id_rsa
23       - echo "$PUBLIC_KEY"         > ~/.ssh/id_rsa.pem
24       - chmod 600 ~/.ssh/id_rsa
25
26   pre_build:
27     commands:
28       - echo "Start of PREBUILD Phase"
29       - aws --version
30
31   build:
32     commands:
33       - echo "Start of BUILD Phase"
34       - echo "Do something to build or test your code!!!"
35
```

Рис. 2.6. Приклад використання файлу buildspec.yml

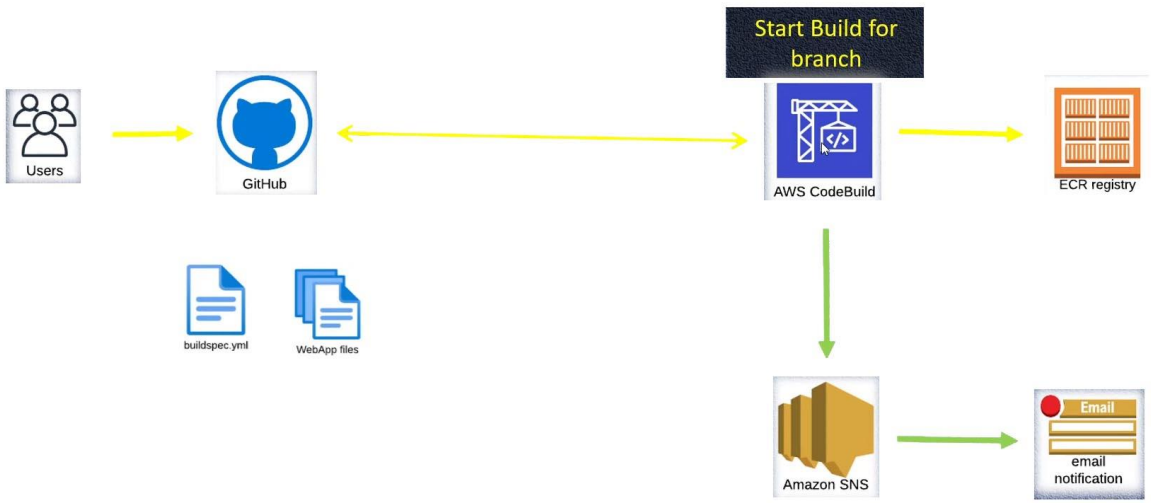


Рис. 2.7. Схема реалізації

Тут ми виконуємо наступне,: створюється CI pipeline, який буде створювати DockerImage з нашим веб-додатком і після того, як він виконає build, він виконає push його в ECR.

2.5.2. CodeBuild + CodePipeline

Комбінація AWS CodeBuild і AWS CodePipeline надає потужні інструменти для автоматизації процесу розробки та розгортання програмного забезпечення. CodeBuild виконує збірку та тестування коду, в той час як CodePipeline керує цим процесом, роблячи автоматичні розгортання через різні середовища.

AWS CodeBuild забезпечує безліч можливостей для налаштування збірочних процесів. Він може інтегруватись з різними системами контролю версій, такими як AWS CodeCommit, GitHub або Bitbucket, для отримання вихідного коду. Після отримання коду CodeBuild виконує задані кроки збирки, такі як компіляція, збірка залежностей та виконання тестів. Результатом роботи CodeBuild є артефакти, які можуть бути використані для розгортання.

AWS CodePipeline є сервісом для автоматизації і управління каналами постачання програмного забезпечення. Він дозволяє створювати послідовності кроків, що включають CodeBuild, а також інші сервіси, такі як CodeDeploy або AWS Elastic Beanstalk. CodePipeline забезпечує автоматичні розгортання програмного забезпечення з одного середовища в інше, забезпечуючи безперебійну доставку програмних змін.

Коли поєднати CodeBuild і CodePipeline, розробники отримують повну автоматизацію процесу CI/CD. CodeBuild виконує збірку, тестування та генерацію артефактів, в той час як CodePipeline керує послідовністю кроків розгортання і автоматично виконує розгортання на визначені середовища. Це дозволяє розробникам швидко та безпечно доставляти зміни в продукцію, забезпечуючи стабільність та надійність процесу розгортання.

Застосування CodeBuild і CodePipeline дозволяє побудувати ефективний та автоматизований конвеєр CI/CD, який забезпечує швидке розгортання програмного забезпечення з високою якістю і надійністю. Ця комбінація сервісів AWS робить процес розробки та розгортання більш простим і ефективним, допомагаючи командам розробників швидше відповідати на зміни та постачати новий функціонал користувачам.

РОЗДІЛ 3
TERRAFORM

3.1. Деякі аспекти розробки з Terraform

3.1.1. Налаштування

Рис. 3.1. Завантаження з офіційного сайту

Завантажуємо з офіційного сайту Terraform. На диску C зробимо директорію Terrraform і помістимо туди скачаний та розпакований файл terraform.exe. Прописуємо в system environment variables в змінну Path директорію з terraform. Для перевірки встановлення Terraform виконаємо в командному рядку: terraform --version

Рис. 3.2. Версія terraform

Будемо використовувати Sublime. Встановимо плагін для terraform. Для цього обираємо Tools – Install Package Control. Тепер обираємо Preferences – Package Control. Обираємо Package Control: Add repository і вказуємо repository url (https://github.com/tmichel/sublime-terraform), знову в Preferences – Package Control обираємо Package Control : Install Package, вказуємо sublime-terraform.

Terraform можна завантажити на домашній сорінці проекту https://www.terraform.io. При цьому вибереться пакет для своєї операційної системи, зберігається ZIP-архів та розпаковується в папку для установки Terraform. Архів містить єдиний файл під назвою terraform, який необхідно добавити в змінну середовища PATH. Як варіант, можна пошукати Terraform в диспетчері пакетів ОС; например, в OS X можна виконати brewinstallterraform. Для того, щоб пересвідчитись, що все працює, запускаємо команду terraform. Має з'явитись інструкція: $ terraform Usage: terraform [-version] [-help] [args] Common commands: apply Builds or changes infrastructure console Interactive console for Terraform interpolations destroy Destroy Terraformmanaged infrastructure env Workspace management fmt Rewrites config files to canonical format (...) Для того, щоб система Terraform могла вносити зміни в обліковий запис AWS, необхідно прописати в змінну середовища AWS_ACCESS_KEY_ID и AWS_SECRET_ACCESS_KEY облікові дані для користувача IAM. Наприклад, в терміналі Unix/Linux/macOS: $ export AWS_ACCESS_KEY_ID=(your access key id) $ export AWS_SECRET_ACCESS_KEY=(your secret access key) .

3.1.2. Створення ресурсів

Для початку роботи маємо вказати з якою Cloud-платформою ми працюємо, в даному випадку – з ASW. Для цього вказуємо:

```
provider "aws" {}
```

Для того, щоб створити ресурс потрібно виконати наступне. Спершу визначаємо що саме будемо створювати. Є різні типи ресурсів – відповідно до тих

обєктів та служб, з якими ми працюємо. Для прикладу, створимо інстанс. Для створення використовується синтаксис: resource “тип” “імя”{список параметрів }. Мінімальний список параметрів складається із параметрів ami та instance_type. Ami визначається регіоном та ОС. instance_type – це відповідно до класифікації aws - тип сервера, що залежить від базових параметрів, наприклад, обємом оперативної памяті і т.д. – основними характеристиками.

```
resource "aws_instance" "my_Amazon" {
  ami           = "ami-03a71cec707bfc3d7"
  instance_type = "t3.small"

  tags = {
    Name    = "My Amazon Server"
    Owner   = "Oleksandr Tatus"
    Project = "Diplom"
  }
}
```

Ми можемо відразу запустити цей *.tf файл – і отримаємо створення інстанса. Для запуску в cmd виконуємо наступні команди. Переходимо в каталог нашого проекту. Запускається:

terraform init (оскільки ми ще не працювали з terraform в даному проекті ініціалізуємо в поточному каталозі, будуть скачані необхідні ресурси і ми отримаємо підкаталог .terraform/)

terraform plan (покаже, що планується створити і що треба зробити, але безпосередньо виконувати не буде).

```
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.


------------------------------------------------------------------------

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.my_Ubuntu will be created
  + resource "aws_instance" "my_Ubuntu" {
      + ami                          = "ami-090f10efc254eaf55"
      + arn                          = (known after apply)
      + associate_public_ip_address  = (known after apply)
      + availability_zone            = (known after apply)
      + cpu_core_count               = (known after apply)
      + cpu_threads_per_core         = (known after apply)
      + get_password_data            = false
      + host_id                      = (known after apply)
      + id                           = (known after apply)
      + instance_state               = (known after apply)
      + instance_type                = "t3.micro"
      + ipv6_address_count           = (known after apply)
      + ipv6_addresses               = (known after apply)
      + key_name                     = (known after apply)
      + network_interface_id         = (known after apply)
      + password_data                = (known after apply)
      + placement_group              = (known after apply)
      + primary_network_interface_id = (known after apply)
```

Рис. 3.3. Terraform plan

terraform apply (створення в нашому аккаунті в вказаному регіоні все, що вказано в скріпті terraform)

В результаті отримаємо запуск інстанса:

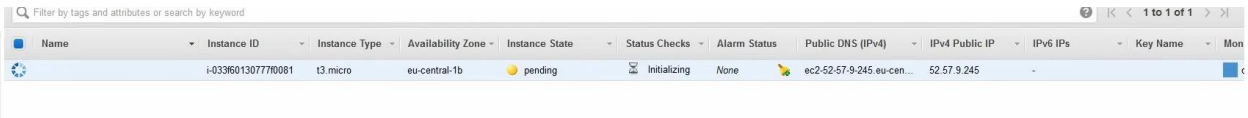


Рис. 3.4. Запуск інстанса

3.1.3. Data Source

Data Source згідно із офіційною документацією дозволяє нам збирати інформацію із різних ресурсів. У кожної платформа свої різні data_sources. Для

AWS їх можна переглянути по url https://registry.terraform.io/providers/hashicorp/aws/latest/docs/data-sources/ami.

В даній роботі використовуються, зокрема наступні.

Data Source: aws_availability_zones – інформація про те, які availability зони існують в даному регіоні. Використовуєть з наступним синтаксисом: data "aws_availability_zones" "working" {}. Для перегляду можна використати output.

```
output "data_aws_availability_zones" {
  value = data.aws_availability_zones.working.names
}
```

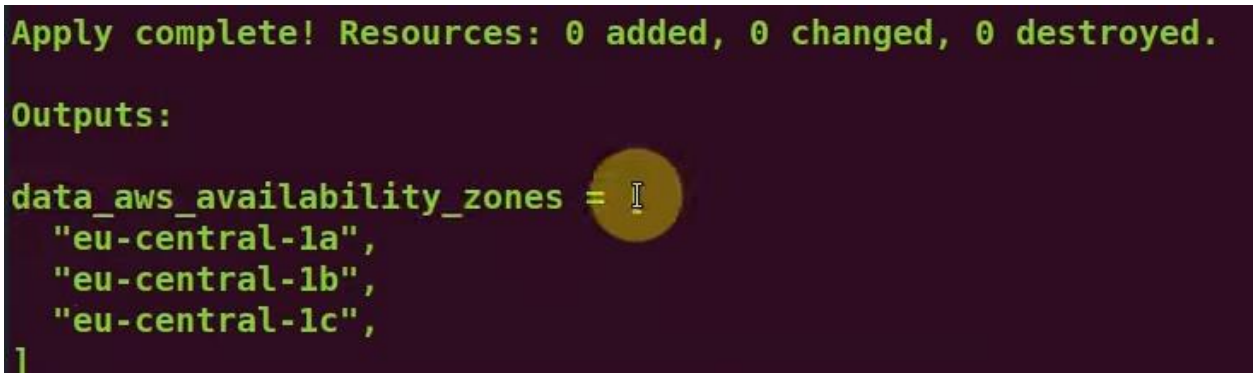


Рис. 3.5. Результат для output "data_aws_availability_zones"

Data Source: aws_caller_identity дає нам три важливі речі, зокерма account_id. Дуже часто використовується, лслбливо коли ми створюємо IAM policies. Синтаксис оголошення: data "aws_caller_identity" "current" {}. Також додамо output:

```
output "data_aws_caller_identity" {
  value = data.aws_caller_identity.current.account_id
}
```

Data Source: aws_region. Аналогічно як і для двох попередніх – синтаксис: data "aws_region" "current" {}. В виводі будемо використовувати name та description.

```
output "data_aws_region_name" {
  value = data.aws_region.current.name
}
output "data_aws_region_description" {
  value = data.aws_region.current.description
}
```

Data Source: aws_vpcs – читає всі vpc які є в нашому регіоні та для прикладу, можна вивести ids їх усіх.

```
data "aws_vpcs" "my_vpcs" {}
output "aws_vpcs" {
  value = data.aws_vpcs.my_vpcs.ids
}
```

Для прикладу, якщо в проекті є наступні VPC (на наступному рисунку), то код, записаний вище, дасть наступний результат (рис. 3.7).

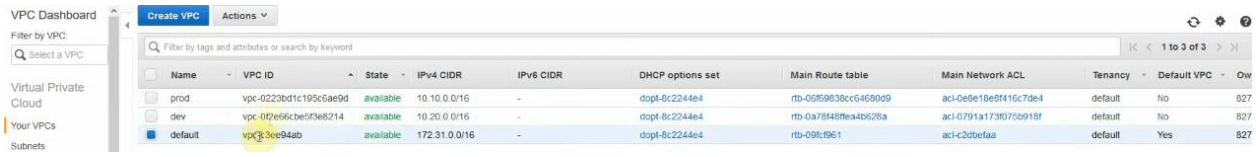


Рис. 3.6. VPC

```
Outputs:

aws_vpcs = [
  "vpc-0223bd1c195c6ae9d",
  "vpc-0f2e66cbe5f3e8214",
  "vpc-c3ee94ab",
]
```

Рис. 3.7. Output aws_vpcs

Нехай в нас є створене VPC. Нам необхідно створити там дві subnet: subnet1, subnet2. ID VPC не відомий, але відомий тег “prod”. Має працювати для будь-якого регіону. Для aws_vpc будемо використовувати для фільтрації тегі.

```
data "aws_vpc" "prod_vpc" {
```

```
  tags = {
    Name = "prod"
  }
}
```

Для створення subnet нам треба спершу створити ресурс "aws_subnet". Для створення ми маємо вказати vpc_id. availability_zone уже відомий – з попередніх розглянутих нами data sources.

```
resource "aws_subnet" "prod_subnet_1" {
  vpc_id            = data.aws_vpc.prod_vpc.id
  availability_zone = data.aws_availability_zones.working.names[0]
  cidr_block        = "10.10.1.0/24"
  tags = {
    Name    = "Subnet-1 in ${data.aws_availability_zones.working.names[0]}"
    Account = "Subnet in Account ${data.aws_caller_identity.current.account_id}"
    Region  = data.aws_region.current.description
  }
}
```

Абсолютно аналогічно для subnet2.

Результат виконання для ствонення двох subnets:

Name	Subnet ID	State	VPC	IPv4 CIDR	Available IPv4	IPv6 CIDR	Availability Zone	Availability Zone ID
Subnet-1 in eu-central-1a	subnet-03718873aaf86de7d	available	vpc-0223bd1c195c6ae9d...	10.10.1.0/24	251	-	eu-central-1a	euc1-az2
Subnet-2 in eu-central-1b	subnet-0f621c2afcfbac867	available	vpc-0223bd1c195c6ae9d...	10.10.2.0/24	251	-	eu-central-1b	euc1-az3
	subnet-44aaa82c	available	vpc-c3ee94ab | default	172.31.16.0/20	4091	-	eu-central-1a	euc1-az2
	subnet-87430cfd	available	vpc-c3ee94ab | default	172.31.32.0/20	4091	-	eu-central-1b	euc1-az3

Рис. 3.8. Subnet’s

3.1.4. Автопошук AMI id

Коли ми запускаємо інстанс, ми використовуємо ami-*. Або власний, або public. Коли версія ОС обновляється – ami міняється. Коли ми переходимо в інший регіон – ami знову міняється. Тому для роботи дуже зручно написати код, який автоматично буде відшукувати необхідний ami.

Рис. 3.9. Приклад ami-*

Для оптимізації можемо використати data source: aws_ami. Напишемо код. Спочатку визначимо регіон, наприклад, ap-southeast-2.

```
provider "aws" {
  region = "ap-southeast-2"
}
```

Тепер давайте опишемо необхідний ресурс. З використанням фільтра по values виконується вибірка із акаунта з id = owners[i], i = 1..len(owners) і вкінці серед вибірки ми обираємо most_recent.

```
data "aws_ami" "latest_ubuntu" {
  owners      = ["099720109477"]
  most_recent = true
  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-bionic-19.04-amd64-server-*"]
  }
}
```

Тепер використати описаний код можна наступним чином:

```
resource "aws_instance" "my_webserver_with_latest_ubuntu_ami" {
  ami           = data.aws_ami.latest_ubuntu.id
  instance_type = "t3.micro"
}
output "latest_ubuntu_ami_id" {
  value = data.aws_ami.latest_ubuntu.id
```

```
}

output "latest_ubuntu_ami_name" {
  value = data.aws_ami.latest_ubuntu.name
}
```

3.2. Створення Web Server з Terraform

Для реалізації нам потрібно створити фактично два ресурса – ec2 та security group, яка потрібна для того, щоб відкрити порти, наприклад порти 80 та 443.

Задамо регіон, в якому будемо запускати інстанс – нехай буде eu-central-1, це Frankfurt.

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

Перший ресурс, який треба створити – це інстанс. Для нього потрібно визначити додаткові дії – установку apache, web server, запуск apache та ще деякі дії по розміщенню сторінки.

```
resource "aws_instance" "my_webserver" {
  ami                 = "ami-03a71cec707bfc3d7"
  instance_type       = "t3.micro"
        ...
```

Через параметр vpc_security_group_ids можна для конкретного інстанса приєднати security group і ми приєднаємо ту, яку створюємо в цьому ж *.tf файлі. В даному випадку отримати доступ до ID групи можна так: aws_security_group.my_webserver.id і відразу ми отримали залежність, що спочатку буде створена групова політика, а потім інстанс, який містить цю групову політику.

```
  vpc_security_group_ids = [aws_security_group.my_webserver.id]
```

...

Також треба прописати команди, які будуть автоматично запускатися при створенні серевера, тобто bootstrapping. Робиться в AWS це через параметр user_data . Цьому параметру ми задаємо в даному випадку – команди bash скріпт, оскільки серевер працюватиме в нас під Linux. Перша команда – обновити Linux: yum -y update, потім інсталюємо apache server: yum -y install httpd, потім читаємо в амазоні локальний внутрішній IP-адрес сервера і потім робимо echo у вказаний файл. Потім стартуємо apache і робимо щоб він статрував завжди при запуску серевера.

Всі команди ми можемо прописати як напряму в *.tf файлі, так і, що значно краще – в статичному файлі, а в *.tf файлі для user_data використати функцію: file("user_data.sh"). Або можемо використати динамічні зовнішні файли, якщо приміром нам треба передати в код якісь параметри чи змінні дані. Тоді будемо використовувати функцію templatefile(параметри).

```
  user_data            = <<EOF
#!/bin/bash
yum -y update
yum -y install httpd
myip=`curl http://169.254.169.254/latest/meta-data/local-ipv4`
echo "<h2>WebServer with IP: $myip</h2><br>Build by Terraform!" > /var/www/html/index.html
sudo service httpd start
chkconfig httpd on
EOF
```

Можемо також добавити деякі теги – для зручності роботи.

```
  tags = {
    Name  = "Web Server Build by Terraform"
```

```
  Owner = " Oleksandr Tatus "
 }
}
```

Другий ресурс – security group. Нам треба відкрити порти 80, 443 і доступ дозволимо з будь-яких адресів ["0.0.0.0/0"] , хоча можемо вказати тільки підмережу чи певний пул адресів. Код для групи буде наступним.

```
resource "aws_security_group" "my_webserver" {
 name        = "WebServer Security Group"
 description = "My First SecurityGroup"
 vpc_id      = aws_default_vpc.default.id

 ingress {
  from_port   = 80
  to_port     = 80
  protocol    = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
 }

 ingress {
  from_port   = 443
  to_port     = 443
  protocol    = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
 }

 egress {
  from_port   = 0
  to_port     = 0
```

```
  protocol    = "-1"
  cidr_blocks = ["0.0.0.0/0"]
 }
```

Таким чином, у лістингу вище для security group ми маємо два правила на incoming трафік, та одне – на outcoming.

Можемо виконати terraform plane, terraform apply.

```
Terraform will perform the following actions:

  # aws_instance.my_webserver will be created
  + resource "aws_instance" "my_webserver" {
      + ami                          = "ami-03a71cec707bfc3d7"
      + arn                          = (known after apply)
      + associate_public_ip_address  = (known after apply)
      + availability_zone            = (known after apply)
      + cpu_core_count               = (known after apply)
      + cpu_threads_per_core         = (known after apply)
      + get_password_data            = false
      + host_id                      = (known after apply)
      + id                           = (known after apply)
      + instance_state               = (known after apply)
      + instance_type                = "t3.micro"
      + ipv6_address_count           = (known after apply)
      + ipv6_addresses               = (known after apply)
      + key_name                     = (known after apply)
      + network_interface_id         = (known after apply)
      + password_data                = (known after apply)
      + placement_group              = (known after apply)
      + primary_network_interface_id = (known after apply)
      + private_dns                  = (known after apply)
      + private_ip                   = (known after apply)
      + public_dns                   = (known after apply)
      + public_ip                    = (known after apply)
      + security_groups              = (known after apply)
      + source_dest_check            = true
      + subnet_id                    = (known after apply)
      + tenancy                      = (known after apply)
      + user_data                    = "4604abd4091c8460d10be1c32dd847cb90f47233"
```

Рис. 3.10. Terraform apply

```
# aws_security_group.my_webserver will be created
+ resource "aws_security_group" "my_webserver" {
    + arn                    = (known after apply)
    + description            = "My First SecurityGroup"
    + egress                 = [
        + {
            + cidr_blocks      = [
                + "0.0.0.0/0",
              ]
            + description      = ""
            + from_port        = 0
            + ipv6_cidr_blocks = []
            + prefix_list_ids  = []
            + protocol         = "-1"
            + security_groups  = []
            + self             = false
            + to_port          = 0
        },
      ]
    + id                     = (known after apply)
    + ingress                = [
        + {
            + cidr_blocks      = [
                + "0.0.0.0/0",
              ]
            + description      = ""
            + from_port        = 443
            + ipv6_cidr_blocks = []
            + prefix_list_ids  = []
```

Рис. 3.11. Create security group with terraform

Результат можна побачити в консолі AWS. Спершу створюється нова security group:

Name	Group ID	Group Name	VPC ID	Description
	sg-02656dd1d9899696f	WebServer Security Group	vpc-c3ee94ab	My First SecurityGroup
	sg-f385d998	default	vpc-c3ee94ab	default VPC security group

Рис. 3.12. Результат в консолі

Та відповідно інстанс:

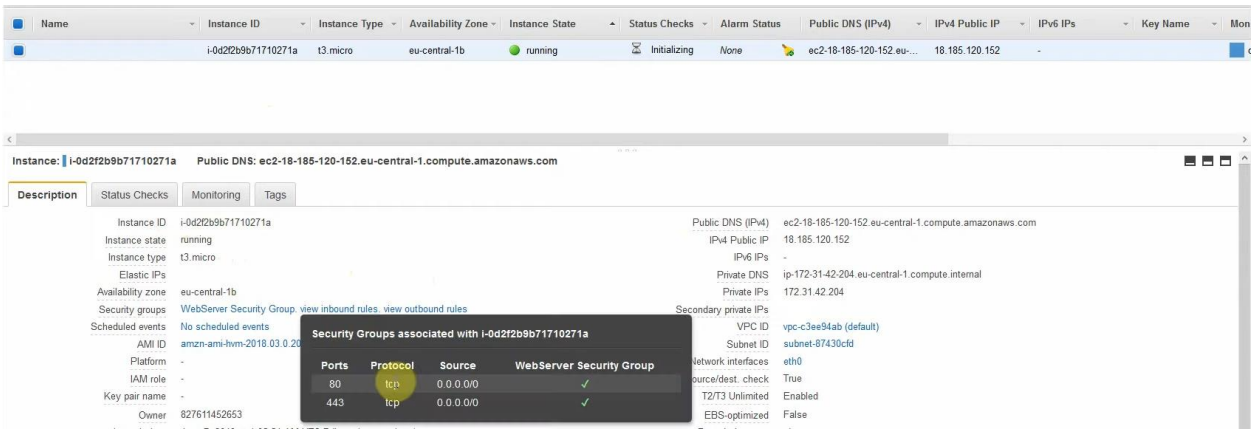


Рис. 3.13. Запущений інстанс

3.3. ZeroDowntime Green/Blue Deployment на прикладі створення Web Cluster

3.3.1. Побудова моделі

Розглянемо можливості terraform для побудови ASG + LaunchTemplate + ALB.

Нехай маємо наступну модель. Є веб-ресурс, для якого потрібно реалізувати модель ZeroDowntime. Для реалізації такої моделі можемо виконати наступне. Створюємо Application Load Balancer. Для ALB потрібний ALB TargetGroup. Він являє собою обєкт, де будуть реєструватися сервери, через які буде проходити трафік. Він буде містити EC2 Launch Template V1. Auto Scaling V1 створить нам необхідну визначену кількість EC2 серверів. Група серверів V1 відноситься до Green Group серверів.

Спочатку увесь трафік йде через сервери з Green group. Коли ж на сервери потрібно внести якісь зміни – тоді запускається модель Green/Blue Deployment по мінімізації часу, коли веб-ресурс не працює із-за наприклад, перезавантаження серверів.

Створимо EC2 Launch Template V2. Створимо також Auto Scaling V2. V2 – Blue group. Усі зміни вносяться на сервери з Blue Group. В деякий проміжок часу користувачі зможуть потрапляти і на сервери групи V1, і на сервери з групи V2. Але як тільки сервери Blue Group пройдуть всі check's перевірки, тобто інсталюється все необхідне ПЗ, запустяться служби і т.д. – трафік до Green Group зупинеться, Auto Scaling V1 почне самознищуватися – спочатку самі сервери, а потім і сама Auto Scaling Group. І тепер користувачі будуть переходити тільки на V2 – Blue Group з оновленими змінами,я кі були необхідні. Після цього Blue Deployment автоматично переходить в Green Deployment.

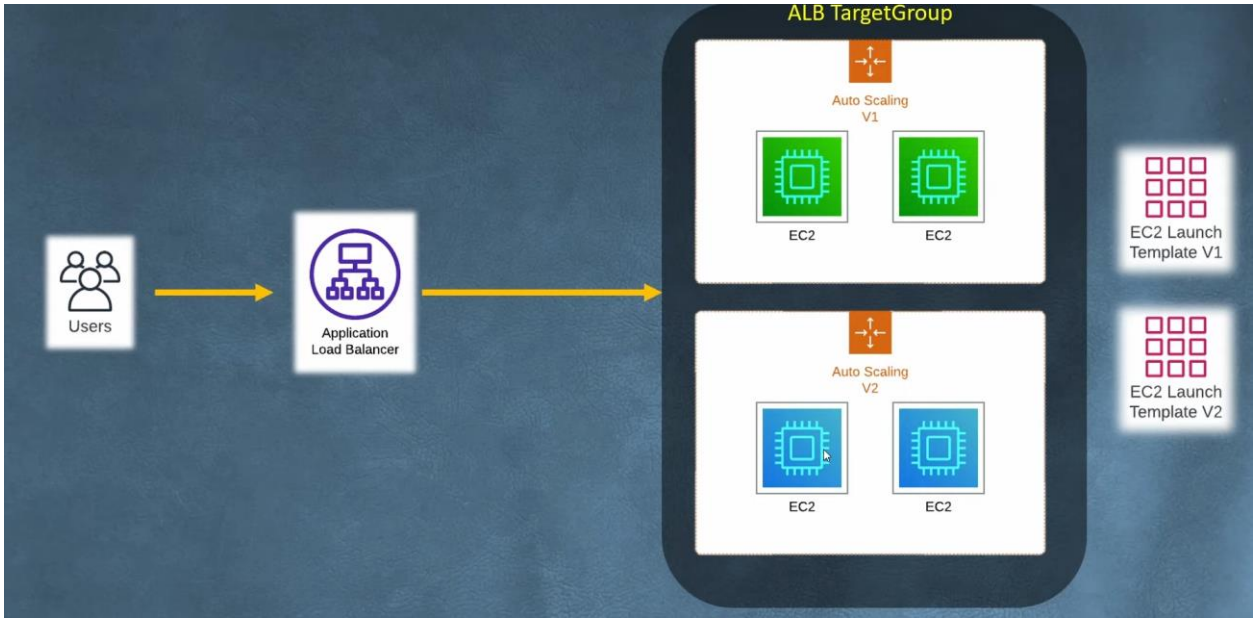


Рис. 3.14. Green/Blue Deployment

3.3.2. Скрипт ініціалізації

Для реалізації моделі з terraform необхідно виконати наступне.

Спочатку створюємо user_data.sh – скріпт, який буде описувати, яке ПЗ має бути встановлене на кожному сервері. Ми будемо робити щось на зразок Web Cluster, де будемо працювати з багатьма серверами та LoadBalancer. Ми будемо працювати з Amazon Linux – оскільки в даних images є наперед встановлене

необхідне нам ПЗ, що спростить нам роботу – ми встановимо тільки те, чого не вистачає.

```
#!/bin/bash
yum -y update
yum -y install httpd
myip=`curl http://169.254.169.254/latest/meta-data/local-ipv4`
cat <<EOF > /var/www/html/index.html
<html>
<body bgcolor="black">
<h2><font color="gold">Build by Power of Terraform <font color="red"> v0.12</font></h2><br><p>
<font color="green">Server PrivateIP: <font color="aqua">$myip<br><br>
<font color="magenta">
<b>Version 3.0</b>
</body>
</html>
EOF
sudo service httpd start
chkconfig httpd on
```

В даному скрипті ми інсталюємо apache, беремо поточний private_ip адрес, створюємо простеньку html-сторінку, запускаємо httpd та налаштовуємо старт apache при запуску сервера.

Створюємо main.tf файл. Спершу конфігуруємо провайдера:

```
provider "aws" {
  region = "ca-central-1"
  default_tags {
    tags = {
      CreatedBy = "Terraform"
```

```
  }
 }
}
```

default_tags будуть застосовуватися до всіх ресурсів, які підтримують теги за замовчуванням.

Будемо використовувати Data Sources, а саме availability_zones, тобто в яких зонах доступності будемо працювати. Тому читаємо поточні availability_zones, а також знаходимо last Amazon Linux ami:

```
data "aws_availability_zones" "working" {}
data "aws_ami" "latest_amazon_linux" {
 owners      = ["137112412989"]
 most_recent = true
 filter {
  name   = "name"
  values = ["amzn2-ami-hvm-*-x86_64-gp2"]
 }
}
```

Тепер потрібно навіть якщо ми хочемо в Amazon використовувати Default VPC, потрібно її все-рівно прописати тут, у цьому файлі проекту, хоча ми її і не створюємо: resource "aws_default_vpc" "default" {}.

Прописуємо subnets:

```
resource "aws_default_subnet" "default_az1" {
 availability_zone = data.aws_availability_zones.working.names[0]
}
resource "aws_default_subnet" "default_az2" {
 availability_zone = data.aws_availability_zones.working.names[1]
}
```

3.3.3. Security group

Наступне – створюємо ресурси. Перший ресурс – security_group, оскільки для створення інстансів, і Load Balancer – дял усього потрібна групова політика. Тому створюємо ресурс:

```
resource "aws_security_group" "web" {
  name   = "Web Security Group"
  vpc_id = aws_default_vpc.default.id
  dynamic "ingress" {
    for_each = ["80", "443"]
    content {
      from_port   = ingress.value
      to_port     = ingress.value
      protocol    = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
    }
  }
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
  tags = {
    Name = "Web Security Group"
  }
}
```

Тут ми обовязково взазуємо vpc_id, відкриваєм два порти 80 та 443 для будь-якого адреса для ingress rule. А для egress – всі порти відкриті для будь-яких IP.

3.3.4. Launch Template

Тепер створюємо Launch Template. У Launch Template будуть змінюватися версії. Ми беремо image Amazon Linux, вказуємо параметри, тип, групову політику та user_data, куди будуть записані команди для авто-запуску на всіх серверах.

```
resource "aws_launch_template" "web" {
  name                   = "WebServer-Highly-Available-LT"
  image_id               = data.aws_ami.latest_amazon_linux.id
  instance_type          = "t3.micro"
  vpc_security_group_ids = [aws_security_group.web.id]
  user_data              = filebase64("${path.module}/user_data.sh")
}
```

Тепер створюємо Autoscaling Group. Назва Autoscaling Group буде мінятися. health_check_type вказуємо Elastic Load Balancer. У vpc_zone_identifier пропусуємо наші створені subnets.

```
resource "aws_autoscaling_group" "web" {
  name                                  = "WebServer-Highly-Available-ASG-Ver-${aws_launch_template.web.latest_version}"
  min_size             = 2
  max_size             = 2
  min_elb_capacity     = 2
  health_check_type    = "ELB"
  vpc_zone_identifier                   = [aws_default_subnet.default_az1.id, aws_default_subnet.default_az2.id]
  target_group_arns    = [aws_lb_target_group.web.arn]
```

Вказуємо який Launch Template використовувати – обираємо власний створений.

```
  launch_template {
```

```
    id      = aws_launch_template.web.id
    version = aws_launch_template.web.latest_version
  }
```

Добавляємо динамічні теги.

```
  dynamic "tag" {
    for_each = {
      Name   = "WebServer in ASG-v${aws_launch_template.web.latest_version}"
      TAGKEY = "TAGVALUE"
    }
    content {
      key                 = tag.key
      value               = tag.value
      propagate_at_launch = true
    }
  }
```

Важливо вказати, що ми будемо працювати по принципу: нове створюється перед тим, як буде видалено старе. Для цього прописуємо:

```
  lifecycle {
    create_before_destroy = true
  }
}
```

3.3.5. Load Balancer

Залишилися Load Balancer, Target Group та Load Balancer Listener. Напишемо для них наступний код.

```
resource "aws_lb" "web" {
  name               = "WebServer-HighlyAvailable-ALB"
  load_balancer_type = "application"
```

```
  security_groups    = [aws_security_group.web.id]
  subnets                                    =    [aws_default_subnet.default_az1.id,
aws_default_subnet.default_az2.id]
}
```

Load Balancer не надсилає просто так трафік, йому потрібний target group. Він буде на порту 80. Якщо не вказати deregistration_delay – може появитися невеличкий downtime.

```
resource "aws_lb_target_group" "web" {
  name                = "WebServer-HighlyAvailable-TG"
  vpc_id              = aws_default_vpc.default.id
  port                = 80
  protocol            = "HTTP"
  deregistration_delay = 10 # seconds
}
```

На Load Balancer потрібно створювати Listener, зокрема щоб вказати на якому порту цей Load Balancer прослуховує трафік і куди трафік анправити.

```
resource "aws_lb_listener" "http" {
  load_balancer_arn = aws_lb.web.arn
  port              = "80"
  protocol          = "HTTP"
  default_action {
    type            = "forward"
    target_group_arn = aws_lb_target_group.web.arn
  }
}
```

До запуску кода в aws можна побачити, що немає ніяких серверів, Launche Template також немає:

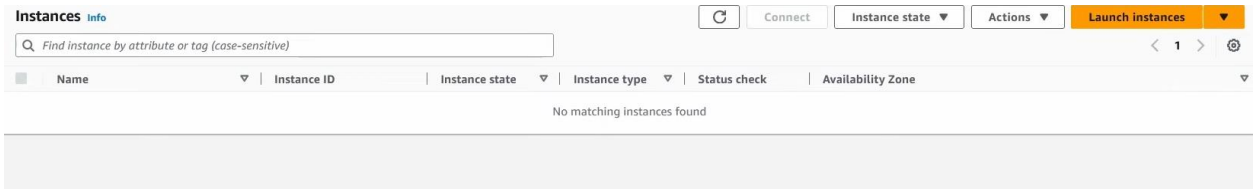


Рис. 3.15. Стан до запуска

Запускаємо в terraform наш файл з кодом. Тепер можна переглянути створені ресурси.

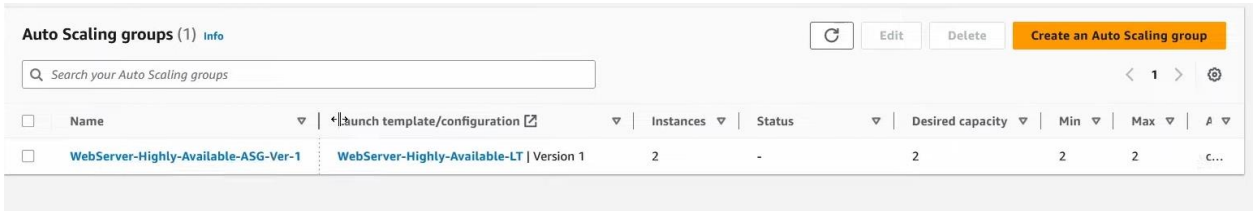


Рис. 3.16. Auto Scaling Group

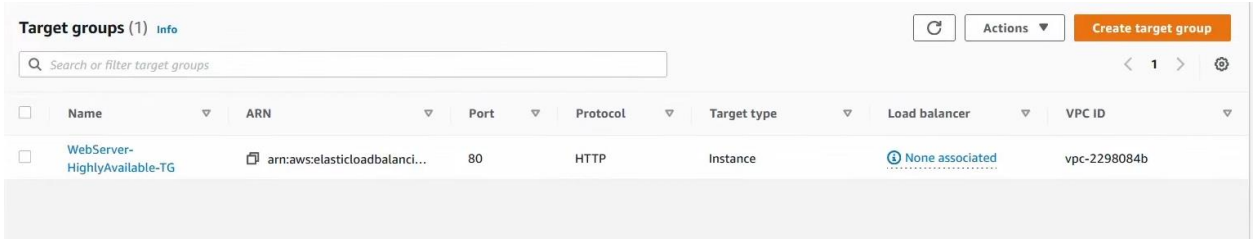


Рис. 3.17. Target Group

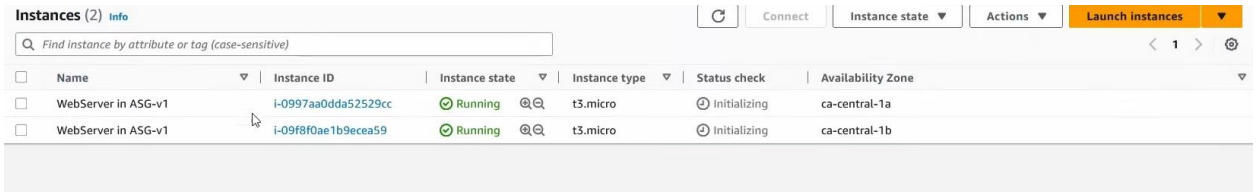


Рис. 3.18. Сервера

Якщо тепер ми внесемо зміни, наприклад, в html – код сторінки, що розміщується на веб-серевері, то виконаються автоматично наступні дії: replace autoscaling_group, міняється launche template.

РОЗДІЛ 4
ПРОГРАМНА РЕАЛІЗАЦІЯ

4.1. Особливості задачі

Створимо код pipeline з використанням terraform, який буде розгортати наш додаток, написаний на Python, в AWS ECS . Код ми розмістимо в AWS S3 Bucket.

Для реалізації створимо конвеєр pipeline, який буде містити code command CodeBuild та CodeDeploy. Як це реалізувати без terraform – ми описали в пунктах 2.4 – 2.5. Тепер наша задача автоматизувати увесь процес з використанням terraform.

Код додатка можна розмістити наприклад в github, або ж ми можемо використати наш акаунт AWS та скористатися S3 Bucket. Це хороша альтернатива для розміщення коду в хмарі. Вона дозволяє ефективно організовувати роботу в проекті.

Увесь проект поділимо на дві частини – робота з S3 Bucket та безпосередньо pipeline. Створимо для кожної частини свій підкаталог.

4.2. AWS S3 Bucket

Для зберігання змінних використаємо variables. Їх будемо зберігати в окремому файлі variables.tf. Взагалі структура проекта буде наступна. Два підкаталога. Для роботи з бакетами маємо:

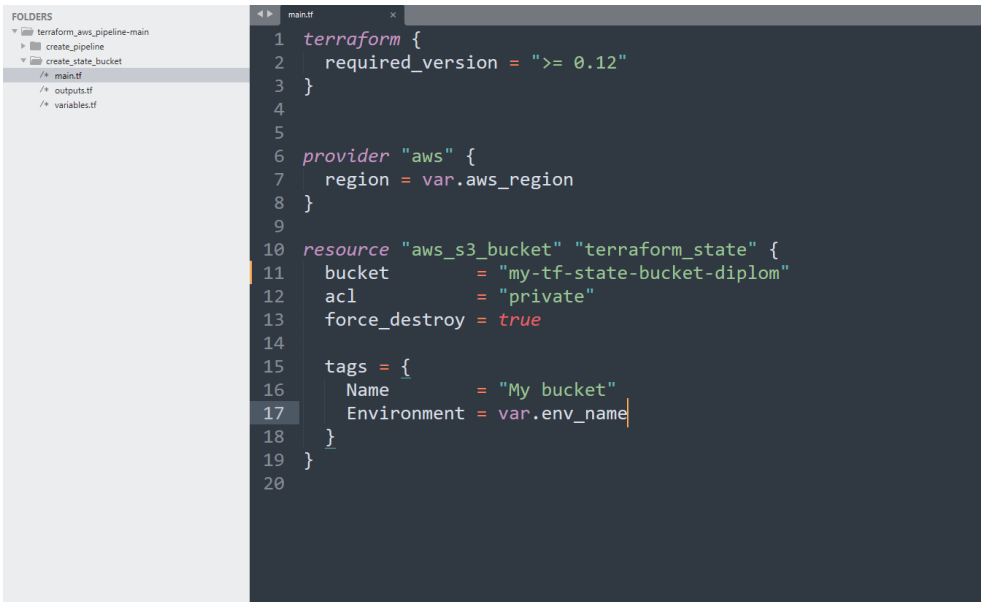


Рис. 4.1. Aws s3

Опишемо AWS S3 Bucket. В main.tf спочатку задамо регіон через variables, потім класично створюємо бакет, задаємо параметри. Всі variables будемо зберігати окремо у файлі variables.tf у форматі:

```
variable "aws_region" {
  description = "AWS region to provision"
  default     = "us-west-2"
}
```

Будемо працювати у регіоні us-west-2, хоча можемо обрати будь-який інший.

Зокрема, вкажемо в параметрі bucket назву нашого бакета. Якщо цей параметр опустити, Terraform призначить випадкову унікальну назву.

Задамо force_destroy як TRUE для того, щоб всі об’єкти (включаючи будь-які заблоковані об’єкти) були видалені з бакета, коли бакет буде знищено, щоб цей бакет можна було знищити без помилок. Ці об'єкти не підлягають відновленню.

Також задамо tags – декілька тегів.

Ми будемо фактично використовавати цей бакет для збереження стану terraform.

Можемо окремо запустити цей файл на виконання і terraform створить нам наш бакет. Виконаємо terraform init. Потім terraform apply і подивимося результат.

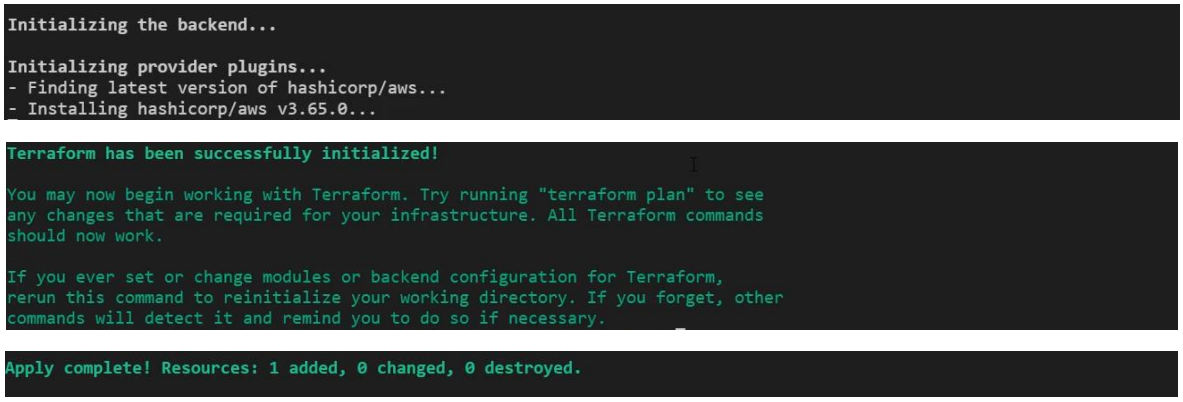


Рис. 4.2. Ініціалізація terraform init

В output.tf просто задамо один додатковий вивід з іменем бакета:

```
output "tf-state-bucket-name"
{
  value = aws_s3_bucket.terraform_state.bucket
}
```

4.3. Pipeline

4.3.1. Реалізація

Структура буде наступна. Фактично в файлі з реалізацією конвеєра ми поєднуємо все, що написали в проекті в одне ціле.

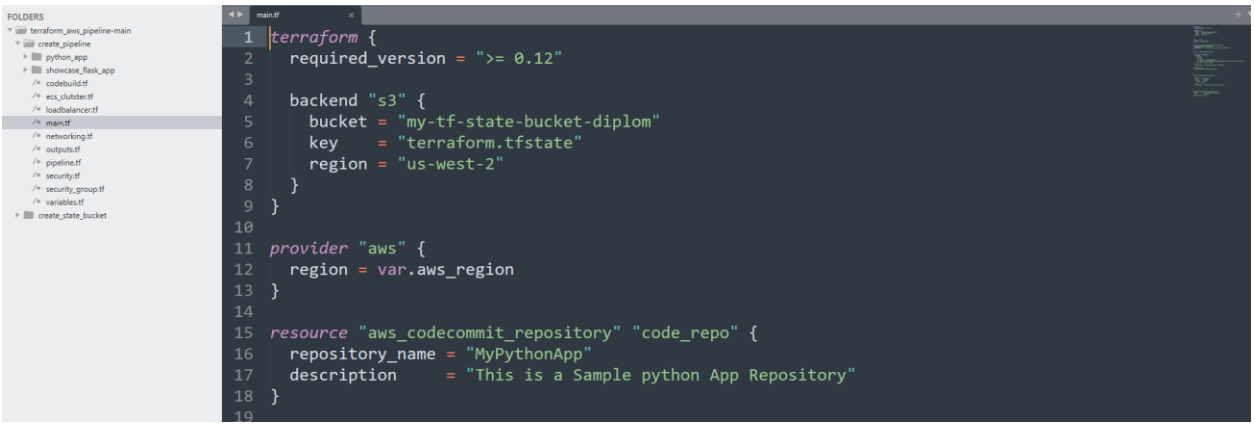


Рис. 4.3. Pipeline

Усі змінні поміщаємо в variables.tf. Із найважливіших – пишем свій ACCOUNT_ID, зберігається в environment :

ACCOUNT_ID = 1111111111111

В основному файлі ми вказуємо імя нашого створеного бакета:

```
backend "s3" {
   bucket = "my-tf-state-bucket-diplom"
   key    = "terraform.tfstate"
   region = "us-west-2"
  }
```

Далі ми вказуємо регіон, задаємо його через var’s:

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

Для зручності та ефективності розробки ми створюємо codecommit repository:

```
resource "aws_codecommit_repository" "code_repo" {
 repository_name = "MyPythonApp"
 description     = "This is a python App Repository"
}
```

Далі працюємо з provisioner, який дозволяє нам запускати на виконання команди terraform локально на нашій машині. Ми вказуємо робочий каталог – він є підкаталогом нашого проекту – python_app:

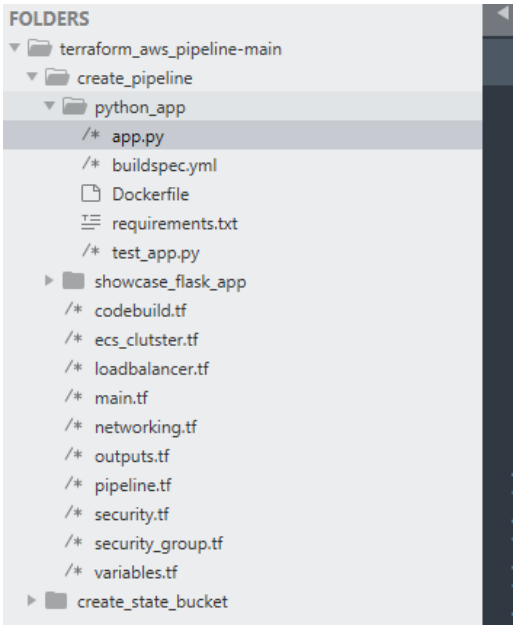


Рис. 4.4. python_app

```
provisioner "local-exec" {
   command     = <<EOF
      git init
      git add .
      git commit -m "Initial Commit"
      git remote add origin ${aws_codecommit_repository.code_repo.clone_url_http}
      git push -u origin master
  EOF
   working_dir = "python_app"
 }
 depends_on = [
   aws_codecommit_repository.code_repo,
 ]
```

В командах ми ініціалізуємо git та додаємо в репозиторій усе необхідне. Тут ми задаємо що будемо працювати з репозиторієм, який ми створюємо в AWS, який ми описали вище. Тобто ми спершу маємо створити aws репозиторій перше ніж зможемо зафіксувати наші зміни.

Також створимо s3 bucket щоб зберігати так звані artefacts, які будуть фактично створюватись на наному конвеєрі, який ми створюємо.

```
resource "aws_s3_bucket" "cicd_bucket" {
  bucket        = var.artifacts_bucket_name
  acl           = "private"
  force_destroy = true
}
```

Напишемо наступний ресурс для реалізації нашого pipeline. Спершу визначаємо ресурс з відповідними параметрами – name, role і можна задати теги.

```
resource "aws_codepipeline" "python_app_pipeline" {
  name     = "python-app-pipeline"
  role_arn = aws_iam_role.apps_codepipeline_role.arn
  tags = {
    Environment = var.env
  }
```

Далі вказуємо artifact_store, який ми визначили перед цим.

```
  artifact_store {
    location = var.artifacts_bucket_name
    type     = "S3"
  }
```

Також тут можна побачить поетапно усе, що відбувається. Спершу маємо наш вихідний етап Stage "Source", де маємо вхідний код, який надходить в наш конвеєр. Ми посилаємося на назву нашого сховища - "RepositoryName" = aws_codecommit_repository. code_ repo. Repo sitory _ name.

```
stage {
  name = "Source"
  action {
    category = "Source"
    configuration = {
      "BranchName" = var.python_project_repository_branch
      "RepositoryName" = aws_codecommit_repository.code_repo.repository_name
    }
    input_artifacts = []
    name             = "Source"
    output_artifacts = [
      "SourceArtifact",
    ]
    owner     = "AWS"
    provider  = "CodeCommit"
    run_order = 1
    version   = "1"
  }
}
```

Наступний етап Stage "Build". Тут маємо посилання на наші змінні, тобто цей етап посилається на наш файл збірки buildspec.yml і якщо переглянемо в ньому відповідний етап то побачимо результат, зображений на Рис.4.5

```
build:
  commands:
    - echo Build started on `date`
    - echo Building the Docker image...
    - docker build -t $IMAGE_REPO_NAME:$IMAGE_TAG .
    - docker tag $IMAGE_REPO_NAME:$IMAGE_TAG $AWS_ACCOUNT_ID.dkr.ecr.$AWS_DEFAULT_REGION.ama
    - echo scanning image for vunerability
    -  #docker run --rm -v /root/.cache/:/root/.cache/ aquasec/trivy:0.18.3 --exit-code=1 $I
```

Рис. 4.5. Buildspec.yml

```
stage {
  name = "Build"

  action {
    category = "Build"
    configuration = {
      "EnvironmentVariables" = jsonencode(
        [
          {
            name  = "environment"
            type  = "PLAINTEXT"
            value = var.env
          },
          {
            name  = "AWS_DEFAULT_REGION"
            type  = "PLAINTEXT"
            value = var.aws_region
          },
          {
            name  = "AWS_ACCOUNT_ID"
            type  = "PARAMETER_STORE"
            value = "ACCOUNT_ID"
          },
          {
            name  = "IMAGE_REPO_NAME"
            type  = "PLAINTEXT"
            value = aws_ecr_repository.python_app_repo.name
          },
```

```
        {
          name  = "IMAGE_TAG"
          type  = "PLAINTEXT"
          value = "latest"
        },
        {
          name  = "CONTAINER_NAME"
          type  = "PLAINTEXT"
          value = var.container_name
        },
      ]
    )
    "ProjectName" = aws_codebuild_project.containerAppBuild.name
  }
  input_artifacts = [
    "SourceArtifact",
  ]
  name = "Build"
  output_artifacts = [
    "BuildArtifact",
  ]
  owner     = "AWS"
  provider  = "CodeBuild"
  run_order = 1
  version   = "1"
 }
}
```

Далі переходимо в Deploy. Тут вказуємо ecs імя, яке ми створили, python_service.name, яке також ми визначили раніше:

```
 stage {
   name = "Deploy"
   action {
    category = "Deploy"
    configuration = {
     "ClusterName" = aws_ecs_cluster.python_app_cluster.name
     "ServiceName" = aws_ecs_service.python_service.name
     "FileName"    = "imagedefinitions.json"
     #"DeploymentTimeout" = "15"
    }
    input_artifacts = [
     "BuildArtifact",
    ]
    name             = "Deploy"
    output_artifacts = []
    owner            = "AWS"
    provider         = "ECS"
    run_order        = 1
    version          = "1"
   }
 }

 depends_on = [
   aws_codebuild_project.containerAppBuild,
   aws_ecs_cluster.python_app_cluster,
   aws_ecs_service.python_service,
```

```
    aws_ecr_repository.python_app_repo,
    aws_codecommit_repository.code_repo,
    aws_s3_bucket.cicd_bucket,
  ]
}
```

4.3.2. Networking

Спочатку для роботи нам треба створити VPC, дві subnets.

```
resource "aws_vpc" "web_vpc" {
  cidr_block           = var.cidr_block
  enable_dns_hostnames = true

  tags = {
    Name = "Web VPC"
  }
}

resource "aws_subnet" "private_subnet" {
  count  = 2
  vpc_id = aws_vpc.web_vpc.id
  cidr_block = cidrsubnet(var.cidr_block, 2, count.index)
  availability_zone = element(var.availability_zones, count.index)

  tags = {
    Name = "Private Subnet ${count.index + 1}"
  }
}
resource "aws_internet_gateway" "web_igw" {
```

```
  vpc_id = aws_vpc.web_vpc.id
}
```

Також визначаємо таблицю маршрутів шлюзу aws підмережі.

```
resource "aws_route_table" "public_rt" {
  vpc_id = aws_vpc.web_vpc.id
  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.web_igw.id
  }
  tags = {
    Name = "Public Subnet Route Table"
  }
}
resource "aws_subnet" "public_subnet" {
  count                   = 2
  vpc_id                  = aws_vpc.web_vpc.id
  cidr_block              = cidrsubnet(var.cidr_block, 2, count.index + 2)
  availability_zone       = element(var.availability_zones, count.index)
  map_public_ip_on_launch = true
  tags = {
    Name = "Public Subnet ${count.index + 1}"
  }
}
resource "aws_route_table_association" "public_subnet_rta" {
  count          = 2
  subnet_id      = aws_subnet.public_subnet.*.id[count.index]
  route_table_id = aws_route_table.public_rt.id
}
```

4.3.3. LoadBalancer

Створимо простий балансувальник навантаження. Задаємо ресурс, описуємо параметри:

```
resource "aws_lb" "loadbalancer" {
  name               = "test-lb-tf"
  internal           = false
  load_balancer_type = "application"
  security_groups    = [aws_security_group.allow_elb.id]
  subnets            = [aws_subnet.public_subnet[0].id, aws_subnet.public_subnet[1].id]

  enable_deletion_protection = false
  tags = {
    Environment = var.env
  }
}
```

В нас є target група визначена наступним чином:

```
resource "aws_lb_target_group" "lb_target" {
  name     = "tf-example-lb-tg"
  port     = 80
  protocol = "HTTP"
  vpc_id   = aws_vpc.web_vpc.id
  depends_on = [
    aws_lb.loadbalancer
  ]
}
resource "aws_lb_listener" "front_end" {
  load_balancer_arn = aws_lb.loadbalancer.arn
  port              = "80"
```

```
  protocol          = "HTTP"

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

Також визначаємо політику безпеки, в нас є одне правило на вхідні зєднання – відкриваєм порт 80 tcp та правило на вихідний трафік – без обмежень по портах чи адресах.

```
resource "aws_security_group" "allow_http" {
  name        = "allow_http"
  description = "Allow HTTP inbound traffic"
  vpc_id      = aws_vpc.web_vpc.id
  ingress = [
    {
      description      = "HTTP from VPC"
      from_port        = 80
      to_port          = 80
      protocol         = "tcp"
      cidr_blocks      = ["0.0.0.0/0"]
      ipv6_cidr_blocks = ["::/0"]
      prefix_list_ids  = null
      security_groups  = null
      self             = null
    }
  ]
```

```
  egress = [
    {
      description      = "HTTP to VPC"
      from_port        = 0
      to_port          = 0
      protocol         = "-1"
      cidr_blocks      = ["0.0.0.0/0"]
      ipv6_cidr_blocks = ["::/0"]
      prefix_list_ids  = null
      security_groups  = null
      self             = null
    }
  ]

  tags = {
    Name = "allow_http"
  }
}
```

4.4. Security Roles

Також нам треба створить security roles. Ми їх винесемо в окремий файл – security.tf та визначимо там усі необхідні ролі. Зокрема, для aws_iam_role визначимо роль "apps_codepipeline_role". Це все базові ролі – їх можна було би задавати через консоль, але з terraform вони визначаються і створюються значно швидше.

```
resource "aws_iam_role" "apps_codepipeline_role" {
  name = "apps-code-pipeline-role"
```

```
 assume_role_policy = <<EOF
{
 "Version": "2012-10-17",
 "Statement": [
   {
    "Effect": "Allow",
    "Principal": {
     "Service": "codepipeline.amazonaws.com"
    },
    "Action": "sts:AssumeRole"
   }
 ]
}
EOF
}
```

Всі ці політики містять налаштування що дозволено робити для конкретної ролі. Наприклад, для "apps_codepipeline_role_policy" дозволено "codecommit:CancelUploadArchive","codecommit:GetBranch","codecommit:GetCommit","codecommit:GetRepository","codecommit:GetUploadArchiveStatus", "codecommit:UploadArchive" і т.д. – дуже багато різних налаштувань.

```
resource "aws_iam_role_policy" "apps_codepipeline_role_policy" {
 name = "apps-codepipeline-role-policy"
 role = aws_iam_role.apps_codepipeline_role.id
 policy = <<EOF
{
   "Statement": [
      {
         "Action": [
```

```
            "iam:PassRole"
        ],
        "Resource": "*",
        "Effect": "Allow",
        "Condition": {
            "StringEqualsIfExists": {
                "iam:PassedToService": [
                    "cloudformation.amazonaws.com",
                    "elasticbeanstalk.amazonaws.com",
                    "ec2.amazonaws.com",
                    "ecs-tasks.amazonaws.com"
                ]
            }
        }
    },
    {
        "Action": [
            "codecommit:CancelUploadArchive",
            "codecommit:GetBranch",
            "codecommit:GetCommit",
            "codecommit:GetRepository",
            "codecommit:GetUploadArchiveStatus",
            "codecommit:UploadArchive"
        ],
        "Resource": "*",
        "Effect": "Allow"
    },
...
```

4.5. Результати

Після визначення усіх необхідних частин, можемо запустити pipeline.tf.

```
Initializing the backend...

Successfully configured the backend "s3"! Terraform will automatically
use this backend unless the backend configuration changes.
```

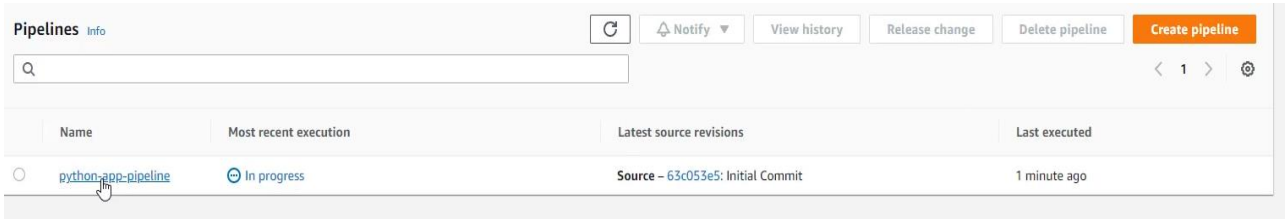
```
Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Рис. 4.6. Terraform init.

В AWS можемо перевірити наявність щойно створеного CodePipeline:



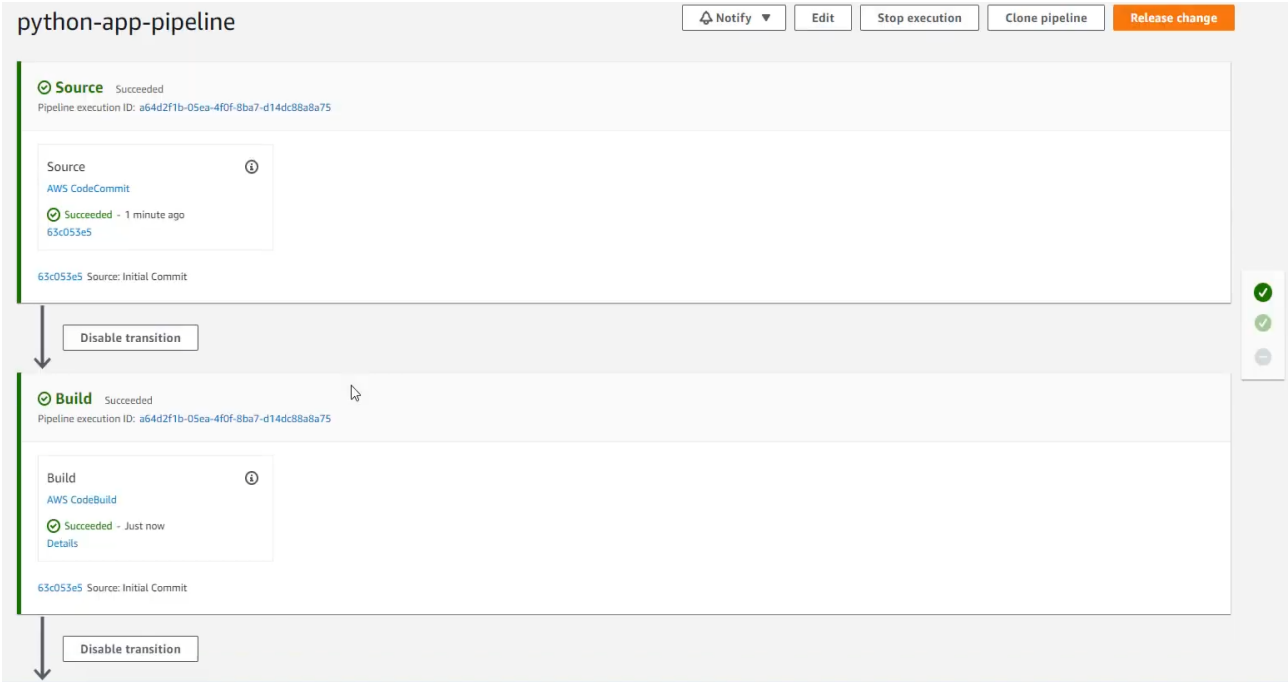


Рис. 4.7. Перевірка CodePipeline

ВИСНОВКИ

Розробка pipeline CI/CD з використанням Terraform та AWS є важливою задачею для забезпечення ефективності та надійності процесу розгортання програмного забезпечення. Використання Terraform та AWS дозволяє автоматизувати розгортання та керування інфраструктурою, а CI/CD pipeline забезпечує безперервну поставку та інтеграцію нових функцій у програмний продукт.

У роботі визначено вимоги до CI/CD pipeline. Це включало в себе вимоги до автоматизації розгортання, тестування, інтеграції та моніторингу програмного продукту.

Налаштовано AWS-акаунт та створено необхідні сервіси AWS для розгортання та керування інфраструктурою. Це включало створення інстансів (EC2 instances), мережевих налаштувань та інших ресурсів.

Розроблено конфігураційні файли Terraform, в яких визначено бажану структуру та параметри інфраструктури AWS. Ці файли містять опис ресурсів, їх взаємозв'язки та конфігураційні параметри.

Створено CI/CD pipeline за допомогою інструментів AWS, таких як AWS CodePipeline, AWS CodeBuild та AWS CodeDeploy. Цей pipeline включає етапи збирання коду, тестування, пакування та розгортання програмного продукту.

Налаштовано інтеграцію між CI/CD pipeline та Terraform. Це дозволяє автоматично розгортати та керувати інфраструктурою на основі змін в коді та конфігурації.

Виконано тестування та перевірку працездатності pipeline. Це включало запуск тестів на різних етапах pipeline та перевірку результатів.

Документовано процес налаштування та використання pipeline CI/CD з використанням Terraform та AWS. Це допомагає забезпечити легку розгортання та управління pipeline, а також полегшує спільну роботу команди розробників.

Результатом роботи є повноцінний CI/CD pipeline, який автоматизує процес розгортання та керування інфраструктурою за допомогою Terraform та AWS. Цей pipeline дозволяє швидко та надійно поставляти програмне забезпечення у середовище виробництва, полегшує процес розробки та підвищує якість програмного продукту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Костромицкий А.І., Жижченко М.А., Жижченко А.В. Аналіз варіантів організації хмарної інфраструктури // Актуальні питання розвитку сучасної науки. Тези 3-ї Міжнародної науково-практичної конференції. Видавничий будинок "АКЦЕНТ". Софія, Болгарія. 2019. С. 236-240. URL: http://sci-conf.com.ua.

2. Хмарні технології [Електронний документ] - Режим доступу до ресурсу: http://www.multitest.ua/blog/oblachnye-texnologii-chto-eto-takoe/ - 03.10.2022.

3. Хмарні системи та технології [Електронний документ] - Режим доступу до ресурсу: https://apprenda.com/library/cloud/cloud-systems-and-technologies/ - 03.10.2022.

4. IaaS [Електронний документ] - Режим доступу до ресурсу: https://habr.com/ru/company/it-grad/blog/257295/ - 03.10.2022.

5. Iaas PaaS SaaS [Електронний документ] - Режим доступу до ресурсу: https://3data.ru/services/cloud/iaas-paas-saas-daas - 03.10.2022.

6. Історія хмарних обчислень [Електронний документ] - Режим доступу до ресурсу: https://www.dataversity.net/brief-history-cloud-computing/ - 04.10.2022.

7. Огляд AWS [Електронний документ] - Режим доступу до ресурсу: https://docs.aws.amazon.com/en_us/whitepapers/latest/aws-overview/introduction.html - 04.10.2022.

8. Огляд GCP [Електронний документ] - Режим доступу до ресурсу: https://cloud.google.com/docs/overview/cloud-platform-services- 04.10.2022.

9. Продукти GCP [Електронний документ] - Режим доступу до ресурсу: https://cloud.google.com/products/ - 04.10.2022.

10. Огляд Azure-сервісів [Електронний документ] - Режим доступу до ресурсу: https://victorops.com/blog/microsoft-azure-services-overview - 04.10.2022.

11. Життєвий цикл ПЗ [Електронний документ] - Режим доступу до ресурсу: https://qaevolution.ru/zhiznennyj-cikl-programmnogo-obespecheniya/- 04.10.2022.

13. Етапи розробки ПЗ [Електронний документ] - Режим доступу до ресурсу: https://qalight.com.ua/baza-znaniy/stadii-tsikla-razrabotki-po/- 04.10.2022.

14. Модель "Waterfall" [Електронний документ] - Режим доступу до ресурсу: https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm - 01.11.2022.

15. Модель спіралі життєвого циклу програмного забезпечення [Електронний документ] - Режим доступу до ресурсу: https://pidruchniki.com/1701120547727/informatika/modeli_zhittyevogo_tsiklu - 01.11.2022.

16. Модель ітераційна [Електронний документ] - Режим доступу до ресурсу: https://airbrake.io/blog/sdlc/iterative-model - 01.11.2022.

17. Модель RAD [Електронний документ] - Режим доступу до ресурсу: http://tryqa.com/what-is-rad-model-advantages-disadvantages-and-when-to-use-it/ - 01.11.2022.

18. Модель Agile [Електронний документ] - Режим доступу до ресурсу: https://www.tutorialspoint.com/sdlc/sdlc_agile_model.htm - 01.11.2022.

19. Модель Scrum [Електронний документ] - Режим доступу до ресурсу: https://www.mountaingoatsoftware.com/agile/scrum - 01.11.2022.

20. Модель Kanban [Електронний документ] - Режим доступу до ресурсу: https://www.inflectra.com/methodologies/kanban.aspx - 01.11.2022.

21. Модель DevOps vs Agile [Електронний документ] - Режим доступу до ресурсу: https://www.guru99.com/agile-vs-devops.html - 01.11.2022.

22. Модель DevOps [Електронний документ] - Режим доступу до ресурсу: https://www.atlassian.com/agile/devops - 01.11.2022.

23. IaC [Електронний документ] - Режим доступу до ресурсу: https://techbeacon.com/enterprise-it/infrastructure-code-engine-heart-devops - 01.11.2022.

24. DevOps IaC [Електронний документ] - Режим доступу до ресурсу: https://medium.com/faun/infrastructure-as-code-a-devops-way-to-manage-it-infrastructure-e4f63bfc98fb - 01.01.2023.

25. Автоматизація DevOps [Електронний документ] - Режим доступу до ресурсу: https://dzone.com/articles/devops-automation-and-iac - 01.01.2023.

26. Скриптові мови [Електронний документ] - Режим доступу до ресурсу: https://www.geeksforgeeks.org/introduction-to-scripting-languages/ - 01.01.2023.

27. Автоматизація інфраструктури [Електронний документ] - Режим доступу до ресурсу: https://medium.com/@kari.marttila/comparing-gcp-deployment-manager-and-terraform-3bc6e1b3aa2d - 01.01.2023.

28. Terraform [Електронний документ] - Режим доступу до ресурсу: https://www.terraform.io/ - 01.01.2023.

29. CloudFormation [Електронний документ] - Режим доступу до ресурсу: https://aws.amazon.com/ru/cloudformation/ - 01.01.2023.

30. Керування ресурсами Azure [Електронний документ] - Режим доступу до ресурсу: https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-group-overview - 02.01.2023.

31. Менеджер розгортання [Електронний документ] - Режим доступу до ресурсу: https://cloud.google.com/deployment-manager/docs/ - 02.01.2023.

32. Контейнеризація [Електронний документ] - Режим доступу до ресурсу: https://www.docker.com/resources/what-container - 02.01.2023.

33. Docker [Електронний документ] - Режим доступу до ресурсу: https://www.docker.com/ - 02.01.2023.

34. Kubernetes [Електронний документ] - Режим доступу до ресурсу: https://kubernetes.io/ - 02.01.2023.

35. Віртуалізація сервера [Електронний документ] - Режим доступу до ресурсу: https://www.redhat.com/en/topics/virtualization/what-is-server-virtualization - 03.01.2023.

36. Віртуалізація Hyper-V [Електронний документ] - Режим доступу до ресурсу: https://www.altaro.com/hyper-v/what-is-hyper-v/ - 03.01.2023.

37. Віртуалізація VMware [Електронний документ] - Режим доступу до ресурсу: https://www.vmware.com/topics/glossary/content/virtualization - 03.01.2023.

38. Хмарний моніторинг [Електронний документ] - Режим доступу до ресурсу: https://www.site24x7.com/cloud-monitoring.html - 04.01.2023.

39. Моніторинг Azure [Електронний документ] - Режим доступу до ресурсу: https://azure.microsoft.com/en-us/services/monitoring-and-management/ - 04.01.2023

40. Моніторинг GCP [Електронний документ] - Режим доступу до ресурсу: https://cloud.google.com/products/operations - 04.01.2023.