**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**
**СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ**
Кафедра електроніки і комп'ютерної техніки

# Пояснювальна записка

ДО КВАЛІФІКАЦІЙНОЇ РОБОТИ БАКАЛАВРА
НА ТЕМУ:

**Створення та налаштування інфраструктури в хмарному середовищі з використанням інструментів DevOps**

Завідувач кафедрою електроніки
і комп'ютерної техніки _____А.С. Опанасюк

Керівник роботи _____О.В. Д’яченко

Виконав студент гр. ТК-91 _____Д.О. Босенко

Суми 2023

# Реферат

В кваліфікаційній роботі бакалавра була розроблена масштабована, відмовостійка та гнучка інфраструктура в хмарному середовищі. Цей проект виконаний відповідно до методології DevOps та включає набір практик, характерних для цієї методології. Для реалізації інфраструктури використовувалися такі інструменти, як GitLab, Terraform та хмарна платформа AWS. Завдяки цим інструментам та підходам до розробки та управління, була досягнута швидкість, якість та масштабованість процесів розробки та розгортання інфраструктури.

В рамках роботи було проведено детальний опис інструментів для створення інфраструктури, зокрема Terraform файлів та CI/CD конвеєру. Також була розглянута архітектура проекту та взаємодія між усіма його елементами. Це дозволило розробити ефективну та добре організовану інфраструктуру.

В результаті проведених досліджень та роботи над проектом було досягнуто високої рівня якості та ефективності розробки та розгортання інфраструктури. Отримані результати свідчать про успішну реалізацію методології DevOps та впровадження передових практик управління IT-проектами. Дана робота є важливим внеском у сферу розробки інфраструктури та підтверджує значимість методології DevOps для досягнення високої продуктивності та якості роботи технологічних організацій.

Для написання випускної роботи було використано 15 літературних джерел, що дозволило підтвердити та підкріпити здійснені рішення та практики.

Кваліфікаційна робота бакалавра містить 41 сторінку та 28 рисунків.

Ключові слова: DevOps, CI/CD, IaC, Terraform, AWS, GitLab.

**Апаппа**
**впвавап**
**вапвапв**
**апвапва**
**пвапвап**
**ввапвап**
**вапвапв**
**апваппв**
**апп**

| | | | | | ЕлІТ 6.172.00.02.049 ПЗ | | | |
|---|---|---|---|---|---|---|---|---|
| *Змін.* | *Лист* | *№ докум.* | *Підпис* | *Дата* | | | | |
| *Виконав* | | Босенко Д.О. | | | Створення та налаштування інфраструктури в хмарному середовищі з використанням інструментів DevOps | *Літ.* | *Лист* | *Листів* |
| *Перевірив* | | Д`яченко О.В. | | | | | 3 | 41 |
| *Реценз.* | | | | | | СумДУ, гр. ТК-91 | | |
| *Н. контр.* | | Д'яченко О.В. | | | | | | |
| *Підтвердив* | | Опанасюк А.С. | | | | | | |

## Введення

У сучасному світі розробка програмного забезпечення вимагає швидкості, ефективності та надійності. Застосування хмарних середовищ та DevOps може значно спростити та прискорити процес створення та налаштування інфраструктури. В цьому контексті використання хмарних середовищ та DevOps стає все більш невід'ємною частиною розробки програмного забезпечення. Хмарні середовища надають гнучкість, масштабованість та доступність для створення та розгортання інфраструктури без необхідності власного фізичного обладнання. З іншого боку, DevOps вводить концепцію автоматизації, співпраці та зниження часу циклу розробки. Це дозволяє розробникам швидше та ефективніше створювати, тестувати та впроваджувати програмне забезпечення.

Враховуючи цей контекст, важливість створення та налаштування інфраструктури в хмарному середовищі з використанням DevOps стає очевидною. Це дозволить розробникам зосередитись на створенні програмного забезпечення, мінімізуючи зусилля, час та витрати, пов'язані зі створенням та управлінням інфраструктурою. Автоматизація процесу створення та налаштування інфраструктури через інструменти, такі як Terraform та CI/CD конвеєри, сприятиме прискоренню розгортання, забезпеченню стабільності та зменшенню ризиків помилок у процесі розробки та розгортання програмного забезпечення.

В рамках даної кваліфікаційної роботи розглянуто реалізацію такої інфраструктури в хмарному середовищі з використанням DevOps підходу, зосереджуючись на використанні GitHub, Terraform та AWS. Це дозволить нам дослідити переваги цих технологій та показати, як їх поєднання може привести до ефективного та автоматизованого процесу створення та управління інфраструктурою в хмарному середовищі

# 1. ОГЛЯД ЛІТЕРАТУРИ ТА ПОСТАНОВКА ЗАВДАННЯ ПРОЕКТУВАННЯ

## 1.1 Методологія DevOps

DevOps – це підхід до розробки програмного забезпечення, який поєднує розробку (Development) та експлуатацію (Operations) в єдиний цілісний процес. Він спрямований на поліпшення співпраці, комунікації та взаєморозуміння між розробниками програмного забезпечення та операторами, з метою забезпечення швидкого впровадження змін, зменшення часу на відновлення після виникнення проблем та забезпечення стабільності та якості виробленого програмного продукту.

Підхід сприяє зміні управління IT-проектами, дозволяє уникнути проблем, що виникають через відокремлення розробки та експлуатації, і сприяє ефективному впровадженню програмного забезпечення та забезпеченню високої якості та надійності продукту.

DevOps включає різні практики, інструменти та принципи, які спільно працюють для досягнення швидкої та надійної доставки програмного забезпечення. Основні складові DevOps включають:

1. Спільна комунікація та співпраця: DevOps підтримує злиття розробки та експлуатації шляхом покращення комунікації та співпраці між розробниками, тестувальниками, операторами та іншими зацікавленими сторонами. Це включає спільні збори, обмін знаннями та надання зворотного зв’язку один одному.

2. Автоматизація: DevOps сприяє автоматизації процесів розробки, тестування, розгортання та моніторингу програмного забезпечення. Це дозволяє знизити ризик помилок, покращити ефективність та забезпечити швидку поставку змін.

3. Інструменти для керування конфігурацією: Використання інструментів керування конфігурацією, таких як наприклад Git, допомагає відстежувати зміни в коді, контролювати версії та забезпечувати спільну роботу на проекті.

4. Continuous Integration (CI) та Continuous Deployment (CD): Ці практики включають автоматичну збірку, тестування та розгортання програмного забезпечення в автоматизованому режимі. CI дозволяє швидко інтегрувати зміни розробників, а CD допомагає автоматично розгортати зміни в продуктивне середовище.

5. Моніторинг та логування: Використання інструментів моніторингу та логування допомагає відстежувати стан і продуктивність системи, виявляти проблеми та швидко реагувати на них.

6. Інфраструктура як код (Infrastructure as Code): DevOps сприяє використанню інструментів, таких як Terraform або Ansible, для автоматизованого створення та керування інфраструктурою, що дозволяє забезпечити консистентність, масштабованість та відтворюваність середовищ.

7. Культура безпеки: DevOps включає посилення уваги до аспектів безпеки в процесі розробки, тестування та експлуатації програмного забезпечення. Це включає автоматизовану перевірку вразливостей, безпечне зберігання конфіденційної інформації та забезпечення контролю доступу.

## 1.2 Хмарні середовища

Хмарні середовища є популярним варіантом для забезпечення інфраструктури та послуг у сучасному IT-середовищі. Основні концепції, які варто дослідити, включають типи хмар, моделі обчислення та хмарні провайдери. На ринку існує багато хмарних провайдерів, які надають різні послуги та функціонал. Найпопулярніші хмарні провайдери включають AWS, Azure, GCP і Alibaba Cloud. Кожен провайдер має свої унікальні особливості, пакети послуг та цінові моделі. В кваліфікаційній роботі була використана саме хмара AWS, від компанії Amazon.

AWS надає широкий набір сервісів, що охоплюють обчислення, зберігання даних, мережі, бази даних, аналітику, штучний інтелект та багато іншого. Це дає можливість використовувати різноманітні інструменти та послуги для вирішення різних бізнес-потреб.

Сервіси, що використовуються в кваліфікаційній роботі:

- Amazon Elastic Compute Cloud (EC2) – є одним з основних сервісів, який надається AWS. Він дозволяє користувачам створювати та управляти віртуальними серверами у хмарному середовищі.
- Amazon Virtual Private Cloud (VPC) – це сервіс, що надає можливість створювати та налаштовувати власні віртуальні приватні мережі в AWS.
- Amazon Relational Database Service (RDS) – це повністю керований сервіс баз даних, який надається AWS. RDS дозволяє легко налаштовувати, керувати та масштабувати реляційні бази даних в хмарному середовищі.
- ELB (Elastic Load Balancing) та ASG (Auto Scaling Group) – це два сервіси, які надаються AWS та часто використовуються разом для підтримки високої доступності та масштабованості веб-додатків або сервісів. ASG дозволяє автоматично масштабувати групу EC2-інстанцій вертикально або горизонтально залежно від змінного навантаження. ELB дозволяє розподілити трафік між кількома веб-серверами або ресурсами за допомогою різних алгоритмів балансування навантаження.
- IAM (Identity and Access Management) – це сервіс управління доступом, який надається AWS. Він дозволяє контролювати й управляти доступом користувачів, ролей та груп до ресурсів AWS.
- CloudWatch – це сервіс моніторингу та керування ресурсами AWS. Він надає можливість відстежувати метрики ресурсів, таких як використання CPU, мережевий трафік, навантаження на диск тощо. Ви можете відслідковувати та аналізувати ресурси, щоб забезпечити їх ефективне використання та виявляти проблеми.

- DynamoDB – це повністю керований сервіс NoSQL бази даних, який надається AWS. Він призначений для зберігання та обробки великого обсягу даних з високою швидкодією та масштабованістю.

### 1.3 Автоматизація інфраструктури

У методології DevOps, ключовим елементом, який допомагає забезпечити швидку та надійну доставку програмного забезпечення та автоматизувати підняття інфраструктури є IaC (Infrastructure as Code).

IaC - це практика автоматизації та керування інфраструктурою за допомогою коду. Замість мануального налаштування та управління інфраструктурою, використовується код, який описує потрібні ресурси, конфігурацію та залежності між ними. Основні переваги автоматизації інфраструктури включають:

1. Прогнозованість і повторюваність: Автоматизація дозволяє нам описати інфраструктуру у вигляді коду, що забезпечує прогнозованість результатів і можливість повторювати процеси розгортання і конфігурації безперервно.

2. Швидкість розгортання: За допомогою автоматизації можливо швидко створювати та налаштовувати ресурси в інфраструктурі. Це дозволяє скоротити час розгортання нових сервісів і додатків, що поліпшує ефективність команди розробки та здатність без зволікання реагування на зміни на ринку.

3. Легка впроваджуваність змін: Завдяки автоматизації можливо швидко впроваджувати зміни в інфраструктурі, оновлювати сервіси, встановлювати виправлення та розгортати нові функції. Це допомагає забезпечити високу гнучкість нашої інфраструктури.

4. Зниження ризиків та помилок: Автоматизація дозволяє уникати помилок, пов'язаних з ручними процесами конфігурації та розгортання. Ви можете використовувати контроль версій для коду інфраструктури, робити прогнозовані зміни та відкатувати їх у разі необхідності.

5. Збільшена ефективність ресурсів: Автоматизація дозволяє оптимізувати використання ресурсів, таких як сервери, мережі та сховища даних. Ви можете автоматично масштабувати ресурси відповідно до потреб, використовувати їх більш ефективно та уникати непотрібних витрат.

Один з популярних інструментів для автоматизації інфраструктури, який був розглянутий у роботі це є Terraform.

Terraform — це інструмент для інфраструктурного програмування, який дозволяє описувати інфраструктуру проекту у вигляді декларативних конфігураційних файлів, які пишуться на спеціальній мові, що називається HashiCorp Configuration Language (HCL). Основна перевага Terraform включають: декларативний код, мультипровайдерна підтримка, версіонування та керування станом.

Принцип роботи Terraform полягає у встановленні і керуванні ресурсами інфраструктури за допомогою провайдерів. Провайдери - це компоненти Terraform, які взаємодіють з конкретними хмарними платформами або іншими провайдерами інфраструктури. Провайдери забезпечують Terraform доступ до API та можливість створювати, змінювати та видаляти ресурси.

Основний процес роботи з Terraform включає наступні кроки:

- Ініціалізація: Виконуючи команду **terraform init**, щоб ініціалізувати робочий каталог проекту. Під час ініціалізації Terraform завантажує необхідні провайдери, модулі та налаштування.
- Конфігурація: Створюючи конфігураційні файли, в яких описана бажана інфраструктура. Використовуючи HCL, визначаються ресурси, їх параметри та залежності між ними.

- Планування: Виконуючи команду **terraform plan**, Terraform аналізує конфігураційні файли і створює план розгортання. План показує, які зміни будуть внесені в інфраструктуру для досягнення бажаного стану.
- Виконання: Запуск команди **terraform apply** застосовує зміни, описані в плані. Terraform автоматично взаємодіє з провайдерами, створює необхідні ресурси, налаштовує параметри та встановлює залежності.
- Управління станом: Terraform зберігає поточний стан інфраструктури в спеціальному файлі, який називається файлом стану (state file). Цей файл використовується для відстеження ресурсів та контролю змін. Це дає можливість управляти станом, роблячи зміни в конфігураційних файлах та застосовуючи їх за допомогою команд **terraform plan** та **terraform apply**.

Таким чином, Terraform дозволяє вам описати, розгорнути та керувати інфраструктурою вашого проекту з використанням коду, забезпечуючи прогнозованість, швидкість і повторюваність процесу автоматизації.

Головна перевага Terraform полягає в його крос-платформеності і підтримці різних провайдерів. Таким чином можна використовувати Terraform для керування інфраструктурою в різних хмарних платформах та технологіях, що забезпечує єдиний інтерфейс для автоматизації різнорідної інфраструктури.

## 1.4 Continuous Integration та Continuous Deployment (CI/CD) з використанням GitLab

Continuous Integration/Continuous Deployment (CI/CD) є ключовим елементом DevOps-практик, спрямованих на автоматизацію розробки та постачання програмного забезпечення, забезпечуючи швидкий, стабільний та надійний процес розгортання.

CI/CD - це практики розробки програмного забезпечення, що включає автоматизовану збірку, тестування та розгортання програмного коду.

Цей процес дозволяє розробникам швидко та безпечно впроваджувати зміни в продукт, забезпечуючи високу якість інтеграції та доставки програмного забезпечення.

Continuous Integration (CI) - це практика, при якій розробники регулярно відправляють свій код в спільний репозиторій (наприклад, GitLab), після чого відбувається автоматична збірка та тестування цього коду. Це дозволяє виявляти можливі проблеми і конфлікти між різними частинами програмного коду та швидко їх виправляти.

Continuous Deployment (CD) - це практика автоматичного розгортання розробленого програмного коду в реальне середовище після проходження всіх необхідних тестів. Це означає, що нові зміни автоматично розгортатимуться на сервери або хмарні середовища без необхідності ручного втручання. Це дозволяє швидко впроваджувати нові функції та виправлення помилок у продукті.

GitLab є системою керування версіями та веб-платформою для розробки програмного забезпечення. Вона надає повний набір інструментів для управління кодом, спільної роботи, автоматизації процесу розробки та CI/CD.

GitLab CI (Continuous Integration) - це частина GitLab, яка дозволяє автоматизувати процеси збирання, тестування та розгортання програмного коду. За допомогою GitLab CI, розробники можуть організувати неперервний цикл розробки, де кожне змінене розробником повідомлення в Git-репозиторії спричиняє автоматичну збірку, запуск тестів та, в разі успіху, розгортання програмного коду до виробничого середовища. Він використовує файл конфігурації ***.gitlab-ci.yml***, в якому визначаються етапи, задачі та умови для CI/CD конвеєру. Цей файл містить інструкції для збирання, тестування та розгортання коду, а також визначає налаштування середовища, ролі розробників та інші параметри.

Цей інструмент автоматизує рутинні завдання, забезпечуючи швидке та надійне впровадження змін у програмному коді. Він підтримує паралельну обробку завдань, можливість налаштування різних середовищ (test, stage, dev, prod) та інтеграцію з іншими інструментами розробки. Також він сприяє автоматизації процесу розробки, поліпшенню якості програмного забезпечення та прискоренню впровадження нових функцій, роблячи його важливим інструментом методології DevOps.

# 2. АРХІТЕКТУРА ПРОЕКТУ

## 2.1 Постановка задачі та мета

На початку будь-якого проекту в методології DevOps важливо чітко визначити цілі, які потрібно досягти. Цілі мають включати поліпшення продуктивності, зменшення часу на розгортання змін, покращення якості продукту або автоматизацію рутинних завдань. Ціль має бути конкретною, вимірюваною та досяжною.

Загалом, в цій роботі, головною метою було створення гнучкої, надійної та ефективної інфраструктури, яка підтримує швидкий розвиток та впровадження програмного продукту, забезпечуючи високу якість та задоволення потреб користувачів. Використовуючи такі інструменти, як GitLab, Terraform і AWS, було створено автоматизовану інфраструктуру, яка дозволяє ефективно керувати розробкою, тестуванням та розгортанням програмного продукту за допомогою коду.

## 2.2 Структура та складові проекту

Структура файлів проекту наведена на рисунку 2.2.1. В структурі проекту наявні наступні файлі:

***.terraform*** - каталог, що містить дані, пов'язані з роботою Terraform.

***modules*** - каталог, який містить модулі Terraform для організації коду та забезпечення модульності інфраструктури.

**.gitignore** - файл, що містить список ігнорованих Git-ом файлів та каталогів.

**.gitlab-ci.yml** - конфігураційний файл GitLab CI/CD, в якому описані кроки налаштування та створення інфраструктури за допомогою Terraform.

***main.tf*** - цей файл містить головний опис інфраструктури проекту, включаючи модулі, backend, та їх налаштування.

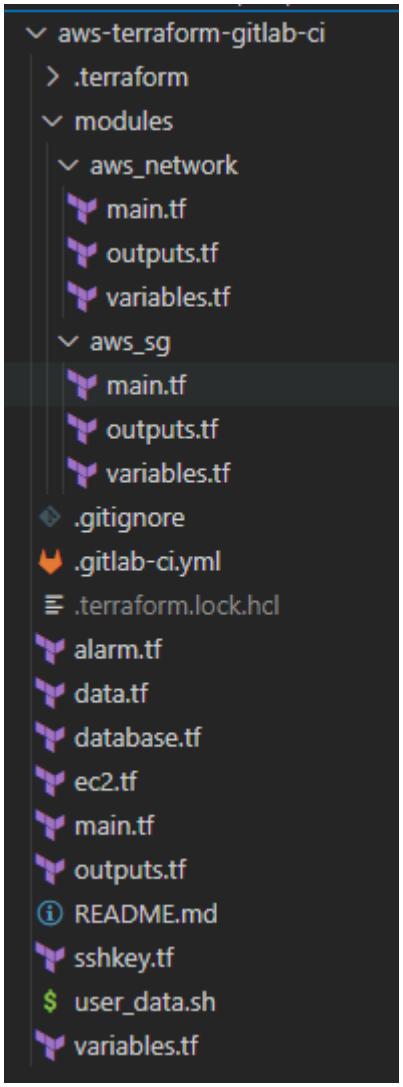



Рисунок 2.2.1 – «Структура файлів»

Інші файли, такі як ***.terraform.lock.hcl***, ***alarm.tf***, ***data.tf***, ***database.tf, ec2.tf***, ***main.tf***, ***outputs.tf***, ***sshkey.tf***, ***user_data.sh***, ***variables.tf***, місять описи різних аспектів проекту, такі як блокування версій, налаштування сповіщень, отримання даних, налаштування баз даних, опис EC2 екземплярів, виведення інформації про ресурси, створення SSH-ключа, сценарій конфігурації та змінні.

### 2.3 Архітектура та взаємозв'язки компонентів інфраструктури

При внесенні змін у інфраструктуру, усі зміни будуть записані у систему контролю версій – Git. Після, вони відправляються у віддалений репозиторій в GitLab та зберігаються в ньому.

GitLab надає відкрите програмне забезпечення, яке використовується для виконання CI/CD конвеєру, під назвою GitLab Runner. Ця програма встановлюється на окремому сервері або в контейнері і підключається до GitLab та виконує автоматизовані завдання описані у файлі ***.gitlab-ci.yml***.

Після двостороннього підключення, на GitLab Runner виконується клонування репозиторію з GitLab, збирається програмний код та розгортається відповідно до налаштувань. GitLab Runner також забезпечує зворотній зв'язок до GitLab, повідомляючи про стан виконання задач та результати тестування. На рисунку 2.3.1 наведена схема цього процесу.

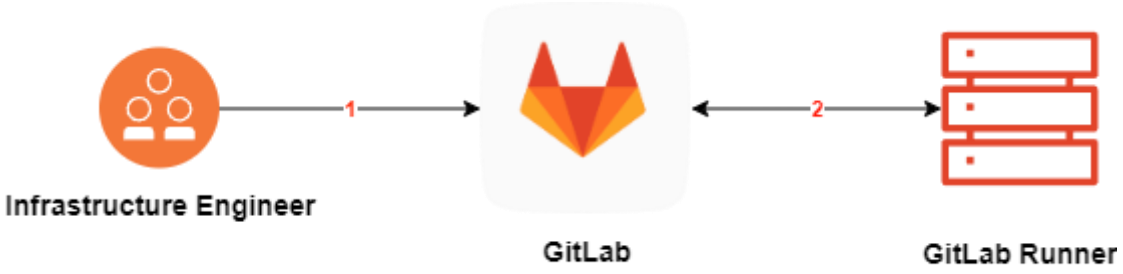



Рисунок 2.3.1 – «Схема взаємодії між інженером інфраструктури та сервісом GitLab»

Отримавши репозиторій та файл .gitlab-ci.yml, GitLab Runner починає виконувати послідовно наступні дії описані в файлі:

1. Встановлюються змінні середовища, які прописані у налаштуваннях GitLab, потрібні для взаємодії з сервісами AWS.
2. Виконується команда ***terraform init.***

3. Перевіряються ***.tf*** файли, на наявність помилок у коді, за допомогою допомогою команди ***terraform validate***.
4. Планування змін ***terraform plan***.
5. Виконання змін ***terraform apply***.
6. Опціонально виконується знищення інфраструктури ***terraform destroy***.

Весь цей процес дозволяє автоматизувати розгортання, управління та знищення інфраструктури в хмарному середовищі за допомогою Terraform у GitLab CI.

Terraform, за допомогою конфігурації backend, визначає, де буде зберігатись та контролюватись файл стану інфраструктури. Конфігурація backend встановлює параметри для вибору інфраструктурного сервісу, який використовується для зберігання та блокування файлу стану.

На рисунку 2.3.2 наведена схема зберігання та блокування файлу стану (state file).

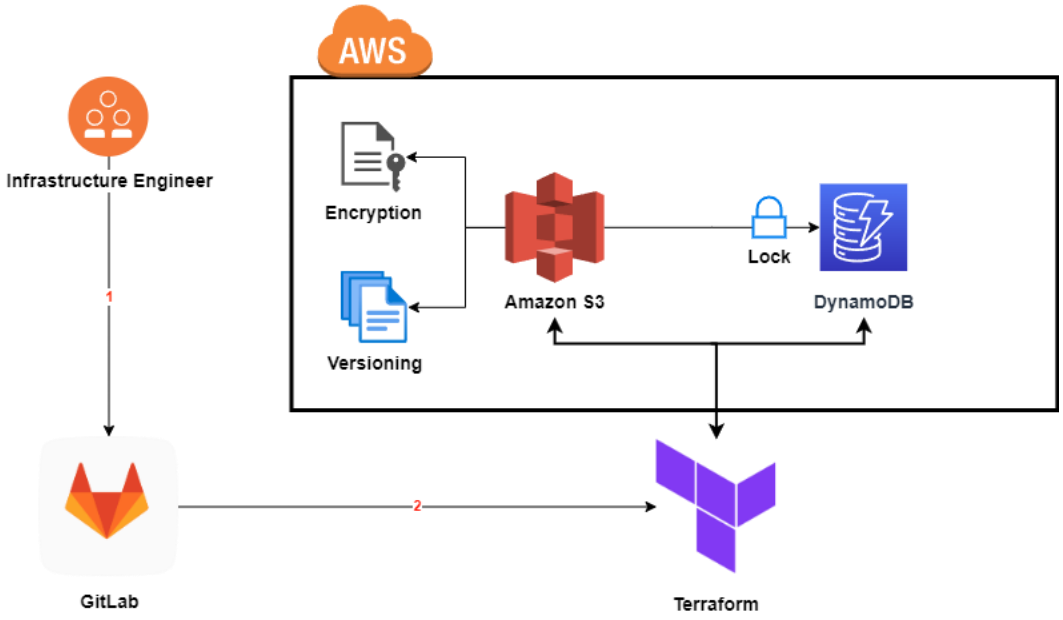



Рисунок 2.3.2 – «Схема зберігання та блокування файлу стану»

У даному випадку, Terraform використовує Amazon S3 як сервіс зберігання файлів стану. Крім того, використовується таблиця DynamoDB, для блокування та керування доступом до файлу стану. Це дозволяє уникнути ситуацій конфлікту доступу до стану при паралельних операціях з розгортанням.

Загальний результат використання конфігурації backend полягає в тому, що Terraform безпечно та надійно зберігає файл стану інфраструктури у хмарному сервісі, такому як Amazon S3, і використовує DynamoDB для управління блокуванням стану. Це сприяє спільному використанню та управлінню інфраструктурою з командою розробників та забезпечує надійність розгортання.

Після налаштування backend, Terraform починає створювати інфраструктуру в хмарному середовищі AWS, яка описана в конфігураційних файлах Terraform.

На рисунку 2.3.3 показана схема майбутньої інфраструктури, яка буде створена у хмарному середовищі. Основні етапи створення інфраструктури послідовні та включають наступне:

1. Віртуальна хмара (Virtual Private Cloud): Terraform створює віртуальну приватну хмару, що включає мережу, підмережі та необхідні мережеві компоненти для ізоляції та організації ресурсів.
2. Групи безпеки (Security Groups): Terraform створює безпекові групи для контролю доступу до EC2 та інших ресурсів. Встановлюються правила фільтрації трафіку, які дозволяють або блокують з'єднання на основі визначених правил.
3. База даних (RDS MySQL): Terraform створює ресурси для бази даних, включаючи встановлення параметрів, таких як розмір, тип бази даних, ідентифікатори, імена користувачів та паролі. Це дозволяє налаштовувати та управляти базою даних, необхідною для зберігання та обробки даних додатку.

4. Навантажувальний балансувальник (Elastic Load Balancer): Створює навантажувальний балансувальник, який розподіляє трафік між різними екземплярами EC2 додатку, забезпечуючи високе розподілення навантаження.

5. Группа автоматичного масштабування (Auto Scaling Group): Terraform створює групу автоматичного масштабування, включаючи налаштування запуску екземплярів EC2, їх тип та сценарій конфігурації, політик масштабування. Це дозволяє динамічно змінювати кількість ресурсів відповідно до потреб додатку.

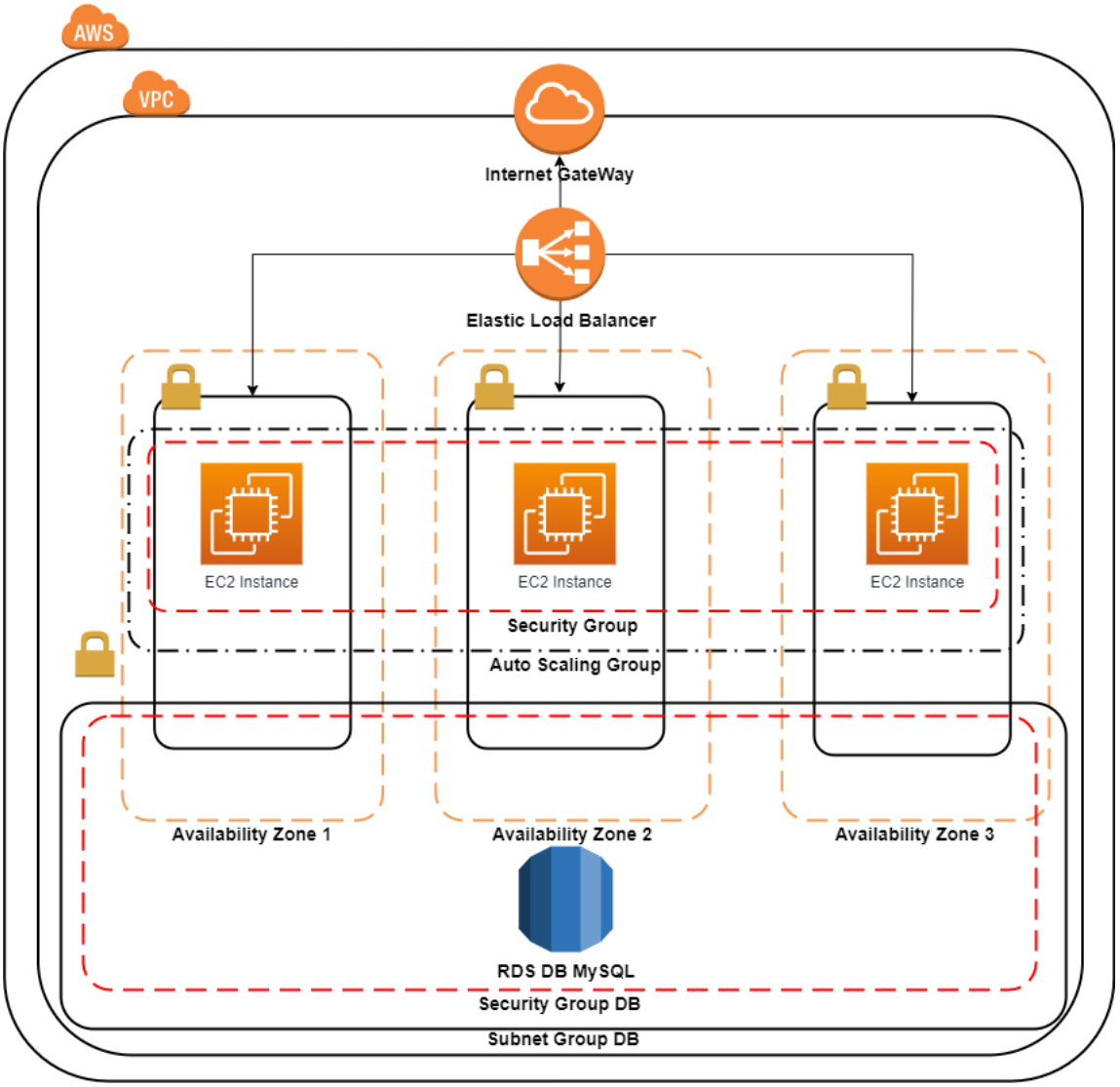



Рисунок 2.3.3 – «Схема інфраструктури в хмарному середовищі»

Для прикладу роботи інфраструктури у файлі конфігурації екземплярів EC2 (user_data.sh), було написано скрип, який встановлює docker images, з невеличким сайтом написаним на мові Java, який використовує, для своєї роботи базу даних.

Після завершення роботи Terraform, отримуємо екземпляри EC2, з працюючим веб-додатком, які з'єднані з базою данних та складаються зі створених образів, ресурсів, конфігуровані за допомогою скрипту, мережевих налаштувань та інших налаштувань необхідних для запуску веб-додатку в хмарному середовищі AWS.

Загальну схему CI/CD конвеєру можна побачити на рисунку 2.3.4.

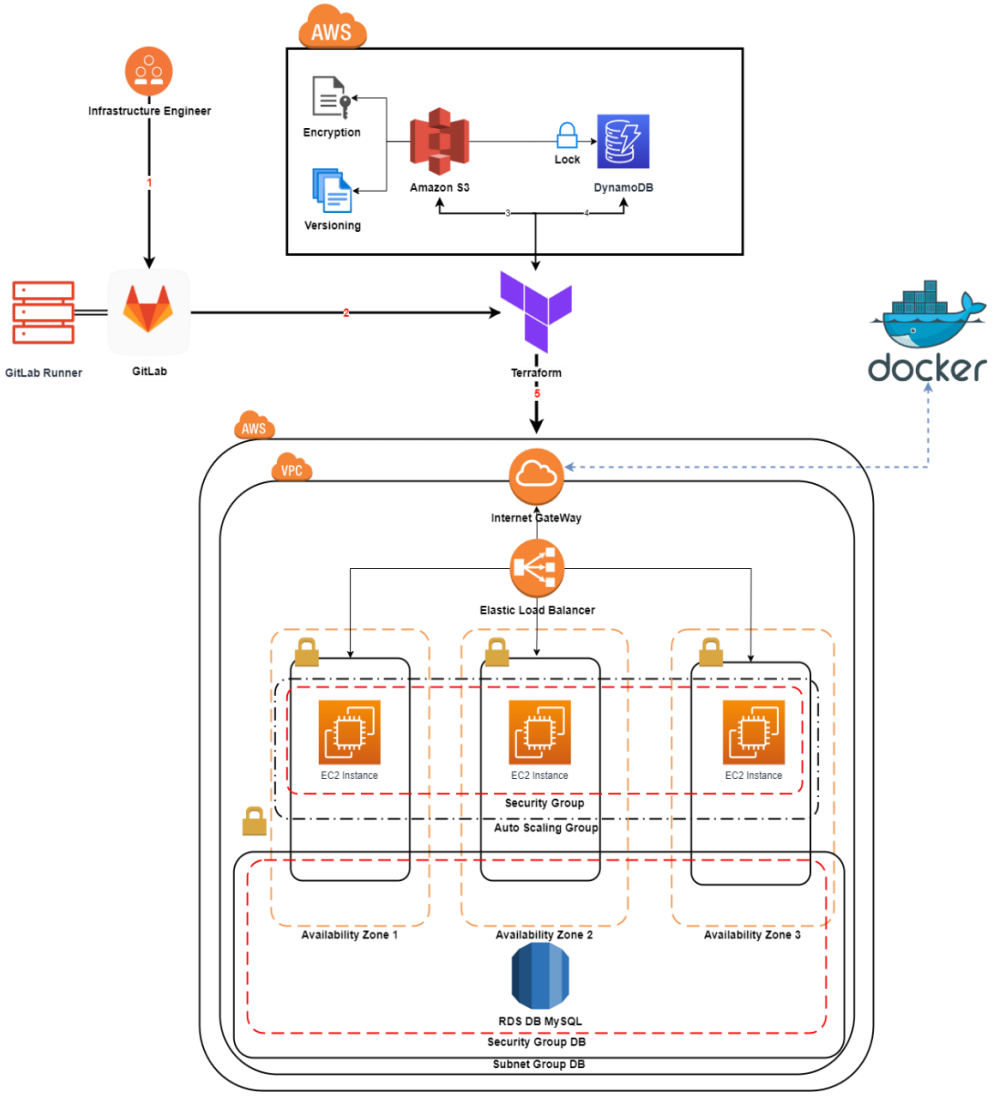



Рисунок 2.3.4 – «Схема CI/CD конвеєру»

# 3. РЕАЛІЗАЦІЯ ПРОЕКТУ

## 3.1 Створення Terraform коду

Починаємо написання коду зі створення головного файлу для створення інфраструктури проекту – ***main.tf***. Він наведений на рисунку 3.1.1.

main.tf > ...

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

#=================BACKEND=======================

terraform {
  backend "s3" {
    bucket         = "aws-terraform-gitlab-ci"
    region         = "eu-central-1"
    dynamodb_table = "dynamodb-terraform-state-lock"
  }
}

#================Module====================

module "vpc_web" {
  source              = "./modules/aws_network"
  // source           = "git@github.com..."
  env                 = var.current_environment
  vers                = var.current_version
  project             = var.project
  vpc_cidr            = "120.10.0.0/16"
  public_subnet_cidrs = ["120.10.1.0/24", "120.10.2.0/24", "120.10.3.0/24"]
}

module "sg_web" {
  source  = "./modules/aws_sg"
  name    = "web"
  env     = var.current_environment
  vers    = var.current_version
  project = var.project
  vpc_id  = module.vpc_web.vpc_id
  ports   = var.allow_ports_web
}

module "sg_db" {
  source  = "./modules/aws_sg"
  name    = "db"
  env     = var.current_environment
  vers    = var.current_version
  project = var.project
  vpc_id  = module.vpc_web.vpc_id
  ports   = var.allow_ports_db
}
```

Рисунок 3.1.1 – «Код файлу ***main.tf»***

У цьому файлі ми визначаємо провайдера AWS, налаштовуємо backend для збереження стану Terraform. Задаємо ім'я S3 (bucket), регіон та назву таблиці DynamoDB (dynamodb_table), яка використовується для блокування стану, а також використовуємо модулі для налаштування віртуальної приватної хмарної мережі (VPC) та груп безпеки (Security Group).

Використання модулів дозволяє нам повторно використовувати налаштування та забезпечує шаблонізацію інфраструктури для різних середовищ і проектів.

Модуль **aws_network**, код якого наведений на рисунку 3.1.2, виконує налаштування мережі та публічних підмереж (subnets), які потрібні для коректного функціювання інфраструктури.

Код цього модуля починається з визначення даних про доступні зони доступності AWS (Availability Zones). Ці дані використовуються для розташування публічних підмереж у різних зонах доступності, щоб забезпечити високу доступність та відмовостійкість.

Далі йде визначення ресурсів. Першим ресурсом є **aws_vpc**, який визначає віртуальну приватну хмарну мережу (VPC) з вказаним діапазоном IP-адрес. Він також включає налаштування тенантності (instance_tenancy) та підтримку DNS-хостів (enable_dns_hostnames).

Наступний ресурс - **aws_internet_gateway**, який створює шлюз до Інтернету для забезпечення зовнішнього доступу до публічних підмереж VPC.

Після цього йде визначення ресурсу **aws_route_table**, який встановлює таблицю маршрутизації для публічних підмереж. В ній встановлюється маршрут для адресного діапазону "0.0.0.0/0", який спрямовує трафік через інтернет-шлюз.

Ресурс **aws_route_table_association** пов'язує кожну публічну підмережу з відповідною таблицею маршрутизації.

```
data "aws_availability_zones" "availability"{}

resource "aws_vpc" "main" {
    cidr_block = var.vpc_cidr

    instance_tenancy        = "default"
    enable_dns_hostnames    = true
    tags                    = {Name = "VPC-${var.project}-${var.env}-V${var.vers}"}
}

resource "aws_internet_gateway" "main" {
  vpc_id    = aws_vpc.main.id
  tags      = {Name = "GateWay-${var.project}-${var.env}-V${var.vers}"}

}

resource "aws_route_table" "public_subnets" {
  vpc_id = aws_vpc.main.id

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.main.id
  }
  tags = {Name = "Route-${var.project}-${var.env}-V${var.vers}"}
}

resource "aws_route_table_association" "public_routes" {
  count          = length(aws_subnet.public_subnets[*].id)
  route_table_id = aws_route_table.public_subnets.id
  subnet_id      = element(aws_subnet.public_subnets[*].id, count.index)
}
#=========================Subnets=======================================

resource "aws_subnet" "public_subnets" {
  count = length(var.public_subnet_cidrs)
  vpc_id = aws_vpc.main.id
  availability_zone = data.aws_availability_zones.availability.names[count.index]
  cidr_block = element(var.public_subnet_cidrs, count.index)
  map_public_ip_on_launch = true
  tags = {Name = "Subnet-public-${var.project}-${var.env}-V${var.vers}-${count.index + 1}"}
}
```

Рисунок 3.1.2 – «Код модулю aws_network»

Останній ресурс - **aws_subnet**, визначає публічні підмережі, які розташовані у різних зонах доступності. Кожна публічна підмережа має свій власний IP-діапазон і пов'язується з таблицею маршрутизації та VPC.

Цей модуль допомагає створити і налаштувати основну мережеву інфраструктуру, яка включає в себе VPC, публічні підмережі та необхідні налаштування маршрутизації для забезпечення зовнішнього доступу до ресурсів в цій мережі.

Наступний модуль, **aws_sg** визначає ресурс **aws_security_group**, який виконує налаштування групи безпеки для контролю доступу до ресурсів у мережі. Код цього модулю наведений на рисунку 3.1.3.

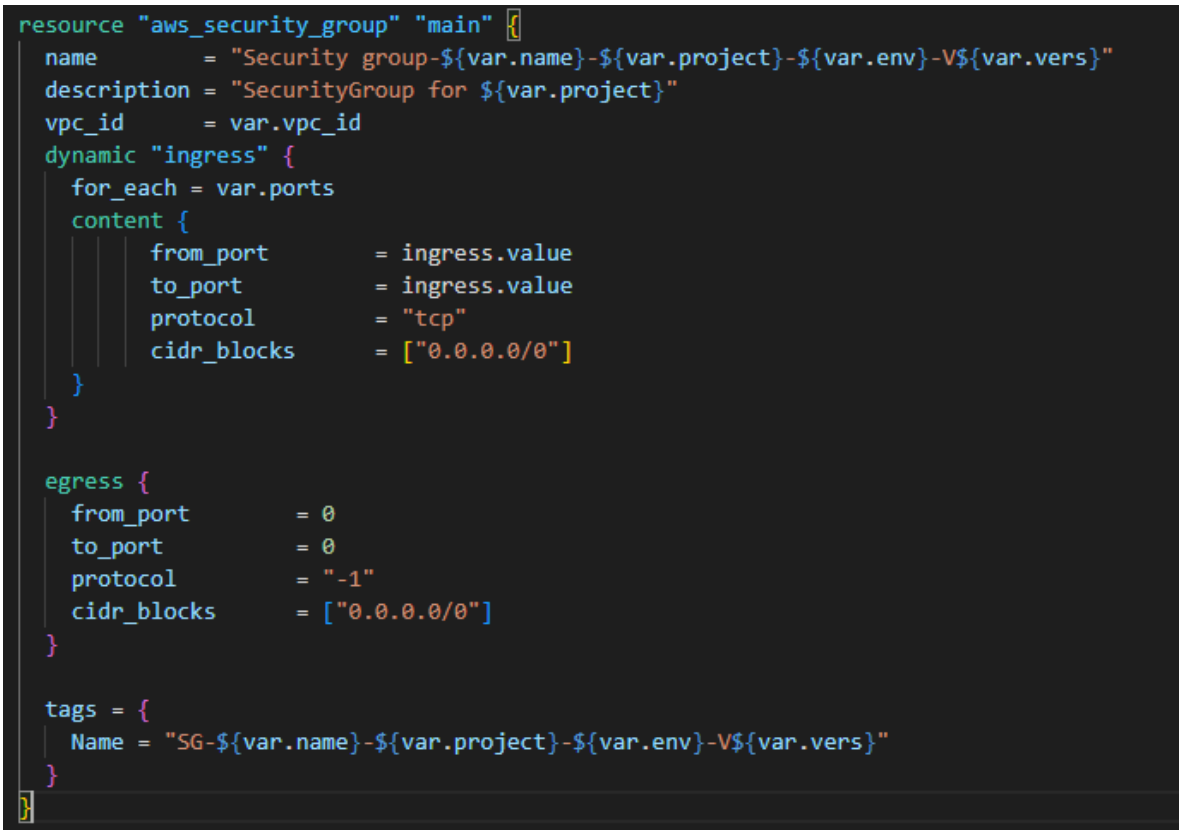



Рисунок 3.1.3 – «Код модулю aws_sg»

Основні атрибути ресурсу включають назву групи безпеки (name), опис (description) та ідентифікатор VPC (vpc_id), до якого вона відноситься. Також використовується директива "dynamic" для визначення вхідних правил доступу. Кожне правило визначається за допомогою блоку "content", де вказується порт (from_port, to_port), протокол (protocol) та дозволені IP-адреси (cidr_blocks). В даному випадку, вхідні правила дозволяють доступ з будь-яких IP-адрес ("0.0.0.0/0") на вказані порти.

Також визначається блок "egress", який встановлює правила для вихідного трафіку з групи безпеки. В даному випадку, всім вихідним з'єднанням дозволяється відправляти будь-який трафік ("0.0.0.0/0").

Назва та теги групи безпеки також визначаються на основі вхідних змінних, що містять інформацію про назву (name), проект (project), середовище (env) та версію (vers).

Модуль допомагає створити і налаштувати групу безпеки, яка визначає правила доступу до ресурсів у мережі, що забезпечує контроль трафіку та безпеку системи.

Після створення VPC та Security Group починаємо описувати як буде виглядати наша база даних RDS у файлі ***database.tf***. Код цього файлу наведений на рисунку 3.1.4.

```
resource "aws_db_instance" "db" {
  allocated_storage       = 10

  db_subnet_group_name    = aws_db_subnet_group.main.name

  identifier              = "db-${var.project}-${var.current_environment}"
  engine                  = "mysql"
  engine_version          = "5.7"
  instance_class          = "db.t3.micro"
  parameter_group_name    = "default.mysql5.7"
  skip_final_snapshot     = true
  #publicly_accessible     = true

  db_name                 = var.namedb
  username                = var.usernamedb
  password                = var.passworddb

  vpc_security_group_ids  = [module.sg_db.sg_id]
  availability_zone       = data.aws_availability_zones.availability.names[1]
  #lifecycle {
  #  prevent_destroy = true
  #}
}

resource "aws_db_subnet_group" "main" {
  name            = "${var.project}-${var.current_environment}-v${var.current_version}"
  subnet_ids      = flatten([module.vpc_web.public_subnet_ids])
  tags            = merge(var.common_tag, {Name = "DB-SubnetGroup-${var.project}-${var.current_environment}-v${var.current_version}"})

}
```

Рисунок 3.1.4 – «Код файлу database.tf »

Цей код визначає два ресурси: **aws_db_instance** та **aws_db_subnet_group**, які виконують налаштування бази даних і групи підмереж бази даних в середовищі AWS.

Ресурс **aws_db_instance** визначає параметри і налаштування для створення екземпляра бази даних. Він включає такі атрибути, як обсяг

виділеного сховища (allocated_storage), група підмереж бази даних (db_subnet_group_name), ідентифікатор (identifier), тип бази даних (engine), версія бази даних (engine_version), клас екземпляра бази даних (instance_class), група параметрів (parameter_group_name), наявність публічного доступу (publicly_accessible) і т.д. Крім того, вказуються назва бази даних (db_name), ім'я користувача (username) та пароль (password), а також ідентифікатори групи безпеки VPC (vpc_security_group_ids) та доступні зони доступності (availability_zone). Цей ресурс допомагає створити та налаштувати екземпляр бази даних MySQL.

Ресурс **aws_db_subnet_group** визначає групу підмереж бази даних, яка використовується для налаштування доступу до бази даних. Він включає назву (name), ідентифікатори підмереж (subnet_ids) та теги (tags). У цьому випадку, ідентифікатори підмереж беруться з модуля **vpc_web** для використання публічних підмереж. Цей ресурс допомагає створити групу підмереж бази даних для використання у відповідних ресурсах бази даних.

Код дозволяє створити та налаштувати екземпляр бази даних та групу підмереж бази даних, що забезпечує інфраструктуру для зберігання та доступу до даних у середовищі AWS.

Наступним кроком буде створення файлу ***data.tf***, код якого наведений на рисунку 3.1.5. В цьому коді перший блок data "aws_ami" "latest_ubuntu", використовується для отримання найновішого AMI для Ubuntu. Він встановлює фільтри для пошуку AMI, вказуючи, що власником повинна бути "099720109477" (Canonical) та назва повинна відповідати певному шаблону. Цей блок допомагає обрати правильну AMI для створення екземплярів EC2 з операційною системою Ubuntu.

Блок data "aws_availability_zones" "availability" отримує дані про зони доступності для нашої бази даних.

```
data "aws_ami" "latest_ubuntu" {
    owners = ["099720109477"]
    most_recent = true
    filter {
        name = "name"
        values = ["ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-*"]
    }
}

data "aws_availability_zones" "availability"{}


data "template_file" "user_data" {
  template = "${file("user_data.sh")}"

  vars = {
    registry          = "${var.registry}"
    db_user           = "${var.usernamedb}"
    db_pass           = "${var.passworddb}"
    db_ip             = "${aws_db_instance.db.endpoint}"
  }
}
```

Рисунок 3.1.5 – «Код файлу data.tf»

Останній блок data "template_file" "user_data" використовується для читання файлу ***user_data.sh*** та обробки його як шаблону. У цьому блоку вказується шлях до файлу шаблону (template) та передаються змінні (vars), для запуску docker images з нашими параметрами для бази даних. Цей блок допомагає генерувати конфігураційний скрипт, який буде використовуватись під час створення екземплярів EC2.

У наступному файлі ***sshkey.tf***, код якого наведений на рисунку 3.1.6, створюємо наступні ресурси: для створення приватного ключа TLS – **tls_private_key**, для створення ключової пари AWS EC2 – **aws_key_pair**, для створення локального файлу – **local_file**.

Перший ресурс **tls_private_key** використовується для генерації приватного ключа TLS. Використовується алгоритм RSA з довжиною ключа 4096 біт.

```
resource "tls_private_key" "web" {
  algorithm = "RSA"
  rsa_bits  = 4096
}

resource "aws_key_pair" "generated_key_web" {
  key_name   = "Web-Key-${var.region}-${var.current_environment}"
  public_key = tls_private_key.web.public_key_openssh
  lifecycle {
    ignore_changes = [public_key]
  }
}


resource "local_file" "private_key" {
    content  = tls_private_key.web.private_key_pem
    filename = "~/${var.project}.pem"
}
```

Рисунок 3.1.6 – «Код файлу sshkey.tf»

Другий ресурс **aws_key_pair** використовує приватний ключ TLS для створення ключової пари AWS EC2. У цьому ресурсі вказується назва ключової пари та вихідний публічний ключ, що отриманий з приватного ключа TLS.

Третій ресурс **local_file** використовується для створення локального файлу. Вмістом цього файлу є приватний ключ TLS. Вказується шлях та назва файлу, до якого буде збережено приватний ключ.

Ці ресурси дозволяють створити приватний ключ, щоб надати можливість під'їдатися до екземплярів EC2 та зберігають його у локальний файл для подальшого використання.

Далі у файлі ***ec2.tf*** створюємо конфігурацію для автоматичного масштабування групи EC2 екземплярів, підключаємо їх до балансувальника навантаження та налаштовуємо політику масштабування для контролю за кількістю екземплярів, що запущені в залежності від навантаження.

На рисунку 3.1.7 показаний код для створення ресурсу **aws_launch_configuration**. Він потрібен для налаштування конфігурації запуску EC2 екземплярів. Вказуються ім'я, образ для створення EC2, тип екземпляру, групи безпеки, ключ SSH, дані з файлу ***user_data.tf*** та інші параметри.

```
resource "aws_launch_configuration" "web" {
  name_prefix          = "Web-${var.current_environment}-V${var.current_version}-"
  image_id             = data.aws_ami.latest_ubuntu.id
  instance_type        = var.instance_type
  security_groups      = [module.sg_web.sg_id]
  key_name             = aws_key_pair.generated_key_web.key_name
  user_data            = data.template_file.user_data.rendered
  lifecycle {
    create_before_destroy = true
  }

  depends_on = [
      aws_db_instance.db
    ]
}
```

Рисунок 3.1.7 – «Код для створення ресурсу конфігурації запуску EC2 екземплярів»

Блок lifecycle з параметром create_before_destroy встановлює життєвий цикл ресурсу таким чином, що новий ресурс буде створений перед тим, як старий буде знищений. Це дозволяє уникнути проблем з перервами в роботі системи під час оновлення ресурсів.

На рисунку 3.1.8 показаний код ресурсу **aws_autoscaling_group**, який встановлює групу автоматичного масштабування EC2 екземплярів, вказуючи мінімальний та максимальний розмір, кількість екземплярів для завантаження балансу, мережу VPC та екземпляри EC2, які входять до групи. Також встановлюються теги для ідентифікації ресурсів.

```
resource "aws_autoscaling_group" "web" {
  name = "ASG-${aws_launch_configuration.web.name}"
  launch_configuration = aws_launch_configuration.web.name
  min_size             = 3
  max_size             = 4
  min_elb_capacity     = 2
  health_check_type    = "ELB"

  vpc_zone_identifier  = module.vpc_web.public_subnet_ids
  load_balancers       = [aws_elb.web.name]

  dynamic "tag" {
    for_each = {
        Name = "Web-${var.current_environment}-V${var.current_version}"
        Owner = "Danylo Bosenko"
        Project = "bachelor"
    }
  content {
        key      = tag.key
        value    = tag.value
        propagate_at_launch = true
   }
  }

  lifecycle {
    create_before_destroy = true
  }

  depends_on = [
      aws_db_instance.db
    ]
}
```

Рисунок 3.1.8 – «Код для створення ресурсу автоматичного масштабування EC2 екземплярів»

Ресурс **aws_elb**, код на рисунку 3.1.9, створює балансувальник навантаження Elastic Load Balancer (ELB) для розподілу трафіку між EC2 екземплярами. Зазначаються ім'я, групи безпеки, підмережі VPC та налаштування перевірки стану здоров'я екземплярів.

```
resource "aws_elb" "web" {
    name = "WebServer-${var.current_environment}-V${var.current_version}"
    security_groups = [module.sg_web.sg_id]
    subnets = module.vpc_web.public_subnet_ids
    listener {
      instance_port     = 80
      instance_protocol = "http"
      lb_port           = 80
      lb_protocol       = "http"
    }
    health_check {
      healthy_threshold = 2
      unhealthy_threshold = 2
      timeout = 3
      target = "HTTP:80/"
      interval = 10
    }
    tags = merge(var.common_tag, {Name = "WebServer-Highly-Avaibility-ELB"})

    depends_on = [
      aws_db_instance.db
    ]
}
```

Рисунок 3.1.9 – «Код балансувальника навантаження»

depends_on - це важливий параметр, який вказаний на усіх ресурсах та вказує на залежності між ресурсами. У даному випадку, ресурс **aws_launch_configuration** та **aws_autoscaling_group** залежать від ресурсу **aws_db_instance.db**. Це означає, що перед створенням або оновленням **aws_launch_configuration** та **aws_autoscaling_group** буде спочатку створено або оновлено **aws_db_instance.db**, щоб забезпечити правильний порядок дій та доступ веб-додатку до бази даних.

Ресурс **aws_autoscaling_policy**, на рисунку 3.1.10, встановлює політику автоматичного масштабування, що визначає кількість екземплярів для додавання або видалення залежно від зміни навантаження

```
resource "aws_autoscaling_policy" "web" {
  name                   = "ASG-policy${aws_launch_configuration.web.name}"
  scaling_adjustment     = 4
  adjustment_type        = "ChangeInCapacity"
  cooldown               = 300
  autoscaling_group_name = aws_autoscaling_group.web.name
}
```

Рисунок 3.1.10 – «Політика автоматичного масштабування»

Зміна навантаження відслідковується у файлі ***alarm.tf***, на рисунку 3.1.11, за допомогою ресурсу **aws_cloudwatch_metric_alarm**.

```
resource "aws_cloudwatch_metric_alarm" "bat" {
  alarm_name          = "CW-metric-${var.project}"
  comparison_operator = "GreaterThanOrEqualToThreshold"
  evaluation_periods  = 2
  metric_name         = "CPUUtilization"
  namespace           = "AWS/EC2"
  period              = 120
  statistic           = "Average"
  threshold           = 80

  dimensions = {
    AutoScalingGroupName = aws_autoscaling_group.web.name
  }

  alarm_description = "This metric monitors ec2 cpu utilization"
  alarm_actions     = [aws_autoscaling_policy.web.arn]
}
```

Рисунок 3.1.11 – «Код файлу alarm.tf»

Ресурс **aws_cloudwatch_metric_alarm** використовується для створення спостережень метрик у сервісі AWS CloudWatch. В цьому коді створюється метрика спостереження для моніторингу використання CPU EC2 екземплярів. alarm_actions визначає дії, які повинні бути виконані, якщо спостереження спрацьовує (у цьому випадку, використовується політика автомасштабування).

Останнім кроком при пишемо файл конфігурації екземплярів EC2 – user_data. Це скрипт, на рисунку 3.1.12, який дозволяє нам налагодити екземпляри під час їх створення.


Рисунок 3.1.12 – «Код скрипту user_data.sh»

Цей скрипт виконує наступні дії: оновлення пакетів системи та встановлення пакету docker.io, отримання публічної IP-адреси EC2-екземпляра з метаданих AWS, виконання команди sudo docker run hello-world, що запускає контейнер "hello-world" для перевірки, чи працює Docker на екземплярі, завантаження образу контейнера з реєстру, зазначеного змінною ${registry}, за допомогою команди sudo docker pull ${registry}, запуск контейнера з налаштуваннями середовища для доступу до бази даних.

Змінні DATABASE_IP, MYSQL_USER та MYSQL_PASS встановлюються на відповідні значення ${db_ip}, ${db_user} та ${db_pass}. Контейнер слухатиме на порту 80 і буде проксіювати трафік до порту 8080 контейнера з образом ${registry}.

## 3.2 Налаштування AWS

Для Terroform, щоб отримати доступ до AWS потрібно попередньо створити акаунт відвідавши веб-сайт aws.amazon.com. Далі налаштовуємо політику Identity and Access Management (IAM), створити юзера та надати необхідні дозволи йому, а саме AdministratorAccess, для доступу до ресурсів AWS.

Після цього, створюємо та зберігаємо необхідні ідентифікатори доступу (Access Key ID) та секретні ключі (Secret Access Key) для автентифікації нашого коду Terraform.

Для зберігання стану та блокування ресурсів Terraform також потрібно створити S3 bucket та таблицю DynamoDB.

### 3.3 Налаштування GitLab та створення конфігураційного файлу

Попередньо створивши акаунт на сайті gitlab.com, створюємо репозиторій в якому будемо зберігати наші файли. Далі створюємо потрібні нам гілки, вони будуть вказувати на наше середовище, тобто для кожної обраної гілки (dev, prod, test, main), буде створюватись своя інфраструктура, для майбутнього керування та розвитку коду.

Наступним кроком, перед написанням конфігураційного файлу ***.gitlab-ci.yml*** потрібно встановити у GitLab, в налаштуваннях CI/CD, наші змінні, як показано на рисунку 3.3.1. Вони будуть використовуватись у конвеєрі, щоб передати у GitLab Runner ідентифікатори доступу (Access Key ID) та секретні ключі (Secret Access Key), а також пароль, ім'я користувача та назва docker image, який буде встановлений на EC2.

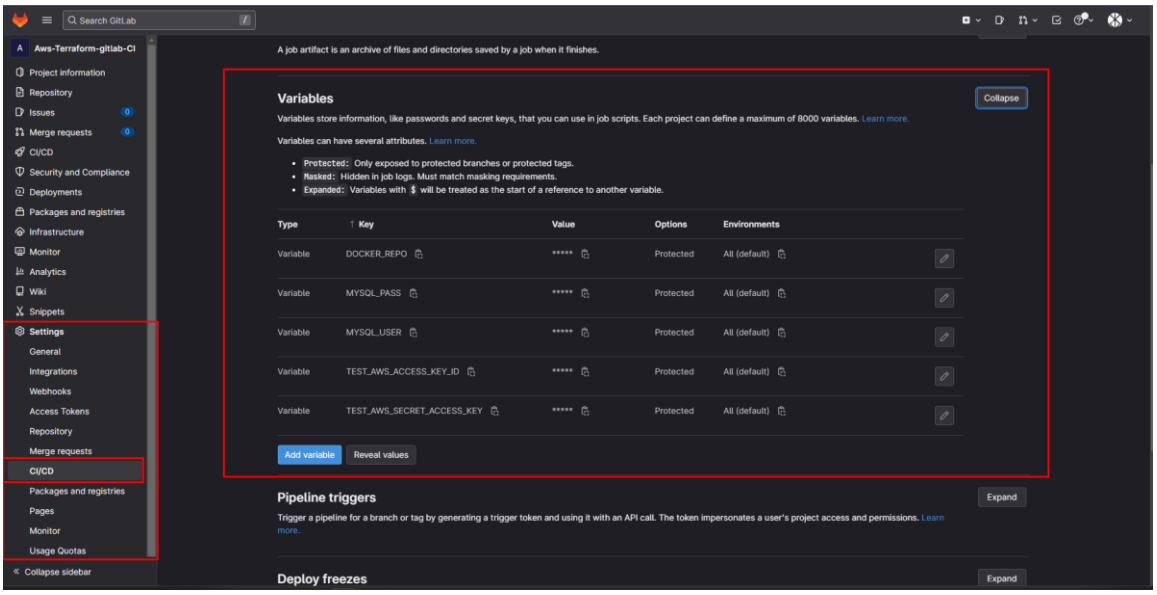



Рисунок 3.3.1 – «Налаштування змінних у GitLab CI/CD »

Після чого, переходимо до створення GitLab Runner. Він може бути встановлений будь де, на будь якій операційній системі та хмарному середовищі. Для зручності, GitLab Runner, було встановлено локально на віртуальну машину Ubuntu.

В налаштуваннях CI/CD, знаходимо поле Runners та встановлюємо GitLab Runner на Ubuntu за інструкцією. На рисунку 3.3.2 показані команди для встановлення Runner на лінукс.

```
danylo@danylo-VirtualBox:~$ sudo curl -L --output /usr/local/bin/gitlab-runner https://gitlab-runner-downloads.s3.amazon
aws.com/latest/binaries/gitlab-runner-linux-amd64
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 57.6M  100 57.6M    0     0  5309k      0  0:00:11  0:00:11 --:--:-- 6012k
danylo@danylo-VirtualBox:~$ sudo chmod +x /usr/local/bin/gitlab-runner
danylo@danylo-VirtualBox:~$ sudo useradd --comment 'GitLab Runner' --create-home gitlab-runner --shell /bin/bash
danylo@danylo-VirtualBox:~$ sudo gitlab-runner install --user=gitlab-runner --working-directory=/home/gitlab-runner
Runtime platform                                    arch=amd64 os=linux pid=42838 revision=79704081 version=16.0.1
danylo@danylo-VirtualBox:~$ sudo gitlab-runner start
Runtime platform                                    arch=amd64 os=linux pid=42950 revision=79704081 version=16.0.1
danylo@danylo-VirtualBox:~$ sudo systemctl status gitlab-runner
● gitlab-runner.service - GitLab Runner
     Loaded: loaded (/etc/systemd/system/gitlab-runner.service; enabled; vendor preset: enabled)
     Active: active (running) since Wed 2023-05-24 21:42:05 CEST; 17s ago
   Main PID: 42955 (gitlab-runner)
      Tasks: 6 (limit: 2272)
     Memory: 11.2M
     CGroup: /system.slice/gitlab-runner.service
             └─42955 /usr/local/bin/gitlab-runner run --working-directory /home/gitlab-runner --config /etc/gitlab-runn>

Mai 24 21:42:05 danylo-VirtualBox systemd[1]: Started GitLab Runner.
Mai 24 21:42:05 danylo-VirtualBox gitlab-runner[42955]: Runtime platform                                    arch=amd64 >
Mai 24 21:42:05 danylo-VirtualBox gitlab-runner[42955]: Starting multi-runner from /etc/gitlab-runner/config.toml...  b>
Mai 24 21:42:05 danylo-VirtualBox gitlab-runner[42955]: Running in system-mode.
Mai 24 21:42:05 danylo-VirtualBox gitlab-runner[42955]:
Mai 24 21:42:05 danylo-VirtualBox gitlab-runner[42955]: Created missing unique system ID                    system_id=s>
Mai 24 21:42:05 danylo-VirtualBox gitlab-runner[42955]: Configuration loaded                                builds=0
Mai 24 21:42:05 danylo-VirtualBox gitlab-runner[42955]: listen_address not defined, metrics & debug endpoints disabled >
```

Рисунок 3.3.2 – «Команди для встановлення GitLab Runner»

Далі треба зареєструвати Runner, щоб з’єднати його з GitLab. Для цього потрібно використати токен, який вказаний в налаштуваннях CI/CD Runners.

На рисунку 3.3.3 показаний приклад команди за допомогою якої це можна зробити. Після її запуску данні, вводяться в інтерактивному режимі, залишаємо усі за замовчуванням, окрім description for the runner, tags та executor.

```
danylo@danylo-VirtualBox:~$ sudo gitlab-runner register --url https://gitlab.com/ --registration-token GR1348941uvjcC1Az
yeDP6CEGmLWy
[sudo] password for danylo:
Runtime platform                                    arch=amd64 os=linux pid=1914 revision=79704081 version=16.0.1
Running in system-mode.

Enter the GitLab instance URL (for example, https://gitlab.com/):
[https://gitlab.com/]:
Enter the registration token:
[GR1348941uvjcC1AzyeDP6CEGmLWy]:
Enter a description for the runner:
[danylo-VirtualBox]: terraform
Enter tags for the runner (comma-separated):
terraform
Enter optional maintenance note for the runner:

WARNING: Support for registration tokens and runner parameters in the 'register' command has been deprecated in GitLab R
unner 15.6 and will be replaced with support for authentication tokens. For more information, see https://gitlab.com/git
lab-org/gitlab/-/issues/380872
Registering runner... succeeded                     runner=GR1348941uvjcC1Az
Enter an executor: instance, docker, docker-windows, shell, ssh, docker-autoscaler, docker+machine, custom, parallels, v
irtualbox, kubernetes:
shell
Runner registered successfully. Feel free to start it, but if it's running already the config should be automatically re
loaded!

Configuration (with the authentication token) was saved in "/etc/gitlab-runner/config.toml"
```

Рисунок 3.3.3 – «Приклад реєстрації GitLab Runner»

Важливо щоб сервер, на якому встановлений GitLab Runner, мав доступ в інтернет, а також щоб на ньому був встановлений Terraform.

Після встановлення GitLab Runner, створюємо YAML-файл ***.gitlab-ci.yml***, рисунок 3.3.4, в якому налаштовуємо розгортання інфраструктури за допомогою Terraform.

```yaml
image: terraform

default:
  tags:
    - terraform

stages:
  - validate
  - plan
  - apply
  - destroy

before_script:
  - export AWS_ACCESS_KEY_ID=$TEST_AWS_ACCESS_KEY_ID
  - export AWS_SECRET_ACCESS_KEY=$TEST_AWS_SECRET_ACCESS_KEY
  - terraform init -backend-config="key=$CI_COMMIT_BRANCH/terraform.tfstate" -reconfigure
validate:
  stage: validate
  script:
    - terraform validate
plan:
  stage: plan
  script:
    - terraform plan -var usernamedb=$MYSQL_USER -var passworddb=$MYSQL_PASS -var registry=$DOCKER_REPO
  dependencies:
    - validate
apply:
  stage: apply
  script:
    - terraform apply -input=false -var current_version='1' -var current_environment=$CI_COMMIT_BRANCH -var usernamedb=$MYSQL_USER -var passworddb=$MYSQL_PASS  -var registry=$DOCKER_REPO -no-color --auto-approve
  dependencies:
    - validate
destroy:
  stage: destroy
  script:
    - terraform destroy -var usernamedb=$MYSQL_USER -var passworddb=$MYSQL_PASS -var registry=$DOCKER_REPO -no-color -auto-approve
  when: manual
```

Рисунок 3.3.4 – «Конфігураційний файл .gitlab-ci.yml»

Розглянемо цей файл більш поетапно:

1. Визначення образу (image) для виконання задач у різних стадіях конвеєру. У даному випадку використовується сервер, на який встановлено Terraform та GitLab Runner, з попередніх дій.

2. Визначення стадій конвеєру, які відповідають окремим етапам розгортання інфраструктури: validate (перевірка конфігурації), plan (створення плану розгортання), apply (застосування змін), destroy (знищення інфраструктури).

3. Визначення команд, які виконуються перед початком кожного етапу конвеєру. У цьому випадку встановлюються змінні середовища AWS_ACCESS_KEY_ID і AWS_SECRET_ACCESS_KEY для доступу до AWS.

4. Визначення задачі validate, яка виконує перевірку правильності конфігураційних файлів Terraform за допомогою команди **terraform validate**. Ця задача відноситься до стадії validate.

5. Визначення задачі plan, яка генерує план розгортання інфраструктури за допомогою команди **terraform plan**. У цій задачі передаються змінні, такі як usernamedb, passworddb, registry, які використовуються в конфігурації Terraform. Задача plan залежить від завершення задачі validate.

6. Визначення задачі apply, яка застосовує зміни до інфраструктури за допомогою команди **terraform apply**. У цій задачі також передаються змінні, такі як current_version, current_environment, usernamedb, passworddb, registry. Задача apply залежить від завершення задачі validate.

7. Визначення задачі destroy, яка знищує створену інфраструктуру за допомогою команди **terraform destroy**. Ця задача виконується тільки вручну (when: manual).

Таким чином ми отримуємо CI/CD конвеєр, який автоматично при зміні будь якого файлу в нашому репозиторії буде змінювати нашу інфраструктуру так як нам це потрібно.

# 4. ДЕМОНСТРАЦІЯ ПРОЦЕСУ РОБОТИ

Для перевірки роботи проекту зробимо зміни, а саме змінимо cidrs блоки у модулі **vpc_web**, у гілці dev та відправимо зміни до GitLab.

На рисунку 4.1 показаний приклад команд Git, для коментування внесених змін – **git commit -am**, та відправки їх у віддалений репозиторій – **git push origin «branch»**.

```
PS C:\Users\Danil\aws-terraform-gitlab-ci> git commit -am "change vpc_cidr block and public subnet cidrs"
[dev d17c37c] change vpc_cidr block and public subnet cidrs
 1 file changed, 2 insertions(+), 2 deletions(-)
PS C:\Users\Danil\aws-terraform-gitlab-ci> git push origin dev
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 16 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 337 bytes | 337.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
remote:
remote: To create a merge request for dev, visit:
remote:   https://gitlab.com/danildoter.db/aws-terraform-gitlab-ci/-/merge_requests/new?merge_request%5Bsource_branch%5D=dev
remote:
To gitlab.com:danildoter.db/aws-terraform-gitlab-ci.git
   3203b4b..d17c37c  dev -> dev
```

Рисунок 4.1 – «Приклад команд Git»

Після внесених змін одразу запускається конвеєр. В робочій панелі GitLab у розділі CI/CD, бачимо процес його виконання, рисунок 4.2.

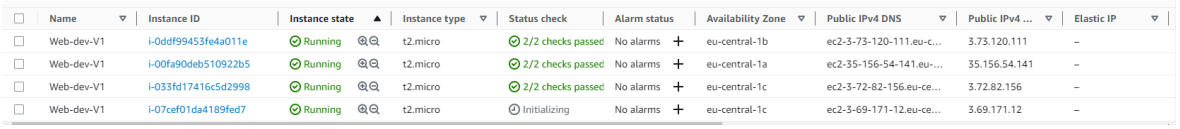

Рисунок 4.2 – «Процес виконання конвеєру»

Після успішного виконання конвеєру, у консолі AWS, в сервісі EC2, з’являються вже готові екземпляри  рисунок 4.3.

| Name | Instance ID | Instance state | Instance type | Status check | Alarm status | Availability Zone |
|---|---|---|---|---|---|---|
| Web-dev-V1 | i-0ddf99453fe4a011e | Running | t2.micro | 2/2 checks passed | No alarms | eu-central-1b |
| Web-dev-V1 | i-00fa90deb510922b5 | Running | t2.micro | 2/2 checks passed | No alarms | eu-central-1a |
| Web-dev-V1 | i-033fd17416c5d2998 | Running | t2.micro | 2/2 checks passed | No alarms | eu-central-1c |

Рисунок 4.4 – «Екземпляри EC2»

Далі перевіряємо роботу Auto Scaling Grop, заходимо на екземпляри, та запускаємо команду – yes > /dev/null, для того, щоб навантажити наші веб сервери. Як результат, в EC2 створиться новий додатковий екземпляр для, рисунок 4.5. При зменшенні навантаження, цей новий екземпляр видалиться



| Name | Instance ID | Instance state | Instance type | Status check | Alarm status | Availability Zone | Public IPv4 DNS | Public IPv4 ... | Elastic IP |
|---|---|---|---|---|---|---|---|---|---|
| Web-dev-V1 | i-0ddf99453fe4a011e | Running | t2.micro | 2/2 checks passed | No alarms | eu-central-1b | ec2-3-73-120-111.eu-c... | 3.73.120.111 | – |
| Web-dev-V1 | i-00fa90deb510922b5 | Running | t2.micro | 2/2 checks passed | No alarms | eu-central-1a | ec2-35-156-54-141.eu-... | 35.156.54.141 | – |
| Web-dev-V1 | i-033fd17416c5d2998 | Running | t2.micro | 2/2 checks passed | No alarms | eu-central-1c | ec2-3-72-82-156.eu-ce... | 3.72.82.156 | – |
| Web-dev-V1 | i-07cef01da4189fed7 | Running | t2.micro | Initializing | No alarms | eu-central-1c | ec2-3-69-171-12.eu-ce... | 3.69.171.12 | – |

Рисунок 4.5 – «Результат роботи автоматичного масштабування»

Для перевірки Elastic Load Balancer треба отримати DNS-ім'я ELB: Після створення ELB в консолі AWS або в лозі конвеєру GilLab CI (наприклад - example-elb-1234567890.us-west-2.elb.amazonaws.com). Відкрити веб-браузер і ввести DNS-ім'я ELB. На рисунку 4.6 показаний наш веб-додаток.

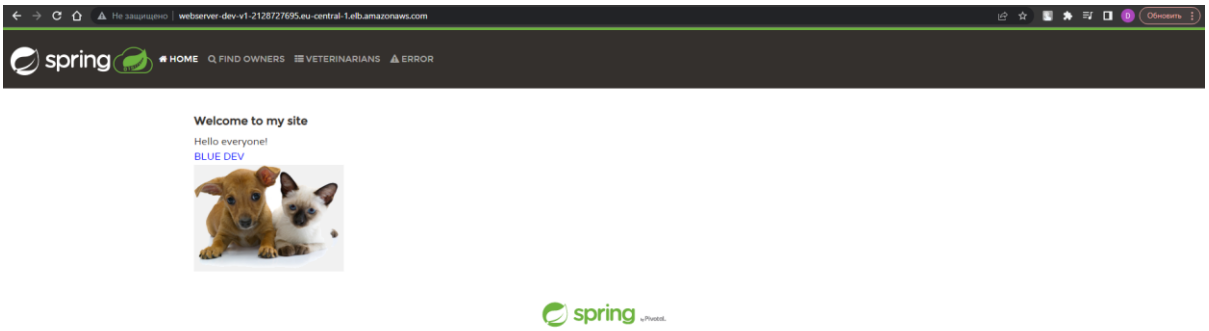



Рисунок 4.6 – «Веб сторінка додатку»

На нашій сторінці, одразу ж перевіримо роботу бази даних, переходимо у вкладку Find Owners та натискаємо Add Owner. Вводимо данні про нашого власника та натискаємо Add Owner. Рисунок 4.7.

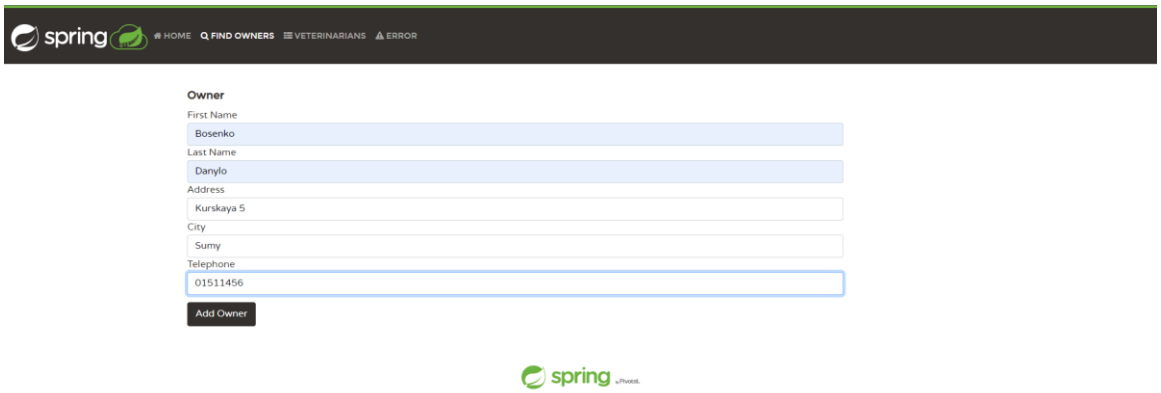



Рисунок 4.7 – «Вкладка Find Owners у веб-додатку»

Веб додаток успішно додав нашого нового власника, тому можемо зробити висновок, що він має доступ до бази даних. Рисунок 4.8.

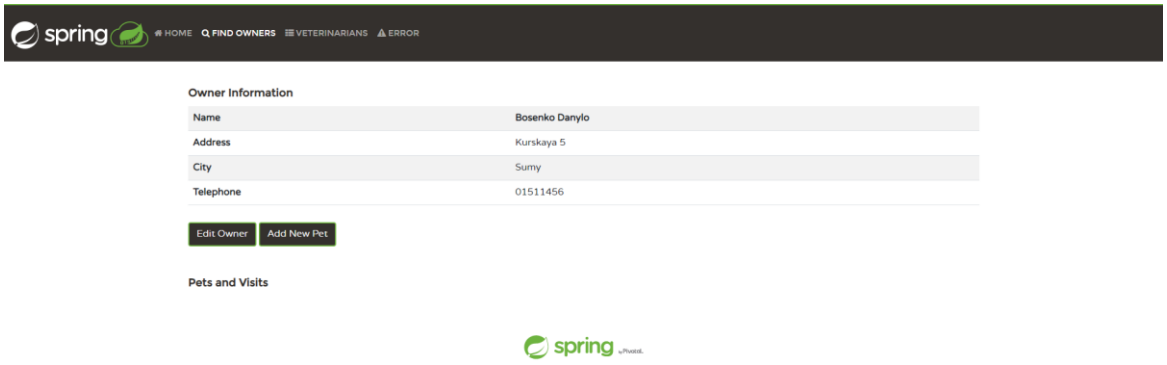



Рисунок 4.8 – «Вкладка Find Owners у веб-додатку після натискання Add Owner»

# ВИСНОВОК

Таким чином можна зробити висновок про те, що була створений CI/CD конвеєр, за допомогою GitLab CI, який автоматично створює масштабовану, відмовостійку та гнучку для змін інфраструктуру у хмарі AWS, написану у вигляді коду, з Terraform. Цю інфраструктуру легко підтримувати, перевикористовувати в інших проектах, тому що все описано як код. Данна інфраструктура може буде змінена в разі потреби, додані нові ресурси, обрані інші регіони, тощо.

# СПИСОК ЛІТЕРАТУРИ

1. Офіційна документація Terraform. URL: https://developer.hashicorp.com/terraform/docs

2. Офіційна документація GitLab. URL:https://docs.gitlab.com/

3. Офіційна документація Amazon Web Services. URL:https://docs.aws.amazon.com/

4. Gene Kim, Kevin Behr, George Spafford - The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win

5. Gene Kim, Jez Humble, Patrick Debois John Willis - The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations

6. Paul Swartout - Continuous Delivery and DevOps: A Quickstart Guide"