MINISTRY OF EDUCATION AND TRAINING
CAN THO UNIVERSITY
COLLEGE OF INFORMATION AND COMMUNICATION TECHNOLOGY

PROJECT – SPECIALIZED TOPICS IN INFORMATION TECHNOLOGY

Topic

DEPLOYMENT AUTOMATION

Student: Truong Dang Truc Lam
ID: B2111933
Course: K47

Can Tho, 11/2024

MINISTRY OF EDUCATION AND TRAINING
CAN THO UNIVERSITY
COLLEGE OF INFORMATION AND COMMUNICATION TECHNOLOGY
DEPARTMENT OF INFORMATION TECHNOLOGY

PROJECT – SPECIALIZED TOPICS IN INFORMATION TECHNOLOGY

Topic

DEPLOYMENT AUTOMATION

Advisor:
Dr. Thai Minh Tuan

Student:
Truong Dang Truc Lam
ID: B2111933
Course: K47

Can Tho, 11/2024

REMARK OF ADVISOR

.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....

Can Tho, 2024

Advisor

Dr. Thai Minh Tuan

ACKNOWLEDGEMENTS

I would first like to thank my advisor, Dr. Thai Minh Tuan. Without his assistance and dedicated involvement in every step throughout the process, this paper could have never been accomplished. Moreover, his knowledge and experience have inspired me throughout my academic career and everyday life.

I take this opportunity to express gratitude to all of The College of Information and Communication Technology members for for sharing expertise, giving valuable guidance and encouragement extended to me from first year until now.

Finally, I must express my very profound gratefulness to my family for constantly encouraging and supporting me during my years of studies.

Author

Truong Dang Truc Lam

CONTENTS

ABSTRACT 5
INTRODUCTION 6
1. Problems 6
2. Object 6
3. Research scope 7
4. Research method 7
MAIN CONTENTS 8
CHAPTER 1: REQUIREMENT SPECIFICATION 8
1. Introduction 8
2. System Description 8
3. Requirements 8
3.1. Functional Requirements 8
3.2. Non-Functional Requirements 9
CHAPTER 2: THEORETICAL BASIS 10
1. Cloud Computing 10
1.1. Infrastructure as a Service 13
1.2. Amazon Web Services 15
2. DevOps 16
2.1. Terraform 19
2.2. GitLab 20
CHAPTER 3: SOLUTION DESIGN 22
1. Conceptual design 22
2. Detailed design 23
CHAPTER 4: SOLUTION IMPLEMENTATION 24
1. AWS Provisioning 24
1.1. AWS IAM 24
1.2. AWS S3 25
1.3. AWS DynamoDB 25
1.4. AWS security credentials 26
1.5. AWS key pairs 27
1.6. AWS CLI 27
2. Terraform for AWS deployment 28
2.1. Terraform installation 28
2.2. Terraform configuration 28
2.3. Terraform tests 31
3. GitLab for Terraform CI/CD 31
3.1. GitLab project preparation 31
3.2. GitLab CI/CD pipeline 33
CHAPTER 4: TEST AND EVALUATE 35
1. Test and evaluate AWS deployment with Terraform 35
2. Test and evaluate CI/CD pipeline on GitLab 42
3. Test and evaluate AWS autoscaling 45
CONCLUSION 46
1. Result 46
2. Development orientation 46
REFERENCES 47

TABLE OF IMAGES

Figure 1: Cloud computing architecture. Source: spiceworks.com 12
Figure 2: Comparison of IaaS and other models. Source: redhat.com 14
Figure 3: Amazon Web Services. Source: aws.amazon.com.....16
Figure 4: DevOps Lifecycle. Source: cloud.z.com 18
Figure 5: Operation of Terraform. Source: whizlabs.com 20
Figure 6: GitLab features. Source: GitLab.....21
Figure 7: Cloud infrastructure automation conceptual design 22
Figure 8: Cloud infrastructure detailed design.....23
Figure 9: AWS IAM user.....24
Figure 10: AWS S3 bucket 25
Figure 11: AWS DynamoDB table 26
Figure 12: AWS access keys.....27
Figure 13: AWS key pairs.....27
Figure 14: Terraform installed 28
Figure 15: Terraform configuration 30
Figure 16: AWS resources 30
Figure 17: Terraform unit tests 31
Figure 18: GitLab repository.....32
Figure 19: Define variables for GitLab CI/CD 33
Figure 20: The content of `.gitlab-ci.yml` file 34
Figure 21: Run command `terraform init` successfully.....35
Figure 22: Run command `terraform validate` successfully.....35
Figure 23: Run command `terraform test` successfully.....36
Figure 24: Run command `terraform plan` successfully.....36
Figure 25: Run command `terraform apply` successfully.....37
Figure 26: Verify result on AWS console after applying 37
Figure 27: Information of the Amazon Linux instance on AWS.....38
Figure 28: Connect to the Amazon Linux instance via SSH 38
Figure 29: Information of the Ubuntu instance on AWS.....39
Figure 30: Connect to the Ubuntu instance via SSH 39
Figure 31: Information of the Windows instance on AWS 40
Figure 32: Connect to the Windows instance via RDP.....40
Figure 33: Run command `terraform destroy` successfully.....41
Figure 34: Verify result on AWS console after destroying 41
Figure 35: Automated run validation, testing and planning stages after a commit ..42
Figure 36: Automated stages on GitLab CI/CD passed successfully 42
Figure 37: Manual applying with GitLab CI/CD passed successfully 43
Figure 38: Verify result on AWS console after applying with GitLab CI/CD 43
Figure 39: Manual destroying with GitLab CI/CD passed successfully.....44
Figure 40: Verify result on AWS console after destroying with GitLab CI/CD 44
Figure 41: GitLab CI/CD pipline passed successfully after upscaling 45
Figure 42: Verify result on AWS console after upscaling.....45

ABSTRACT

In the rapidly evolving digital landscape, the velocity and reliability of software delivery are indispensable factors in achieving success, especially with the rise of cloud computing and DevOps practices. To meet these demands, organizations are increasingly adopting automation tools and practices to streamline their development and deployment processes. This paper will research the key technologies that constitute modern deployment automation, including Infrastructure as Code (IaC) and Continuous Integration/Continuous Delivery (CI/CD).

By adopting Infrastructure as Code, organizations can leverage declarative IaC syntax to define infrastructure resources, automating their provisioning in a consistent, repeatable, and scalable way. This approach leverages code-based templates to automate the creation and management of infrastructure components like servers, networks, and storage systems. CI/CD pipelines, on the other hand, will automate the software development lifecycle. Integrated IaC and CI/CD will create a seamless workflow, automating infrastructure builds, tests, and deployments.

Additionally, the advent of cloud computing and DevOps practices has further amplified the benefits of deployment automation. Cloud platforms like AWS, Azure or GCP will provide scalable and flexible infrastructure, enabling organizations to rapidly provision and manage resources on-demand. While DevOps is a collaborative approach that unites development and operations teams, promoting automation and continuous delivery. This enables organizations to define their cloud resources in IaC and integrate them into CI/CD pipelines for automated DevOps processes.

In conclusion, this paper will delve into the specific practices for implementing deployment automation, including the selection of appropriate tools and technologies for designing efficient IaC principles and CI/CD pipelines. By automating the process from code commit to production, organizations can significantly reduce manual intervention, minimize the risk of human error, and accelerate the deployment of new features and updates. This research will also discuss the challenges and potential problems associated with deployment automation, such as security risks and configuration management.

INTRODUCTION

1. Problems

Nowadays, cloud computing [1] revolutionizes how we consume resources, transforming everything from infrastructure to software into accessible services. While this paradigm shift offers unparalleled flexibility and scalability, manual management of these resources can be time-consuming. To address these challenges and fully utilize the potential of cloud computing, DevOps principles can be applied to automate resource provisioning, configuration, and management. By integrating automation tools and scripts into the DevOps pipeline, organizations can streamline the deployment of cloud resources, reducing manual effort and minimizing the risk of human error. This automated approach enables teams to respond rapidly to changing business needs, assuring the perpetual accessibility and peak operational efficiency of applications.

DevOps [2] has become an indispensable component of modern software development, revolutionizing the way teams collaborate and deliver high-quality products. By fostering a culture of collaboration between development and operations teams, DevOps breaks down silos and promotes efficient workflows. This streamlined approach leads to faster time-to-market, improved product quality, and enhanced customer satisfaction. DevOps practices, such as continuous integration and continuous delivery (CI/CD) [3], automate repetitive tasks, reducing the risk of human error and enabling teams to focus on innovation. Additionally, DevOps emphasizes monitoring and feedback loops, allowing organizations to identify and address issues proactively, ensuring a reliable software infrastructure.

2. Object

The object of this research is to investigate and demonstrate the effectiveness of utilizing Infrastructure as Code (IaC) [4] in conjunction with a CI/CD pipeline for automating infrastructure deployment on a cloud platform.

This approach involves defining infrastructure resources as code and integrating these definitions into a CI/CD workflow. By incorporating DevOps principles into this process, we can further enhance the efficiency and reliability of infrastructure management with faster deployment cycles.

3. Research scope

Cloud computing platform: Amazon Web Services (AWS) [5]

DevOps platform: GitLab [6]

Infrastructure as code tool: Terraform [7]

4. Research method

This research is about the implementation of a Terraform CI/CD pipeline using GitLab to automate the deployment of infrastructure on AWS cloud computing platform. By leveraging GitLab's CI/CD capabilities and Terraform's infrastructure as code (IaC) approach, we aim to streamline the provisioning and management of AWS resources, ensuring efficient and reliable deployments.

MAIN CONTENTS

CHAPTER 1: REQUIREMENT SPECIFICATION

1. Introduction

This document outlines the requirements for a DevOps project aimed at automating the deployment of cloud resources through Infrastructure as Code (IaC) and Continuous Integration/Continuous Delivery (CI/CD) pipelines. The primary objective is to streamline the process of provisioning and managing cloud infrastructure, ensuring consistency, reliability, and efficiency.

2. System Description

This system is designed to automate the deployment of AWS resources using Terraform and GitLab CI/CD. It leverages the power of Infrastructure as Code (IaC) through Terraform, defining and managing cloud infrastructure in a declarative manner, while GitLab CI/CD provides a robust platform for continuous integration and delivery.

3. Requirements

3.1. Functional Requirements

Automated Infrastructure Provisioning: The system should automate the provisioning of AWS infrastructure using Terraform scripts. These scripts will define and create all necessary resources, such as EC2 instances, VPCs, S3 buckets, and IAM roles, based on a configuration file. The provisioning process should be fully automated and repeatable.

GitLab CI/CD Integration: A GitLab CI/CD pipeline should be integrated to trigger Terraform runs automatically when code changes are pushed to the repository. The pipeline will include stages for building, testing, and deploying the application.

Configuration Management: Terraform state should be utilized to track and manage infrastructure changes, enabling seamless rollback if needed. To promote collaboration and maintain a record of modifications, configuration files should be versioned in Git.

Environment Management: The system should support the creation of multiple environments, such as development, staging, and production, with distinct configurations. Environment-specific variables should be securely managed and injected into the Terraform scripts.

Infrastructure as Code (IaC) Best Practices: The system should adhere to best practices for IaC, including modularity, reusability, and readability. The pipeline should also include linting and validation steps to ensure code quality.

3.2. Non-Functional Requirements

Security: The system must prioritize security by ensuring all AWS resources are configured with appropriate measures, such as IAM policies and security groups. Sensitive information should be stored securely using secrets management tools.

Reliability: The deployment process should be reliable and resilient to failures. Rollback mechanisms should be in place to revert to a previous state if necessary.

Scalability: The infrastructure should be designed to scale horizontally and vertically to accommodate increasing workloads. The deployment process should also be scalable to handle large-scale deployments.

Performance: The deployment pipeline should be optimized for performance to minimize deployment times. Infrastructure resources should be configured to meet performance requirements.

Maintainability: The infrastructure and pipeline should be well-documented and easy to maintain. Changes to the infrastructure should be easy to implement and test.

CHAPTER 2: THEORETICAL BASIS

1. Cloud Computing

According to the NIST (National Institute of Standards and Technology) definition [8]: Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models.

Essential Characteristics:

- On-demand self-service: A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider.
- Broad network access: Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, tablets, laptops, and workstations).
- Resource pooling: The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, and network bandwidth.
- Rapid elasticity: Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate on demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.

- Measured service: Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

Service Models:

- Software as a Service (SaaS): The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure . The applications are accessible from various client devices through either a thin client interface, such as a web browser or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited userspecific application configuration settings.
- Platform as a Service (PaaS): The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment.
- Infrastructure as a Service (IaaS): The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components.

Deployment Models:

- Private cloud. The cloud infrastructure is provisioned for exclusive use by a single organization comprising multiple consumers. It may be owned, managed, and operated by the organization, a third party, or some combination of them, and it may exist on or off premises.
- Community cloud. The cloud infrastructure is provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns. It may be owned, managed, and operated by one or more of the organizations in the community, a third party, or some combination of them, and it may exist on or off premises.
- Public cloud. The cloud infrastructure is provisioned for open use by the general public. It may be owned, managed, and operated by a business, academic, or government organization, or some combination of them. It exists on the premises of the cloud provider.
- Hybrid cloud. The cloud infrastructure is a composition of two or more distinct cloud infrastructures that remain unique entities, but are bound together by standardized or proprietary technology that enables data and application portability.

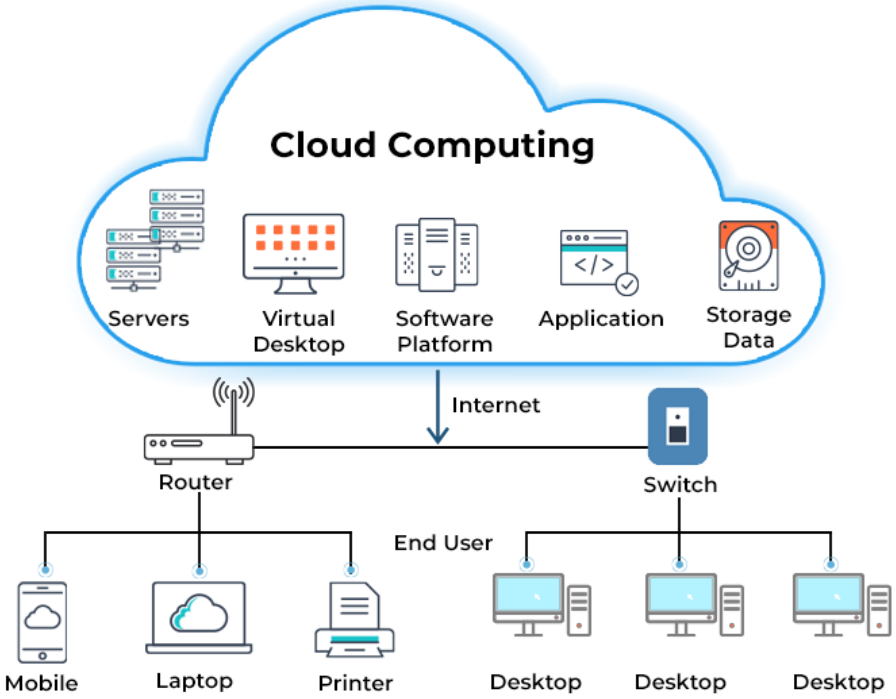

Figure 1: Cloud computing architecture. Source: spiceworks.com

1.1. Infrastructure as a Service

Infrastructure as a Service (IaaS) is a transformative paradigm in the realm of cloud computing, offering organizations a scalable, flexible, and cost-effective approach to provisioning and managing IT infrastructure. By abstracting the underlying hardware and software, IaaS providers enable businesses to focus on their core competencies while benefiting from the advantages of cloud-based solutions.

IaaS provides organizations with a virtualized computing environment where they can rent hardware resources, such as servers, storage, and networking equipment, on a pay-as-you-go basis. This model eliminates the need for significant upfront capital expenditures and allows businesses to scale their infrastructure resources up or down to meet fluctuating demand. The key features of IaaS include:

- On-demand provisioning: Organizations can quickly provision and deploy virtual machines and other resources as needed.
- Scalability: IaaS allows businesses to easily scale their infrastructure to accommodate growth or fluctuations in workload.
- Pay-as-you-go pricing: Organizations only pay for the resources they use, avoiding unnecessary costs.
- Self-service: Users can manage and control their infrastructure resources through a web-based portal or API.

When compared to other cloud computing models, IaaS offers a greater degree of control and flexibility. Platform as a Service (PaaS) provides a pre-configured platform for application development and deployment, while Software as a Service (SaaS) delivers applications as a service over the internet. IaaS, on the other hand, gives organizations the freedom to choose their own operating systems, software, and applications, providing maximum customization and control. However, IaaS also presents some challenges. Organizations must manage and secure their own operating systems, software, and applications, which can be complex and time-consuming.

The IaaS market is highly competitive, with several major players dominating the landscape. A comparison of prominent IaaS providers reveals key differences in terms of features, pricing, and target audience:

Amazon Web Services (AWS): As the pioneer of cloud computing, AWS offers a comprehensive suite of IaaS services, including EC2 (Elastic Compute Cloud), S3 (Simple Storage Service), and VPC (Virtual Private Cloud). AWS is known for its extensive feature set, global reach, and enterprise-grade capabilities.

Microsoft Azure: Azure provides a cloud platform that integrates seamlessly with Microsoft's existing software ecosystem. It offers a wide range of IaaS services, including virtual machines, storage, and networking. Azure is particularly popular among organizations that rely heavily on Microsoft technologies.

Google Cloud Platform (GCP): GCP is a relatively new entrant to the IaaS market, but it has quickly gained traction due to its innovative features and competitive pricing. GCP is well-suited for organizations that require high-performance computing, data analytics, and machine learning capabilities.

IBM Cloud: IBM offers a comprehensive cloud platform that includes IaaS, PaaS, and SaaS offerings. IBM Cloud is known for its focus on security, compliance, and enterprise-grade solutions.

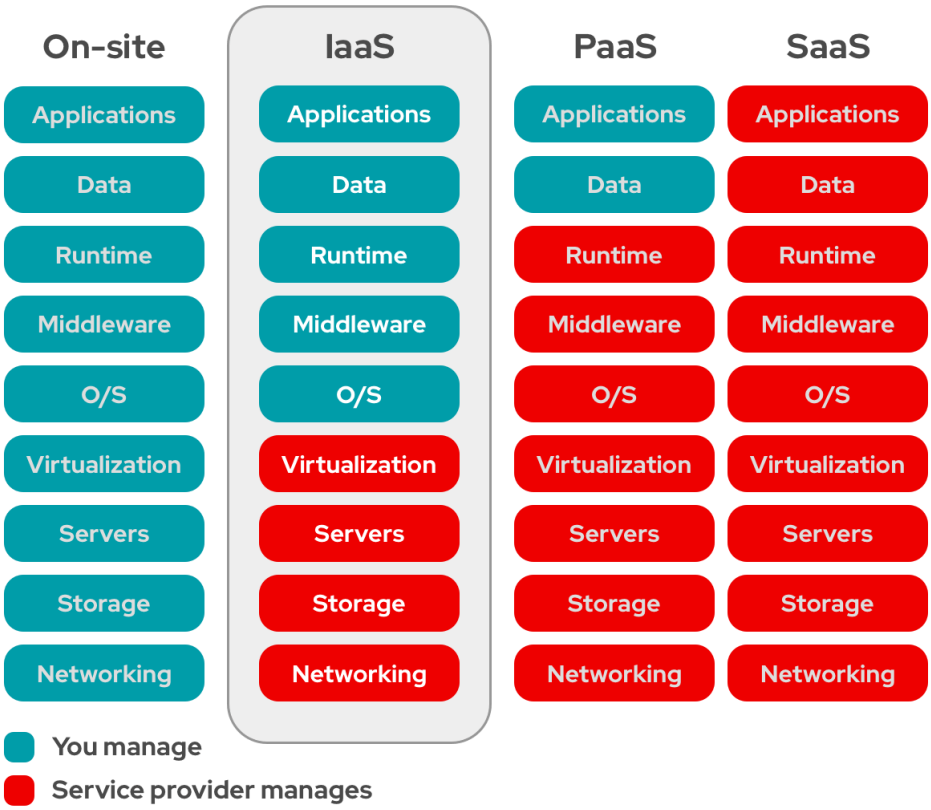

Figure 2: Comparison of IaaS and other models. Source: redhat.com

1.2. Amazon Web Services

Amazon Web Services (AWS) has become a dominant force in the realm of cloud computing, revolutionizing the way businesses and organizations operate. By providing a comprehensive suite of cloud services, AWS has empowered individuals and enterprises to scale their operations, reduce costs, and innovate at an unprecedented pace.

Launched in 2006, AWS was a pioneering venture that aimed to offer scalable, reliable, and affordable cloud computing services. Building upon Amazon's vast infrastructure and expertise in e-commerce, AWS introduced a pay-as-you-go model that allowed customers to access computing resources without the need for significant upfront investments. This innovative approach quickly gained traction, attracting a diverse range of users from startups to large corporations. At the heart of AWS's success lies its extensive portfolio of cloud services, each designed to address specific needs and requirements:

- Compute: Amazon Elastic Compute Cloud (EC2) provides virtual servers (instances) that can be customized to meet various workloads.
- Storage: Amazon Simple Storage Service (S3) offers durable and scalable object storage for data of any size.
- Database: Amazon Relational Database Service (RDS) manages and simplifies the administration of relational databases, such as MySQL, PostgreSQL, and Oracle.
- Networking: Amazon Virtual Private Cloud (VPC) enables customers to create isolated virtual networks within the AWS cloud.
- Analytics: Amazon Redshift provides a fully managed, petabyte-scale data warehouse for analytics and reporting.

The impact of AWS on the technology landscape has been profound. It has democratized access to computing resources, empowering individuals and small businesses to compete with larger enterprises. Additionally, AWS has fostered innovation by enabling developers to build and deploy applications at a faster pace. The widespread adoption of AWS has led to the creation of a vibrant ecosystem of partners, tools, and services that further enhance its value proposition.

In conclusion, Amazon Web Services has revolutionized the cloud computing industry by providing a comprehensive suite of scalable, reliable, and cost-effective services. Its impact has been felt across various sectors, from startups to large corporations. As AWS continues to evolve and expand its offerings, it is poised to remain a dominant force in the cloud computing landscape for years to come.

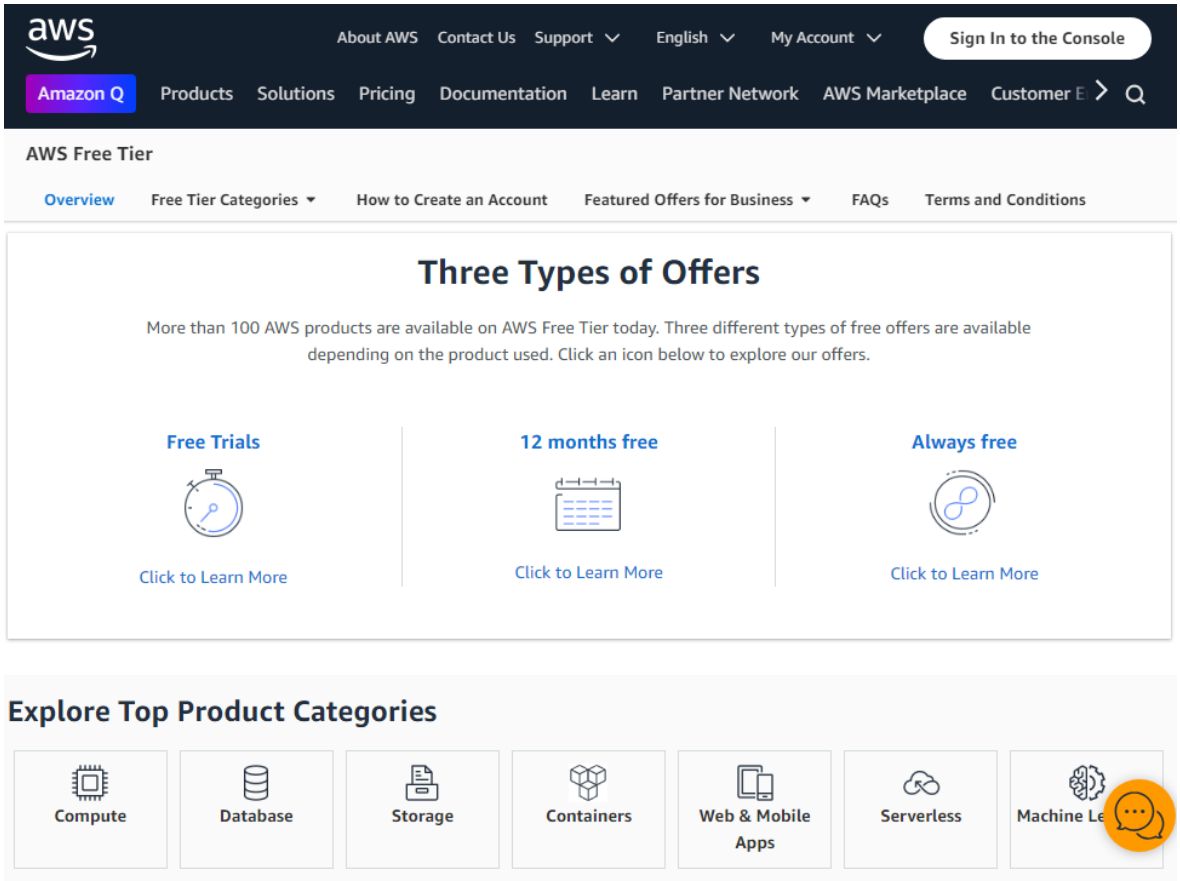

Figure 3: Amazon Web Services. Source: aws.amazon.com

2. DevOps

DevOps is a portmanteau of "development" and "operations" which has emerged as a transformative approach in contemporary software engineering and delivery in the early 2000s. At the core of DevOps is a culture of collaboration and shared responsibility. Traditional software development models often siloed development and operations teams, leading to communication breakdowns and delays. DevOps seeks to break down these barriers by encouraging cross-functional teams to work together throughout the software development process. DevOps is typically broken down into eight distinguished phases as an operational model. The phases operate in a continuous loop:

- Plan: The planning stage is the foundation of the DevOps process. It involves defining project goals, identifying requirements, and outlining the development roadmap. Teams collaborate to create a shared vision and ensure alignment between business objectives and technical implementation.
- Code: In the coding stage, developers write the actual code that will bring the project to life. They follow coding standards and best practices to ensure code quality and maintainability. Version control systems are used to track changes and facilitate collaboration among team members.
- Build: The build stage involves compiling the code into executable artifacts. Automation tools are used to streamline the build process, ensuring consistency and efficiency. Static code analysis can be performed at this stage to identify potential issues early on.
- Test: Testing is a critical stage to verify that the software meets the specified requirements and functions as expected. Various testing techniques, such as unit testing, integration testing, and system testing, are employed to identify and address defects. Automation tools can be used to accelerate testing and improve coverage.
- Release: The release stage marks the transition of the software from development to production. Release management processes are followed to ensure a smooth and controlled deployment. This may involve creating release packages, updating documentation, and notifying stakeholders.
- Deploy: The deployment stage involves installing the software on production servers. Automation tools can be used to automate the deployment process, reducing the risk of human error and ensuring consistency. Configuration management practices are essential to maintain a consistent environment across different systems.
- Operate: Once deployed, the software enters the operations stage. Teams are responsible for monitoring the application's performance, addressing issues, and providing ongoing support. Infrastructure management and maintenance are also crucial aspects of this stage.

- Monitor: Continuous monitoring is essential to ensure the software's performance, reliability, and security. Monitoring tools are used to track key metrics, detect anomalies, and proactively address issues. Feedback from monitoring is used to inform future improvements and optimizations.

By automating repetitive tasks and implementing continuous integration and continuous delivery (CI/CD) pipelines, DevOps teams can rapidly deploy new features and updates to production. This not only improves customer satisfaction but also allows organizations to respond more quickly to market changes and competitive pressures. Additionally, DevOps can help organizations improve software quality by incorporating testing and feedback loops into the development process, ensuring that only high-quality code is released.

While DevOps offers numerous advantages, it is not devoid of challenges. Implementing DevOps requires a significant cultural shift within organizations, as it necessitates a willingness to embrace new tools, processes, and ways of working. Additionally, the integration of cloud computing into the DevOps pipeline can introduce new complexities and challenges. Overcoming these challenges often involves a combination of training, education, and a gradual adoption of DevOps practices, along with careful consideration of cloud-specific factors such as security, scalability, and cost management.

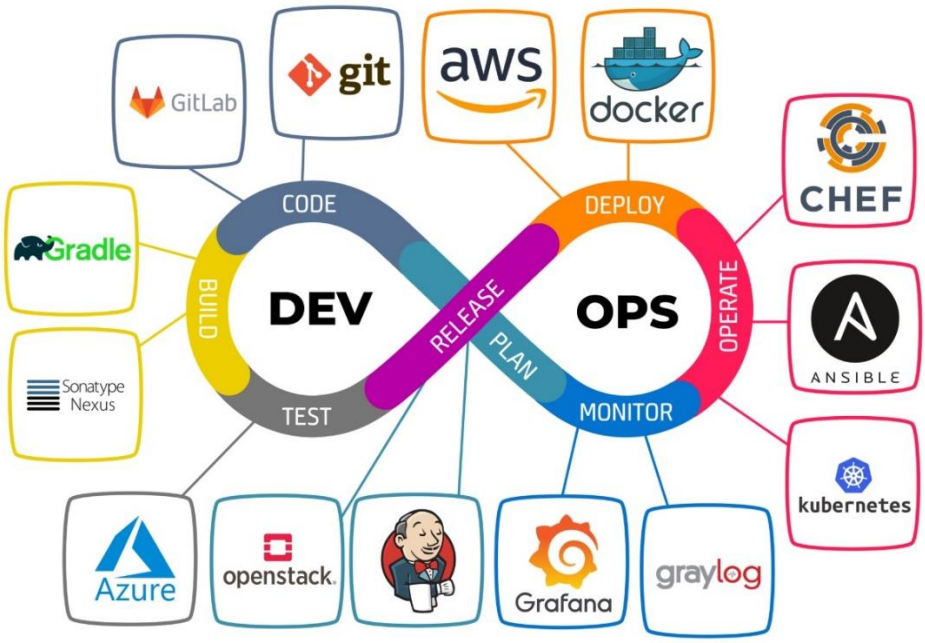

Figure 4: DevOps Lifecycle. Source: cloud.z.com

2.1. Terraform

Terraform, a popular open-source infrastructure as code (IaC) tool, has revolutionized the way IT professionals manage and deploy cloud resources. By using a declarative language, Terraform allows users to define the desired state of their infrastructure, and the tool automatically creates, updates, or destroys resources to match that state. This approach significantly improves efficiency, consistency, and reproducibility in infrastructure management.

One of the key benefits of Terraform is its ability to handle infrastructure across multiple cloud providers, including Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP), and many others. This portability enables organizations to avoid vendor lock-in and leverage the best features of different cloud platforms. Terraform's modular design allows users to create reusable modules, promoting code reusability and reducing the likelihood of errors.

Terraform's declarative syntax, written in the HashiCorp Configuration Language (HCL), is easy to learn and understand. It provides a human-readable format that is both concise and expressive. This makes it easier for teams to collaborate on infrastructure projects and ensure consistency in their approach. Additionally, Terraform's state management capabilities allow users to track the current state of their infrastructure, making it easier to troubleshoot issues and roll back changes if necessary. Another advantage of Terraform is its integration with version control systems like Git. This enables teams to track changes to their infrastructure code, collaborate effectively, and easily revert to previous versions if needed. Version control also provides a reliable audit trail, which is essential for compliance and regulatory purposes.

Terraform has become an indispensable tool for infrastructure management, offering numerous benefits such as portability, reusability, ease of use, and integration with version control. By adopting Terraform, organizations can improve their infrastructure efficiency, reduce the risk of errors, and enhance their ability to respond to changing business needs. As the cloud computing landscape continues to evolve, Terraform is likely to play an even more critical role in helping organizations manage their infrastructure effectively.

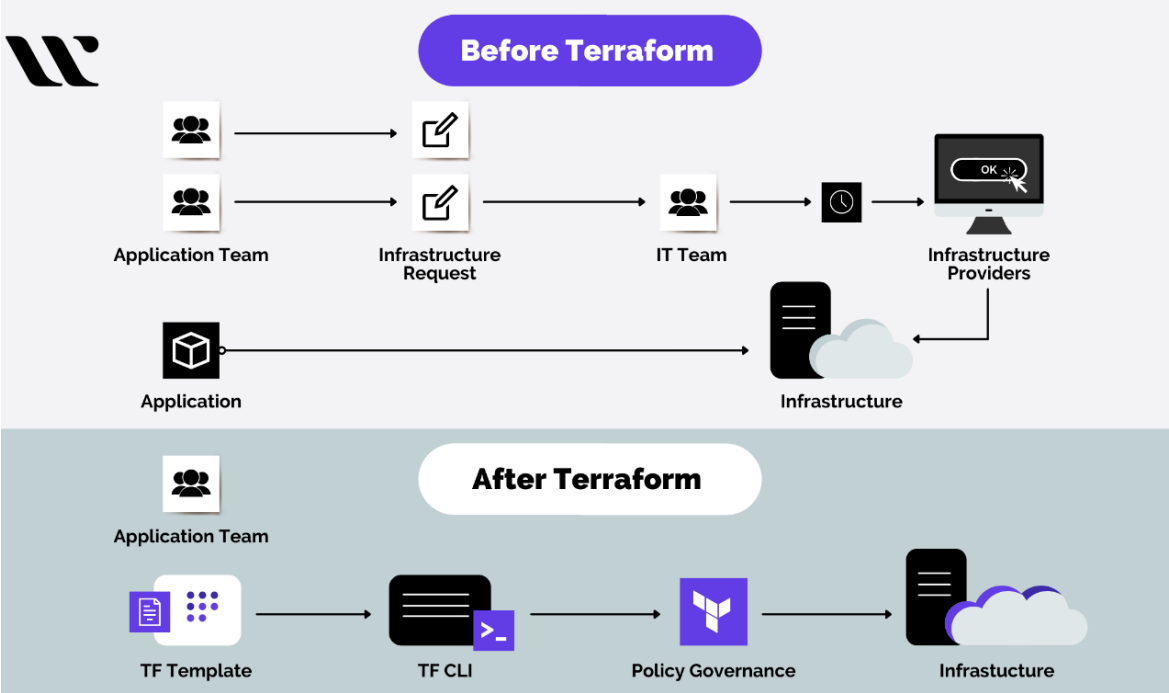

Figure 5: Operation of Terraform. Source: whizlabs.com

2.2. GitLab

GitLab is a comprehensive software development lifecycle (SDLC) platform that has established itself as a premier choice for organizations seeking efficient and collaborative development practices. By integrating source code management, continuous integration/continuous delivery (CI/CD), and project management tools, GitLab offers a unified environment that streamlines the entire software development process. GitLab offers a centralized platform for managing source code, utilizing the Git version control system. It facilitates collaboration among teams by providing features like branching, merging, and pull requests, ensuring efficient tracking, review, and integration of code changes. This approach enhances code quality and minimizes the likelihood of conflicts.

Beyond source code management, GitLab provides a comprehensive suite of CI/CD tools that automate the building, testing, and deployment of software. By integrating with various programming languages and frameworks, GitLab can automatically trigger pipelines whenever code changes are pushed to the repository. This enables teams to continuously test and deploy their applications, ensuring that they are always in a releasable state. Furthermore, GitLab's CI/CD features support parallel execution and caching, accelerating the build and test process.

GitLab also serves as a comprehensive platform for project management, development, and collaboration. Its integrated features, including issue tracking, task assignment, Kanban boards, and agile methodologies, streamline project planning and execution. By encouraging code reviews, issue discussions, and shared dashboards, GitLab fosters a collaborative environment that improves code quality and accelerates development. Its open-source nature and active community provide organizations with a robust ecosystem of resources and support.

To summarize, GitLab is a powerful and versatile platform that can significantly enhance the software development process. GitLab provides a comprehensive solution for teams of all sizes. Its focus on collaboration, efficiency, and quality makes it an invaluable tool for organizations seeking to deliver high-quality software products. As the software development landscape continues to evolve, GitLab's innovative features and commitment to open-source development position it as a leading platform for the future.

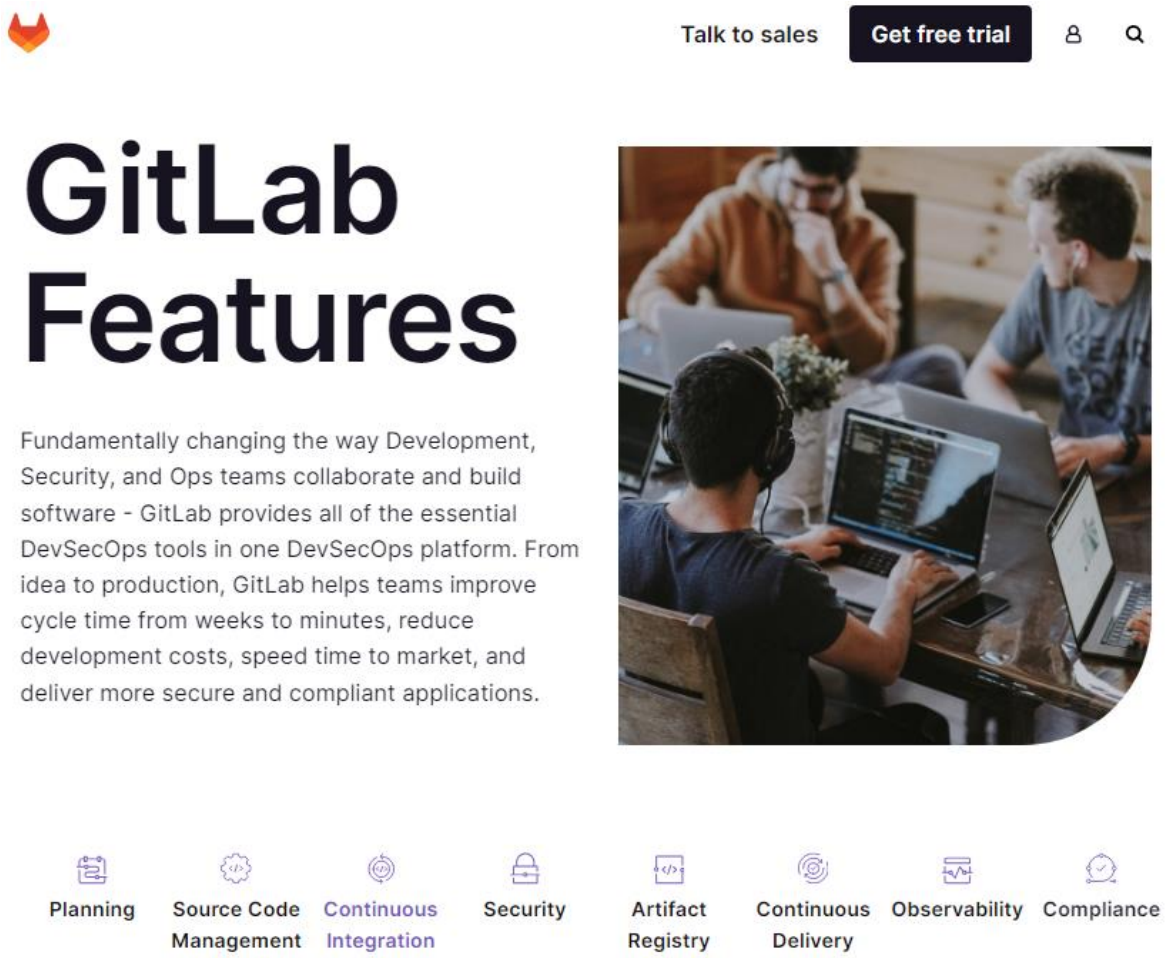

Figure 6: GitLab features. Source: GitLab

CHAPTER 3: SOLUTION DESIGN

1. Conceptual design

Cloud platforms like AWS, Azure, GCP,… can offer a vast array of services and resources, providing immense value to organizations of all scales. However, managing these platforms manually can lead to time-consuming tasks, errors and excessive resource consumption. To address these challenges, automated management solutions have been introduced.

Automated deployment with IaC and CI/CD tools can streamlines the process of delivering software updates and changes to production environments. By using Infrastructure as Code (IaC) tools like Terraform or Ansible, infrastructure can be defined and managed as code, ensuring consistency and reproducibility. While CI/CD pipelines, using tools like Jenkins or GitLab CI/CD, can automate the testing, building, and deployment of code.

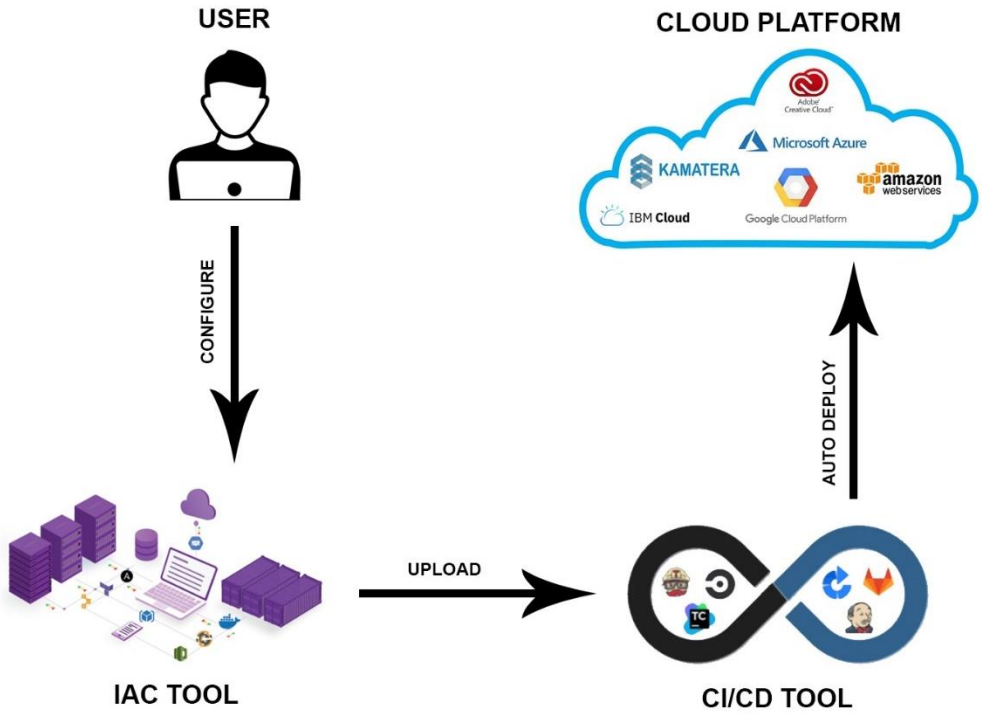

Figure 7: Cloud infrastructure automation conceptual design

2. Detailed design

In this topic, we will delve into the practice of using Terraform and GitLab CI/CD to streamline and automate the deployment process of applications to AWS. Terraform is a popular Infrastructure as Code (IaC) tool that allows us to define and manage AWS resources using a declarative language named HCL, making it easier to version control and reproduce our infrastructure. While GitLab CI/CD provides a robust framework for automating the build, test, and deployment pipelines.

With Terraform and GitLab CI/CD, we can build a flexible deployment workflow that automates most steps, but also offers manual options for specific tasks. We will start by defining our AWS infrastructure using Terraform modules. Then we will integrate this configuration into a GitLab CI/CD pipeline, where it will be automatically built, tested, and deployed to AWS. GitLab CI/CD offers various features such as environment variables, artifacts, and pipelines to customize the deployment process and ensure quality control.

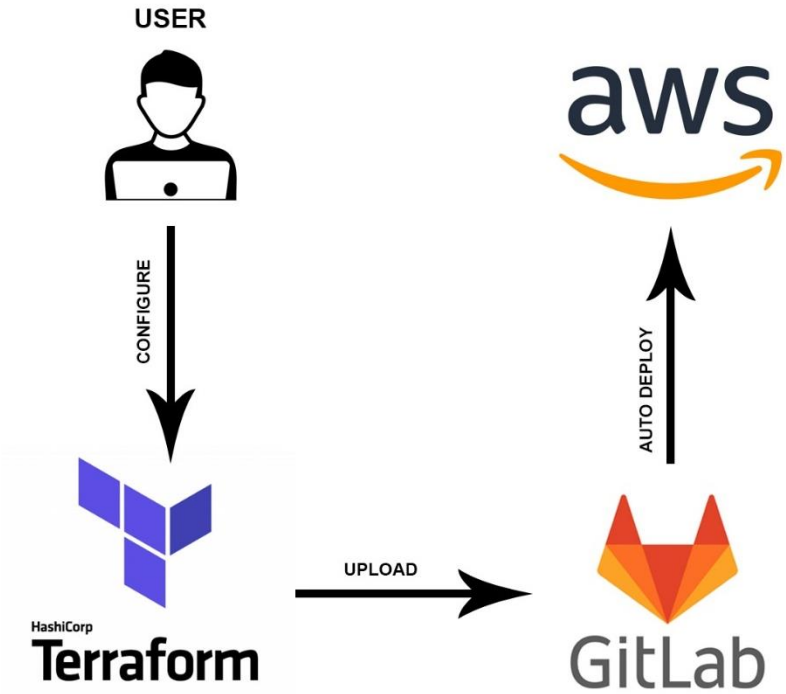

Figure 8: Cloud infrastructure detailed design

CHAPTER 4: SOLUTION IMPLEMENTATION

1. AWS Provisioning

First of all, to use AWS services, we need to have an active AWS account. If you don't already have one, you can create a free tier account at https://aws.amazon.com. This account provides us with a certain amount of resources to explore and experiment with AWS services at no cost.

However, we still need to set up some additional configurations. This typically involves establishing trust relationships between our on-premises infrastructure or other cloud platforms and our AWS account. These trust relationships allow us to securely manage and access resources across different environments using temporary security credentials.

1.1. AWS IAM

AWS IAM (Identity and Access Management) is a fundamental service within the AWS ecosystem that allows us to securely manage access to our AWS resources. IAM will provides a central point for controlling who can access our AWS account and what they can do. Through IAM, we can create users, groups, roles and assign them specific permissions to control which AWS services and resources they can access. This granular control helps us implement the principle of least privilege, ensuring that users only have the permissions necessary to perform their jobs.

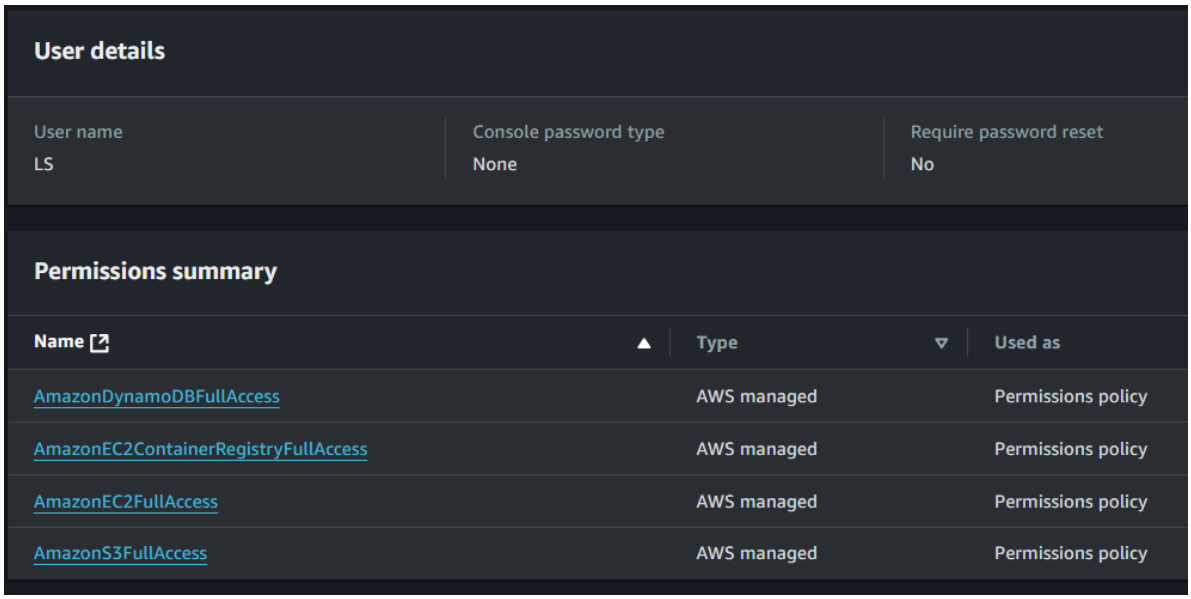

Figure 9: AWS IAM user

1.2. AWS S3

AWS S3 (Simple Storage Service) is a highly scalable and cost-effective object storage service that can be used for backend of IaC. By storing configuration files, scripts, and templates in S3 buckets, we can centralize our IaC artifacts, making them easily accessible and manageable. S3 can serves many robust security features, including versioning and access control lists, ensure the integrity and confidentiality of our IaC data.

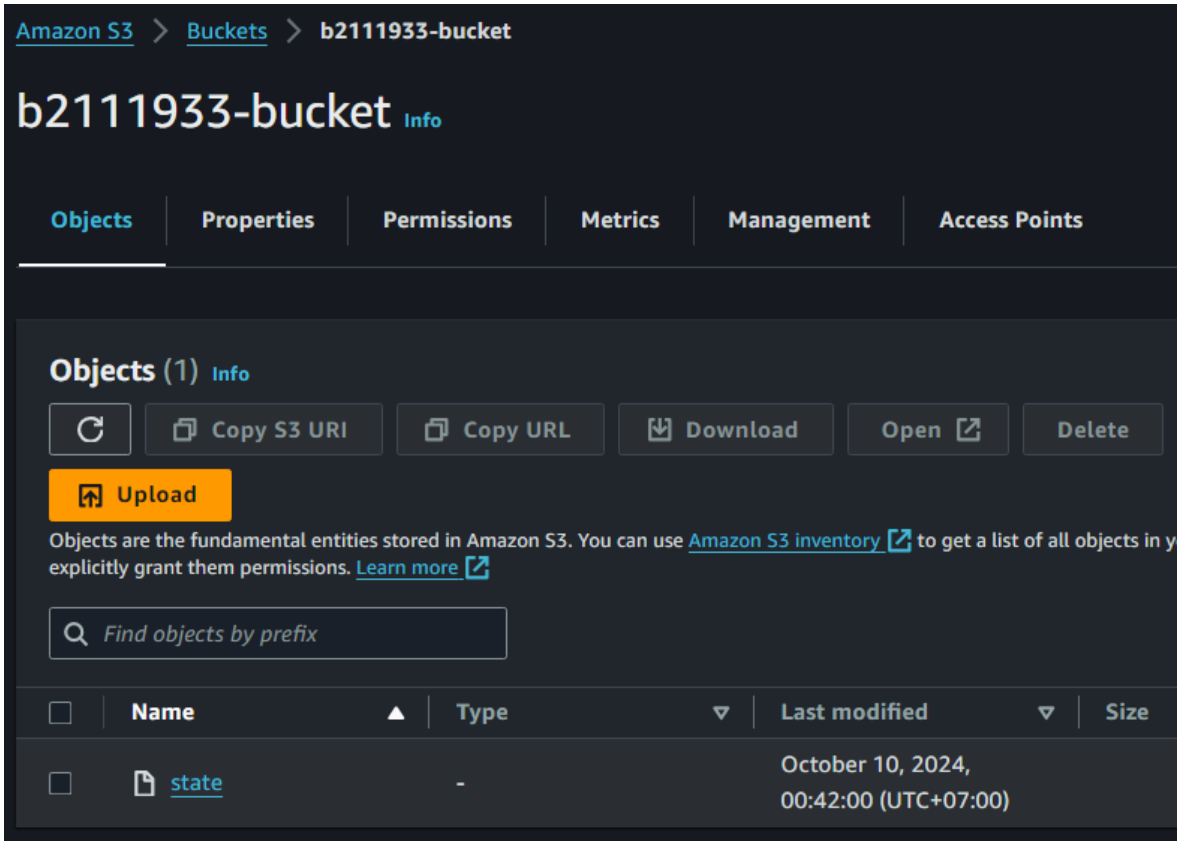

Figure 10: AWS S3 bucket

1.3. AWS DynamoDB

AWS DynamoDB is a fully managed NoSQL database which often paired with S3 for robust and scalable backend solutions in IaC. Its key-value model and flexible schema can simplify data access and development. To ensure consistency and avoid conflicts in Terraform-managed infrastructure, employing the locking mechanism in Terraform is crucial. By acquiring a lock on the state file, Terraform can prevent multiple users from making concurrent changes to the same infrastructure resources, safeguarding data integrity and avoiding unintended modifications.

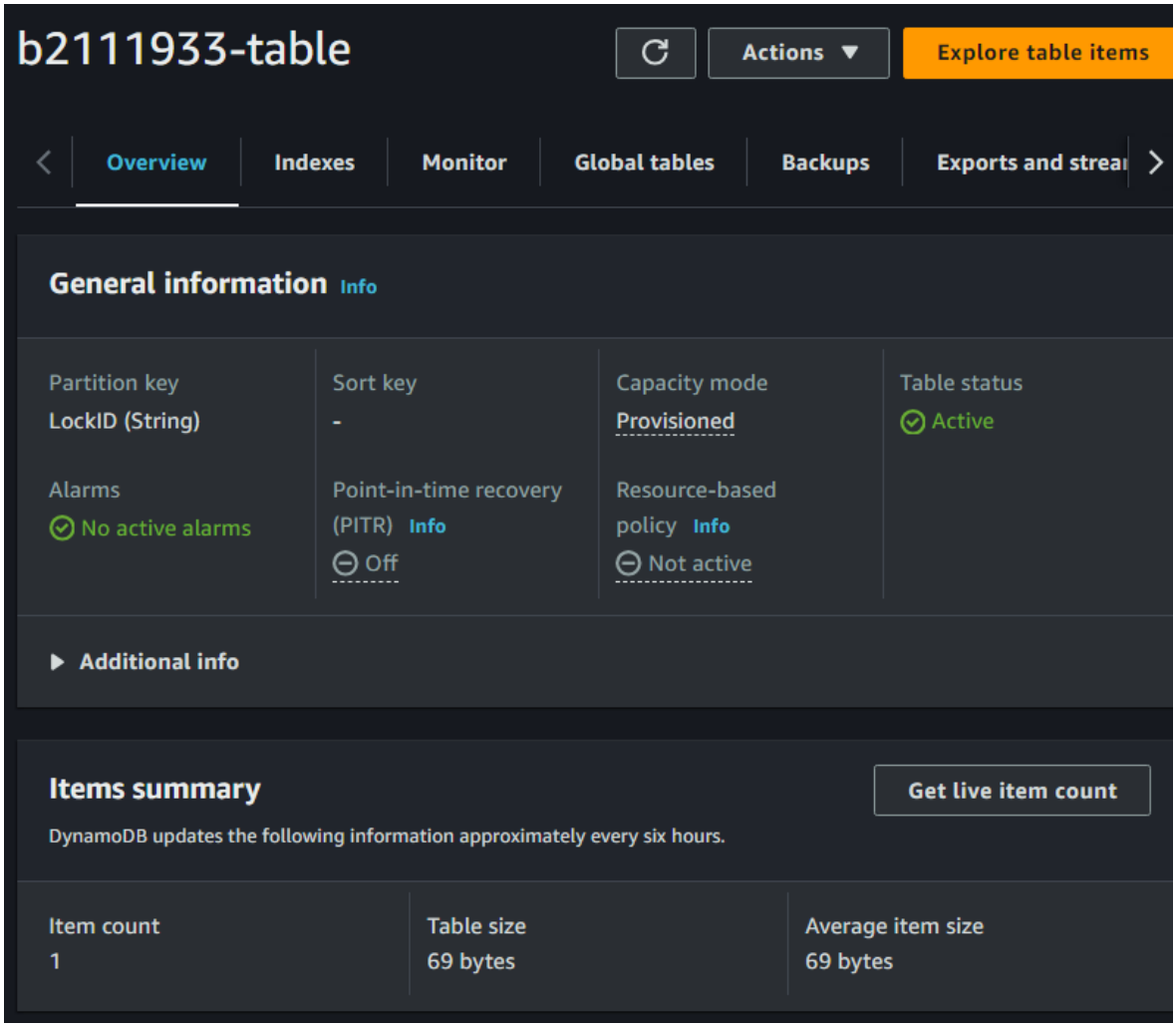

Figure 11: AWS DynamoDB table

1.4. AWS security credentials

AWS security credentials are essential for authenticating users and applications to access AWS services. These credentials typically consist of an access key ID and a secret access key. The root account can has its own root access key pair. However, this root access key should be treated with extreme caution as it grants full administrative privileges to our AWS account. As we have metioned above, it is recommended to use IAM users with limited permissions for specific tasks rather than relying solely on the root account. IAM users can also have their own access key pairs, which can be rotated regularly to mitigate the risk of unauthorized access.

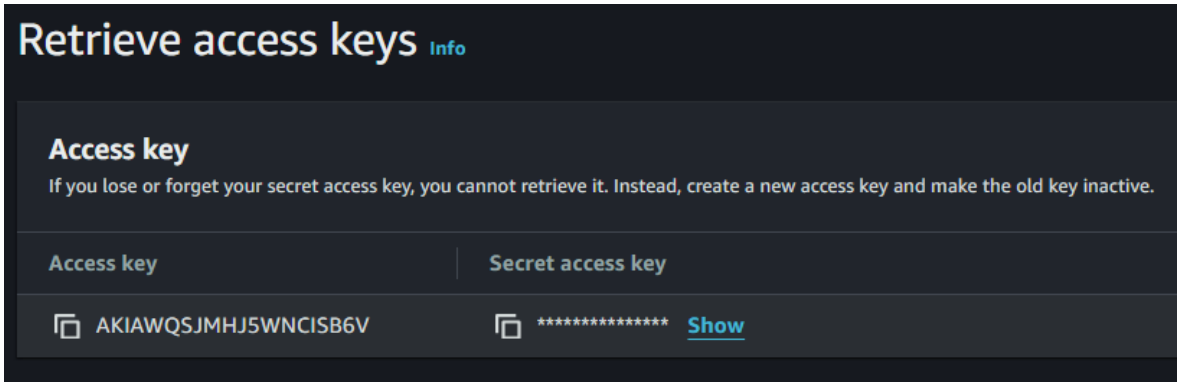

Figure 12: AWS access keys

1.5. AWS key pairs

Each AWS key pair is a pair of cryptographic keys used for secure access Amazon EC2 instances, consists of a public key and a private key. The public key is stored on the EC2 instance and used to encrypt data, while the private key is stored locally by ourselves and we can use it to decrypt data. When we launch an EC2 instance, we can specify a key pair, allowing us to securely connect to that instance through SSH [9] for Linux instances or RDP [10] for Windows instances. It is crucial to keep our private key secure as it grants access to our instances.

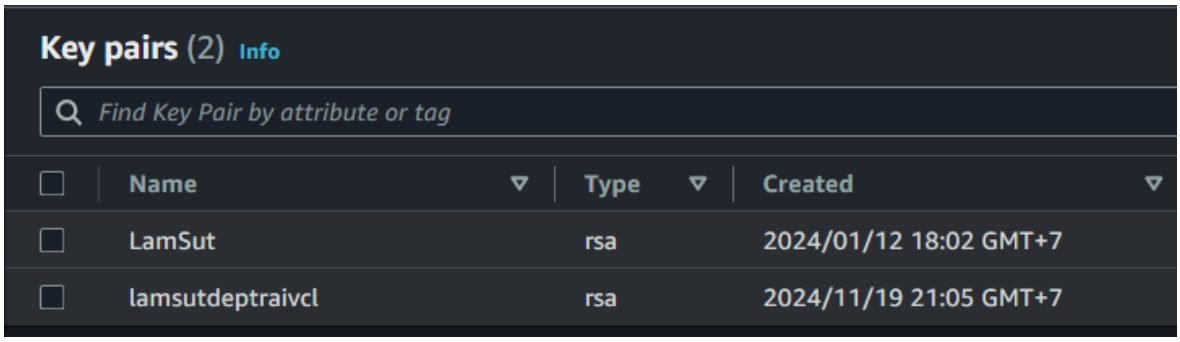

Figure 13: AWS key pairs

1.6. AWS CLI

AWS CLI (Command Line Interface) is a versatile tool for interacting with AWS services programmatically. AWS CLI provides a simple and flexible way to manage and automate various tasks including creating, modifying and deleting AWS resources. With AWS CLI, we can execute commands directly from our terminal or script them for automated workflows. AWS CLI supports a wide range of AWS services, making it a valuable tool for those who work with AWS resources. We can follow the guide for installing and configuring the AWS CLI by visiting the official documentation.

2. Terraform for AWS deployment

Once we have established the foundational AWS resources required for our infrastructure, we can proceed to configure Infrastructure as Code (IaC) on our local environment. This typically involves installing a suitable tool, such as Terraform or AWS CloudFormation, and configuring it with our AWS credentials. Then we can create configuration files that define our infrastructure's desired state using the tool's syntax. These files will serve as blueprints for provisioning and managing AWS resources. By setting up IaC locally, we can iterate on our infrastructure design, test changes, and ensure consistency before deploying them to our AWS environment.

2.1. Terraform installation

We need to install Terraform for beginning, a great resource to guide us through this process is the official Terraform documentation. This tutorial specifically guides us through installing Terraform for our operating system, whether it's Windows macOS, or Linux. Please follow these steps to ensure we have Terraform ready to manage our infrastructure.

```
lamb2111933@lamb2111933-VirtualBox:~/deployment-automation$ terraform version
Terraform v1.9.8
on linux_amd64
+ provider registry.terraform.io/hashicorp/aws v5.65.0
lamb2111933@lamb2111933-VirtualBox:~/deployment-automation$
```

Figure 14: Terraform installed

2.2. Terraform configuration

Virtual Private Cloud (VPC) is a logically isolated section of AWS cloud platform where users can launch AWS resources like instances, databases, and storage. VPC can serve as a virtual network within the public cloud, providing us with complete control over our virtual networking environment. With a VPC, we can define IP address range, configure subnets, route traffic, and establish security groups to control inbound and outbound traffic.

Amazon Elastic Compute Cloud (EC2) is a AWS service that allows users to launch virtual machines (VM) called EC2 instances that easily launch and terminate as required. These instances can provide scalable computing capacity on demand, enabling us to run various applications and workloads in the cloud. EC2 offers a wide range of instance types with different processing power, memory, storage, and networking capabilities to suit diverse needs.

In this topic, we will configure Terraform to provision three EC2 instances within an AWS VPC environment. These instances will run on three different operating systems, catering to a wide range of deployment scenarios, including : one Windows, one Amazon Linux and one Ubuntu. To enable remote access to the Linux instances, SSH keys will be configured. For the Windows instance, Remote Desktop Protocol (RDP) will be set up instead. This is because SSH connections are primarily designed for Linux and Unix-like systems, while RDP is the standard protocol for remote access to Windows systems.

Additionally, to facilitate internet access, we will have to deploy those instances in a public subnet within our VPC. This subnet will be associated with an Internet Gateway that allows communication between our VPC and the public internet. Then we will create a route table that directs traffic destined for the internet through the Internet Gateway. By associate this route table with the public subnet, we can allow our instances to be able to communicate with internet. Beside, to ensure the security of our network traffic, we have to configure security rules to restrict inbound and outbound traffic, allowing only necessary connections such as SSH or HTTPS.

About the configuration process, first of all, we have to define a `vpc` module that create a VPC, a subnet, and a security group. Next, we will create an `ec2` module that will utilize the resources defined in the VPC module. Then we will create a main configuration file that references all those module. It will be used to trigger Terraform commands that deploy the VPC and those EC2 instances in a specified AWS region. With this modular approach, we can promote code reusability and maintainability by breaking down the infrastructure into smaller, manageable modules. It also helps to improve collaboration and teamwork by allowing different team members to work on different modules simultaneously.

About our infrastructure state storage, we will use an S3 bucket as the backend of Terraform, which will store the state file. This state file is a JSON document that records the configuration of our infrastructure and the current state of our resources. By using an S3 bucket, we ensure that the state file is stored remotely and can be accessed by multiple users or teams. Additionally, we can configure Terraform to use DynamoDB as a locking mechanism to prevent concurrent modifications of the state file. This helps maintain consistency and avoid conflicts when multiple users are working on the same infrastructure.

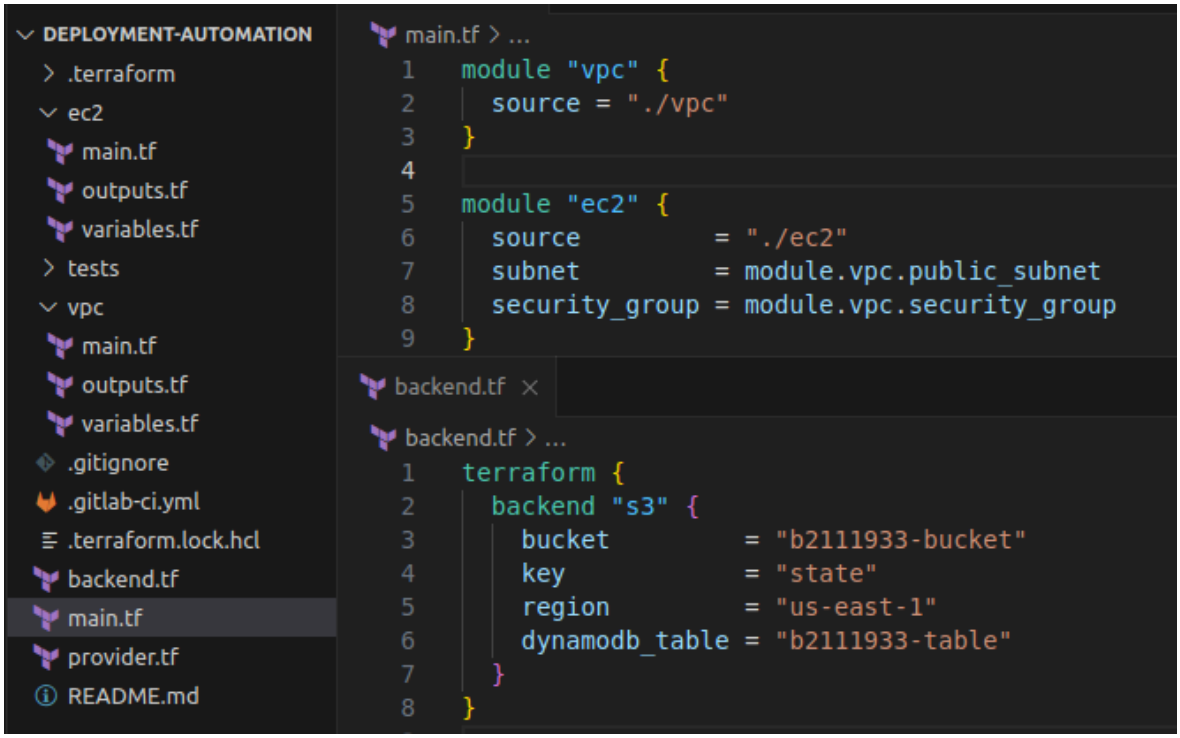

Figure 15: Terraform configuration

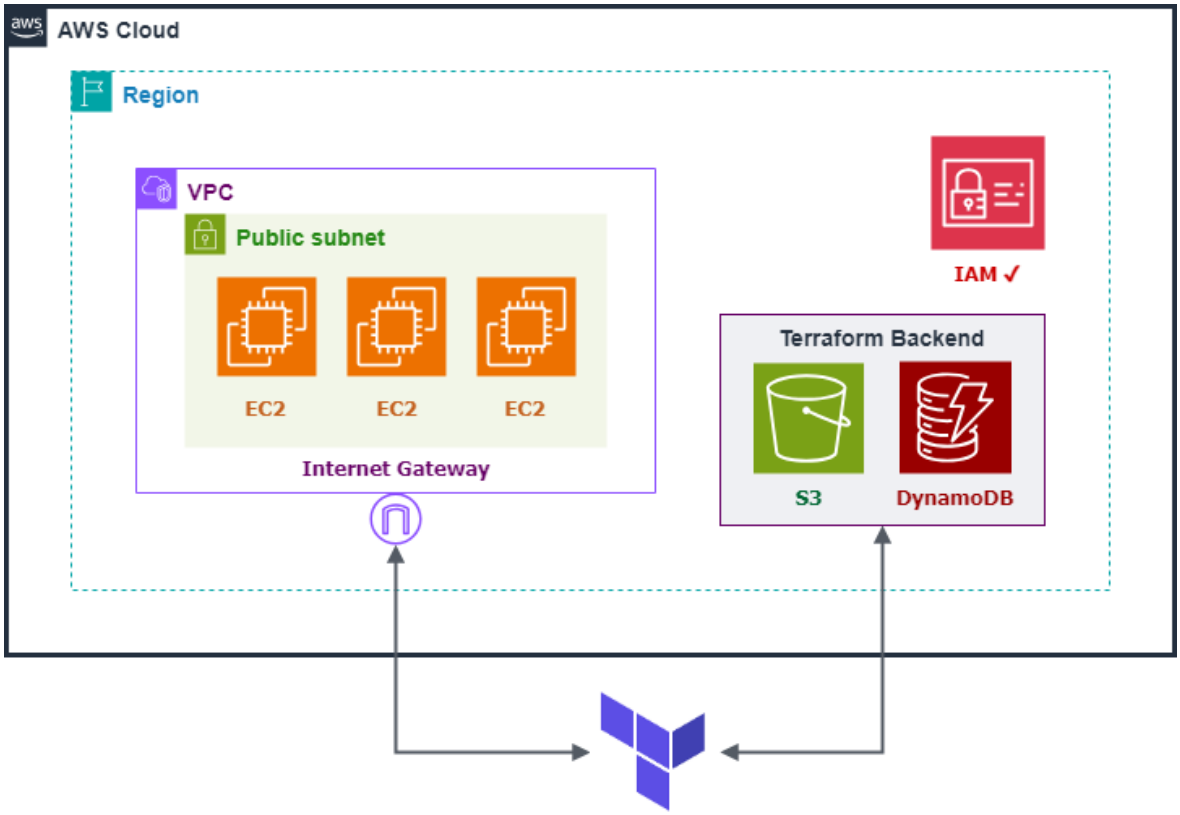

Figure 16: AWS resources

2.3. Terraform tests

We can test Terraform configurations using `tftest.hcl` files. These files define test cases that can validate each of Terraform modules and their outputs, allow us to simulate different scenarios with testing resources. We can also verify the accuracy of output values and check for compliance with best practices. By writing unit tests, we can ensure that our infrastructure code is reliable and maintainable. This proactive approach helps us detect issues early in the development cycle, leading to more robust and dependable infrastructure deployments.

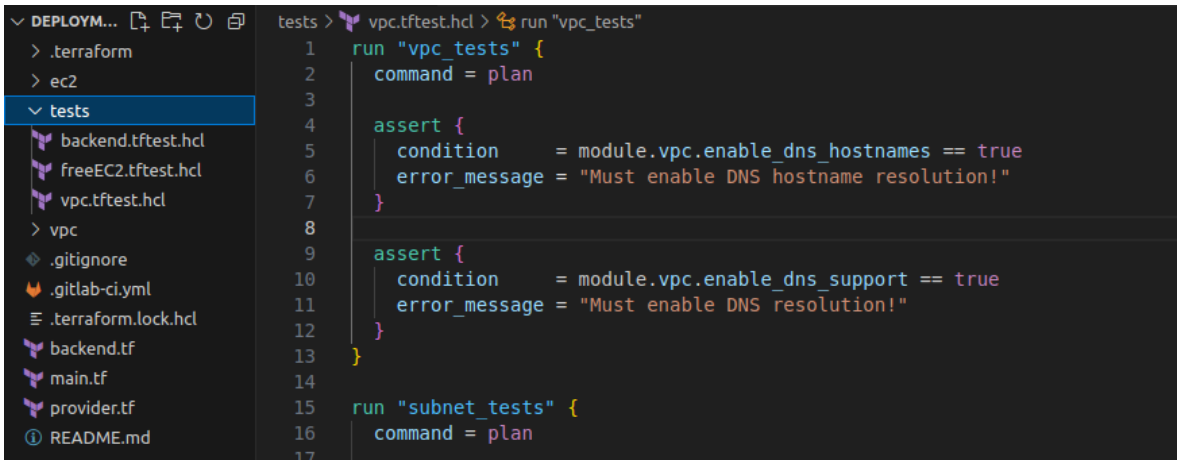

Figure 17: Terraform unit tests

3. GitLab for Terraform CI/CD

Terraform CI/CD can be seamlessly integrated with GitLab, a popular DevOps platform, to streamline the infrastructure as code (IaC) workflow. By configuring GitLab CI/CD pipelines, we can automate tasks such as code validation, testing, building, and deploying Terraform modules. This integration promotes collaboration, reduces development time, and improves the dependability of the infrastructure.

3.1. GitLab project preparation

After finalizing your Terraform code, the next step is to integrate it into a CI/CD pipeline for automated provisioning. To achieve this, we will need to upload our code to a GitLab repository. However, GitLab CI/CD relies on a YAML file, typically named `.gitlab-ci.yml`, to define the build and deployment process. This file specifies the actions to be taken on code pushes and automates tasks like infrastructure provisioning using Terraform. To get started with GitLab repositories and YAML configuration, you can refer to their documentation.

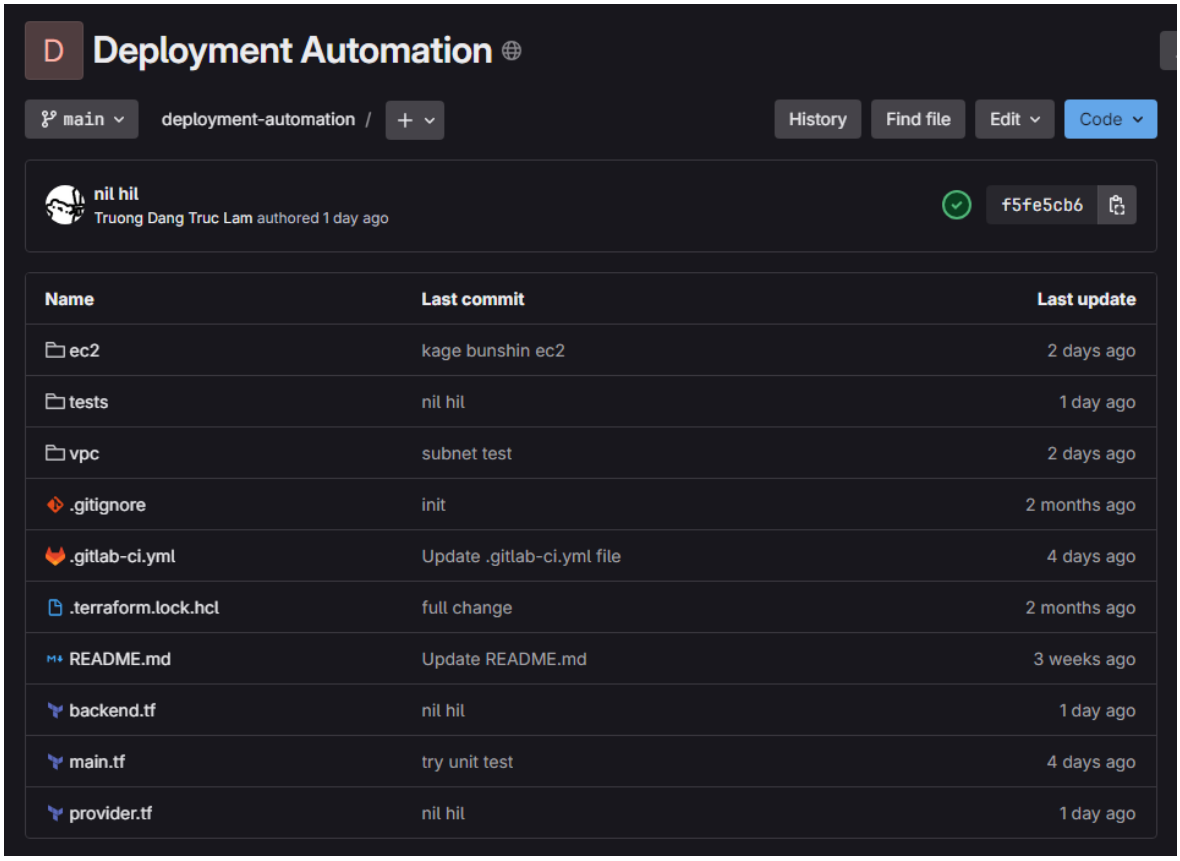

Figure 18: GitLab repository

Once we have set up a GitLab repository for our Terraform code and established a `.gitlab-ci.yml` file, the next critical step is to define variables within the YAML file. These variables store sensitive information such as API keys, passwords and other credentials, which should not be coded directly into our YAML file. By using variables, we can securely manage and reference secret data within our pipeline.

In this setup, we need to define variables including AWS access key, AWS secret key and GitLab access token. These credentials are essential for interacting with AWS services and GitLab. The AWS access key and secret key provide authentication and authorization for accessing AWS resources, while the GitLab access token allows the pipeline to interact with GitLab's API, such as triggering builds or accessing repository information.

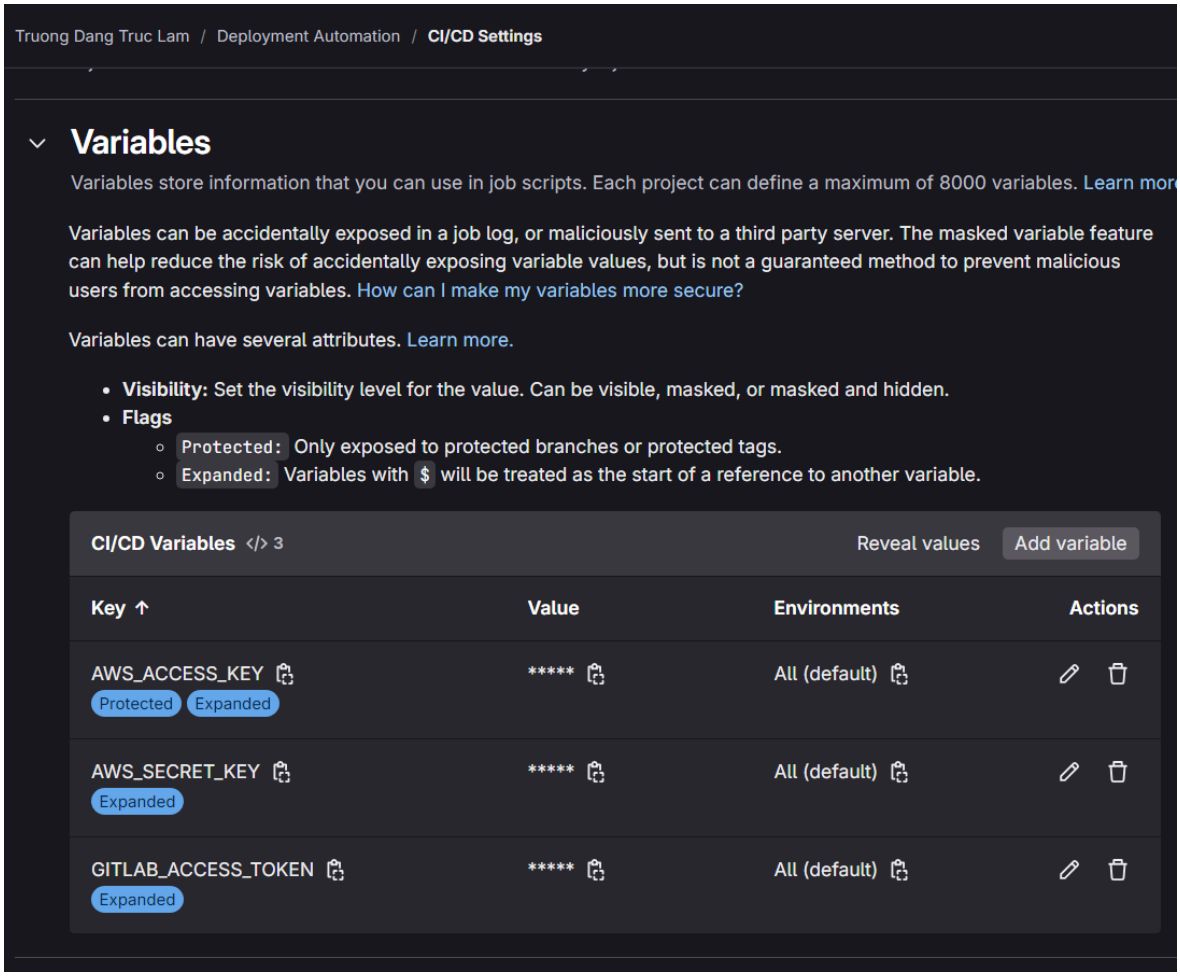

Figure 19: Define variables for GitLab CI/CD

3.2. GitLab CI/CD pipeline

A `.gitlab-ci.yml` file for Terraform code will provides a structured way to automate the building, testing, and deployment of our infrastructure. When using a Terraform image from the GitLab registry, we can leverage pre-configured environments that include Terraform CLI and necessary dependencies. The YAML file would define stages like build, test and deploy with corresponding jobs that execute specific tasks.

Our YAML file will configure a CI/CD pipeline for managing infrastructure using Terraform within GitLab CI/CD. It defines a pre-built Docker [11] image for running OpenTofu (an open-source alternative to Terraform). and sets environment variables for GitLab access token, AWS credentials, and the default AWS region. It also caches the .terraform directory for efficiency. The most important section is the stages definition which outlines a series of automated jobs in our CI/CD pipeline:

- validate: This stage runs `terraform validate` to check the syntax and structure of your Terraform configuration files.
- test: This stage allows us to run custom Terraform unit tests defined using the `terraform test` command.
- plan: This stage runs `terraform plan` to preview the changes that Terraform will make to our infrastructure based on our configuration. The output is saved to a file named "planfile".
- apply: This stage deploys the infrastructure changes identified in the plan by running `terraform apply` with the saved “planfile”.
- destroy: This stage destroys the infrastructure managed by our Terraform configuration with `terraform destroy`. It uses the `--auto-approve` flag to bypass further confirmation prompts.

Notably, applying and destroying require manual approval to prevent accidental changes. By defining these stages, we can ensure a controlled workflow for managing our infrastructure using Terraform in GitLab CI/CD pipeline.

.gitlab-ci.yml 1017 B

```
image:
  name: registry.gitlab.com/gitlab-org/terraform-images/releases-tofu/1.6
  entrypoint:
    - '/usr/bin/env'
    - 'PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin'

variables:
  TF_VAR_gitlab_token: ${GITLAB_ACCESS_TOKEN}
  AWS_ACCESS_KEY_ID: ${AWS_ACCESS_KEY}
  AWS_SECRET_ACCESS_KEY : ${AWS_SECRET_KEY}
  AWS_DEFAULT_REGION: "us-east-1"
```

Figure 20: The content of `.gitlab-ci.yml` file

CHAPTER 4: TEST AND EVALUATE

1. Test and evaluate AWS deployment with Terraform

To begin testing our Terraform code after configuration, we can start by opening a terminal window or command prompt then navigate to the directory where our Terraform code is located using the `cd` command. Then we use `terraform init` command to initialize our Terraform workspace and download any necessary providers, this will create a hidden `.terraform` directory in our project.

```
lamb2111933@lamb2111933-VirtualBox:~/deployment-automation$ terraform init
Initializing the backend...
Initializing modules...
Initializing provider plugins...
- Reusing previous version of hashicorp/aws from the dependency lock file
- Using previously-installed hashicorp/aws v5.65.0

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.
lamb2111933@lamb2111933-VirtualBox:~/deployment-automation$
```

Figure 21: Run command `terraform init` successfully

After initializing a project, it is crucial to run the `terraform validate` command to ensure the accuracy of our configuration files. This command performs a thorough syntax check, verifying that our code adheres to the correct syntax and structure of the Terraform language. It also examines the configuration for internal consistency, including the proper usage of attributes, values, and dependencies.

```
lamb2111933@lamb2111933-VirtualBox:~/deployment-automation$ terraform validate
Success! The configuration is valid.
```

Figure 22: Run command `terraform validate` successfully

Once we have validated Terraform configuration with `terraform validate`, the next logical step is to perform unit testing with the command `terraform test`. It will initiate a series of tests defined in `.tftest.hcl` files, allowing us to verify the behavior of individual resources and modules in isolation. We will test if the EC2 AMIs and instance types are free tier eligible. Additionally, we will validate the VPC, subnet and backend setup are properly configured.

```
lamb2111933@lamb2111933-VirtualBox:~/deployment-automation$ terraform test
tests/backend.tftest.hcl... in progress
  run "bucket_tests"... pass
  run "dynamodb_table_tests"... pass
tests/backend.tftest.hcl... tearing down
tests/backend.tftest.hcl... pass
tests/freeEC2.tftest.hcl... in progress
  run "server_1_tests"... pass
  run "server_2_tests"... pass
  run "server_3_tests"... pass
tests/freeEC2.tftest.hcl... tearing down
tests/freeEC2.tftest.hcl... pass
tests/vpc.tftest.hcl... in progress
  run "vpc_tests"... pass
  run "subnet_tests"... pass
tests/vpc.tftest.hcl... tearing down
tests/vpc.tftest.hcl... pass

Success! 7 passed, 0 failed.
lamb2111933@lamb2111933-VirtualBox:~/deployment-automation$
```

Figure 23: Run command `terraform test` successfully

After unit tests, we can run `terraform plan` command to generate a preview of the infrastructure changes that will be made when we apply our configuration. This command generates an execution plan, outlining the infrastructure changes required to bring the current state into alignment with the desired state defined in the Terraform configuration files. By reviewing this plan, we can assess the impact of the proposed changes, including resource creation, modification, or deletion.

```
lamb2111933@lamb2111933-VirtualBox:~/deployment-automation$ terraform plan
Acquiring state lock. This may take a few moments...

Terraform used the selected providers to generate the following execution plan. Resource
actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # module.ec2.aws_instance.server_1 will be created
  + resource "aws_instance" "server_1" {
      + ami                                  = "ami-06b21ccaeff8cd686"
      + 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)

Plan: 6 to add, 0 to change, 0 to destroy.

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to
take exactly these actions if you run "terraform apply" now.
Releasing state lock. This may take a few moments...
lamb2111933@lamb2111933-VirtualBox:~/deployment-automation$
```

Figure 24: Run command `terraform plan` successfully

With the execution plan thoroughly reviewed and approved, the final step in the Terraform workflow involves executing the `terraform apply` command. It will trigger the actual provisioning of infrastructure resources as outlined in the plan. By carefully monitoring the output of the `terraform apply` command, we can track the progress of the deployment and address any potential errors or warnings that may arise. Upon successful completion, the desired infrastructure will be provisioned, ready to serve its intended purpose.

```
lamb2111933@lamb2111933-VirtualBox:~/deployment-automation$ terraform apply
Acquiring state lock. This may take a few moments...

Terraform used the selected providers to generate the following execution plan. Resource
actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # module.ec2.aws_instance.server_1 will be created
  + resource "aws_instance" "server_1" {
      + ami                                  = "ami-06b21ccaeff8cd686"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = (known after apply)
      + availability_zone                    = (known after apply)
```

```
module.vpc.aws_route_table_association.public_subnet_route_table: Creating...
module.ec2.aws_instance.server_3: Creating...
module.ec2.aws_instance.server_2: Creating...
module.ec2.aws_instance.server_1: Creating...
module.vpc.aws_route_table_association.public_subnet_route_table: Creation complete after
 1s [id=rtbassoc-01b63342198f2d7cb]
module.ec2.aws_instance.server_3: Still creating... [10s elapsed]
module.ec2.aws_instance.server_2: Still creating... [10s elapsed]
module.ec2.aws_instance.server_1: Still creating... [10s elapsed]
module.ec2.aws_instance.server_1: Creation complete after 15s [id=i-02e3e3ea8c34e92bf]
module.ec2.aws_instance.server_3: Creation complete after 17s [id=i-0246196aadce2686d]
module.ec2.aws_instance.server_2: Creation complete after 17s [id=i-00b0ff112e90eb28c]
Releasing state lock. This may take a few moments...

Apply complete! Resources: 9 added, 0 changed, 0 destroyed.
lamb2111933@lamb2111933-VirtualBox:~/deployment-automation$
```

Figure 25: Run command `terraform apply` successfully

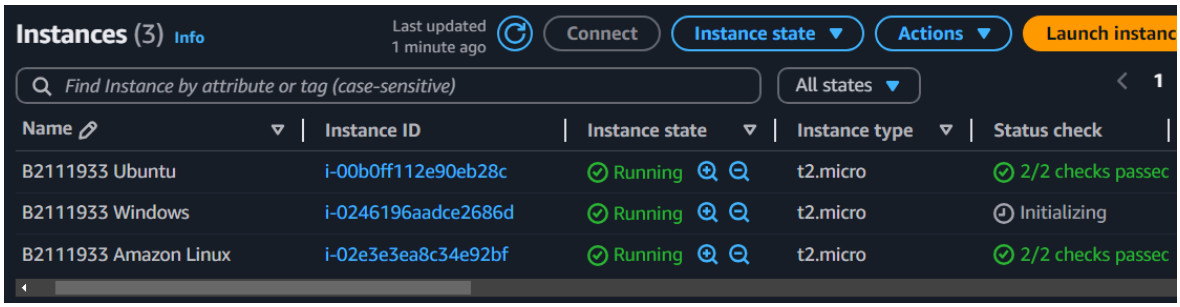

Figure 26: Verify result on AWS console after applying

Figure 27: Information of the Amazon Linux instance on AWS

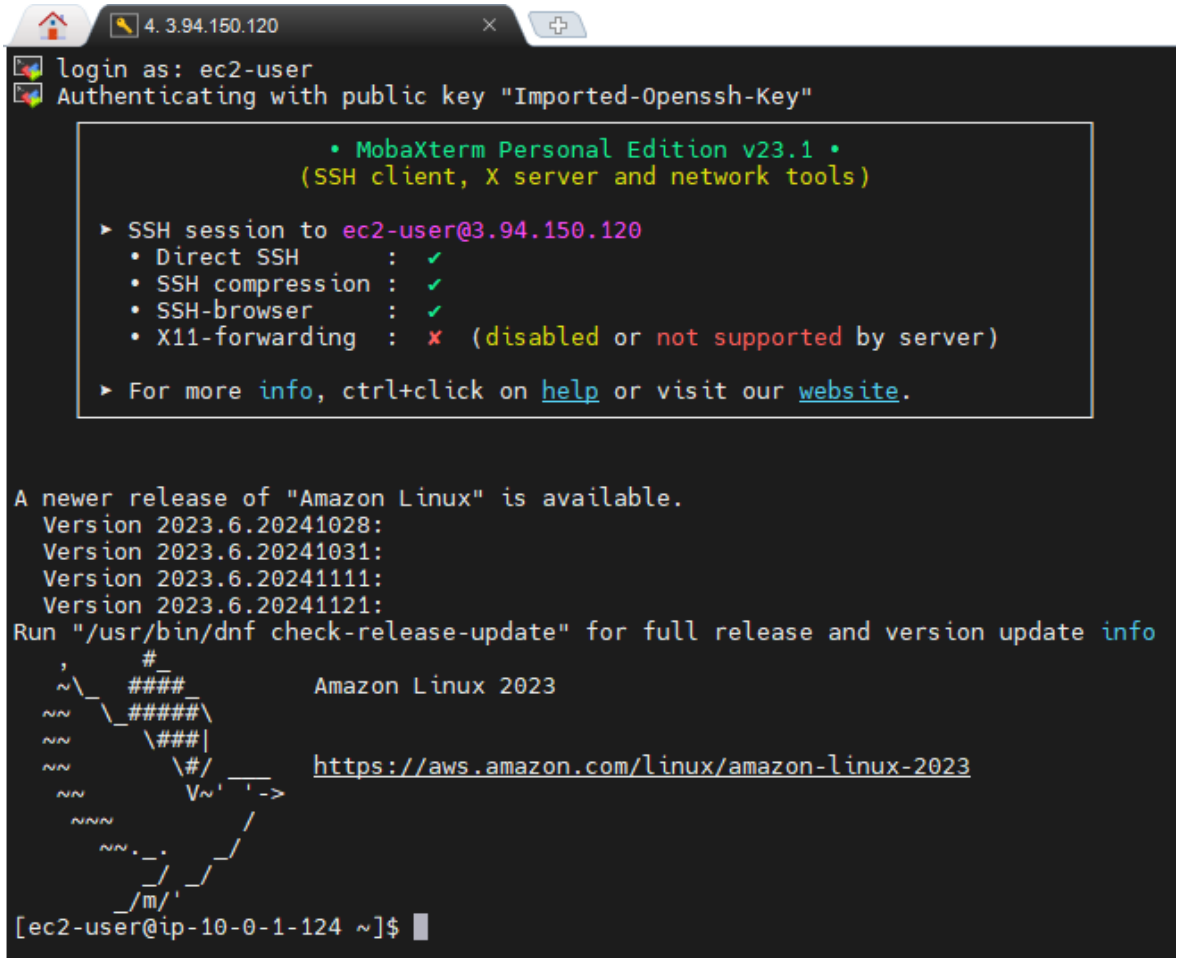

Figure 28: Connect to the Amazon Linux instance via SSH

Figure 29: Information of the Ubuntu instance on AWS

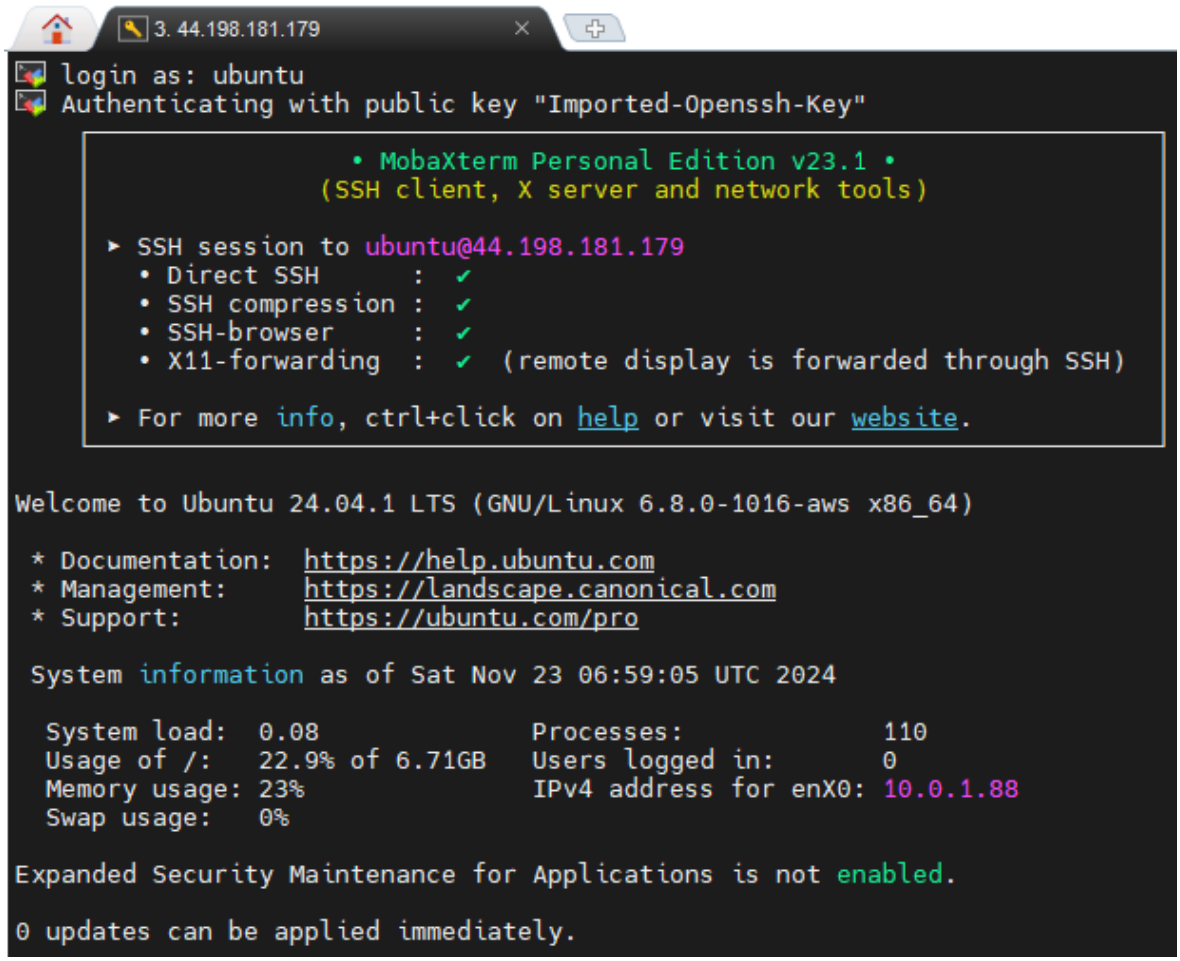

Figure 30: Connect to the Ubuntu instance via SSH

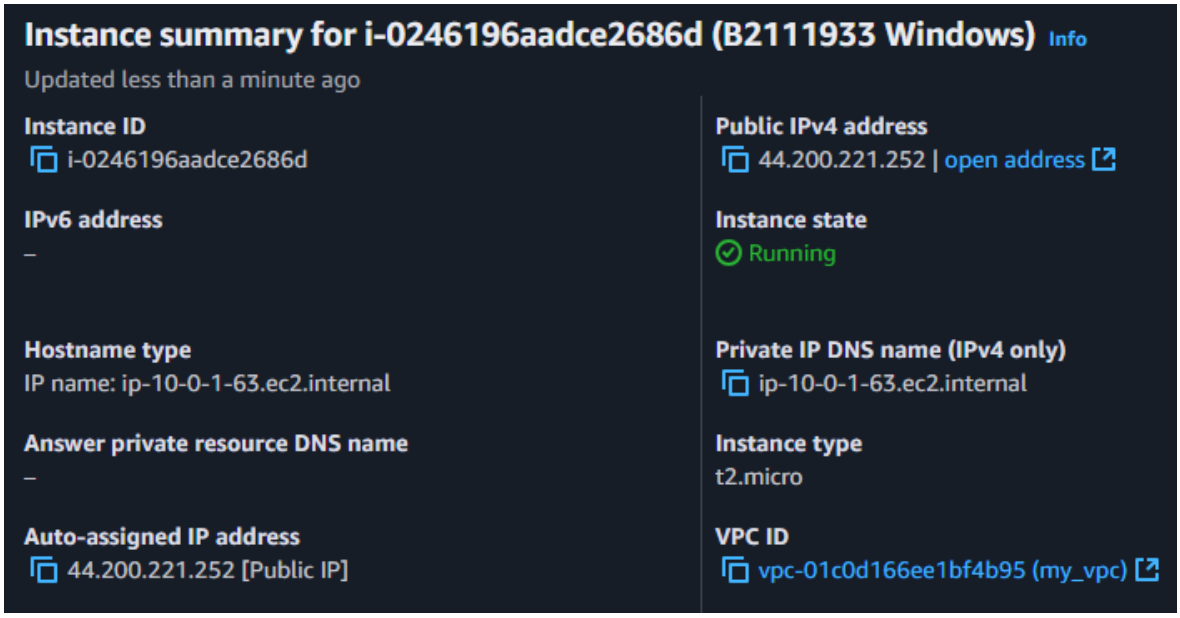

Figure 31: Information of the Windows instance on AWS

Figure 32: Connect to the Windows instance via RDP

By the way, to ensure a controlled removal of infrastructure, we can leverage the `terraform destroy` command. It systematically deletes the resources defined in the Terraform configuration, following the specified dependency order. By executing this command, we can decommission unwanted infrastructure and reclaim associated resources. It is essential to exercise caution during this process, as destroying resources is an irreversible action. To mitigate risks, it is highly recommended to review the generated destruction plan carefully before proceeding.

```
lamb2111933@lamb2111933-VirtualBox:~/deployment-automation$ terraform destroy
Acquiring state lock. This may take a few moments...
module.vpc.aws_vpc.my_vpc: Refreshing state... [id=vpc-01c0d166ee1bf4b95]
module.vpc.aws_internet_gateway.my_vpc_igw: Refreshing state... [id=igw-006b72d9e9660acc0
]
module.vpc.aws_subnet.public_subnet: Refreshing state... [id=subnet-06e1e4005b6d9cb1c]
module.vpc.aws_security_group.security_group: Refreshing state... [id=sg-005e60444b0d6bb5
2]
module.vpc.aws_route_table.public_subnet_route_table: Refreshing state... [id=rtb-0721423
77e50d6a2f]
module.ec2.aws_instance.server_1: Refreshing state... [id=i-02e3e3ea8c34e92bf]
module.ec2.aws_instance.server_3: Refreshing state... [id=i-0246196aadce2686d]
module.ec2.aws_instance.server_2: Refreshing state... [id=i-00b0ff112e90eb28c]
module.vpc.aws_route_table_association.public_subnet_route_table: Refreshing state... [id
=rtbassoc-01b63342198f2d7cb]

Terraform used the selected providers to generate the following execution plan. Resource
actions are indicated with the following symbols:
  - destroy

Terraform will perform the following actions:

  # module.ec2.aws_instance.server_1 will be destroyed
  - resource "aws_instance" "server_1" {
      - ami                                  = "ami-06b21ccaeff8cd686" -> null
      - arn                                  = "arn:aws:ec2:us-east-1:447904234107:instan
```

```
module.vpc.aws_security_group.security_group: Destroying... [id=sg-005e60444b0d6bb52]
module.vpc.aws_subnet.public_subnet: Destruction complete after 1s
module.vpc.aws_security_group.security_group: Destruction complete after 2s
module.vpc.aws_vpc.my_vpc: Destroying... [id=vpc-01c0d166ee1bf4b95]
module.vpc.aws_vpc.my_vpc: Destruction complete after 1s
Releasing state lock. This may take a few moments...

Destroy complete! Resources: 9 destroyed.
lamb2111933@lamb2111933-VirtualBox:~/deployment-automation$
```

Figure 33: Run command `terraform destroy` successfully

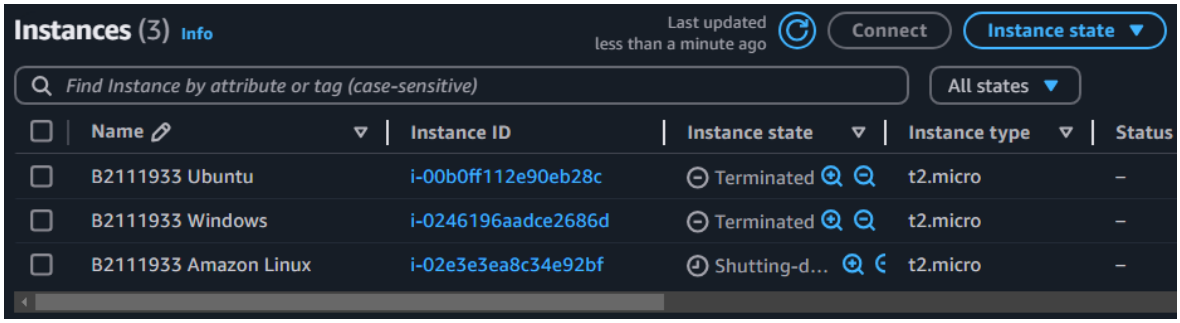

Figure 34: Verify result on AWS console after destroying

2. Test and evaluate CI/CD pipeline on GitLab

We can significantly enhance our infrastructure management workflow by implementing a GitLab CI/CD pipeline with Terraform code. Upon each Git commit, our pipeline is triggered, automatically initiating a comprehensive code validation, unit testing, and planning provision process for our infrastructure changes. This approach guarantees that our code aligns with Terraform best practices, preventing potential errors and inconsistencies. If the `validate`, `test`, and `plan` stages are successfully completed, we can proceed to apply the changes, ensuring that our infrastructure accurately reflects the desired state. This streamlined process not only saves time and effort but also significantly reduces the risk of human error, leading to a more reliable and efficient infrastructure.

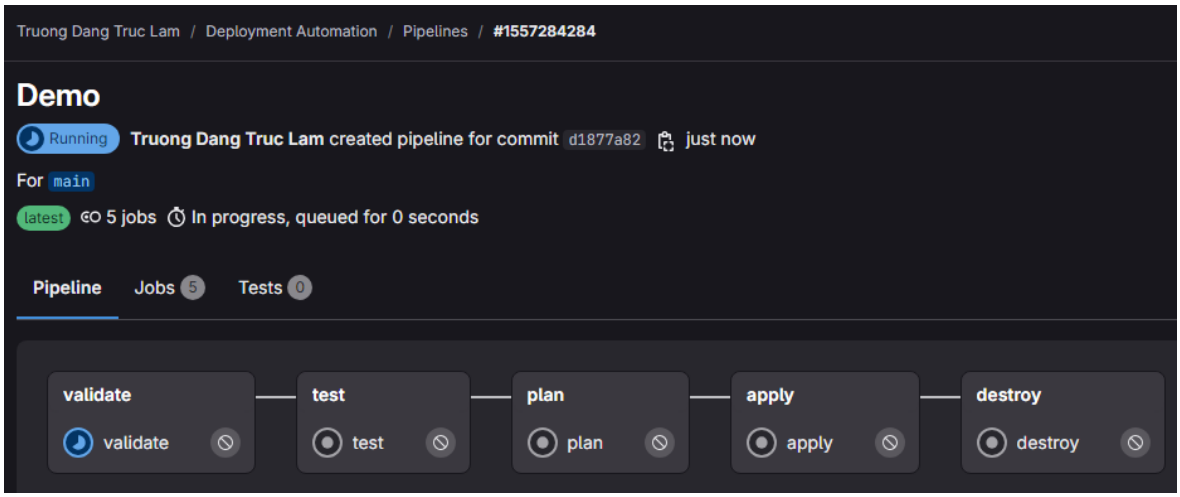

Figure 35: Automated run validation, testing and planning stages after a commit

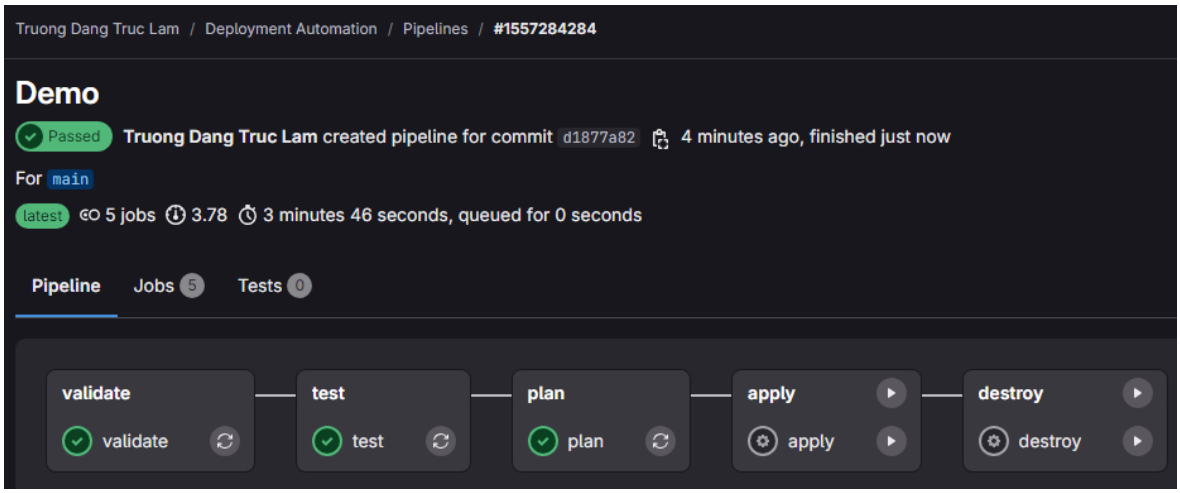

Figure 36: Automated stages on GitLab CI/CD passed successfully

However, the `apply` and `destroy` stages will remain manual, requiring explicit approval to execute these actions. This approach safeguards against unintended changes and provides a controlled environment for deploying and removing infrastructure resources on AWS.

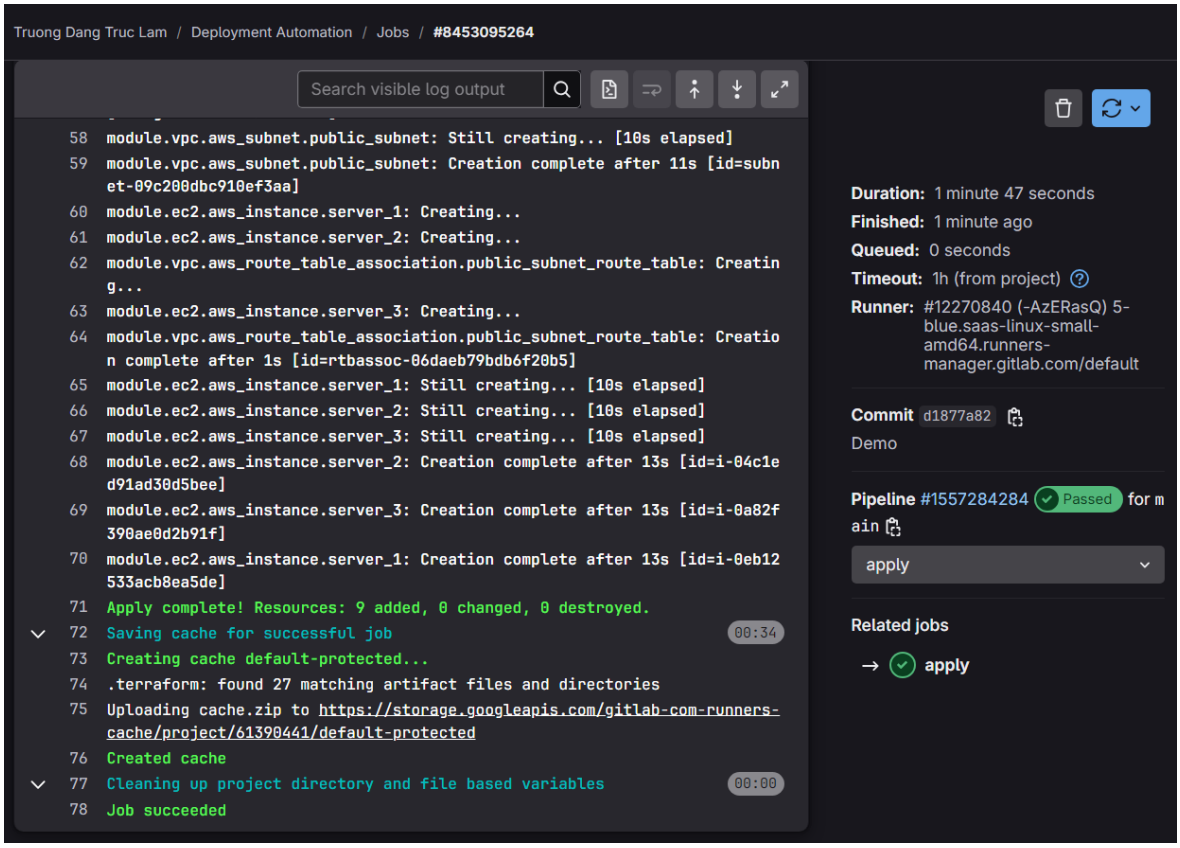

Figure 37: Manual applying with GitLab CI/CD passed successfully

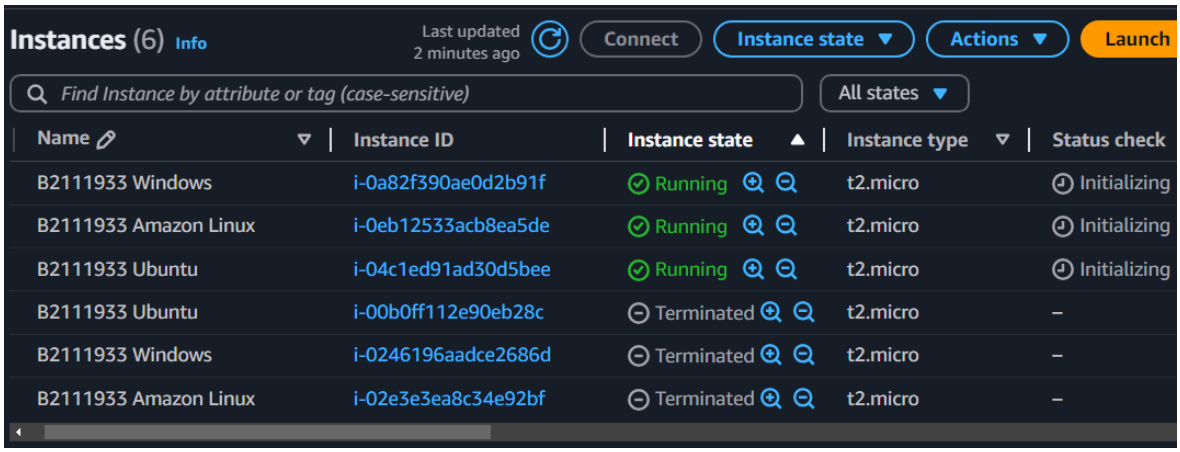

Figure 38: Verify result on AWS console after applying with GitLab CI/CD

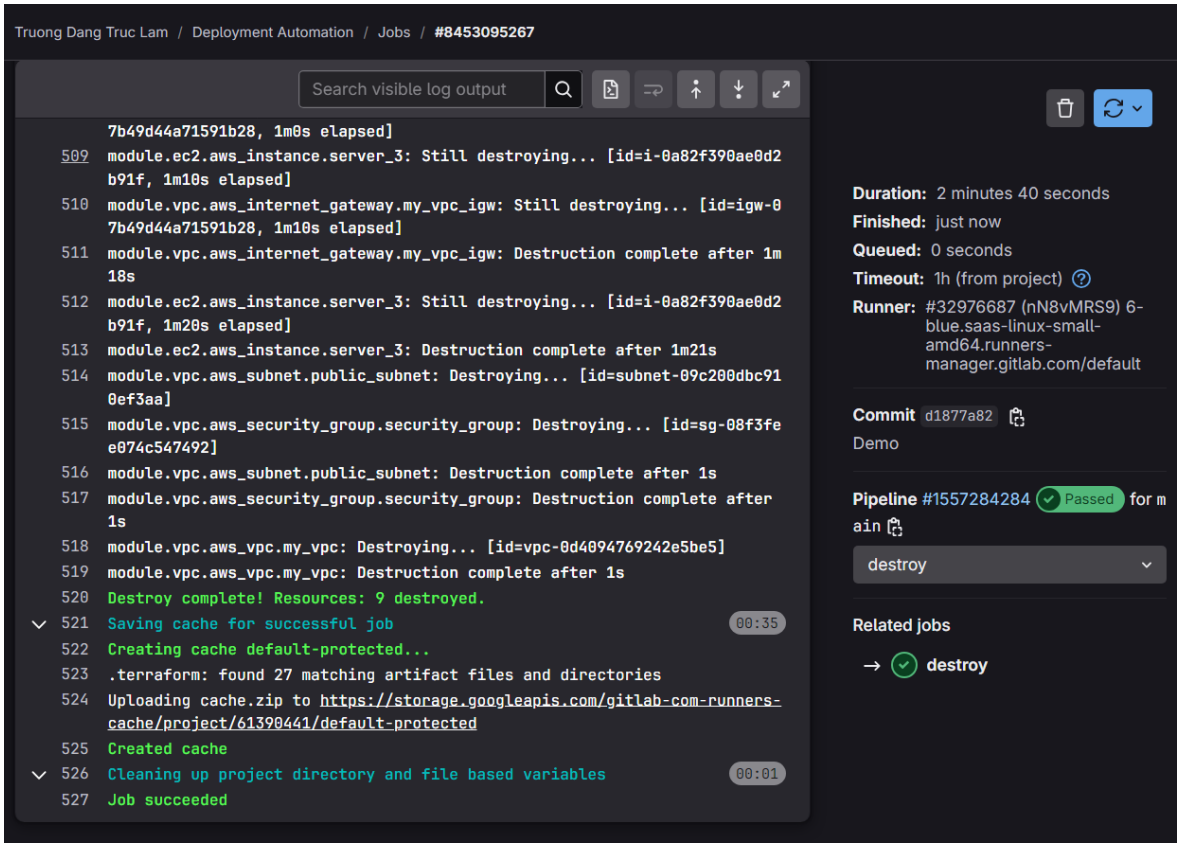

Figure 39: Manual destroying with GitLab CI/CD passed successfully

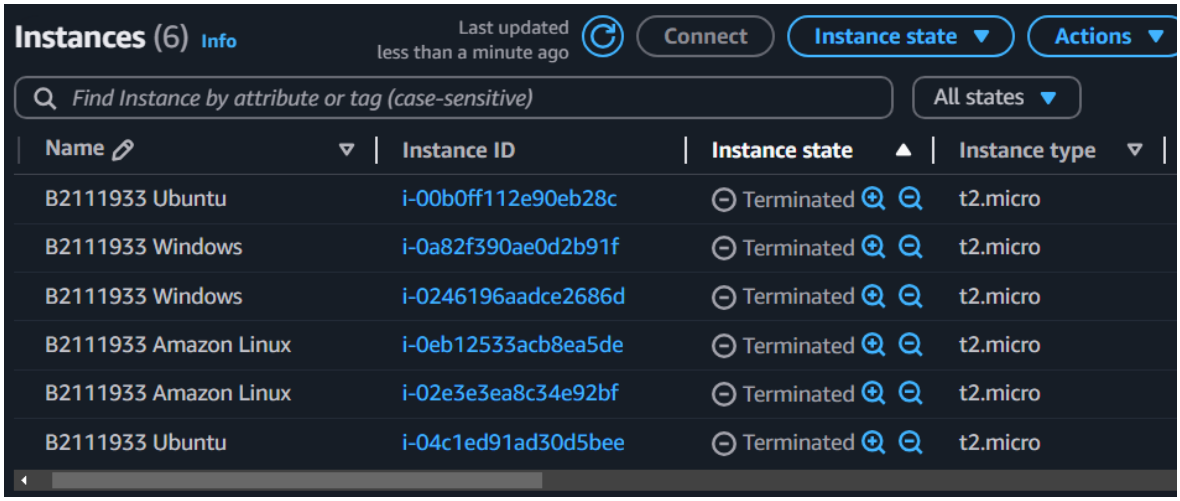

Figure 40: Verify result on AWS console after destroying with GitLab CI/CD

3. Test and evaluate AWS autoscaling

To ensure scalability and reusability of our Terraform configuration, we can implement our code with modular approach. By breaking down the infrastructure into smaller, reusable modules, we can easily modify and scale specific components independently with variables and output. For instance, to increase the number of EC2 instances, we can adjust the count parameter within the resource block, allowing us to deploy multiple instances with minimal changes to the overall configuration. This modular design promotes efficient management and simplifies the process of scaling our infrastructure as needed.

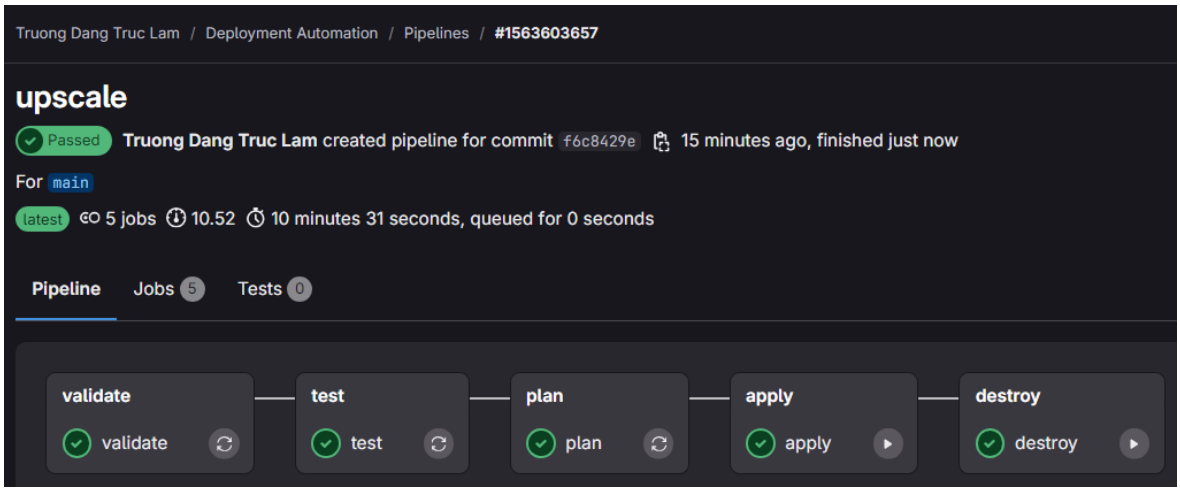

Figure 41: GitLab CI/CD pipline passed successfully after upscaling

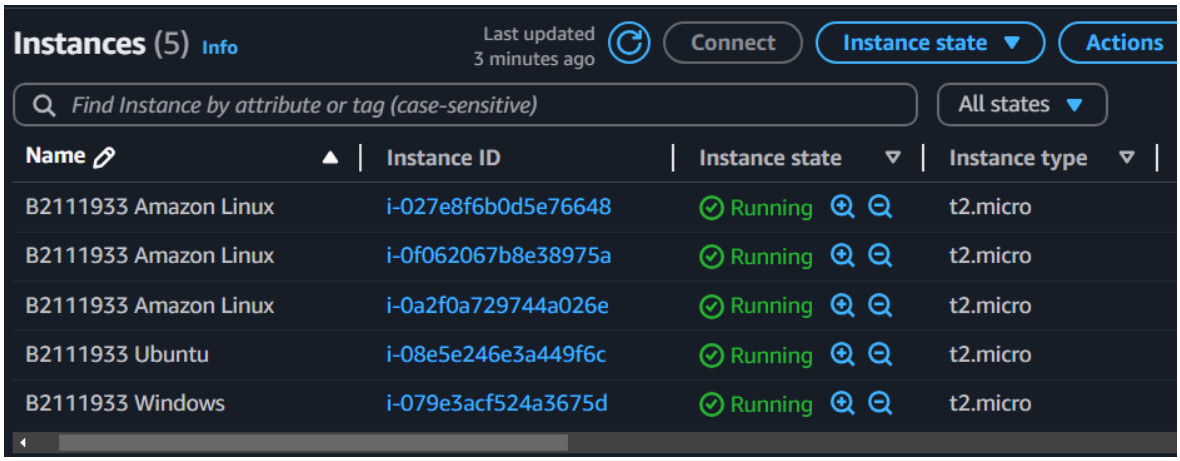

Figure 42: Verify result on AWS console after upscaling

CONCLUSION

1. Result

The research underscores the substantial benefits of automated deployment compared to manual cloud infrastructure configuration. This was demonstrated by using Terraform and GitLab CI/CD on AWS infrastructure. Terraform is IaC tool, allowed us to define cloud resources in a declarative manner, making them versionable and reusable. While GitLab CI/CD is a DevOps platform which automated the process of building, testing, and deploying infrastructure changes.

In conclusion, automated deployment can simplify the configuration and provisioning of cloud computing resources, leading to fewer errors and greater efficiency. It enables consistent and repeatable deployments, ensuring adherence to standards and best practices. Moreover, this automation fosters improved collaboration among development, operations, and infrastructure teams, promoting a DevOps culture and accelerating product delivery.

2. Development orientation

This research is just the initial exploration into DevOps aspect, particularly focusing on AWS infrastructure provisioning using Terraform with GitLab CI/CD. To delve deeper, we could explore additional DevOps tools like Ansible for configuration management, Docker for containerization or Kubernetes for container orchestration. Furthermore, expanding the AWS resource utilization to include services like RDS for relational databases, Lambda for serverless functions, and API Gateway for RESTful APIs could provide a more comprehensive understanding of cloud-native development.

Furthermore, to enhance the security posture of the DevOps pipeline, incorporating DevSecOps practices is essential. This involves integrating security measures throughout the development lifecycle, from code scanning and vulnerability assessments to infrastructure hardening and compliance checks. Additionally, integrating machine learning into the pipeline can enable automated security testing, anomaly detection, and predictive threat modeling. ML-powered tools can analyze vast amounts of data to identify patterns and potential security risks, thereby improving the overall security of the DevOps process.

REFERENCES

[1] Benneth Uzoma and Bonaventure Okhuoya. "A RESEARCH ON CLOUD COMPUTING."

[2] Anna wiedemann, Nicole Forsgren, Manuel Wiesche, Heiko Gewald and Helmut krcmar. The DevOps Phenomenon, December 2019.

[3] A. Phillips, M. Sens, A. de Jonge, and M. van Holsteijn, The IT Manager's Guide to Continuous Delivery: Delivering business value in hours, not months: XebiaLabs, 2015.

[4] Akond Rahman, Rezvan Mahdavi-hezaveh, and Laurie Williams. A Systematic Mapping Study of Infrastructure as Code Research.

[5] Amazon Web Services. AWS Documentation. https://docs.aws.amazon.com/. Accessed on September 9, 2024.

[6] GitLab Inc. GitLab Documentation. https://docs.gitlab.com. Accessed September 9, 2024.

[7] HashiCorp. Terraform Documentation. https://developer.hashicorp.com/terraform. Accessed September 9, 2024.

[8] National Institute of Standards and Technology. The NIST definition of cloud computing. NIST Special Publications. 800-145, 2011.

[9] OpenSSH. SSH Documentation. https://www.openssh.com/manual.html. Accessed November 23, 2024.

[10] Microsoft. "Remote Desktop Protocol (RDP) Background Channel." https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-rdpbcgr. Accessed November 23, 2024.

[11] Docker. "Docker Overview." Docker Documentation. https://docs.docker.com/. Accessed November 28, 2024.