A Project Report

on

Automated Web Hosting Solution With Single click

Replication to Uat and Prod Environment

Submitted

by

Janhvi Kurkure

TABLE OF CONTENT

1. Introduction

1.1. Introduction Overview

This project uses Infrastructure as Code with Terraform to automate the deployment of fault-tolerant, scalable web servers on Azure.
Eliminating manual server provisioning and configuration—which are prone to errors, inconsistent environments, and high operational overhead—is the driving force behind the project.
The solution uses Azure Load Balancer to provide high availability and horizontal scaling by serving dynamically generated web pages that show the current environment and serving virtual machines.

Uses:

- Deployments of enterprise clouds
- Integration of the CI/CD pipeline
- Environments for testing and staging
- Solutions for scalable web hosting

Benefits:

- Repeatable, reliable, and automated deployments
- Decreased configuration drift and manual labour
- Fault tolerance and high availability
- Environments (Dev, UAT, and Prod) are clearly separated.
- Infrastructure that is both scalable and maintainable

1.2. Problem Statement

An IAC Tool leveraged to deploy Virtual machine-based web servers, Load balancer. A website hosted on two virtual machine instances added to the Backend of an Azure Load balancer. IAC tool capable of dynamically deploying the setup to Development, Uat (test), and Production Environment by using Parameters specific to an environment.

1.3. Objectives

The following objectives were successfully achieved in this project:

- Automate the deployment of Azure infrastructure using Terraform.
- Provision separate environments (Dev, UAT, Prod) using Terraform workspaces.
- Deploy two Linux virtual machines behind an Azure Standard Load Balancer.
- Configure each VM to serve a custom HTML page indicating the environment and VM instance.
- Automate Nginx installation and web content configuration via custom startup scripts.

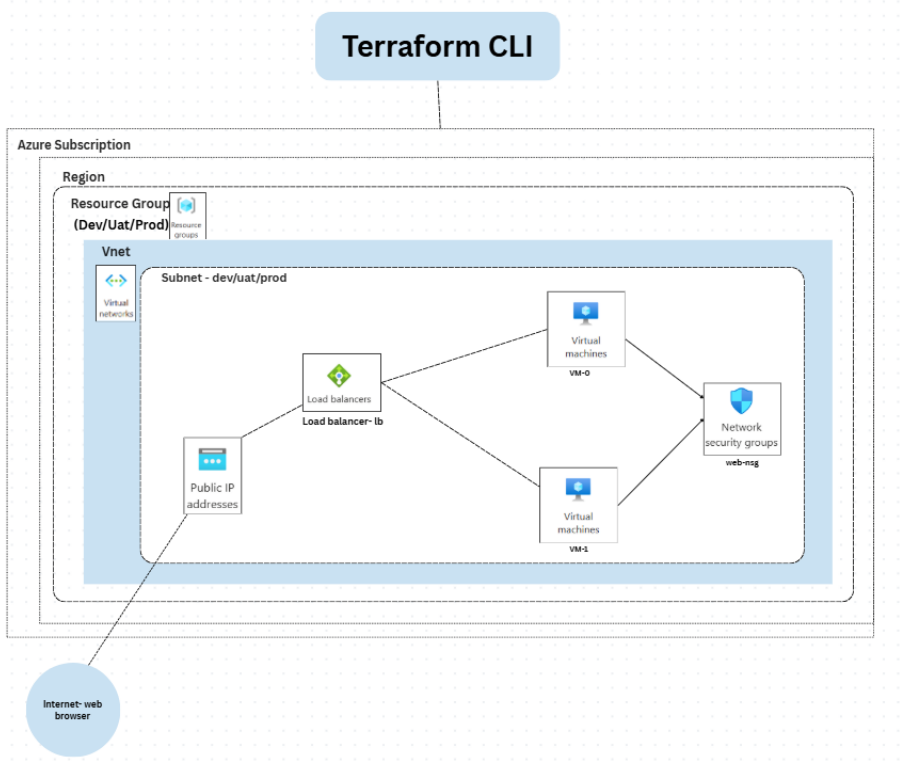
1.4. Scope of Project

The study and application of Infrastructure as Code principles using Terraform to automate Azure infrastructure deployments are covered in this project. Deploying and setting up a load-balanced web server configuration that dynamically adjusts to various environments is its main goal.
Cloud computing, DevOps methodologies, Azure infrastructure services, Terraform IaC workflows, Linux server administration, and automation scripting are among the topics of study. Beyond serving custom web pages, the scope excludes advanced monitoring, auto-scaling policies, and application-level deployment; these topics can be investigated in subsequent upgrades.

2. System Design and Methodology

2.1 System Architecture

The project follows a client–load balancer–VMs architecture on Microsoft Azure. Below is the high-level design:



Components:
User/Client: Sends HTTP requests to the application.
Load Balancer: Azure Standard Load Balancer distributes traffic evenly across backend pool VMs.
Backend VMs: Two Linux (Ubuntu) virtual machines serving environment-specific web pages via Nginx.
NSG (Network Security Group): Allows only HTTP traffic to VMs.

2.2 Development environment

Hardware:

- Local Machine:
  - CPU
    RAM 4GB+
    Internet

- Azure Cloud:
  - 2 vCPUs
    Standard Load Balancer
    Standard IP addresses
    Standard_LRS disks

Software:

- Cloud Platform:
  - Microsoft Azure

- Iac Tool:
  - Terraform v1.x

- OS (on vms):
  - Ubuntu Server 18.04 LTS

- Web server:
  - Nginx

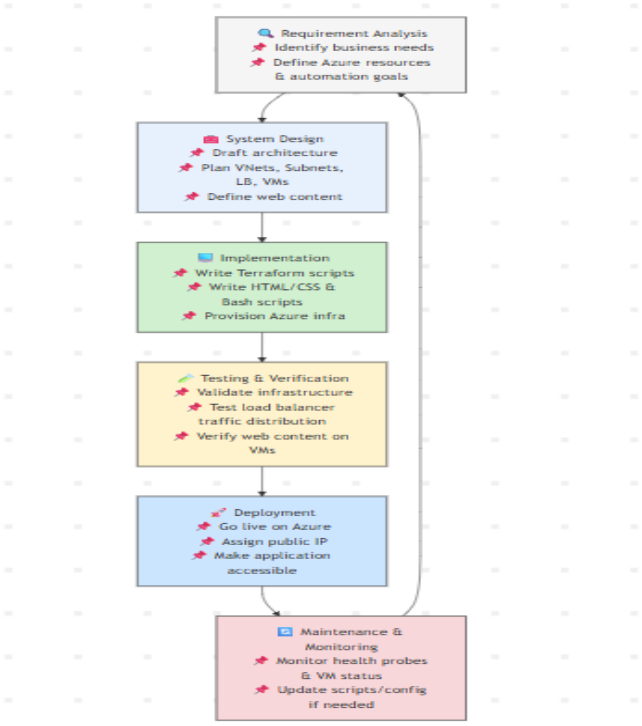
- Scripting:
  - Bash

- Language:
  - HCL(Terraform),HTML, CSS

- Editor:
  - VS Code

- CLI Tools:
  - Azure CLI / Terraform CLI

2.3 Methodology



Software development cycle

Phase-1 Requirement analysis

The first actionable phase involved identifying and documenting the requirements:

- Understand business goals for deploying a website on Azure with load balancing and high availability.
- Identify the target environments: Development (Dev), User Acceptance Testing (UAT), and Production (Prod).
- List the required components of the infrastructure:
  - Virtual Machines (VMs) as web servers.
  - Load Balancer (LB) to distribute traffic across VMs.
  - Network Security Groups (NSG) to secure traffic.
  - Virtual Networks (VNets) and subnets for network segmentation.

Phase-2 Design Infrastructure

With the requirements in place, the architecture was designed:

- Plan the virtual network topology and IP address ranges.
- Define VM specifications (size, OS image, admin credentials).
- Plan the Load Balancer frontend and backend configuration.
- Specify NSG inbound and outbound rules for HTTP/HTTPS access.

Phase-3 Implementation

Terraform was used to codify the entire infrastructure:

- Create reusable .tf files with modular code.
- Use variables to parameterize environment-specific settings.
- Create templates for VM initialization, including automated installation of nginx and deployment of custom HTML/CSS.
- Implement workspaces to isolate Dev, UAT, and Prod deployments.

Phase-4 Test in Dev, Uat and Prod Workspace

The code was first applied in the **Dev workspace**:
Deploy all resources.
Validate that the VMs are provisioned correctly, nginx is installed, and the load balancer alternates between VM-0 and VM-1 properly.
Debug and refine the code where necessary.

Phase-5 Monitor & Cleanup

Post-deployment, monitoring was set up:

- Check load balancer health probes.
- Verify uptime and traffic distribution between VMs.
- Destroy unused resources in Dev/UAT to save costs.

3. Implementation and Result

3.1. Modules of Project

The following elements form the modular design of the project:

- Module for Infrastructure-as-Code: Terraform was used to write this.
  Generates Azure resources, including public IP, resource groups, VNets, subnets, NSGs, NICs, virtual machines, and load balancers.

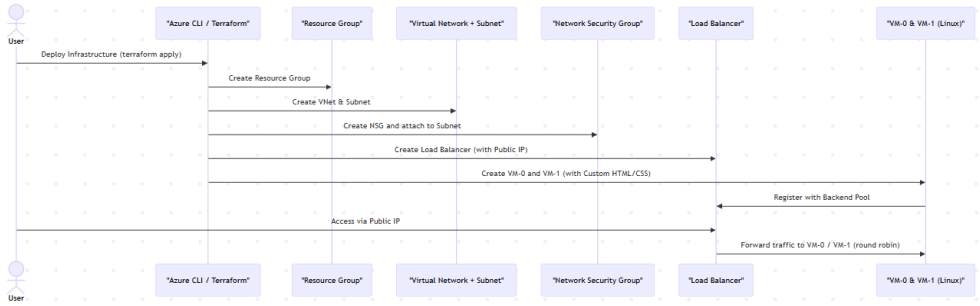
- Environment-specific parameters: Dev, UAT, and Prod.

- Module for Web Server:
  Nginx is provisioned on Linux virtual machines by bash scripts.
  deploys environment-specific style.css and dynamic index.html files for virtual machines.

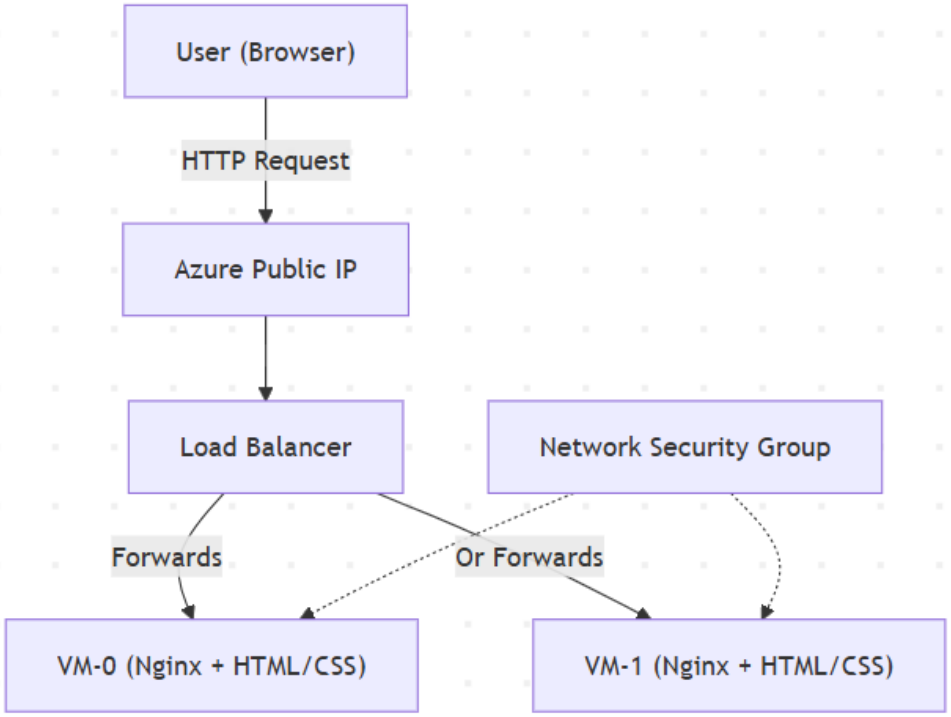
- Health probes were tracked via the Azure portal.

- To guarantee clarity and reusability, each module is logically divided.

3.2. Implementation detail



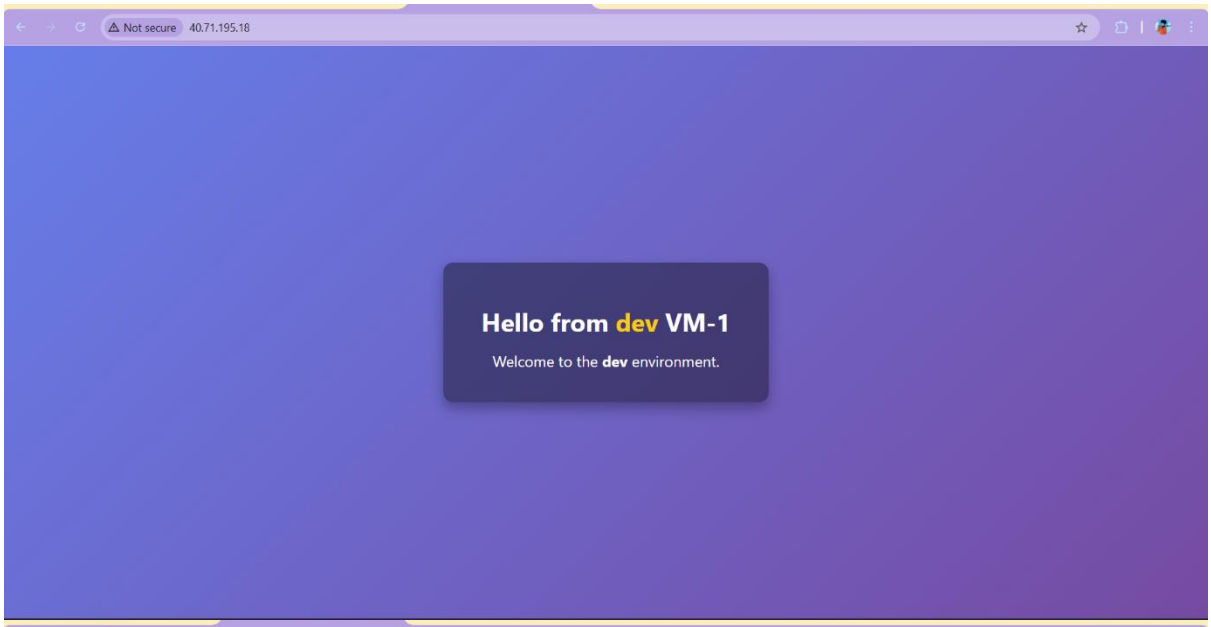
Sequence diagram of the Project

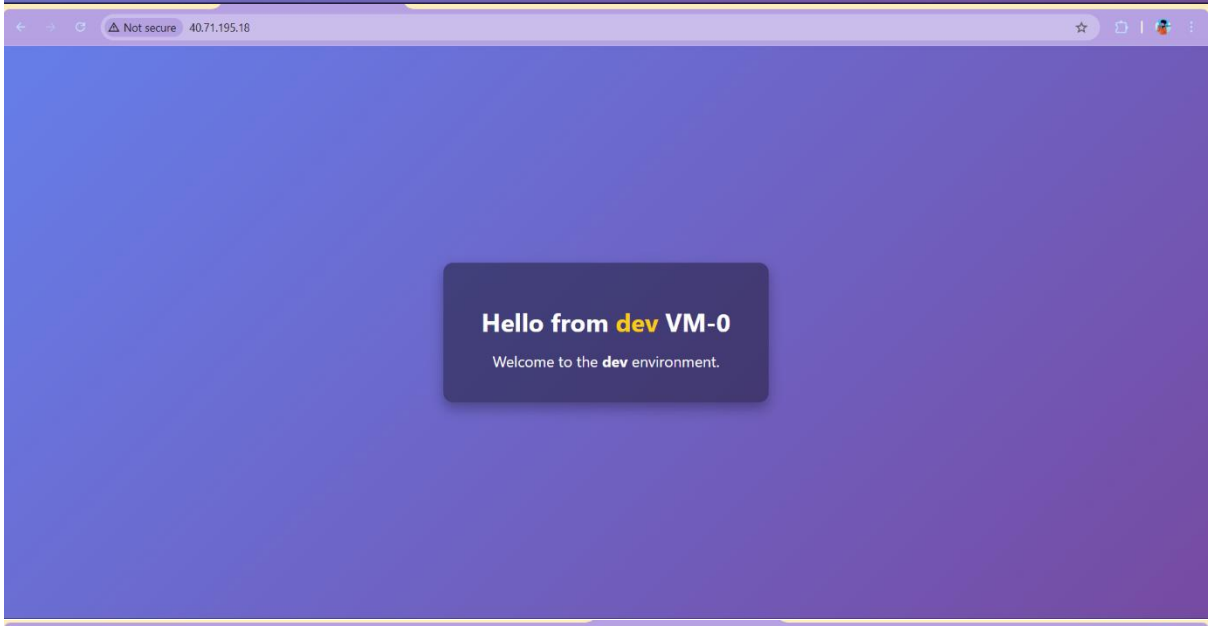


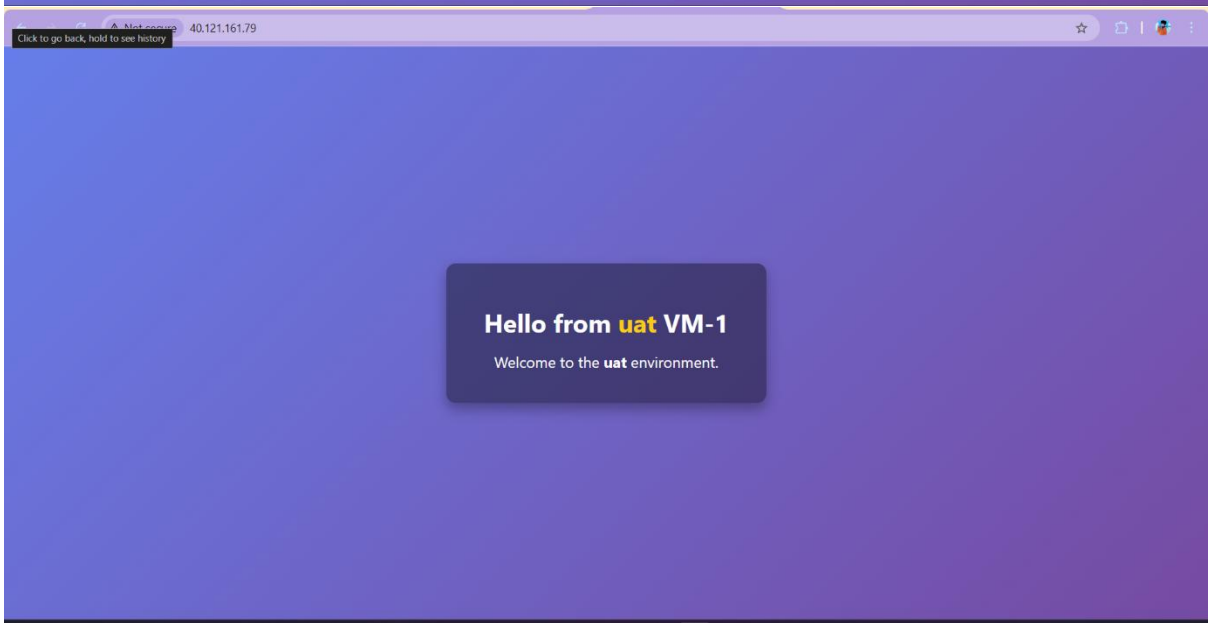
Data Flow Diagram of the Project

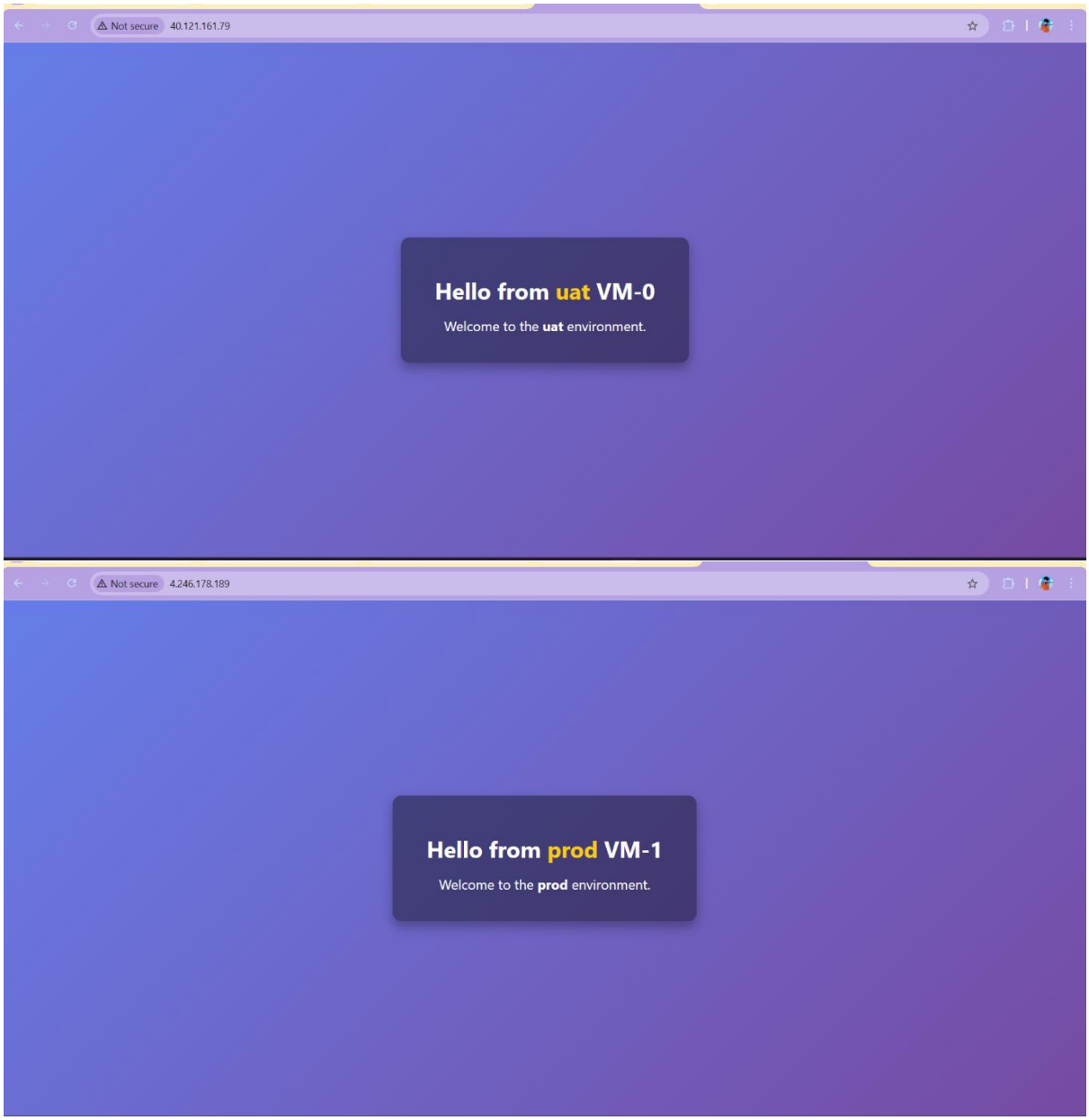
3.3. Result

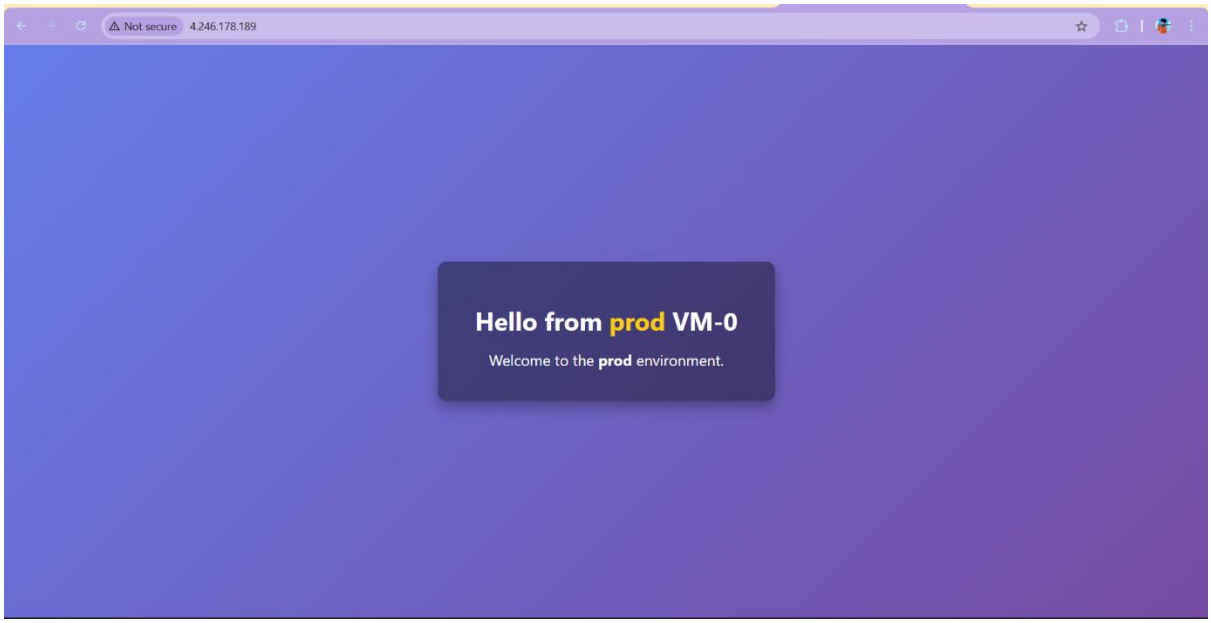
- Successfully deployed separate environments (*Dev, UAT, Prod*) with isolated resources.
- Web page showed correct environment and VM index on each request.
- Azure Load Balancer distributed traffic evenly between VMs.
- Infrastructure was fully automated, reproducible, and easy to tear down.


Not secure
40.71.195.18
Hello from dev VM-1
Welcome to the dev environment.


Not secure
40.71.195.18
Hello from dev VM-0
Welcome to the dev environment.


40.121.161.79
Click to go back, hold to see history
Hello from uat VM-1
Welcome to the uat environment.


Not secure
40.121.161.79
Hello from uat VM-0
Welcome to the uat environment.
Not secure
4.246.178.189
Hello from prod VM-1
Welcome to the prod environment.



4. Conclusion

This project effectively showcased the capabilities of cloud automation and Infrastructure-as-Code (IaC) in deploying environment-specific, scalable, and reproducible infrastructure on Microsoft Azure. We provisioned whole environments (Dev, UAT, and Prod) with load balancers, virtual networks, subnets, network security groups, and Linux virtual machines with dynamic, environment-aware webpages using Terraform.

What we accomplished:

- Cloud infrastructure deployment pipelines that are completely automated.
- Terraform workspaces are used to isolate environments.
- Reusable and adaptable scripts for customisation and provisioning.
- Verification of proper dynamic content delivery and load balancing.

5. References

- https://developer.hashicorp.com/terraform
- https://developer.hashicorp.com/terraform/tutorials
- https://developer.hashicorp.com/terraform/tutorials/azure-get-started