Multi-Cloud Load Balancing

Distributing Live Traffic Across AWS, Azure & Google Cloud

Ahmed Abdelwahed
ahmed@abdelwahed.me
www.abdelwahed.me
LinkedIn

What This Lab Builds

In this lab you will build a working multi-cloud environment in which a single, centralized load balancer distributes live web traffic across three web servers — one hosted in Amazon Web Services (AWS), one in Microsoft Azure, and one in Google Cloud (GCP). The result is a hands-on simulation of how large enterprises run resilient, vendor-independent infrastructure.

By the end you will have deployed virtual machines in three different clouds, connected them securely, configured a reverse-proxy load balancer, tested automatic failover between providers, and optionally layered on TLS, DNS, and monitoring.

Phase 1 — Prepare Cloud Accounts

Before provisioning anything, make sure you can log in to all three provider consoles and that billing/free-tier is active on each.

Create or Verify Accounts

1. Sign in to the AWS Management Console.
2. Sign in to the Microsoft Azure Portal.
3. Sign in to the Google Cloud Console and ensure a billing-enabled project exists.

Console links:

- AWS Console — aws.amazon.com/console
- Azure Portal — portal.azure.com
- Google Cloud Console — console.cloud.google.com

2.2 Recommended Conventions

- Use free-tier-eligible instance sizes throughout.
- Use Ubuntu 22.04 LTS on every VM for a consistent command set.
- Where possible, pick geographically close regions across the three providers to reduce cross-cloud latency.

Phase 2 — Deploy the Web Servers

You will now create one Ubuntu web server in each cloud. Each runs NGINX and serves a single page that identifies its provider, so you can visually confirm which backend answered each request.

AWS — Launch an EC2 Web Server

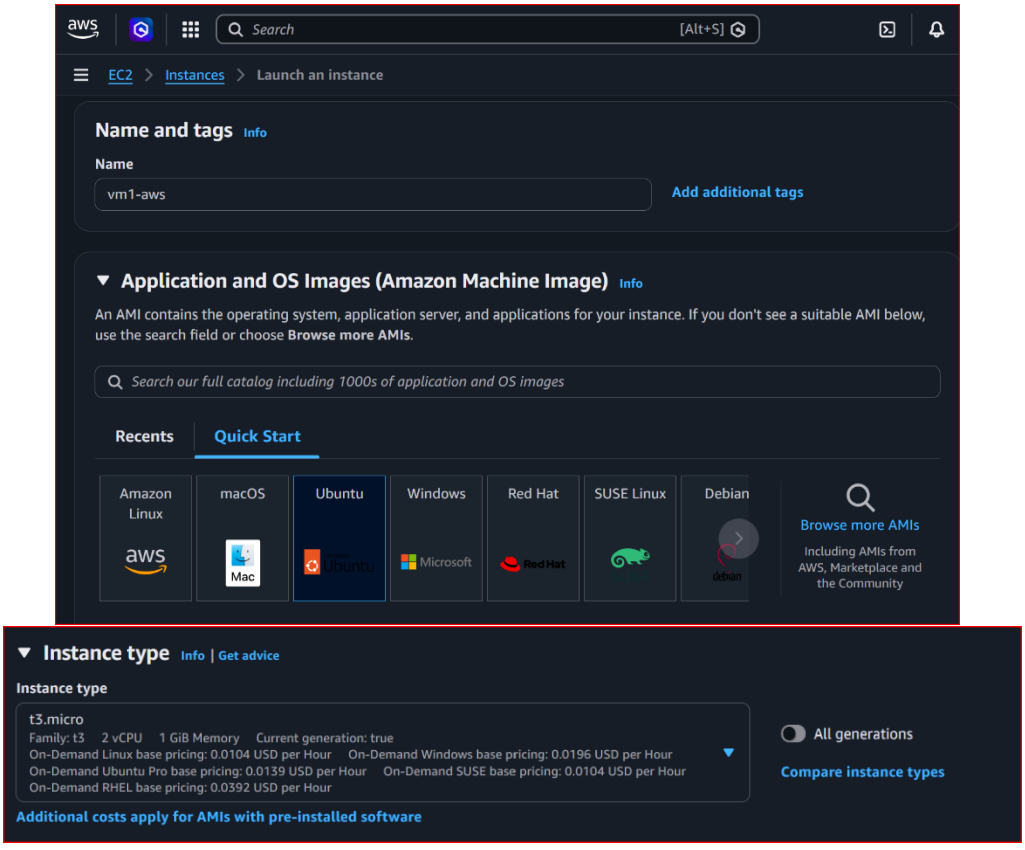
1. In the EC2 console choose Launch instance.
2. Select Ubuntu Server 22.04 LTS as the AMI.
3. Choose instance type t2.micro or t3.micro (free-tier).
4. Create or select a key pair for SSH access.
5. Configure the Security Group to allow inbound SSH (22) and HTTP (80).
6. Launch the instance and note its public IPv4 address.

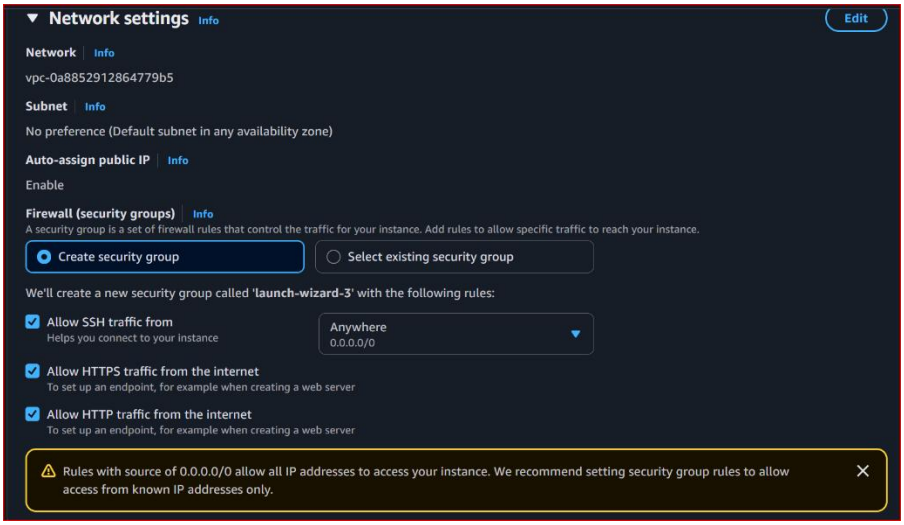
Security group inbound rules:

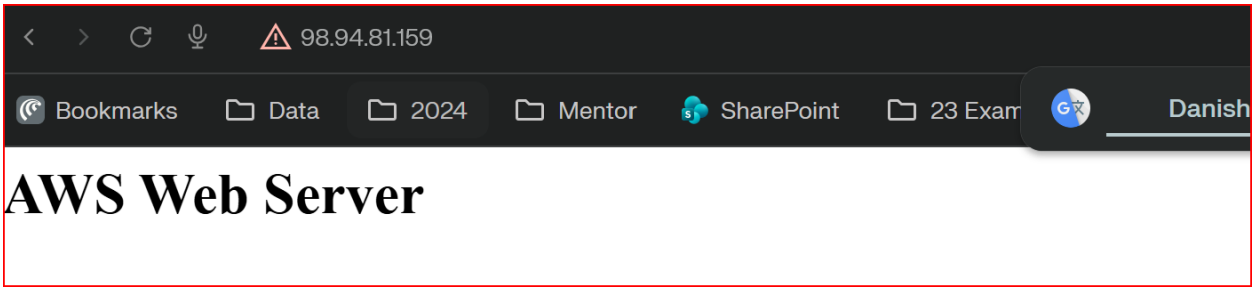
Type	Protocol	Port	Source
SSH	TCP	22	Your IP /32
HTTP	TCP	80	0.0.0.0/0

SSH in, then install NGINX and publish the identifying page:

```
sudo apt update
sudo apt install nginx -y
echo "<h1>AWS Web Server</h1>" | sudo tee /var/www/html/index.html
```



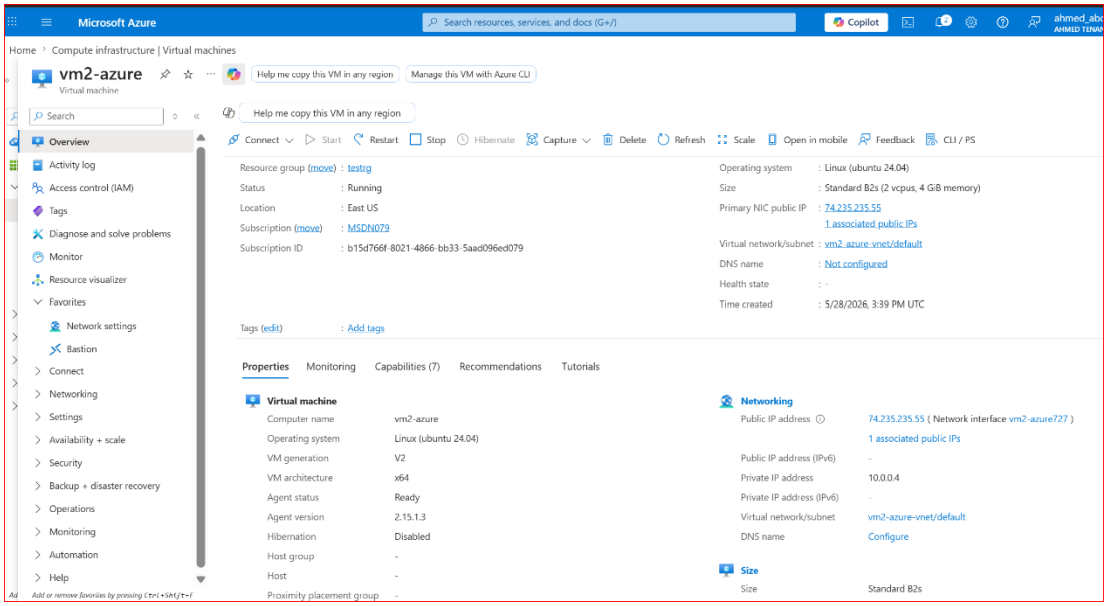

Network settings Info
Edit
Network Info
vpc-0a8852912864779b5
Subnet Info
No preference (Default subnet in any availability zone)
Auto-assign public IP Info
Enable
Firewall (security groups) Info
A security group is a set of firewall rules that control the traffic for your instance. Add rules to allow specific traffic to reach your instance.
Create security group
Select existing security group
We'll create a new security group called 'launch-wizard-3' with the following rules:
Allow SSH traffic from
Helps you connect to your instance
Anywhere
0.0.0.0/0
Allow HTTPS traffic from the internet
To set up an endpoint, for example when creating a web server
Allow HTTP traffic from the internet
To set up an endpoint, for example when creating a web server
Rules with source of 0.0.0.0/0 allow all IP addresses to access your instance. We recommend setting security group rules to allow access from known IP addresses only.


98.94.81.159
Bookmarks
Data
2024
Mentor
SharePoint
23 Exam
Danish
AWS Web Server

Azure — Create an Ubuntu VM

1. In the Azure Portal choose Create a resource > Virtual machine.
2. Select the Ubuntu 22.04 LTS image and size Standard_B1s.
3. Configure SSH public key authentication.
4. Under Networking, open inbound ports 22 (SSH) and 80 (HTTP).
5. Create the VM and note its public IP address.

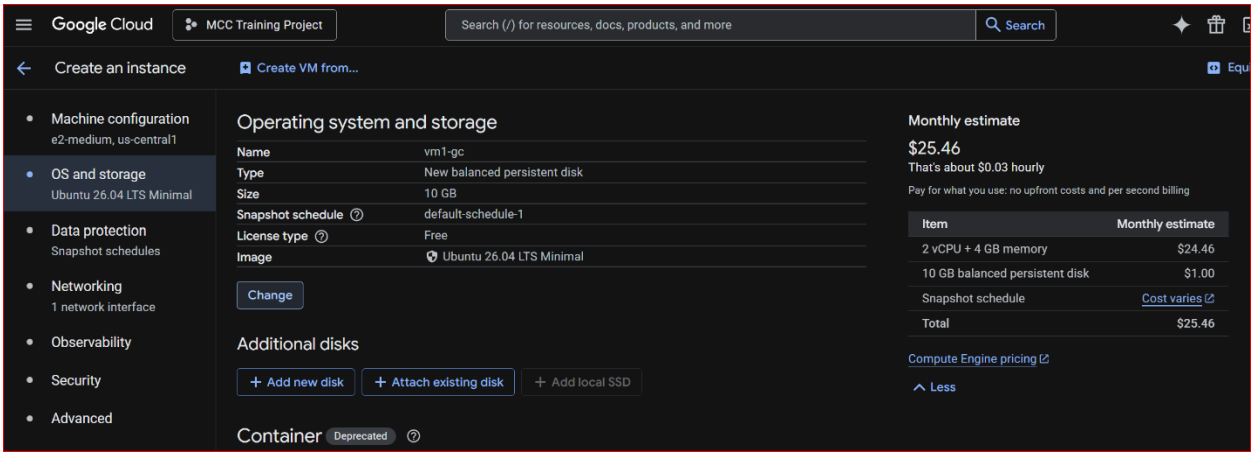
```
sudo apt update
sudo apt install nginx -y
echo "<h1>Azure Web Server</h1>" | sudo tee /var/www/html/index.html
```

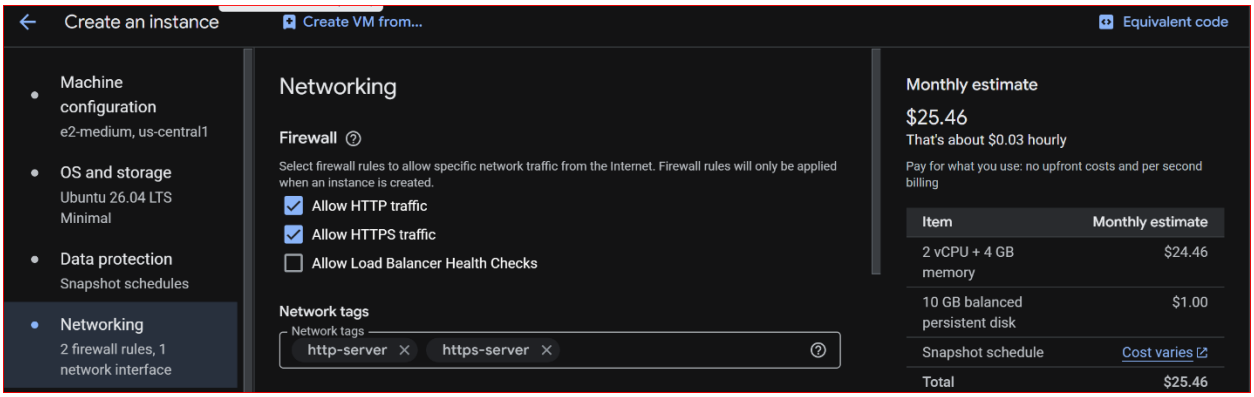


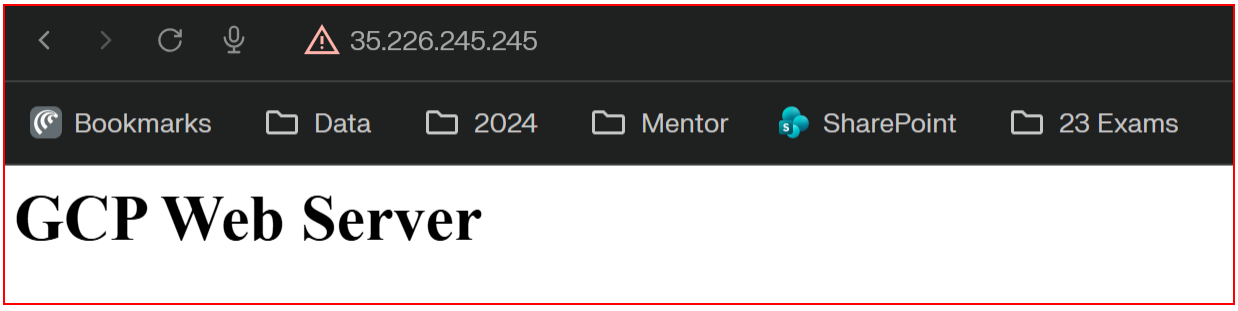
GCP — Create a Compute Engine VM

1. In the Google Cloud Console open Compute Engine > VM instances > Create instance.
2. Choose machine type e2-micro and an Ubuntu 22.04 LTS boot disk.
3. Under Firewall, enable Allow HTTP traffic.
4. Ensure SSH is permitted (allowed by default via the console's SSH button).
5. Create the VM and note its external IP address.

```
sudo apt update
sudo apt install nginx -y
echo "<h1>GCP Web Server</h1>" | sudo tee /var/www/html/index.html
```







Phase 3 — Verify Connectivity

Confirm each web server is independently reachable before introducing the load balancer. Visiting each public IP in a browser should display its provider's banner.

```
http://AWS_PUBLIC_IP      ->  AWS Web Server
http://AZURE_PUBLIC_IP    ->  Azure Web Server
http://GCP_PUBLIC_IP      ->  GCP Web Server
```

You can also verify from the command line with curl:

```
curl http://AWS_PUBLIC_IP
curl http://AZURE_PUBLIC_IP
curl http://GCP_PUBLIC_IP
```

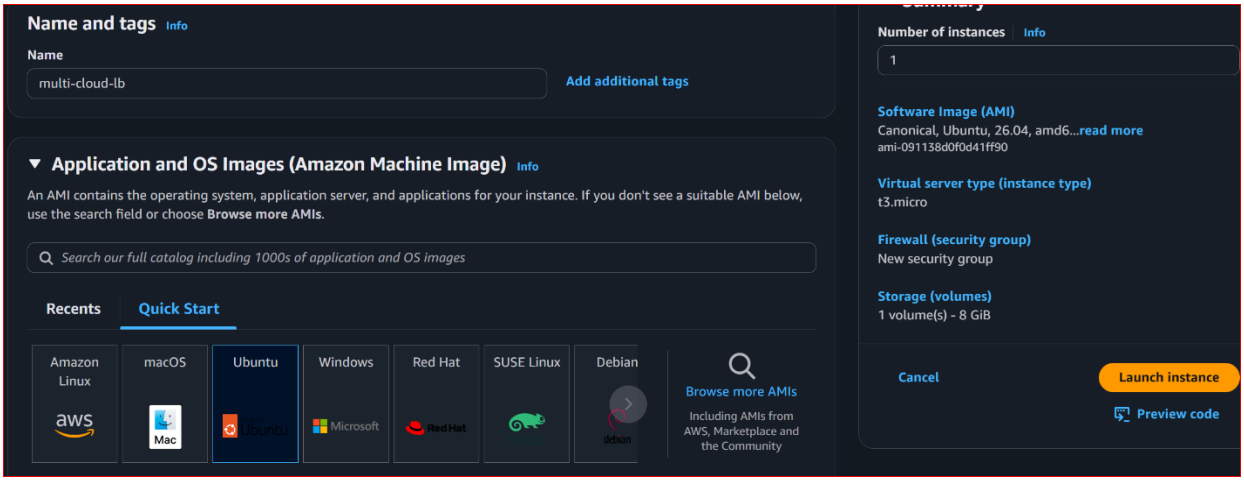
Phase 4 — Create the Central Load Balancer VM

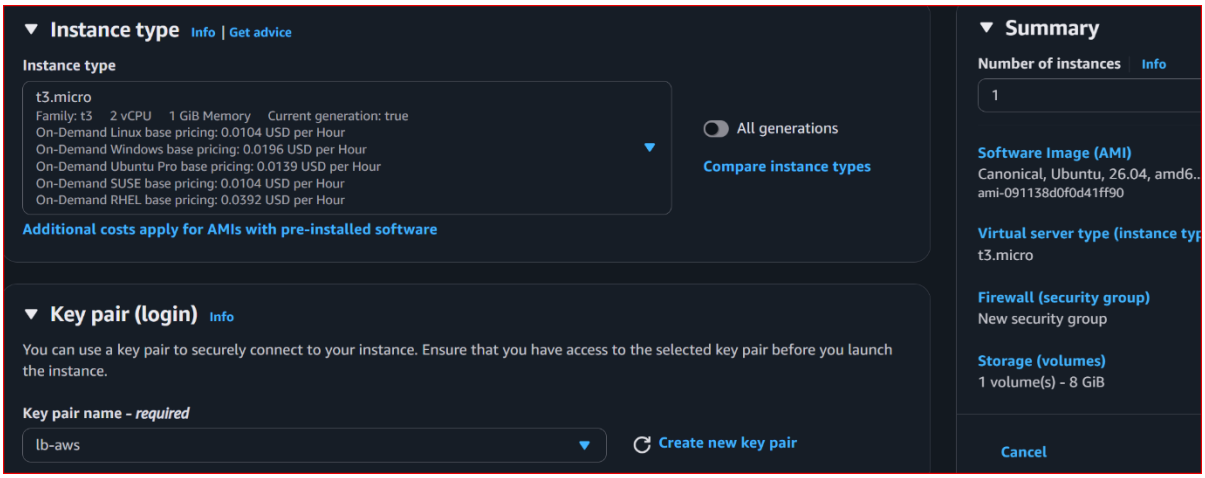
Provision a fourth Ubuntu VM that will act as the dedicated reverse-proxy load balancer. AWS is used here for simplicity, but any cloud will work.

1. Launch another EC2 Ubuntu 22.04 instance.
2. Name it multi-cloud-lb.
3. Open inbound ports 22 (SSH) and 80 (HTTP).
4. Note its public IP — this becomes the single entry point for all traffic.

Install NGINX on the load balancer host:

```
sudo apt update
sudo apt install nginx -y
```





Phase 5 — Configure the NGINX Load Balancer

Define an upstream pool containing the three cloud web servers, then create a server block that proxies all incoming requests to that pool.

Edit the NGINX Configuration

```
sudo rm /etc/nginx/sites-enabled/default
sudo nano /etc/nginx/conf.d/multicloud.conf

upstream multi_cloud_backend {
    random;

    server 74.235.235.55;
    server 98.94.81.159;
    server 35.226.245.245;
}

server {
    listen 80 default_server;
    server_name _;

    location / {
        proxy_pass http://multi_cloud_backend;

        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

Validate and Apply

Always test the configuration syntax before reloading:

```
sudo nginx -t
sudo systemctl restart nginx
```

Phase 6 — Test Load Balancing

Open the load balancer's public IP in a browser and refresh repeatedly. Because NGINX uses round-robin by default, successive requests are answered by different clouds.

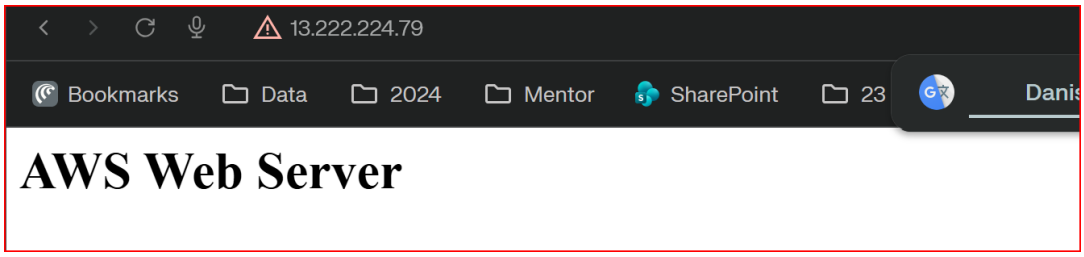
```
http://LOAD_BALANCER_IP
```

With each refresh you should rotate through the three banners:

- AWS Web Server
- Azure Web Server
- GCP Web Server

To see the rotation clearly from the command line, send several requests in a loop:

```
for i in $(seq 1 9); do curl -s http://LOAD_BALANCER_IP | grep -o
'<h1>.*</h1>'; done
```



Phase 7 — Configure Health Checks

Passive health checks let NGINX detect an unresponsive backend and temporarily remove it from rotation, so failing nodes stop receiving traffic automatically.

Update the upstream block to add failure thresholds:

```
    server 74.235.235.55 max_fails=3 fail_timeout=30s;
    server 98.94.81.159 max_fails=3 fail_timeout=30s;
    server 35.226.245.245 max_fails=3 fail_timeout=30s;
}
```

Directive	Effect
max_fails=3	After 3 failed attempts within the fail_timeout window, the backend is marked unavailable.
fail_timeout=30s	Defines both the failure-counting window and how long the backend stays marked down before NGINX retries it.

Reload NGINX to apply:

```
sudo systemctl restart nginx
```

Phase 8 — Failover Testing

Deliberately take one cloud offline and confirm the service keeps responding from the surviving providers — the essence of high availability.

Simulate a Provider Outage

On any one of the three web servers (for example, the Azure VM), stop NGINX:

```
sudo systemctl stop nginx
```

Observe Continued Service

Refresh the load balancer page or rerun the curl loop. After the failed node is detected, only the two healthy providers should answer:

```
for i in $(seq 1 9); do curl -s http://LOAD_BALANCER_IP | grep -o
'<h1>.*</h1>'; done
```

Restore the Node

Bring the stopped server back and confirm it rejoins the rotation after the fail_timeout window:

```
sudo systemctl start nginx
```

Phase 9 — DNS-Based Load Balancing (Optional)

Replace the raw IP entry point with a friendly hostname, and optionally add a second layer of resilience at the DNS level. Any managed DNS service works — Cloudflare DNS, AWS Route 53, or Azure DNS.

Create a DNS Record

1. Open your DNS provider's dashboard.
2. Create an A record named lab.yourdomain.com.
3. Point it at the load balancer's public IP address.
4. Wait for propagation, then browse to the new hostname.

```
http://portal.abdelwahed.me
```