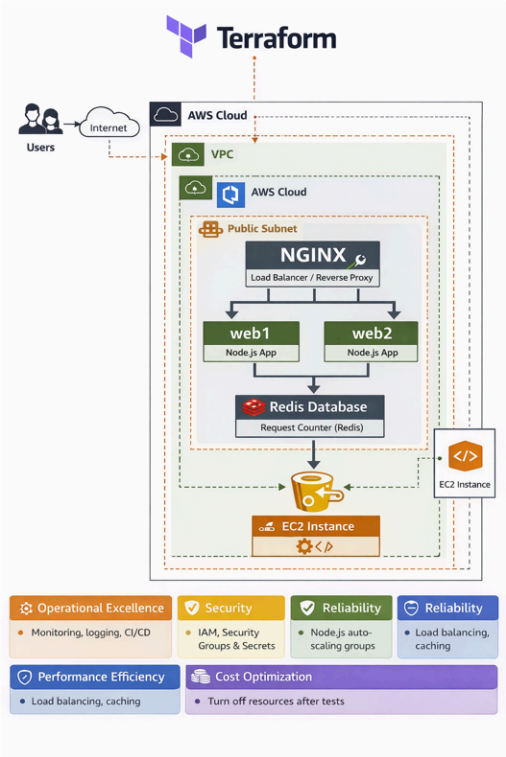
CI/CD for Terraform with GitHub Actions: Deploying a Node.js + Redis App on AWS

This is a **beginner-friendly DevOps project** designed to demonstrate how to deploy a simple **Node.js + Redis Request Counter application** on AWS using **Terraform** and **GitHub Actions CI/CD**.

While the app itself is simple, the real value lies in learning how modern DevOps tools work together to create a **repeatable, automated, and production-ready workflow**



AWS Well-Architected Pillars Covered

- **Operational Excellence** – CI/CD, logging, automation
- **Security** – IAM, Security Groups, least privilege
- **Reliability** – Multiple app instances
- **Performance Efficiency** – Load distribution via NGINX
- **Cost Optimization** – Resource teardown via CI/CD

What This Project Demonstrates

- Infrastructure as Code (IaC) using **Terraform**
- Containerization with **Docker & Docker Compose**
- Reverse proxy and load distribution using **NGINX**
- CI/CD automation using **GitHub Actions**
- Secure and cost-aware AWS deployments
- Real-world change propagation using Git push 🚀

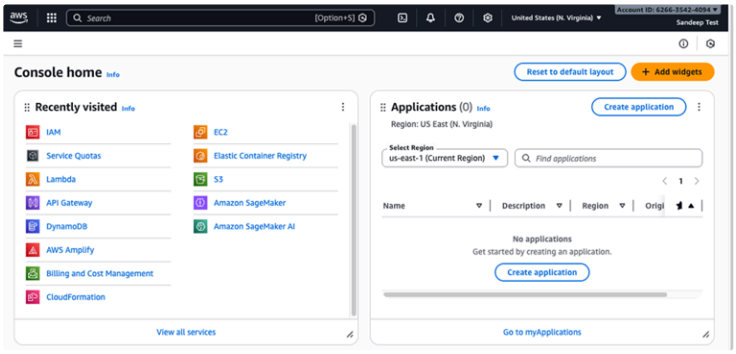
🧠 Application Concept

The application is a **Request Counter** that:

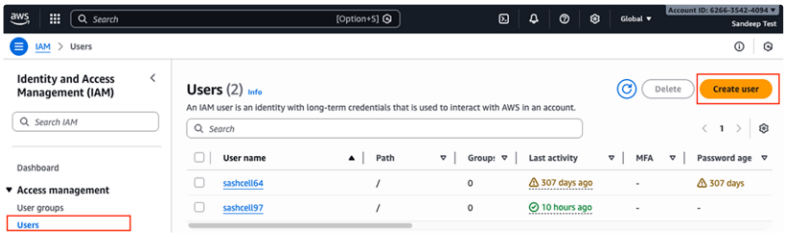
- Tracks total page visits using **Redis**
- Displays:
  - Total visit count
  - Hostname serving the request (`web1` or `web2`)
- Runs on **Node.js (Express)**
- Served via **NGINX**
- Enhanced with a **beautiful HTML UI** 🎨

Step 1: Pre-requisites – AWS Account + IAM User

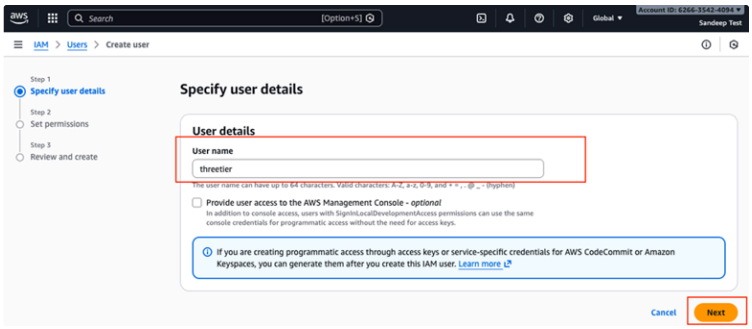
Navigate to AWS Console and click on IAM



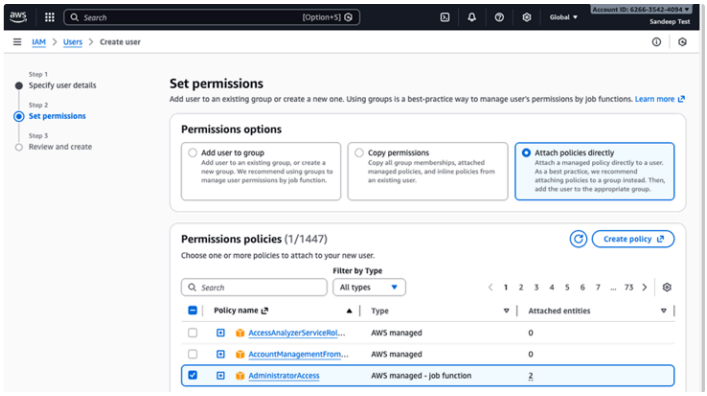
Now click on **Users** and click on **Create User**



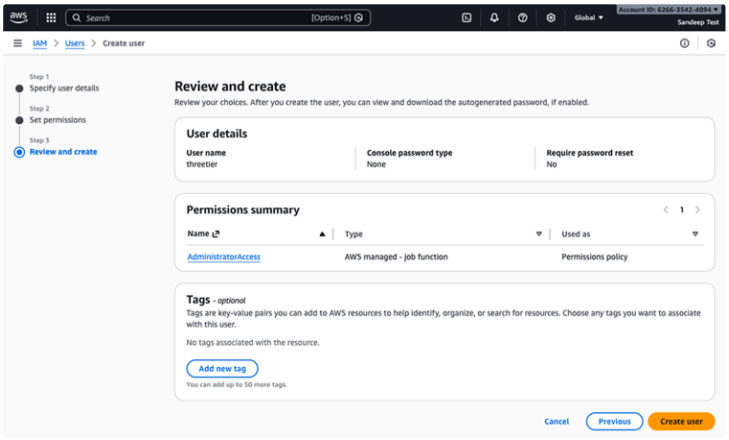
Next Step add a **Name** for the User and then click on Next



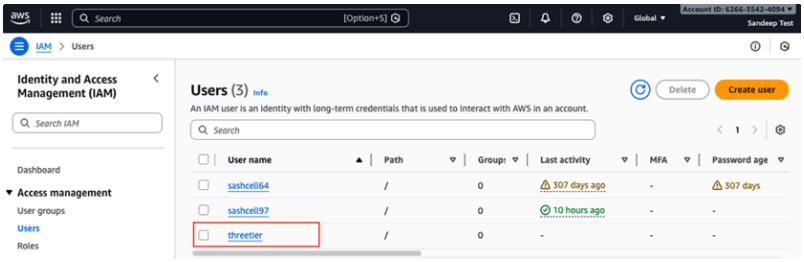
Now select the level of Permissions you need for access. We've selected **AdministratorAccess** however this depends in an enterprise or corporate setup. Then click on Next



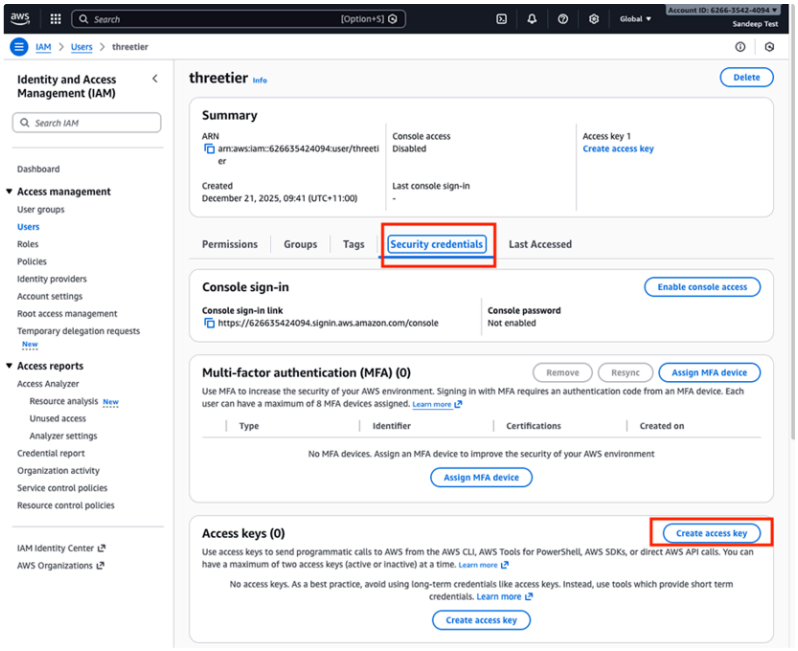
Review the options and then click on **Create User**



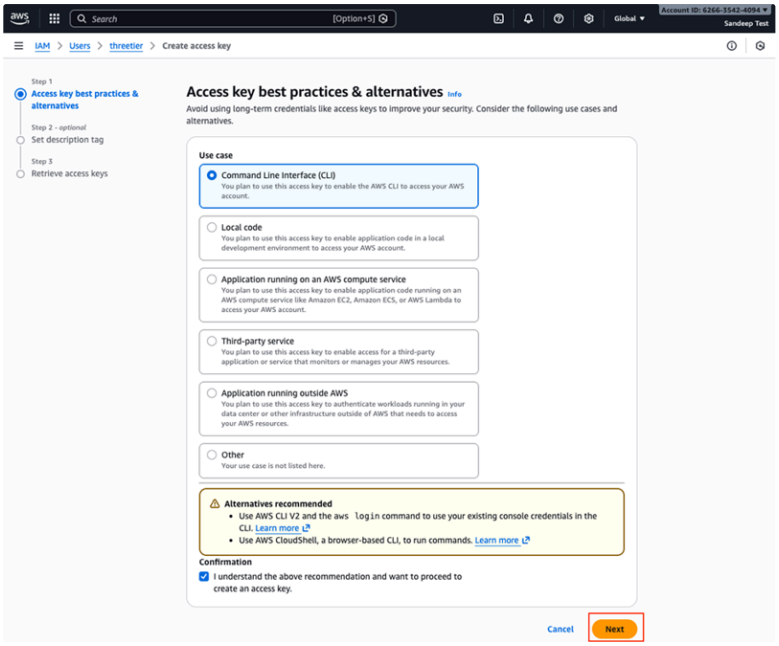
Once this is setup – we need to ensure that we can provide CLI access – Hence for this we now need to click on the newly created Administrator access



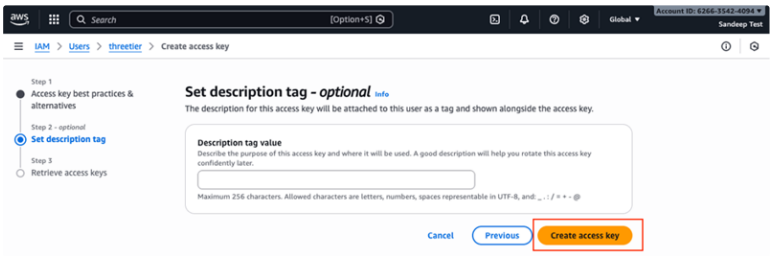
Select **Security Credentials** and then click on **Create access key**



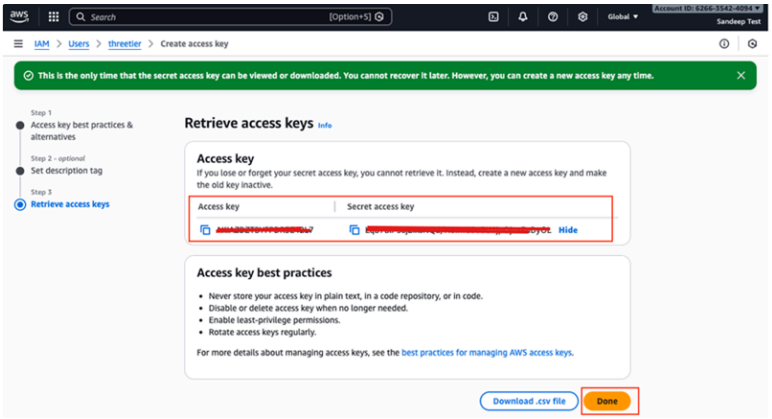
Now select **Command Line Interface**, select the Confirmation box and then click on **Next**



Move on to the next step and click on **Create Access Key**



Now we will have the Access Keys. We need to make sure that we keep these keys secure and safe.



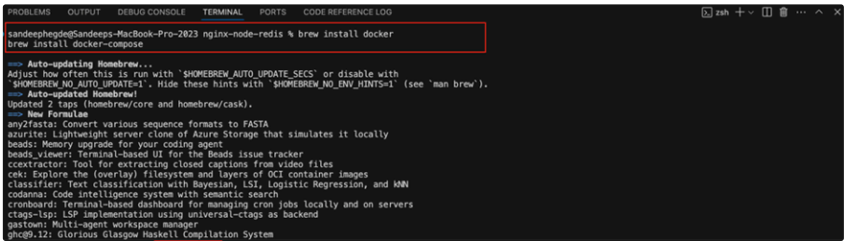
Keep these keys handy in a notepad as we will need this later to be used for Github actions.

Installing Docker

Before we deploy the app to AWS using Terraform, let's test it locally to make sure everything runs smoothly

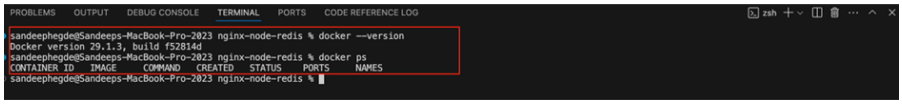
> Tip: If Docker is not installed we have to install this on the machine. Below instructions are for MAC.

```
brew install docker
brew install docker-compose
```



To validate which version is running please run

```
docker --version
docker ps
```



Step 2: Introduction to Repository files

Complete code for this project is available at GitHub - sandeep-ssh/nginx-node-redis

To begin, you can fork this repository under your own GitHub username and then clone locally.

```
git clone https://github.com/<your-username>/nginx-node-redis.git
cd nginx-node-redis/
```

Key takeaways of this mini-project:

web/server.js → **A basic Node.js app that connects to Redis on port 6379 and displays:**

- Total visits (increments with every page refresh)
- The hostname (*web1* or *web2*) serving the request
- Runs on port 5000

✨ **Enhanced with a visually appealing HTML UI instead of plain text.**

nginx/nginx.conf → **Custom NGINX configuration with:**

- An upstream load balancer pointing to web1:5000 and web2:5000
- Proxy rules to evenly distribute traffic

- Dockerfile that replaces the default NGINX configuration with this custom setup

docker-compose.yml → **Orchestrates all the containers:**

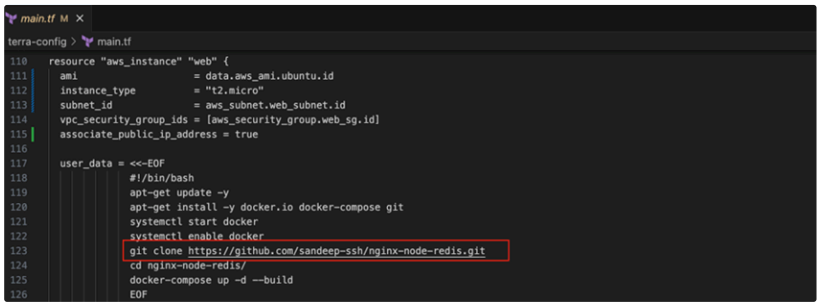
- Redis (database)
- Two Node.js apps (web1 and web2)
- NGINX (reverse proxy and load balancer)

terra-config → **Infrastructure as a Code**

- This is the Terraform setup that creates all the AWS resources needed for the project.
- It builds a VPC, subnet, and security group, launches a t2.micro EC2 instance, Creates a Security Group with port22(SSH) and port80(HTTP) and automatically deploys the Dockerized Node.js + Redis + NGINX app so everything works together seamlessly.

> Tip : In the user_data section of *main.tf*, update the GitHub repo URL to point to **your forked repo**:

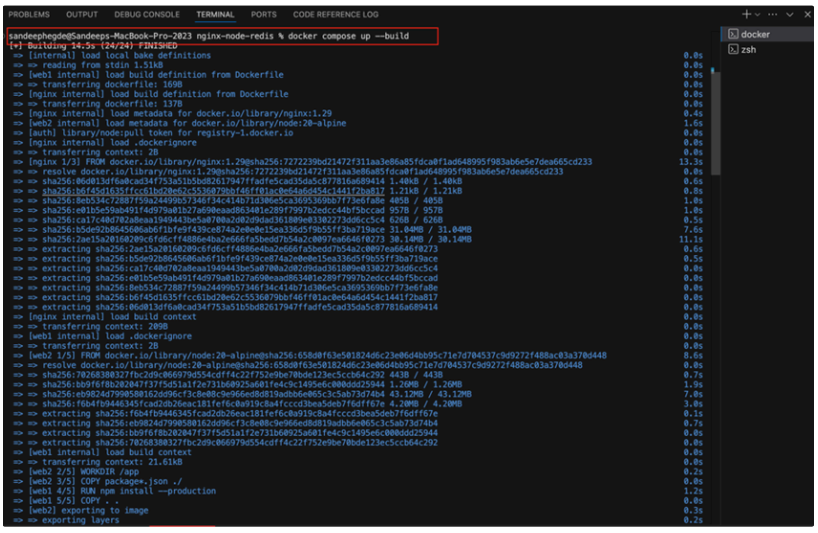
```
git clone https://github.com/<your-github-username>/nginx-node-redis.git
```



Testing Locally with Docker Compose

Navigate to the folder where the source code is for this project and then click on

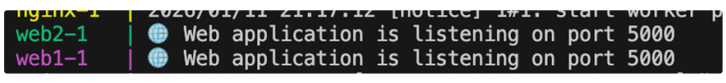
```
docker compose up -build
```



This will:

- Build Docker images for Redis, Node.js web servers, and NGINX
- Start up all the containers in the correct order
- Expose the application through NGINX on port 80

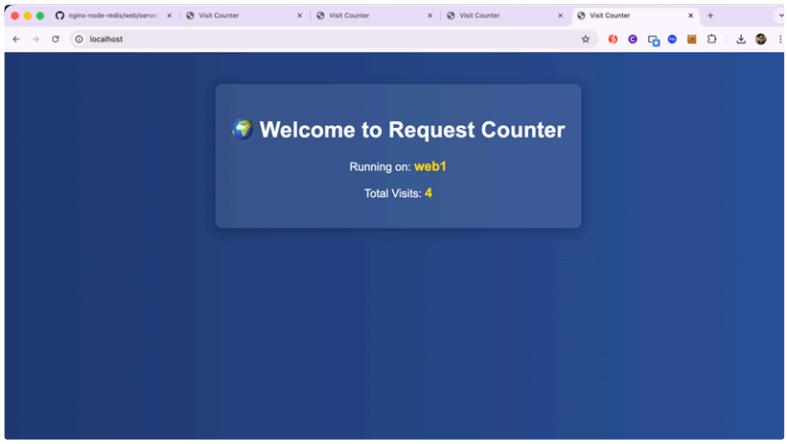
If the set up is running properly – you will be able to see this message



Once done – run this on localhost:80 – you will see the below

To test further – I duplicated the page locally to confirm if the counter works. In the step below -it shows 4 as I loaded the page 4 times on my MAC.

> Tip: You can also refresh this to check the counter.



Step 3 : Integrating GitHub Actions with Terraform

Once the app is running locally, we can automate deployment using GitHub Actions and Terraform. This setup lets us:

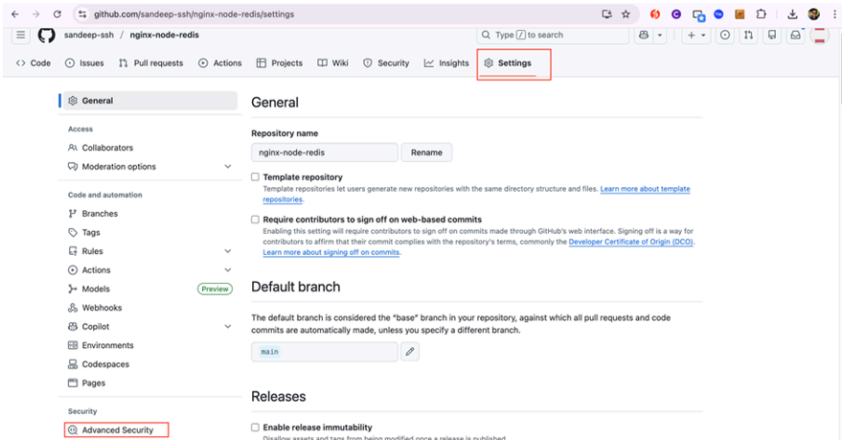
- Automatically provision AWS infrastructure
- Deploy the app to an AWS EC2 instance
- Perform health checks to ensure everything is running
- Tear down resources after testing to save 💰 on AWS bills

Providing Access for GitHub Action to AWS

Now since the application is running locally on our machine – we will proceed to GitHub to enable GitHub actions.

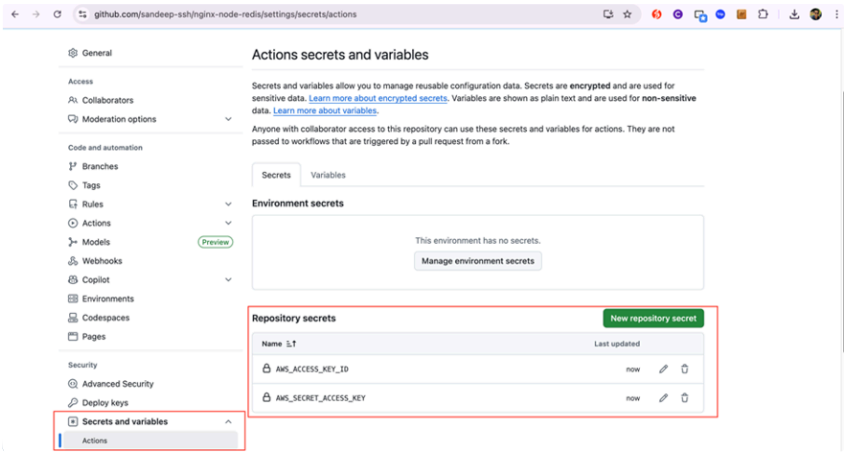
Earlier in Step 1, We generated the AWS CLI credentials. Lets now set this up.

To enable this – go to your *GitHub Repository → Settings →Advanced Security*

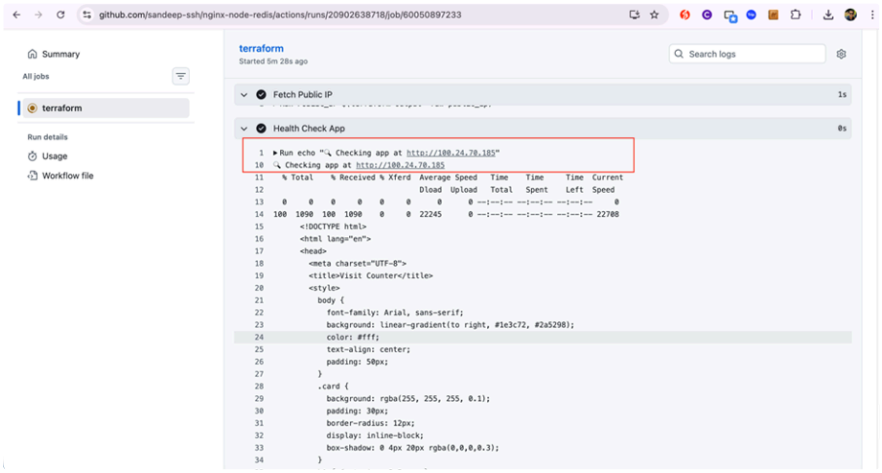


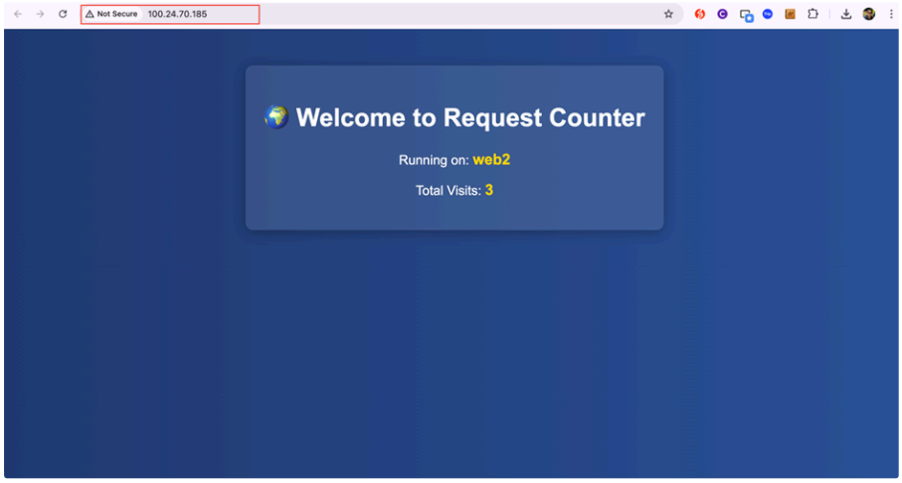
You will now need to select Actions and under Repository Secret – we will need to add

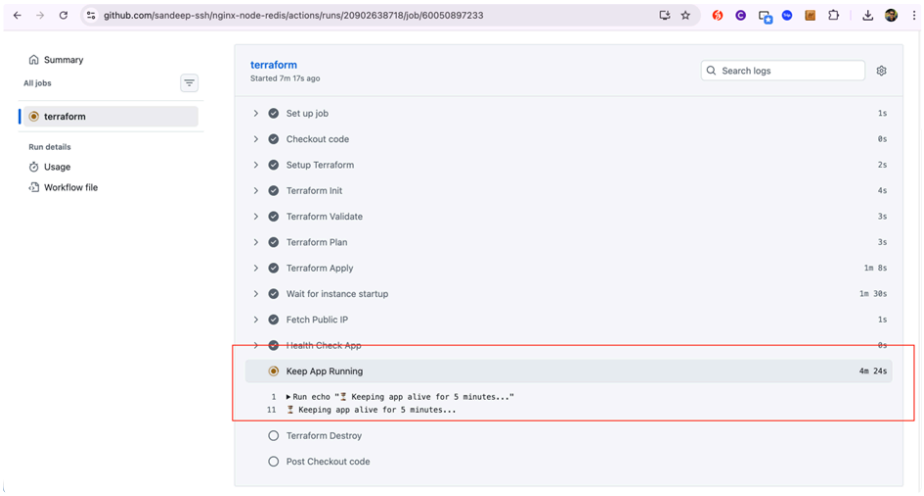
```
AWS_ACCESS_KEY_ID: AKIAZXAYFPDR3E42XXX (sample)
AWS_SECRET_ACCESS_KEY: Lqb7ulPSajZIxsNQd/H8im5323o3U1jpOjnxEuDyOL
(sample)
```

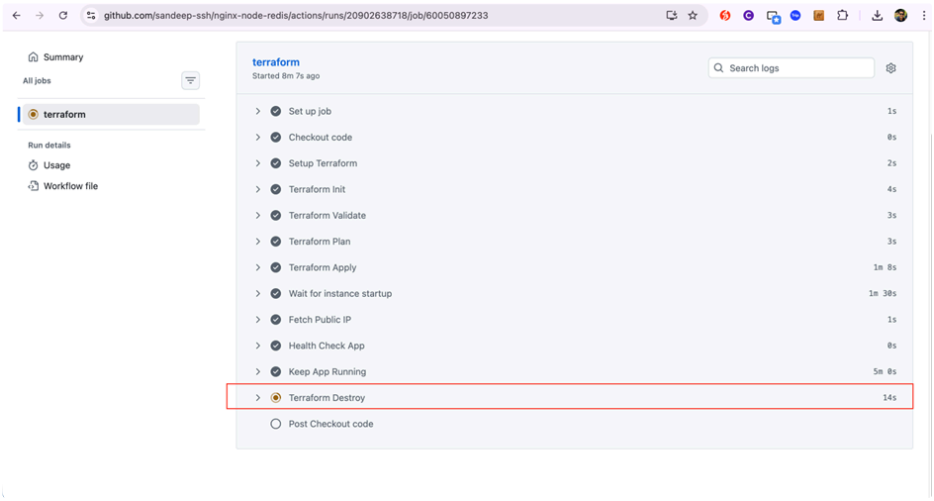

github.com/sandeep-ssh/nginx-node-redis/settings/secrets/actions
General
Access
Collaborators
Moderation options
Code and automation
Branches
Tags
Rules
Actions
Models
Preview
Webhooks
Copilot
Environments
Codespaces
Pages
Security
Advanced Security
Deploy keys
Secrets and variables
Actions
Actions secrets and variables
Secrets and variables allow you to manage reusable configuration data. Secrets are encrypted and are used for sensitive data. Learn more about encrypted secrets. Variables are shown as plain text and are used for non-sensitive data. Learn more about variables.
Anyone with collaborator access to this repository can use these secrets and variables for actions. They are not passed to workflows that are triggered by a pull request from a fork.
Secrets
Variables
Environment secrets
This environment has no secrets.
Manage environment secrets
Repository secrets
New repository secret
Name
Last updated
AWS_ACCESS_KEY_ID
now
AWS_SECRET_ACCESS_KEY
now

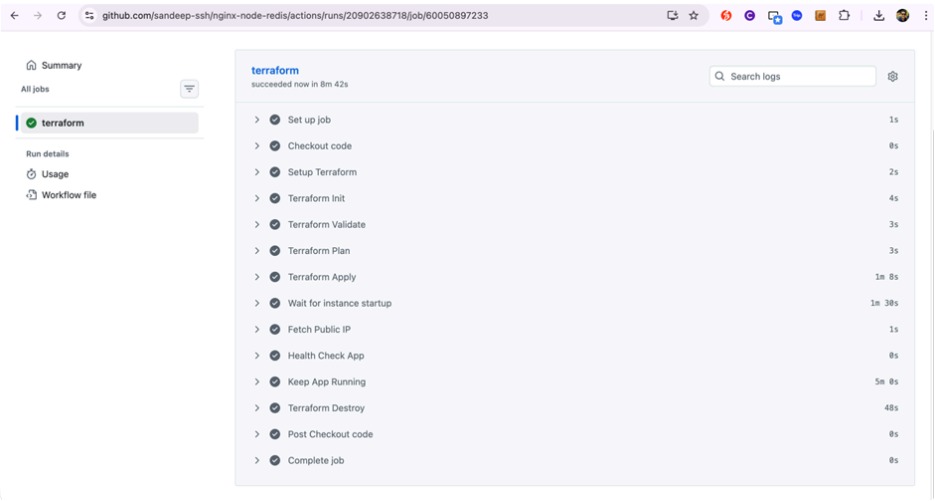
Once it runs –


github.com/sandeep-ssh/nginx-node-redis/actions/runs/20902638718/job/60050897233
Summary
All jobs
terraform
Run details
Usage
Workflow file
terraform
Started 5m 28s ago
Search logs
Fetch Public IP
1s
Health Check App
0s
Run echo "🔍 Checking app at http://100.24.70.185"
🔍 Checking app at http://100.24.70.185
% Total % Received % Xferd Average Speed Time Time Time Current
Dload Upload Total Spent Left Speed
0 0 0 0 0 0 0 0 --:--:-- --:--:-- --:--:-- 0
100 1090 100 1090 0 0 22245 0 --:--:-- --:--:-- --:--:-- 22708
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Visit Counter</title>
<style>
body {
font-family: Arial, sans-serif;
background: linear-gradient(to right, #1e3c72, #2a5298);
color: #fff;
text-align: center;
padding: 50px;
}
.card {
background: rgba(255, 255, 255, 0.1);
padding: 30px;
border-radius: 12px;
display: inline-block;
box-shadow: 0 4px 20px rgba(0,0,0,0.3);
}


Not Secure
100.24.70.185
🌍 Welcome to Request Counter
Running on: web2
Total Visits: 3

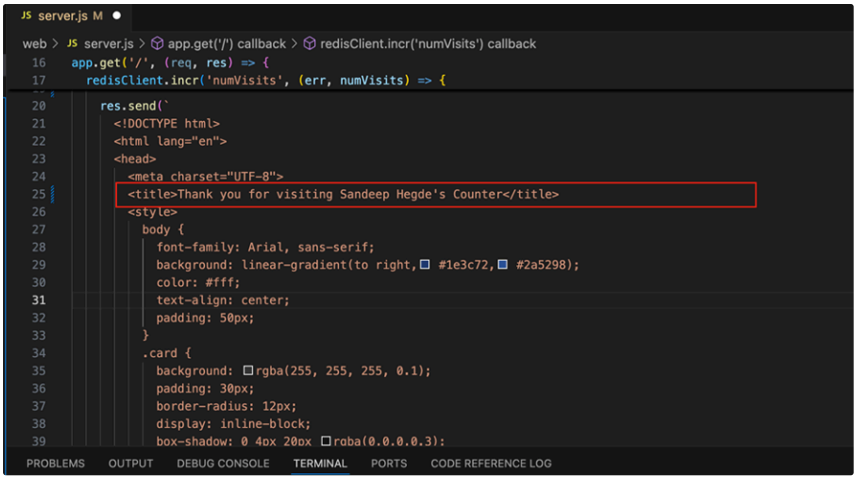




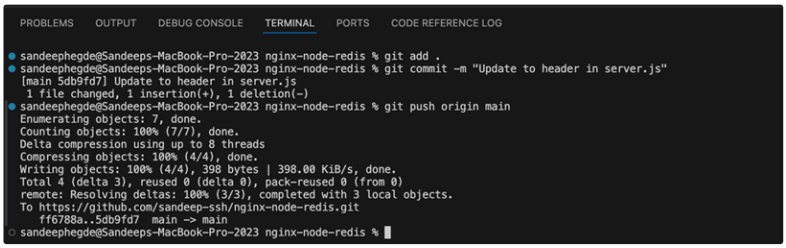


Step 4: Real world change: Updating app due to a change

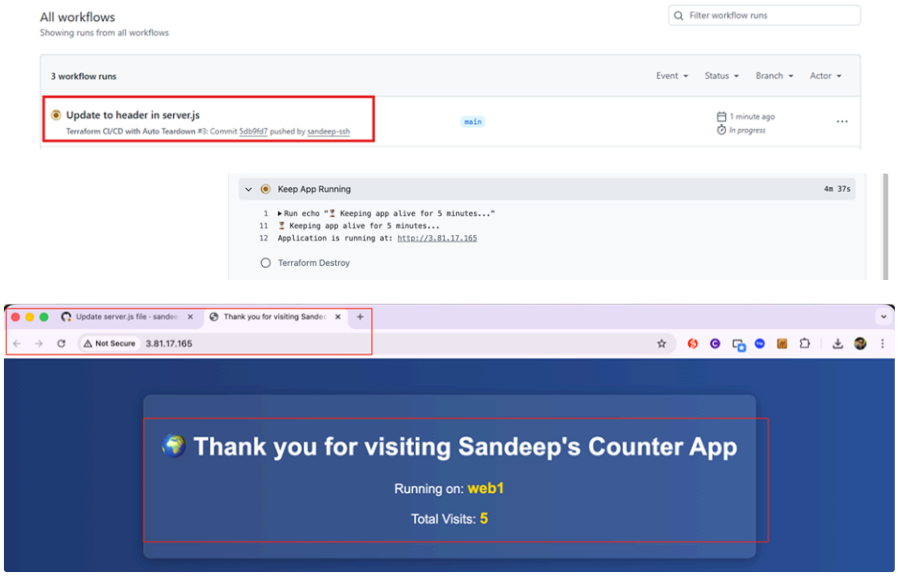
I've made a real-world change to the *server.js* file in my GitHub repo to demonstrate that once changes are made and pushed to Github – Github actions along with Terraform will push these changes live.



After pushing the update to GitHub → Github actions will automatically pre-process the change and it will show the below



Now when we visit GitHub actions :



In this project, we took a simple **Node.js + Redis counter application** and supercharged it with **Nginx, Docker, Terraform, and GitHub Actions**.

What started as a local demo quickly transformed into a **cloud-ready, automated CI/CD pipeline**:

- 🚀 **Docker + Docker Compose** handled local testing and containerization
- ⚡ **Terraform** provisioned AWS infrastructure seamlessly
- 🔄 **GitHub Actions** automated deployments and teardown, giving us a clean, cost-effective workflow
- 🎯 And most importantly — we saw how easy it is to make real-time changes that go live with just a git push.

This mini project proves how **infrastructure as code + automation** can save developers hours of repetitive work and make production-ready workflows more reliable.

🚀 Future Enhancements

This project is intentionally kept simple to highlight core DevOps principles. However, it can be extended in several real-world directions to make it more production-grade and scalable:

☁️ Infrastructure & Scalability

- Migrate from a single EC2 instance to **Auto Scaling Groups**
- Introduce an **Application Load Balancer (ALB)** in front of NGINX
- Add **Route 53** for DNS-based routing
- Move Redis to **Amazon ElastiCache** for managed availability

🐳 Container Orchestration

- Migrate from Docker Compose to **Amazon EKS**
- Use **Kubernetes Deployments & Services** instead of static containers
- Implement **Horizontal Pod Autoscaling (HPA)**

🔐 Security Enhancements

- Use **IAM Roles for EC2** instead of static credentials
- Store secrets securely using **AWS Secrets Manager**
- Enable **VPC endpoints** for private service access
- Add **HTTPS (TLS)** with ACM certificates

🔄 CI/CD Improvements

- Add **Terraform validation & linting** (`tflint` , `terraform fmt`)
- Introduce **environment-based deployments** (dev / staging / prod)
- Implement **blue-green or canary deployments**
- Add approval gates for production changes

📊 Observability & Monitoring

- Integrate **CloudWatch logs & metrics**
- Add **application-level monitoring** (Prometheus / Grafana)
- Implement **health checks & alerting**
- Track request latency and error rates

💰 Cost Optimization

- Use **Spot Instances** for non-production workloads
- Automatically destroy environments after test completion
- Add cost visibility using **AWS Cost Explorer**

🧪 Testing & Quality

- Add **unit and integration tests** for the Node.js app
- Implement **container image scanning**
- Add **security scanning** in CI/CD pipeline

Why This Matters

These enhancements demonstrate how a simple demo can evolve into a **production-grade cloud platform**, mirroring how real-world systems grow over time.

> *This roadmap shows not just what works today, but how to scale responsibly tomorrow.*

📘 Lessons Learned

Building this project provided hands-on exposure to real-world DevOps and cloud engineering practices. Key takeaways include:

🏗️ Infrastructure as Code (Terraform)

- Defining infrastructure declaratively ensures **repeatability and consistency**
- Terraform plans act as a **safety net** before making cloud changes
- Small misconfigurations (VPCs, subnets, security groups) can have big impact
- `user_data` is a powerful way to bootstrap infrastructure automatically

🐳 Containers & Docker

- Containerization makes applications **portable and environment-independent**
- Docker Compose is ideal for **local development and testing**
- Clear service naming simplifies networking between containers
- Layered Docker images help optimize build performance

🔄 CI/CD with GitHub Actions

- Automation reduces human error and speeds up deployments
- Secrets management is critical when integrating CI/CD with cloud providers
- Even simple pipelines can deliver **production-grade workflows**
- A single `git push` can safely trigger infrastructure and app changes

☁️ AWS & Cloud Architecture

- Starting simple (single EC2 + NGINX) avoids unnecessary complexity
- Understanding VPCs and security groups is foundational
- Cost awareness is just as important as functionality
- Not every workload needs managed services or Kubernetes

🧠 System Design & Reliability

- Stateless applications scale more easily
- Shared state (Redis) enables horizontal scaling
- Reverse proxies improve traffic handling and resilience
- Designing for failure early leads to better architectures

💡 DevOps Mindset

- Infrastructure should be treated like application code
- Automation is not optional—it's essential
- Observability and monitoring should be planned early
- Simplicity is often the best design choice

🌟 Why This Project Stands Out

This project goes beyond a basic demo by showcasing **real-world DevOps and cloud engineering practices**:

- Demonstrates **end-to-end automation** — from local development to cloud deployment
- Uses **Infrastructure as Code (Terraform)** instead of manual AWS setup
- Integrates **CI/CD with GitHub Actions**, mirroring modern engineering workflows
- Balances **simplicity and scalability** without over-engineering
- Emphasizes **cost awareness**, security, and operational discipline
- Designed with the **AWS Well-Architected Framework** in mind

---

🏁 Final Reflection

> *This project reinforced the idea that strong DevOps practices are less about tools and more about mindset—automation, reliability, and continuous improvement.*