techsavvyrc.com

AWS EC2 Web Hosting

ravi.chandran@techsavvyrc.com

July 11, 2025

Table of Contents

1 Overview

This documentation provides a step-by-step guide for deploying a static personal or portfolio website on an AWS EC2 instance using Terraform. The deployment process follows Infrastructure-as-Code (IaC) principles to automate the provisioning of cloud resources, configuration of Nginx as the web server, and synchronization of website content directly from a public GitHub repository.

The setup is designed to offer a lightweight and cost-effective solution for hosting personal websites without relying on traditional web hosting platforms. It also includes optional enhancements such as integrating a custom domain and enabling HTTPS with free SSL certificates via Cloudflare. This approach demonstrates how cloud infrastructure, version control, and automation can be combined to create a professional and scalable web hosting environment suitable for developers, students, and tech professionals showcasing their work online.

2 Prerequisite

2.1 Access to AWS Account

- You must have access to the AWS Management Console.
- A dedicated IAM user should be created for remote access and development purposes, rather than using the root account.

2.2 Terraform

- Terraform must be installed and properly configured on your local system.
- If Terraform is not already installed, refer to the official installation guide:
  https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli

2.3 Code Editor

- Any code editor can be used to work with the Terraform scripts and website files.
- Visual Studio Code is recommended for ease of use and community support.
  To get started: https://code.visualstudio.com/docs/getstarted/getting-started

2.4 Website Package or Template

- If you are proficient in front-end development, you can build your website manually using HTML, CSS, JavaScript, and other relevant technologies.
- Alternatively, you may download free website templates from trusted sources online. These templates typically include prebuilt HTML, CSS, and JavaScript files.
- Customize the template by editing the existing files to suit your personal or professional website requirements.
- Upload your modified website files to a public GitHub repository. This repository will later be referenced by the Terraform deployment script.

3 AWS CLI

3.1 Install or Update AWS CLI

- Download the AWS CLI installer for Windows (64-bit):
  https://awscli.amazonaws.com/AWSCLIV2.msi
- To install via command line:
  ```
  C:\> msiexec.exe /i https://awscli.amazonaws.com/AWSCLIV2.msi
  ```
- For a silent installation (no user prompts):
  ```
  C:\> msiexec.exe /i https://awscli.amazonaws.com/AWSCLIV2.msi /qn
  ```

3.2 Configure AWS CLI

- After installation, configure your AWS CLI using your IAM credentials:

  ```
  C:\> aws configure
  AWS Access Key ID [None]: XXXXXXXXXXXXXXXX
  AWS Secret Access Key [None]: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  Default region name [None]: eu-north-1
  Default output format [None]: json
  ```

3.3 Verify Installation

- Run the following command to check if AWS CLI is correctly installed:
  ```
  C:\> aws --version
  ```

  Expected output:
  ```
  aws-cli/2.19.1 Python/3.11.6 Windows/10 exe/AMD64 prompt/off
  ```

  If aws is not recognized, restart the command prompt or refer to the AWS CLI troubleshooting documentation.

- To confirm that the AWS CLI is correctly connected to your AWS account:
  ```
  aws iam get-user
  ```

  Example output:
  ```
  {
      "User": {
          "Path": "/",
          "UserName": "user01",
          "UserId": "xxxxxxxxxx",
          "Arn": "arn:aws:iam::xxxxxxxx6:user/user01",
          "CreateDate": "2025-07-09T12:47:15+00:00",
          "PasswordLastUsed": "2025-07-09T13:11:25+00:00",
          "Tags": [
              {
                  "Key": "AXXXXXXXXXXXXXXXXXXX",
                  "Value": "Terraform User"
              }
          ]
      }
  }
  ```

4 TERRAFORM CONFIGURATION

This section outlines the structure and function of the individual Terraform configuration files that collectively automate the provisioning of the EC2 instance and deployment of the static website. These files work together as modular building blocks to ensure a maintainable, scalable, and declarative infrastructure-as-code (IaC) approach.

4.1 providers.tf

The providers.tf file configures the AWS provider used by Terraform. It specifies the target AWS region and the CLI profile under which the resources will be provisioned. These values are parameterized using input variables to allow flexibility across different environments or AWS accounts.

```
############################################################
# File: providers.tf
#
# Description:
# This file configures the primary cloud provider used by
# the Terraform project. In this case, it sets up the AWS
# provider by specifying the region and credentials profile.
#
# Purpose:
# - To initialize the AWS provider required to provision
#   and manage cloud infrastructure.
# - The region and profile used are externalized as input
#   variables for portability and environment-based flexibility.
#
# Contribution to Overall Setup:
# This is a foundational configuration file. Without it,
# Terraform cannot interact with the AWS APIs to provision
# resources such as EC2 instances, VPCs, or S3 buckets.
#
# Best Practices:
# - Avoid hardcoding credentials.
# - Always externalize environment-specific values (e.g.,
#   region, profile) to `variables.tf`.
# - Use separate AWS profiles for production and staging
#   to enforce environment separation.
############################################################

# Configure the AWS provider with the specified region and profile
provider "aws" {
  region  = var.aws_region    # AWS region where resources will be provisioned (e.g., eu-
north-1)
  profile = var.aws_profile   # Named AWS CLI profile used for authentication
}
```

4.2 variables.tf

The variables.tf file defines all configurable parameters used across the Terraform project. This includes AWS region, profile name, EC2 instance type, AMI ID, key pair settings, SSH key paths, and the GitHub repository containing the static website code. Default values are provided to simplify initial usage, while still allowing overrides via terraform.tfvars or CLI inputs.

```
############################################################
# File: variables.tf
#
# Description:
# This file defines all input variables used across the
# Terraform project. These variables externalize critical
# values like AWS region, instance type, AMI ID, key paths,
# and GitHub repository URL.
#
# Purpose:
# - To parameterize the infrastructure setup for flexibility.
# - To allow customization across environments (e.g., dev, prod).
#
# Contribution to Overall Setup:
# This file makes the code reusable and environment-agnostic
# by decoupling hardcoded values from resource definitions.
#
# Best Practices:
# - Use descriptive variable names and include helpful
#   descriptions.
# - Store secrets (e.g., private keys) securely and avoid
#   committing sensitive values to version control.
# - Set defaults for development, but override via CLI or
#   workspace-specific `*.tfvars` files for production.
############################################################

# AWS region where the infrastructure will be deployed
variable "aws_region" {
  description = "AWS region to deploy resources"
  type        = string
  default     = "<your_aws_region>"
}

# Named AWS CLI profile used for Terraform authentication
variable "aws_profile" {
  description = "AWS CLI profile for Terraform"
  type        = string
  default     = "default"
}

# EC2 instance type to provision (Free Tier eligible: t3.micro or t2.micro)
variable "instance_type" {
  description = "EC2 instance type"
  type        = string
```

```
  default     = "t3.micro"
}

# Amazon Machine Image (AMI) ID for Amazon Linux 2023
variable "ami_id" {
  description = "AMI ID for Amazon Linux 2023"
  type        = string
  default     = "ami-00c8ac9147e19828e"
}

# Name of the EC2 Key Pair to associate with the instance
variable "key_name" {
  description = "Name of the EC2 key pair"
  type        = string
  default     = "<your_key_name>"
}

# GitHub repository URL from which the EC2 instance will clone the website
variable "github_repo" {
  description = "GitHub repo URL for website content"
  type        = string
  default     = "<your_git_repository_link>"
}

variable "ec2_instance_name" {
  description = "Name of the EC2 instance"
  type        = string
  default     = "<your_key_name>"
}
```

4.3 keypair.tf

This file provisions an SSH key pair that is used to securely access the EC2 instance. A new RSA private key is generated using the tls_private_key resource, and both the private and public keys are stored locally. The public key is then uploaded to AWS using the aws_key_pair resource to associate it with the EC2 instance. This ensures secure and automated access to the server.

```
#############################################################
# File: keypair.tf
#
# Description:
# This file manages the generation and provisioning of SSH
# key pairs used to securely access the EC2 instance.
#
# Purpose:
# - Generates a new RSA key pair (4096-bit).
# - Writes the private and public keys to local files.
# - Registers the public key with AWS as a Key Pair so it
#   can be attached to EC2 instances.
```

```
#
# Contribution to Overall Setup:
# Enables secure SSH access to EC2 instances for setup and
# troubleshooting. The private key can later be converted
# to `.ppk` for PuTTY if needed.
#
# Best Practices:
# - Keys are written to paths defined in `variables.tf`.
# - Use appropriate file permissions (`0600` for private).
# - DO NOT commit the generated keys to version control.
############################################################

# Generate a new RSA 4096-bit private/public SSH key pair
resource "tls_private_key" "ssh_key" {
  algorithm = "RSA"
  rsa_bits  = 4096
}

# Store the private key securely on the local machine
resource "local_file" "private_key" {
  content         = tls_private_key.ssh_key.private_key_pem
  filename        = "${path.module}/.ssh/<your_private_key_name>.pem"
  file_permission = "0600"
}

# Store the public key on the local machine
resource "local_file" "public_key" {
  content         = tls_private_key.ssh_key.public_key_openssh
  filename        = "${path.module}/.ssh/<your_public_key_name>.pub"
  file_permission = "0644"
}

# Register the public key with AWS EC2 as a named Key Pair
resource "aws_key_pair" "deployer" {
  key_name   = var.key_name
  public_key = tls_private_key.ssh_key.public_key_openssh
}
```

4.4 security_group.tf

The security_group.tf file defines the security group associated with the EC2 instance. It allows inbound access on ports 22 (SSH), 80 (HTTP), and 443 (HTTPS) from any source IP. Outbound traffic is allowed to all destinations. This configuration is sufficient to enable web traffic and administrative access for a basic public-facing website hosted on EC2.

```
############################################################
# File: security_group.tf
#
# Description:
# Defines the AWS Security Group used by the EC2 instance
```

```
# hosting the static web application.
#
# Purpose:
# - Controls inbound (ingress) and outbound (egress) traffic
#   to the EC2 instance.
# - Opens necessary ports for SSH (22), HTTP (80), and HTTPS (443).
#
# Contribution to Overall Setup:
# This resource ensures that:
# - You can securely SSH into the instance.
# - Web traffic (HTTP/HTTPS) is allowed from the internet.
# - The instance can make outbound calls if needed (e.g., for package installs).
#
# Best Practices:
# - Restrict SSH (`port 22`) to known IP ranges in production.
# - Keep rules tightly scoped for enhanced security.
################################################################

# Define a security group to allow SSH, HTTP, and HTTPS traffic
resource "aws_security_group" "web_sg" {
  name        = "<your_security_group_name>"
  description = "Allow SSH, HTTP, HTTPS"

  # Allow incoming SSH traffic (port 22) from all IPs
  # NOTE: Restrict in production to specific IPs
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  # Allow incoming HTTP traffic (port 80) for website access
  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  # Allow incoming HTTPS traffic (port 443) for secure access
  ingress {
    from_port   = 443
    to_port     = 443
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  # Allow all outbound traffic
  # Necessary for updates, git clone, etc.
```

```
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

4.5 ec2_instance.tf

This file provisions the actual EC2 instance and includes a detailed user_data script that automates the entire server configuration. The script performs a system update, installs Nginx and Git, clones the static website from the specified GitHub repository, configures the Nginx web server, and starts the service. Additional Nginx configuration is embedded to add caching, security headers, compression, and other best practices for static site performance and security.

```
############################################################
# File: ec2_instance.tf
#
# Description:
# Defines the main EC2 instance used to host the Static
# static website using NGINX on Amazon Linux 2023.
#
# Purpose:
# - Provisions an EC2 instance with appropriate configuration.
# - Installs and configures NGINX with custom headers and caching.
# - Automatically clones the static website from GitHub repo.
#
# Contribution to Overall Setup:
# This is the compute resource running the production website.
# Bootstrap is handled via `user_data`, ensuring zero manual steps.
#
# Best Practices:
# - Uses variables for flexibility.
# - Includes tags for easier identification.
# - Automates full provisioning using `user_data`.
# - NGINX config is declared inline within EC2 instance creation.
############################################################

# Define the EC2 instance resource
resource "aws_instance" "web" {
  # Use the specified AMI (Amazon Linux 2023)
  ami = var.ami_id

  # EC2 instance type, default is t3.micro (Free Tier eligible)
  instance_type = var.instance_type

  # Associate the EC2 Key Pair created by Terraform
  key_name = aws_key_pair.deployer.key_name
  #key_name = "<your_existing_key_name>"
```

```
  # Attach security group that allows SSH, HTTP, and HTTPS
  vpc_security_group_ids = [aws_security_group.web_sg.id]
  #vpc_security_group_ids = ["sg-0ee5f6f626e80c130"]

  # Inline user_data script to fully bootstrap the EC2 instance
  user_data = <<-EOF
    #!/bin/bash

    # 1. System update and install packages
    dnf update -y
    dnf install -y nginx git

    # 2. Configure web root directory
    mkdir -p /var/www/html
    chown -R nginx:nginx /var/www
    chmod -R 755 /var/www

    # 3. Deploy website from GitHub (clone if first time, pull otherwise)
    if ! git clone ${var.github_repo} /var/www/html; then
      cd /var/www/html && git pull origin main
    fi

    # 4. Write custom NGINX configuration with security and performance settings
    cat > /etc/nginx/conf.d/<your_ngnix_file_name>.conf << 'NGINX_CONF'
    server {
        listen 80;
        server_name _;

        root /var/www/html;
        index index.html;

        # Security headers
        add_header X-Frame-Options "SAMEORIGIN" always;
        add_header X-Content-Type-Options "nosniff" always;
        add_header Referrer-Policy "strict-origin-when-cross-origin" always;
        add_header Permissions-Policy "geolocation=(),midi=(),sync-
xhr=(),microphone=(),camera=(),magnetometer=(),gyroscope=(),fullscreen=(self),payment=()"
always;

        # Compression
        gzip on;
        gzip_types text/plain text/css application/json application/javascript text/xml
application/xml application/xml+rss text/javascript;
        gzip_min_length 1024;
        gzip_proxied any;
        gzip_comp_level 5;
        gzip_vary on;

        # Static file caching
```

```
        location ~* \.(?:css|js|jpe?g|png|gif|ico|svg|woff2?)$ {
            expires 30d;
            add_header Cache-Control "public, no-transform";
            access_log off;
            log_not_found off;
        }

        location / {
            try_files $uri $uri/ =404;
            add_header Content-Security-Policy "default-src 'self'; script-src 'self'
'unsafe-inline'; style-src 'self' 'unsafe-inline'; img-src 'self' data:;" always;
        }

        # Security settings
        server_tokens off;
        client_max_body_size 1m;
        client_body_buffer_size 16k;
        client_header_buffer_size 1k;

        # Logging
        access_log /var/log/nginx/access.log;
        error_log /var/log/nginx/error.log warn;
    }
    NGINX_CONF

    # 5. Validate and start NGINX
    nginx -t && systemctl enable nginx && systemctl restart nginx
  EOF

  # Resource tagging for identification and filtering
  tags = {
    Name = var.ec2_instance_name
  }
}
```

4.6 outputs.tf

The outputs.tf file defines output variables that display useful information after the Terraform deployment completes. Specifically, it reveals the public IP address of the EC2 instance and a pre-formatted SSH command that the user can execute to connect to the server securely. These outputs streamline post-deployment access and verification.

```
#############################################################
# File: outputs.tf
#
# Description:
# This file defines Terraform output variables that expose
# useful information after infrastructure provisioning is complete.
#
# Purpose:
```

```
# - Display the public IP of the deployed EC2 instance.
# - Provide the exact SSH command to access the instance securely.
#
# Contribution to Overall Setup:
# This file is essential for quickly retrieving access details
# without having to manually inspect AWS console or Terraform state.
#
# Best Practices:
# - Descriptive names and explanations.
# - Clear guidance for SSH access using generated private key.
# - Outputs simplify post-deploy troubleshooting and operations.
############################################################

# Output the public IP address of the EC2 instance for browser/SSH access
output "public_ip" {
  description = "Public IP of the web server"
  value       = aws_instance.web.public_ip
}

output "ssh_command" {
  description = "SSH command to connect via PuTTY/OpenSSH"
  value       = "ssh -i ${path.module}/.ssh/<your_private_key_name>.pem ec2-
user@${aws_instance.web.public_ip}"
  sensitive = true
}
```

4.7 Summary of Infrastructure Provisioning

The Terraform configuration described above performs the following actions:

- Provisions a new EC2 instance based on Amazon Linux 2023
- Installs required packages including Nginx and Git using an inline user_data script
- Clones a static website from a public GitHub repository into /var/www/html
- Configures and starts the Nginx web server to serve the website
- Applies basic Nginx best practices including compression, caching, and security headers
- Outputs the instance's public IP and an SSH command for direct access

Sample Terraform code for this setup is publicly available at: https://github.com/TechSavvyRC/aws-static-site-deploy.git

You can fork or clone the repository and update the variables/configuration to match your setup.

5 Deploying Infrastructure Using Terraform

Once all Terraform configuration files are prepared and tailored to your environment, follow these steps to deploy the infrastructure on AWS:

Step 1: Initialize Terraform

- This command initializes the working directory containing Terraform configuration files. It downloads necessary provider plugins and prepares the directory for deployment.

```
terraform init
```

Step 2: Review the Execution Plan

- This step helps verify what resources will be created or modified before applying the changes.

```
terraform plan
```

Step 3: Apply the Terraform Configuration

- This command provisions the actual resources defined in the Terraform configuration. It will prompt you to confirm the action.

```
terraform apply
```

You will see output variables (like the EC2 public IP and SSH command) after successful execution.

Step 4: Verify Deployment in Browser

- After the infrastructure is successfully created and the EC2 instance is running, verify that the static website is live and served correctly via Nginx.
- Open a web browser and enter the EC2 public IP in the address bar:

```
http://<your-ec2-public-ip>
```

You should see your personal website as hosted from the EC2 instance. If the site does not load, ensure:

- The EC2 instance is in the running state.
- Port 80 (HTTP) is open in the security group.
- Nginx is running correctly on the instance.
- Your GitHub repository is public and accessible.

Step 5: Connect via SSH (Optional)

- To log into the EC2 instance:

```
ssh -i <path-to-your-private-key.pem> ec2-user@<your-ec2-public-ip>
```

Step 6: Destroy the Infrastructure (Optional)

- When you no longer need the deployed infrastructure, run the following to delete all resources:

```
terraform destroy
```

This helps avoid unintended AWS charges.

6 Domain and DNS Configuration

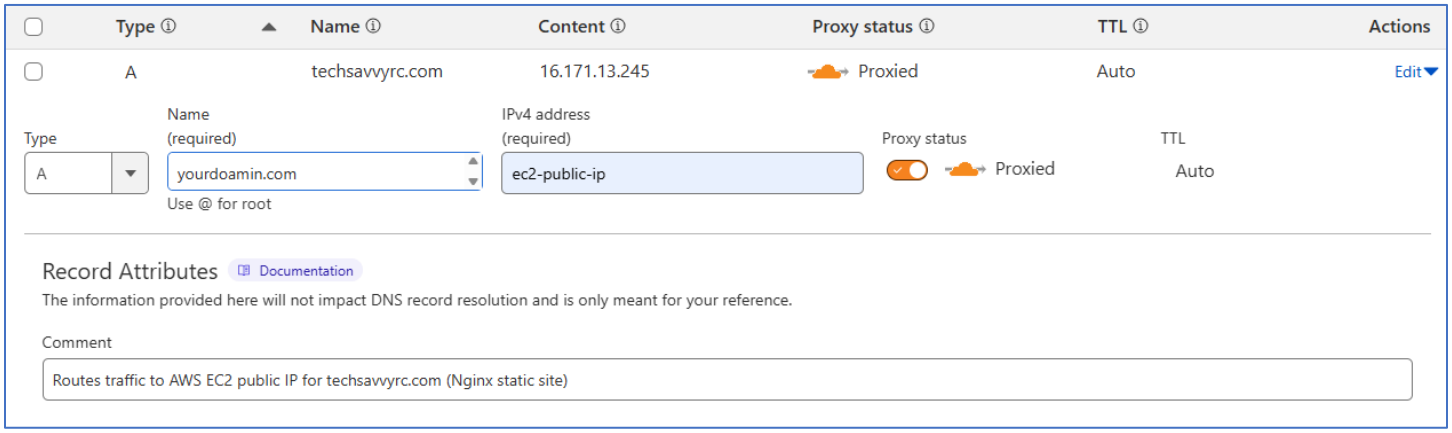
If you have purchased a custom domain (e.g., from Hostinger, GoDaddy, Namecheap, etc.), you can link it to your EC2 instance to make your website accessible using a branded URL like www.yourdomain.com.

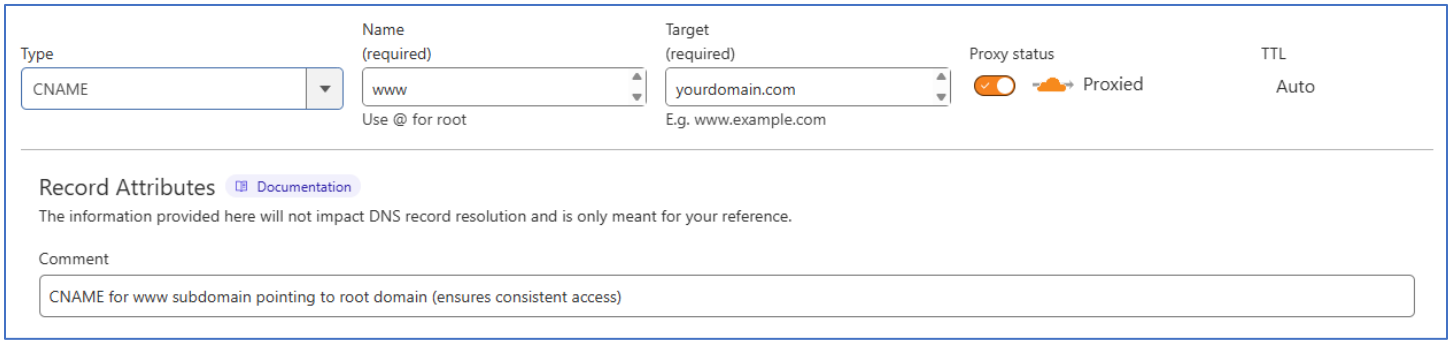
6.1 Update DNS Records

Regardless of your domain registrar, the DNS configuration process generally involves adding an A Record:

DNS Record Type	Name	Value	TTL	Description
A	@	<your-ec2-public-ip>	Auto	Maps root domain to EC2 instance
CNAME	www	<your-ec2-public-ip>	Auto	Maps www subdomain to EC2 instance

- Replace <your-ec2-public-ip> with the actual public IP of your EC2 instance.
- These records ensure that when users visit your domain, they are directed to your hosted website.
- DNS changes can take up to 24–48 hours to fully propagate, but usually work within a few minutes to a few hours.





6.2 SSL Certificate (HTTPS Support)

- Some domain providers (like Namecheap or Hostinger) may offer free or paid SSL certificates.
- If your provider does not offer free SSL, you can use Cloudflare to manage DNS and enable free SSL with HTTPS:
  - Create a free Cloudflare account at https://cloudflare.com
  - Add your domain to Cloudflare and follow the instructions to update your domain registrar's nameservers to point to Cloudflare.
  - Once DNS is managed through Cloudflare:
    → Go to SSL/TLS settings and enable:
    - Flexible or Full SSL
    - Always Use HTTPS
    - Automatic HTTPS Rewrites

Your site will now support secure https:// access with a free SSL certificate.

7 Important Considerations and Cautions

7.1 Free SSL and Public IP Caution

While Cloudflare's free SSL and DNS services offer convenience and security for personal or portfolio websites, they are not a substitute for enterprise-grade security. Publicly exposing your EC2 instance's IP address may leave it vulnerable to unauthorized access. This configuration is not suitable for production systems or sensitive applications.

7.2 Monitor AWS Billing Regularly

Even though the AWS Free Tier provides cost-effective resources like t3.micro instances, it's important to actively monitor your AWS billing dashboard. Unexpected charges can arise due to:

- Region-specific pricing
- Data transfer costs
- Manual configuration errors
- Exceeding free tier limits

Regularly check the AWS Billing Console to stay informed and prevent cost overruns.

7.3 Manual Cleanup

If you're testing or no longer need the deployed infrastructure, be sure to destroy it manually using Terraform. Idle resources like EC2 instances, EBS volumes, or Elastic IPs can incur charges over time. Use:

```
terraform destroy
```

This will clean up all resources provisioned by your Terraform code.

7.4 IAM & Key Security

Use a dedicated IAM user with the minimum required permissions rather than the root account. Always:

- Store .pem SSH keys securely (never commit them to version control)
- Avoid sharing IAM credentials
- Rotate access keys periodically

Neglecting these practices can lead to serious security breaches.

7.5 Domain Visibility

Even with Cloudflare proxy enabled, DNS records may still expose metadata about your hosting configuration. Use the Cloudflare "Proxied" mode to hide your EC2 public IP and enable caching, DDoS protection, and SSL termination. However, keep in mind:

- Some online tools may still reveal underlying IPs via HTTP headers or DNS history
- Full anonymity requires advanced configuration and may not be achievable with basic setups