Terraform Blue-Green Deployment on AWS

DevOps and Cloud Architecture Deliverable

Author: Fekri Saleh

Date: July 3, 2025

aws

Contents

1. Executive Summary

This document presents a comprehensive overview of a Terraform-based cloud infrastructure designed to support blue-green deployments on AWS. The infrastructure enables teams to provision and manage isolated **staging** and **production** environments, ensuring *zero-downtime* deployments, robust security, and automated monitoring.

The project leverages key AWS services such as EC2, RDS, ElastiCache, and ALB, alongside Terraform for infrastructure-as-code and GitHub Actions for continuous integration and delivery (CI/CD). A bootstrap phase initializes secure backend storage using S3 and DynamoDB, while OpenID Connect (OIDC) integration enables GitHub workflows to securely assume roles in AWS without long-lived credentials.

A modular Terraform design improves code reusability and environment isolation. CI/CD pipelines allow on-demand deployments with controlled inputs, environment selection, and destruction capabilities. CloudWatch-based observability, fine-grained IAM policies, and automated role provisioning further enhance the operational posture of the platform.

This deliverable documents the architecture, deployment workflow, automation tooling, monitoring strategy, and security controls applied throughout the project. It is intended for technical reviewers, DevOps engineers, cloud architects, and operations teams responsible for maintaining or extending this infrastructure.

2. Project Scope

The scope of this project is to design, provision, and automate a production-ready cloud infrastructure on AWS using Infrastructure as Code (IaC) principles. The primary focus is to implement a reliable and maintainable blue-green deployment model that supports rapid iteration, continuous delivery, and operational resilience.

This infrastructure is defined entirely using Terraform, with GitHub Actions serving as the CI/CD engine to manage environment lifecycles securely through OpenID Connect (OIDC). The system is designed to accommodate two isolated environments — **staging** and **production** — each deployed in its own Terraform workspace with dedicated network and compute resources.

2.1 In-Scope Deliverables

- Creation of modular Terraform code to provision AWS resources.
- Implementation of a secure remote backend using S3 and DynamoDB.
- Deployment of scalable infrastructure components, including:
  - VPC with public/private subnets, NAT, and route tables.
  - EC2 instances via Auto Scaling Groups (ASG).
  - Application Load Balancer (ALB) for traffic routing.
  - RDS (MySQL) and ElastiCache (Redis) with Multi-AZ support.
  - CloudWatch for logging, metrics, dashboards, and alarms.
- Secure GitHub Actions workflow with OIDC authentication.
- Automated environment selection and deployment through CI/CD.
- Monitoring and alerting setup using CloudWatch and SNS.
- Resource tagging, parameterization, and best-practice security policies.

2.2 Out-of-Scope Items

- Application-level deployment logic (e.g., container images, code deployments).
- Ingress TLS/SSL termination and ACM/Route53 integration.

- Frontend/CDN configurations such as CloudFront or S3 hosting.
- Post-deployment functional testing or QA pipelines.

3. Architecture Design

The architecture follows a modular and multi-environment design that leverages Terraform workspaces to provision fully isolated **staging** and **production** environments. Each environment includes its own networking, compute, storage, and monitoring resources, enabling safe deployment and rollback strategies via a blue-green approach.

3.1 High-Level System Overview

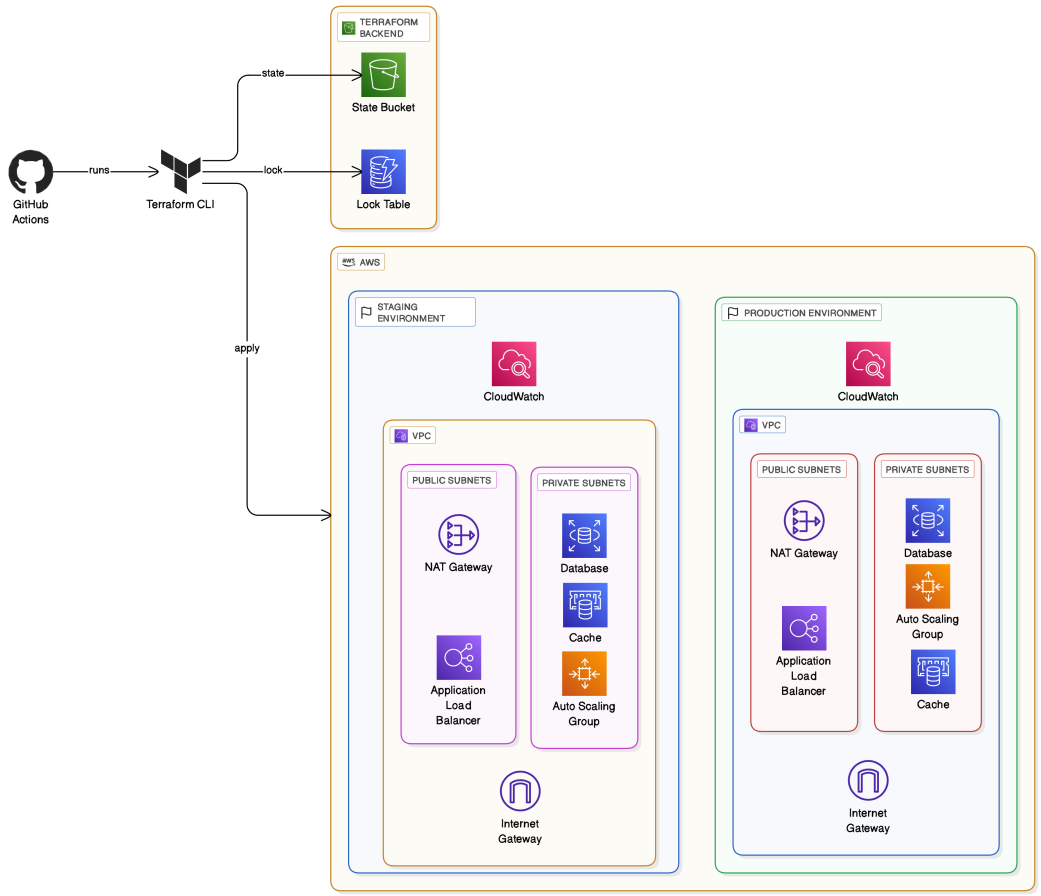

Figure 1: High-Level Architecture for Staging and Production Environments

Each environment is deployed in a dedicated AWS Virtual Private Cloud (VPC) and contains:

- **Public Subnets:** Hosting the Application Load Balancer (ALB) and NAT Gateways.
- **Private Subnets:** Hosting Auto Scaling Groups (EC2), RDS, and Redis resources.
- **Internet Gateway (IGW):** Attached to the public subnets to allow external access.
- **NAT Gateways:** Enable internet-bound traffic from private subnets.
- **Route Tables:** Configured for appropriate routing to IGW or NAT.

3.2 Terraform Module Architecture

The infrastructure is composed of the following reusable Terraform modules:

- **bootstrap:** Initializes the remote state backend (S3 + DynamoDB) and configures IAM for GitHub Actions with OIDC authentication.
- **network:** Provisions VPC, subnets, NAT Gateways, route tables, and IGW.
- **environment:** Creates EC2 Auto Scaling Groups, RDS instances, ElastiCache clusters, and associated IAM roles and profiles.
- **cloudwatch:** Configures metric alarms, dashboards, log groups, and alerting via SNS.

3.3 Deployment Model: Blue-Green Strategy

The blue-green deployment strategy is enabled by provisioning **two isolated environments** and switching traffic at the ALB level or by controlling which environment is deployed at a given time via GitHub Actions input parameters.

This design allows for:

- Testing new infrastructure changes in staging before affecting production.
- Fast rollback in case of errors by re-selecting the stable workspace.
- Independent scaling and monitoring per environment.

The following diagram illustrates the blue-green deployment model with fully isolated infrastructure for staging and production environments. Each environment includes its own VPC, subnets, Auto Scaling Group, RDS and Redis components, NAT gateway, and ALB.

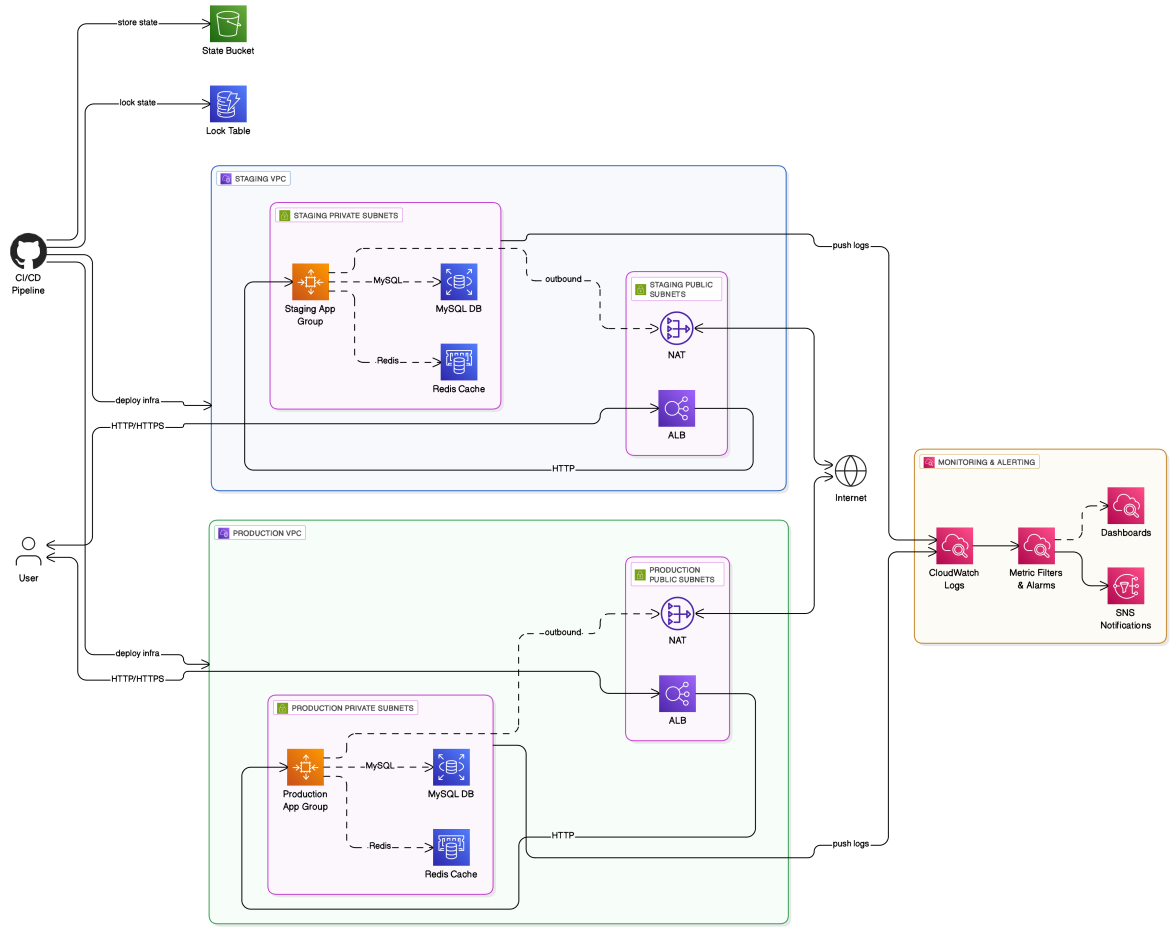

Figure 2: Blue-Green Environment Architecture: Independent Staging and Production VPCs

3.4 Security Controls and IAM Integration

Security is enforced through the use of scoped IAM roles and least-privilege policies:

- GitHub Actions assumes a role via OIDC to avoid hardcoded credentials.
- EC2 instances use instance profiles with access to SSM and CloudWatch only.
- Security groups are created per resource type (ALB, EC2, RDS, Redis) with fine-grained ingress/egress rules.

3.5 Cloud Services Overview

The following AWS services are provisioned in two categories: backend services (used to bootstrap and secure Terraform operations) and environment-specific resources (used to deploy application infrastructure for staging and production).

A. Bootstrap Services (Backend Infrastructure)

These services are initialized via the ‘bootstrap‘ module and only need to be deployed once:

- **Amazon S3:** Stores Terraform remote state files with versioning and encryption (AES256).
- **Amazon DynamoDB:** Implements state locking to prevent concurrent Terraform runs.
- **IAM Roles and Policies:**
  - OIDC trust policy for GitHub Actions to assume roles via ‘sts:AssumeRoleWithWebIdentity‘.
  - Permissions to access EC2, S3, RDS, ElastiCache, Auto Scaling, CloudWatch, and more.
- **OpenID Connect (OIDC) Provider:** Enables GitHub Actions to authenticate to AWS securely without long-lived secrets.

B. Deployment Resources (Per Environment)

Each environment (‘staging‘ and ‘production‘) provisions the following resources using Terraform modules (‘network‘, ‘environment‘, and ‘cloudwatch‘):

- **Amazon VPC:** Custom VPC with public and private subnets, NAT gateways, and routing.
- **Elastic Load Balancer (ALB):** Fronts incoming traffic and routes to EC2 instances using target groups.
- **Amazon EC2 (Auto Scaling):** Application servers launched via Launch Templates and grouped into Auto Scaling Groups with CloudWatch integration.
- **Amazon RDS (MySQL):** Multi-AZ relational database with IAM authentication, backup policies, and subnet groups.
- **Amazon ElastiCache (Redis):** Deployed for caching, managed within private subnets and protected by security groups.
- **Amazon CloudWatch:**
  - Custom log groups for EC2, NGINX, system, Redis, and RDS logs.
  - Metric filters and alarms for application errors, Redis/RDS performance, CPU/memory, and connection metrics.
  - Dashboards per environment.
- **Amazon SNS:** Sends email notifications for triggered CloudWatch alarms.

3.6 Networking and Security

The infrastructure design includes dedicated networking and security controls for each environment (`staging`, `production`), deployed in isolated AWS Virtual Private Clouds (VPCs). The network layout is structured to separate public and private resources while ensuring secure and efficient routing.

A. VPC and Subnet Layout

Each environment provisions its own VPC with the following configurations:

- **CIDR Range:**
  - Staging: `10.0.0.0/16`
  - Production: `10.1.0.0/16`
- **Public Subnets:** For NAT Gateways and Application Load Balancer (ALB)
- **Private Subnets:** For EC2, RDS, and Redis resources
- **Availability Zones:** Resources are spread across `us-east-1a` and `us-east-1b`

B. Routing and Gateways

- **Internet Gateway (IGW):** Attached to each VPC for outbound access from public subnets.
- **NAT Gateways:** One per public subnet, allowing instances in private subnets to access the internet securely.
- **Route Tables:** Separate tables for public and private subnets with appropriate default routes:
  - Public subnets: route to IGW
  - Private subnets: route to respective NAT Gateway

C. Security Groups

Security groups are tightly scoped for each tier of the architecture:

- **ALB Security Group:**
  - Ingress: HTTP (80), HTTPS (443) from `0.0.0.0/0`
  - Egress: All traffic to internet
- **EC2 Security Group:**
  - Ingress: HTTP from ALB security group
  - Egress: Full internet and internal access (for DB, Redis)
- **RDS Security Group:**
  - Ingress: MySQL (3306) from EC2 security group
- **Redis Security Group:**
  - Ingress: Redis (6379) from EC2 security group

D. IAM and Access Control

The project uses AWS Identity and Access Management (IAM) to enforce secure, role-based access:

- **GitHub Actions OIDC Role:** Assumed by CI/CD pipelines via OpenID Connect.

- **EC2 Instance Profile:** Grants instances access to SSM, CloudWatch, and RDS IAM authentication.
- **IAM Policies:** Defined via templated JSON files and attached to roles dynamically through Terraform.

4. Infrastructure as Code (IaC)

This infrastructure is fully defined and managed using Terraform (v1.11.4), following a modular, reusable, and environment-isolated design. Each environment — `staging` and `production` — is deployed via separate Terraform workspaces, ensuring safe experimentation and zero-downtime blue-green deployments.

4.1 Terraform Structure and Modules

The codebase is organized into two main phases:

a) **Bootstrap Phase:** Initializes the backend and IAM trust chain for GitHub Actions.

b) **Deployment Phase:** Provisions all infrastructure components for each environment using shared modules.

A. Bootstrap Phase

Located under the '`bootstrap/`' directory, this step provisions the foundational backend and access layers:

- `module.backend_setup` – Creates:
  - S3 bucket for remote state (versioned and encrypted)
  - DynamoDB table for state locking
- `module.oidc` – Sets up:
  - OpenID Connect provider for GitHub Actions
  - IAM trust policy and permission policy
  - Role assumption via 'sts:AssumeRoleWithWebIdentity'

The figure below illustrates how Terraform initializes backend storage and configures IAM access using GitHub OIDC.

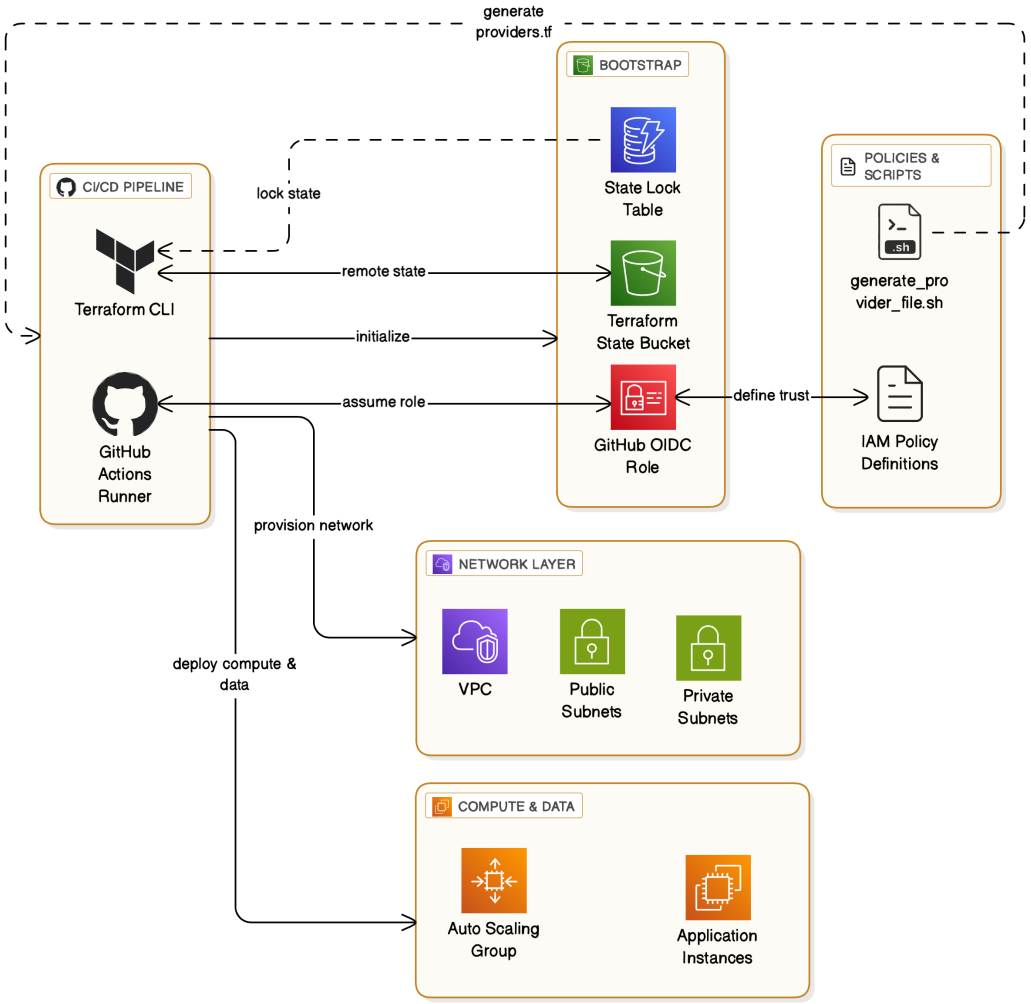

Figure 3: Terraform Bootstrap Flow: Remote Backend and OIDC Trust Establishment

B. Deployment Phase

The main deployment logic is structured using three core modules under '`modules/`' and invoked per workspace:

- **module.network:** Provisions the VPC, subnets, NAT gateways, internet gateway, and routing tables.
- **module.environment:** Manages compute (EC2 ASG), RDS, ElastiCache, IAM roles, launch templates, and Auto Scaling.
- **module.cloudwatch:** Sets up metric alarms, log groups, filters, dashboards, and SNS alerts.

Each module accepts environment-specific variables (e.g., subnet CIDRs, instance types, DB sizes), and output values are passed between modules for composition.

The following diagram provides a complete view of the end-to-end deployment workflow, including the CI/CD pipeline, bootstrap initialization, AWS infrastructure layers, and observability components.

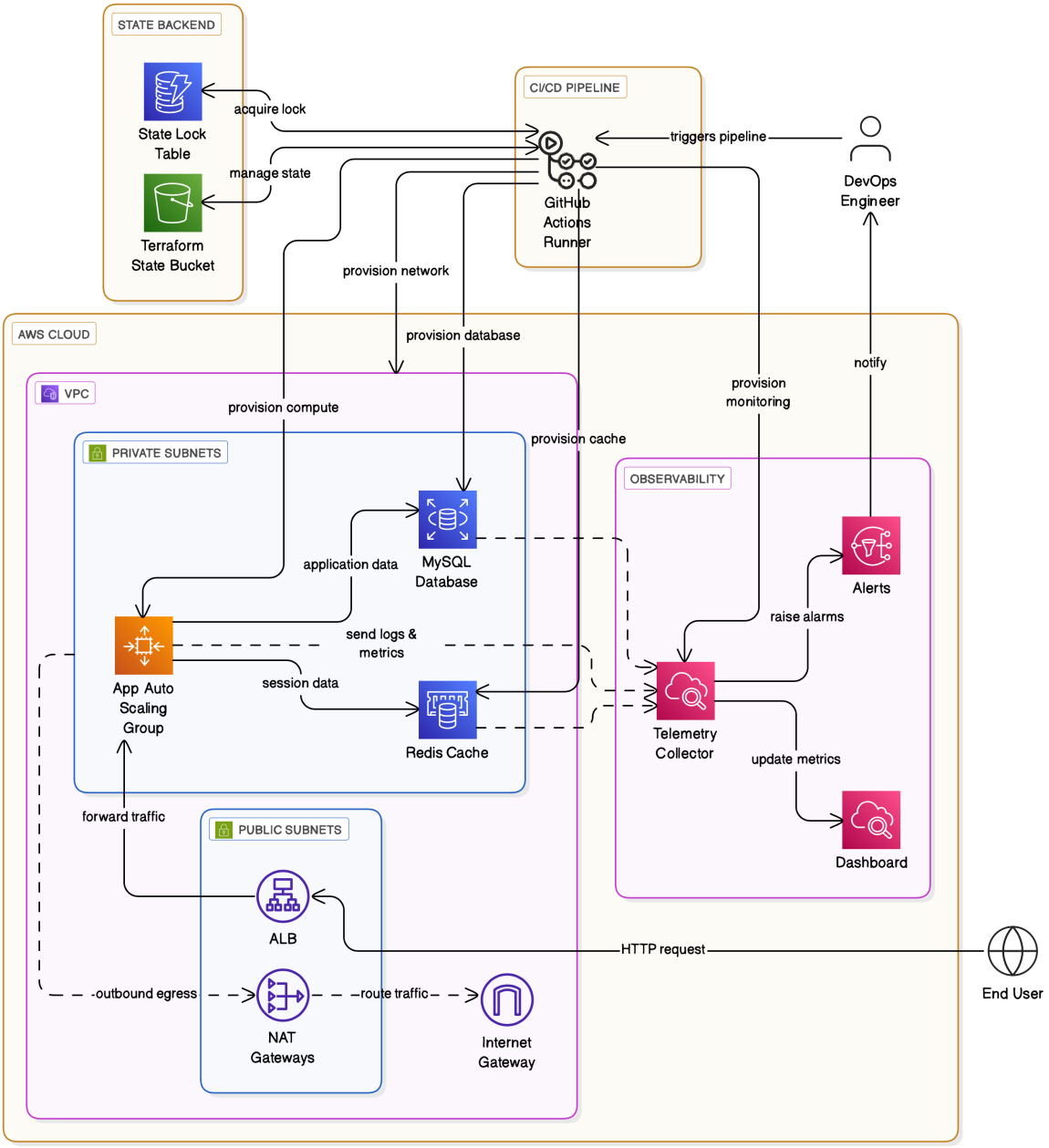

Figure 4: Full Blue-Green Deployment Architecture with CI/CD, Monitoring, and AWS Resources

4.2 Workspaces and Environment Isolation

Terraform workspaces are used to isolate deployments. For each environment (`staging`, `production`), a separate backend state, VPC, and infrastructure stack is created. This supports safe testing and deployment promotion using the same underlying code.

4.3 Backend Configuration

After the bootstrap process, the backend is auto-configured using a generated ‘`providers.tf`‘ file with the following structure:

Listing 1: Generated Terraform Backend

```
terraform {
  backend "s3" {
    bucket         = "backend-s3-bucket-1290"
    key            = "terraform/state.tfstate"
    region         = "us-east-1"
    dynamodb_table = "backend-d-db-table"
    encrypt        = true
  }
}
```

This setup enables team collaboration, remote state sharing, and full state locking.

4.4 Makefile and Automation

The infrastructure is managed using a '`Makefile`' with convenient targets:

Listing 2: Available Make Targets

```
make deploy-bootstrap    # Runs backend and OIDC setup
make destroy-bootstrap   # Cleans all backend infra and role bindings
```

4.5 CI/CD Integration

The '`.github/workflows/deploy.yml`' pipeline automates the apply/destroy process per workspace using GitHub Actions and OIDC authentication, ensuring secure, credential-free automation.

5. Deployment Process

The deployment lifecycle is split into two main phases: a one-time bootstrap setup for Terraform backend and access configuration, followed by repeatable environment deployments (staging or production) through a fully automated GitHub Actions workflow.

5.1 Bootstrap Phase (One-Time Setup)

Before deploying any environments, a backend for Terraform state management and GitHub OIDC access must be initialized. This is accomplished using the '`make deploy-bootstrap`' target, which performs the following actions:

1. Initializes and applies the '`bootstrap/`' Terraform project.
2. Provisions an encrypted and versioned S3 bucket to store state files.
3. Creates a DynamoDB table for locking to prevent concurrent state modifications.
4. Sets up an IAM OIDC provider and trust relationship for GitHub Actions.
5. Generates a '`providers.tf`' file dynamically, pre-filled with backend configuration values.

5.2 Environment Deployment (Staging and Production)

Deployments are environment-specific and driven either manually or automatically via GitHub Actions workflows. The logic is environment-isolated using Terraform workspaces.

A. Workflow Inputs

The CI/CD pipeline defined in '`.github/workflows/deploy.yml`' accepts two inputs:

- **environment:** One of `staging`, `production`, or `both`.
- **destroy:** Optional flag to destroy infrastructure instead of applying it.

B. Deployment Steps

Each environment deployment follows this sequence inside the CI/CD pipeline:

1. Checkout the repository source.
2. Configure AWS credentials using the GitHub OIDC role.
3. Run '`terraform init`' with remote backend reconfiguration.
4. Create or select the appropriate Terraform workspace.
5. Execute `terraform apply` or `terraform destroy` based on inputs.

C. Secrets Management

The pipeline uses GitHub Secrets for database credentials and securely assumes the IAM role via OIDC without needing long-lived access keys:

- `TRUST_ROLE_GITHUB`: IAM role ARN with full permissions to deploy resources.
- `STAG_DB_PASS`, `PROD_DB_PASS`: Injected as environment variables during deployment.

5.3 Destruction Workflow

If the `destroy` input is set to `true`, the pipeline performs an environment-specific destruction process. This includes removing all EC2, RDS, Redis, and networking resources while preserving the Terraform state unless explicitly deleted from the backend.

5.4 Reusability and Promotion

The deployment process supports:

- Re-deploying the same environment with updated configurations.
- Promoting tested changes from `staging` to `production` by rerunning the workflow.
- Isolated state and resources per environment for safe blue-green style rollouts.

6. Automation & DevOps Workflow

The project leverages a fully automated DevOps pipeline built with GitHub Actions and Terraform. This workflow handles all provisioning and destruction tasks securely, reproducibly, and without manual intervention. It integrates tightly with AWS Identity and Access Management (IAM) using OpenID Connect (OIDC), eliminating the need for AWS access keys.

6.1 CI/CD Pipeline Overview

The primary workflow is defined in the file `.github/workflows/deploy.yml`. It supports:

- Environment selection (`staging`, `production`, or `both`)
- Safe apply and destroy operations
- Environment-level parallelism via Terraform workspaces

A. Workflow Steps

a) **Checkout Code:** Clones the repository to the GitHub Actions runner.
b) **Install Terraform & Make:** Ensures required CLI tools are available.
c) **OIDC-Based AWS Authentication:** Uses 'aws-actions/configure-aws-credentials' to assume the pre-configured IAM role via GitHub OIDC.

d) **Terraform Init and Workspace Select:** Reconfigures backend, then selects or creates workspace.
e) **Terraform Apply or Destroy:** Executes the requested action using ‘terraform apply -auto-approve‘ or ‘terraform destroy‘.

6.2 GitHub OIDC Integration

To avoid static AWS credentials, GitHub Actions authenticates to AWS using a federated IAM role with OIDC trust. This is configured during the bootstrap phase and includes:

- An OIDC provider pointing to `token.actions.githubusercontent.com`
- A trust policy allowing GitHub Actions to assume the IAM role
- A permission policy granting access to required services (S3, DynamoDB, EC2, RDS, etc.)

6.3 Automation Highlights

- **Dynamic Role Assumption:** Securely authenticates GitHub workflows to AWS without secrets.
- **Modular Terraform Execution:** Separate modules for network, environment, and monitoring.
- **Dynamic Environment Support:** Supports on-demand deployment or teardown of `staging`, `production`, or both.
- **State-Aware Workflows:** Automatically handles backend locking and state isolation using Terraform’s remote backend configuration.

6.4 Makefile Integration

A local ‘`Makefile`‘ is also provided to support command-line management of the bootstrap infrastructure:

- `make deploy-bootstrap`: Initializes and deploys backend, OIDC provider, and IAM roles.
- `make destroy-bootstrap`: Destroys bootstrap resources and cleans up generated output files.

7. Resource Inventory

The following infrastructure resources are provisioned across two isolated environments: `staging` and `production`. Each environment maintains its own copy of all components, provisioned using Terraform workspaces and parameterized variables. This section summarizes the core resources managed by the system.

7.1 1. Backend Infrastructure (Bootstrap Phase)

- **S3 Bucket:** `backend-s3-bucket-1290`
  - Used for remote Terraform state storage
  - Server-side encryption: AES256
  - Versioning: Enabled
- **DynamoDB Table:** `backend-d-db-table`
  - Used for state locking and concurrency control
  - Billing mode: PAY_PER_REQUEST
- **IAM Role for GitHub Actions:** `TRUST_ROLE_GITHUB`

  - Integrated via OpenID Connect (OIDC)
  - Scoped permissions to provision and destroy all Terraform-managed services

7.2 2. Per-Environment Resources (Staging and Production)

Note: Each of the following resource types is provisioned *independently* in both environments, with workspace-specific names and isolated state.

- **VPC:** Custom VPC per environment
  - Public subnets: 2 (one per AZ)
  - Private subnets: 2 (one per AZ)
  - NAT Gateways: 2 (one per public subnet)
  - Internet Gateway: 1 per VPC
- **Compute (EC2):**
  - Auto Scaling Group with Launch Template
  - Instance Type: `t2.micro`
  - Architecture: `x86_64`
- **Load Balancer:**
  - Application Load Balancer (ALB)
  - Target group with health checks on `/`
  - HTTP listener on port 80
- **Database (Amazon RDS):**
  - Engine: MySQL
  - Class: `db.t3.micro`
  - Multi-AZ deployment: Enabled
  - IAM Authentication: Enabled
  - Backup Retention:
    * Staging: 0 days
    * Production: 7 days
- **Caching (ElastiCache - Redis):**
  - Node Type: `cache.t3.micro`
  - Number of cache clusters: 1
- **Monitoring (Amazon CloudWatch):**
  - Custom Dashboards per environment
  - Metric Alarms for:
    * EC2 CPU
    * RDS Connections
    * Redis Connections
    * Log-derived metrics (e.g., NGINX 5xx, application errors)
  - Log Groups:
    * `/aws/ec2/nginx`
    * `/aws/ec2/system`
    * `/aws/ec2/application`

    - `/aws/rds/mysql-logs`
    - `/aws/elasticache/redis-logs`
- **Alerts (Amazon SNS):**
  - Email notifications on alarm triggers
  - Configured per environment

8. Security and Compliance

Security and compliance are foundational components of this infrastructure. The design adheres to the principle of least privilege, secure remote execution, encrypted storage, and audit-friendly configurations. IAM roles, networking controls, and logging mechanisms are implemented to minimize risk and ensure traceability.

8.1 IAM and Role-Based Access

- **GitHub OIDC Integration:**
  - An IAM OIDC identity provider is configured for GitHub Actions using `token.actions.githubuser`
  - A trust policy allows workflows in this repository to assume the role `TRUST_ROLE_GITHUB` securely using `sts:AssumeRoleWithWebIdentity`.
  - No long-lived AWS credentials are stored or used in CI/CD.
- **EC2 IAM Role:**
  - An instance profile is attached to EC2 instances to grant scoped access to Amazon SSM, CloudWatch, and RDS IAM authentication.
  - IAM policies are dynamically rendered using Terraform `templatefile()` functions to avoid hardcoding sensitive identifiers.

9. Monitoring and Observability

This infrastructure incorporates a comprehensive observability strategy using Amazon CloudWatch, covering log aggregation, metric tracking, anomaly detection, and visual dashboards. The system is designed to provide real-time insights into application behavior, infrastructure health, and operational trends, with environment-specific granularity.

9.1 Log Collection and Aggregation

Multiple log sources are configured to route logs to centralized CloudWatch log groups, including:

- `/aws/ec2/nginx-{env}` – Access and error logs from the NGINX web server
- `/aws/ec2/system-{env}` – OS-level system logs (e.g., syslog, messages)
- `/aws/ec2/application-{env}` – Custom application logs
- `/aws/rds/mysql-logs` – RDS engine logs
- `/aws/elasticache/redis-logs` – Redis operational logs

These logs are collected via the Amazon CloudWatch Agent, configured through ‘user$_{d}ata.sh.tmpl$‘$andapplie$

9.2 Log-Based Metric Filters

To support proactive alerting, CloudWatch metric filters are applied to log streams using defined patterns. These filters generate custom metrics for critical issues, including:

- **ApplicationErrorCount:** Extracted from '/var/log/application.log' entries containing `ERROR`
- **Nginx5xxErrorCount:** Matches HTTP 5xx status codes in access logs
- **RDSErrorCount**, **RedisErrorCount**: Monitors database and cache errors from service logs

Each metric filter is configured in Terraform and attached to corresponding CloudWatch alarms.

9.3 Infrastructure Metrics and Alarms

Native AWS metrics are monitored and alarmed using CloudWatch alarms with thresholds defined in the Terraform variable file. Examples include:

- **EC2:**
  - CPUUtilization
  - FreeableMemory
- **RDS:**
  - DatabaseConnections
- **ElastiCache (Redis):**
  - CurrConnections

All alarms use a unified configuration for threshold, evaluation period, and notification logic.

9.4 Dashboards

Environment-specific CloudWatch dashboards are dynamically generated using Terraform, each combining the following:

- CPU, memory, and disk usage charts for EC2
- DB connection and error graphs for RDS
- Redis performance metrics
- Custom log-based error counts

These dashboards are labeled and organized per environment (`staging`, `production`).

9.5 Alerting and Notification

Alarms are linked to an Amazon SNS topic named `alerts`. If any metric exceeds its threshold, an email is sent to the configured recipient.

- SNS subscription is environment-aware and configurable via variables
- Email recipients are managed through `alarm.alert_email`

9.6 Retention and Compliance

- All log groups are configured with a `7-day retention` policy by default
- Retention is adjustable per environment via Terraform variables
- Metric alarms and dashboards are included in the Terraform state for traceability

10. Backup and Disaster Recovery

The infrastructure includes built-in backup and recovery mechanisms to ensure data durability, business continuity, and resilience against accidental deletions or failures. Backup strategies vary based on the environment and service criticality.

10.1 Amazon RDS (MySQL)

RDS backup and recovery features are configured per environment:

- **Staging Environment:**
  - Backup Retention: **0 days**
  - Snapshots are disabled to reduce cost
- **Production Environment:**
  - Backup Retention: **7 days**
  - Multi-AZ deployment ensures high availability
  - Backup window: `22:00{23:00 UTC`
  - IAM authentication enabled for secure access

All backups are managed natively by AWS and stored in a regionally resilient manner. In the event of a failure, production databases can be restored from the latest automated snapshot.

10.2 Terraform State

Terraform state management ensures full infrastructure recovery via:

- **S3 Backend:** Versioning is enabled on the state bucket to allow recovery from any previous state.
- **DynamoDB Locking:** Prevents simultaneous state writes, reducing the risk of state corruption.
- **Manual Recovery:** Previous state versions can be restored using S3 object versioning in case of unintended changes.

10.3 CloudWatch Logs

All log data from EC2, RDS, and ElastiCache is centralized in CloudWatch and retained for **7 days**. Logs can be exported to S3 manually for long-term archiving if needed.

- Supports diagnostics and post-incident analysis
- Log-based metrics can help detect system anomalies before full failures occur

10.4 High Availability Features

- **RDS Multi-AZ:** Automatically promotes standby instance during failures
- **NAT Gateways:** Distributed across multiple AZs for redundant outbound access
- **Auto Scaling Groups:** Replace unhealthy EC2 instances automatically
- **ALB Health Checks:** Remove failing instances from load balancer targets

10.5 Disaster Recovery Procedures

In the event of a full environment failure:

1. Terraform state can be restored from S3.
2. Terraform apply can re-provision all infrastructure using the last working configuration.
3. Production RDS can be restored from a snapshot to a new instance or cluster.
4. Alerting and monitoring assist in rapid incident identification and triage.

11. Maintenance and Handover

The infrastructure has been designed to support ease of operation, extensibility, and long-term maintainability. This section outlines the key processes, automation scripts, and documentation practices intended to support ongoing maintenance and knowledge transfer.

11.1 Operational Maintenance

- **Infrastructure Updates:**
  - All updates to compute, networking, or IAM policies can be applied via Terraform.
  - Apply changes safely per environment using:
    - GitHub Actions workflow (recommended)
    - Local CLI with ‘terraform workspace select‘ and ‘terraform apply‘
- **Resource Scaling:**
  - Update desired/min/max capacity in Auto Scaling configuration.
  - Modify RDS or Redis instance size using Terraform variables.
- **Logging and Monitoring:**
  - CloudWatch Dashboards should be reviewed regularly for anomalies.
  - Alarms should be tuned based on operational baselines over time.

11.2 Scripts and Tooling

- `generate_provider_file.sh`:
  - Automatically generates `providers.tf` after backend bootstrap
  - Populates region, bucket name, and DynamoDB table
- `user_data.sh.tmpl`:
  - Template for EC2 instance initialization
  - Installs NGINX, configures CloudWatch Agent, creates default log files
- `Makefile`:
  - `make deploy-bootstrap`: Initialize S3, DynamoDB, and OIDC
  - `make destroy-bootstrap`: Tear down backend and clean outputs

11.3 Knowledge Transfer and Handoff Notes

- This document serves as the primary technical reference for infrastructure design, automation, and operations.
- All Terraform modules are parameterized and reusable across environments.
- Terraform state files are stored centrally and version-controlled via S3 backend.
- GitHub repository contains:
  - CI/CD logic

  - Policy templates
  - Scripts and modules
- Teams should use GitHub Actions for deployments whenever possible to ensure role-based secure execution.

11.4 Future Recommendations

- Consider enabling S3 bucket access logging and encryption with customer-managed keys (CMKs).
- Automate tagging policies and cost attribution.
- Extend log retention and support log archiving to S3.
- Introduce tests or health probes in CI to validate instance behavior post-deploy.
- Monitor GitHub OIDC usage and rotate permissions regularly.

12. Cost Estimation and Resource Impact

This section outlines the estimated infrastructure costs and identifies the key AWS resources that most significantly impact billing. All cost assumptions are based on AWS public pricing in the `us-east-1` region as of the time of deployment.

12.1 Estimated Monthly Cost (Per Environment)

- **Amazon EC2 (t2.micro in ASG):** $\sim$ \$8–\$10 per instance (assuming low utilization and on-demand rates)
- **Amazon RDS (db.t3.micro, Multi-AZ):** $\sim$ \$30–\$50 (includes storage + standby instance)
- **Amazon ElastiCache (Redis, cache.t3.micro):** $\sim$ \$17–\$20
- **NAT Gateway (2 per environment):** $\sim$ \$60–\$80 (Charged per-hour + data processed)
- **S3 Bucket (Terraform backend):** $\sim$ \$0.10–\$0.25 (Minimal usage for state files and versioning)
- **DynamoDB (state locking):** $\sim$ \$1–\$2 (Pay-per-request mode)
- **CloudWatch Logs + Metrics + Alarms:** $\sim$ \$5–\$10 depending on log volume and alarm count
- **Total Estimated Monthly Cost: \$120–\$170 per environment**

12.2 Resources with High Cost Sensitivity

- **NAT Gateways:** Among the most expensive components due to hourly pricing and data transfer.
- **Multi-AZ RDS:** Increases cost by provisioning a standby instance and synchronizing data.
- **Auto Scaling Groups:** May incur higher cost as traffic grows or instance count increases.
- **CloudWatch Metrics & Logs:** Cost scales with log ingestion volume and number of metrics/alarms.
- **Redis (ElastiCache):** Costs can increase quickly with larger instance sizes or replication.
- **Snapshots and Backups:** Long retention periods for RDS or log data can accumulate storage charges.

12.3 Cost Optimization Recommendations

- Use **Graviton-based instances** (e.g., `t4g.micro`) where possible.
- Enable **data compression and retention policies** on logs.

- Schedule non-production environments to shut down outside business hours.
- Consider transitioning to **Savings Plans or Reserved Instances** for production workloads.
- Use budget alerts and tagging to track cost per environment or project.

13. Appendices

This section includes supporting materials, sample configurations, and generated output references that aid in reproducing or extending the infrastructure.

13.1 A. Terraform Backend Configuration (Generated)

The following backend block is generated automatically after the bootstrap phase and written into `providers.tf`:

Listing 3: Sample S3 + DynamoDB Terraform Backend

```
terraform {
  backend "s3" {
    bucket         = "backend-s3-bucket-1290"
    key            = "terraform/state.tfstate"
    region         = "us-east-1"
    dynamodb_table = "backend-d-db-table"
    encrypt        = true
  }
}
```

13.2 C. Makefile Targets

Listing 4: Available Make Targets

```
make deploy-bootstrap    # Initialize backend + GitHub IAM role
make destroy-bootstrap   # Tear down backend and role
```

13.3 D. Terraform Modules Overview

- `bootstrap/` – Initializes S3, DynamoDB, IAM trust policy, and OIDC.
- `modules/network` – VPC, subnets, routing, gateways.
- `modules/environment` – EC2, RDS, Redis, IAM roles.
- `modules/cloudwatch` – Dashboards, alarms, log groups.

13.4 E. Environment Naming Convention

- All resources are prefixed using the format: {`prefix`}-{`environment`}-{`type`}.
- Example: `fs-staging-alb`, `fs-prod-db`, `fs-production-ec2-sg`

13.5 F. GitHub Workflow Inputs

- `environment`: `staging`, `production`, or `both`
- `destroy`: `true` or `false`