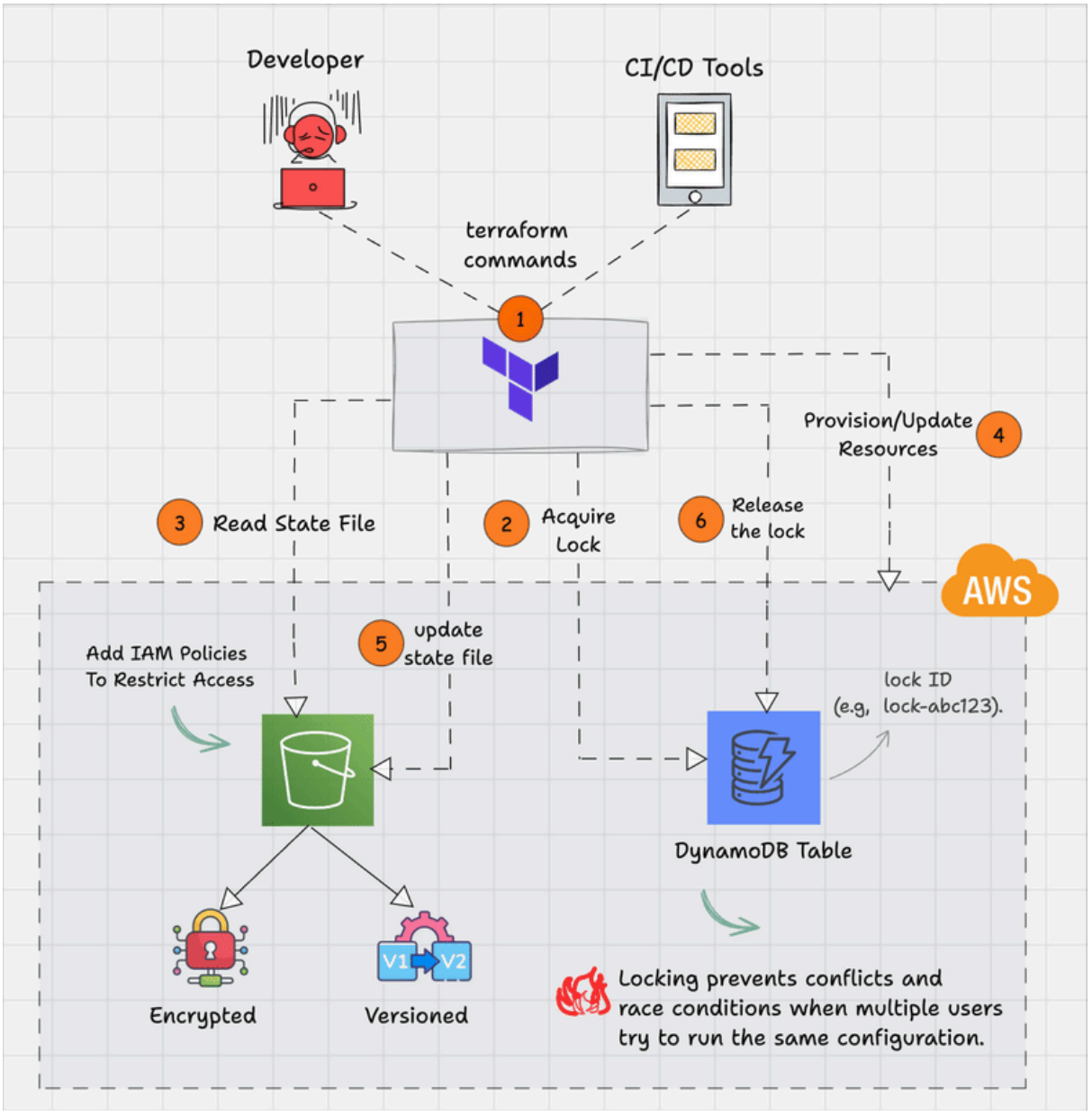
Project Overview

This project implements a production-style Terraform state management workflow in an AWS environment. It covers the configuration and operation of a remote Terraform backend and demonstrates how state is securely stored, inspected, modified, and recovered in scenarios commonly encountered in real infrastructure operations.

The implementation reflects how Terraform state is managed in collaborative production environments, with emphasis on state security, concurrency control, and safe recovery from lock-related failures.



Terraform State Operations Implemented

- Implemented a remote Terraform backend using Amazon S3 with encryption enabled
- Enforced state locking with Amazon DynamoDB to prevent concurrent modifications
- Migrated Terraform state from local storage to an S3-backed remote state
- Validated remote state persistence and lock behavior during Terraform operations
- Inspected Terraform-managed resources directly from state without relying on the AWS Console
- Exported Terraform state resource listings for auditing and operational visibility
- Removed resources from Terraform state without impacting underlying AWS infrastructure
- Imported an existing EC2 instance created outside Terraform into managed state
- Refactored Terraform state by renaming resources without downtime or recreation
- Identified and resolved Terraform state locks using both DynamoDB and Terraform mechanisms

---

Architecture Components

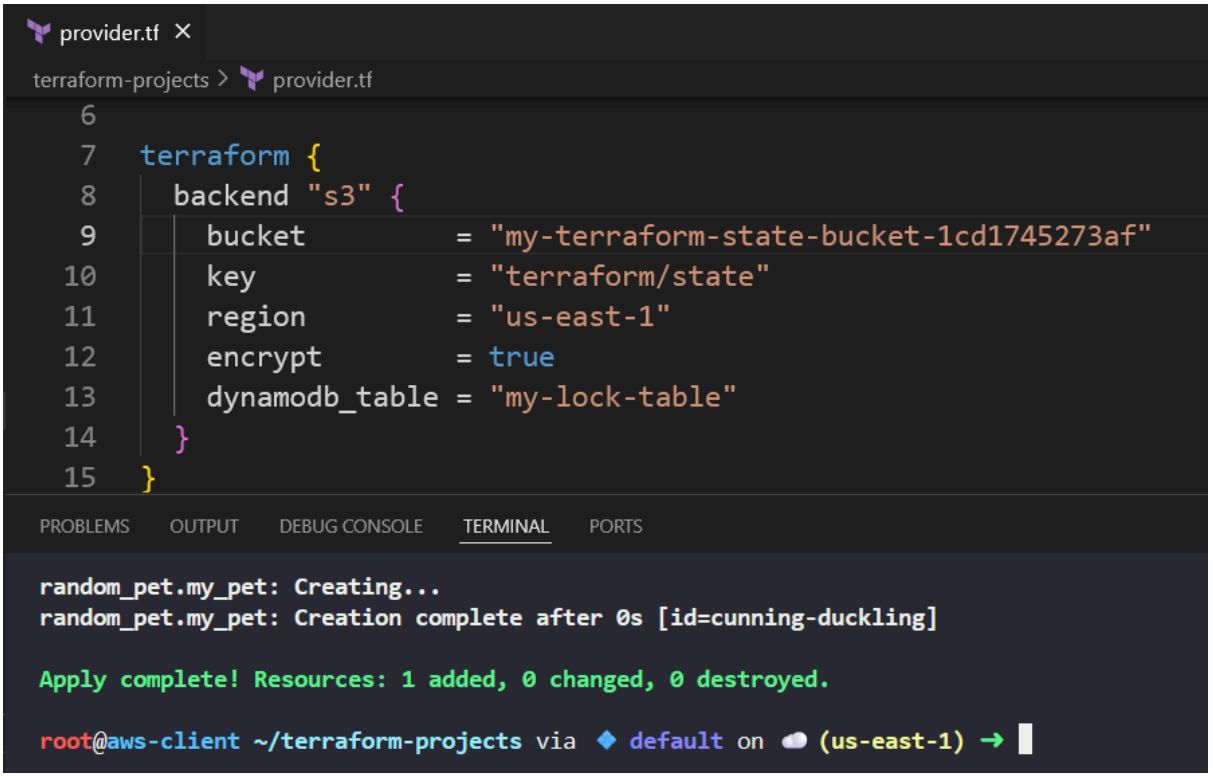
- **Terraform** – Infrastructure as Code engine
- **Amazon S3** – Remote state storage with encryption
- **Amazon DynamoDB** – State locking mechanism
- **Amazon EC2** – Compute resources managed via Terraform
- **AWS CLI** – Validation and inspection tool

Terraform State Operations

Operation 1: Configure Remote Backend

- Configured Terraform to use S3 for remote state storage
- Enabled encryption
- Implemented state locking with DynamoDB

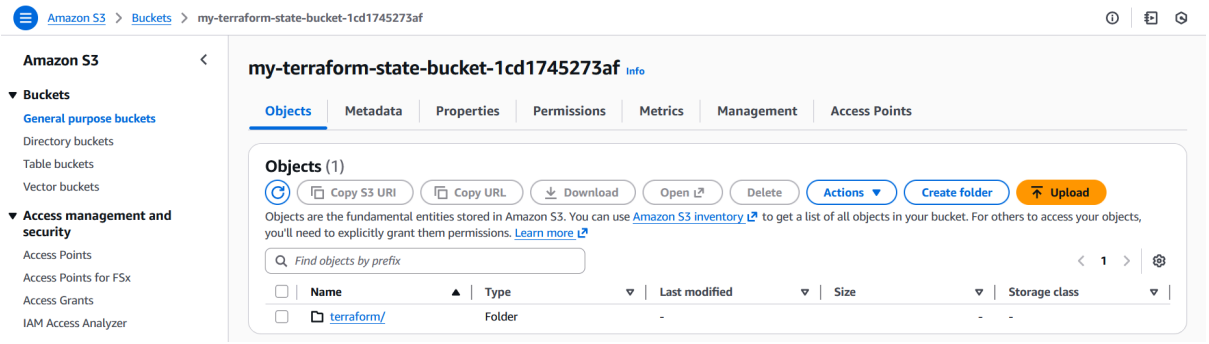
📸 *Screenshot: provider.tf configuration*



Operation 2: Migrate Local State to S3

- Initialized Terraform backend
- Migrated existing local state to S3
- Verified no local state file exists

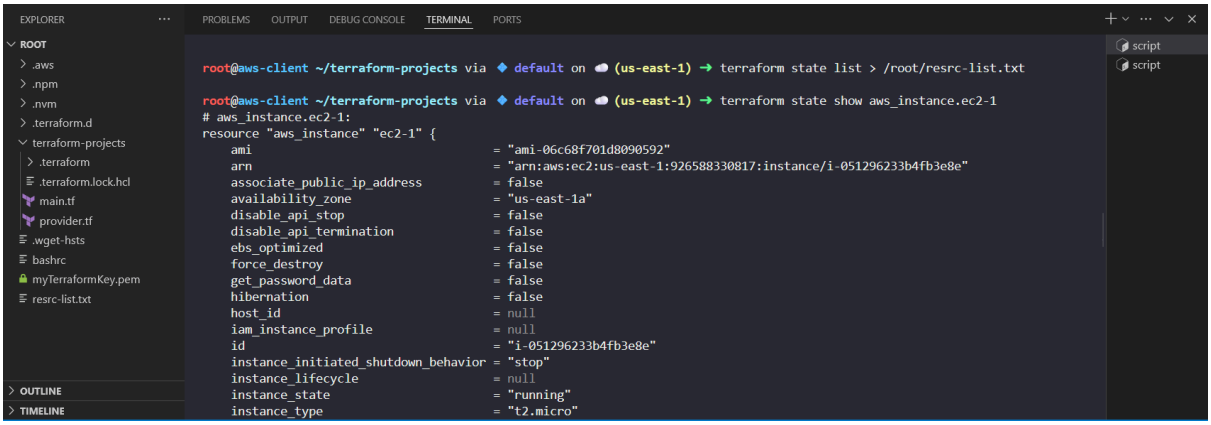
📸 *Screenshot: S3 bucket with terraform.tfstate*



Operation 3: Inspect Terraform State

- Listed all managed resources
- Exported resource list to a file
- Retrieved EC2 instance ID from Terraform state

📸 *Screenshot: terraform state list output*
📸 *Screenshot: terraform state show output*



Operation 4: Modify State Without Infrastructure Changes

- Removed a resource from Terraform state without deleting it
- Imported an EC2 instance created via AWS CLI
- Renamed Terraform-managed resources using state refactoring

📸 *Screenshot: terraform state rm*
📸 *Screenshot: terraform import*
📸 *Screenshot: terraform state mv*

```
root@aws-client ~/terraform-projects via 🔹 default on ☁️ (us-east-1) ➜ terraform state rm aws_instance.ec2-1
Removed aws_instance.ec2-1
Successfully removed 1 resource instance(s).
```

provider.tf | main.tf

terraform-projects > main.tf

More Actions...

```
38
39  resource "aws_instance" "ec2-2" {
40    ami           = "ami-06c68f701d8090592"
41    instance_type = "t2.micro"
42
43    subnet_id            = aws_subnet.main_subnet.id
44    vpc_security_group_ids = [aws_security_group.allow_ssh.id]
45    key_name             = "myTerraformKey"
46
47    tags = {
48      Name = "ec2-instance-2"
49    }
50  }
51
```

```
root@aws-client ~/terraform-projects via 🔹 default on ☁️ (us-east-1) ➜ terraform state rm aws_instance.ec2-1
Removed aws_instance.ec2-1
Successfully removed 1 resource instance(s).

root@aws-client ~/terraform-projects via 🔹 default on ☁️ (us-east-1) ➜ terraform init -upgrade
Initializing the backend...
Initializing provider plugins...
- Finding latest version of hashicorp/random...
- Finding latest version of hashicorp/aws...
- Using previously-installed hashicorp/random v3.7.2
- Using previously-installed hashicorp/aws v6.27.0

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

```
root@aws-client ~/terraform-projects via 🔹 default on ☁️ (us-east-1) ➜ terraform import aws_instance
.ec2-2 i-03af113a452d735de
aws_instance.ec2-2: Importing from ID "i-03af113a452d735de"...
aws_instance.ec2-2: Import prepared!
  Prepared aws_instance for import
aws_instance.ec2-2: Refreshing state... [id=i-03af113a452d735de]

Import successful!

The resources that were imported are shown above. These resources are now in
your Terraform state and will henceforth be managed by Terraform.
```

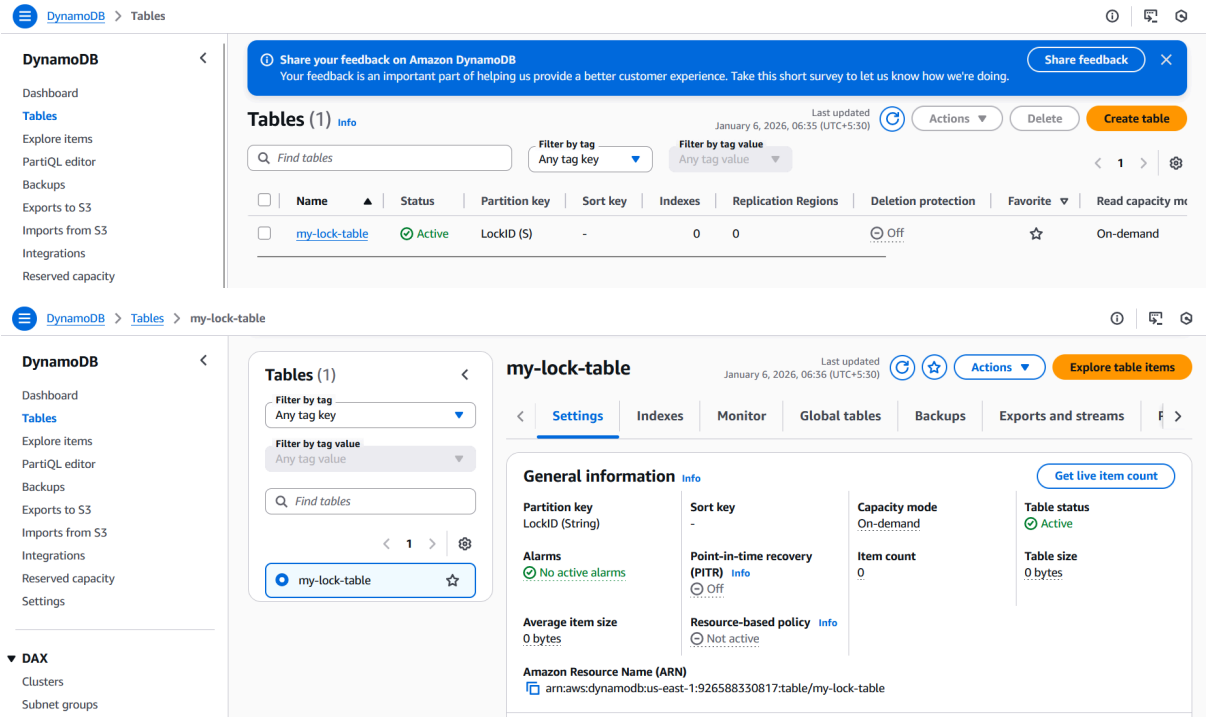
```
root@aws-client ~/terraform-projects via 🔹 default on ☁️ (us-east-1) ➜ terraform state mv aws_instance.ec2-2 aws_instance.
ec2-backup
Move "aws_instance.ec2-2" to "aws_instance.ec2-backup"
Successfully moved 1 object(s).

root@aws-client ~/terraform-projects via 🔹 default on ☁️ (us-east-1) ➜ terraform plan
terraform apply
random_id.bucket_suffix: Refreshing state... [id=EvZiSQ]
aws_vpc.main_vpc: Refreshing state... [id=vpc-026323a9441d79770]
aws_s3_bucket.sample_bucket: Refreshing state... [id=sample-bucket-12f66249]
aws_subnet.main_subnet: Refreshing state... [id=subnet-00a43c36c8113891b]
aws_security_group.allow_ssh: Refreshing state... [id=sg-0e686820848616140]
aws_instance.ec2-backup: Refreshing state... [id=i-03af113a452d735de]
```

Operation 5: Analyze and Recover from State Locks

- Inspected DynamoDB lock table
- Identified lock entries
- Demonstrated safe unlock methods using:
    - AWS CLI
    - Terraform force-unlock

📸 *Screenshot: DynamoDB lock table*
📸 *Screenshot: terraform force-unlock*



```
root@aws-client ~/terraform-projects via 🔹 default on ☁️ (us-east-1) ✕terraform force-unlock b5e19a601bc66a6965e845733efb3
7ae
Do you really want to force-unlock?
  Terraform will remove the lock on the remote state.
  This will allow local Terraform commands to modify this state, even though it
  may still be in use. Only 'yes' will be accepted to confirm.

  Enter a value: yes
```

Key Terraform Commands Used

- terraform init
- terraform apply
- terraform state list
- terraform state show
- terraform state rm
- terraform import
- terraform state mv
- terraform force-unlock

This implementation demonstrates how Terraform state can be safely operated in AWS using S3 and DynamoDB. The workflows covered: remote state configuration, state refactoring, resource import, and lock recovery, mirror real operational scenarios encountered in production environments.
The techniques shown are directly applicable to collaborative Terraform usage and CI/CD-driven infrastructure management.