PW SKILLS | DevOps and Cloud Computing

# Terraform for Cost Management, Security, and AWS Infrastructure Automation

# Objective

- Implement cost-saving strategies in cloud infrastructure using Terraform.
- Secure Terraform configurations, credentials, and state files using encryption and IAM policies.
- Automate AWS infrastructure including EC2, S3, VPCs, and RDS using Terraform.
- Configure auto-scaling, monitoring, and backup solutions i AWS with Terraform.

# Explaining key cost optimization strategies in cloud infrastructure:

# Let’s see

1. **Reserved & Spot Instances:**
- Reserved Instances offer significant discounts for long-term commitments.
- Spot Instances provide low-cost, short-term compute capacity for flexible or non-critical workloads.

2. **Auto-Scaling & Right-Sizing:**
- Auto-scaling adjusts resources automatically based on demand, preventing overuse.
- Right-sizing matches instance types/sizes to workload needs, avoiding over-provisioning.

3. **Data Optimization:**
- Use compression to reduce storage size.
- Move infrequently accessed data to archival storage like AWS Glacier or Azure Archive for lower costs.

# Pop Quiz

Q. Spot Instances are best suited for which of the following scenarios?

**A**

Workloads that can tolerate interruptions

**B**

Predictable and stable workloads

# Pop Quiz

Q. Spot Instances are best suited for which of the following scenarios?

**A**

Workloads that can tolerate interruptions

**B**

Predictable and stable workloads

# Discussing how Terraform helps enforce cost controls through policies and automated scaling.

# Let’s discuss

Terraform helps enforce cost controls by:

- **Policies with Sentinel:** Enforces rules (e.g., restrict instance types, limit regions) to prevent costly configurations.
- **Automated Scaling:** Manages infrastructure as code, enabling auto-scaling groups and resource adjustments based on demand.
- **Consistent Provisioning:** Avoids manual errors and ensures cost-efficient setups across environments.

# Explaining best practices for version control of Terraform configurations using Git.

# Let’s see

Best practices for version control of Terraform configs using Git:

- Use Git Repos
- Branching Strategy
- Use .gitignore
- Code Reviews
- Tag Releases
- Commit Often

PW SKILLS

# Discussing branching strategies for Terraform workflows (feature branches, main/stable environments).

# Let’s discuss

Branching strategies for Terraform workflows:

- **Feature Branches:** Used for developing and testing changes safely without affecting main infrastructure.
- **Main Branch:** Holds stable, reviewed, and approved code—typically reflects production-ready infrastructure.
- **Environment Branches:** Separate branches (e.g., dev, staging, prod) represent different deployment environments for isolated changes and testing.
- **Pull Requests:** Used to merge feature branches into main/stable branches after review and validation.

PW SKILLS

# Demonstrating how to structure Terraform code in a Git repository for modular reuse.

# Let's do it

To structure Terraform code in a Git repo for modular reuse:

- **modules/ Folder:** Store reusable modules (e.g., modules/vpc, modules/ec2).
- **environments/ Folder:** Separate configs for each environment (e.g., environments/dev, environments/prod) using modules.
- **main.tf, variables.tf, outputs.tf:** Use these in each environment for clarity and organization.
- **Version Control Modules:** Reference modules using Git tags or paths to ensure consistency.

# Explaining how modules enhance reusability and maintainability in Terraform.

# Let’s see

Modules in Terraform enhance:

- **Reusability:** Write once, use across multiple environments or projects (e.g., a VPC module).
- **Maintainability:** Centralize updates—changes in a module apply wherever it's used.
- **Organization:** Break infrastructure into logical, manageable components.
- **Consistency:** Enforce standards and reduce configuration drift.

# Discussing how workspaces separate environments (dev, staging, prod) for infrastructure.

# Let’s discuss

Terraform workspaces help separate environments like dev, staging, and prod by:

- Maintaining separate state files for each workspace, isolating resources.
- Using the same codebase to manage different environments.
- Enabling safe testing and deployment without affecting production.

# Demonstrating how to use Terraform modules for AWS networking, compute, and storage.

# Let’s do it

To use Terraform modules for AWS networking, compute, and storage:

- **Networking Module:**

```
module "vpc" {
  source = "./modules/vpc"
  cidr_block = "10.0.0.0/16"
}
```

# Let's do it

- **Compute Module:**

```
module "ec2" {
  source = "./modules/ec2"
  instance_type = "t2.micro"
  ami_id = "ami-123456"
}
```

- **Storage Module:**

```
module "s3" {
  source = "./modules/s3"
  bucket_name = "my-app-bucket"
}
```

# Explaining why storing credentials in Terraform code is a bad practice.

# Let’s see

Storing credentials in Terraform code is a bad practice because:

- Security Risk
- Version Control Issues
- Lack of Flexibility

# Demonstrating using environment variables and HashiCorp Vault for secrets management.

# Let’s do it

To use environment variables and HashiCorp Vault for secrets management in Terraform:

1. **Environment Variables:**
   - Set environment variables:

```
export AWS_ACCESS_KEY_ID="your-access-key"
export AWS_SECRET_ACCESS_KEY="your-secret-key"
```

   - Reference in Terraform:

```
provider "aws" {
  access_key = var.AWS_ACCESS_KEY_ID
  secret_key = var.AWS_SECRET_ACCESS_KEY
}
```

# Let’s do it

## 2. HashiCorp Vault:

- Authenticate with Vault:

```
vault login <token>
```

- Retrieve secrets in Terraform:

```
data "vault_secret" "my_secret" {
  path = "secret/myapp"
}
resource "aws_secretsmanager_secret" "example" {
  secret_string = data.vault_secret.my_secret.data["password"]
}
```

# Discussing role-based access control (RBAC) for Terraform workflows.

# Let’s discuss

Role-Based Access Control (RBAC) for Terraform workflows:

- Defines Permissions
- Least Privilege
- Separation of Duties
- Integration with Cloud Providers

# Pop Quiz

Q. What is the primary purpose of RBAC in Terraform workflows?

**A**

To manage networking and compute resources

**B**

To define granular permissions for users and teams

# Pop Quiz

Q. What is the primary purpose of RBAC in Terraform workflows?

**A**

To manage networking and compute resources

**B**

To define granular permissions for users and teams

PW SKILLS

# Take A 5-Minute **Break!**


- Stretch and relax
- Hydrate
- Clear your mind
- Be back in 5 minutes

# Explaining the risks of storing Terraform state locally and why encryption is necessary.

# Let’s see

**Risks of storing Terraform state locally:**

- **Data Loss:** Local files can be accidentally deleted or lost.
- **Collaboration Issues:** Teams can't safely share or manage local state.
- **Security Risk:** State files often contain sensitive data (e.g., passwords, resource metadata).

**Why encryption is necessary:**

- **Protects Sensitive Info:** Encrypts secrets stored in the state file.
- **Prevents Unauthorized Access:** Secures data at rest, especially in remote backends like S3.
- **Compliance:** Helps meet security and regulatory requirements.

# Demonstrating storing Terraform state in AWS S3 with encryption enabled.

# Let’s do it

To store Terraform state in AWS S3 with encryption enabled:

```
terraform {
  backend "s3" {
    bucket         = "my-terraform-state-bucket"
    key            = "envs/prod/terraform.tfstate"
    region         = "us-east-1"
    encrypt        = true
    dynamodb_table = "terraform-locks"  # for state locking
  }
}
```

- encrypt = true: Enables server-side encryption.
- Use dynamodb table for locking to avoid conflicts in team environments.

# Discussing using IAM roles/policies to restrict state file access to authorized users.

# Let’s discuss

Using IAM roles/policies restricts access to Terraform state files by:

- Controlling Access
- Enforcing Least Privilege
- Audit & Compliance

Example policy snippet:

```
{
  "Effect": "Allow",
  "Action": ["s3:GetObject", "s3:PutObject"],
  "Resource": "arn:aws:s3:::my-terraform-state-bucket/*"
}
```

# Explaining how Terraform interacts with AWS using the AWS provider.

# Let’s see

Terraform interacts with AWS using the AWS provider by:

- Authenticating via AWS credentials (env vars, IAM roles, or config files).
- Mapping Terraform code to AWS resources, like EC2, S3, VPC, etc.
- Using the AWS API to create, update, or delete resources based on the declared infrastructure.

Example:

```
provider "aws" {
  region = "us-east-1"
}
```

# Pop Quiz

Q. What is the purpose of the AWS provider in Terraform?

**A**

To manage Terraform state files

**B**

To allow Terraform to interact with AWS services

# Pop Quiz

Q. What is the purpose of the AWS provider in Terraform?

**A**

To manage Terraform state files

**B**

To allow Terraform to interact with AWS services

# Demonstrating writing Terraform configurations to provision (Provide pre-built configurations)

# Let’s do it

1. **EC2 with Security Group:**

```
resource "aws_security_group" "web_sg" {
  name        = "web-sg"
  description = "Allow HTTP and SSH"
  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_instance" "web" {
  ami             = "ami-123456"
  instance_type = "t2.micro"
  security_groups = [aws_security_group.web_sg.name]
}
```

# Let's do it

## 2. S3 with Versioning & Encryption:

```
resource "aws_s3_bucket" "my_bucket" {
  bucket = "my-secure-bucket"
}

resource "aws_s3_bucket_versioning" "versioning" {
  bucket = aws_s3_bucket.my_bucket.id
  versioning_configuration {
    status = "Enabled"
  }
}

resource "aws_s3_bucket_server_side_encryption_configuration" "encryption" {
  bucket = aws_s3_bucket.my_bucket.id
  rule {
    apply_server_side_encryption_by_default {
      sse_algorithm = "AES256"
    }
  }
}
```

# Let’s do it

3. **VPC with Subnets & Route Tables:**

```
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
}

resource "aws_subnet" "subnet1" {
  vpc_id     = aws_vpc.main.id
  cidr_block = "10.0.1.0/24"
}

resource "aws_route_table" "rt" {
  vpc_id = aws_vpc.main.id
}

resource "aws_route_table_association" "rta" {
  subnet_id      = aws_subnet.subnet1.id
  route_table_id = aws_route_table.rt.id
}
```

# Let’s do it

4. **RDS with Automated Backups:**

```
resource "aws_db_instance" "db" {
  identifier          = "mydb"
  engine              = "mysql"
  instance_class      = "db.t3.micro"
  allocated_storage   = 20
  username            = "admin"
  password            = "password123"
  skip_final_snapshot = true
  backup_retention_period = 7
  backup_window           = "03:00-06:00"
}
```

PW SKILLS

# Explaining Auto Scaling Groups (ASG) and Elastic Load Balancing (ELB).

# Let’s see

**Auto Scaling Groups (ASG):**

- Automatically adjust the number of EC2 instances based on demand to ensure performance and cost-efficiency.

**Elastic Load Balancing (ELB):**

- Distributes incoming traffic across multiple EC2 instances to improve availability, fault tolerance, and scalability.

# Demonstrating how Terraform can dynamically scale EC2 instances based on CPU usage.

# Let’s do it

To dynamically scale EC2 instances based on CPU usage with Terraform:

```
resource "aws_autoscaling_group" "web_asg" {
  launch_configuration = aws_launch_configuration.web_lc.name
  min_size             = 1
  max_size             = 5
  desired_capacity     = 2
  target_group_arns    = [aws_lb_target_group.web_tg.arn]
  vpc_zone_identifier  = ["subnet-abc123"]
}

resource "aws_autoscaling_policy" "cpu_scale_up" {
  name                   = "scale-up"
  scaling_adjustment     = 1
  adjustment_type        = "ChangeInCapacity"
  cooldown               = 300
  autoscaling_group_name = aws_autoscaling_group.web_asg.name
}

resource "aws_cloudwatch_metric_alarm" "high_cpu" {
  alarm_name          = "high-cpu"
  comparison_operator = "GreaterThanThreshold"
  evaluation_periods  = 2
  metric_name         = "CPUUtilization"
  namespace           = "AWS/EC2"
  period              = 120
  statistic           = "Average"
  threshold           = 70
  alarm_actions       = [aws_autoscaling_policy.cpu_scale_up.arn]
  dimensions = {
    AutoScalingGroupName = aws_autoscaling_group.web_asg.name
  }
}
```

PW SKILLS

# Discussing best practices for high availability and failover using ASG and ELB.

# Let’s discuss

Best practices for high availability and failover with ASG & ELB:

- Use Multi-AZ Deployment
- Health Checks
- Auto Scaling
- Elastic Load Balancer
- Monitoring & Alerts

# Pop Quiz

Q. How does an Elastic Load Balancer (ELB) contribute to high availability?

**A**

It distributes incoming traffic across multiple healthy targets

**B**

It replaces unhealthy instances automatically

# Pop Quiz

Q. How does an Elastic Load Balancer (ELB) contribute to high availability?

**A**

It distributes incoming traffic across multiple healthy targets

**B**

It replaces unhealthy instances automatically

# Explaining the importance of monitoring and logging in cloud environments.

# Let’s see

Monitoring and logging in cloud environments are crucial for:

- Visibility
- Troubleshooting
- Security
- Compliance
- Optimization

# Discussing how Terraform can automate the setup of CloudWatch dashboards, alarms, and log groups.

# Let’s discuss

Terraform can automate CloudWatch setup by:

- **Dashboards:** Create visual summaries of metrics.

```
resource "aws_cloudwatch_dashboard" "main" {
  dashboard_name = "app-dashboard"
  dashboard_body = jsonencode({ ... })
}
```

# Let’s discuss

- **Alarms:** Trigger actions based on thresholds (e.g., high CPU).

```
resource "aws_cloudwatch_metric_alarm" "cpu_alarm" { ... }
```

- **Log Groups:** Collect and manage logs for resources.

```
resource "aws_cloudwatch_log_group" "app_logs" {
  name              = "/aws/app/logs"
  retention_in_days = 7
}
```

# Demonstrating configuring CloudWatch alarms for instance health checks and resource utilization.

# Let’s do it

To configure CloudWatch alarms for instance health and resource usage:

```
resource "aws_cloudwatch_metric_alarm" "ec2_cpu_high" {
  alarm_name          = "HighCPUUsage"
  metric_name         = "CPUUtilization"
  namespace           = "AWS/EC2"
  statistic           = "Average"
  period              = 300
  evaluation_periods  = 2
  threshold           = 80
  comparison_operator = "GreaterThanThreshold"
  dimensions = {
    InstanceId = "i-0123456789abcdef0"
  }
  alarm_actions = ["arn:aws:sns:region:account-id:alarm-topic"]
}
```

# Explaining how EBS snapshots work for disaster recovery.

# Let’s see

EBS snapshots help in disaster recovery by:

- **Backing up volumes:** Capture point-in-time copies of EBS volumes.
- **Storing in S3:** Snapshots are stored durably and securely in Amazon S3.
- **Restoring quickly:** Use snapshots to create new volumes in any AZ.
- **Automation:** Schedule regular snapshots for consistent backups.

# Demonstrating using Terraform to schedule automated EBS snapshots.

# Let’s do it

To schedule automated EBS snapshots with Terraform, use Data Lifecycle Manager (DLM):

```
resource "aws_dlm_lifecycle_policy" "ebs_backup" {
  description = "Daily snapshot of EBS volumes"
  execution_role_arn = "arn:aws:iam::123456789012:role/AWSDataLifecycleManagerDefaultRole"

  policy_details {
    resource_types = ["VOLUME"]
    target_tags = {
      Backup = "true"
    }
    schedules {
      name = "daily-backup"
      create_rule {
        interval      = 24
        interval_unit = "HOURS"
      }
      retain_rule {
        count = 7
      }
    }
  }
}
```

# Discuss failover mechanisms using multi-region S3 replication and RDS read replicas.

# Let’s discuss

Failover mechanisms using:

**S3 Multi-Region Replication:**

- Automatically replicates objects across AWS regions for durability and quick recovery if a region fails.

**RDS Read Replicas:**

- Create read-only copies in different regions for load balancing and promote them to primary in case of failure, ensuring minimal downtime.

# Time for case study!

# Important

- Complete the post-class assessment
- Complete assignments (if any)
- Practice the concepts and techniques taught in this sessior
- Review your lecture notes
- Note down questions and queries regarding this session ar consult the teaching assistants

# Thanks
# !